TÉCNICO
LISBOA

SQUARES : A SQL Synthesizer Using Query Reverse Engineering

Pedro Miguel Orvalho Marques da Silva

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Professor Vasco Miguel Gomes Nunes Manquinho
Engineer Miguel Monteiro Ventura

November 2019

”*Old Man: What is your quest?*

King Arthur: To seek the Holy Grail!”

Monty Python and the Holy Grail, 1975

Acknowledgments

Before anything else, I would also like to acknowledge my dissertation supervisors Professor Vasco Manquinho, Miguel Ventura and Ruben Martins, my advisor at Carnegie Mellon, for their precious support, priceless insight, and their invaluable guidance throughout this work. I would also like to thank Miguel Terra-Neves and Rodrigo Bernardo for their advice and support which have been a great help.

I would like to offer my special thanks to Ruben Martins and Professor Justine Sherry for their support and friendship during my stay at Carnegie Mellon.

I would like to express my deep gratitude to Professor Vasco for accepting me as his student and for giving me access to computer hardware that was vital for this work. I would also like to thank Professor Vasco for allowing me to attend SAT 2019, which helped me to improve my understanding of the area of SAT and how the academic world works.

A very special gratitude goes out to all at OutSystems and OutSystems' AI Team for the financial support and for giving me access to their offices and hardware.

In addition, I would like to express my very great appreciation to Professor Vasco and OutSystems for the possibility of attending CP 2019 at Stamford. CP is the top conference on Constraint Programming and is where I published and presented part of this work.

I would like to thank my mother for her friendship, encouragement throughout my study, for always being there for me and without whom I would not be who I am today.

I would also like to thank my father and Inês, for all their support, valuable guidance and for always being there for me throughout all these years.

I would also like to extend my thanks to my aunt Cristina and my cousins Leonor and Diogo for their friendship and for letting me stay at their home these past few months. I also want to thank my grandfather Alfredo, for his unconditional love and support, and for always believing in me.

Last but equally important, I would also like to thank my brother, my sisters and all of my friends (and colleagues) that helped me throughout this journey. They helped me to grow as a person and managed to put up with me for all these years which is not an easy task. ;)

To each and every one of you, I would like to express my sincere gratitude for your support.

Pedro

This work was supported by OutSystems and by national funds through FCT with references UID-/CEC/50021/2019, PTDC/CCI-COM/31198/2017 and DSAIPA/AI/0044/2018, and by NSF award #1762363 and the CMU-Portugal program with reference CMU/AIR/0022/2017.

"*Those who can imagine anything, can create the impossible.*"
Alan Turing

Resumo

Actualmente, com a enorme quantidade de dados com que os analistas de dados precisam de lidar diariamente, estes frequentemente encontram tabelas com dados interessantes e não sabem como estas tabelas foram geradas a partir de uma base de dados. Deste modo, há uma necessidade crescente de sistemas capazes de resolver o problema de *Engenharia Reversa de Consultas* (ERC, *Query Reverse Engineering* da literatura inglesa). Dada uma base de dados $\mathcal{D}$ e uma tabela de saída $\mathcal{Q}(\mathcal{D})$, estes sistemas precisam encontrar uma consulta $\mathcal{Q}$, de modo que, ao executar $\mathcal{Q}$ em $\mathcal{D}$, o resultado é igual a $\mathcal{Q}(\mathcal{D})$. ERC pertence à área de Síntese de Programas.

O objetivo da área de Síntese de Programas é gerar automaticamente programas que atendem a uma determinada especificação de alto nível. Desde os anos 60, Síntese de Programas é um problema bem estudado e tem sido considerado o Santo Graal da Ciência da Computação. Até agora, os sintetizadores de programas usavam uma única árvore para representar programas. Neste trabalho propomos um novo sintetizador de SQL baseado em enumeração, SQUARES, que usa uma nova representação por linhas na qual cada linha do programa é representada pela sua própria subárvore.

Resultados experimentais na síntese de consultas SQL mostram que a codificação proposta baseada em linhas permite uma enumeração mais rápida de programas quando comparada à codificação usual baseada em árvore. Adicionalmente, embora a codificação baseada em árvore não ultrapasse um pequeno número de operações, a nova codificação baseada em linhas permite encontrar programas com uma sequência maior de operações. Resultados experimentais na síntese de consultas SQL da OutSystems mostram que o SQUARES supera o Scythe, um sintetizador de SQL de última geração.

Palavras-chave: Síntese de Programas, Engenharia Reversa de Consultas (ERC), Teorias de módulos de satisfação (TMS), Síntese de programas com base em enumeração, SQL

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

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

The goal of Program Synthesis is to automatically generate programs that satisfy a given high-level specification [42]. A theme of research since the 60's, Program Synthesis is not only a well-studied problem, but it has even been considered the Holy Grail of Computer Science [29, 37]. Once considered a utopian dream, the recent advances in Program Synthesis are making this approach more practical and have shown that it can be useful to both end-users and programmers.

According to Manna and Waldinger [41], winners of the 2016 Herbrand Award: "*It is often easier to describe what a computation does than it is to define it explicitly. That is, we may be able to write down the relation between the input and the output variables easily, even when it is difficult to construct a program to satisfy that relation*". Nowadays, this use of input-output examples as specifications is a common solution [29]. 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 [25, 64], list transformations [4, 22], table reshaping [19], code completion [57], helping programmers to use libraries [20], and SQL queries [75, 79, 81].

Even though there are many approaches to Program Synthesis [29], one of the most commonly used solutions is to perform an enumerative search over the space of programs that satisfy the specification [19, 21, 44]. Fig. 1.1 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.

Recent approaches combine enumerative search with deduction. The goal is to perform early pruning of infeasible programs [19], or to learn from past failed candidate programs in order to prune all equivalent infeasible programs [21].

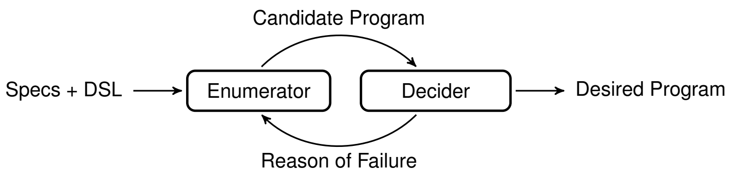


Figure 1.1: Enumeration-based Program Synthesis

Program synthesis is not merely an academic research topic since it is also transitioning into industry. Microsoft's FlashFill [25] is the most successful application of Program Synthesis by Microsoft for string manipulation and it is integrated into Microsoft Excel. Other companies are also starting to look for applications of Program Synthesis to their products (e.g. OutSystems [50] and Query Synthesis).

Since the beginning of the 21st century, with the Big Data revolution, several companies started having trouble with the management of their databases, concerning the massive amount of data that suddenly appeared. According to Zhang and Sun [81], there is a growing population of non-expert database end-users that have limited programming knowledge and do not always know how to query a relational database. 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 [39, 70, 71, 81]. This subfield of Program Synthesis became known as Query Synthesis, where the goal is to find the query desired by the user [70].

Over the past few years, the two most studied approaches to allow the user to give specifications about the desired query in Query Synthesis are input-output examples [71, 80] and natural language descriptions [38, 79]. In both approaches, the user provides an input database. In the first approach the user also provides the desired query's output table, while in the second the user gives a description in natural language of what he wishes to query.

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 $\mathcal{Q}(\mathcal{D})$ [70]. We focus on a new system, SQUARES, whose objective is to solve QRE for a subset of SQL.

In addition to the previous motivation for Query Synthesis, there are more examples where QRE may be helpful. Suppose that a data analyst finds a table with promising data, but she does not know which SQL query generated that specific table. She only knows the database where the data came from and when it was queried. Then, having a copy of the database from that time, the data analyst could use a QRE system to discover what was the query used to obtain such interesting data. Nowadays, not knowing which SQL query generated some table can easily happen, due to changes in the software used to manage the databases or just by changing the data analyst. Furthermore, consider that a data analyst wrote a query in SQL, but she is unaware whether there exists a similar query that produces the same result, but with lower complexity regarding the number of table joins and conditions. If a QRE system searches for a query in a growing number of table joins and conditions, it can return the query

Table 1.1: Table parts, supplier and an output table. The output table is the result of selecting the pnames after joining parts and supplier.

parts	
id	pname
1	Tire
2	Suspension

supplier	
id	sname
1	Michelin

output
pname
Tire

that is consistent with the output table and has the lowest complexity possible [80]. Currently, optimizing queries is an important area of research [61].

As mentioned earlier, QRE can be used in other query languages. For instance, SPARQL is a query language that has received a lot of attention from the QRE community in the past few years [1, 2, 17].

1.1 Motivating Example

Suppose that a user wants to synthesize an SQL query using examples. In particular, given tables *parts* and *supplier*, in Table 1.1, with the schema "parts(id: integer, pname: string)" and "supplier(id: integer, sname: string)", the user wants to find the *names* of parts, *pnames*, for which there is some supplier [1]. This can be represented by the output table presented in Table 1.1. Therefore if the user provides these three tables, *supplier* and *parts* as input tables and the output table, to a program synthesizer, it would return 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 1.2 shows the typical tree representation used by program synthesizers [10, 19, 21], 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 [19, 21]. 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 1.2 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 ε 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 1.2 would need 40 nodes to represent the search space for 3

[1]This corresponds to exercise 5.2.1 from a classic textbook on databases [56].

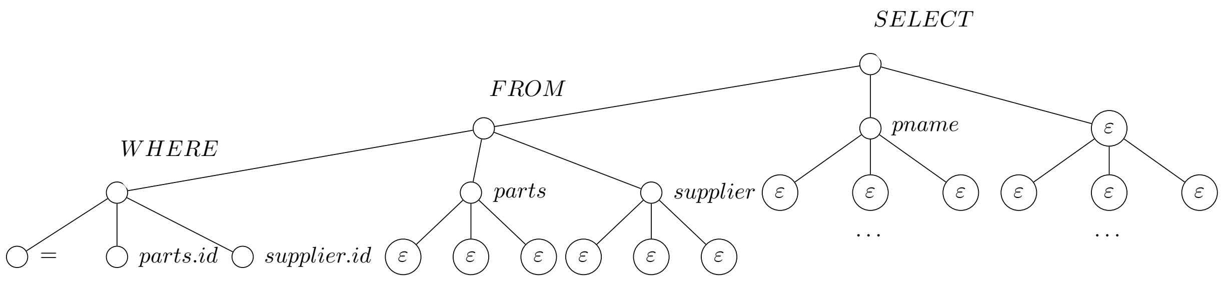


Figure 1.2: Tree-based representation of the search space

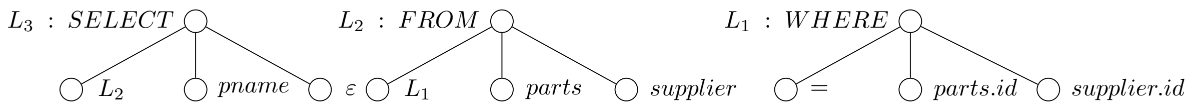


Figure 1.3: Line-based representation of the search space

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 thesis, we propose a new line representation illustrated in Figure 1.3, 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 [44]. Trinity uses the traditional tree-based representation of a program, while SQUARES incorporates our new line representation illustrated in Figure 1.3.

SQUARES' goal is to synthesize SQL queries from input-output examples (tables), i.e., solve the problem of Query Reverse Engineering. We gathered SQL instances previously used by well-known SQL synthesizers [19, 75, 81] and some query examples used in OutSystems' Engineering Database [50]. We evaluate our system with these instances and compare its performance against Scythe [75], a state-of-the-art synthesizer presented at PLDI'17, whose goal is also to solve the problem of Query Reverse Engineering.

Part of our work was published at the 25th International Conference on Principles and Practice of Constraint Programming (CP'19) [49]. CP is classified as an *A* conference in the CORE ranking [2].

To summarize, this thesis makes the following contributions:

- We formalize how to encode the traditional tree-based representation of a program into SMT which has an exponential growth with respect to the number of lines of a program [49].
- We propose a new compact SMT encoding based on a line representation of programs that grows linearly with the number of lines of a program [49].
- We propose a new enumeration-based program synthesizer whose goal is to solve the problem of Query Reverse Engineering.
- We integrate the line-based encoding into SQUARES, and empirically evaluate our approach using several SQL instances. Experimental results show that the line-based encoding significantly outperforms the tree-based encoding and allows program synthesizers to more effectively enumerate the search space and to synthesize larger programs [49].
- We compare SQUARES against Scythe [75]. We compare both systems in terms of performance and SQL generation quality, using instances from a classic database textbook [56] and examples from the real world provided by OutSystems [50].

1.3 Organization

This document is organized as follows. Chapter 2 presents the basic definitions and notation used in the following chapters.

Afterwards, Chapter 3 provides some background on Program Synthesis: the main challenges (section 3.1), the subfield of Programming by Example (section 3.2), as well as, the tree-based encoding used in Enumeration-Based Program Synthesis (section 3.3). Finally, Section 3.4 presents the subfield of Query Synthesis and covers the most relevant related work regarding the problem of Query Reverse Engineering: TALOS (section 3.4.1), SQLSynthesizer (section 3.4.2), QFE (section 3.4.3) and Scythe (section 3.4.4).

Chapter 4 provides the description of SQUARES, our enumeration-based QRE system. Sections 4.1 and 4.2 explain the overall architecture of our framework. In Section 4.3, our new encoding for Enumeration-Based Program Synthesis is presented.

Experimental results are presented and discussed in Chapter 5. Section 5.1, presents the evaluation between the classic encoding for Enumeration-Based Program Synthesis using the tree-based representation and our line-based representation. Next, Section 5.2 evaluates our heuristics to cut the search space in order to speed up the search for programs. Lastly, Section 5.3, compares SQUARES against the state-of-the-art QRE system Scythe [75], using instances from a classic database textbook [56] and examples from the real world used in OutSystems [50].

[2]http://portal.core.edu.au/conf-ranks/1175/

Finally, Chapter 6 presents the main conclusions about our work and discusses directions for possible future work.

Chapter 2

Preliminaries

This chapter provides some definitions and notation that will be used throughout the document. The following definitions are inspired by Feng et al. [19, 21] and Tran et al. [70].

Definition 1 (**Propositional Literal**). *A propositional literal is a Boolean variable x (positive literal) or its negation $\neg x$ (negative literal).*

Definition 2 (**Clause**). *The disjunction of literals (e.g. $x \vee y$) is called a clause.*

Definition 3 (**Formula**). *A propositional formula ϕ in the conjunctive normal form (CNF) is a conjunction of disjunctions of literals, i.e., a conjunction of clauses (e.g. $\phi = (x \vee y) \wedge (x \vee z)$).*

Definition 4 (**Interpretation**). *Let c be a clause, L be the set of c's literals and m be an interpretation, such that, $\forall l \in L, m : l \rightarrow \{0, 1\}$. The clause c is satisfied by m if and only if at least one literal in L is satisfied by m. A formula ϕ is satisfied by an interpretation m' if and only if all of ϕ's clauses are satisfied by m'.*

Definition 5 (**Boolean Satisfiability Problem (SAT)**). *SAT is the decision problem for propositional logic, i.e. to decide if a given propositional formula ϕ has a satisfying interpretation or prove that such an interpretation does not exist [6].*

SAT was the first problem proven to be NP-Complete in 1971 by Cook [14]. There are innumerous problems that can be modeled as a propositional formula. The satisfiability of such formulas is checked by logical engines called *solvers*.

Definition 6 (**Solver**). *A Solver is a logic engine capable of deciding if a given propositional formula, ϕ, is satisfiable. In this case, the solver produces an interpretation (attribution) of ϕ such that ϕ evaluates to true. Otherwise, the solver returns that ϕ is unsatisfiable [43].*

Example 1. *Let $\phi_1 = (x_1 \vee \neg x_2) \wedge (x_1 \vee x_2)$ be a propositional logic formula and $\{x_1, x_2\}$ be the set of ϕ_1's variables. A SAT solver would produce one of two possible interpretations of ϕ_1: $\{(x_1, 1), (x_2, 0)\}$ or $\{(x_1, 1), (x_2, 1)\}$.*

Consider another propositional logic formula $\phi_2 = (x_1 \vee \neg x_2) \wedge \neg x_1 \wedge x_2$. There is no possible interpretation that makes ϕ_2 satisfiable. Hence, a SAT solver would return "unsatisfiable" for this formula.

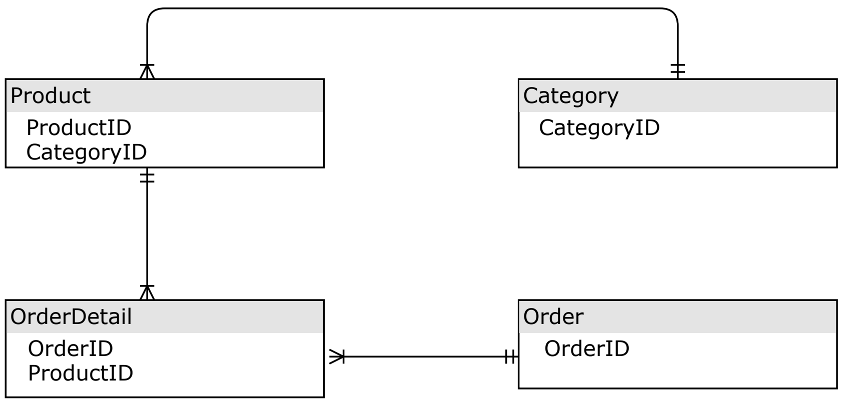


Figure 2.1: Schema graph of a small database with four tables: *Order, OrderDetail, Product, Category*.

Definition 7 (**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 ϕ, the SMT problem consists of deciding if there exists a complete assignment over the variables of ϕ such that ϕ is satisfied. Depending on the theory $\mathcal{T}$, the variables can be of type integer, real, Boolean, among others [49].*

Example 2. *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, ϕ_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, ϕ_2 is unsatisfiable since there is no assignment to the problem variables such that ϕ_2 is evaluated to true.

Definition 8 (**Table**). *A Table Γ is a 3-tuple (C, R, τ). The number of columns is denoted by C, and the numbers of rows by R. Γ_{c_1,r_1} is the element in the column c_1 and row r_1. The list τ represents the types of every column in Γ. Let τ_i represent the type of column i, τ is the collection of every τ_i. A column of a table Γ is normally called an attribute of Γ [19].*

Definition 9 (**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 Γ in $\mathcal{D}$. Each edge between two distinct tables Γ_1 and Γ_2 is in E_S if a join is possible between these two tables, i.e. if both tables share an attribute [36].*

Example 3. *Consider the schema graph $\mathcal{G}_S$ of a small database presented in Fig. 2.1. This database has four tables (Order, OrderDetail, Product, Category). The collection of $\mathcal{G}_S$'s nodes is equal to the collection of tables, so $N_S = \{Order, OrderDetail, Product, Category\}$. Regarding the edges present in $\mathcal{G}_S$, $E_S = \{(OrderDetail, Order), (OrderDetail, Product), (Product, Category)\}$. An edge exists if two tables share a common attribute (e.g. OrderDetail and Order share the OrderId attribute).*

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

Figure 2.2: 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)".

Definition 10 (**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, Σ 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 [31].*

Definition 11 (**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 [21].*

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 σ.

Example 4. *Consider the DSL D in Fig. 2.2, and suppose that a user wants to solve the query presented in Chapter 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 $p_1 =$ select_from, $p_2 =$ column and $p_3 =$ join.*

Definition 12 (**Program Space**). *Program space is the space with all possibilities for program candidates syntactically correct in a certain programming language. The program space typically grows exponentially with the size of the desired program [29].*

A program synthesizer is, according to Manna and Waldinger [41], "*a system that takes such a relational description (e.g. input-output examples) and tries to produce a program that is guaranteed to satisfy the relationship*". Program synthesizers search the space of programs described by a given domain-specific language (DSL).

Definition 13 (**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 [19].*

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

Definition 14 (**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 consistent with Ops and (3) $\mathcal{P}(\mathcal{E}_{in}) = \mathcal{E}_{out}$ [19].*

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

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

Definition 15 (**Query Reverse Engineering (QRE)**). *Let $\mathcal{D}$ be a database with schema graph $\mathcal{G}_S$ 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 $(\mathcal{G}_S, \mathcal{Q}(\mathcal{D}))$ [70].*

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

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

Definition 16 (**PSPACE**). *The class of problems that are decidable in polynomial space on a deterministic Turing machine [3].*

Sarma et al. [59] proved that QRE is a PSPACE-hard problem, i.e., QRE is at least as hard as any problem in PSPACE. Hence, if we can solve QRE, then we can solve any other problem in PSPACE. Therefore, a brute-force approach such as enumerating all possible queries and testing all of them is intractable [81].

Chapter 3

Background

This chapter provides some background on Program Synthesis and Programming By Example. Afterwards, the problem of Query Reverse Engineering is presented.

3.1 Program Synthesis

To the best of our knowledge, the first references to Program Synthesis date back to the 60's [13, 74]. Since then a large body of research has been conducted regarding the problem of synthesizing programs automatically, in different communities such as artificial intelligence [24, 40, 58], automata theory [5], machine learning [46, 63, 73] and programming languages [19, 55, 79]. The problem of Program Synthesis has three fundamental characteristics [29]: program space, user intent and search technique.

Challenges For Turing-complete programming languages, the problem of Program Synthesis is undecidable [29]. There are two main challenges in Program Synthesis: the extensive program space and the variety of user intent.

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 on searches in an exponentially growing tree. Modern approaches use heuristics for cutting the tree, thereby reducing the search space [35]. As a result of the technological revolution of the 21st century, many constraint-based synthesis applications were created [37, 65], as well as several stochastic approaches [47, 52, 60] and deductive top-down search [25, 55]. The program space is also called search space.

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. Several approaches allow a user to provide specifications in the form of: natural-language descriptions [16, 28, 79], a few input-output examples [21, 25, 46], partial programs [37, 65] or related programs [61, 70]. Other types of specifications, like formal specifications, are usually too complex for real-life problems. Such descriptions can be almost as large as the desired program. Hence, approaches based on formal specifications are not usually adopted because they would require too much effort from the user.

Users who are not programmers may be more comfortable giving input-output examples instead of partial/related programs. Examples can be used to handle ambiguity through interaction with the user. One approach is to ask the user to select from a collection of new input-output examples, generated by the system, which of them are valid for the desired program [26, 39, 45, 75]. Section 3.2.2 presents two methods for dealing with ambiguity.

3.1.3 Search Techniques

There exist four main search techniques for Program Synthesis: constraint solving, deductive, enumerative and statistical [29]. However, it is possible to use a combination of these techniques [21].

Constraint Solving. This approach can be divided into two parts: generation and resolution of constraints. The generation of constraints is the part where a logical formula is built such that its solution corresponds to a program that satisfies the program specification.

There are three methods for generating the logical formula: input-based [37], invariant-based [66] and path-based [67]. The constraints' resolution is the part where the formula previously generated is solved using a constraint solver (e.g. SAT or SMT solver) [19, 21, 44].

Deductive. This technique follows a top-down search based on the divide-and-conquer method. The *divide-and-conquer* method consists in a search algorithm that recursively breaks down the problem into subproblems that are easier to solve, then the algorithm just combines all the results of the subproblems in order to build a solution for the original problem.

Given a grammar, deductive search finds a sequence of DSL production rules that represent a program. This sequence starts in a production rule whose type is equal to the DSL's output type and finishes in a production rule that uses at least one symbol of the input type. Therefore, the divide-and-conquer method makes a backward traversal from the outputs to the input data [55].

Enumerative. The enumerative search consists in the enumeration of programs that are in the search space [19, 21]. First, this search sorts the program candidates according to some ranking heuristic. Second, following that ranking, the system tries every candidate searching for one that satisfies the specification provided by the user.

Given a grammar, this search finds a sequence like the deductive approach. The difference is that this sequence is built bottom-up, connecting inputs to outputs, the opposite of the deductive approach.

Enumerative search should be used wisely, otherwise, it does not scale. Hence, some pruning or a good ranking system is important in order to generate first the more likely programs.

Statistical. There are four main statistical techniques used in Program Synthesis: genetic programming, machine learning, Markov Chain Monte Carlo (MCMC) sampling and probabilistic inference [29].

Genetic programming is a technique where programs are seen as genes which are evolved using evolutionary algorithms. Through random changes (mutations) and sharing pieces of code (crossover), new programs are created and evaluated using a fitness function. This technique has been used to fix bugs in programs [77].

Machine learning is normally used to guide other search approaches, by providing likelihoods of several options at a level where a choice is needed. The likelihood can be a function of the input-output examples provided by the user, and these functions are learned from training data [4, 46, 73].

MCMC sampling is used in some systems, with STOKE [60] representing a well-known example. This system uses a Markov Chain Monte Carlo sampler to search the space of all possible programs. Starting from a given candidate and by making local changes, this technique obtains a program that is a local optimization of the initial candidate [48].

Probabilistic inference has been used to evolve a given program by making local changes [29]. This technique models a program as a graph of instructions and states, connected by constraint nodes. Then through belief propagation the intermediate program states are inferred as well as the instructions used in the program [27, 34].

3.2 Programming by Example (PBE)

One subfield of Program Synthesis is *Programming by Example*, in which the system learns the user's intent through the input-output examples provided by the user [29]. The input-output examples specify the expected behaviour for the desired program, i.e., given an input, what is the expected output, as we can see in Example 5:

Example 5. *Let an example be defined as a pair (input, output). The following set of examples can be used to describe the function* $x + 2 : \mathbb{N} \mapsto \mathbb{N}$*:*

$$\{(1, 3), (2, 4), (4, 6), (6, 8)\}$$

3.2.1 Properties

Programming by Example is a well-studied subfield of Program Synthesis because of the distinctive properties of the input-output examples. These two distinctive properties are ambiguity and ease of use [29].

Example 6. *The following input-output example, is a subset of examples that can help describe the function* $x + 2 : \mathbb{N} \mapsto \mathbb{N}$ *as well as the function* $x^2 : \mathbb{N} \mapsto \mathbb{N}$*:*

$$\{(2, 4)\}$$

Ambiguity. In most cases, for a given set of input-output examples, there is more than one possible candidate that is consistent with the examples. This can be seen in Example 6. For this reason, input-output examples are an under-specification of the desired program. Therefore, the system does not simply have to find a program that is consistent with the set of examples, it has to choose one to give to the user from a collection of possible candidates.

Ease of use. As explained in section 3.1.2, examples are a good way to express the user's intent, because they are simple to specify, explain and check. For users who are not programmers or use applications where programming is not an option, examples are usually the easiest method of specification.

3.2.2 Ambiguity Resolution

As exposed in Example 6, the problem with using input-output examples as specifications is that multiple programs with different semantics may satisfy the examples provided by the user. This can be solved just by asking for more examples. At a certain point, with sufficient examples, only the program desired by the user, or similar programs, are consistent with the examples. Obviously, it is not practical to require from the user a vast number of examples. For this reason, there exist two procedures to reduce ambiguity in PBE [29]: active learning and ranking.

Active Learning. A typical approach for disambiguation when the system has more than one program consistent with the input-output examples is to apply an interactive loop with the user [45]. The loop's objective is to ask for more examples or to select between several possibilities. This approach is called *active learning*, and there are various methods to interact with the user.

The most common way to disambiguate between programs consistent with the examples is to generate an input that produces different outputs on those programs. This approach is called *distinguishing inputs* [33]. Once those inputs are generated there are two ways to get the correct outputs. One can ask the user to introduce the expected output given the distinguishing inputs [33]. Another possibility is to generate the correspondent output for all the program candidates. Next, the input and all the outputs are presented to the user that then chooses the correct output. This approach was implemented in Scythe [75].

Generating examples that achieve different outputs can be a difficult task so several systems choose other forms of learning. By listing all possible programs consistent with the input-output examples, the user can select the one he wants [28]. Instead of listing the program themselves, another possibility is to list the programs descriptions in natural language and let the user choose the one he desires [29].

Ranking. The concept of ranking programs has origins in ranking documents in the area of information retrieval [9, 23, 32].

Given a set of input-output examples, the objective of ranking is to prioritize programs more likely to satisfy the true user intent, in order to have the desired program in the highest scores.

Although many approaches until now designed these functions manually [30, 53], there are several systems that use machine learning techniques to learn ranking functions [18, 63]. The main reason for this change is the amount of time that takes designing such functions manually. Moreover, in some applications, like FlashFill [25], these machine learning techniques already produced results better than manually constructed functions [29].

3.3 Enumeration-Based Program Synthesis

Even though there are many approaches to Program Synthesis (see section 3.1.3), one of the most common solutions is to perform an enumerative search over the space of programs that satisfy the specification. This section presents the idea behind several state-of-the-art synthesizers that use enumeration-based Program Synthesis (e.g. Morpheus [19], Neo [21], Trinity [44]). Since these synthesizers [19, 21, 44] 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) encoding.

As illustrated in Fig. 3.1, these systems take as input a set of input-output examples and a set of library components (DSL) that define the search space and enumerate programs, typically in increasing order of number of components.

The synthesis process (Fig. 3.1), can be divided into two main components: enumerator and decider. The enumerator is responsible for enumerating all possible programs accepted by the DSL provided as input. For each program $\mathcal{P}$, the decider checks if $\mathcal{P}$ satisfies the specification provided by the user. In the case of PBE frameworks, this is done by executing $\mathcal{P}$ on the input examples and checking if the output matches the expected one. Recent approaches combine enumerative search with deduction with the goal of performing early pruning of infeasible programs [19], or to learn from past failed candidate programs in order to prune all equivalent programs that are not consistent with the input-output examples, referred to as infeasible programs [21] (Reason of Failure).

For example, if we are synthesizing programs (with just one line) that use lists and we know that the output list is the same length as the input list, then, we know that such operations as `pop`, `concat`, `remove`, that alter the list's length, will not be used.

3.3.1 Tree-based Encoding

This section describes the tree-based encoding used on several state-of-the-art synthesizers [19, 21, 44] to perform program enumeration [49]. Given a DSL, Program Synthesis frameworks search for a program that is consistent with the input-output examples provided by the user. For the search process to be complete, these frameworks use a structure capable of representing every possible program up

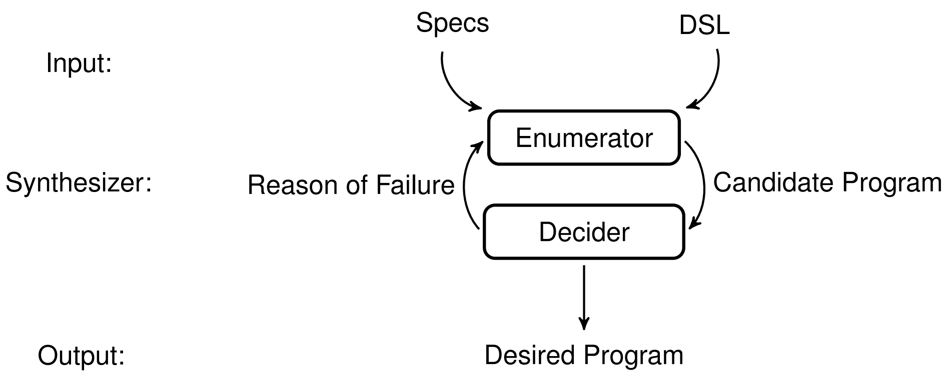


Figure 3.1: Enumeration-based Program Synthesis.

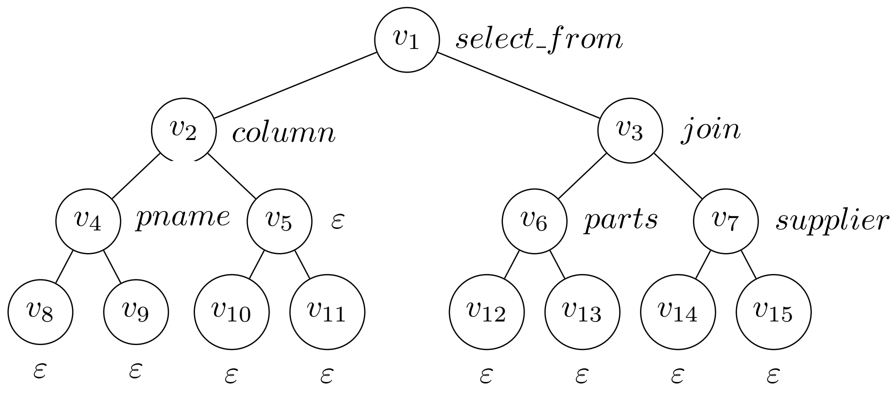


Figure 3.2: K-tree representation of the query presented in Example 4.

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. 3.2 illustrates a 2-tree of depth 3.

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.

Example 7. *Consider the query select_from(column(pname), join(parts, supplier)), from Example 4. A 2-tree of depth 3 can be used to represent this program, as shown in Fig. 3.2. When a node is not assigned to any production or symbol then it is assigned by default to ε, i.e., the empty value (e.g. $v_5, v_8, v_9, ...v_{15}$).*

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

3.3.2 Encoding Variables

Let s be the size 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 Σ. In our encoding, we assume that the empty production symbol (ε) 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;
- $B = \{b_i : 1 \leq i \leq |N|\}$: each variable b_i is a Boolean variable that denotes if node i is associated to a production symbol (true) or a terminal symbol (false);

3.3.3 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 enumerated program 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:

$$\forall i \in L : \bigvee_{p \in Term(D)} v_i = id(p) \tag{3.1}$$

Example 8. *Given the DSL D from Fig. 2.2, the set of terminal symbols is $Term(D) = \{parts, supplier, pname, sname, id, *, \varepsilon\}$ and the set of leaves is $L = \{8, 9, \ldots, 14, 15\}$, assuming a depth of 3. Each leaf node in L must be assigned to a symbol in $Term(D)$. Hence, each leaf $i \in L$ must satisfy: $v_i = id(parts) \lor v_i = id(supplier) \lor v_i = id(pname) \lor v_i = id(sname) \lor v_i = id(id) \lor v_i = id(*) \lor v_i = id(\varepsilon)$.*

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 Σ of type $Type(p, j)$.

$$\forall i \in I,\ j \in C(i),\ p \in \Sigma\ : v_i = id(p) \Rightarrow \bigvee_{t \in \Sigma(Type(p,j))} v_j = id(t) \tag{3.2}$$

With constraint (3.2), 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 9. *Consider again the query in Example 4. 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 constraints must be satisfied: $v_1 = id(select\_from) \Rightarrow \big(v_2 = id(column) \vee v_2 = id(columns)\big)$ and $v_1 = id(select\_from) \Rightarrow \big(v_3 = id(select\_from) \vee v_3 = id(join) \vee v_3 = id(parts) \vee v_3 = id(supplier)\big)$.*

Output. Let t be the output type. Furthermore, consider that the program root identifier is 1. Then, v_1, must be assigned to a symbol that is consistent with the output type t. Hence, the following constraint must be satisfied.

$$\bigvee_{s \in \Sigma(t)} v_1 = id(s) \tag{3.3}$$

Example 10. *Given the DSL D from Fig. 2.2 the set of symbols whose type is equal to the output type is $\{select\_from, join, parts, supplier\}$. Therefore, the program's root v_1 must be assigned to the identifier of one of these symbols. $v_1 = id(select\_from) \ \vee \ v_1 = id(join) \ \vee \ v_1 = id(parts) \ \vee \ v_1 = id(supplier)$.*

Input. Let IN be the set of symbols provided by the user as input. In order to guarantee that all generated programs use all the inputs provided by the user, the following constraint is added:

$$\forall p \in IN : \bigvee_{i \in N} v_i = id(p) \tag{3.4}$$

Note that this is not required for the encoding's correction. Nevertheless, we are only interested in enumerating programs that use all inputs provided by the user. Since we consider that the user only provides input tables that are required in the desired program.

Example 11. *Consider the DSL D from Fig. 2.2 and the program from Example 4. Since we have two tables in the input, $IN = \{parts, supplier\}$, at least one of the nodes must be assigned to $id(parts)$ and to $id(supplier)$: $v_2 = id(parts) \ \vee \ v_3 = id(parts) \ \vee \ \ldots \ \vee \ v_{15} = id(parts)$ and $v_2 = id(supplier) \ \vee \ v_3 = id(supplier) \ \vee \ \ldots \ \vee \ v_{15} = id(supplier)$.*

Exactly $n-1$ production rules. Finally, we are interested in enumerating programs using exactly $n-1$ production rules. Hence, the following constraints are added:

$$\forall i \in N : b_i = 1 \iff \bigvee_{p \in Prod(D)} v_i = id(p) \tag{3.5}$$

$$\Big(\sum_{i \in N} b_i\Big) = n - 1 \tag{3.6}$$

With constraints (3.5) and (3.6), we guarantee that given a k-tree of depth n, each enumerated program will have exactly $n-1$ production rules. State-of-the-art program synthesizers iteratively search for programs in increasing depth. Thus, constraint (3.6) allows to prune the search space, in order to avoid enumerating repeated programs in future iterations of depth greater than n.

3.3.4 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 that in tree-based enumeration, the number of nodes increases exponentially with the number of productions, as follows:

$$\frac{k^{n+1}-1}{k-1} \tag{3.7}$$

In terms of variables complexity, for each node in the k-tree, the tree-based encoding uses two variables, v and b. Therefore, the number of variables increases with the number of nodes, as follows:

$$2 \times \frac{k^{n+1}-1}{k-1} \tag{3.8}$$

In terms of constraints complexity, for each leaf, we add one constraint. Therefore, we add k^n constraints, k^n is the number of leaves in a k-tree. The number of internal nodes is equal to number of nodes minus number of leaves. Equation (3.2) adds a constraint for each: internal node, child (k) and symbol of the DSL, $|Prod(D)| + |Term(D)|$, as follows:

$$(\frac{k^{n+1}-1}{k-1} - k^n) \times k \times \big(|Prod(D)| + |Term(D)|\big) \tag{3.9}$$

Furthermore, we add one constraint for each input-output example: one for the output node, equation (3.3), and one constraint for each input example, $i \in IN$, equation (3.4). Equation (3.5) adds a constraint for each node in the k-tree, equation (3.7). Finally, we add one constraint using equation (3.6). Hence, in the worst case, the number of constraints used by the tree-based encoding increases exponentially, as follows:

$$\mathcal{O}\Big(k^n \times \big(|Prod(D)| + |Term(D)|\big) + |IN|\Big) \tag{3.10}$$

3.4 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 find the desired query. In Query Synthesis, like in Program Synthesis, the two most studied types of specifications are input-output examples [71, 82] and natural language descriptions [38, 79].

Query Reverse Engineering (QRE). As defined in Chapter 2, QRE is a special case of Programming By Example.

Let $\mathcal{D}$ be a database with schema graph $\mathcal{G}_S$ and let $\mathcal{Q}(\mathcal{D})$ be an output table, which is the result of running some unknown query $\mathcal{Q}$ on $\mathcal{D}$. Given $(\mathcal{G}_S, \mathcal{Q}(\mathcal{D}))$, the goal of QRE is to produce the query $\mathcal{Q}$ whose result is $\mathcal{Q}(\mathcal{D})$.

The QRE problem first appeared in 1975 in the work of Zloof [82]. Zloof introduced a new language called "Query By Example" (QBE) [82, 83, 84] 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 [82]. QRE has numerous applications [71] like database usability, data analysis and data security.

The last decade witnessed several systems for solving QRE: TALOS [70, 71], VDP [59], SQLSynthesizer [81], QFE [39], STAR [80], REGAL [68], FastQRE [36], REGAL+ [69], PALEO [51], Morpheus [19], Neo [21], Scythe [75, 76] and Trinity [44].

Ambiguity Resolution. Quite a few solutions have been proposed in the last decade regarding the system's capability to solve ambiguity, i.e. when the system can not choose between more than one query candidate. Systems like SQLSynthesizer [81], use a *ranking system* (see section 3.2.2), to order the query candidates by preference. Instead of choosing between the query candidates, other systems show the user *a top with k queries*, where these *k* queries are the ones most likely to satisfy the user's true intent [51].

As explained in section 3.2.2, some systems do *active learning* using *distinguishing inputs* in order to disambiguate between different programs. Some QRE systems like Scythe [75] and QFE [39] use this technique, asking the user to choose the correct output given some distinguishing input.

3.4.1 TALOS

One of the most important systems in QRE is TALOS [70]. Created by Tran et al. [70] in 2009 to solve the problem of *"Query by Output"*, its main goal is to derive select-project-join instance-equivalent queries (SPJ-IEQs, see Definitions 17 and 18) from a query $\mathcal{Q}$, a database $\mathcal{D}$ and the query's result $\mathcal{Q}(\mathcal{D})$.

Later, in 2014, Tran et al. [71] proposed an extension of TALOS with three different characteristics: the initial query $\mathcal{Q}$ is unknown, more expressive queries than SPJ queries can be derived and $\mathcal{D}$ can have multiple versions.

This work will only focus on the most recent version of TALOS [71], regarding the derivation of SPJ queries being the query unknown. The next definitions are inspired by TALOS [71] and will be used in the remainder of the chapter.

Definition 17 (**SPJ Query**). *A Select-Project-Join (SPJ) query $\mathcal{Q}$, is a SQL query containing only three clauses: select, from and where-clause. The select-clause is where the desired attributes to project are specified, the from-clause specifies the tables from which those attributes come from and the where-clause is where the predicates for selecting those attributes are.*

Definition 18 (**Instance-Equivalent Queries (IEQs)**). *Let $\mathcal{Q}$ and $\mathcal{Q}'$ be queries, $\mathcal{Q}$ and $\mathcal{Q}'$ are instance-equivalent queries if both produce the same output w.r.t some database $\mathcal{D}$, i.e., $\mathcal{Q} \equiv_{\mathcal{D}} \mathcal{Q}'$.*

Algorithm 3.1: TALOS($\mathcal{D}$,T), algorithm for dealing with unknown input query, from TALOS [71]. T denotes the output table of $\mathcal{Q}$, $\mathcal{Q}(\mathcal{D})$.

```
1  Let $C_1, \ldots, C_k$ be the columns in $T$
2  Let $\mathcal{Q}'_s$ denote the set of IEQs of $\mathcal{Q}$
3  foreach $column\ C_i\ in\ T$ do
4      $S_i \leftarrow \{R.A$ is an attribute in $\mathcal{D} \mid R.A$ covers $C_i\}$
5  end
6  if $all\ S_i\text{'s}\ are\ non\text{-}empty$ then
7      foreach $R_{i_1}.A_{j_1} \in S_1, \ldots, R_{i_k}.A_{j_k} \in S_k$ do
8          $\mathcal{R} \leftarrow \{R_{i_1}, \ldots, R_{i_k}\}$
9          $\mathcal{Q}'_s = \mathcal{Q}'_s \bigcup TALOS(\mathcal{R}, \mathcal{D}, \mathcal{Q}(\mathcal{D}))$        ▷ see Algorithm 3.2
10     end
11 end
12 return $\mathcal{Q}'_s$
```

Definition 19 (**Projection**). *Let α be a list of attributes from a table $\mathcal{T}$. $\pi_\alpha(\mathcal{T})$, or just π_α, is the projection of α, i.e., α is the list of attributes contained in the output table.*

Definition 20 (***proj(***$\mathcal{Q}$***)***). *Given a query $\mathcal{Q}$, let proj($\mathcal{Q}$) denote the collection of projected attributes of $\mathcal{Q}$. proj($\mathcal{Q}$) can be found in the select-clause of a SQL query.*

Definition 21 (***rel(***$\mathcal{Q}$***)***). *Given a query $\mathcal{Q}$, let rel($\mathcal{Q}$) denote the collection of relations of $\mathcal{Q}$. rel($\mathcal{Q}$) can be found in the from-clause of a SQL query.*

Definition 22 (***sel(***$\mathcal{Q}$***)***). *Given a query $\mathcal{Q}$, let sel($\mathcal{Q}$) denote the collection of selection predicates of $\mathcal{Q}$. sel($\mathcal{Q}$) can be found in the where-clause of a SQL query.*

Definition 23 (***Covering Attribute***). *Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two distinct tables and α_1 (resp. α_2) be an attribute of $\mathcal{T}_1$ (resp. $\mathcal{T}_2$). It follows that α_1 covers α_2 if $\pi_{\alpha_2} \subseteq \pi_{\alpha_1}$, i.e., $\pi_{\alpha_1} - \pi_{\alpha_2} \neq \emptyset$.*

Definition 24 (**Core relations**). *Let T and R be tables of a database $\mathcal{D}$ and α_i (resp. α_j) be an attribute of T (resp. R). Given a query $\mathcal{Q}$, $S \subseteq rel(\mathcal{Q})$ is a set of core relations of $\mathcal{Q}$ if S is a minimal set of relations such that for every attribute $\alpha_i \in proj(\mathcal{Q})$, (1) $\alpha_i \in S$, or (2) $\alpha_i = \alpha_j$ s.t. $\alpha_j \in S$.*

Description. Let $\mathcal{D}$ be a database and $\mathcal{G}_S$ its schema graph, $\mathcal{Q}$ an unknown query, $\mathcal{Q}'$ an IEQ of $\mathcal{Q}$ and $\mathcal{Q}(\mathcal{D})$ the result of $\mathcal{Q}$ on $\mathcal{D}$. In order to generate all SPJ-IEQs of $\mathcal{Q}$, TALOS needs to determine three components for each IEQ $\mathcal{Q}'$: *proj($\mathcal{Q}'$)*, *rel($\mathcal{Q}'$)* and *sel($\mathcal{Q}'$)* (Definitions 20, 21 and 22).

TALOS starts by executing Algorithm 3.1 discovering the set of schema attributes S_i that covers each column C_i in $\mathcal{Q}(\mathcal{D})$ (Definition 23). If all generated S_i are non-empty then TALOS enumerates every possible set of core relations (see Definition 24) from every S_i generated. For each generated set of core relations R_i, TALOS calls its main algorithm (Algorithm 3.2) to generate all possibles IEQs from $\mathcal{Q}$.

Concerning TALOS' main algorithm, Algorithm 3.2, it receives from Algorithm 3.1 the set of $\mathcal{Q}'$ core relations (*rel($\mathcal{Q}'$)*), easily *proj($\mathcal{Q}'$)* can be derived from *rel($\mathcal{Q}'$)*. It is not necessary that *rel($\mathcal{Q}$)* $\subseteq$ *rel($\mathcal{Q}'$)*,

Algorithm 3.2: TALOS' main algorithm, from TALOS [71].

```
Function TALOS($\mathcal{R}, \mathcal{D}, \mathcal{Q}(\mathcal{D})$):
    Let $R$ be the set of core relation of $\mathcal{Q}$
    Let $S$ be the set of IEQs of $\mathcal{Q}$ initialized to be empty
    foreach $schema\ subgraph\ G\ that\ contains\ R$ do
        Let $J$ be the join of the relations in $G$
        Enumerate a set of decision trees for $J$
        foreach $decision\ tree\ DT\ for\ J$ do
            Derive IEQ $Q'$ corresponding to $DT$
            Add $Q'$ into $S$
        end
    end
    return $S$
End Function
```

since $\mathcal{Q}$ can contain relations that are not in core relations. TALOS explores different possibilities for *rel($\mathcal{Q}'$)* in order to find interesting alternative characterizations of $\mathcal{Q}(\mathcal{D})$. Regarding the *sel($\mathcal{Q}'$)* generation, it must be succinct and insightful, and has to minimize the difference between $\mathcal{Q}(\mathcal{D})$ and $\mathcal{Q}'(\mathcal{D})$. For this reason, the system deals with the problem of finding the selection predicates as a data classification problem.

Let $\mathcal{J}$ be the result table of joining all the tables in *rel($\mathcal{Q}'$)*, based on the foreign keys joins represented in $\mathcal{G}_S$, and let L be an ordered list of *proj($\mathcal{Q}'$)*, s.t., the schema of π_L (see Definition 19) and $\mathcal{Q}(\mathcal{D})$ are equivalent. The tuples of $\mathcal{J}$ can be divided into two subsets $\mathcal{J}_{in}$ and $\mathcal{J}_{out}$, s.t., $\pi_L(\mathcal{J}_{in}) \subseteq \mathcal{Q}(\mathcal{D})$ and $\pi_L(\mathcal{J}_{out}) \nsubseteq \mathcal{Q}(\mathcal{D})$.

In Table 3.1, the first table corresponds to an input table and the second table is an output table. In this example, $\mathcal{J}_{in} = \{t_2, t_3, t_6\}$ and in $\mathcal{J}_{out}$ are all the other tuples. Therefore, TALOS enumerates a set of decision trees for $\mathcal{J}$ and derives an IEQ of $\mathcal{Q}$ for each tree. The tree decides if a row belongs or not in the output table. Finally, TALOS returns the collection of IEQs of $\mathcal{Q}$.

Regarding the classification problem, a naive approach is to put all tuples in $\mathcal{J}_{in}$ as positive. The problem with this technique is that it is too extreme and can overconstrain the classification problem, limiting its effectiveness. Multiple tuples in $\mathcal{J}_{in}$ can be projected to the same tuple in $\pi_L(\mathcal{J}_{in})$, for example the tuples t_2 and t_3 in Table 3.1. Hence, it is possible to only label a subset ($\mathcal{J}_{in}^+$) of $\mathcal{J}_{in}$ and the rest of the tuples in $\mathcal{J}$ can be labeled negative without compromising $\pi_L(\mathcal{J}_{in}^+) = \pi_L(\mathcal{J}_{in})$. Tran et al. [71] tested various approaches to label as positive the tuples in $\mathcal{J}_{in}$, the most important one was the *at-least-one semantics*.

If $\pi_L(\mathcal{J}_{in})$ = $\{t_1, \ldots, t_n\}$, then $\mathcal{J}_{in}$ can be divided into i subsets, P_i with $1 \leq i \leq n$, where each set P_i is the subset of $\mathcal{J}_{in}$ with the tuples that project the same tuple, t_i, in $\pi_L(\mathcal{J}_{in})$. Therefore, $\mathcal{J}_{in}^+$ can be defined to be made of at least one tuple from each subset P_i of $\mathcal{J}_{in}$, being $\pi_L(\mathcal{J}_{in}^+) = \pi_L(\mathcal{J}_{in})$. Hence, the tuples in $\mathcal{J}_{in}^+$ are labeled as positive and the rest of the tuples in $\mathcal{J}_{in}$ as negative.

	pID	Name	Country	Weight	Bats	Throws
t_1	P1	A	USA	85	L	R
t_2	P2	B	USA	72	R	R
t_3	P3	B	Spain	84	R	R
t_4	P4	C	USA	80	R	L
t_5	P5	D	Germany	72	L	R
t_6	P6	E	Japan	72	R	R

Name
B
E

Table 3.1: TALOS [71]: Example of running a query that selects from the first table the names of the players that have *bats* and *throws* equal to ”R”. The second table, ”Name”, is the query’s result.

Tran et al. [71] created two different classes to classify the tuples in $\mathcal{J}$: *bound tuples* and *free tuples.* The *bound tuples* are the ones that are from the start bounded to a value, i.e., if a tuple belongs to $\mathcal{J}_{out}$ then it is labeled as negative and if a tuple is the only one in a subset P_i of $\mathcal{J}_{in}$ then it has to be labeled as positive. Regarding the *free tuples*, when a subset P_i of $\mathcal{J}_{in}$ has more than one tuple then each tuple can be labeled as positive or as negative, but at least one must be labeled as positive. On account of this characteristic, this technique is called *at-least-one semantics* and thus the name TALOS, which stands for **T**ree-based classifier with **A**t **L**east **O**ne **S**emantics.

In Table 3.1, $\mathcal{J}_{in}$ can be divided in two subsets, $P_1 = \{t_2, t_3\}$ and $P_2 = \{t_6\}$. All tuples in $\mathcal{J}_{out}$ are bound tuples labeled as negatives. The only tuple in P_2 is a bound tuple labeled as positive. Regarding the tuples in P_1, they are free tuples.

The main difference between this technique and the usual classification problems is that in TALOS there is some flexibility with the labeling of tuples in contrast to the other problems where the labels are explicitly assigned.

Drawbacks. TALOS has three main disadvantages: (1) can be a bottleneck, (2) cannot derive queries with arithmetic expressions and (3) does not select between IEQs simply returns all of them. The table of pre-computed join indices [72] TALOS uses may be a bottleneck with industrial databases. In respect to the arithmetic operations, if TALOS was able to derive more complex IEQs it could generate a vast query space that would be useful in more applications. Finally, in contrast with recent approaches [51, 81], TALOS does not have a ranking system or a top-k selection. It simply returns all the IEQs computed, QRE users usually prefer a unique query that generates the given result.

3.4.2 SQLSynthesizer

SQLSynthesizer, introduced by Zhang and Sun [81] in 2013, was developed for two purposes: to be fully automated, in contrast with *Query-by-Example* [83], and to be capable of generating more complex queries with aggregates (like `MAX, MIN, SUM`) and the `GROUP BY`, `ORDER BY` and `HAVING` clauses, which was not possible with TALOS [70].

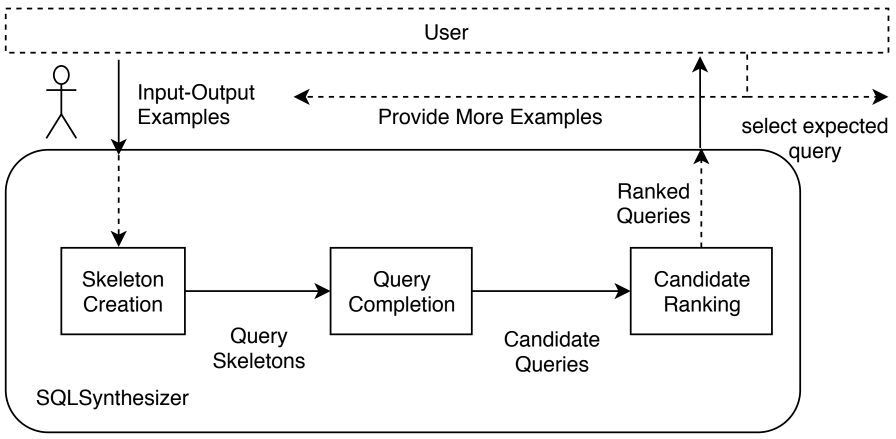


Figure 3.3: SQLSynthesizer's work flow.

Definition 25 (Query Skeleton)**.** *Query skeletons are incomplete queries with a defined structure that needs to be completed, such as the following query:*

SELECT a

FROM x, y

WHERE ??

Description. Given some input tables from a database $\mathcal{D}$ and an output table $\mathcal{Q}(\mathcal{D})$, SQLSynthesizer tries to discover a query $\mathcal{Q}$, whose result on $\mathcal{D}$ is $\mathcal{Q}(\mathcal{D})$. The overall architecture consists in three main steps (see Fig. 3.3): Skeleton Creation, Query Completion and Candidate Ranking.

Skeleton Creation is the process of creating a skeleton (see Definition 25), which is an incomplete query where the specific parts (*proj(*$\mathcal{Q}'$), *rel(*$\mathcal{Q}'$) and *sel(*$\mathcal{Q}'$)) are missing and have to be discovered searching the space of possible candidates. SQLSynthesizer searches through these candidates guided by three heuristics to determine: *rel(*$\mathcal{Q}'$), *sel(*$\mathcal{Q}'$) and *proj(*$\mathcal{Q}'$).

In order to determine the query's tables, *rel(*$\mathcal{Q}'$), SQLSynthesizer uses a heuristic to estimate it. If a column from a table in $\mathcal{D}$ appears more than once in $\mathcal{Q}(\mathcal{D})$ with a different attribute name, then it is likely that this table will be used often. Hence, being N the number of times that the column appeared in $\mathcal{Q}(\mathcal{D})$, SQLSynthesizer creates N copies of this table in the set of possible tables.

Regarding the determination of join conditions, *sel(*$\mathcal{Q}'$), instead of enumerating all possible ways to join the tables, SQLSynthesizer has two rules to join two different tables. The first one is to join tables on columns with the same name and same data type. The second rule is used for tables that do not share columns' names but share columns' data types. In this case, SQLSynthesizer creates one skeleton for every possible join between these columns.

Finally, the determination of which columns are projected, *proj(*$\mathcal{Q}'$), is done by scanning each column of $\mathcal{Q}(\mathcal{D})$. If there exists a column in $\mathcal{D}$ with the same name, the first one matched is chosen. Otherwise, the column being scanned must be the result of some aggregate.

Query Completion is where the system completes the skeletons created in the previous step and produces the list of IEQs (see Definition 18) consistent with $\mathcal{Q}(\mathcal{D})$. This step is divided into three substeps:

Table 3.2: Example of Aggregation and Comparison Features, SQLSynthesizer [81].

An input table	
C1	*C2*
2	4
2	1
2	1
1	1

Comparison Features		
C1 = C2	*C1 <C2*	*C1 >C2*
0	1	0
0	0	1
0	0	1
1	0	0

GROUP BY C1					
COUNT (C2)	*COUNT (DISTINCT C2)*	*MIN (C2)*	*MAX (C2)*	*SUM (C2)*	*AVG (C2)*
3	2	1	4	6	2
3	2	1	4	6	2
3	2	1	4	6	2
1	1	1	1	6	1

GROUP BY C2					
COUNT (C1)	*COUNT (DISTINCT C1)*	*MIN (C1)*	*MAX (C1)*	*SUM (C1)*	*AVG (C1)*
1	1	2	2	2	2
3	2	1	2	5	5/3
3	2	1	2	5	5/3
3	2	1	2	5	5/3

inferring query conditions, searching for aggregates and searching for the `ORDER BY` clause.

The inference of query conditions is a classification problem, as in TALOS. Tuples from the joined table (join of all tables in *rel*($\mathcal{Q}'$)) are divided into positive and negative tuples, being the positive ones those who belong to $\mathcal{Q}(\mathcal{D})$, while all the other tuples are in the negative part.

The problem of simply using a decision tree where the tuples' values are features, is that the decision tree fails to infer conditions using aggregates or comparisons between two columns. Therefore, SQLSynthesizer creates two additional features per tuple and uses these features along with the value of the tuple for learning. These two additional features are aggregation and comparison features.

For aggregation features, for each column of the joined table (result from the joins of the tables discovered in the previous step), the system groups all tuples by value and applies every possible aggregate to the other columns and stores the corresponding result. Regarding the comparison features, the system compares, for each tuple the values between two type-compatible columns and stores the logical value of each comparison.

For better understanding, an example from SQLSynthesizer [81] is presented in Table 3.2. SQLSynthesizer computes two tables, one stores the comparison features and the other stores the aggregation features. Considering the two columns $C1$ and $C2$ in Table 3.2, in the table named *Comparison Features* are stored the Boolean values for each comparison between every tuple in the two columns, in this case, the comparisons are $=, <$ and $>$. The Aggregation Features are stored in multiple tables. In this example, we have the two tables `Group by C1` and `Group by C2`. Each of these tables stores the values of applying the several aggregates to some column when grouped by the other column. Therefore, the first table, `Group by C1`, shows the values after applying the aggregates `COUNT, COUNT DISTINCT, MIN, MAX, SUM, AVG` to $C2$, after grouping by $C1$. *Mutatis mutandis*, the second table is computed.

Regarding the search for aggregates, for every column in $\mathcal{Q}(\mathcal{D})$ whose name does not match with any column in $\mathcal{D}$, the system will try every possible aggregate of every column in $\mathcal{D}$ and checks if the result is equal to the column in $\mathcal{Q}(\mathcal{D})$. To prune this extensive search, SQLSynthesizer uses two heuristics: only applies aggregates if a column A in $\mathcal{D}$ is type-compatible with the one in $\mathcal{Q}(\mathcal{D})$ and check if every value in $\mathcal{Q}(\mathcal{D})$'s unknown column exists in column A.

Verifying if the resulting query should have a `ORDER BY` clause is easy. If some column in $\mathcal{Q}(\mathcal{D})$ is sorted then SQLSynthesizer appends its name to the `ORDER BY` clause.

Candidate Ranking is done using the Occam's razor principle, which states that the simplest explanation tends to be the correct one. Therefore, if two queries satisfy the same input-output examples, the complex one is more likely to overfit than the simpler one. SQLSynthesizer gives a score to each query, and prefers the one with the lowest score. A query has a lower score if uses fewer query conditions or simpler aggregates.

Drawbacks. After testing the framework, four limitations of SQLSynthesizer were discovered [81]: (1) limitation in queries' complexity, (2) examples must be noise-free, (3) offer weak guidance to the user and (4) users must have complete knowledge about the database.

A vast number of queries can not be generated by SQLSynthesizer because the system's DSL is a subset of SQL. Hence there are some unsupported features. SQLSynthesizer can not proceed in the presence of a typo, so the examples given by the user must be noise-free. Additionally, SQLSynthesizer does not explain if the user should give more examples or if simply the system can not generate the desired query. Finally, if the user does not know the database's schema very well, SQLSynthesizer may not be very helpful.

3.4.3 Query From Examples (QFE)

Query **F**rom **E**xamples (QFE) is a framework developed in 2015 by Li et al. [39]. The novelty of this framework is in the interactive loop used to receive feedback from the user. The main objective of this loop is to minimize the number of questions the user needs to answer and, at the same time, the system tries to present the user with new examples similar to the initial input-output examples. This similarity is important in order to minimize the user's effort to determine if the new example is consistent with the desired query. QFE was the first system to be concerned about the user's effort since previous QRE systems did not.

Description. Given a pair $(\mathcal{D}, R)$, being $\mathcal{D}$ a database and R the result of the query $\mathcal{Q}$ desired by the user, QFE tries to produce $\mathcal{Q}$. QFE can be divided into four steps (see Fig. 3.4): Query Generator, Database Generator, Result Feedback and User Interaction.

Query Generator. This step is based on TALOS [70, 71]. This step takes the pair $(\mathcal{D}, R)$ and generates a set of IEQs of $\mathcal{Q}$ (see Definition 18), $\mathcal{Q}C = \{\mathcal{Q}_1, \mathcal{Q}_2, \ldots, \mathcal{Q}_n\}$ such that, the result of each query in $\mathcal{Q}C$ is R.

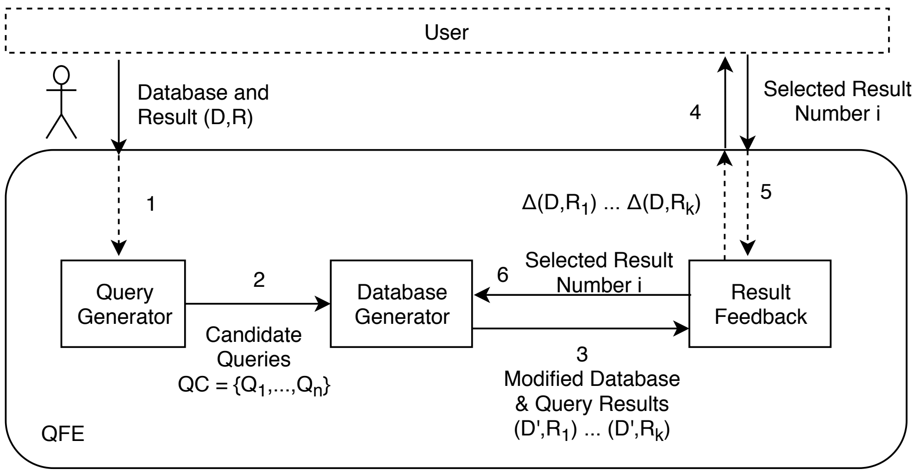


Figure 3.4: Overall Architecture of QFE [39].

In order to disambiguate between these n queries in $\mathcal{QC}$, QFE applies a *divide-and-conquer* strategy (see section 3.1.3) at each iteration of the sequence of steps 3, 4, 5 and 6 (see Fig. 3.4).

Database Generator. This step takes as input $(\mathcal{D}, R)$ and a set of queries $\mathcal{QC}'$. The purpose of this step is to generate a new database $\mathcal{D}'$, which is used to distinguish the queries in $\mathcal{QC}'$ based on their results when running on $\mathcal{D}'$. Therefore, $\mathcal{QC}'$ is divided in k $(k > 1)$ subsets, where each $\mathcal{QC}'_i \subset \mathcal{QC}'$, $1 \leq i \leq k$, contains the queries of $\mathcal{QC}'$ that produce equal result, R_i, on $\mathcal{D}'$.

Result Feedback. This step deals with the pruning of the search space. Presenting the new database $\mathcal{D}'$ and the k distinguishing outputs (R's) generated in the previous step, the user chooses which output i, $1 \leq i \leq k$, would be the correct one when running $\mathcal{Q}$ on $\mathcal{D}'$ (*User Interaction*). Thus, the system can discard all of the other subsets of $\mathcal{QC}'$, whose index is different from i.

Afterwards, QFE can start a new iteration, passing to the database generator the new $\mathcal{QC}' = \{\mathcal{QC}'_i\}$, if the size of $\mathcal{QC}'_i$, $|\mathcal{QC}'_i|$, is greater than 1. Otherwise, QFE achieved a $\mathcal{QC}'_i$ with only one candidate query, supposedly the desired one.

Instead of presenting the user the whole $\mathcal{D}'$ and the whole new R's, QFE only presents the differences between the initial pair $(\mathcal{D}, R)$ and the new pairs $\{(\mathcal{D}', R_1), \ldots, (\mathcal{D}', R_k)\}$, in order to reduce the user's effort since the user is familiar with the initial pair. This difference is denoted by $\Delta(\mathcal{D}, R)$.

If none of the R's is chosen by the user, then $\mathcal{Q}$ is not in $\mathcal{QC}'$, hence it is not in $\mathcal{QC}$. In this event, the system should generate the IEQs again exploiting the information gathered from the user.

The main objective of QFE is to reduce the user's effort, therefore the system tries to do this by reducing:

1. k, the number of distinguishing outputs in each iteration.

2. The difference between $\mathcal{D}$ and $\mathcal{D}'$.

3. The difference between R and $\overset{k}{\underset{i=1}{\forall}} R_i$.

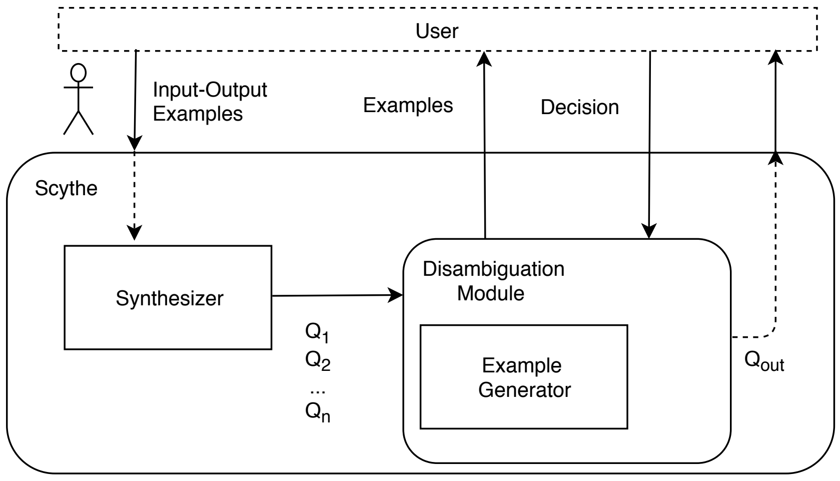


Figure 3.5: Scythe's Overview.

Although reducing k may increase the number of iterations, which may be a problem.

The cost model that quantifies user's effort and how to do the difference between the original database and the modified one at each iteration are beyond the scope of this work. The interested reader is referred to the literature [39] for more details.

Drawbacks. The limitations of this approach are similar to TALOS' limitations since the QFE's synthesis is based on TALOS [70]. For this reason, the main limitation of QFE is the complexity of queries that can be generated. QFE can only generate SPJ (select-project-join) queries. Additionally, QFE has the overhead of generating distinguishing inputs and new databases at each step.

3.4.4 Scythe

Wang et al. [75, 76] 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.

The architecture of Scythe is presented in Fig. 3.5. 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. Next in order, we have the disambiguation module which is the step responsible for choosing between the n queries provided by the previous step (synthesizer) which one is desired by the user. This disambiguation is done interacting with the user (*active learning*). In the disambiguation module, we have the example generator responsible for producing distinguishing inputs (see Section 3.2.2), a small input on which the n queries generate different outputs.

Regarding the synthesis process, Scythe decomposes this problem into two main parts: (1) synthesis of skeletons (see Definition 25) and (2) synthesis of filter predicates.

Scythe starts by enumerating all possible query skeletons that can be obtained considering the schema graph (see Definition 9) provided by the user. 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 since it is not possible to produce the output table with these skeletons.

Afterwards, all candidates use the input examples and their schema graph contain (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. [75] proposed two optimizations: locally grouping candidate predicates and encoding tables using bit-vectors. The local grouping of candidate predicates is done by constructing equivalence classes of predicates. Each class contains predicates that behave the same on a given skeleton (e.g. consider an integer attribute `id` that does not contain the value 0 (zero), so `"id > 0"` and `"id >= 0"` are equivalent predicates). Therefore, the system only needs to check if each class is consistent with the output. If this is not the case Scythe can simply remove all predicates of that class. With these optimizations the synthesis process is considerably accelerated, which allow the synthesis of a wider range of SQL operators.

Drawbacks. Scythe encodes the tables' data into constraints. Hence, its memory usage increases with the size of the tables provided by the user as input-output examples. Therefore, memory usage is the main disadvantage of Scythe.

3.5 Summary

In this chapter, we presented some background on Program Synthesis, focusing on techniques and tools that use Programming By Example. Moreover, the problem of Query Reverse Engineering was also presented. Finally, we described the most relevant systems in the area of Query Reverse Engineering: TALOS [71], SQLSynthesizer [81], QFE [39] and Scythe [75]. These frameworks have some drawbacks:

- TALOS has three main disadvantages: (1) can be a bottleneck, (2) cannot derive queries with arithmetic expressions and (3) does not select between IEQs simply returns all of them.
- SQLSynthesizer has four main drawbacks: (1) limitation in queries' complexity, (2) examples must be noise-free, (3) offer weak guidance to the user and (4) users must have complete knowledge about the database.
- QFE has two main weaknesses: (1) QFE can only generate SPJ (select-project-join) queries and (2) QFE has the overhead of generating distinguishing inputs and new databases at each step.
- Scythe's downside is its memory usage, it depends on the size of the tables used as input-output examples.

Therefore, in the next chapter, we propose a new QRE system that tries to overcome some of these difficulties. Since Scythe [75] is considered one of the best state-of-the-art SQL synthesizers, in Chapter 5, we compare our framework against Scythe, in terms of SQL generation.

In section 3.1, we explained the general idea and encoding used by enumeration-based program synthesizers such as Morpheus [19], Neo [21] and Trinity [44]. However, we did not explain in detail their architecture since our framework, presented in the next chapter, is built on top of these synthesizers. Hence, the architecture and flow of these synthesizers are similar to the ones used by our framework.

Chapter 4

SQUARES

This chapter describes 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 [44]. SQUARES' goal is to synthesize SQL queries from input-output examples (tables).

SQUARES, like Trinity, receives as input a set of input-output examples, a DSL and an interpreter. Fig. 4.1 illustrates the architecture of SQUARES' synthesizer that 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 [21], 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.

Trinity [44] by default uses tree-based encoding to search for programs. As discussed in section 3.3.1, the number of nodes used by Trinity's encoding grows exponentially with the number of production rules in a program. Therefore, we developed a new encoding, line-based encoding described in section 4.3, that scales better than the tree-based approach.

In the next two sections, 4.1 and 4.2, the input-output examples, DSL and interpreter used by our framework will be explained. SQUARES, like Trinity [44], starts by searching for programs with one production rule and iteratively increases this bound until a program that satisfies all input-output examples is found. In order to cut undesired programs, we created several *predicates* (explained in section 4.4) that encode the user's domain knowledge to the system. This knowledge is useful to guide the search through the program space and to block undesired programs.

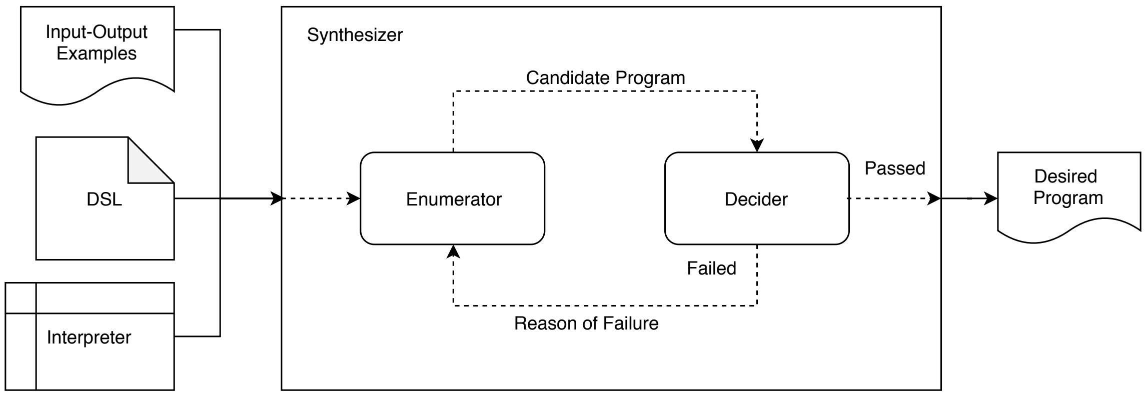


Figure 4.1: SQUARES' Overview.

4.1 Input-Output Examples

SQUARES, like most of the PBE state-of-the-art SQL synthesizers [19, 21, 44], takes as input a DSL, a set of examples, and any constants or aggregate functions (e.g., `sum`, `mean`) that the query may need.

Our framework receives as input a file with all the information provided by the user, except the DSL. The user can provide five types of information: input-output examples, constants, aggregates functions, attributes and how many lines of code (loc) the program will have. The following paragraphs contain brief descriptions of these types of information.

Input-Output Examples. In the first line of the input file, the user specifies the path to the input tables that will be used by the synthesizer as input. In the second line, the user puts the path to the output table. The input and output examples are mandatory.

Constants. The user can also provide constants that will be used in the program. SQUARES will only enumerate programs that contain these constants provided by the user. Therefore, these constants help to reduce the vast search space.

Aggregates. In the fourth line of the input file, the user can provide the name of aggregate functions that appear in the desired program. Aggregates function like: `max, min, count (n), mean, like`.

Attributes. When the user specifies a constant or an aggregate, she needs to specify which attribute is supposed to be compared against the constant or used in the aggregate function. This way, the synthesizer can generate valid conditions that use the constants and aggregates provided by the user.

Lines of Code (loc). In order to search for a program with exactly *n* production rules, the user can specify this number in the input file. Therefore, the framework can start searching from programs with n production rules instead of searching for programs with 1 production rule and iteratively increasing this bound.

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

Figure 4.2: SQUARES' DSL.

The following example, Example 12, shows one possible case how these hints provided by the user can be easily provided.

Example 12. *After an exam, a professor wants to count how many students have grades better than 9.5,* $0.0 <= grade <= 20.0$. *She does this every year and wants to find a query that can do this for her, automatically. Therefore, she provides information from the previous year to SQUARES. A table containing all the students' numbers and respective grades (input example) and a table containing the students' numbers of students with grades greater than* 9.5 *(output example). When preparing the input file she clearly can think of one constant (*9.5*), one aggregate (`count`), two attributes (*$student\ number$ *to be counted and* $grade$ *to be used in the filter) and the user can also guess that the desired query will have at least 3 lines of code (operators): `filter, count, select`.*

4.2 Domain-Specific Language (DSL)

This section describes the Domain-Specific Language (DSL) and interpreter 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[1]. Having a query in R, we can easily translate it to SQL as explained in the end of this section.

Fig. 4.2 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* [2] library (e.g. `inner_join, filter, summarise, left_join`). The terminals belonging to *filterCondition, summariseCondition, Cols, selectCols* (Fig. 4.2), are computed on the fly, because they depend on the input-output examples, as well as, the number of input tables. In Example 13, we show how these terminals can be computed.

Example 13. *Consider the following four tables: Student(snum: integer, sname: string, major: string, level: string, age: integer), Class(name: string, meets_at: string, room: string, fid: integer), Enrolled(snum: integer, cname: string), Faculty(fid: integer, fname: string, deptid: integer). A user wants to find the names of all Juniors (level = "JR") who are enrolled in a class taught by Professor I. Teach*

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

Table 4.1: Translating from our DSL to R.

Our DSL	**R**
select(x, a, True)	select(x, a) %>% distinct()
filters(x, a==1, a==2, Or)	filter(x, a == 1 | a==2)
summariseGrouped(x, maxa = max(a), a)	group_by(x, a) %>% summarise(maxa = max(a))
inner_join3(w, x, y)	inner_join(w, x) %>% inner_join(y)
inner_join4(w, x, y, z)	inner_join(w, x) %>% inner_join(y) %>% inner_join(z)

(fname="I. Teach") [3]*. In our DSL, this query can be represented by three production rules originating the following query:*

```
select(filters(inner_join4(Student,Class,Enrolled,Faculty),
fname == "I.Teach", level == "JR", And), Sname, distinct)
```

In this case, the user specifies the constants "I. Teach" and "JR", and their respective attributes. Then, the enumerator generates all possible filterConditions, with the two operators that can appear in the comparison of strings (==, !=). In this case, the set of possible filter conditions is {fname == "I. Teach", fname != "I. Teach", level == "JR", level != "JR"}.

Using our DSL, a program is valid if it corresponds to a sequence of DSL production rules, like in Enumerative Search (see Section 3.1.3). As represented in equation (4.1), a program in our framework is a sequence, that starts in a production rule that uses at least one symbol of type *table*, the input type. This sequence ends with a production rule of type *tableSelect*, the DSL's output type. Therefore, our framework uses enumerative search connecting inputs to output.

$$program\ Squares(table*)\ \rightarrow\ tableSelect \tag{4.1}$$

From R to SQL. Each production rule present in our DSL, Fig. 4.2, has a direct translation (interpretation) to R. Table 4.1 shows part of our interpreter. Only the non-trivial translations of our production rules to R are presented. All the other production rules of our DSL are equivalent in R.

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 ...`). Table 4.2 shows these direct translations. 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.

[3]This corresponds to exercise 5.1.1 from a classic textbook on databases [56].
[4]https://dbplyr.tidyverse.org/articles/sql-translation.html
[5]https://cran.r-project.org/web/packages/dbplyr/vignettes/dbplyr.html

Table 4.2: From R to SQL.

R	**SQL**

select(x, a)	SELECT x.a
	FROM x

filter(x, a == 1)	SELECT *
	FROM x
	WHERE (x.a = 1)

summarise(group_by(x, a), maxa = max(a))	SELECT MAX(x.a) as maxa
	FROM x
	GROUP BY x.a

inner_join(x, y)	SELECT *
	FROM x JOIN y
	ON x.a = y.a

left_join(x, y)	SELECT *
	FROM x LEFT JOIN y
	ON x.a = y.a

anti_join(x, y)	SELECT *
	FROM x
	WHERE NOT EXISTS
	(SELECT 1
	FROM y
	WHERE x.a = y.a)

intersect(x, y)	SELECT *
	FROM x
	INTERSECT
	SELECT *
	FROM y

bind_rows(x, y)	SELECT *
	FROM x
	UNION ALL
	SELECT *
	FROM y

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

Figure 4.3: The query *select_from(column(pname), join(parts, supplier))*, from Example 4, divided into three lines.

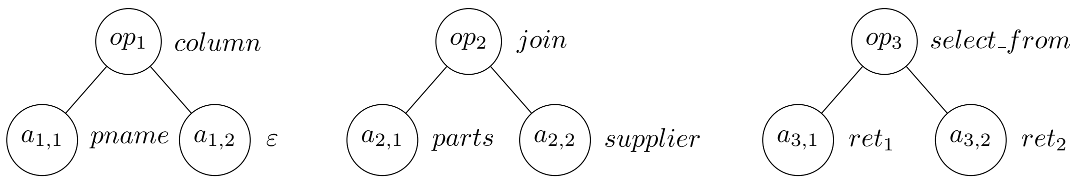


Figure 4.4: Each tree represents a production rule. The first tree represents line 1, the second tree represents line 2 and the third tree represents line 3. ret_1 (resp. ret_2) denotes the value returned in line 1 (resp. line 2).

Finally, Example 14 presents the steps taken by our system to translate a query from R to SQL.

Example 14. *Using our interpreter to translate Example 13's query from our DSL to R, the R translation would be the following query:*

```
inner_join(Student,Class) %>% inner_join(Enrolled) %>% inner_join(Faculty)
%>% filter(fname == "I.Teach", level == "JR") %>%
select(sname) %>% distinct()
```

Having the desired program in R, we can now call function show_query to get the following SQL query:

```
SELECT DISTINCT S.Sname
FROM Student S, Class C, Enrolled E, Faculty F
WHERE S.snum = E.snum AND E.cname = C.name AND C.fid = F.fid AND
F.fname = "I.Teach" AND S.level = "JR"
```

4.3 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. 4.3. One can represent this program as three trees of depth 1 as shown in Fig. 4.4. Note that the result of the program is the value returned by the third tree. Observe that ret_i is a new symbol that represents the return value of line i.

4.3.1 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.

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.

Consider the encoding for a program with n lines where the maximum arity of the operators is k, then we have the following integer variables:

- $O = \{op_i : 1 \leq i \leq n\}$: each variable op_i denotes the production rule used in line i;
- $T = \{t_i : 1 \leq i \leq n\}$: each variable t_i denotes the return type of line i;
- $A = \{a_{ij} : 1 \leq i \leq n, 1 \leq j \leq k\}$: each variable a_{ij} denotes the symbol corresponding to argument j in line i.

4.3.2 Constraints

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

$$\forall 1 \leq i \leq n : \bigvee_{p \in Prod(D)} (op_i = id(p)) \tag{4.2}$$

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

$$\forall 1 \leq i \leq n, p \in Prod(D) : (op_i = id(p)) \Rightarrow (t_i = tid(Type(p))) \tag{4.3}$$

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

$$\forall 1 \leq i \leq n, 1 \leq j \leq k : \bigvee_{s \in Term(D) \cup \{ret_r : r < i\}} (a_{ij} = id(s)) \tag{4.4}$$

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:

$$\begin{gathered}\forall 1 \leq i \leq n, p \in Prod(D), 1 \leq j \leq arity(p), 1 \leq r < i : \\ ((op_i = id(p)) \wedge (a_{ij} = id(ret_r))) \Rightarrow (t_r = tid(Type(p, j)))\end{gathered} \tag{4.5}$$

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:

$$\begin{gathered}\forall 1 \leq i \leq n, p \in Prod(D), 1 \leq j \leq arity(p), \\ s \in \{t \in Term(D) : Type(t) \neq Type(p, j)\} : \\ (op_i = id(p)) \Rightarrow \neg(a_{ij} = id(s))\end{gathered} \tag{4.6}$$

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:

$$\begin{gathered}\forall 1 \leq i \leq n, p \in Prod(D), arity(p) < j \leq k : \\ (op_i = id(p)) \Rightarrow (a_{ij} = id(empty))\end{gathered} \tag{4.7}$$

Output. Let $Type(output)$ denote the type of the program's output and let $P_O \subseteq Prod(D)$ be the subset of production rules with return type equal to Type(output), i.e., $P_O = \{p \in Prod(D) : Type(p) = Type(output)\}$. The following constraint ensures that the program's output (last line, n^{th}) has the desired type.

$$\bigvee_{p \in P_O} (op_n = id(p)) \tag{4.8}$$

Input. Let IN be the set of symbols provided by the user as input. In order to guarantee that all generated programs use all the inputs, the following constraint is used:

$$\forall s \in IN : \bigvee_{1 \leq i \leq n} \bigvee_{1 \leq j \leq k} (a_{ij} = id(s)) \tag{4.9}$$

Lines used exactly once. A feature of this new encoding is that the result of a given operation can be used more than once. Notice that in the tree-based encoding, one would have to reproduce the same operations in a different branch of the tree. In order to compare the two types of enumeration, tree-based and line-based, we can add a set of constraints restricting the usage of each operation's result to only one usage. Clearly, the following constraints are not necessary to the encoding's correction.

$$\forall ret_r \in Ret(D) : \left(\sum_{r < i \leq n, 1 \leq j \leq k} (a_{ij} = id(ret_r)) \right) = 1 \tag{4.10}$$

4.3.3 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. In tree-based enumeration, the number of nodes increases exponentially with the number of productions, (3.7). In contrast, equation (4.11) gives the number of nodes used by line-based enumeration for a given number of productions n.

$$(k+1) \times n \tag{4.11}$$

The number of nodes used by line-based enumeration increases linearly because the enumerator uses n trees, with $k+1$ nodes each, to represent a program with n production rules. On the other hand, tree-based enumeration uses a k-tree of depth $n+1$ to represent programs with n lines of code.

In terms of variables complexity, for each program line, the line-based encoding uses one operation (op), k arguments (a) and a type variables (t). Therefore, the number of variables increases as follows:

$$(k+2) \times n \tag{4.12}$$

In terms of constraints complexity, from equation (4.2), we add one constraint for each operation, n. Secondly, using equation (4.3), we add one constraint for each operation, n, and for each production rule of the DSL, $|Prod(D)|$. Therefore, we add $n \times |Prod(D)|$ constraints.

Furthermore, using equation (4.4), we add one constraint for each operation, n, and for each operation's argument, k. Hence, we add $n \times k$ constraints in the worst case. In addition, using equation (4.5), we add a constraint for each operation (n), operation's argument (k), production rule of the DSL ($|Prod(D)|$) and for each previous operation (n). Therefore, in the worst case, we add $n \times k \times |Prod(D)| \times n$ constraints.

Moreover, from equation (4.6), we add one constraint for each operation, operation's argument, production rule and terminal symbol of the DSL. As a results, in the worst case, we add $n \times k \times |Prod(D)| \times |Term(D)|$ constraints. Using equation (4.7), we add one constraint for each operation, operation's argument and production rule of the DSL. As a results, in the worst case, we add $n \times k \times |Prod(D)|$ constraints.

Regarding the input-output examples, we add one constraint for the output node, equation (4.8), and one constraint for each input example, $i \in IN$, equation (4.9). Lastly, we add one constraint for each return symbol, equation (4.10). Hence, we add n constraint because there are n return symbols.

Therefore, in the worst case, the number of constraints used by line-based encoding increases quadratically, with the number of production rules (n), as follows:

$$\mathcal{O}\Big(n \times k \times |Prod(D)| \times \big(|Term(D)| + n\big) + |IN|\Big) \tag{4.13}$$

$L_1 \ : \ ret_1 \ \leftarrow \ column(pname)$	$L_1 \ : \ ret_1 \ \leftarrow \ join(parts,\ supplier)$
$L_2 \ : \ ret_2 \ \leftarrow \ join(parts,\ supplier)$	$L_2 \ : \ ret_2 \ \leftarrow \ column(pname)$
$L_3 \ : \ ret_3 \ \leftarrow \ select\_from(ret_1,\ ret_2)$	$L_3 \ : \ ret_3 \ \leftarrow \ select\_from(ret_2,\ ret_1)$

Figure 4.5: Two different ways of representing the program from Example 4 into three lines.

4.3.4 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. There are two main reasons for this difference: (1) in the line-based encoding, the output of some line of code can be used more than once, and (2) the same program can have more than one representation, i.e. symmetric programs.

Regarding reason (1), with constraint (4.10), we guarantee that the return value of each line is used exactly once. Concerning reason (2), in the line-based encoding, some programs can be represented with different sequences of lines. However, in the tree-based encoding, as a result of the single tree representation, the arguments of each production rule will always come from the same branch.

Example 15. *Consider the DSL in Fig. 2.2 and the program `select_from(column(pname), join(parts, supplier))` from Example 4. In tree-based encoding this program has a single representation shown in Fig. 3.2. However, for the same program, line-based encoding has two possible representations shown in Fig. 4.5.*

In order for the line-based process to enumerate the same number of models than the tree-based enumeration, it is necessary to find the symmetries in the line-based encoding and block them. Otherwise, symmetric programs as the one in Fig. 4.5 will be enumerated and the synthesizer will have to check both programs. Therefore, if we have a model α 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 α can be blocked.

A simple way to find these symmetries is through a Directed Acyclic Graph (DAG) of dependencies. Let $G = (\mathcal{V}, E)$ denote a DAG, $\mathcal{V}$ is the set of vertices of G and E is the set of edges of G. A vertex is defined for each program line, and edges correspond to the line dependencies in the program. Let v_i and v_j denote the vertexes in the graph, $v_i, v_j \in \mathcal{V}$, corresponding to lines i and j with $i < j$. If the return value of line i is used as argument in line j, then a directed edge (v_i, v_j) must be added to the graph, $(v_i, v_j) \in E$. After building the graph, one can enumerate all possible topological orders of vertexes in the dependency graph. Each topological ordering corresponds to a different symmetric program. Next, each symmetric program associated with a topological order is blocked in the SMT formula.

Example 16. *Consider the program from Example 15. Line 3 (L_3) depends on line 1 (L_1) and line 2 (L_2). Therefore, lines 1 and 2 must occur before line 3. However, the order of lines 1 and 2 can be changed. Hence, two models would be blocked corresponding to permutations $L_1 - L_2 - L_3$ and $L_2 - L_1 - L_3$ of the program.*

Breaking Offline vs Online. The calculation of all possible permutations of lines in a program can be done offline, since these permutations change only with the number of lines in a program.

Therefore, to check if our implementation of symmetry breaking on the fly has a significant impact on our performance, we computed offline all programs' representations up to seven lines of code. Hence, we generated all Directed Acyclic Graphs (DAGs) that represent programs, for a given program's depth. Then, we compared both approaches, computing these permutations on the fly (online) and offline. The results of such comparison will be presented and discussed in section 5.1.2.

Regarding the offline computation of all possible permutations, first, we computed all possible DAGs, for a given number of lines of code. Secondly, for each graph we calculated all possible permutations. Then, we stored these permutations in order to use them on the fly. Therefore, for each program α blocked by our enumerator, all programs symmetric to α can be easily found and blocked. The enumerator just needs to check the file computed offline to find out all permutations of α.

4.4 Predicates

In order to cut some invalid or undesired programs from the search space we developed some predicates: *is_not_parent, happens_before, constant_occurs and distinct_inputs*. When constructing the DSL a user can make use of these predicates to provide the system with some domain knowledge. In this section these predicates are presented using the terminology explained in section 4.3. The enumerator translates these predicates into constraints and adds them to the solver in order to block some classes of programs not desired by the user.

Is Not Parent. Given two production rules, P_1 and P_2, the predicate *is_not_parent(*P_1*,*P_2*)* (4.14) guarantees that if P_1 is assigned to a line i then i will not use a previous line ($j < i$) whose production is P_2. In the tree-based encoding this means that if a node is assigned to P_1 then its children will not be assigned to P_2. This predicate is encoded as follows:

$$\begin{gathered} \forall 1 \leq i \leq n, 1 \leq j \leq k, 1 \leq r < i : \\ (op_i = id(P_1) \wedge a_{ij} = id(ret_r)) \implies \neg(op_r = id(P_2)) \end{gathered} \tag{4.14}$$

This predicate is useful to block some combinations that are not possible for a given DSL. For example, if the user provides the predicate *is_not_parent(FROM, FROM)* and a DSL for SQL, then the enumerator will block every program where the production rule `FROM` is followed by the same production rule `FROM`. This pattern is not valid in SQL but since the enumerator is using our DSL instead of a SQL grammar, then all these patterns are enumerated unless the user specifies otherwise employing this predicate. This predicate is used by other systems that use a tree-based encoding [12, 44].

Happens Before. This predicate, *happens_before(pos,pre)*, (4.15), ensures that constant pos appears in a program after constant pre, i.e., if pos is assigned to a leaf of line 3 then pre must be assigned to a leaf of line 1 or 2. This predicate is encoded as follows:

$$\begin{gathered}\forall 1 \leq i \leq n, 1 \leq j \leq k : \\ (a_{ij} = id(pos)) \implies \bigvee_{1 \leq r < i, 1 \leq m \leq k} (a_{rm} = id(pre))\end{gathered} \tag{4.15}$$

This predicate (4.15) is very useful when we want to use a variable generated previously in a program (e.g. MAX) in a condition (e.g. WHERE). For example, if we want to select the students in a class whose grade is equal to the maximum grade, we need to calculate the maximum, max. Then, filter the students whose grade is equal to max. Therefore, to only generate programs where the filter using max appears only after the calculation of max, we need to express it in the DSL using *happens_before($grade == max,\ max = max(grade)$)*.

Constant Occurs. In our system, like in most PBE systems, the user can provide constants that will be used in the program, as explained in section 4.1. Therefore, we assume that if the user provides a constant c then c must appear in the program. Hence, we created the predicate *constant_occurs(c)*, (4.16), that receives a constant c and guarantees that at least one leaf in the n trees of the program (n lines) will be assigned to c. This predicate is encoded as follows:

$$\bigvee_{1 \leq i \leq n, 1 \leq j \leq k} (a_{ij} = id(c)) \tag{4.16}$$

Distinct Inputs. The goal of *distinct_input* predicate is to cut some redundant programs from the search space. For example, the program `SELECT a FROM x WHERE x.a == 1 AND x.a == 1` is redundant since it returns the same result when comparing to the program `SELECT a FROM x WHERE x.a == 1` because the conditions being used are equivalent. Therefore, some production rules, like `WHERE` and `INNER JOIN`, should not receive the same parameter more than once. Hence, this predicate, (4.17), guarantees exactly that, i.e., that all the inputs for a given production rule are distinct. This predicate is encoded as follows:

$$\begin{gathered}\forall 1 \leq i \leq n, 1 \leq j \leq k, 1 \leq m \leq k, j \neq m : \\ (op_i = id(P)) \Rightarrow \neg(a_{ij} = a_{im})\end{gathered} \tag{4.17}$$

Chapter 5

Experimental Results

SQUARES is implemented in Python and uses the Z3 SMT solver [15] 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 [44] synthesis framework.

All of the experiments presented in this chapter 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. The goal of our evaluation was to answer the following questions:
Q1. How does line-based enumeration compare against tree-based enumeration in terms of encoding complexity? (Section 5.1)
Q2. How does line-based enumeration compare against tree-based enumeration in terms of performance? (Section 5.1.1)
Q3. How does line-based enumeration compare against tree-based enumeration for programs with more than three lines of code? (Section 5.1.1)
Q4. What is the performance impact of breaking symmetries in line-based enumeration? (Section 5.1.2)
Q5. What is the performance impact of breaking symmetries online vs offline? (Section 5.1.2)
Q6. What is the performance impact of using predicates? (Section 5.2)
Q7. What is the performance of SQUARES, in terms of SQL generation, on instances from a SQL Textbook? (Section 5.3.1)
Q8. How does SQUARES' SQL generation compare against Scythe on the instances from a SQL Textbook? (Section 5.3.1)
Q9. What is the performance of SQUARES, in terms of SQL generation, on instances from industry (OutSystems)? (Section 5.3.2)
Q10. How does SQUARES' SQL generation compare against Scythe on the instances from industry (OutSystems)? (Section 5.3.2)

Table 5.1: Number of tree nodes, variables and mean number of constraints used by each encoding for a given program's size.

	Model					
	Tree-based			**Line-based**		
Lines of Code	Nodes	Variables	Constraints (mean)	Nodes	Variables	Constraints (mean)
1	5	10	379	5	6	44
2	21	42	1,703	10	16	118
3	85	170	6,999	15	30	224
4	341	682	28,183	20	48	362
5	1,365	2,730	112,919	25	70	532
6	5,461	10,922	451,863	30	96	734

5.1 Line-based vs Tree-based Encodings

Benchmark. We designed a DSL that can solve classic SQL queries from a database textbook [56]. These instances were previously used by well-known SQL synthesizers [19, 75, 81]. We started with an initial set of 23 SQL instances (corresponding to Sections 5.1.1 and 5.1.2 of the database textbook [56]) and created variants of these instances resulting in a total of 55 instances.

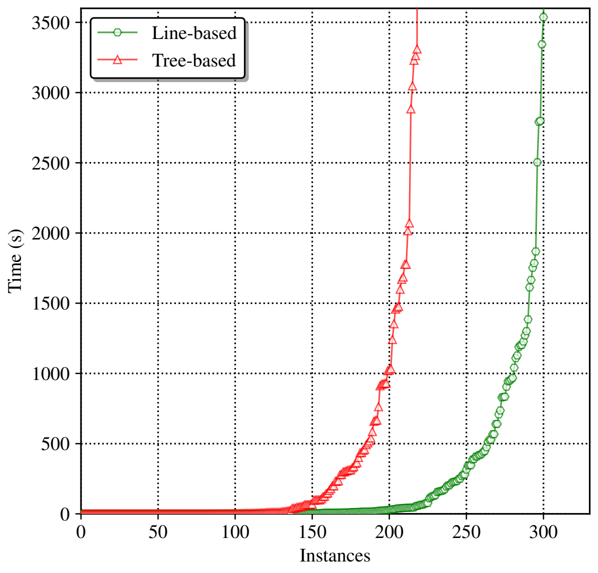
Since we want to study the performance of each encoding with respect to the size of the synthesized query, for each of these instances, we generate six different SMT formulas to search for programs that use exactly n production rules from our DSL, for a total of 330 instances (55 $\times$ 6, $1 <= n <= 6$). The SMT formulas differ in the number of productions that their programs must have, and it simulates the search performed by a program synthesizer until a solution with n production rules is found.

Encoding Complexity. As presented in the previous chapters the number of nodes used by line-based enumeration increases linearly, as presented in equation (4.11), because the enumerator uses n trees, with $k + 1$ nodes each, to represent a program with n production rules. On the other hand, tree-based enumeration uses a k-tree of depth $n + 1$ to represent programs with n lines of code. Therefore, in tree-based enumeration the number of nodes increases exponentially with the number of productions, as presented in equation (3.7).

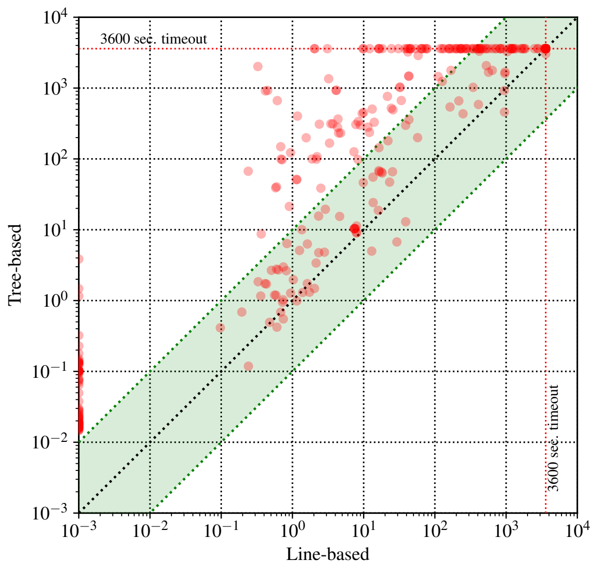
The number of variables and constraints used by each type of enumeration varies with the number of nodes. Table 5.1 shows the number of nodes, variables and the mean number of constraints used by each type of enumeration on the 330 SQL instances. Clearly, we can see that the number of nodes, variables and constraints used by line-based enumeration increase linearly while in tree-based enumeration these numbers increase exponentially. The number of nodes and variables are always the same for a given program's size. The number of constraints varies with the DSL since each instance may use different constants and aggregate functions.

Table 5.2: Number of solved instances by each encoding.

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%



(a) Running times.



(b) Comparison between encodings.

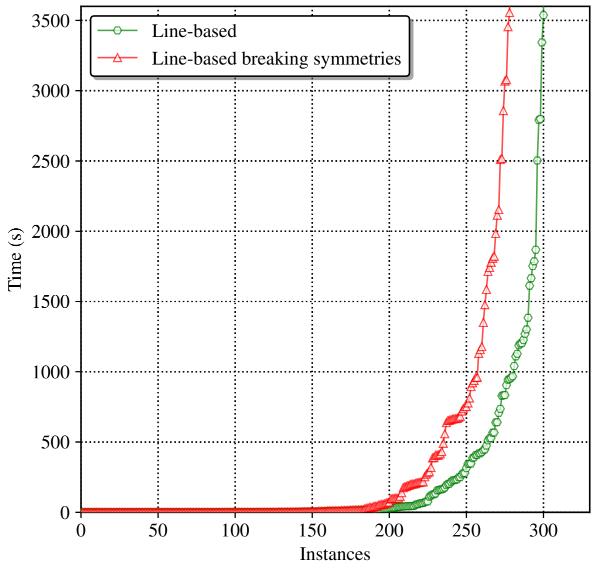
Figure 5.1: Tree-based vs Line-based Enumerators.

5.1.1 Performance

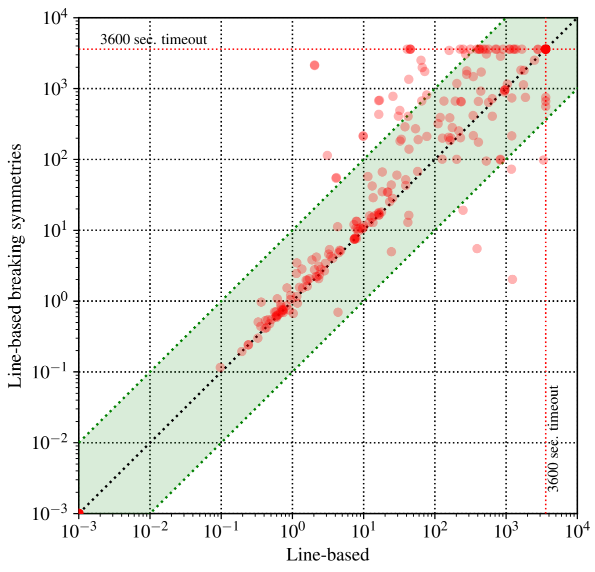
Table 5.2 shows the number of instances solved by each model for a given number of lines in our DSL. The performance of both encodings is similar for programs with three or fewer lines of code. However, when the program size increases, the difference between these approaches becomes clear. The last column of Table 5.2, 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%.

In terms of time spent in each instance, Fig. 5.1 shows two plots, a cactus plot in Fig. 5.1a and a scatter plot in Fig. 5.1b. The cactus plot shows the synthesis time (y-axis) against the number of instances solved (x-axis). Each point in the cactus plot corresponds to an instance, where the y-axis is the run time, in seconds, required by the corresponding encoding to solve that instance. Each point in the scatter plot represents an instance where the x-value (resp. y-value) is the time spent by the line-based (resp. tree-based) enumerator. Both plots in Fig. 5.1, support the results shown in Table 5.2. Additionally, the plots show that tree-based enumeration is, in general, significantly slower than line-based enumeration.

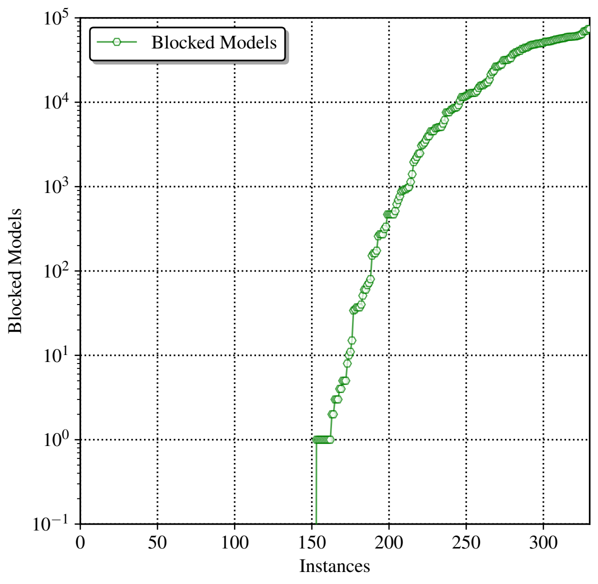
These differences in time and number of instances solved, in particular for the instances with more than 3 lines, can be justified by the exponential number of variables and constraints required by tree-based enumeration.



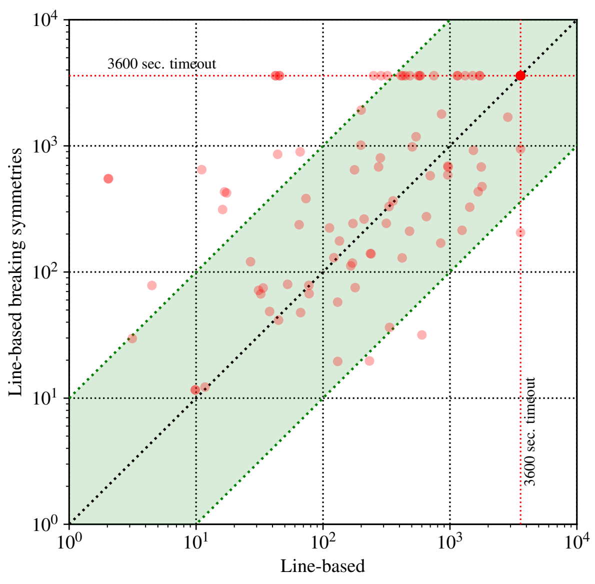
(a) Running times.



(b) Comparison between approaches.



(c) Number of blocked symmetric models per instance.



(d) Runtime comparison without considering the overhead to find symmetries.

Figure 5.2: Impact of breaking symmetries.

5.1.2 Impact of Breaking Symmetries

We evaluate the impact of symmetry breaking on the performance of line-based enumeration. For every model α, we find all models symmetric to α and add constraints to block them all. Our experiments show that symmetry breaking does not improve the performance of line-based enumeration. Fig. 5.2a and 5.2b show a cactus and scatter plot comparing line-based enumeration with and without symmetry breaking. Both figures show that symmetry breaking is, in general, significantly slower and solves fewer instances than not breaking symmetries. We conjecture that this may be due to: (i) the number of symmetries is only significant for programs with several lines, and (ii) the overhead to find and block all symmetric models is too large when compared to the time taken by each SMT call.

Fig. 5.2c shows the total number of symmetric models blocked in each instance. Programs with one or two lines of code do not have symmetries because they have only one representation. Programs with three lines of code have at most one symmetry. Therefore, only programs with more than three lines of code, have a significant number of blocked models, i.e., blocked more than a thousand symmetric models (117 instances). If we only look at these 117 instances, we observe that not breaking symmetries solves 87 instances, while breaking symmetries only solves 68 instances.

Since breaking symmetries is ineffective even when a large number of symmetries is present, we analyzed the current overhead of finding and blocking symmetric models. For each model, we spend on average 0.091 seconds to find and block all symmetric models. Fig. 5.2d shows, per instance, the time spent by the line-based enumerator with and without symmetry breaking, ignoring the time spent searching for and blocking symmetric models. This plot shows that, even if symmetry breaking was ”free”, it does not improve the performance of the line-based enumerator.

We observed that, without symmetry breaking, each SMT call takes on average 0.015 seconds. If we add symmetry breaking predicates, each SMT call doubles its time to 0.030 seconds, on average. Since our enumeration relies on solving many easy SMT calls, we concluded that the search space reduction enabled by symmetry breaking does not compensate the extra effort required to break symmetries.

Breaking Online vs Offline. As explained in section 4.3.4, we also tried to use a symmetry breaking system that uses symmetries computed offline and not on the fly. Therefore, if a program α is not consistent with the input-output examples, we block all symmetries that can be generated from α.

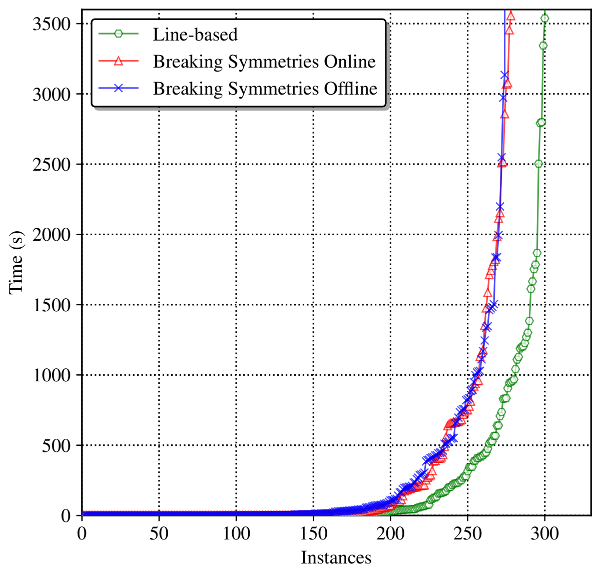
Fig. 5.3 shows the comparison between the three studied approaches: line-based, line-based breaking symmetries online and offline. From Fig. 5.3a and Fig. 5.3c, we can clearly see that both symmetry breaking approaches are quite similar. Fig. 5.3a and Fig. 5.3b shows that the basic approach, that does not break symmetries, solves more instances and is usually faster than any of the other approaches.

Based on the results presented in this section, SQUARES does not use symmetry breaking by default. Nevertheless, SQUARES can break symmetries of every program enumerated, the user just needs to provide the appropriate flags.

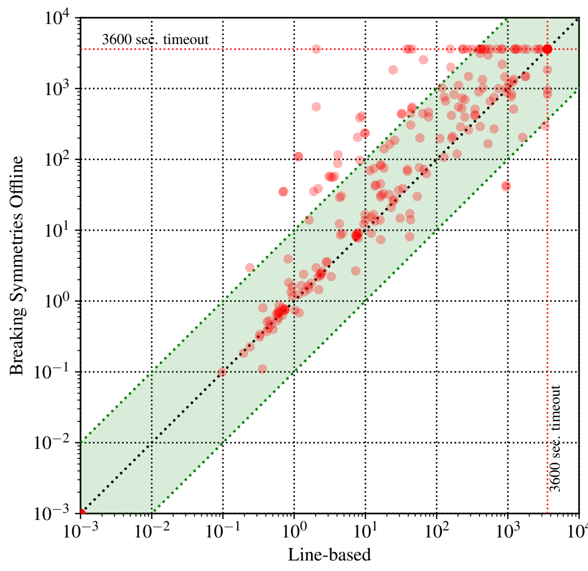
5.2 Predicates

In this section, we discuss the performance of SQUARES using the predicates presented previously: *is_not_parent, happens_before, constant_occurs and distinct_inputs*. As explained in section 4.4, these predicates allow the user to provide some domain knowledge which will help to reduce the search space.

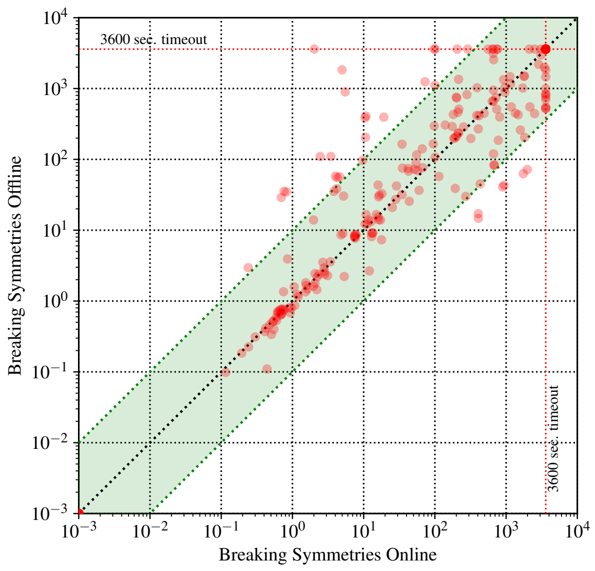
Benchmark. In this experiment we also used classic SQL queries from a database textbook [56]. As explained in the previous section, part of these instances was previously used by well-known SQL synthesizers [19, 75, 81]. In the previous section, we used the first 23 instances presented in the textbook. In this section, we had already expanded our DSL in order to use examples from OutSystems [50] and to compare SQUARES with Scythe [75]. Therefore, we were able to use the first 28 SQL instances (corresponding to Sections 5.1.1 and 5.1.2 and part of 5.1.3 of the database textbook [56]).



(a) Running times.



(b) Comparison between line-based and breaking symmetries offline.

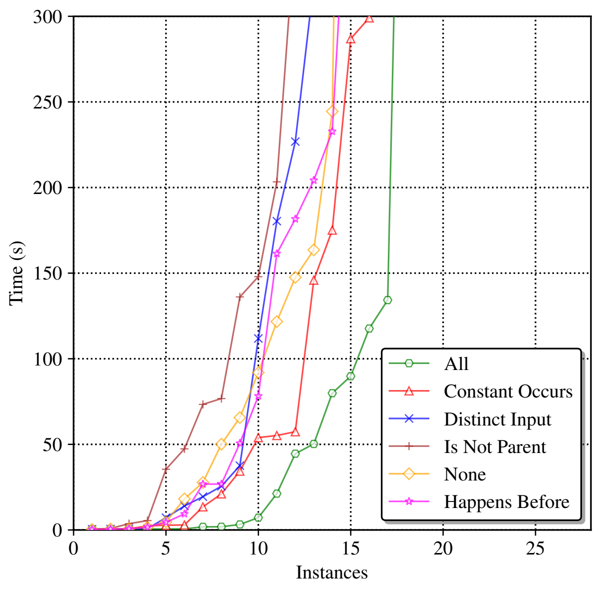


(c) Comparison between breaking symmetries online and offline.

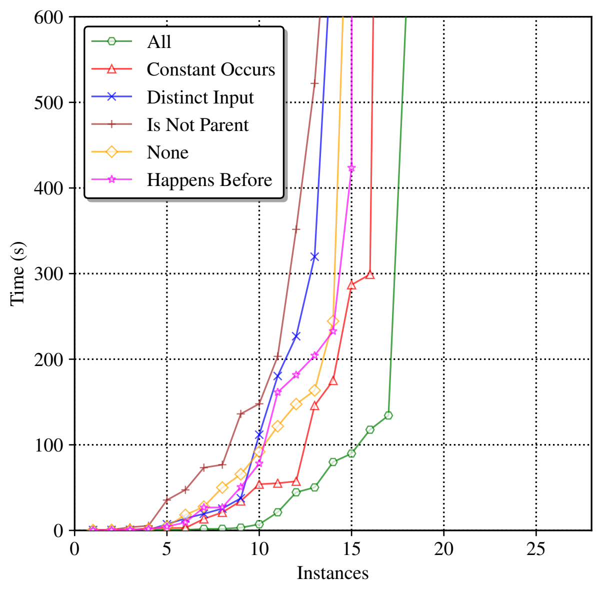
Figure 5.3: Breaking symmetries online vs offline.

Performance and Discussion. We ran six different variants of SQUARES: four using just one of the four predicates (*Is Not Parent, Happens Before, Constant Occurs and Distinct Inputs*), another using all of them together (*All*) and a final one using no predicate at all (*None*). Fig. 5.4 shows the performance of each one of these variants using timeouts of 300 (Fig. 5.4a), 600 (Fig. 5.4b) and 3600 seconds (Fig. 5.4c). The legends in Fig. 5.4 are sorted in decreasing order of number of instances solved.

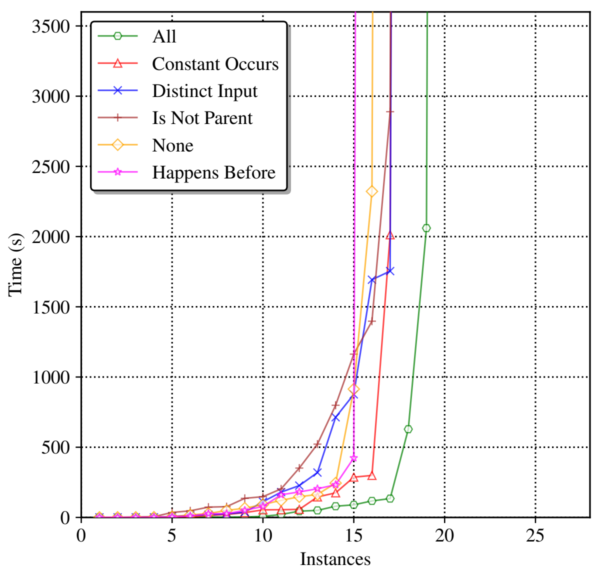
Looking at Fig. 5.4, we can see that the variant that uses all predicates (*All*) solves more instances than the other variants, no matter the timeout used. Furthermore, we can also see that the variant that uses none of the predicates (*None*) was always second or third to last which means that normally using at least one predicate helps to have a better performance.



(a) Timeout of 300s.



(b) Timeout of 600s.



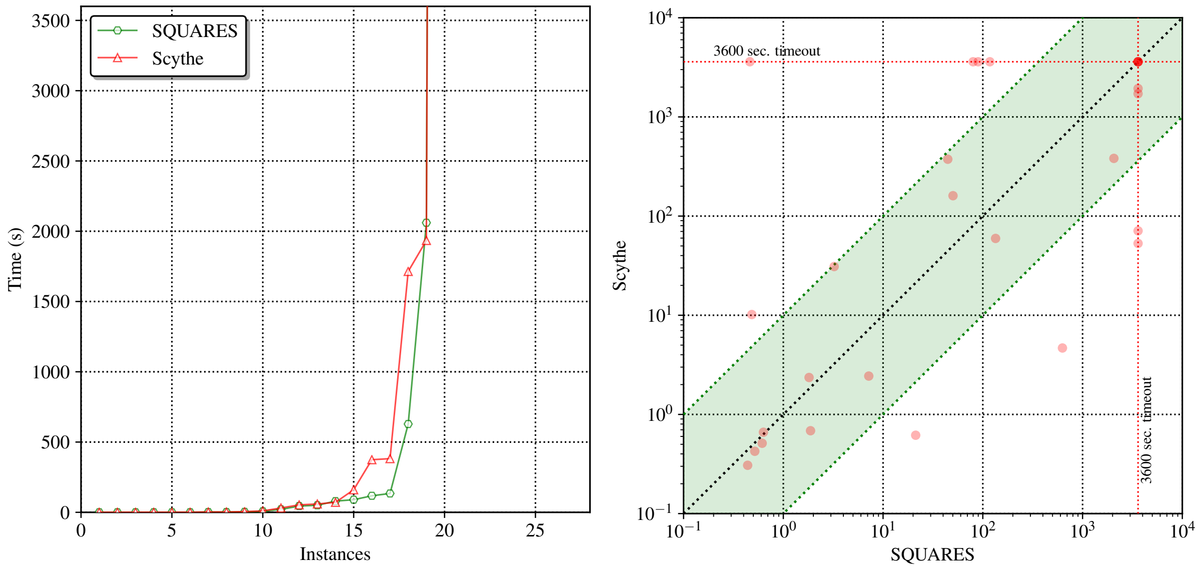
(c) Timeout of 3600s.

Figure 5.4: Cactus plots - performance of each predicate using timeouts of 300, 600 and 3600 seconds.

We gathered results of three distinct timeouts because with 3600s of timeout we could not see a clear distinction between the variants. Therefore, we also checked the results with timeouts of 300s and 600s. With only a timeout of 300 seconds, we can see that the variant *None* solves only fourteen instances, on the other hand, the variant *All* solves seventeen.

Additionally, if we look at Fig. 5.4c, we can see that the variant that uses all predicates (*All*) solves nineteen instances, while variant (*None*) only solves sixteen instances and was the second variant to solve the least number of instances.

Thus, we observed that using predicates can improve the performance of SQUARES. Therefore, the final version of SQUARES uses all of the four predicates presented and evaluated in this section.



(a) Cactus plot.

(b) Scatter plot.

Figure 5.5: Performance of SQUARES and Scythe on 28 instances from a database textbook [56].

5.3 SQL Generation

In this section, we present and discuss the performance of SQUARES concerning SQL generation. To evaluate the SQL produced by SQUARES we gather two sets of instances: one set from the database textbook [56] and the other one from OutSystems' [50] examples. Both sets will be explained in the following sections.

In the interest of evaluating SQUARES' SQL, we will compare the performance of SQUARES against Scythe [75], a state-of-the-art SQL synthesizer presented in Section 3.4.4. Thus, in Section 5.3.1 we use instances from the database textbook [56] and in Section 5.3.2 we use the examples from OutSystems. In both sections we compare the obtained results with Scythe's results.

5.3.1 Textbook Benchmark

In this subsection, we are going to present SQUARES' evaluation with well-known instances used in the area of QRE [19, 75, 81] from the database textbook [56].

Benchmark. We used the same set of tests described in Section 5.2. 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 [56]).

Performance and Discussion. Fig. 5.5 shows a cactus (Fig. 5.5a) and a scatter plot (Fig. 5.5b). The cactus plot shows the synthesis time (y-axis) against the number of instances solved (x-axis). Each point in the scatter plot represents an instance where the x-value (resp. y-value) is the time spent by SQUARES (resp. Scythe) on a given instance. These results were obtained considering a timeout of 3600s.

Both systems had a similar performance being able to solve 19 instances. However, we can see in Fig. 5.5a that SQUARES is slightly faster than Scythe. This happens in 5 of the instances solved (almost 26 percent).

Regarding the SQL generation, both systems produce verbose SQL queries. For example, Fig. 5.6 shows the queries returned by both systems for the first instance.

Example 17. *The following query would be a possible answer for exercise 5.1.1, in Fig. 5.6, if a human was writing the SQL query.*

```
SELECT DISTINCT S.S_name
FROM Student S, Class C, Enrolled E, Faculty F
WHERE S.S_key = E.S_key AND E.C_name = C.C_name AND C.F_key = F.F_key
AND F.F_name = "faculty1" AND S.level = "JR"
```

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. In Example 17, we show a possible human answer for the same exercise. However, some SQL dialects accept at most two tables in a *JOIN*. Therefore, in these cases the number of *SELECT*s would be higher. Fig. 5.6 is one of those cases.

As shown in Fig. 5.6, SQUARES provides a cleaner presentation of the SQL comparing to Scythe thanks to *sqlparse* [1], a python library to parse SQL. With this library, we parse the query and make it more readable.

In several instances, both systems produce similar queries as shown in Fig. 5.7. The two queries are quite similar in terms of the number of *JOIN*s and inner *SELECT*s. However, in other examples, Scythe produced queries with not quite as much inner *SELECT*s as SQUARES. In these cases, where the query returned by SQUARES has more inner *SELECT*s, one possible reason for this to happen is that the automatic translation from R to SQL turns every operator in an inner query.

As explained in Chapter 4, SQUARES uses a built-in function offered by *R*, called *show_query*. In some cases, the use of *show_query* can have a downside since it is still being developed and there are some operators in *R* that do not have direct translations to SQL. Therefore, there are some instances which SQUARES can only return the solution written in *R* owing to the fact that the translation is still not available.

To conclude, in this section we compared SQUARES with Scythe [75], a state-of-the-art SQL synthesizer, on instances used in the area of QRE. We observed that SQUARES solved as many instances as Scythe. Moreover, regarding the generated SQL, SQUARES produces SQL queries that are easier to read and understand.

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

```sql
Select t5.S_name
 From
  (Select t6.C_name, t6.F_key, t6.S_key, t6.C_name1, t6.F_key1, t6.F_name, input4.S_key
     As S_key1, input4.S_name, input4.level
  From ((Select *
       From
        (Select t7.C_name, t7.F_key, t7.S_key, t7.C_name1, input3.F_key As F_key1,
          input3.F_name
        From ((Select *
             From
              (Select input1.C_name, input1.F_key, input2.S_key,
                input2.C_name As C_name1
              From (input1 Join
                  input2)) As t7
             Where t7.C_name = t7.C_name1) Join
            (Select *
             From
              input3
             Where input3.F_name = faculty1))) As t6
       Where t6.F_key = t6.F_key1) Join
      (Select *
       From
        input4
       Where input4.level = JR))) As t5
 Where t5.S_key = t5.S_key1;
```

(a) Scythe SQL query.

```sql
SELECT DISTINCT `S_name`
FROM
  (SELECT `C_name`,
          `F_key`,
          `S_key`,
          `S_name`,
          `level`,
          `F_name`
   FROM
     (SELECT `C_name`,
             `F_key`,
             `S_key`,
             `S_name`,
             `level`
      FROM
        (SELECT `C_name`,
                `F_key`,
                `S_key`
         FROM `input0` AS `LHS`
         INNER JOIN `input1` AS `RHS` ON (`LHS`.`C_name` = `RHS`.`C_name`)) AS `LHS`
      INNER JOIN `input3` AS `RHS` ON (`LHS`.`S_key` = `RHS`.`S_key`)) AS `LHS`
   INNER JOIN `input2` AS `RHS` ON (`LHS`.`F_key` = `RHS`.`F_key`))
WHERE (`F_name` = 'faculty1'
       AND `level` = 'JR')
```

(b) SQUARES SQL query.

Figure 5.6: Exercise 5.1.1 from a database textbook [56] - Find the names of all Juniors (level = JR) who are enrolled in faculty1 (F_name = faculty1).

```sql
Select t3.P_name
 From
  (Select input1.S_key, input1.P_id, input2.P_id As P_id1, input2.P_name
  From (input1 Join
      input2)) As t3
 Where t3.P_id = t3.P_id1;
```

(a) Scythe SQL query.

```sql
SELECT `P_name`
FROM
  (SELECT `P_id`,
          `P_name`,
          `S_key`
   FROM `input1` AS `LHS`
   INNER JOIN `input0` AS `RHS` ON (`LHS`.`P_id` = `RHS`.`P_id`))
```

(b) SQUARES SQL query.

Figure 5.7: Exercise 5.2.1 from a database textbook [56] - Find the P_names of parts for which there is some supplier.

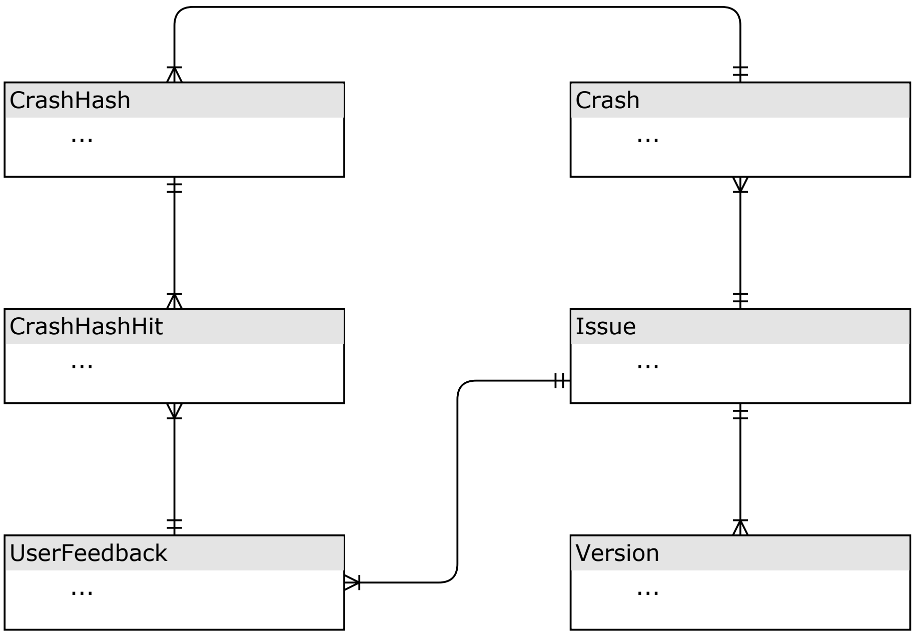
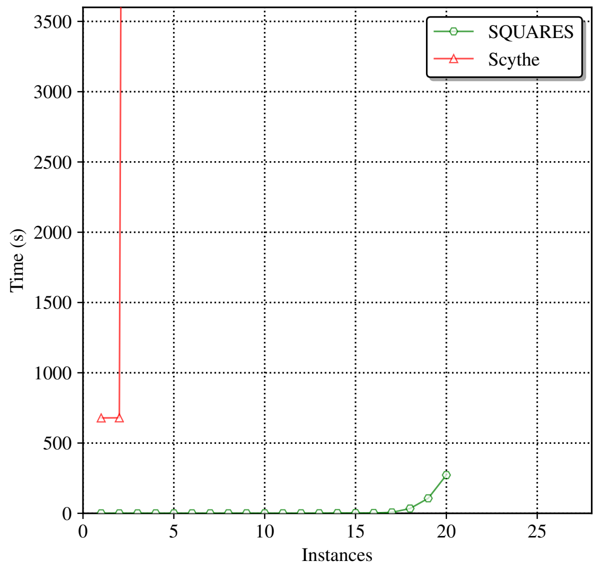


Figure 5.8: Tables from Outsystems' Database used to evaluate our work.

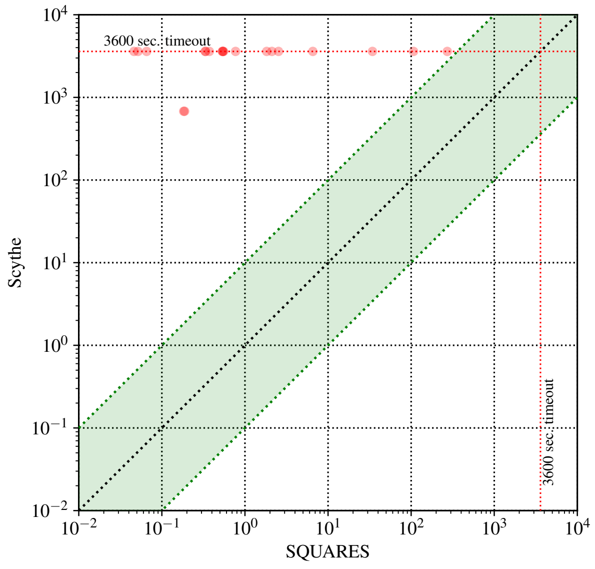
5.3.2 OutSystems Benchmark

This Master Thesis is supported by OutSystems [50] and this work was accomplished during an internship there. Hence, part of the system's evaluation was done using examples from OutSystems' database. Therefore, OutSystems provided a copy of its engineering database. With this copy, SQUARES can be tested without affecting the main database.

Moreover having a copy of the database $\mathcal{D}_C$, stagnated in time, allows us to be sure in the future that, given an output table of a query $\mathcal{Q}$ on $\mathcal{D}_C$, the result will always be the same, since the database will not be modified.



(a) Cactus plot.



(b) Scatter plot.

Figure 5.9: Performance of SQUARES and Scythe on 20 instances from Outsystems.

Benchmark. In collaboration with Outsystems' engineers and using SQL Server Management Studio [2], the 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. This way we could focus on a subgraph of the database's schema, containing these tables. Let $\mathcal{G}_{S'}$ denote this subgraph. These tables are: *Issue, UserFeedback, Crash, CrashHash, CrashHashHit, Version*. Their schema graph is presented in Fig 5.8.

Each table from $\mathcal{G}_{S'}$ has approximately two million entries, hence it was better to work with subsets of these tables. Therefore, for efficiency purposes, these tables were reduced to two thousand five hundred entries, these entries being a representative subset of the full tables.

Once we collected the queries that ran on $\mathcal{D}$ on those five days, we had a collection of 458 examples. Most of of the examples (421) used more tables than the six tables present in $\mathcal{G}_{S'}$. Moreover, seventeen of the remaining examples were equivalent between themselves. Therefore, after removing the copies and choosing only the examples that use the six tables present in $\mathcal{G}_{S'}$ we achieved a set of 20 queries.

Performance and Discussion. Fig. 5.9 presents a cactus and a scatter plot comparing SQUARES and Scythe on the OutSystems' examples. We can observe, in Fig. 5.9a, that Scythe only solves two instances out of twenty (10 percent). On the other hand, SQUARES solved all twenty instances. Fig. 5.9b supports the same results presented in Fig. 5.9a.

The poor performance of Scythe showed in Fig. 5.9 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*.

[2]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. If a human was writing the other eighteen instances, she would produced similar queries when comparing to the ones produced by SQUARES. The only significant difference is the same presented in Section 5.3.1, the number of inner `SELECT`s. However, this is due to the fact that R's function *show_query* only accepts two tables in each `JOIN`.

To conclude, this section compares SQUARES and Scythe [75] on real-world examples created by OutSystems [50]. We observed that SQUARES has a great performance in such examples, solving all of them. However, Scythe has some problems with memory usage so it can only solve ten percent of the examples. Regarding the SQL generation, both systems produced similar queries in the two instances solved by Scythe.

Chapter 6

Conclusions and Future Work

Program Synthesis has been considered the Holy Grail of Computer Science since the late 60's [29, 37]. According to Pnueli, 1996 Turing Award winner, "*One of the most central problems in the theory of programming is the systematic construction of a program from its specification*" [54].

In recent years, new platforms for software development, such as OutSystems' Low-Code Platform [1], have been made available where users with minor programming skills can create and modify software applications. These tools are able to hide many aspects of programming, but some coding experience is still needed for some operations.

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).

Since the 70's, QRE is a well-studied problem [11, 83]. Nowadays, with the massive amount of data companies have to deal with, a good and scalable QRE system is more important than ever. There already exist several approaches to deal with this problem. Some of the existent approaches solve QRE for a superset of the output table through classification [62, 71] or asking the user to select the tuples that should not be in the output [7, 8, 78]. However, SQUARES produces a query whose result is equal to the output table provided by the user.

Currently, the most common approach to Program Synthesis is to perform an enumerative search on the space of programs and find one that satisfies the specifications. Until now, enumeration-based program synthesizers [19, 21, 44] have been using a tree-based encoding (see Section 3.3.1) 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 [49].

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.

[1] https://www.outsystems.com/p/low-code-platform/

We compared SQUARES against Scythe [75], a state-of-the-art QRE framework, in order to evaluate SQUARES in terms of SQL generation. We used SQL instances from a database textbook [56] and instances from OutSystems' database. Concerning textbook instances, both systems show similar performance 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, i.e Scythe does not scale for large amounts of data. Therefore, SQUARES showed a considerable performance and was able to compete with and even to outperform a state-of-the-art SQL synthesizer.

6.1 Future Work

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. For example, consider a program with three lines of code, if the third line (`L3`) uses both line 1 (`L1`) and line 2 (`L2`) (e.g. `L3 : f(L1, L2)`) then, we can enforce that `L1` must be used as the first parameter of the function `f` assigned to `L3`, and, `L2` as `f`'s second parameter. Hence, this program `L3 : f(L2, L1)` would not be acceptable. 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. It would also be interesting to encode our representation using a SAT encoding and then compare it in terms of performance with the current SMT encoding to check if the SAT encoding allows a faster enumeration of the search space. Furthermore, it would be beneficial to develop a wider collection of predicates that could be used to prune the search space of possible programs. Finally, our encoding was designed with modularity in mind. Therefore, it should be integrated into other enumeration-based program synthesizers that are currently using the tree-based encoding, such as Morpheus [19] or Trinity [44].

With regard to our DSL, it should be extended in order to allow SQUARES to generate queries with a larger diversity of SQL operators. Lastly, a good way to deal with the problem of inner `SELECT`s in the SQL queries generated by SQUARES (see Section 5.3.1), would be to implement our own translator that would translate directly from our DSL to SQL. This way, we would not be dependent on the R's libraries (e.g. `show_query`), consequently, we could generate a higher-quality human-readable SQL.

Bibliography

[1] E. Abramovitz, D. Deutch, and A. Gilad. Interactive inference of SPARQL queries using provenance. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pages 581–592, 2018.

[2] M. Arenas, G. I. Diaz, and E. V. Kostylev. Reverse engineering SPARQL queries. In *Proceedings of the 25th International Conference on World Wide Web, WWW 2016, Montreal, Canada, April 11 - 15, 2016*, pages 239–249, 2016.

[3] S. Arora and B. Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009.

[4] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow. Deepcoder: Learning to write programs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[5] I. M. Barzdin. *Finite automata-behavior and synthesis*. North-Holland Publishing Company, 1973.

[6] 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.

[7] A. Bonifati, R. Ciucanu, and S. Staworko. Interactive inference of join queries. In *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014.*, pages 451–462, 2014.

[8] A. Bonifati, R. Ciucanu, and S. Staworko. Learning join queries from user examples. *ACM Trans. Database Syst.*, 40(4):24:1–24:38, 2016.

[9] Z. Cao, T. Qin, T. Liu, M. Tsai, and H. Li. Learning to rank: from pairwise approach to listwise approach. In *Machine Learning, Proceedings of the Twenty-Fourth International Conference (ICML 2007), Corvallis, Oregon, USA, June 20-24, 2007*, pages 129–136, 2007.

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

[11] H. Chen, G. Shankaranarayanan, L. She, and A. Iyer. A machine learning approach to inductive query by examples: An experiment using relevance feedback, id3, genetic algorithms, and simulated annealing. *JASIS*, 49(8):693–705, 1998.

[12] Y. Chen, R. Martins, and Y. Feng. Maximal multi-layer specification synthesis. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019.*, pages 602–612, 2019.

[13] 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.

[14] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158, 1971.

[15] 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.

[16] 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.

[17] G. I. Diaz, M. Arenas, and M. Benedikt. Sparqlbye: Querying RDF data by example. *PVLDB*, 9 (13):1533–1536, 2016.

[18] K. Ellis and S. Gulwani. Learning to learn programs from examples: Going beyond program structure. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 1638–1645, 2017.

[19] 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.

[20] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. W. Reps. Component-based synthesis for complex apis. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 599–612, 2017.

[21] 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.

[22] J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 229–239, 2015.

[23] Y. Freund, R. D. Iyer, R. E. Schapire, and Y. Singer. An efficient boosting algorithm for combining preferences. *J. Mach. Learn. Res.*, 4:933–969, 2003.

[24] C. C. Green and D. R. Barstow. On program synthesis knowledge. *Artif. Intell.*, 10(3):241–279, 1978.

[25] 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.

[26] S. Gulwani. Programming by examples: Applications, algorithms, and ambiguity resolution. In *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*, pages 9–14, 2016.

[27] S. Gulwani and N. Jojic. Program verification as probabilistic inference. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 277–289, 2007.

[28] 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.

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

[30] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. Complete completion using types and weights. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 27–38, 2013.

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

[32] K. Järvelin and J. Kekäläinen. IR evaluation methods for retrieving highly relevant documents. In *SIGIR 2000: Proceedings of the 23rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, July 24-28, 2000, Athens, Greece*, pages 41–48, 2000.

[33] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 215–224, 2010.

[34] V. Jojic, S. Gulwani, and N. Jojic. Probabilistic inference of programs from input/output examples. Technical report, MSR-TR-2006-103, July, 2006.

[35] 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.

[36] 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.

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

[38] F. Li and H. V. Jagadish. Nalir: an interactive natural language interface for querying relational databases. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 709–712, 2014.

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

[40] Z. Manna and R. Waldinger. A deductive approach to program synthesis. In *Proceedings of the Sixth International Joint Conference on Artificial Intelligence, IJCAI 79, Tokyo, Japan, August 20-23, 1979, 2 Volumes*, pages 542–551, 1979.

[41] Z. Manna and R. J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14(3):151–165, 1971.

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

[43] 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.

[44] 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.

[45] M. Mayer, G. Soares, M. Grechkin, V. Le, M. Marron, O. Polozov, R. Singh, B. G. Zorn, and S. Gulwani. User interaction models for disambiguation in programming by example. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology, UIST 2015, Charlotte, NC, USA, November 8-11, 2015*, pages 291–301, 2015.

[46] 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.

[47] A. Neelakantan, Q. V. Le, and I. Sutskever. Neural programmer: Inducing latent programs with gradient descent. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[48] A. V. Nori, S. Ozair, S. K. Rajamani, and D. Vijaykeerthy. Efficient synthesis of probabilistic programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 208–217, 2015.

[49] 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.

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

[51] 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.

[52] E. Parisotto, A. Mohamed, R. Singh, L. Li, D. Zhou, and P. Kohli. Neuro-symbolic program synthesis. *CoRR*, abs/1611.01855, 2016.

[53] D. Perelman, S. Gulwani, T. Ball, and D. Grossman. Type-directed completion of partial expressions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 275–286, 2012.

[54] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *Automata, Languages and Programming, 16th International Colloquium, ICALP89, Stresa, Italy, July 11-15, 1989, Proceedings*, pages 652–671, 1989.

[55] O. Polozov and S. Gulwani. Flashmeta: a framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 107–126, 2015.

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

[57] V. Raychev, M. T. Vechev, and E. Yahav. Code completion with statistical language models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 419–428, 2014.

[58] M. Raza and S. Gulwani. Disjunctive program synthesis: A robust approach to programming by example. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 1403–1412, 2018.

[59] A. D. Sarma, A. G. Parameswaran, H. Garcia-Molina, and J. Widom. Synthesizing view definitions from data. In *Database Theory - ICDT 2010, 13th International Conference, Lausanne, Switzerland, March 23-25, 2010, Proceedings*, pages 89–103, 2010.

[60] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pages 305–316, 2013.

[61] 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.

[62] Y. Shen, K. Chakrabarti, S. Chaudhuri, B. Ding, and L. Novik. Discovering queries based on example tuples. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 493–504, 2014.

[63] R. Singh and S. Gulwani. Predicting a correct program in programming by example. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 398–414, 2015.

[64] 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.

[65] 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.

[66] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 313–326, 2010.

[67] S. Srivastava, S. Gulwani, S. Chaudhuri, and J. S. Foster. Path-based inductive synthesis for program inversion. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 492–503, 2011.

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

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

[70] 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.

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

[72] P. Valduriez. Join indices. *ACM Trans. Database Syst.*, 12(2):218–246, 1987.

[73] A. J. Vijayakumar, A. Mohta, O. Polozov, D. Batra, P. Jain, and S. Gulwani. Neural-guided deductive search for real-time program synthesis from examples. *CoRR*, abs/1804.01186, 2018.

[74] 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.

[75] 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.

[76] 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.

[77] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 364–374, 2009.

[78] Y. Y. Weiss and S. Cohen. Reverse engineering spj-queries from examples. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 151–166, 2017.

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

[80] 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.

[81] 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.

[82] 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.

[83] 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.

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