TÉCNICO LISBOA

CUBES: A New Dimension in Query Synthesis From Examples

Ricardo Miguel Bacala Brancas

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Professor Vasco Miguel Gomes Nunes Manquinho
Doctor Miguel Ângelo da Terra Neves

Examination Committee

Chairperson: Professor José Luís Brinquete Borbinha
Supervisor: Professor Vasco Miguel Gomes Nunes Manquinho
Member of the Committee: Professor João Fernando Peixoto Ferreira

October 2020

“From now on you’re going to have to think.
There’s a reason why we’re born with brains in our heads, not rocks.”

— Christopher Paolini, Eragon

Acknowledgments

I would like to start by thanking my advisors, Professor Vasco Manquinho and Miguel Neves, for guiding me throughout this thesis and introducing me to the world of academic research. I would also like to thank Ruben Martins and Miguel Ventura for their invaluable insights and perspectives.

To my parents, I must be eternally grateful for encouraging me to always pursue my goals and be the best person I can be.

To Sampata, I would like to say thank you guys. This surely would not have been possible without you.

I would also like to thank Cristina, Clara, Óscar, and Pedro. Without them, my first years in Lisbon would not have been the same.

Finally, I would like to extend my thanks to all my friends and family who contributed to this work directly or indirectly.

Thank you.

Ricardo

This work was supported by OutSystems and by national funds through FCT, under projects UIDB/50021/2020, DSAIPA/AI/0044/2018, and project ANI 045917 funded by FEDER and FCT.

Resumo

À medida que a transformação digital global ganha velocidade, mais e mais pessoas vêem o seu trabalho dependente de tarefas relacionadas com a manipulação de dados. Um caso particular em que isso acontece são as Plataformas de Desenvolvimento Baixo-Código (PDBC), que permitem que utilizadores sem experiência em programação desenvolvam soluções digitais de forma rápida e eficaz. No entanto, quando é necessário implementar lógica complexa durante o desenvolvimento de uma aplicação, por exemplo ao lidar com consultas a bases de dados, essas plataformas podem ainda assim ser demasiado complexas para que um utilizador iniciante tenha sucesso. A solução para este problema é a Síntese de Programas: a tarefa de derivar automaticamente um programa com base numa especificação. Nos últimos anos, muitos avanços foram feitos nesta área. Ainda assim, devido à natureza indecidível do problema, a Síntese de Programas ainda se limita fundamentalmente a programas pequenos e simples. Para além disso, as ferramentas atuais não tiram proveito dos aumentos recentes no número de núcleos por processador.

Nesta tese de dissertação, apresentamos CUBES, um sintetizador paralelo de programas para o domínio de consultas SQL usando exemplos de entrada-saída. Usamos SQUARES como ponto de partida, e modificamo-lo, estendendo a sua Linguagem Específica de Domínio, mudando a forma como os programas são enumerados e introduzindo novas formas de poda. De seguida, usamos este novo sintetizador, CUBES-SEQ, como um bloco de construção para o desenvolvimento de um sintetizador paralelo de SQL. Exploramos técnicas usadas em *solvers* paralelos de Satisfatibilidade Proposicional e adaptamo-las ao campo de Síntese de Programas. Em particular, exploramos abordagens de portfólio e dividir-para-conquistar, que implementamos em CUBES-PORT e CUBES-DC, respectivamente. Por fim, realizamos uma extensa análise da ferramenta CUBES, comparando-a com o estado da arte anterior, em instâncias novas e pré-existentes.

Palavras-chave: Síntese de Programas, Síntese Paralelo de Programas, Engenharia Reversa de Consultas, Linguagem de Consulta Estruturada (LCE), Portefólio, Dividir-para-conquistar

Abstract

As the global digital transformation gains traction, more and more people see their work dependent on data manipulation tasks. One particular case where this is happening are Low-Code Development Platforms (LCDPs) which allow users with no background in programming to quickly develop digital solutions. Nevertheless, when complex logic is required during the development of an application, such as when dealing with queries to databases, these platforms can still be too complex for a novice user to succeed. The solution for this problem is Program Synthesis: the task of automatically deriving a program from a specification. In recent years, many advances have been made in program synthesizers. However, due to the undecidable nature of the problem, Program Synthesis is still mostly limited to small and simple programs. Furthermore, current tools do not take advantage of recent increases in the number of cores per processor.

In this dissertation thesis, we introduce CUBES, a parallel program synthesizer for the domain of SQL queries using input-output examples. We use SQUARES as a starting point, and modify it by extending its Domain Specific Language, changing how programs are enumerated and introducing new forms of pruning. We then use this new synthesizer, CUBES-SEQ, as a building block for the development of a parallel SQL synthesizer. We explore techniques used in Parallel Propositional Satisfiability solvers and adapt them to the field of Program Synthesis. In particular, we explore portfolio and divide-and-conquer approaches, which we implement in CUBES-PORT and CUBES-DC, respectively. Finally, we perform an extensive analysis of CUBES, comparing it with previous state of the art, on both pre-existing and new benchmarks.

Keywords: Program Synthesis, Parallel Program Synthesis, Query Reverse Engineering, Structured Query Language (SQL), Portfolio, Divide-and-conquer

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the age of digital transformation, more and more people are being re-assigned to tasks that require familiarity with programming or database usage. However, many users have limited knowledge in these areas [45]. A crucial tool for accelerating this digital transformation are Low-Code Development Platforms (LCDPs), such as OutSystems[1], which according to Gartner, will account for more than 65% of application development activity by 2024 [12]. These platforms allow users with very little programming knowledge to quickly and easily develop digital solutions. However, one area that is still lacking is the implementation of custom domain logic. In the particular case of database manipulations, it is common that new data analysts using these tools are domain experts, but lack the technical skills to build queries in a language such as Structured Query Language (SQL). As a result, several new systems have been proposed in order to automatically handle table manipulations in R or generate SQL queries for relational databases [9, 10, 28, 42].

The goal of Query Synthesis is to automatically generate an SQL query that corresponds to the user's intent. In many cases, the user specifies their intent through one or more examples, where each example contains a database and an output table that results from querying the database.

Figure 1.1 illustrates an input-output example with two input tables (Courses and Grades) and an output table. The output table corresponds to counting the number of grades in each course. In this example, the goal is to synthesize the following SQL query:

```
SELECT CourseName, count(*) AS 'GradeCount'
FROM Grades
    NATURAL JOIN Courses
GROUP BY CourseName
```

Observe that, for a person with limited database training, in many situations it is easier to define one or more examples than to learn how to write the desired SQL query. Even for people that work with LCDPs, with some SQL knowledge, query synthesizers can decrease the time to write database queries. In this scenario, reducing the time spent in query synthesis becomes crucial. A possible technique that

[1] https://www.outsystems.com/

CourseID	CourseName
10	Programming
11	Algorithms
12	Databases

(a) The `Courses` table.

CourseID	StudentID	Grade
10	36933	A
11	36933	B
12	36933	A
10	37362	A
12	37362	C
11	37453	A
10	37510	B
12	37510	A
10	37955	A

(b) The `Grades` table.

CourseName	GradeCount
Programming	4
Algorithms	2
Databases	3

(c) The output table.

Figure 1.1: Two input tables: `Courses` and `Grades`. Output table: number of grades per course.

remains as of yet unexplored in the field of table manipulation synthesis is to use parallelism to reduce synthesis time while also increasing the number of problems that can be solved.

1.1 Contributions

In this thesis, we introduce CUBES, a novel parallel synthesizer for SQL queries. We start by extending an existing synthesizer, SQUARES, in order to improve its performance and expand the range of queries it supports. Next, we use that new synthesizer as a building block for the development of parallel synthesis algorithms. To summarize, this thesis makes the following contributions:

- We support additional table manipulation operations when compared to current state-of-the-art synthesizers, thus increasing the range of supported programs;
- We improve the enumeration of possible programs in the synthesis procedure by using a more suitable constraint solver configuration, and integrate additional pruning techniques based on properties of the input;
- We introduce a parallel SQL synthesizer using portfolio approaches, which consist in using different strategies to explore the full search space in parallel, and divide-and-conquer approaches, which consist in splitting the search space in smaller partitions and searching those partitions in parallel;
- We perform an extensive experimental analysis on a very large set of benchmarks in order to evaluate each of the proposed techniques.

Overall, we implement three modes of operation in CUBES:

- CUBES-SEQ: a sequential SQL synthesizer;
- CUBES-PORT: a parallel SQL synthesizer using portfolio solving;
- CUBES-DC: a parallel SQL synthesizer using divide and conquer.

1.2 Document Structure

This document is organized as follows. In chapter 2 we start by introducing preliminary concepts needed to understand the rest of this document. Next, in chapter 3 we introduce Program Synthesis and Parallel Propositional Satisfiability solving and discuss state of the art techniques for these fields. Afterwards, chapter 4 discusses sequential state of the art tools for the synthesis of SQL queries that are relevant to our work.

Following that, in chapter 5, we present our new sequential synthesizer, CUBES-SEQ, that improves upon and extends the SQUARES synthesizer. In chapter 6, we propose CUBES-PORT and CUBES-DC, two new parallel synthesizers that integrate known techniques from Parallel Constraint Reasoning solvers and adapt them to Parallel Program Synthesis. Next, in chapter 7 we evaluate the different configurations of our solver and compare them to previous state of the art in SQL synthesis. Finally, we conclude with some final remarks in chapter 8.

Chapter 2

Preliminaries

In this chapter we introduce background concepts that are required to understand the rest of the document. In particular, we start by introducing Propositional Satisfiability and Satisfiability Modulo Theories. Next, we introduce tables and queries, since this work focuses on the synthesis of table manipulations. Finally, we define languages, programs, and the Program Synthesis problem.

2.1 Propositional Satisfiability

Let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of Boolean variables. A literal is a variable $x \in X$ or its negation $\neg x$. A clause is a disjunction of literals: $c = (l_1 \vee l_2 \vee \cdots \vee l_k)$. Finally, a formula in Conjunctive Normal Form (CNF) is a conjunction of clauses: $\phi = (c_1 \wedge c_2 \wedge \cdots \wedge c_m)$.

An assignment to the formula's variables is a mapping from X to {true, false}. We say that a clause is satisfied by some assignment α if any of its literals are true under α. Moreover, a formula is satisfied by an assignment α if all its clauses are satisfied.

The Propositional Satisfiability (SAT) problem consists in, given a CNF formula ϕ, finding an assignment such that ϕ is satisfied or prove that no such assignment exists.

Example 2.1.1 *Let $\phi = (x_1 \vee x_2) \wedge (x_3 \vee \neg x_2)$. This formula is* satisfiable *(SAT) and a possible satisfying assignment is $\{x_1 \mapsto$ true$, x_2 \mapsto$ false$, x_3 \mapsto$ true$\}$.*

Now consider the formula $\phi' = (x_1 \vee x_2) \wedge (x_3 \vee \neg x_2) \wedge \neg x_1 \wedge \neg x_3$. The formula is unsatisfiable *(UNSAT) because no assignment exists that satisfies ϕ'.*

2.2 Satisfiability Modulo Theories

SMT is a generalization of SAT where the domain of variables is extended with regard to some background theory, $\mathcal{T}$ [3]. A *theory* is defined by a set of axioms in the underlying logic.

Given a 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 negation $\neg t$. Finally, a $\mathcal{T}$-formula is like a propositional formula but composed of $\mathcal{T}$-literals instead

of propositional literals. The SMT problem consists of finding an assignment to the variables in a given formula ϕ such that ϕ is satisfied, or prove that no such assignment exists.

Some examples of theories include the theory of Equality with Uninterpreted Functions (EUF), the theory of Linear Integer Arithmetic (LIA) and the theory of data types [2]. It is also possible to combine different theories [3].

Example 2.2.1 *Consider the formula $\phi := x > z+1 \wedge x < 0$, in the theory of LIA. A satisfying assignment to the formula is, for example, $\{x = -1, z = -5\}$.*

Consider now the formula $\phi' := x \geq 0 \wedge y > 42 \wedge x + y = 7$. It is easy to see that there can be no satisfying assignment to ϕ'. Therefore, we say that ϕ' is unsatisfiable.

2.3 Tables and Queries

Definition 2.3.1 (Table) *A table is a tuple (H, B), where H is a named tuple of domains and B is a set of named tuples such that for all name-value pairs, (n, v), of all elements of B, $v \in H.n$ should hold. That is, the elements of B should respect the domains defined in H.*

Example 2.3.1 *In Figure 2.1, we present a few examples of tables. In the rest of this document, we omit the domains when they can be easily inferred or are not relevant. The first row of each table represents the header, H, and the following rows represent the body, B.*

recipe_id [Nat]	recipe_name [Str]
1	Cake
2	Scrambled Eggs

(a) The `recipe` table.

ing_id [Nat]	ing_name [Str]
1	Eggs
2	Flour
3	Sugar

(b) The `ingredient` table.

recipe_id [Nat]	ing_id [Nat]	amount [Str]
1	1	2
1	2	1 cup
1	3	2/3 cup
2	1	3

(c) The `amount` table.

Figure 2.1: Example tables. These tables are used throughout the document.

Definition 2.3.2 (Query) *A query is a function $\mathcal{Q} : \mathcal{D}^n \mapsto \mathcal{D}$, where $\mathcal{D}$ is the domain of tables.*

Example 2.3.2 *Consider a function `join` that takes as argument two tables, and returns a new table that corresponds to the natural join between the two tables.*

A possible query using this function would be `join(join(recipe, amount), ingredient)` where `recipe`, `amount` and `ingredient` are the input tables. This would be equivalent to the SQL query:

```
SELECT * FROM recipe
NATURAL JOIN amount
NATURAL JOIN ingredient;
```

Applying this query to the tables from Example 2.3.1 results in the following new table:

ing_id	*recipe_id*	*recipe_name*	*amount*	*ing_name*
1	*1*	*Cake*	*2*	*Eggs*
1	*2*	*Scrambled Eggs*	*3*	*Eggs*
2	*1*	*Cake*	*1 cup*	*Flour*
3	*1*	*Cake*	*2/3 cup*	*Sugar*

2.4 Languages and Programs

A formal language $\mathcal{L}$ over an alphabet Σ is a subset of Σ^*, where Σ^* is the set of finite sequences composed of elements of Σ. For the purpose of this document, we restrict languages to those that can be defined using a context-free grammar.

Definition 2.4.1 (Context-free Grammar) *A Context-free Grammar (CFG) is represented as a tuple* $G = (V, \Sigma, P, s)$ *[37] where:*

1. V *is a finite set of symbols, called non-terminal symbols;*
2. Σ *is a finite set of symbols, called terminal symbols;*
3. $P \in V \to (V \cup \Sigma)^*$ *is a finite set of production rules;*
4. $s \in V$ *is the starting symbol.*

In this document terminal symbols are typeset in `monospace` while non-terminal symbols are highlighted in *italic*.

Example 2.4.1 *Consider the following context-free grammar for a small set of operations with tables:*

$$
\begin{aligned}
\mathit{table} &\to \texttt{recipe} \mid \texttt{ingredient} \mid \texttt{amount} \\
&\quad \mid \texttt{select}(\mathit{table}, \mathit{cols}, \mathit{distinct}) \mid \texttt{join}(\mathit{table}, \mathit{table}) \\
\mathit{cols} &\to \mathit{col} \mid \mathit{col}, \mathit{cols} \\
\mathit{col} &\to \texttt{recipe\_id} \mid \texttt{recipe\_name} \mid \texttt{ing\_id} \mid \texttt{ing\_name} \mid \texttt{amount} \\
\mathit{distinct} &\to \texttt{true} \mid \texttt{false}
\end{aligned}
$$

Some examples of valid elements of the language this grammar represents are: `recipe`; `join(recipe, amount)` *and* `select(recipe, recipe_name, false)`.

Definition 2.4.2 (Domain Specific Language) *A DSL is a language that can be represented by a context-free grammar, augmented with a* semantics *that specifies the meaning of the productions of the language.*

Unlike full programming languages like C or Python, DSLs are only suitable for a specific purpose, such as table manipulation tasks.

Definition 2.4.3 (DSL Component) *The components of a DSL are the productions of the underlying CFG that represent an operation of the language. Components typically have arguments which are the non-terminal symbols of the right-hand side of the production.*

Example 2.4.2 *The components of the DSL with the CFG presented in Example 2.4.1 are* `select` *and* `join`*.*

Definition 2.4.4 (DSL Semantics) *The semantics of a DSL is a mapping, Ψ, from each construct of the grammar to an SMT formula that relates the inputs of that construct with the output. The semantics may be under-specified, or not, depending on the usage.*

Example 2.4.3 *Consider a DSL that contains a construct* `add`(n, n)*. A possible semantics for this construct could be $y = x_1 + x_2$, where y represents the return value and x_1 and x_2 represent the inputs.*

Definition 2.4.5 (Program Space) *Program space refers to the set of possible productions of the grammar in a given DSL.*

Definition 2.4.6 (Complete Program) *A* complete program *on some DSL is a production of the grammar of that DSL, containing only terminal symbols.*

Definition 2.4.7 (Partial Program) Partial programs *on some DSL are productions of the grammar of that DSL, containing non-terminal symbols and as such represent many possible complete programs. Furthermore, a* sketch *is a partial program where all missing constructs are either inputs or constants.*

Example 2.4.4 *Some complete programs produced by the grammar in Example 2.4.1 are:* `recipe` *or* `join(recipe, amount)`*. By contrast, some examples of incomplete programs are:* `join(recipe,`$table$`)` *or* `select(`$table$`,`$distinct$`)`*.*

Programs can be represented in textual form or using an AST. In Figure 2.2 we present a program and the corresponding AST, using the grammar in Example 2.4.1.

2.5 Synthesis Problem

According to Gulwani et al. [14], Program Synthesis is a second-order search problem where the goal is to find a program that satisfies a given specification.

```
join(join(recipe, amount),
     ingredient)
```

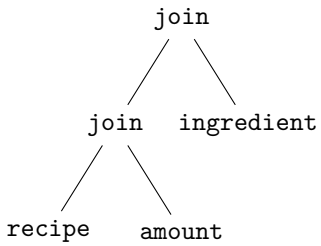


Figure 2.2: Example of a program produced by the grammar from Example 2.4.1, and the corresponding AST.

Definition 2.5.1 (Problem Formulation) *The synthesis problem consists in solving the second-order formula*

$$\exists f \, \forall \mathbf{x}. \, P[f, \mathbf{x}]$$

where f is the second-order variable representing the program and P is a relation between programs and arguments, such that $P[f, \mathbf{x}]$ is true if and only if program f satisfies the specification for input $\mathbf{x}$.

Under this formulation, finding a solution to the synthesis problem is equivalent to finding a valid assignment to f. However, being a second-order problem means that the problem is undecidable in general and that a complete and sound algorithm does not exist.

The specification for a Program Synthesis problem can be of many types, such as: type-based [29], natural language [4, 5, 44], formal specifications [30], code snippets [5] or input-output examples [4, 7, 9, 10, 33, 36, 42]. In our case, we focus on problems specified using input-output examples.

A specification consisting of input-output examples is a set, $\mathcal{E}$, containing pairs $(\mathbf{x_i}, y_i)$. Each pair is called an example where $\mathbf{x_i}$ are the inputs and y_i is the expected output.

Example 2.5.1 *The tuple $((\texttt{Courses}, \texttt{Grades}), \textit{Output})$, using the tables from Figure 1.1, constitutes an input-output example for the following query:*

```
SELECT CourseName, count(*) AS 'n'
FROM (SELECT *
      FROM Grades NATURAL JOIN Courses)
GROUP BY CourseID
```

Finally, the problem of Program Synthesis based on input-output examples can be stated as:

Definition 2.5.2 (Programming by Example) *Given a set, $\mathcal{E}$, of input-output examples $(\mathbf{x_i}, y_i)$ the Programming by Example problem can be defined as follows:*

$$\exists f. \bigwedge_{(\mathbf{x_i}, y_i) \in \mathcal{E}} f(\mathbf{x_i}) = y_i$$

Chapter 3

Background

This chapter describes the theoretical background required for the development of a parallel program synthesizer, and is divided in two sections. In the first section, we describe the Program Synthesis problem and the different techniques that can be used for synthesis. In the second section, we explain different methods for solving constraint problems using multiprocessing.

3.1 Program Synthesis

In this section, we make a brief overview of previous work on the field of Program Synthesis. We begin with some of the first approaches and work through their shortcomings and evolution. Finally, we discuss how these techniques can be applied to synthesize queries from examples.

3.1.1 Deductive Synthesis

Deductive approaches to Program Synthesis require a complete formal specification of the intended behavior for the program. Some of the first automated systems [13] for Program Synthesis worked by using a system of axioms and deductive rules. These rules were then used to construct a proof of the specification [31]. Finally, it was possible to derive a correct program from the proof.

The deductive approach to Program Synthesis has become less popular over time as writing the required formal specifications can sometimes be as hard as writing the program itself [14].

3.1.2 Inductive Synthesis

A different approach is to work with incomplete specifications that only partially specify the expected behavior. One particularly interesting instance of inductive synthesis is Programming by Example (PBE) [31], as formalized in Definition 2.5.2. Unlike deductive synthesis approaches, PBE tools [7, 9, 10, 31, 33, 36, 42] are much easier to use as they require only a set of input-output examples as specification. Therefore, using PBE does not require knowledge of formal logic.

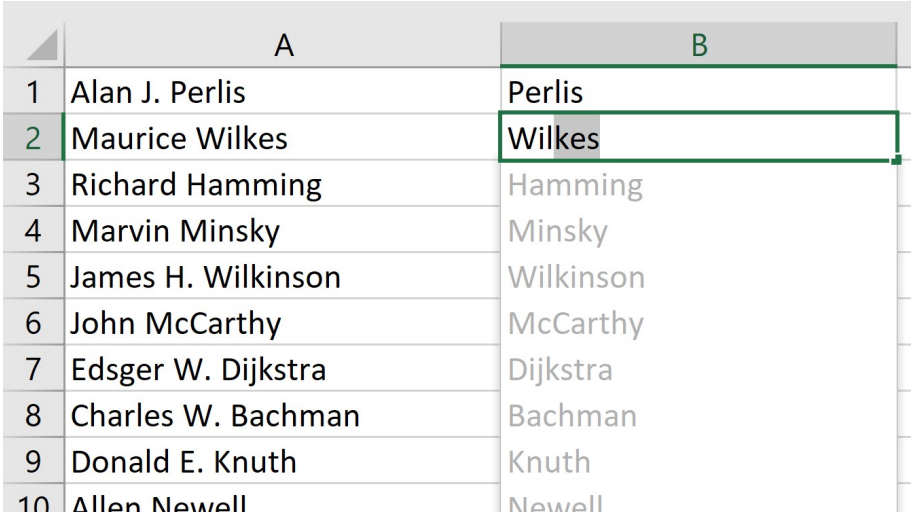


Figure 3.1: Example of Flash Fill in action, running on Excel version 1910.

The main drawback of this approach is that, by definition, there may be different programs, with different behaviors, that satisfy the given partial specification. This means that one may need to disambiguate between several candidate programs.

Distinguishing Inputs One technique for choosing between different possible programs is called *distinguishing inputs* [19]. When using this approach, the synthesizer returns a set of new input-output pairs such that the candidate programs behave differently from each other. The user needs only to choose the correct pair and the synthesizer adds it to the specification and tries to synthesize a new program. This is done until only one candidate program is left.

Flash Fill A particularly prominent real-world application of PBE is Flash Fill [15], available in Excel since 2013. This tool can be used to auto-complete columns after manually filling in the first few examples. In Figure 3.1 we show an example of Flash Fill in execution. The user has already filled the first column of the sheet with the names of the ACM Turing award recipients and is now filling the second column. Flash Fill uses the first few lines of the table as examples and induces a program that implements the users' intent. In this particular case, it induces that the user wants the column to contain the last names, offering to fill the rest of the entries.

3.1.3 Syntax-Guided Synthesis

A popular method to improve inductive synthesis procedures is to restrict the space of possible programs using syntactic constraints. By using a DSL instead of a full programming language, the search space is reduced, leading to improvements to the tractability of the problem [1].

This approach has been formalized under the name Syntax-Guided Synthesis (SyGuS). The work by Alur et al. [1] describes the input to the SyGuS problem as consisting of a background theory (like the ones described in section 2.2), a semantic correctness specification (as an SMT formula) and a syntactic

description of candidate implementations (as a DSL). This approach results in a general framework that is not restricted to a single programming language or paradigm.

However, SyGuS also has the disadvantage that it requires both the DSL and the specification to be defined in terms of the theories supported by SMT solvers. This makes some problems very difficult to specify, as is the case of table manipulation operations.

DSL Design

According to Gulwani et al. [14] there are four main factors to take into account when designing a DSL for Program Synthesis:

- **Balanced Expressivity** – the DSL should be expressive enough to enable all relevant tasks to be articulated, but restrictive enough so that the search space does not become so large that the synthesizer is unable to provide a solution in a reasonable amount of time;
- **Operator Choice** – the operators in the DSL should be such that the synthesizer can reason about the semantics of the programs;
- **Naturalness** – the programs allowed by the DSL should be easy to understand by the users;
- **Efficiency** – the operators should have fast implementations, such that the resulting programs are not prohibitively slow (there have been developments on this front by Knoth et al. [21]).

3.1.4 Counter-Example Guided Inductive Synthesis

As described in section 2.5, Program Synthesis is a second-order problem:

$$\exists f \, \forall \mathbf{x}. \, P[f, \mathbf{x}] \tag{3.1}$$

However, verifying if a given program f satisfies a specification is a first-order problem:

$$\forall \mathbf{x}. \, P[f, \mathbf{x}] \tag{3.2}$$

Furthermore, instead of trying to prove that P holds for all possible inputs, we can try to prove that there is no input such that P does not hold. To that end, we can equivalently try to refute the following formula:

$$\exists \mathbf{x}. \, \neg P[f, \mathbf{x}] \tag{3.3}$$

Therefore, a common technique for Program Synthesis is to perform a search over the space of programs and then verify each candidate solution, by trying to falsify the formula above. In Equation 3.3, $\mathbf{x}$ is called a counter-example – an input such that the program behaves incorrectly, i.e., does not satisfy the specification. This approach is called CEGIS and is used in several synthesizers, such as SKETCH [38] and BRAHMA [19].

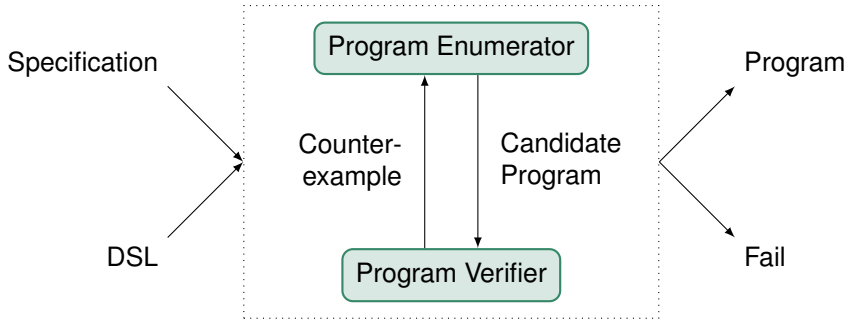


Figure 3.2: Diagram of a CEGIS loop.

In Figure 3.2 we present a diagram of the typical CEGIS loop. The synthesizer receives the specification and the DSL. In each iteration, the program enumerator generates a new program based on the DSL, such that it satisfies the specification for all current counter-examples.

For each such program, f, the verifier checks, by solving Equation 3.3, if there exists an input such that the program specification is violated. If such an input does not exist, then f is deemed correct and the synthesis ends. Otherwise, that input becomes a new counter-example which is then used to refine the search for new candidate programs. If the enumerator runs out of programs without finding a solution, the synthesis fails.

3.1.5 Program Enumeration

CEGIS relies in continuously enumerating the programs supported by a given DSL. This subsection describes two techniques for enumerating programs using an SMT solver, along with a 2-step approach.

Tree-based Enumeration

When using tree-based enumeration, the goal is to encode programs as a tree of operations, similarly to an AST. A k-tree is a tree that allows all programs of a given grammar and a given size to be represented. In a k-tree all nodes except the leaves have exactly k children. The k should be the maximum number of arguments of any given component of the grammar. In the case of the grammar presented in Example 2.4.1, k would be 3, as the component that has the largest number of arguments is `select`($table$, $cols$, $distinct$). In Figure 3.3 we present the k-tree for this grammar, with maximum depth 2, together with an assignment to the nodes of the tree corresponding to the program `join(join(recipe, amount), ingredient)`.

By encoding this k-tree representation in an SMT formula, we can generate candidate programs using an SMT solver. However, the k-tree representation has the big downside of growing exponentially as the maximum depth increases [27]. This means that generating larger programs using a k-tree quickly becomes intractable.

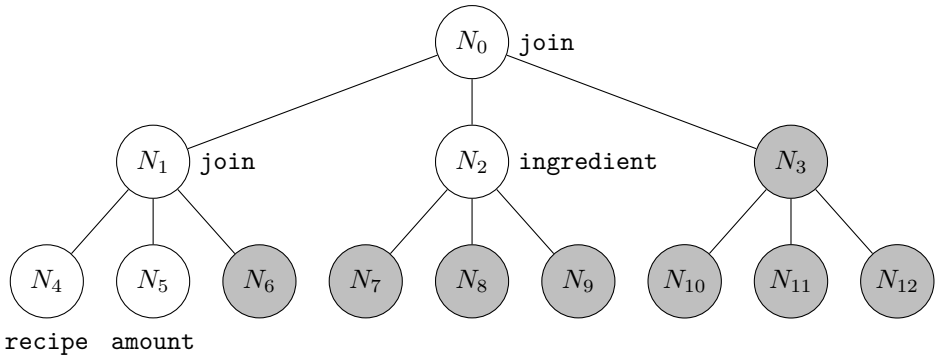


Figure 3.3: k-tree of depth 2 for the grammar in Example 2.4.1, together with the node assignments for the program `join(join(recipe, amount), ingredient)`. Grayed-out nodes are not used for representing this program.

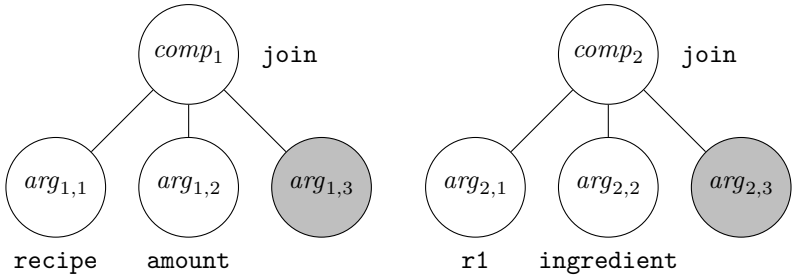


Figure 3.4: Representation of the line-based encoding for the grammar in Example 2.4.1 considering two lines. Also shown are the node assignments for the program in Example 3.1.1. Grayed-out nodes are not used for representing this program.

Line-based Enumeration

An alternative to the k-tree representation is to encode the program space using a line-based representation instead of as a tree [27], which is particularly useful for enumerating imperative-style programs.

Example 3.1.1 *Consider once again the program* `join(join(recipe, amount), ingredient)`*. A possible line-based representation of this program would be:*

```
r1 = join(recipe, amount)
r2 = join(r1, ingredient)
```

In Figure 3.4, we show a line-based representation for 2-line programs of the grammar from Example 2.4.1. For each line, i a fresh variable, $\texttt{r}i$, is created. This variable holds the result of that line and allows it to be used in following lines. In particular, for a given line p, it is possible to access the results of previous lines using the variables $\texttt{r}m$, $1 \leq m < p$. In this representation adding new lines would require just $k + 1$ extra nodes per line, whereas the original k-tree requires an exponentially increasing number of nodes as we increase the depth.

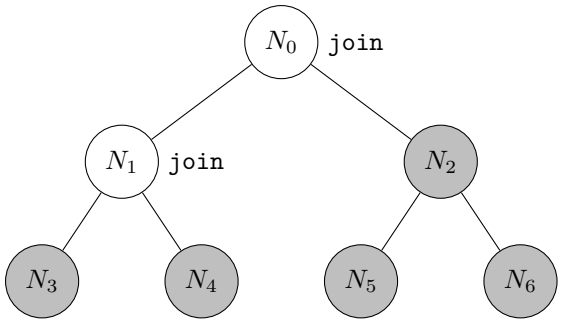


Figure 3.5: k-tree of depth 2 for the grammar in Example 3.1.2, together with the node assignments for the sketch `join(join(`$table$`,`$table$`),`$table$`)`. Grayed-out nodes are not used for representing this sketch.

2-step Enumeration

A different way to improve CEGIS-based Program Synthesis is to use 2-step enumeration: (i) sketch generation – generating sketches based on the grammar, and (ii) sketch completion – filling in the missing parts of those sketches.

Several synthesizers use this technique [10, 11, 33, 42, 44] and they differ in how each of the phases is performed. One possible method is to use SMT-based enumeration for the first step, and graph-based enumeration for the second [33].

In order to do sketch generation using SMT, the original grammar needs to be transformed so that it no longer contains any inputs or constants. In Example 3.1.2 we show an example of such a transformation. After obtaining a sketch, the algorithm relies on a simple graph search, like a Depth-first Search (DFS), to fill-out the missing nodes.

Example 3.1.2 *Applying the transformation to the grammar in Example 2.4.1, so that no inputs or constants are present, results in the following new grammar:*

$$table \rightarrow \texttt{select(}table\texttt{)} \mid \texttt{join(}table\texttt{,}\ table\texttt{)}$$

To generate the sketches we can, once again, use either the tree-based or the line-based encodings. Suppose we choose to use a k-tree. Since this transformation typically reduces the maximum number of arguments in the grammar, the k for the new grammar will be smaller. In Figure 3.5 we present the new k-tree for the sketch `join(join(`$table$`,`$table$`),`$table$`)`.

The advantages of this method are two-fold:

1. It reduces the cost of using an SMT solver by only using it to compute the semantic-rich parts of the program: the functions and operations. Meanwhile, constants and inputs can be filled in later, using a simpler, less expensive, method;

2. It naturally lends itself to parallelization by splitting the original problem into many smaller subproblems.

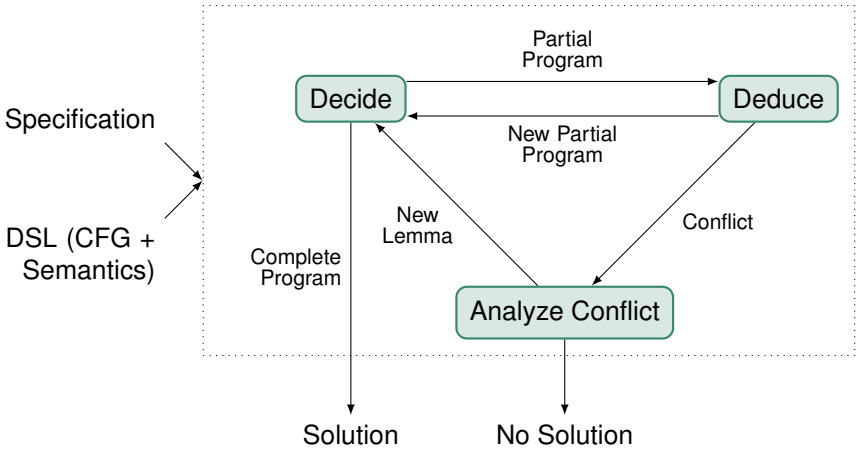


Figure 3.6: Diagram of the architecture of the NEO synthesizer [9].

3.1.6 Conflict-driven Learning

A different approach to program synthesis is to start with an empty program, and continuously try to extend it until we reach a point where the specification is satisfied [9]. This approach is called Conflict-driven Learning. Figure 3.6 presents the architecture of NEO [9], a synthesizer that implements this technique. We now briefly describe the components of this synthesizer.

Decide

The algorithm starts with an empty partial program containing just the starting symbol. At each iteration, the decide component chooses a non-terminal symbol in the partial program and a production rule to expand it. For example, if the partial program at step t is $table$, the program at step $t + 1$ might be `select(`$table$`,` $cols$`,` $distinct$`)`, by using the production rule $table \rightarrow$ `select(`$table$`,` $cols$`,` $distinct$`)`.

Deduce

For each partial program generated, NEO checks if it may be expanded into a program that satisfies the specification. To do this, it combines the semantics of the DSL with the specification and generates an SMT formula that represents the feasibility of the partial program. This procedure may have one of the following outcomes: (i) nothing new is learned and the control is returned to the decide component, (ii) some non-terminal symbols are deduced to only have one feasible completion, in which case a new partial program containing those changes is returned to the decision procedure, or (iii) the partial program is deemed unfeasible, in which case the control is passed to the analyze conflict component.

Analyze Conflict

If a conflict is found (i.e., the partial program cannot satisfy the specification), the root cause for that conflict is analyzed, which may result in a set of other partial programs also being deemed unfeasible. This information is passed to the decide component, which will use it in order to prune the program

space. It is also possible that this component determines that there is no solution, based on previous lemmas and the current conflict. In that case, the synthesis procedure terminates.

Example 3.1.3 *Consider the following set of examples:* $\mathcal{E} = \{([1,2,3],[1,2])\}$*, and that the synthesizer generated the candidate program* `map(input, ...)`*, where* `map` *is an example of a higher-order function. By analyzing this conflict, we can deduce that any program that always maintains or increases the number of elements of the input will be rejected, since it will never satisfy the example.*

3.1.7 Query Reverse Engineering

A particularly useful type of PBE problems are those pertaining to the manipulation of tabular data. Such tasks include consolidating multiple data sources or reshaping tables [10].

Query Reverse Engineering (QRE) is a special case of PBE where the domain being considered is that of tables. Given a set, $\mathcal{E}$, of input-output table examples $(\mathbf{d_i}, q_i)$, where $\mathbf{d_i}$ is a set of input tables and q_i is an output table, the QRE problem can be stated as follows:

$$\exists \mathcal{Q}. \bigwedge_{(\mathbf{d_i}, q_i) \in \mathcal{E}} \mathcal{Q}(\mathbf{d_i}) = q_i \tag{3.4}$$

where $\mathcal{Q}$ is a query.

Applications in the area of cloud computing or machine learning require their users to manipulate large amounts of data, which may not always be easy if the user is not an expert in data science. It is also possible for users to be domain experts but have no knowledge of programming. QRE allows the intended transformations to be specified using a few small examples. The generated query can then be applied to larger data sets.

One common technique for tackling QRE problems is to use CEGIS and SyGuS, restricting the search space to a DSL that is a subset of some query language, such as SQL or Python/R libraries [7, 9, 10, 33, 42].

3.2 Parallel Constraint Solving

When designing parallel algorithms for constraint solving there are two main approaches: divide and conquer and portfolio solving [26]. According to Hamadi and Sais [16] there are two main issues we need to deal with when developing such a system: (i) minimizing the idle time for each processing core, and (ii) minimizing duplicate work and communication overhead between the processing cores.

In this section we introduce both approaches, their advantages and drawbacks. We also discuss how the two issues can be addressed. We focus on research done in parallel SAT solvers as, to the best of our knowledge, there is no work on the topic of Parallel Program Synthesis. However, the techniques presented here should be adaptable to the Program Synthesis domain, since both SAT and Program Synthesis are constraint problems.

3.2.1 Divide and conquer

A divide-and-conquer algorithm is one where the problem we want to solve is *divided* into smaller sub-problems. After solving the sub-problems, called the *conquer* phase, the solution to the original problem can be derived from the different sub-problem results [6].

One of the ways of designing a parallel system for solving constraint problems is to find a way to divide the search space, such that the simpler problems are independent and can be solved in parallel. In this section, we discuss how to do that for the SAT problem.

Guiding Paths

Many of the techniques for Parallel SAT solving using divide and conquer make use of something called *guiding paths*. A guiding path is a conjunction of literals and represents a partition of the search space.

Example 3.2.1 *Consider a formula ϕ containing variable x. You can divide this problem into the two sub-problems $\phi \wedge x$ and $\phi \wedge \neg x$. The original formula is satisfiable if at least one of the sub-problems is satisfiable. In this context, (x) and $(\neg x)$ can be seen as two guiding paths.*

Load Balancing

Even though guiding paths can be used to split search spaces, there is no guarantee that the sub-problems are of similar difficulty. Therefore, blindly splitting the search space may lead to load balancing issues. In order to solve that problem, load balancing techniques need to be introduced.

Several parallel SAT solvers based on search space splitting use a master-slave architecture [26]. At any given point all slaves should be solving a sub-problem, defined by a guiding path. The master process is responsible for balancing the load between these different processes, using *dynamic work stealing* [26]. When one of the slaves becomes idle, it requests the master for some work. The master is then responsible for choosing one of the other slaves and requesting it to split its search space. This generates a new guiding path that is given as a starting point to the previously idle process. This is done until a satisfying assignment is found, or until the search space has been fully explored.

Cube and Conquer

There are two main types of state-of-the-art sequential SAT solvers: Conflict-driven Clause Learning (CDCL) solvers [24] and look-ahead solvers [17]. CDCL solvers use easy-to-compute heuristics and are very good at solving industry-type problems. Look-ahead solvers, on the other hand, use very expensive heuristics and are typically better at solving small but very hard problems. The cube and conquer approach tries to combine these two methods for solving SAT.

The cube phase uses look-ahead solvers to try to identify which partial assignments to the variables of a formula, ϕ, are most likely to produce small sub-formulas. Each of these sections is then represented by a conjunction of literals, also called a cube. As such, cubes are analogous to guiding paths. In Figure 3.7, we present a possible result of executing the cube phase of the algorithm.

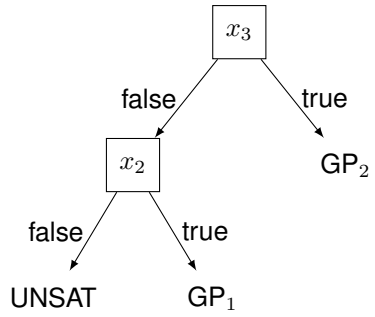


Figure 3.7: Possible partition of a formula ϕ. The two guiding paths/cubes GP$_1$ and GP$_2$ are represented, respectively, by $(\neg x_3 \wedge x_2)$ and (x_3).

Through this process, the search space is split into many smaller sub-formulas [18]. Finally, for each cube c the sub-problem $\phi \wedge c$ is given to a CDCL solver. Since these sub-formulas are smaller, CDCL solvers are expected to solve them very fast. The original formula ϕ is satisfiable if any of the sub-formulas is satisfiable, and unsatisfiable otherwise.

For some problem instances, cube and conquer represents a major improvement in the performance of SAT solvers [16]. But this approach also lends itself to parallelization on the second phase: the different sections of the search space, represented by the cubes, are independent, and as such can be solved in parallel.

3.2.2 Portfolio Solving

A different approach to designing a parallel solver is to take several complementary sequential solvers and run them in parallel. As soon as one of them returns an answer, that answer is the solution and the other processes can be stopped.

Virtual Best Solver

In SAT competitions, it is usual to present the results for the Virtual Best Solver (VBS), along with the results for the competing solvers. The VBS solver consists in choosing, for each benchmark, the best time among all the solvers. This results in a *virtual* solver that is as good as the best for each benchmark. Using parallelization, and given enough resources, we can create a portfolio solver, based on all the sequential solvers, that implements the VBS.

Diversification

It is important to note that the different sequential solvers in a portfolio must explore different parts of the search space in order to have a good speed-up. This is called diversification and can be accomplished by using different solvers or by using different initialization or configuration parameters for the same solver.

3.2.3 Clause Sharing

One of the main aspects of CDCL solvers is that they deduce new clauses as they explore the search space. Furthermore, when CDCL solvers are used in parallel, it is possible to share these learned clauses between them. This behavior is generally very helpful as it allows one to prune the search space in all processes when one of them makes a new deduction. However, sharing all learned clauses between all processes has proven to cause an exponential blow-up in the number of clauses, which would slow the solver to a crawl [26]. Furthermore, it also incurs in communication overhead penalties.

Therefore, an estimate of the usefulness a given clause is needed, in order to share only clauses that are likely to be helpful. Some examples of how to do this are to favor smaller clauses, or to consider how many literals of the clause appear on the guiding path [16].

Chapter 4

SQL Synthesis Tools

In recent years, many SQL synthesis tools have been proposed. These tools vary greatly in the types of specification they require, with some using natural language [22, 35, 43, 44], some using input-output examples [7, 9, 10, 23, 28, 39, 40, 42, 45], and others using multi-modal specifications [5]. In this chapter we focus on two tools, SCYTHE [42] and SQUARES [7, 28], which use input-output examples as their specification. We choose these tools because: (i) examples are often readily available to users and are easy to understand even with limited technical knowledge, (ii) they have expressive DSLs which cover many common queries, (iii) they are very efficient when compared with other SQL synthesis tools, (iv) SQUARES' DSL, in particular, is very easily extended, because SQUARES is built on top of the TRINITY framework [25], and (v) their source code is available online and can be integrated in our tool.

4.1 SCYTHE

SCYTHE is a Programming by Example (PBE) synthesizer for SQL queries. As such, the desired program is specified by stating what the output should be for some set of known inputs. An input-output example consists of a set of tables as input, I, and an output table, T_{out}, that results from executing the desired program over the input tables. Since tables are very rich structures, it is considered that one input-output example is sufficient. The user may also specify a set of constants, c, that must be used somewhere in the produced query.

SCYTHE follows an approach similar to the 2-step enumeration described in section 3.1.5. In the first phase SCYTHE enumerates abstract queries, that is, queries where all filter conditions are replaced with "holes". Abstract queries can be evaluated by replacing holes with `True`, and therefore never filter out any rows. The evaluation procedure follows the over-approximation rule: for any concrete query q instantiated from an abstract query $\tilde{q}$ (by filling the holes with filter predicates) the output of q is contained in the output of $\tilde{q}$. As such, by looking at the evaluation of a given abstract query it is possible to determine if there is any instantiation of $\tilde{q}$ that can possibly lead to a solution. This allows SCYTHE to discard unfeasible abstract queries before the second phase of the synthesis procedure.

In the second phase, after the possibly correct abstract queries have been enumerated, SCYTHE

tries to instantiate them until a correct concrete query is found. Two optimizations are proposed that make the second phase more efficient:

1. Equivalence classes are used to group programs so that the number of filter conditions that must be evaluated is reduced;

2. Using the over-approximation rule, we know that all rows in the output table of a query q that results from the instantiation of an abstract query $\tilde{q}$, must also be present in the output table of that abstract query. As such, scythe represents intermediate tables as a tuple $(\tilde{T}, b)$ where $\tilde{T}$ is the output of the corresponding over-approximation of the abstract query and b is a bitvector with as many bits as there are rows in $\tilde{T}$, and where bit i represents if row i of $\tilde{T}$ is present in the intermediate table. This allows SCYTHE to reduce memory requirements and execution time [42]. Even so, SCYTHE's memory usage depends heavily on the size of the input and output tables.

SCYTHE generates several possible solutions (that are all consistent with the user's specification). An heuristic is then used to rank the generated solutions, favoring those that are simpler and that use all of the constants provided as input. Finally, the top-ranked queries are returned to the user.

4.2 SQUARES

SQUARES, like SCYTHE, is a PBE synthesizer for SQL queries, and receives one input-output example as specification. Besides that input-output example, SQUARES uses some extra information about which elements should appear in the query. The full list of specification elements is:

- a list of input tables (in Comma-Separated Values (CSV) format);
- an output table (in CSV format);
- an optional list of aggregation functions (ex. `sum`, `avg`, etc...);
- an optional list of constants that must appear in the query;
- an optional list of table columns that can appear in the query.

SQUARES uses a Domain Specific Language (DSL) to specify the space of possible programs. This DSL is inspired by the operations available in the popular R data-manipulation library, `dplyr`, from `tidyverse`[1]. In Figure 4.1 we present SQUARES' DSL. Productions rules such as those corresponding to $cols$ or $filterCondition$ depend on the program being synthesized, and as such are not presented. As explained in section 2.4 a DSL used for enumeration must be translated into some programming language in order to evaluate candidate programs and return the final answer to the user. In SQUARES, the DSL in Figure 4.1 is translated into R for evaluation. However, SQUARES is also a SQL synthesizer and as such an automated translation layer is used to convert the generated R program into a SQL query when presenting the final answer to the user.

[1] `https://www.tidyverse.org/`

$$
\begin{aligned}
table \quad & \rightarrow input \mid \texttt{inner\_join}(table, table) \mid \texttt{inner\_join3}(table, table, table) \\
& \mid \texttt{inner\_join4}(table, table, table, table) \\
& \mid \texttt{filter}(table, filterCondition) \\
& \mid \texttt{filters}(table, filterCondition, filterCondition, op) \\
& \mid \texttt{summariseGrouped}(table, summariseCondition, cols) \\
& \mid \texttt{anti\_join}(table, table) \mid \texttt{left\_join}(table, table) \\
& \mid \texttt{bind\_rows}(table, table) \mid \texttt{intersect}(table, table) \\
tableSelect \quad & \rightarrow \texttt{select}(table, selectCols, distinct) \\
op \quad & \rightarrow \texttt{or} \mid \texttt{and} \\
distinct \quad & \rightarrow \texttt{true} \mid \texttt{false}
\end{aligned}
$$

Figure 4.1: DSL used by the SQUARES synthesizer [7].

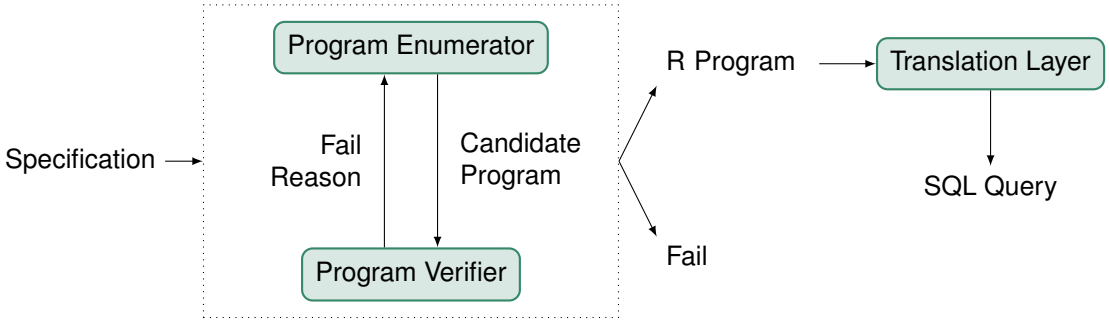


Figure 4.2: Diagram of SQUARES architecture.

Figure 4.2 shows that the synthesizer itself is composed of the Program Enumerator and the Program Verifier. It receives a specification from the user and, if the synthesis is successful, returns an R program that satisfies the specification. This R program is then automatically translated into an equivalent SQL query.

4.2.1 Program Enumeration

At the core of SQUARES is a Program Enumerator. The purpose of the Program Enumerator is to continuously generate new candidate programs based on the specification. Programs are enumerated with the help of an SMT solver, using the line-based representation introduced by Orvalho et al. [27] and described in section 3.1.5. Programs are enumerated in increasing number of lines of code. Here, we explain how to encode this representation in an SMT formula and use it to enumerate valid programs.

Variables

Let us consider an encoding for a program with n lines and where the maximum number of arguments for any given component is k. There are three main sets of variables to consider:

- $C = \{comp_i : 1 \leq i \leq n\}$, where each integer variable $comp_i$ denotes the component used in line i;

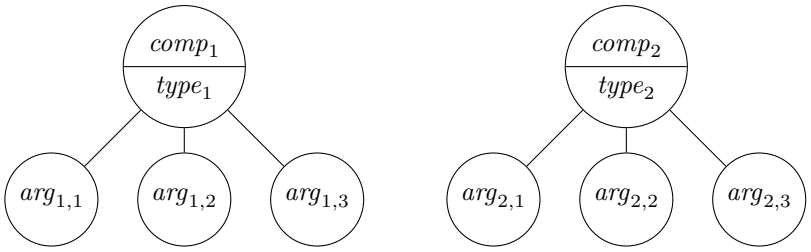


Figure 4.3: Line-based representation for the grammar in Example 2.4.1 considering two lines. Each tree represents a different line of the program.

- $T = \{type_i : 1 \leq i \leq n\}$, where each integer variable $type_i$ denotes the return type of line i;
- $A = \{arg_{ij} : 1 \leq i \leq n, 1 \leq j \leq k\}$, where each integer variable arg_{ij} denotes the symbol corresponding to argument j of line i.

Figure 4.3 shows an illustration of how the aforementioned variables relate to the line based representation for the grammar from Example 2.4.1 ($k = 3$).

Consider also the following definitions:

- A set, $\mathcal{I}$, containing all the productions that correspond to program inputs;
- A set, $\mathcal{C}$, containing all components of the current DSL;
- A set, $\mathcal{A}$, containing all productions that are valid arguments for some DSL component;
- A set, $\mathcal{R} = \{ret_i : 1 \leq i \leq n\}$, containing pseudo-productions that represent the return of each line in the program;
- A function, $\mathbf{id} : P \mapsto \mathbb{N}_0$, that maps DSL productions to unique integer identifiers;
- A function, $\mathbf{type} : P \mapsto V$, that maps DSL productions to their corresponding left-hand side;
- A function, $\mathbf{type} : \mathcal{C} \times \mathbb{N} \mapsto V$, that given a DSL component and a number, j, returns the type of the j-th argument of that component. If there is no argument with position j, then a special type, ϵ, is returned;
- A function, $\mathbf{arity} : \mathcal{C} \mapsto \mathbb{N}_0$, that maps DSL components to the number of arguments they receive;
- A function, $\mathbf{tid} : V \mapsto \mathbb{N}_0$, that maps DSL non-terminal symbols (in particular, those returned by $\mathbf{type}$) to unique integer identifiers;
- Let $type_{output}$ be the expected return type of the complete program, i.e., the starting symbol of the DSL.

Constraints

- Each line should be assigned a valid DSL component:

$$\bigwedge_{i=1}^{n} \bigvee_{c \in \mathcal{C}} comp_i = \mathbf{id}(c) \tag{4.1}$$

- The component used in a given line determines the return type of that line:

$$\bigwedge_{i=1}^{n} \bigwedge_{c \in \mathcal{C}} (comp_i = \mathbf{id}(c)) \implies (type_i = \mathbf{tid}(\mathbf{type}(c))) \tag{4.2}$$

- For each line, the arguments should be either valid program symbols, or the return of a previous line:

$$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{k} \bigvee_{s \in \mathcal{A} \cup \{ret_r : 1 \leq r < i\}} arg_{ij} = \mathbf{id}(s) \tag{4.3}$$

- For each line, the types of the arguments should match with the expected types for the selected component:

$$\bigwedge_{i=1}^{n} \bigwedge_{c \in \mathcal{C}} \bigwedge_{j=1}^{\mathbf{arity}(c)} \bigwedge_{s \in \{a \in \mathcal{A} : \mathbf{type}(a) \neq \mathbf{type}(c,j)\}} (comp_i = \mathbf{id}(c)) \implies \big(arg_{ij} \neq \mathbf{id}(s)\big) \tag{4.4}$$

- When a return symbol is used as an argument, its type must match the expected argument type:

$$\bigwedge_{i=1}^{n} \bigwedge_{c \in \mathcal{C}} \bigwedge_{j=1}^{\mathbf{arity}(c)} \bigwedge_{r=1}^{i-1} \big(comp_i = \mathbf{id}(c) \wedge arg_{ij} = \mathbf{id}(ret_r)\big) \implies (type_r = \mathbf{tid}(\mathbf{type}(c, j))) \tag{4.5}$$

- Unused arguments must be assigned the special symbol ϵ:

$$\bigwedge_{i=1}^{n} \bigwedge_{c \in \mathcal{C}} \bigwedge_{j=\mathbf{arity}(c)+1}^{k} (comp_i = \mathbf{id}(c)) \implies \big(arg_{ij} = \mathbf{id}(\epsilon)\big) \tag{4.6}$$

- The type of the last line must be the same as the expected return type of the program:

$$\bigvee_{s \in \{c \in \mathcal{C} : \mathbf{type}(c) = type_{output}\}} comp_n = \mathbf{id}(s) \tag{4.7}$$

- Regarding the inputs, it is required that all of them appear at least once in the program:

$$\bigwedge_{input \in \mathcal{I}} \bigvee_{i=1}^{n} \bigvee_{j=1}^{k} arg_{ij} = \mathbf{id}(input) \tag{4.8}$$

Predicates

In addition to these constraints there are 4 predicates that can be used to further restrict the generated programs.

The predicate `is_not_parent(`c_1`,` c_2`)` means that the result of a line with component c_2 can not be used as argument to a line with component c_1. This predicate is encoded by the following constraint:

$$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{k} \bigwedge_{r=1}^{i-1} \left(comp_i = \mathbf{id}(c_1) \wedge arg_{aj} = \mathbf{id}(ret_r)\right) \implies \left(comp_r \neq \mathbf{id}(c_2)\right) \tag{4.9}$$

The predicate `happens_before`(a_1, a_2) means that the symbol a_1 can only appear in a given line if symbol a_2 appears in a previous line. This predicate is encoded by the following constraint:

$$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{k} \left(\left(arg_{ij} = \mathbf{id}(a_1)\right) \implies \left(\bigvee_{r=1}^{i-1} \bigvee_{m=1}^{k} arg_{rm} = \mathbf{id}(a_2) \right) \right) \tag{4.10}$$

The predicate `constant_occurs`$(c_1, \ldots, c_x)$ can be called with any number of symbols and makes it so that at least one of them has to appear in the program. This predicate is encoded by the following constraint:

$$\bigvee_{l=1}^{x} \bigvee_{i=1}^{n} \bigvee_{j=1}^{k} arg_{ij} = \mathbf{id}(c_l) \tag{4.11}$$

The predicate `distinct_inputs`(c) states that for each line assigned component c, the arguments of that line should be distinct from one another. This predicate is encoded by the following constraint:

$$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{k} \bigwedge_{1 \leq l \leq k \wedge l \neq j} arg_{ij} \neq arg_{il} \tag{4.12}$$

4.2.2 Program Verification and Translation

In order to evaluate the candidate programs and check if they satisfy the user's specification, SQUARES translates them into R. Consider the following program produced using SQUARES' DSL:

```
r1 = filters(input0, age == 46, age == 50, or)
out = select(r1, country, false)
```

This program is translated into the following R program for evaluation:

```
df1 <- input0 %>% ungroup() %>% filter(age == 46 | age == 50)
out <- df1 %>% ungroup() %>% select(country)
```

Next, the program is executed using the input example that the user provided. The output of the program is then compared with the expected output. The comparison treats tables as a multi-set of rows, meaning that row order is ignored. If the tables match, a solution to the problem has been found.

Finally, before presenting the program to the user, it must be converted into SQL. To do this the `dbplyr` library is used. This library allows one to use regular databases as a back-end for `dplyr` operations and extract the corresponding SQL queries. The previous R program would be translated into the following SQL query:

```
SELECT `country`
FROM `input0`
WHERE (`age` = 46.0 OR `age` = 50.0)
```

Chapter 5

CUBES: Sequential Synthesis

This work uses SQUARES as a starting point for creating a parallel SQL synthesizer. SQUARES was chosen for this task because it allows for easy extension of the DSL and modification of the enumeration and decision procedures.

In this chapter we describe the changes made to SQUARES that are not directly related to multi-processing. In particular, how the range of supported programs was extended and how new forms of pruning were introduced in order to improve synthesis performance. From now on, we will refer to the improved version of SQUARES as CUBES-SEQ.

5.1 Extending the Domain Specific Language

In order to support a wider range of programs, SQUARES' DSL was modified to be more expressive. In this section, we will describe those changes. The new DSL is presented in Figure 5.1. Table 5.1 shows the mapping from CUBES' DSL to the corresponding R functions used for evaluation.

5.1.1 Changes to DSL Components

Select The select component was removed from the DSL and introduced as a post-processing step. This allows for two important changes:

- The column names of the output table no longer have to match the input table names, nor contain knowledge about the the desired program. For instance, in SQUARES a column in the output table that was the result of computing the `max` of column `colA` would have to be called `maxcolA` or the synthesizer would not be able to synthesize the program.

- During this post-processing step it is also checked if any columns should be sorted, according to the expected output. If that is the case, then an `arrange` / `ORDER BY` instruction (in R/SQL, respectively) is also added to the synthesized program.

$$
\begin{aligned}
table \rightarrow\ & input \mid \texttt{natural\_join}(table, table) \mid \texttt{natural\_join3}(table, table, table) \\
& \mid \texttt{natural\_join4}(table, table, table, table) \mid \texttt{left\_join}(table, table) \\
& \mid \textbf{\texttt{inner\_join}}(table, table, joinCondition) \\
& \mid \textbf{\texttt{cross\_join}}(table, table, crossJoinCondition) \\
& \mid \texttt{filter}(table, filterCondition) \\
& \mid \texttt{summarise}(table, summariseCondition, cols) \\
& \mid \textbf{\texttt{mutate}}(table, summariseCondition) \\
& \mid \texttt{union}(table, table) \mid \texttt{intersect}(table, table, col) \\
& \mid \texttt{anti\_join}(table, table, cols) \mid \textbf{\texttt{semi\_join}}(table, table)
\end{aligned}
$$

Figure 5.1: DSL used by the CUBES synthesizer. New components are highlighted in `bold`.

Natural join The three inner join components have been renamed to `natural_join`, as they did not allow to join columns with different names. Furthermore, these components were changed so that the natural join between two tables with no columns in common returns the Cartesian product between those tables (the previous behavior was to produce an evaluation error, which effectively disallowed such joins). This change means that for a given set of arguments, all permutations of those arguments now result in the same table. As such, for a given set of arguments, we only allow one specific order for those arguments, removing equivalent programs from the search space.

Inner join A true `inner_join` component was added that allows to join tables using pairs of columns with different names (consider the following very common use case: joining two tables, `student` and `class`, by the respective columns `id` and `student_id`).

Cross join A `cross_join` component was also added that allows to join two tables with non equality conditions. This component is implemented by computing the full Cartesian product of the two tables and then selecting a subset of the rows, according to the specified condition. It is equivalent to the `JOIN ON` operation in SQL.

Filter The `filters` component, which was used to combine several filter conditions, was removed. As a replacement, $filterCondition$ now contains compound conditions such as `name == 'John' & age > 22`. This change allows CUBES-SEQ to exclude many redundant combinations that previously had to be individually evaluated. Consider the following examples:

- `a > 2 & a < 2` is an unsatisfiable condition, and thus does not need to be considered;
- `a > 2 | a < 2` is equivalent to `a != 2`, which makes it redundant.

Mutate The `mutate` component was added to the DSL. This component is similar to the `summarise` component, in that it applies a function to some column, but differs in that it does not group rows, and also does not reduce the number of columns nor rows in its output. Common situations where a `mutate`

Table 5.1: Mapping from CUBES' DSL to the corresponding implementation in R.

DSL Representation	R code
natural_join($t1$,$t2$)	`if (length(intersect(colnames(t1), colnames(t2))) > 0) { inner_join(t1, t2) } else { full_join(t1, t2, by=character()) }`
natural_join3($t1$,$t2$,$t3$)	Implemented as natural_join(natural_join($t1$,$t2$),$t3$)
natural_join4($t1$,$t2$,$t3$,$t4$)	Implemented as natural_join(natural_join3($t1$,$t2$,$t3$),$t4$)
left_join($t1$,$t2$)	`left_join(t1, t2)`
inner_join($t1$,$t2$,$joinCondition$)	`inner_join(t1, t2, by=c(joinCondition), suffix = c(", '.other'))`
cross_join($t1$,$t2$,$crossJoinCondition$)	`full_join(t1, t2, by=character(), suffix = c("", ".other")) %>% filter(crossJoinCondition)`
filter(t,$filterCondition$)	`filter(t, filterCondition)`
summarise(t,$summariseCondition$,$cols$)	`t %>% group_by(cols) %>% summarise(summariseCondition) %>% ungroup()`
mutate(t,$summariseCondition$)	`mutate(t, summariseCondition)`
union($t1$,$t2$)	`bind_rows(t1, t2)`
intersect($t1$,$t2$,col)	`intersect(select(t1, col), select(t2, col))`
anti_join($t1$,$t2$,$cols$)	`anti_join(t1, t2, by=c(cols)))`
semi_join($t1$,$t2$)	`semi_join(t1, t2)`

is required are cumulative sums, and the `lead`/`lag` functions which offset the values of a column by one row.

Union The `bind_rows` component has been renamed to the more descriptive name `union`.

Intersect The `intersect` component now takes a column as an argument and returns only the intersection of that column, as in practice that is more useful than intersecting full tables.

Anti join The `anti_join` component also takes a (possibly empty) list of columns as an argument for the same reason as `intersect`; if the list is empty, the original behavior is conserved.

Semi join The sibling component to `anti_join`, `semi_join`, has been added to the DSL.

5.1.2 New Aggregation Functions

Several new aggregation functions are now supported: `n_distinct`, `str_count`, `cumsum`, `pmin`, `pmax`, `mode`, `lead`, `lag`, `median`, `rank` and `row_number`. Some aliases are also supported, in order to facilitate usage by users familiar with SQL: `count` is an alias for both `n` and `n_distinct` (activates both options), and `avg` is an alias for `mean`.

5.1.3 Type Inference

The type inference mechanism has been overhauled, resulting in dates and times now being supported. By default dates are parsed using the ISO 8601 format, but this can be overridden by specifying the desired date format.

5.2 Quantifier-Free Finite Domain Theory

When using constraint solvers, a lower-level encoding is usually more efficient. Since all the variables used in CUBES-SEQ's encoding are either bounded integers or Boolean variables, a possible way to improve performance is to use bit-blasting. Bit-blasting means converting all variables in the SMT formula to Boolean variables and all constraints to CNF. To do this, the integer variables are first converted to bit vectors, and then the bit-vectors are converted to sets of Boolean variables. The constraints are updated to reflect these changes and then transformed into CNF. The result is a propositional logic formula in CNF that can be solved using an off-the-shelf SAT solver.

The SMT solver used by CUBES-SEQ, Z3 [8], implements a theory that performs all these steps automatically, including using an internal SAT solver to solve the resulting formula. This theory is called Quantifier-Free Finite Domain (QF_FD).

5.3 Deducing Invalid Programs

One way to improve a program synthesizer is to reduce the number of incorrect programs that must be tested before finding a solution. In the case of SQUARES, a common example are programs where at some point a column that does not exist in the current context is referenced. Consider the following program, which uses the tables from Example 2.3.1 and the DSL from Figure 5.1:

```
df1 = filter(recipe, ing_name == 'Eggs')
```

In this program, we are taking the `recipe` table and trying to filter its rows by selecting only the ones where column `ing_name` is equal to `'Eggs'`. After a closer look to the `recipe` table it is clear that this program makes no sense, as there is no `ing_name` column in this table. SQUARES enumerates such programs because it uses a DSL that is defined at initialization time, containing all possible conditions, which is then given to the SMT solver for it to generate candidate programs. Without some extra guidance, there is no way for the SMT solver to only generate valid candidates.

We introduce a new form of pruning that eliminates these invalid programs. All component arguments are annotated with a pair of sets of columns. These annotations are then used to further constrain the set of programs that can be return by the program enumerator. In Example 5.3.1 we show how these annotations can be used to force all `filter` lines to always be valid. Note that the second annotation is only needed for some argument types, in order to record extra information. One such case is presented in Example 5.3.2.

Example 5.3.1 *Consider again the previous program, which contains only one line:* $filter(recipe, ing\_name == `Eggs')$. *That line takes two arguments:* $recipe$ *and* $ing\_name == `Eggs'$.

We annotate all arguments of type table with the columns they contain, so in this case $recipe$ *would be annotated with* $\{recipe\_id, recipe\_name\}$. *Furthermore, we annotate filter condition arguments with the columns they require to be present in order do produce a valid program. In this case* $ing\_name == 'Eggs'$ *would be annotated with* $\{ing\_name\}$. *Finally, we encode that all filter operations must be such that all the columns in the annotation of the second argument appear in the annotation of the first argument in order to be valid.*

The presented program violates these rules, and thus is surely incorrect.

In order to propagate the column information along the several lines of the program, each line is also annotated with the set of columns available in the output table of that line. This information can then be used like that of any other argument of type $table$. By implementing these kind of rules for all the components we can greatly reduce the number of enumerated programs that are invalid due to column name problems. As a result, the overall performance of CUBES-SEQ is improved.

In Figure 5.2 we show the inference rules for all the components of our DSL. Using these rules, we can infer from the arguments of a given operation what columns would be present in the output table if the line were executed. By extension, we can also determine invalid lines because no rule will be applicable to them. The contents of each annotation are described in Figure 5.3. Finally, the introduction of this new type of pruning makes the `happens_before` predicate described in section 4.2.1 redundant. Hence, this predicate was removed.

Example 5.3.2 *Consider the following summariseCondition:* $meanAge = mean(Age)$. *The first annotation of a summariseCondition corresponds to the columns that are "used", that is, the columns that must be present in order for the condition to be applicable. In this case the first annotation would be* $\{Age\}$. *The second annotation corresponds to the columns that are generated by the summariseCondition, in this case:* $\{meanAge\}$.

When this condition is used in a $mutate$ *operation, rule* MUTATE *from Figure 5.2 states that if all of the required columns (first annotation) are present in the table argument, then we can conclude that the output table will be comprised of all columns that were already present in the input table, along with the generated columns (second annotation).*

The rules in Figure 5.2 are implemented directly as SMT constraints, which means the corresponding invalid programs are never generated. To do this, the Bitvector Theory (BV) is used. The first step in

$$\frac{}{output' = table'_1 \cup table'_2}\ \textsc{NaturalJoin} \qquad \frac{filterCondition' \subseteq table'}{output' = table'}\ \textsc{Filter}$$

$$\frac{}{output' = table'_1 \cup table'_2 \cup table'_3}\ \textsc{NaturalJoin3}$$

$$\frac{}{output' = table'_1 \cup table'_2 \cup table'_3 \cup table'_4}\ \textsc{NaturalJoin4}$$

$$\frac{joinCondition' \subseteq table'_1 \quad joinCondition'' \subseteq table'_2}{output' = table'_1 \cup table'_2}\ \textsc{InnerJoin}$$

$$\frac{cols' \subseteq table'_1 \quad cols' \subseteq table'_2 \quad (cols' \neq \emptyset \vee table'_1 \cap table'_2 \neq \emptyset)}{output' = table'_1}\ \textsc{AntiJoin}$$

$$\frac{table'_1 \cap table'_2 \neq \emptyset}{output' = table'_1 \cup table'_2}\ \textsc{LeftJoin} \qquad \frac{}{output' = table'_1 \cup table'_2}\ \textsc{Union}$$

$$\frac{col' \subseteq table'_1 \quad col' \subseteq table'_2}{output' = col'}\ \textsc{Intersect} \qquad \frac{table'_1 \cap table'_2 \neq \emptyset}{output' = table'_1}\ \textsc{SemiJoin}$$

$$\frac{crossJoinCondition' \subseteq table'_1 \quad crossJoinCondition'' \subseteq (table'_1 \cap table'_2)}{output' = table'_1 \cup table'_2}\ \textsc{CrossJoin}$$

$$\frac{summariseCondition' \subseteq table' \quad cols' \subseteq table' \quad (cols' \cap summariseCondition'') = \emptyset}{output' = summariseCondition'' \cup cols'}\ \textsc{Summarise}$$

$$\frac{summariseCondition' \subseteq table'}{output' = table' \cup summariseCondition''}\ \textsc{Mutate}$$

Figure 5.2: Inference rules used to determine valid programs. A' denotes the first annotation of element A, while A'' denotes the second annotation. Where not mentioned, it is assumed that the second annotation is $= \emptyset$.

this process is to determine the list of all possible column names in the program. This list is composed of all columns that appear in input tables, along with all columns generated by a `summarise` or `mutate` operations. Each set of columns in the encoding is then represented by a bitvector with length equal to the size of the full list of tables. For a given set and corresponding bitvector, bit i of the bitvector is set to 1 if and only if the i-th element of the list of all columns is present in the set.

Along with the variables already defined in section 4.2.1 we introduce a new set, $\mathcal{BV}$, comprised of the following variables:

- For each argument variable, arg_{ij}, two new bitvector variables are introduced: $arg_{ij}bv_1$ and $arg_{ij}bv_2$, corresponding to the first and second annotations. The meaning of these annotations depends on the specific argument and is shown in Figure 5.3;

- Each component variable, $comp_i$, gets a bitvector variable, $comp_i bv$, that corresponds to the columns present in the table that results from that line.

Figure 5.4 shows a representation of all variables used in CUBES' SMT encoding.

$table'$: columns present in the table
col' : column required in the table
$cols'$: columns required in the table
$filterCondition'$: columns used in the filter condition
$joinCondition'$: columns required in the first table
$joinCondition''$: columns required in the second table
$crossJoinCondition'$: columns required in the first table
$crossJoinCondition''$: columns required in both tables
$summariseCondition'$: columns used in the summarise condition
$summariseCondition''$: columns generated by the summarise condition

Figure 5.3: Description of the semantics of each annotation. A' denotes the first annotation of element A, while A'' denotes the second annotation.

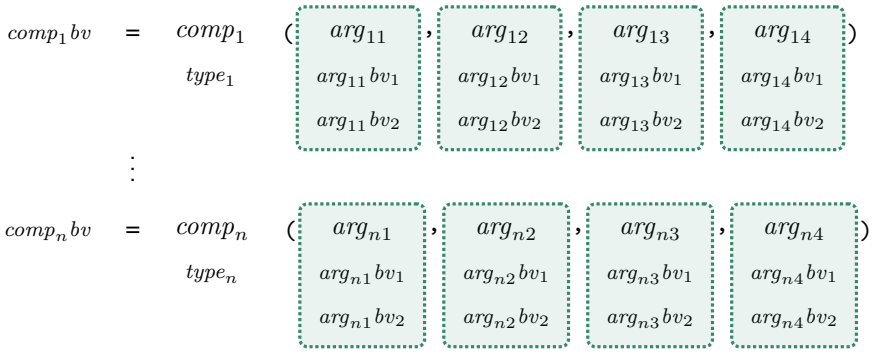


Figure 5.4: Representation of the variables used in the SMT encoding.

Basic constraints

Consider the following definitions:

- A function, $\mathbf{ann1} : \mathcal{A}\backslash\mathcal{R} \mapsto \{0,1\}^c$, that maps component arguments to their respective first annotation. The number c corresponds to the total number of unique columns;
- A function, $\mathbf{ann2} : \mathcal{A}\backslash\mathcal{R} \mapsto \{0,1\}^c$, that maps component arguments to their respective second annotation, if it exists. If it does not exist, it maps to the bitvector which is all zeros, $\mathbf{0}$. The number c corresponds to the total number of unique columns.

Example 5.3.3 *Consider again the summariseCondition* `meanAge = mean(Age)`. *Consider also that for this given program the full list of columns is* `[Name, Age, City, meanAge]`. *Then* $\mathbf{ann1}($`meanAge = mean(Age)`$) = 0100$ *and* $\mathbf{ann2}($`meanAge = mean(Age)`$) = 0001$.

The following constraints encode the basic structure of the annotations and their propagation along the program, and build upon SQUARES' SMT encoding:

- For all arguments, the values in the two bitvector variables, $arg_{ij}bv_1$ and $arg_{ij}bv_2$, will correspond to the respective annotations:

$$\bigwedge_{i=1}^{n} \bigwedge_{c \in \mathcal{C}} \bigwedge_{j=1}^{\mathbf{arity}(c)} \bigwedge_{a \in \mathcal{A} \setminus \mathcal{R}} comp_i = \mathbf{id}(c) \wedge arg_{ij} = \mathbf{id}(a) \implies \big(arg_{ij} bv_1 = \mathbf{ann1}(a) \wedge arg_{ij} bv_2 = \mathbf{ann2}(a)\big) \tag{5.1}$$

- If an argument is not used for a given component, then the bitvector variables for that argument must be assigned the $\mathbf{0}$ bitvector:

$$\bigwedge_{i=1}^{n} \bigwedge_{c \in \mathcal{C}} \bigwedge_{j=\mathbf{arity}(c)+1}^{k} comp_i = \mathbf{id}(c) \implies \big(arg_{ij} bv_1 = \mathbf{0} \wedge arg_{ij} bv_2 = \mathbf{0}\big) \tag{5.2}$$

- Finally, when argument j of line i is the return value of a previous line r, then the first bitvector, $arg_{ij} bv_1$, is assigned to the columns of the table ret_r which are contained in variable $comp_r bv$. The second bitvector variable is assigned $\mathbf{0}$:

$$\bigwedge_{i=1}^{n} \bigwedge_{c \in \mathcal{C}} \bigwedge_{j=1}^{\mathbf{arity}(c)} \bigwedge_{r=1}^{i-1} arg_{ij} = \mathbf{id}(ret_r) \implies \big(arg_{ij} bv_1 = comp_r bv \wedge arg_{ij} bv_2 = \mathbf{0}\big) \tag{5.3}$$

Column set computation

To propagate the available columns throughout the program, the inference rules showed in Figure 5.2 are used. Let us consider the FILTER rule as an example.

Since we only want to generate programs that are valid (i.e., never violate the inference rules) and since there is only one possible rule per component, we assert that when a `filter` operation occurs both the premises and the conclusion must be true. The following constraint encodes the FILTER rule for all lines (note that $\&$ represents a bit-wise and). The dotted box ⬚ encodes the premises, while the dashed box ⬚ represents the conclusion.

$$\bigwedge_{i=1}^{n} comp_i = \mathbf{id}(\texttt{filter}) \implies \Big(\boxed{(arg_{i1} bv_1 \ \& \ arg_{i2} bv_1) = arg_{i2} bv_1} \wedge \boxed{comp_i bv = arg_{i1} bv_1} \Big) \tag{5.4}$$

5.4 Learning from Incorrect Programs

A different way to prune the number of tested programs, is to try and extract some information from failed attempts, and then use that information in order to prune the search space. One of the ways this can be accomplished is by looking at the number of rows of the final table. Consider the following program:

```
r1 = natural_join(input1, input2)
r2 = filter(r1, colA != 20)
```

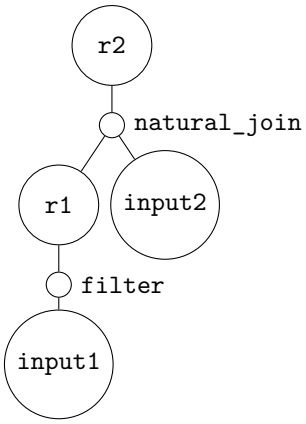


Figure 5.5: Bottom-up representation of a program (from the output to the inputs).

Suppose that the expected output table (provided by the user) has k rows. Moreover, consider also that this program produces an output table with p rows, such that $p < k$. Therefore, we can infer that if we replace the filter condition in the last line with one that is even more restrictive, then the new program will also be incorrect (as it will have at most the same number of rows, p). For instance, if we replace the filter condition `col != 20` with `colA > 20` or `colA < 20` the resulting table would still have at most p rows. Likewise, if the number of rows of the resulting table is larger than the number of rows of the expected output table, all conditions that are less restrictive can be blocked (as they will produce output tables that have at least as many rows).

The intuition that the conditions in the last operation can be used to determine if the correct program needs stricter or looser restrictions naturally breaks down when a condition is used in the middle of the program, like in the following case:

```
r1 = filter(input1, colA != 20)
r2 = natural_join(r1, input2)
```

However, some components (notably `natural_join`, `natural_join3`, `natural_join4`, `mutate`, `filter`, `summarise`, `inner_join`, `left_join` and `union`) are monotonic regarding the number of rows in their output. That is, if we remove/add some rows from/to one of their inputs, then their output will have at most/least the same number of rows as before, respectively. For example, in the previous program the `filter` line can still be pruned because its result is only used by monotonic components. Figure 5.5 shows a representation of the example program, starting at the output and walking up to the inputs. Using this representation, any operation node than can be reached passing through only monotonic components, as is the case with `filter`, can still be used for pruning.

Chapter 6

CUBES: Parallel Synthesis

In this chapter we discuss how techniques used in parallel constraint solvers can be adapted in order to create a Parallel Program Synthesizer. In the first section, we introduce CUBES-PORT, the portfolio mode of CUBES, which takes advantage of a portfolio of synthesizers in order to produce faster results. Next, in section 6.2, we introduce CUBES-DC, the divide-and-conquer mode of CUBES, that divides the synthesis problem into several sub-problems and then solves those sub-problems in parallel.

6.1 Portfolio

In the last decade, new parallel algorithms for combinatorial problems have been devised [18] that try to explore the current multi-core processor architectures. In particular, the portfolio approach has been successfully applied to several decision problems [18]. In this technique, as soon as one of the processes finds a solution, the search ends and there is no need to completely explore the rest of the search space. Therefore, the main goal of a portfolio is to diversify the exploration of the search space by making each thread explore the same search space in different ways.

The Query Synthesis problem can be seen as a decision problem where one wants to find a program that satisfies the user's specification. Therefore, it is possible to devise a portfolio that diversifies the search using different tactics such as: (i) use the same synthesizer with different configurations, or (ii) selecting a set of synthesizers that use different search techniques.

Internally, CUBES uses an SMT formula to enumerate candidate programs. Hence, one can devise a portfolio by providing the same SMT formula to each process, but using different configurations of the Z3 SMT solver [8] in order to diversify the search. A complementary option is to change the active techniques from CUBES in each process, thus changing the learned constraints in the SMT formula and the subsequent search. Another complementary alternative is to use different synthesizers in parallel. Each synthesizer such as SQUARES or SCYTHE uses different techniques, thus increasing the diversity in the exploration of the search space.

Figure 6.1 shows the portfolio presets available in CUBES-PORT. These presets are created by combining different options available in CUBES-SEQ. One of the biggest disadvantages of portfolio

Process	Z3 Phase Selec.	Prog. Deduction
1	Caching	True
2	Caching	False
3	Random	True
4	Random	False

(a) Preset for 4 processes (CUBES-PORT4).

Process	QF_FD	Z3 Phase Selec.	Prog. Deduction
1	True	Caching	True
2	True	Caching	False
3	True	Random	True
4	True	Random	False
5	False	Caching (cons.)	True
6	False	Caching (cons.)	False
7	False	Random	True
8	False	Random	False

(b) Preset for 8 processes (CUBES-PORT8).

Process	Learning from Prog.	QF_FD	Z3 Phase Selec.	Prog. Deduction
1	False	True	Caching	True
2	False	True	Caching	False
3	False	True	Random	True
4	False	True	Random	False
5	False	False	Caching (cons.)	True
6	False	False	Caching (cons.)	False
7	False	False	Random	True
8	False	False	Random	False
9	True	True	Caching	True
10	True	True	Caching	False
11	True	True	Random	True
12	True	True	Random	False
13	True	False	Caching (cons.)	True
14	True	False	Caching (cons.)	False
15	True	False	Random	True
16	True	False	Random	False

(c) Preset for 16 processes (CUBES-PORT16).

Figure 6.1: The three presets available in CUBES-PORT.

solving is that in order to increase the number of processes used, one needs to find new, distinct, ways to search the program space. Besides modifying options from CUBES-SEQ, it would also be possible to combine CUBES-PORT with other SQL synthesizers. SCYTHE, in particular, is a good choice because it uses a very similar form of specification, and thus, it is possible to convert a problem given to CUBES to a problem for SCYTHE automatically.

6.2 Divide and conquer

When using divide and conquer to solve a search problem in parallel, the strategy is to split the problem into smaller sub-problems that can be solved by each of the processes. Instead of diversifying the search (as in the portfolio approach), each process in divide and conquer focuses the search in a particular area of the search space.

Inspired by previous work in solving Propositional Satisfiability formulas [41], we present a strategy to split the Program Synthesis search space in many sub-problems that should be easy to solve. The overall architecture is illustrated in Figure 6.2. In our context, each sub-problem is represented by a cube: a sequence of operations from the DSL, such that the arguments for the operations are still to be determined. Consider the following cube as an example: `[filter, natural_join]`, which represents

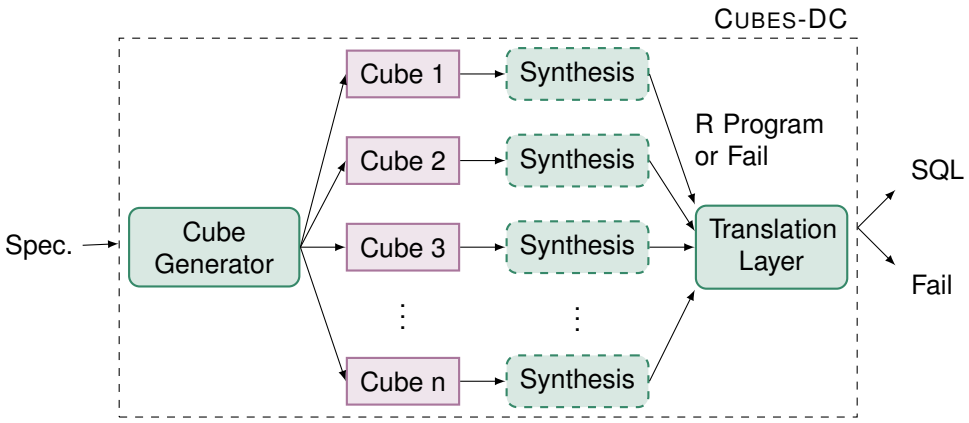


Figure 6.2: Diagram of CUBES' architecture when using divide and conquer.

the partition of the search space composed by programs with two lines, where the first operation is a `filter` and the second operation is a `natural_join`. Each process receives a specific cube to be filled in and determines if a solution can be reached for that particular cube. If the cube cannot be completed such that it satisfies the input-output examples, then the cube is deemed unsatisfiable and the process requests a new cube to explore. Observe that each cube corresponds to a particular sequence of operations, and as such, there is no intersection in the search space of each process.

This approach is very similar to using guiding paths in Parallel SAT, as described in section 3.2. A guiding path is an assignment to a subset of the variables of the formula that defines a partition of the search space. The task is then to generate several (disjoint) guiding-paths that can be solved in parallel. It is also similar to the 2-step enumeration approach described in section 3.1.5. In this case, the first step would consist in generating the cubes using a graph-based algorithm, and the second step would consist in filling in each cube using SMT-based enumeration.

Note that the effectiveness of the search depends heavily on the strategy for cube generation. Next, we describe different strategies explored in CUBES.

6.2.1 Static Cube Generation

In static cube generation, cubes are constructed using a static heuristic. However, the sequence of operations to be tried first is not purely a predetermined order to be followed. Instead, the heuristic, presented in Figure 6.3, selects the operation to be executed next in a given sequence depending on the already selected operations. For instance, if the first operation in a given cube is a `natural_join`, it is unlikely that applying a `natural_join` next will lead to a solution. Therefore, a cube that uses a `natural_join` followed by an `inner_join` is generated before a cube that applies two `natural_join` in sequence.

6.2.2 Dynamic Cube Generation

Considering that the static generation heuristic was empirically designed based on available benchmark instances, its behavior might not be adequate for new instances. Therefore, CUBES also includes a cube

1. natural_join
2. natural_join3
3. natural_join4
4. mutate
5. summarise
6. filter
7. anti_join
8. left_join
9. union
10. intersect
11. semi_join
12. inner_join
13. cross_join

(a) Order used if the previous line is not a natural_join* operation.

1. mutate
2. summarise
3. filter
4. anti_join
5. left_join
6. union
7. intersect
8. semi_join
9. inner_join
10. cross_join
11. natural_join
12. natural_join3
13. natural_join4

(b) Order used if the previous line is a natural_join* operation.

Figure 6.3: Order in which operations are chosen when using Static Cube Generation.

generator inspired on Natural Language Processing (NLP) techniques. Since cubes are constructed as a sequence of operations, a bigram prediction model can be used to decide the operation to be placed next in a given sequence. Therefore, when choosing the operation for a given position in the sequence, the operation immediately preceding it is used to compute the likelihood that each of the possible choices will lead to the desired program. That is, for each pair of operations (a, b) there is a score, $S_{\texttt{a,b}}$, that represents the likelihood that using a b operation after an a operation will lead to the desired program. Scores are updated as programs are evaluated in the following way:

Program scoring

For a given program, p, let output denote the result of running that program in a given example specified by the user. Moreover, let expected denote the desired result in the input-output example. First, we compute the set of all values that occur in the output and expected tables: $\mathbf{unique}(\texttt{output})$ and $\mathbf{unique}(\texttt{expected})$. Next, we compute the score of program p as the percentage of elements of the expected output that appear in the result obtained by executing program p as:

$$score(p) = \frac{|\mathbf{unique}(\texttt{output}) \cap \mathbf{unique}(\texttt{expected})|}{|\mathbf{unique}(\texttt{expected})|} \tag{6.1}$$

A score of 1 indicates that all the expected values occur in the output, and as such, a filtering or restructuring might lead to a correct program. On the other hand, a value of 0 means that the candidate program is probably very far from a correct solution. Note that any program p where $score(p) \neq 1$ is certainly incorrect. This can be used as an optimization in order to avoid expensive table comparisons.

Score updates

For each evaluated program, p, the score, $score(p)$, is used to update the bigram scores. Consider that program p uses the following components: filter, natural_join, summarise (in that order). Then, the

scores for the bigrams that appear in the program will be updated as follows:

$$S_{\emptyset,\texttt{filter}} \mathrel{+}= score(p) \tag{6.2}$$

$$S_{\texttt{filter},\texttt{natural\_join}} \mathrel{+}= score(p) \tag{6.3}$$

$$S_{\texttt{natural\_join},\texttt{summarise}} \mathrel{+}= score(p) \tag{6.4}$$

Furthermore, we update the score of the operations occurring in the first position of the sequence, although with decreasing weights. In particular, the operation selected for position i (zero-based) of the sequence contributes with $\frac{1}{(i+1)^2} \cdot score(p)$. Hence, considering again the program p with components `filter`, `natural_join`, and `summarise`, the updates are as follows:

$$S_{\emptyset,\texttt{filter}} \mathrel{+}= 1/1 \cdot score(p) \tag{6.5}$$

$$S_{\emptyset,\texttt{natural\_join}} \mathrel{+}= 1/4 \cdot score(p) \tag{6.6}$$

$$S_{\emptyset,\texttt{summarise}} \mathrel{+}= 1/9 \cdot score(p) \tag{6.7}$$

These extra score updates are done so that there is a small chance of reordering operations, and has empirically shown to be useful.

Cube selection

Cubes are constructed by adding operations to a sequence. Suppose that the last selected operation is `op` (in case of the first operation, `op` is the empty symbol $\emptyset$). In order to decide which operation should follow, the scores for that prefix, $S_{\texttt{op}}$, are retrieved, normalized and smoothed, using Laplace smoothing [20]. These steps result in a list of probabilities that correspond to the likelihood of each operation. The operation for the current line is then chosen from a distribution using those probabilities. This is done until we have a program of the desired length. A compact tree structure is used to keep track of already generated cubes, as to avoid repetition.

Avoiding biases

The usage of the dynamic cube generation technique may introduce biases since the bigram scores are continuously increasing. In particular, operations that are selected first become more likely to be selected again when generating new cubes. Two methods are used to handle this issue:

- Each time a new program is generated, all scores are multiplied by a number smaller than one, δ, by default 0.99999. This is done so that past information can be gradually forgotten, in order to increase diversification in exploring the search space. These updates are done in batches, in order to not overwhelm inter-process communication.

- A fixed number of processes, by default 2, always solve randomly generated cubes (as long as not previously generated), in order to diversify the search process.

DSL Splitting

Two of the components introduced in the DSL, `inner_join` and `cross_join`, are much more complex than any of the other operations. That is, there are many more ways to complete a `cross_join` line than, for example, a `summarise` line. In fact, the difference in complexity is large enough to make encoding the program space into and SMT formula take a significant amount of time when those operations are enabled. As a compromise we split the available processes into two sets: set F is forced to only attempt programs that contain at least one of these two operations; and set B is configured as if these operations did not exist.

If the desired program does require one of the two complex joins, then the encoding overhead is unavoidable and the fact that some processes are only considering programs with those operations can more directly lead to a solution. On the other hand, if the desired program does not require a complex join, then the overhead is completely avoided. The goal is then to balance the number of processes allocated to each set in order to maximize the number of programs that can be solved. The ratio between sets F and B is configurable and defaults to 1:2.

6.2.3 Optimal and Non-Optimal Solving

As explained in subsection 4.2.1, SQUARES, and by extension CUBES, enumerates programs in increasing size. The same is true for cube generation. However, when splitting the search space, it is common for some processes to finish searching the final cubes for the current program size, while others are still trying candidate programs. CUBES allows these processes to start searching cubes of the next size, so that they do not stall. However, this means that a solution of size n can be found before all programs of size $n-1$ have been explored (and therefore a shorter solution might exist). CUBES allows for the user to choose between:

- Optimal synthesis: if a solution of size n is found while cubes of size $n-1$ are still being solved, all other processes of size n are stopped and the synthesizer waits to check if any of the cubes of size $n-1$ produce a solution. The shortest program is returned to the user. Furthermore, if the user terminates the program while it is searching for a better solution, the shortest program found so far is returned;

- Non-optimal synthesis: the first solution found is immediately returned to the user, even if a shorter solution might exist.

6.2.4 Learning from Unfeasible Cubes

Cubes are implemented by adding supplemental constraints to the SMT solver. A cube stating that the first line should be a `filter` and the second line should be a `summarise` would be implemented as $comp_1 = \mathbf{id}(\texttt{filter}) \wedge comp_2 = \mathbf{id}(\texttt{summarise})$. We can take advantage of UNSAT cores, a capability of SMT solvers, to further prune the search space.

An UNSAT core (unsatisfiable core) is a subset of constraints that by themselves make a formula unsatisfiable. In Z3, the SMT solver used by CUBES, UNSAT cores can be obtained by labeling relevant constraints and then asking Z3 which of those labels are part of the UNSAT core. Suppose that for the cube represented by the constraint $comp_1 = \mathbf{id}(\texttt{filter}) \wedge comp_2 = \mathbf{id}(\texttt{summarise})$, Z3 determines that there is an UNSAT core composed by just the first part, $comp_1 = \mathbf{id}(\texttt{filter})$. That means that even if we tried to use a different component for the second line, it would always fail, as just the first constraint is enough to make the formula unsatisfiable. This information can then be used to prune those other cubes, as they will surely not produce a solution for the problem.

In general, every time a cube fails without producing any candidate program, we use the UNSAT core created by the SMT solver to prune all other cubes that would also fail, according to that UNSAT core.

Chapter 7

Evaluation

In order to test and compare our tool with other state of the art SQL synthesizers we took the set of benchmarks used in SQUARES and expanded it. Table 7.1 summarizes the benchmarks used for evaluation. All results were obtained on a dual socket Intel® Xeon® Silver 4110 @ 2.10GHz, for a total of 16 cores/32 threads, with 64GB of RAM. Furthermore, using `runsolver` [34], a limit of 10 minutes (wall-clock time) and 56GB of RAM was imposed on all solvers.

The set of benchmarks used is:

- `texbook`: 37 instances extracted from exercises from the popular database textbook, *Database Management Systems* [32];
- `55-tests`: 55 instances derived from the `textbook` benchmark;
- `scythe/recent-posts`, `scythe/top-rated-posts`: 55+51 instances collected from recent and top-rated posts, respectively, on the StackOverflow[1] website;
- `spider`: 3765 instances generated from a very large and diverse benchmark of NLP instances for SQL synthesizers. For each original instance, the SQL solution query was used, along with the

[1] `https://stackoverflow.com/`

Table 7.1: Summary of the benchmarks used for evaluation and comparison.

Benchmark	Source	# Instances
`textbook`	*Database Management Systems* [32]	37
`55-tests`	SQUARES [7]	55
`scythe/recent-posts`	SCYTHE [42]	51
`scythe/top-rated-posts`	SCYTHE [42]	57
`spider`	Spider[a]	3765
Total		3965

[a] `https://yale-lily.github.io/spider`

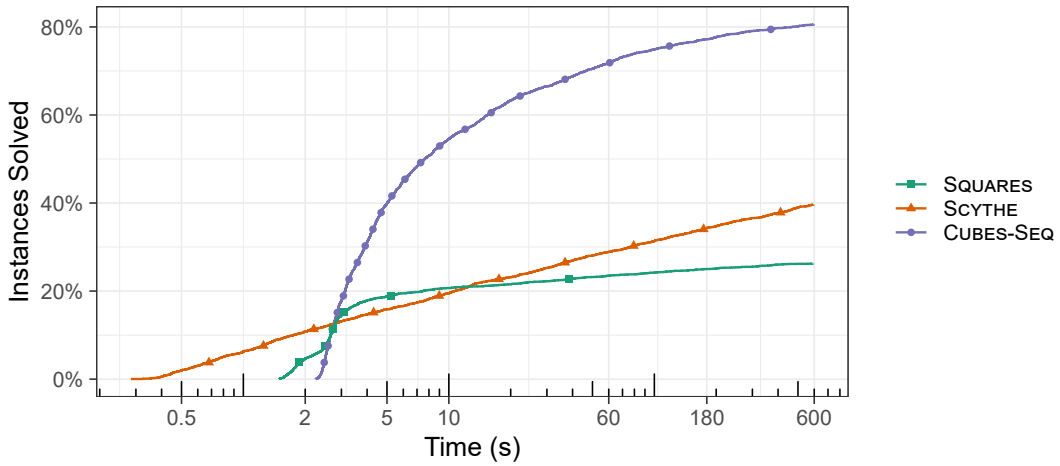


Figure 7.1: Percentage of instances solved by each synthesizer at each point in time. A mark is placed every 150 solved instances.

sample database contents, to create an input-output example that could be used in PBE synthesizers. Instances were transformed without intervention.

In this chapter, we will start by presenting the results for our sequential synthesizer, CUBES-SEQ, along with other state-of-the-art SQL synthesizers. Next, we show the results for both types of parallel synthesis implemented: portfolio and divide and conquer.

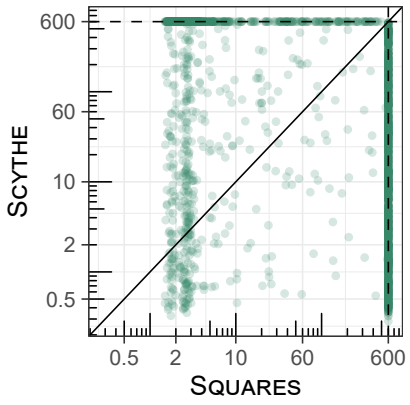
7.1 Sequential Results

In this section we evaluate the performance of CUBES-SEQ, the sequential version of CUBES. As a comparison point, we also present the results for SQUARES and SCYTHE. Figure 7.1 shows the percentage of instances solved by each of these tools at each point in time. Note that the time axis is in log-scale. Overall, SQUARES was able to solve 26.3% of the instances in 10 minutes, while SCTYHE solved 39.7%. CUBES-SEQ was able to solve 80.6%. CUBES-SEQ solved three times more instances than SQUARES, and two times more instances than SCYTHE.

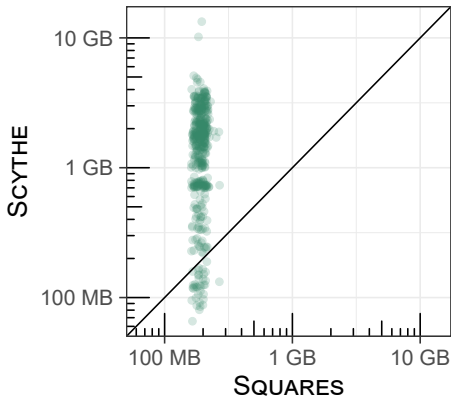
In some use cases, however, 10 minutes might be too long to wait for a solution. For example, the user might be reasonably familiar with SQL (but not proficient) and, as such, it might take less than 10 minutes to write the desired query manually. Therefore, we will also analyze the results using a virtual limit of 10 seconds, which would allow for these scenarios. Under the 10 second limit, CUBES-SEQ was able to solve 54.6% instances, while SQUARES solved 20.7%, and SCYTHE solved 19.5%.

7.1.1 SQUARES and SCYTHE

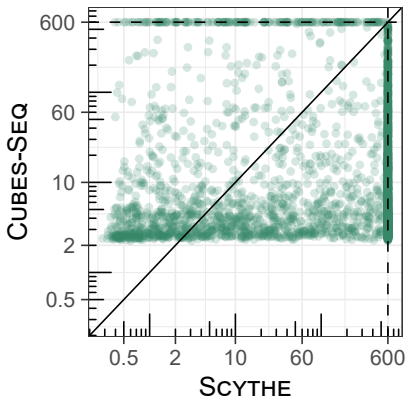
Figure 7.2a compares the time taken to solve each instance when using SQUARES and SCYTHE. Each mark in the plot represents a single instance; marks above the diagonal line mean that SQUARES solved that instance faster, while marks below the line mean the opposite. Finally, marks positioned on the dashed lines represent a timeout for the corresponding synthesizer. Only instances solved by at least



(a) Time-time scatter plot comparing SQUARES and SCYTHE.



(b) RAM-RAM scatter plot comparing SQUARES and SCYTHE.



(c) Time-time scatter plot comparing SCYTHE and CUBES-SEQ.

Figure 7.2: Scatter plots comparing the performance of SQUARES, SCYTHE and CUBES-SEQ.

one of the synthesizers are shown. Figure 7.2b shows a similar plot, but comparing RAM usage; in this plot only instances solved by both synthesizers are shown. Figure 7.2a is similar to Figure 7.2a, except that it compares SCYTHE and CUBES-SEQ.

Looking at Figure 7.2a we can see that there is a great disparity between the set of instances solved by SQUARES and the set solved by SCYTHE (that is, most instances lie on one of the timeout lines). This can be explained by the fact that these synthesizers operate in very different ways.

Furthermore, the great majority (78.7%) of instances solved by SQUARES are solved in the first 10 seconds, while the same is not true for SCYTHE (only 49.1%). This can also be seen in Figure 7.1 where although SCYTHE is generally faster, SQUARES actually comes ahead in the 3 to 10 seconds time-frame.

Finally, Figure 7.2b shows that SQUARES always uses around 200MB of RAM, while SCYTHE's RAM usage varies much more, reaching 10GB for some instances. This is likely because SCYTHE encodes the table's data into constraints, and as such, instances with bigger input tables use more memory. SQUARES, however, focuses mostly on the columns which makes its memory usage more consistent. This means SQUARES is more suited for parallelization as you can run more threads/processes in parallel without running out of RAM.

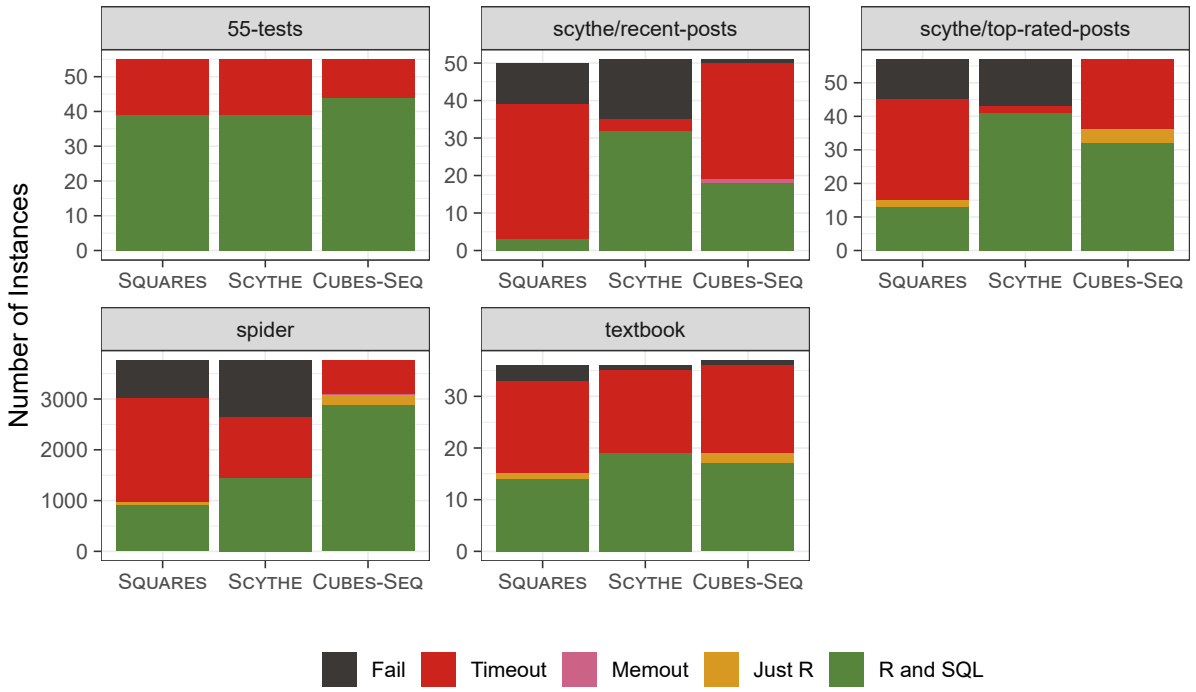


Figure 7.3: Number of instances solved by each synthesizer for each instance. Instances labeled Fail mean that the synthesizer crashed, while instances labeled Just R mean that the synthesizer was able to produce a correct R program but not an SQL query.

7.1.2 Cubes-Seq

Figure 7.3 details Cubes-Seq's performance for each benchmark. We can see that the number of solved instances improved on all benchmarks when comparing with Squares, while when comparing with Scythe the number of solved instances improved on the benchmarks from Squares and Spider, and decreased on both benchmarks from Scythe. The difference in solved instances between Scythe and Cubes-Seq can be seen in Figure 7.2c.

By default, Cubes-Seq is configured with: (i) Learning from Incorrect Programs disabled, (ii) the QF_FD SMT Theory enabled, (iii) the new DSL components introduced in section 5.1 enabled, and (iv) the Invalid Program Deduction enabled.

Accuracy Even though these tools can find queries consistent with the input-output examples, the solutions may not correspond to the user intent. In particular, Scythe has less input parameters and is thus more likely to find solutions that do not satisfy the user intent. We analyzed the percentage of solved queries that actually satisfy the user intent for Squares, Scythe and Cubes-Seq. To that end, we selected 15% of the instances solved by all three tools, resulting in 66 instances, and manually analyzed if the solutions found are equivalent to the ground truth SQL query. Of these 66 instances, Squares finds a solution that satisfies the user intent in 27 of them and Scythe returns such a solution in 33 instances. However, by default, Scythe returns the top 5 queries; if we only consider the queries ranked in first place, Scythe returns only 29 solutions that satisfy the user intent. Finally, Cubes-Seq returns a solution that satisfies the user intent in 46 out of the 66 randomly chosen instances.

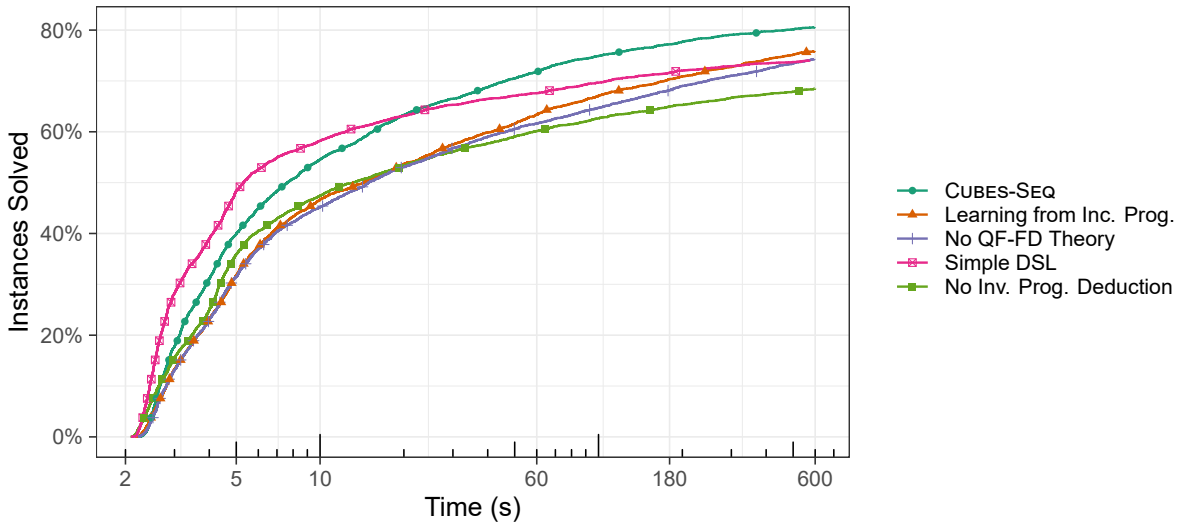


Figure 7.4: Plot showing the percentage of instances solved ate each point in time, when enabling/disabling different features from CUBES-SEQ.

In the rest of this subsection we will analyze the performance impact of CUBES' configuration options. Figure 7.4 summarizes those findings. In this figure, the line labeled "CUBES-SEQ" corresponds to the default configuration, while other lines correspond to the respective option switches.

DSL Extensions Looking at the "Simple DSL" line in Figure 7.4, we can see that when disabling the `inner_join`, `complex_join`, `semi_join` and `mutate` components some instances become easier (more instances are solved in under 20 seconds). That is because those instances did not use the new components and as such benefit from a more restricted program space. However, we can also see that the overall number of solved instances is lower (74% vs 80.6% on CUBES-SEQ), because of all other instances that do require the new DSL components.

Quantifier-Free Finite Domain Theory As shown by the "No QF-FD Theory" line in Figure 7.4, the performance when not using the QF_FD Theory is generally worse than with the the Theory enabled. This is expected, as all variables in CUBES' encoding are either Boolean, bit-vectors, or integers with very small bounds. Enabling this option allows CUBES-SEQ to solve 80.6% of the instances vs. 74.3% with the option disabled. The difference is even larger if we only consider the first 10 seconds: CUBES-SEQ is able to solve 54.6%, while disabling the QF_FD Theory only solves 45.2%.

Invalid Program Deduction As illustrated by the "No Inv. Prog. Deduction" line in Figure 7.4, disabling the invalid program pruning based on column annotations has the single largest impact in CUBES-SEQ performance. Note that when disabling this option, we re-enable the `happens_before` predicate since it becomes helpful again. This difference in performance shows that the new form of pruning is indeed stronger than the `happens_before` predicate. Furthermore, this option is almost always better, with only 18 instances out of 3965 being solved with the option disabled but not with the option enabled.

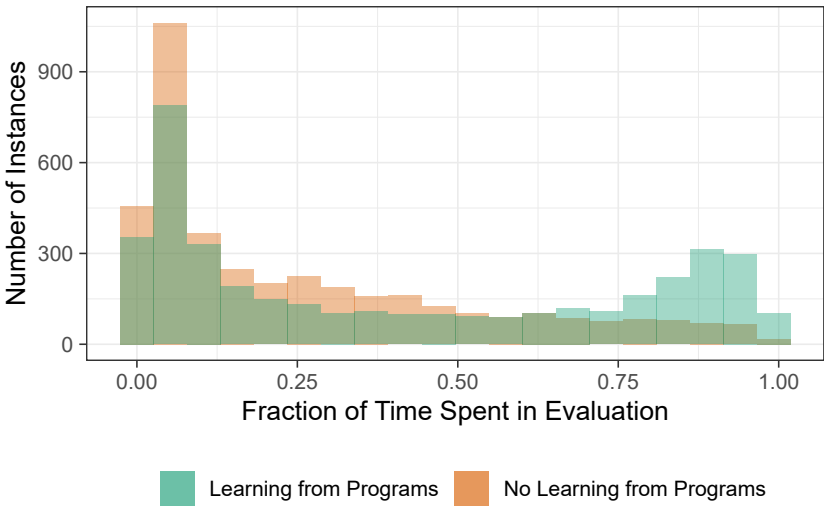


Figure 7.5: Overlaid histograms that compare the fraction of time spent doing program evaluation and verification when enabling/disabling the Learning from Programs option. Darker regions of the figure occur when the histograms overlap.

Learning from Incorrect Programs By looking at the "Learning from Inc. Prog." line in Figure 7.4, we can see that with learning from incorrect programs enabled, the number of solved instances is always lower than with the option disabled. This is because, in the current implementation, the number of pruned programs is quite small, due to the `select` post-processing step described in section 5.1.

Furthermore, there is a negative impact in evaluation time because some optimizations must be disabled. One such optimization is using the program scores in order to skip costly table comparisons (any program, p, where $(score)(p) \neq 1$ can be immediately rejected). When this form of learning is enabled, however, the `select` post-processing step must still be executed in order to generate all possible tables and evaluate if they have more or less rows than expected. Figure 7.5 shows that around 1100 instances spend more than 75% doing program evaluation and verification when enabling the Learning from Incorrect Programs option. With the option disabled, this only happens for about 300 instances.

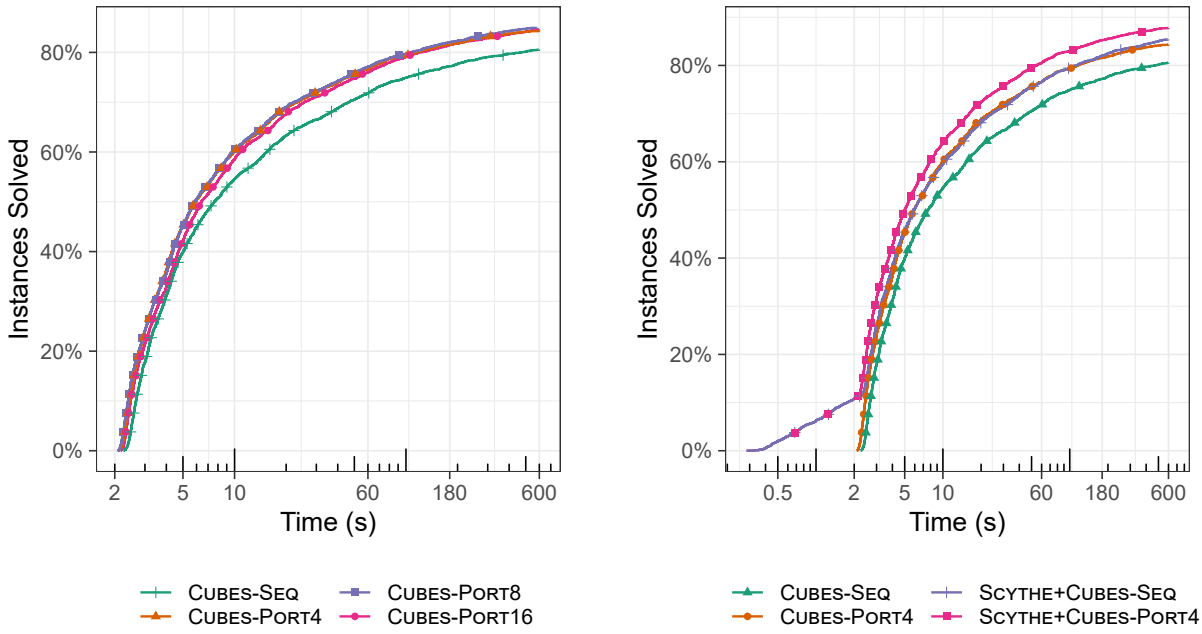
Although CUBES-SEQ is sometimes slower than SQUARES, this occurs only on a very small number of instances. Moreover, the number of newly solved instances within a 600 seconds timeout is very considerable, and the memory footprint, although slightly increased, is generally under 1GB. As a result, the new solver offers an improved starting point to develop a parallel solver for Query Reverse Engineering.

7.2 Parallel Results

Using CUBES-SEQ as a baseline, we now evaluate the performance of CUBES-PORT and CUBES-DC.

7.2.1 Portfolio

We evaluate the performance of CUBES-PORT for the three presets shown in Figure 6.1. Looking at Figure 7.6a, which shows the percentage of solved instances for each of these presets, we can see



(a) Comparison of the different presets available in CUBES-PORT.

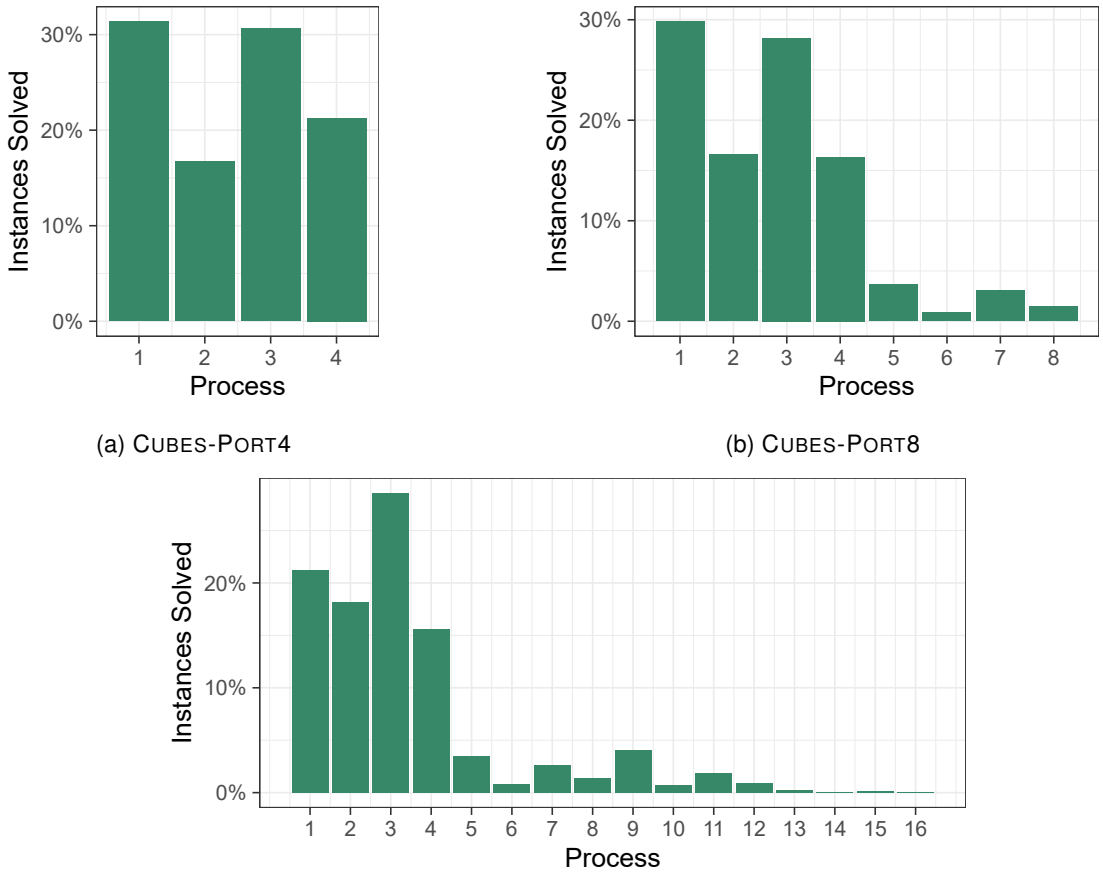
(b) Performance for VBSs combining CUBES and SCYTHE.

Figure 7.6: Performance comparison of different portfolio configurations, resulting from a single execution. Non-determinism effects are negligible due to the large number of total instances.

that CUBES-PORT4 constitutes a modest improvement over CUBES-SEQ, solving 84.4% vs 80.6% of the instances. However, as referenced in section 6.1, increasing the number of portfolio processes in a way that diversifies the search is no straightforward task. With that in mind, it comes at no surprise that the improvements going from CUBES-PORT4 to CUBES-PORT8 and CUBES-PORT16 processes are not as significant, with CUBES-PORT8 solving 84.9% of the instances and CUBES-PORT16 solving 84.6% of the instances. In particular, the diversity gained from the extra configurations considered in CUBES-PORT16 is not enough the overcome the performance penalty of using 16 cores in the system architecture used for testing.

Figure 7.7 shows the percentage of instances that were solved by each of the configurations in the portfolios (as defined in Figure 6.1). We can once again see that there are diminishing returns as we increase the number of processes from 4 to 8 and 16, with some of the extra processes in CUBES-PORT16 solving almost no instances.

Complementary, we can consider adding SCYTHE to the portfolio. In Figure 7.6b we show the percentage of solved instances for several Virtual Best Solvers (VBSs) which serve as a good approximation for the performance of an equivalent portfolio configuration. We can see that running CUBES-SEQ and SCYTHE in parallel has a performance similar to that of CUBES-PORT4. Furthermore, running CUBES-PORT4 and SCYTHE in parallel, which amounts to just 5 processes, has a much more significant impact than using CUBES-PORT8 or CUBES-PORT16.



(a) CUBES-PORT4

(b) CUBES-PORT8

(c) CUBES-PORT16

Figure 7.7: Percentage of instances solved by each of the configurations in the portfolio, for each of the presets in Figure 6.1.

7.2.2 Divide and Conquer

In Figure 7.8, we present the results for the divide-and-conquer approach for 4 processes (CUBES-DC4), 8 processes (CUBES-DC8) and 16 processes (CUBES-DC16). In the plot we can see that going from CUBES-SEQ to CUBES-DC4, CUBES-DC8 and CUBES-DC16, provides small improvements to the number of instances solved: 80.6%, 85.3%, 86.3% and 87.8%, respectively. If limited to 10 seconds, the difference becomes slightly larger with CUBES-SEQ solving 54.6% of the instances, CUBES-DC4 solving 69.9%, CUBES-DC8 solving 72.6% and CUBES-DC16 solving 74.4%. In Table 7.2, we can see that CUBES-DC16 is best overall configuration for both 10 minutes and 10 seconds. Furthermore, it is also the best configuration for all benchmarks except `scythe/recent-posts` under 10 seconds, and all benchmarks except `scythe/recent-posts` and `textbook` for 10 minutes.

Next, we analyze the effectiveness of the work splitting technique in CUBES-DC. For each instance, we compute the equivalent number of processes, which is defined as CPU Time / Wall Clock Time and is a measure of how much time each of the processes was stalled. A value of 1 means that if the work were uniformly distributed among processes, a single one would be enough to perform the same task in the same amount of time. On the other hand, a number equal to the real number of processes used means that every process was used all of the time.

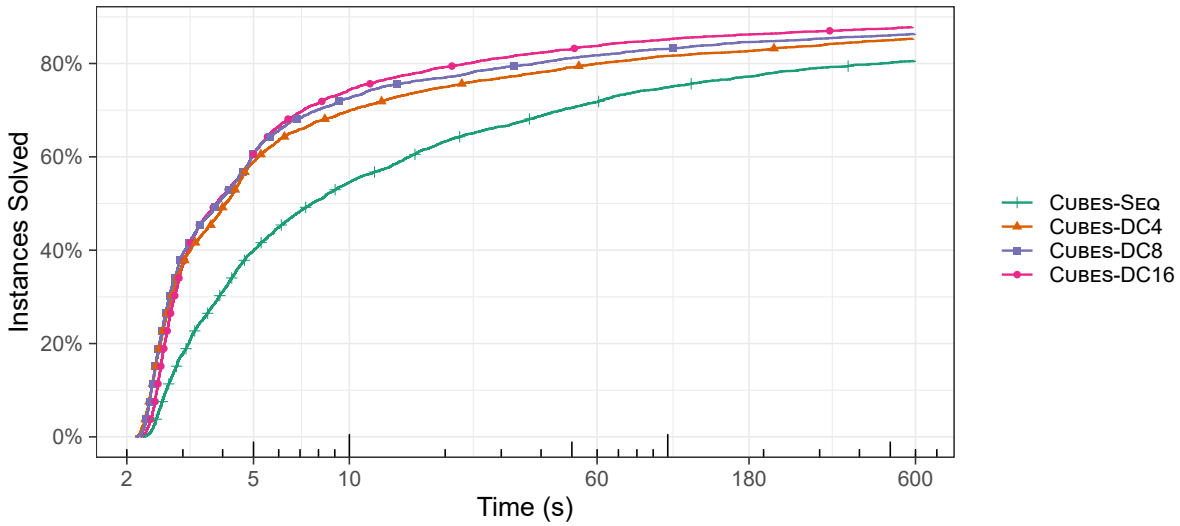


Figure 7.8: Performance impact of using different numbers of processes in CUBES-DC. CUBES-SEQ also shown as a comparison point. Results based on a single execution. Non-determinism effects are negligible due to the large number of total instances.

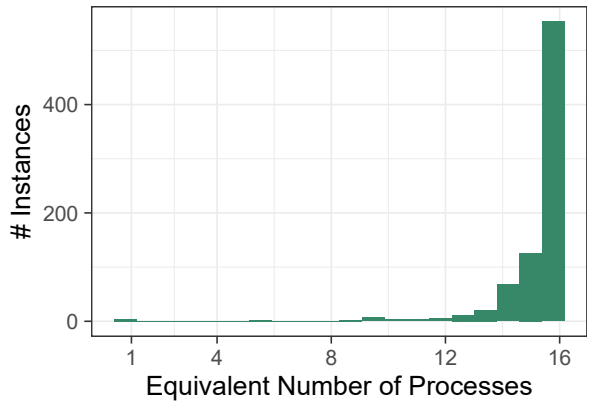


Figure 7.9: Histogram of the number of equivalent processes used by each instance. This number is computed as (CPU time)/(wall clock time) and represents the effectiveness of the work splitting algorithm. Instances that took less than 20 seconds are excluded, as they are skewed by the initialization time which is inherently single-threaded.

Figure 7.9 shows an histogram for this metric, as computed for CUBES-DC16. In this plot we hide instances that took less than 20 seconds to be solved, as the inherently sequential initialization procedure distorts the metric for these instances. We can see that 75.5% of the instances plotted have an equivalent process number ≥ 15, that is, it would require at least 16 processors to do the same work in the same amount of time, even if it were perfectly distributed.

CUBES-DC is non-deterministic, which means that, if run several times, it does not always produce the same solutions nor solve the same instances. We chose a subset of the instances and executed CUBES-DC16 for each of them 10 times, in order to count the number of different outcomes. These tests were executed using 8 processes, with a 5 minute time limit. We randomly selected 1% of instances, which amounts to 38 instances. Of these, CUBES-DC solved 33 of them on all 10 executions, while 2 instances were solved only once, 1 instance was solved in 3/10 executions and 2 instances were not

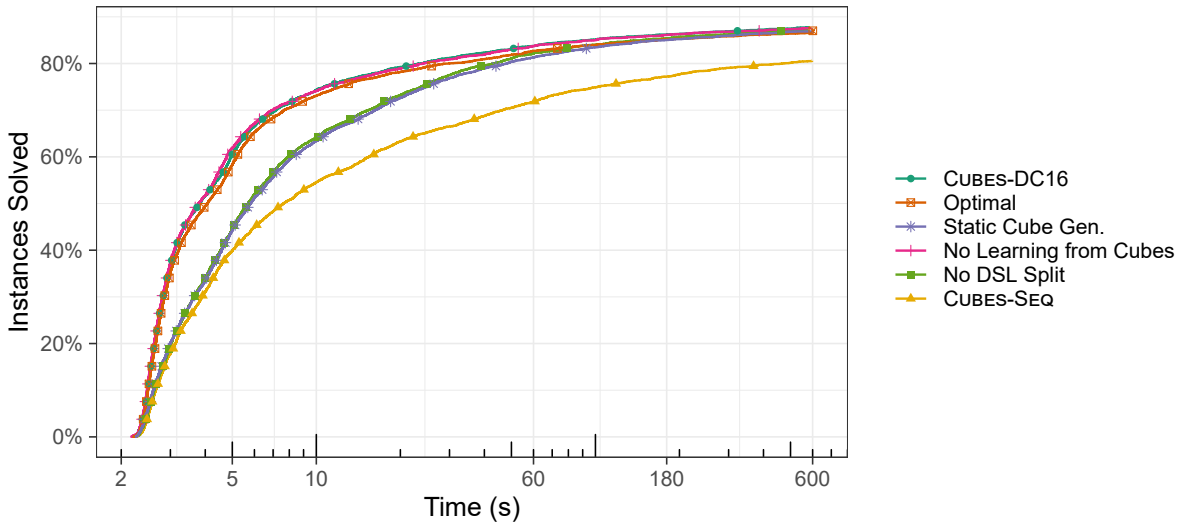


Figure 7.10: Performance impact of different features implemented in CUBES-DC. CUBES-SEQ is also shown as a comparison point. Results based on a single execution. Non-determinism effects are negligible due to the large number of total instances.

solved in any execution. Furthermore, the median number of different solutions was 2, while the average was 2.05, the mode was 1, and the maximum was 6.

By default, CUBES-DC is configured with the dynamic cube generator, non-optimal solving, learning from unfeasible cubes enabled and DSL splitting enabled. In order to evaluate each of these features, we present an ablation study in Figure 7.10. In the rest of this subsection we will analyze the performance impact of each feature.

Static vs. Dynamic Cube Generator In Figure 7.10, we can see that even though the percentage of solved instances is quite similar for 10 minutes (87.8% for the dynamic generator vs. 86.9% for the static), the difference is very large when looking just at the first 10 seconds (74.4% for dynamic vs. 63.4% for static). This implies that the dynamic cube generator can more quickly start exploring promising parts of the program space, as would be expected. Furthermore, looking at the results in Table 7.2, we can see that the performance difference for 10 seconds is much smaller for the `55-tests` and `textbook` benchmarks. This is also expected, as these benchmarks were used to develop the heuristic at the core of the static cube generator.

Optimality Looking again at Figure 7.10, we can see that configuring CUBES-DC to produce optimal solutions causes a general loss in performance, as would be expected. Furthermore, we can see a slight uptick in the percentage of solved instances right on the time limit line. These correspond to instances for which a (possibly) non-optimal solution had already been found and that were waiting for cubes with less lines to finish processing. Overall, enabling this option allows CUBES-PORT16 to solve 87.3% vs. 87.8% with the option disabled. Finally, out of all solved instances, only 29 (0.8% of solutions) correspond to possibly non optimal programs.

Learning from Unfeasible Cubes In Figure 7.10, we can see that the impact from the learning from unfeasible cubes option is almost non-existent, even though, on average, each generated cube allows 1.39 others to be pruned. There are are two possible explanations for this: (i) identifying unfeasible cubes is very fast, and thus has a negligible impact, and/or (ii) the Z3 SMT is able to infer clauses from unfeasible cubes that allow it to immediately identify future cubes that fail for the same reasons. Regarding the second possible explanation, it is important to take into account that since cubes are generated and pruned in the main thread, unfeasible cube learning has an effect on all processes, while clauses inferred by Z3 are process-local.

DSL Splitting As previously referenced in section 6.2.2, the main advantage of forcing programs containing `inner_join` or `complex_join` operations to be searched for in separate processes is that (i) if the instances does require one of those operations, then the enumerator can more quickly direct the search towards a correct program, and (ii) if the instance does not require one of those operations, then the initialization time resulting from the complex joins can be avoided. Therefore, it is no surprise that the percentage of solved instances in 10 minutes does not vary much whether this option enabled or disabled (87.8% with vs. 87.3% without), but that there is a much larger impact on the number of solved instances in under 10 seconds (74.4% with the option enabled vs. 64.1% without).

Table 7.2: Overall results for 10 seconds and 10 minutes, for all configurations tested, grouped by benchmark. The best configuration for each time-limit/benchmark pair is highlighted in **bold**. Results based on a single execution. Benchmarks with a small number of instances are affected by non-determinism when using parallel configurations. This explains, for example, why CUBES-DC16 is not the best run for the `textbook` benchmark for 10 minutes.

	Run	55-tests	scythe/recent-posts	scythe/top-rated-posts	spider	textbook	All
10 sec	SQUARES	32.7%	4.0%	3.5%	20.9%	30.6%	20.7%
	SCYTHE	38.2%	**45.1%**	61.4%	18.2%	27.8%	19.5%
	CUBES-SEQ	50.9%	19.6%	47.4%	55.4%	35.1%	54.6%
	Learning from Programs	47.3%	11.8%	35.1%	47.4%	35.1%	46.7%
	No QF-FD Theory	50.9%	17.6%	40.4%	45.7%	35.1%	45.2%
	Simple DSL	58.2%	15.7%	42.1%	59.2%	40.5%	58.2%
	No Inc. Prog. Deduction	49.1%	9.8%	35.1%	48.2%	35.1%	47.4%
	CUBES-PORT4	70.9%	21.6%	56.1%	60.8%	40.5%	60.2%
	CUBES-PORT8	74.5%	21.6%	56.1%	60.9%	45.9%	60.4%
	CUBES-PORT16	69.1%	21.6%	56.1%	59.1%	43.2%	58.5%
	CUBES-DC4	72.7%	21.6%	61.4%	70.8%	51.4%	69.9%
	CUBES-DC8	80.0%	25.5%	63.2%	73.4%	**54.1%**	72.6%
	CUBES-DC16	**83.6%**	29.4%	**66.7%**	**75.2%**	**54.1%**	**74.4%**
	Optimal	80.0%	29.4%	**66.7%**	73.8%	**54.1%**	73.1%
	Static Cube Gen.	78.2%	21.6%	56.1%	63.9%	**54.1%**	63.4%
	No Learning from Cubes	**83.6%**	27.5%	64.9%	75.0%	**54.1%**	74.1%
	No DSL Split	80.0%	23.5%	59.6%	64.6%	48.6%	64.1%
10 min	SQUARES	70.9%	6.0%	26.3%	25.7%	41.7%	26.3%
	SCYTHE	70.9%	**62.7%**	71.9%	38.3%	52.8%	39.7%
	CUBES-SEQ	80.0%	35.3%	63.2%	81.8%	51.4%	80.6%
	Learning from Programs	80.0%	29.4%	63.2%	76.8%	56.8%	75.9%
	No QF-FD Theory	80.0%	29.4%	71.9%	75.1%	48.6%	74.3%
	Simple DSL	89.1%	25.5%	70.2%	74.6%	67.6%	74.0%
	No Inc. Prog. Deduction	70.9%	33.3%	61.4%	69.3%	43.2%	68.5%
	CUBES-PORT4	90.9%	41.2%	70.2%	85.3%	59.5%	84.4%
	CUBES-PORT8	92.7%	43.1%	73.7%	85.8%	64.9%	84.9%
	CUBES-PORT16	92.7%	43.1%	73.7%	85.5%	62.2%	84.6%
	CUBES-DC4	90.9%	39.2%	**75.4%**	86.1%	**73.0%**	85.3%
	CUBES-DC8	92.7%	45.1%	**75.4%**	87.1%	70.3%	86.3%
	CUBES-DC16	**94.5%**	47.1%	**75.4%**	**88.6%**	70.3%	**87.8%**
	Optimal	**94.5%**	47.1%	**75.4%**	88.1%	**73.0%**	87.3%
	Static Cube Gen.	**94.5%**	41.2%	**75.4%**	87.7%	70.3%	86.9%
	No Learning from Cubes	92.7%	47.1%	**75.4%**	88.4%	**73.0%**	87.6%
	No DSL Split	**94.5%**	47.1%	**75.4%**	88.0%	**73.0%**	87.3%

Chapter 8

Conclusions and Future Work

In this thesis, we explored the topic of Parallel Program Synthesis. We introduced the topics of Program Synthesis and Parallel Constraint Solving, and discussed state-of-the-art techniques used in these fields. We proposed a new sequential program synthesizer, CUBES-SEQ, which is based on SQUARES and constitutes an improvement to the state of the art in sequential SQL synthesis. We also propose CUBES-PORT and CUBES-DC, two parallel synthesizers for SQL, using techniques inspired by parallel constraint solvers.

We performed an extensive evaluation the of the implemented tools, comparing them with SQUARES and SCYTHE. To perform this comparison, we used 200 benchmarks from previous work in PBE SQL synthesis, along with 3765 benchmarks which we adapted from NLP SQL synthesizers. We show that CUBES-SEQ is able to solve 80.6% of the considered instances, while SQUARES and SCYTHE can only solve 26.3% and 39.7%, respectively. We also show, that using parallelism provides a significant performance improvement with CUBES-DC solving 87.8% of the instances and CUBES-PORT solving 84.9%. Finally, we show that CUBES-DC scales better with the number of available processors than CUBES-PORT.

We propose a number of ways in which CUBES can be improved in the future. Firstly, since the number of processing cores available in a CPU is limited by physical and manufacturing constraints, even very high-end processors have at most 72 cores, which limits the scalability of CUBES-DC. A possible solution for this problem is to use a distributed approach, instead of multi-core. This improvement is expected to not have a large impact in the structure of CUBES-DC, since inter-process communication is already done using message passing techniques, and no shared memory is used.

Regarding the cube generation order, more elaborate machine learning techniques could be used such as using pre-trained bigram scores, or using neural networks to predict the most likely cubes. We could also explore other techniques used in Propositional Satisfiability solvers, such as restarting the search after n programs/cubes have been attempted.

It would also be interesting to add an option for CUBES-DC to be more deterministic (at the cost of performance). Proposed changes include: (i) updating the bigram scores in batches and in a deterministic way, (ii) solve cubes in batches so that processes stay synchronized — this would require that cubes

be of approximately the same difficulty in order to reduce stalls, and (iii) either find a deterministic way to assign generated cubes to the available processes or disable some optimizations with are process-local and depend on the order the received cubes.

Finally, CUBES-PORT can be improved by combining it with new state-of-the-art program synthesizers.

Bibliography

[1] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. 2013 Formal Methods in Computer-Aided Design, pages 1–8, Oct. 2013. DOI: 10.1109/FMCAD.2013.6679385.

[2] C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.6, Department of Computer Science, The University of Iowa, 2017.

[3] A. Biere, M. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009. ISBN: 978-1-58603-929-5.

[4] Q. Chen, X. Wang, X. Ye, G. Durrett, and I. Dillig. Multi-modal synthesis of regular expressions. In A. F. Donaldson and E. Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 487–502. ACM, 2020. DOI: 10.1145/3385412.3385988.

[5] Y. Chen, R. Martins, and Y. Feng. Maximal Multi-layer Specification Synthesis. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia), ESEC/FSE 2019, pages 602–612, New York, NY, USA. ACM, 2019. ISBN: 978-1-4503-5572-8. DOI: 10.1145/3338906.3338951.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. ISBN: 978-0-262-03384-8.

[7] P. M. O. M. da Silva. *SQUARES: A SQL Synthesizer Using Query Reverse Engineering*, Instituto Superior Técnico, Universidade de Lisboa, Nov. 2019.

[8] L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg. Springer-Verlag, Mar. 29, 2008. ISBN: 978-3-540-78799-0.

[9] 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* (Philadelphia, PA, USA), PLDI 2018, pages 420–435, New York, NY, USA. ACM, 2018. ISBN: 978-1-4503-5698-5. DOI: 10.1145/3192366.3192382.

[10] Y. Feng, R. Martins, J. Van 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* (Barcelona, Spain), PLDI 2017, pages 422–436, New York, NY, USA. ACM, 2017. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062351.

[11] 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* (Paris, France), POPL 2017, pages 599–612, New York, NY, USA. ACM, 2017. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009851.

[12] Gartner, Inc. Magic Quadrant for Enterprise Low-Code Application Platforms. ID G00361584, Aug. 8, 2019.

[13] C. Green. Application of Theorem Proving to Problem Solving. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence* (Washington, DC), IJCAI'69, pages 219–239, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc., 1969. URL: http://dl.acm.org/citation.cfm?id=1624562.1624585 (visited on 09/27/2019).

[14] S. Gulwani, O. Polozov, and R. Singh. *Program Synthesis*. now, 2017. ISBN: 978-1-68083-292-1.

[15] S. Gulwani. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 317–330, New York, NY, USA. ACM, 2011. ISBN: 978-1-4503-0490-0. DOI: 10.1145/1926385.1926423.

[16] Y. Hamadi and L. Sais, editors. *Handbook of Parallel Constraint Reasoning*. Springer International Publishing, 2018. ISBN: 978-3-319-63515-6. DOI: 10.1007/978-3-319-63516-3.

[17] M. Heule and H. van Maaren. Look-ahead based SAT solvers. In *Handbook of Satisfiability*. Volume 185, Frontiers in Artificial Intelligence and Applications, pages 155–184. IOS Press, 2009.

[18] M. J. H. Heule, O. Kullmann, and A. Biere. Cube-and-conquer for satisfiability. In *Handbook of Parallel Constraint Reasoning*, pages 31–59. Springer, 2018.

[19] 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* (Cape Town, South Africa), ICSE '10, pages 215–224, New York, NY, USA. ACM, 2010. ISBN: 978-1-60558-719-6. DOI: 10.1145/1806799.1806833.

[20] D. Jurafsky and J. H. Martin. *Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition, 2nd Edition*. Prentice Hall series in artificial intelligence. Prentice Hall, Pearson Education International, 2009. ISBN: 9780135041963.

[21] T. Knoth, D. Wang, N. Polikarpova, and J. Hoffmann. Resource-guided Program Synthesis. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA), PLDI 2019, pages 253–268, New York, NY, USA. ACM, 2019. ISBN: 978-1-4503-6712-7. DOI: 10.1145/3314221.3314602.

[22] F. Li and H. V. Jagadish. Nalir: an interactive natural language interface for querying relational databases. In C. E. Dyreson, F. Li, and M. T. Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 709–712. ACM, 2014. DOI: 10.1145/2588555.2594519.

[23] H. Li, C.-Y. Chan, and D. Maier. Query from examples: an iterative, data-driven approach to query construction. *Proc. VLDB Endow.*, 8(13):2158–2169, 2015. DOI: 10.14778/2831360.2831369.

[24] J. Marques-Silva, I. Lynce, and S. Malik. Conflict-Driven Clause Learning SAT Solvers. In *Handbook of Satisfiability*. Volume 185, Frontiers in Artificial Intelligence and Applications, pages 131–153. IOS Press, 2009.

[25] R. Martins, J. Chen, Y. Chen, Y. Feng, and I. Dillig. Trinity: An Extensible Synthesis Framework for Data Science. *Proc. VLDB Endow.*, 12(12):1914–1917, Aug. 2019. ISSN: 2150-8097. DOI: 10.14778/3352063.3352098.

[26] R. Martins, V. Manquinho, and I. Lynce. An overview of parallel SAT solving. *Constraints*, 17(3):304–347, July 1, 2012. ISSN: 1572-9354. DOI: 10.1007/s10601-012-9121-3.

[27] P. Orvalho, M. Terra-Neves, M. Ventura, R. Martins, and V. Manquinho. Encodings for Enumeration-Based Program Synthesis. In T. Schiex and S. de Givry, editors, *Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, pages 583–599, Cham. Springer International Publishing, 2019. ISBN: 978-3-030-30048-7. DOI: 10.1007/978-3-030-30048-7_34.

[28] P. Orvalho, M. Terra-Neves, M. Ventura, R. Martins, and V. Manquinho. SQUARES: a SQL synthesizer using query reverse engineering. *Proceedings of the VLDB Endowment*, 13(12):2853–2856, Aug. 1, 2020. ISSN: 2150-8097. DOI: 10.14778/3415478.3415492.

[29] N. Polikarpova, I. Kuraj, and A. Solar-Lezama. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA), PLDI '16, pages 522–538, New York, NY, USA. ACM, 2016. ISBN: 978-1-4503-4261-2. DOI: 10.1145/2908080.2908093.

[30] N. Polikarpova and I. Sergey. Structuring the Synthesis of Heap-manipulating Programs. *Proc. ACM Program. Lang.*, 3:72:1–72:30, POPL, Jan. 2019. ISSN: 2475-1421. DOI: 10.1145/3290385.

[31] 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* (Pittsburgh, PA, USA), OOPSLA 2015, pages 107–126, New York, NY, USA. ACM, 2015. ISBN: 978-1-4503-3689-5. DOI: 10.1145/2814270.2814310.

[32] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, Inc., USA, 3rd edition, 2002. 1104 pages. ISBN: 978-0-07-246563-1.

[33] D. R. Ramos. *Program Synthesis from Noisy Tabular Data*, Instituto Superior Técnico, Universidade de Lisboa, Nov. 2019.

[34] O. Roussel. Controlling a Solver Execution with the runsolver Tool: System description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(4):139–144, Nov. 1, 2011. ISSN: 15740617. DOI: 10.3233/SAT190083.

[35] J. Sen, C. Lei, A. Quamar, F. Özcan, V. Efthymiou, A. Dalmia, G. Stager, A. R. Mittal, D. Saha, and K. Sankaranarayanan. ATHENA++: natural language querying for complex nested SQL queries. *Proc. VLDB Endow.*, 13(11):2747–2759, 2020. URL: http://www.vldb.org/pvldb/vol13/p2747-sen.pdf.

[36] D. E. Shaw, W. R. Swartout, and C. C. Green. Inferring LISP Programs from Examples. In *Proceedings of the 4th International Joint Conference on Artificial Intelligence - Volume 1* (Tblisi, USSR), IJCAI'75, pages 260–267, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc., 1975. URL: http://dl.acm.org/citation.cfm?id=1624626.1624666 (visited on 05/18/2019).

[37] M. Sipser. *Introduction to the Theory of Computation*. Course Technology Cengage Learning, Boston, MA, 3rd edition edition, 2012. ISBN: 978-1-133-18779-0.

[38] A. Solar-Lezama. *Program Synthesis by Sketching*. PhD Thesis, University of California at Berkeley, Berkeley, CA, USA, 2008.

[39] Q. T. Tran, C. Y. Chan, and S. Parthasarathy. Query reverse engineering. *VLDB J.*, 23(5):721–746, 2014. DOI: 10.1007/s00778-013-0349-3.

[40] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. Query by output. In U. Çetintemel, S. B. Zdonik, D. Kossmann, and N. Tatbul, editors, *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. ACM, 2009. DOI: 10.1145/1559845.1559902.

[41] P. van der Tak, M. Heule, and A. Biere. Concurrent cube-and-conquer - (poster presentation). In A. Cimatti and R. Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 475–476. Springer, 2012. DOI: 10.1007/978-3-642-31612-8_42.

[42] C. Wang, A. Cheung, and R. Bodik. Synthesizing Highly Expressive SQL Queries from Input-output Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain), PLDI 2017, pages 452–466, New York, NY, USA. ACM, 2017. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062365.

[43] H. Wang, L. Chen, M. Li, and M. Chen. Guidesql: utilizing tables to guide the prediction of columns for text-to-sql generation. In *2020 International Joint Conference on Neural Networks, IJCNN 2020, Glasgow, United Kingdom, July 19-24, 2020*, pages 1–7. IEEE, 2020. DOI: 10.1109/IJCNN48605.2020.9206700.

[44] N. Yaghmazadeh, Y. Wang, I. Dillig, and T. Dillig. SQLizer: Query Synthesis from Natural Language. *Proc. ACM Program. Lang.*, 1:63:1–63:26, OOPSLA, Oct. 2017. ISSN: 2475-1421. DOI: 10.1145/3133887.

[45] S. Zhang and Y. Sun. Automatically synthesizing SQL queries from input-output examples. In E. Denney, T. Bultan, and A. Zeller, editors, *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 224–234. IEEE, 2013. DOI: 10.1109/ASE.2013.6693082.