TÉCNICO LISBOA

Sketch-Driven SQL Synthesis

Viviana de Brito Bernardo

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Professor Maria Inês Camarate de Campos Lynce de Faria
Doctor Miguel Ângelo da Terra Neves

Examination Committee

Chairperson: Professor Pedro Tiago Gonçalves Monteiro
Supervisor: Professor Maria Inês Camarate de Campos Lynce de Faria
Member of the Committee: Professor Pedro Miguel dos Santos Alves Madeira Adão

October 2021

Acknowledgments

Firstly I would like to thank my advisors, Professor Inês Lynce and Miguel Neves, for all their relevant insight and for giving me the support I needed to overcome this chapter of my life. As I said multiple times, they are the perfect team both for their encouragement and also, for their valuable guidance. Without them, this work would not have been possible.

To my family for all your help.

To my friends for always being there in this time of pandemic, making every day seem the most normal that it can be. Thank you for all your help, for being there to cheer me on, and also for all the help and insightful discussions, especially to Joana who stood by my side and helped me to keep going.

A special thanks to João Ricardo for always being there for me and being my rock. Without you, this year would have been much harder.

To TMIST, for all the fantastic moments during these years. I will never forget all the nights and festivals and all the laughs and tears we shared together.

Finally, to all my friends and colleagues that helped me be who I am today and were always there for me during the good and bad times in my life. Thank you!

This work was supported by OutSystems, by national funds through *Fundação para a Ciência e Tecnologia* under project UIDB/50021/2020, and project GOLEM (reference ANI 045917) funded by FEDER and FCT.

Abstract

Nowadays, the amount of data that needs to be analyzed and manipulated is rapidly increasing. Performing these manipulations is often a complex and cumbersome task. OutSystems is a Low-Code Development Platform whose goal is to simplify the development of many applications enabling a rapid, agile, and continuous delivery of enterprise-grade applications with most of the tasks automated by the platform itself. This automation of tasks is closely related to the program synthesis problem, which consists in automatically generating a program from a specification. Furthermore, many of these manipulations follow a specific pattern from which we could use information in order to generate a program more efficiently and accurately.

This dissertation thesis introduces SKEL, an SQL query synthesizer that follows a sketch to guide the search process and uses input-output examples as a specification. We start by studying different types of sketches and create a sketch format that supports a wide range of different options and holes. We then use CUBES as a starting point, modifying it to receive sketches and adding several constraints in order to optimize the search process. Finally, we evaluate our tool by comparing it to a baseline and analyzing the accuracy and efficiency improvement when using different types of sketches. Overall, our results show that SKEL is more efficient and more accurate when compared with previous solutions for the same problem.

Keywords

Program Synthesis, Input-Output Examples, Sketching, SQL, Programming by Example, Query Reverse Engineering.

Resumo

Atualmente, a quantidade de dados que necessitam de ser analisados e manipulados está a crescer rapidamente. Estas manipulações são normalmente um processo complexo e moroso. A OutSystems é uma plataforma de desenvolvimento de baixo código, cujo objetivo é simplificar o desenvolvimento de aplicações para permitir uma rápida, ágil, e contínua entrega de aplicações a nível empresarial, com a maioria das tarefas a serem automatizadas pela plataforma. Esta automatização de tarefas está fortemente relacionada com o problema da síntese de programas, que consiste na geração automática de um programa através de uma especificação. Para além disto, muitas destas manipulações seguem um determinado padrão. Esta informação leva-nos a ser capazes de gerar programas de uma forma mais eficiente e com uma maior precisão.

Nesta dissertação de tese, introduzimos SKEL, um sintetizador de queries SQL que utiliza exemplos de input-output como especificação e utiliza também um sketch para guiar o processo de síntese de uma forma mais eficaz. Começamos por estudar diferentes tipos de sketches e criamos um formato de sketch que suporta uma grande variedade de opções e lacunas. Depois, usamos a ferramenta CUBES como ponto de partida, modificando-a para receber sketches de forma a otimizar o processo de procura. Finalmente, avaliamos a nossa ferramenta analisando em que medida a precisão e rapidez melhoram ao usarmos diferentes tipos de sketches. Em geral, os nossos resultados mostram que SKEL é mais eficiente e preciso quando comparado com outras soluções para o mesmo problema.

Palavras Chave

Síntese de Programas, Exemplos de Input-Output, Sketching, SQL, Programação por exemplo, Engenharia Reversa de Queries.

Contents

List of Figures

List of Tables

Acronyms

CEGIS	Counterexample Guided Inductive Synthesis
CNF	Conjunctive Normal Form
DSL	Domain Specific Language
LIA	Linear Integer Arithmetic
LOC	Lines of Code
LCDP	Low-Code Development Platform
NLP	Natural Language Processing
PBE	Programming by Example
QRE	Query Reverse Engineering
SAT	Propositional Satisfiability
SMT	Satisfiability Modulo Theory

Chapter 1

Introduction

In recent years, Low-Code Development Platforms (LCDPs) have become a crucial tool in the web/mobile application development domain. Forrester [27] and Gartner [33] also estimate a significant market increase for LCDPs in the following years. These platforms allow a fast delivery of applications with the need for minimal hand-coding, which makes the task of programming more accessible to users from different backgrounds.

OutSystems[1] is an LCDP that simplifies the development of many applications, enabling a rapid, agile, and continuous delivery of enterprise-grade applications, with most of the tasks automated by the platform itself. This task automation is closely related to the program synthesis problem in which the user has to provide a specification that contains the behavior of the program to be implemented. According to that specification, the synthesizer automatically generates a program that satisfies it.

One of these cumbersome tasks is the manipulation of a large amount of data. Nowadays, many users who have to perform these manipulation tasks lack the knowledge on how to perform them. CUBES [5] is a state-of-the-art synthesizer that generates an SQL query by analyzing a specification consisting of input-output examples. A weak point of CUBES is the time it takes to generate an SQL query and the accuracy of the generated queries since not all correspond to the user intent, although they are according to the specification.

We introduce SKEL, a synthesizer that receives a sketch as part of the specification to tackle this issue. We created our sketch format in order to implement this new feature in CUBES. These sketches contain information about the structure of the query the user intends to generate and also help to guide the synthesis process more efficiently and accurately. We also analyzed different types of sketch structures using our sketch format in order to understand which types of sketch have more impact on the efficiency and accuracy of our synthesizer. By using sketches, our goal is to increase the efficiency of the synthesizer and increase the overall accuracy of the queries generated.

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

id	architect_id	name	length
1	1	Xian Ren Qiao (Fairy Bridge)	121.0
2	2	Landscape Arch	88.0
3	3	Kolob Arch	87.0
4	3	Sipapu Natural Bridge	69.0
5	2	Stevens Arch	67.0
6	1	Shipton's Arch	65.0
7	1	Jiangzhou Arch	65.0

Table 1.1: Table Bridge

id	name	nationality
1	Frank Lloyd	American
2	Frank Gehry	Canadian
3	Zaha Hadid	British

Table 1.2: Table Architect

name
Jiangzhou Arch
Shipton's Arch
Xian Ren Qiao (Fairy Bridge)

Table 1.3: Output Table

1.1 Motivating Example

Suppose that the user wants to generate an SQL query that manipulates Table 1.1 and Table 1.2 (input example) into Table 1.3 (output example). This operation where the output table is generated using a filter between the input tables is a common one, so firstly, we need to perform a `JOIN` between the two input tables. With this, we could provide a possible sketch that could represent this implementation, guiding the synthesis process more efficiently and accurately. A possible sketch is represented in Example 1:

Example 1. *Sketch motivation:*

$$T1 = filter\ (??,\ ??)$$
$$T2 = inner\_join\ (T1,\ ??,\ ??)$$
$$out = select\ (name)\ order\ by(??)$$

With this sketch, the synthesizer only searches for solutions that follow its structure, pruning all others. This way, the performance of the synthesizer is enhanced, and the accuracy increases. The intended solution to this problem instance would be the query in Example 2:

Example 2. *Solution motivation query:*

```
SELECT name
FROM bridge AS b
JOIN architect AS a ON b.architect_id = a.id
WHERE a.nationality = 'American'
ORDER BY b.length
```

It is relevant to note that this type of structure in a query is very common; for example, it may correspond to a `JOIN` and a filter condition or to a `JOIN` and an aggregate. SKEL will benefit from these common types of queries that tend to follow a specific structure.

1.2 Contributions

This dissertation introduces SKEL, a new synthesizer to generate SQL programs, taking an input-output example as specification and a sketch to guide the search process more efficiently.

SKEL was developed on top of the CUBES synthesizer in order to extend it to receive sketches. We also introduced a new sketch format that could be easily integrated into SKEL and allowing it to support a wide range of different options and holes.

Furthermore, we gathered a subset SQL instances used by previous synthesizers in order to analyze SKEL's performance and also understand which types of sketch structures would be the best to provide while using our tool.

In summary, this thesis main contributions are:

- The SKEL synthesizer that extends CUBES into receiving sketches as part of the specification and, by doing so, increases the efficiency and accuracy;
- Introduction of a new sketch format with a wide range of different options and holes;
- An analysis of several types of sketches to understand which ones perform better and more accurately;
- SKEL is publicly available on an online repository[2].

1.3 Organization of the Document

This document is organized as follows. Firstly in Chapter 2, we introduce background concepts necessary to understand the following chapters. In Chapter 3, we analyze techniques that are relevant for our developed work and study several implementations of SQL synthesizers, including CUBES which is going to be the basis of this work. Next, in Chapter 4, we describe our new sketch format and the SKEL tool implementation in detail, followed by Chapter 5, where we evaluate this implementation, discuss the results obtained and analyze which types of sketch exhibit better performance and in which situations. Finally, we conclude in Chapter 6 with some final remarks about our work and insights into future work.

[2] https://github.com/Vivokas20/SKEL

Chapter 2

Background

In this section, we present fundamental concepts necessary to understand the rest of the document. Firstly, we introduce two ground concepts that are extensively used in state-of-the-art program synthesizers: Propositional Satisfiability in section 2.1 and Satisfiability Modulo Theories in section 2.2. These concepts are followed by an introduction to the area of program synthesis in section 2.3. Finally, we have a brief explanation of the Programming by Example (PBE) concept in section 2.4 and Query Synthesis in section 2.5.

2.1 Propositional Satisfiability

To understand the Propositional Satisfiability (SAT) problem, we first need to introduce the literal, clause, and Conjunctive Normal Form (CNF) concepts. A literal is a Boolean variable, x, or its negation, $\neg x$. A clause is a disjunction of literals, and a CNF formula is a conjunction of clauses.

We can say that a clause is satisfied by an assignment if any of its literals are True. Furthermore, a CNF formula is satisfied by an assignment if all of the clauses in such formula are satisfied.

Given a CNF formula, the SAT problem consists of deciding whether or not an assignment that satisfies the formula exists (satisfiable) or prove that it cannot exist (unsatisfiable).

Example 3. *Given the CNF formula* $(x_1 \vee x_2) \wedge (\neg x_1 \vee x_2)$ *we can see that it is satisfiable because there exists at least one assignment that satisfies it, such as* $\{x_1 \mapsto true, x_2 \mapsto true\}$. *On the other hand, this formula* $(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$ *is unsatisfiable (UNSAT) since there exists no assignment that satisfies it.*

2.2 Satisfiability Modulo Theories

Firstly, we introduce ground concepts necessary to understand this section. A theory is a set of axioms in the underlying logic. Several theories exist, such as the theory of Linear Integer Arithmetic (LIA) or the theory of uninterpreted functions. Considering 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 or its negation and a $\mathcal{T}$-formula is composed of $\mathcal{T}$-literals.

The Satisfiability Modulo Theory (SMT) problem is a generalization of the SAT problem in which instead of Boolean literals, $\mathcal{T}$-atoms may appear. For example, in LIA, these $\mathcal{T}$-atoms can represent equalities/inequalities. SMT formulas allow a problem to be modeled in a more expressive way than using SAT formulas. It checks the satisfiability of first-order logic formulas restricted to some background theory; it is also possible to combine different theories.

The SMT problem [3] focuses on models that belong to a given theory that constrains the interpretation of the symbols in the $\mathcal{T}$-formula. It consists of finding an assignment to the variables in a given formula. If this assignment exists, then the formula is satisfiable. If not, then it needs to prove that such an assignment does not exist.

Example 4. *Considering that $\mathcal{T}$ is the LIA theory. The formula $(x > z^2) \wedge (z > 1)$ is an example of an SMT formula in LIA, where x and z are integers. We can see that there exists an assignment $\{x \mapsto 5, z \mapsto 2\}$ that satisfies this formula. On the other hand, if we instead considered the formula $x + y = 2 \wedge y < x \wedge x = 1$ then, no assignment exists that satisfies this formula.*

2.3 Program Synthesis

Program synthesis consists of automatically finding a program, written in a given Domain Specific Language (DSL) (see Definition 1), that satisfies a given specification (user intent). This well-studied problem is mostly characterized by three main components: intent specification, program space, and search technique [12]. A general architecture of this problem's formulation is represented in Figure 2.1.

Definition 1. *The DSL represents the language in which synthesized programs are written by defining its syntax and semantics.*

To ensure an efficient and effective search, the choice of a DSL must take into account several factors such as:

- **Expressivity** that has to be broad enough to represent a wide variety of tasks but restricted enough to allow an efficient search.
- **Naturalness** since a more natural DSL is more user-friendly, potentially increasing the user confidence in the system.

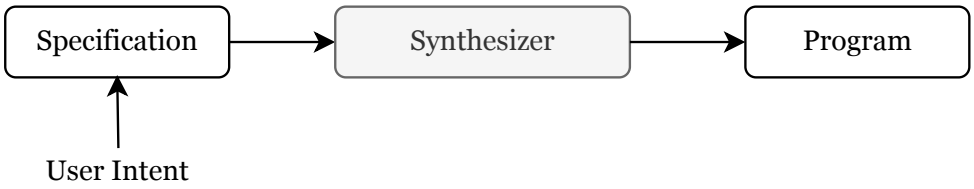


Figure 2.1: Program Synthesis general architecture

- **Efficiency** to enable the synthesis process in this language to be the most efficient possible - this can be achieved, for example, with the choice of operators.

Program synthesis has been a rapidly growing research area since the sixties and is considered the "Holy Grail" of Computer Science [14]. The underlying motivation of this rapid growth revolves around the fact that if we could only "tell" the computer what we want and let it generate a program that solves the proposed problem, the programming task would be simplified to a great extent. This simplification is one of the main reasons why the program synthesis area is considered "one of the most central problems in the theory of programming" [24] as Amir Pnueli described it.

2.3.1 Challenges

Regardless of all the recent improvements into automating the task of programming, program synthesis faces some challenges that researchers are continually trying to overcome by improving existing methods or creating new ones [7]. The two main challenges are:

1. **Program Space** - For Turing-complete programming languages, program synthesis is undecidable, which consequently leads to an infinite number of generated candidate programs. Program space is the space containing the set of all candidate programs that may be written with a given DSL. In the synthesis approach, we need to perform a search over the program space. The number of programs generated in any non-trivial programming language will quickly grow exponentially with the program's size, which makes the task of enumerating all possible programs intractable. For this reason, areas that study how to improve large program space exploration, such as pruning the program space, are an active field of research in program synthesis [14].

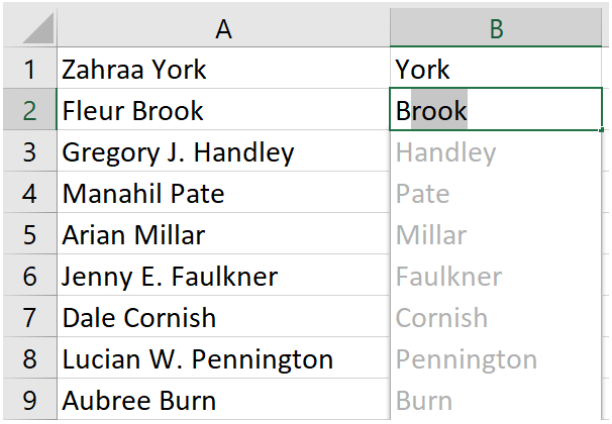
2. **User Intent** - It is not easy to accurately express with completeness and interpret without ambiguity the user intent. To express user intent, different methods were developed, such as logical specifications [20], natural-language descriptions [36], or input-output examples [38], but none of these is necessarily "perfect". Less formal specifications (e.g., input-output examples) are usually more accessible for the user to provide but are typically ambiguous since they are an incomplete specification. Nonetheless, more formal specifications (e.g., logical specifications) have the drawback

of not being user-friendly, sometimes even appearing more complicated for the user to provide than writing the program itself. On the other hand, these specifications are complete, hence more accurate and less ambiguous.

2.3.2 Search Techniques

There exist four main search techniques commonly used in program synthesis: deduction, constraint solving, enumerative search, and statistical techniques. However, it is also possible to use a combination of these techniques in order to yield a better performance.

- **Deduction** Follows a top-down search using a divide-and-conquer technique to recursively divide the problem into simpler sub-problems, combining the respective results in order to create a solution for the original problem [17].
- **Constraint Solving** This approach can be divided into constraint generation and constraint resolution [1].
  - Constraint generation is the process of generating a logical constraint whose model can be used to extract a program that satisfies the given specification.
  - Constraint resolution is the phase where the previously obtained constraints are solved using a constraint solver such as SAT or SMT solvers.
- **Enumerative Search** Consists of the enumeration of programs in the search space. First, it sorts the programs following a given heuristic, and then it checks, in order, if the candidate program satisfies the specification [2].
- **Statistical Techniques** These techniques use probabilistic models in order to create a solution for the original problem. There are many different statistical techniques, such as machine learning, genetic programming, and Markov chain Monte Carlo (MCMC) sampling.
  - In order to improve other search methodologies, machine learning can be used to provide the likelihoods of several possibilities when a choice needs to be made. Such choices can include the selection of a non-terminal symbol from the grammar or the most accurate program to return to the user [9].
  - Genetic programming is inspired by biological evolution [18]. This technique tries to apply to the context of programming the evolution process of populations, namely crossovers - the sharing of genetic material between individuals and mutations - random changes in the genetics of some individuals, which contributes to the diversity of populations. In genetic programming, the random changes that appear in programs (mutations) and the sharing of



	A	B
1	Zahraa York	York
2	Fleur Brook	Brook
3	Gregory J. Handley	Handley
4	Manahil Pate	Pate
5	Arian Millar	Millar
6	Jenny E. Faulkner	Faulkner
7	Dale Cornish	Cornish
8	Lucian W. Pennington	Pennington
9	Aubree Burn	Burn

Figure 2.2: Flash Fill example, in Excel version 2010

pieces of code between programs (crossover) are evaluated in order to create new programs that are then classified by a user-defined fitness function. The success of this technique heavily relies on the chosen fitness function.

2.4 Programming by Example

PBE consists of using input-output examples as a specification for the synthesizer.

Using input-output examples naturally introduces some ambiguity since these are an under-specification of the desired behavior. On the other hand, it also has the benefit that users do not need to know how to code because they can informally describe the desired behavior without the need to understand the underlying programming language.

PBE has many applications, such as generating SQL queries [34, 38] or the FlashFill PBE [13] tool used in Microsoft Excel. This tool is a program synthesizer for string manipulation that is able to auto-complete columns given that the user provides a set of examples. An example of Flash Fill being used to automate the task of writing in column B all the last names presented in column A is illustrated in Figure 2.2. We can see that the tool accurately captured the user intent after analyzing the few lines written and automatically suggests the completion of the task to the user.

PBE represents a possibility of how to capture user intent in the context of program synthesis. Its main advantage is the easiness with which the user can provide some examples instead of a complete formal specification. Since the developers are also more prone to make mistakes when implementing repetitive and cumbersome tasks, this technique helps to mitigate such issues that could arise.

One of the main disadvantages is the introduction of ambiguity that can usually lead to the generation of several valid programs with different semantics since all these programs satisfy the input-output examples given. For this reason, an additional challenge arises when using PBE, which is to not only

find a program that is valid under the given specification but also to choose the program that is most likely to implement the intended behavior in the space of candidate programs.

Another drawback of PBE is that it is cumbersome for the user to provide an extensive set of examples in real-life problems, and often only a small set is provided. Expecting the users to provide a large set of examples is not realistic, so the usability of PBE is often defined by the small number of examples needed to understand the user intent. If a more extensive set of examples were provided, the specification would be more refined, decreasing ambiguity, which could further prune the candidate programs. However, since, frequently, only one input-output example is given, we have to use some techniques to resolve the ambiguity. These techniques usually rely on ranking the candidate programs by assigning each of them a likelihood that measures if a program is the one the user desires and further user interaction.

2.5 Query Synthesis

Query Synthesis is a subfield of program synthesis where the goal is to automatically generate an SQL query that satisfies user intent.

It is often the case that domain experts need to perform complex manipulations on tabular data. They can easily provide an informal description of the task they wish to perform, but lack the necessary knowledge of SQL to implement this themselves. With Query Synthesis, these users are able to create queries that satisfy its specification.

One big challenge with Query Synthesis is to manage nested query generation, i.e., queries within other queries, since the program space often scales exponentially in these situations. Another challenge is the use of aggregated queries, such as `MIN`, `MAX` or `SUM`, since these are not trivially inferred by the synthesizer. This challenge is fundamentally essential to overcome since aggregated queries are commonplace in real-world scenarios [28].

The Query Synthesis problem can be seen as a decision problem where the goal is to find an SQL query that satisfies a given specification. This specification is most commonly given in the form of input-output examples [38] or natural language descriptions [9, 36]. We focus on Query Reverse Engineering (QRE), a special case of Query Synthesis that relies on input-output examples.

Query Reverse Engineering

QRE is a subfield of PBE that focuses on the manipulation of tabular data. In machine learning or cloud computing areas, the users are often required to manipulate large amounts of data. This manipulation becomes very complex if the user is not a data science expert. QRE has several applications such as

name	id	grade
John Smith	1	19
Mary Johnson	2	16
Dale Cornish	3	18
Aubree Burn	4	19
Arian Millar	5	12
Adam Handley	6	19
Robbert Grant	7	15
Peter Mayers	8	17

(a) students

```
SELECT students.name,
       MAX(students.grade)
FROM students
WHERE students.id > 3
GROUP BY students.id
```

(b) Query generated

name	max_grade
Aubree Burn	19
Adam Handley	19

(c) Output_table

Figure 2.3: QRE example

database usability, data analysis, and data security [32], allowing an improvement in the user's overall experience.

In Figure 2.3, we have an example of a QRE application. Given an example of input tables and the respective expected output, the synthesizer generates the query that, applied to the given input tables, returns the expected output.

The examples provided by the user are useful when we have large data sets of information that we want to manipulate. Through the query generation from a simple input-output example, we can then apply that query to the larger data sets [16].

Chapter 3

Related Work

The SQL synthesis problem is an active research area. In this section, we start by describing state-of-the-art techniques for program and query synthesis that have been proposed in the literature. These include Counterexample Guided Inductive Synthesis in section 3.1, Sketching in section 3.2 and Conflict-Driven Learning in section 3.3.

Finally, in section 3.4, we discuss existing SQL synthesizers, including state-of-the-art synthesizers: SQLSynthesizer, Scythe, SQUARES, and CUBES. In the end, we also compare the main advantages and drawbacks of these solutions.

3.1 Counterexample Guided Inductive Synthesis

The key idea of Counterexample Guided Inductive Synthesis (CEGIS) [31] is that we can often find a small set of inputs that generally define a problem, i.e., if we find a program that satisfies the specification for those inputs, that program will also satisfy it for all inputs. This allows an efficient exploration of the search space since it focuses on satisfying the specification for a small but sufficient set of input-output examples.

The core of CEGIS is a constraint-based inductive synthesis procedure, it combines the synthesizer (program generator) with an automated validation procedure (verifier) to help the synthesizer produce a correct answer.

As illustrated in Figure 3.1, by providing an initial input for which the specification needs to be satisfied a program is generated that is then given to the verifier to decide whether or not that candidate is correct, i.e., if it does not violate any specification. If it is correct, the synthesizer has found an appropriate program that satisfies the specification. Otherwise, the new input is a counterexample which is returned to the program generator (feedback) and the new program generated must satisfy the specification for both the initial input and the new counterexample, giving the program generator more accurate

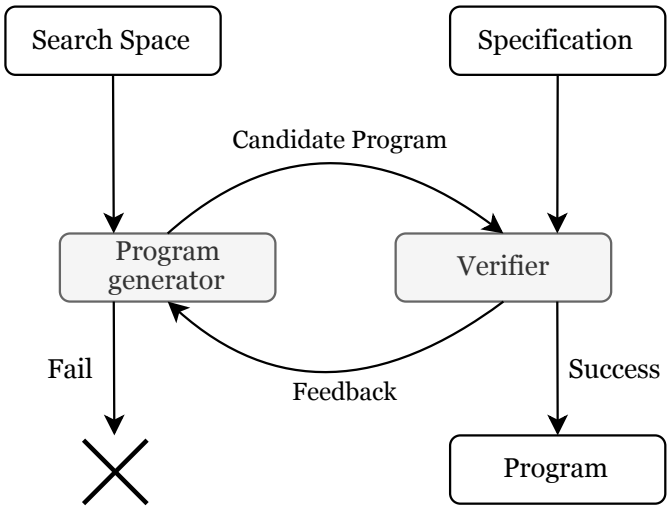


Figure 3.1: Counterexample Guided Inductive Synthesis

information. This process is repeated until a program is generated.

The main advantage of this technique is that it allows to fully constrain a solution from a small set of inputs, allowing a more effective pruning.

3.2 Sketching

Sketching is an approach that has become very popular in program synthesis. It was first introduced by Solar-Lezama [29] and allowed programmers to provide as a specification a skeleton that expresses the high-level specification about a program they want to create.

A sketch consists of a partial program, which represents the high-level structure of the intended program with holes that represent the low-level details, such as arguments or variables names. Besides the sketch, programmers should also provide either a reference implementation or a set of tests the code must pass for the synthesizer to make the final assertions and ensure the program satisfies the specification of the intended behavior.

This approach's main advantage is that it makes synthesis accessible to programmers by allowing them to provide insights easily without needing to learn, for example, a specification language or theorem proving. This accessibility contributes to an improvement in the productivity and performance of programming tasks.

The SKETCH syntax is very similar to the C language with the additional symbol ?? representing the holes in the sketch. This syntax is illustrated in Example 5.

The synthesizer relies on CEGIS [30] to find the correct values to fill all the holes in the sketch and avoid assertion failures. It works by trying to replace the holes in the sketch from a finite set of choices

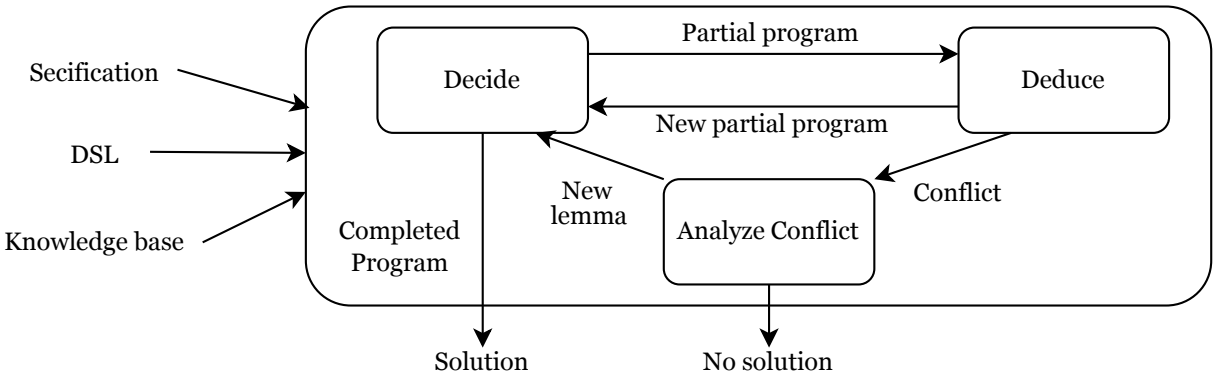


Figure 3.2: High-level architecture of NEO

in a way that satisfies the provided specification. In Example 5, the synthesizer will eventually replace the ?? symbol with a 2, thus satisfying the assertion statement.

Example 5. *Basic sketch structure:*

```
int main(int x){
    int y = x * ??;
    assert y = x + x;
}
```

3.3 Conflict-Driven Learning

For each counterexample produced by the CEGIS synthesizer, the synthesizer can try to find the root cause of the failure. Using this information, we can understand the core reason why a given candidate program is incorrect, i.e., violates the specification, and, consequently, use it to further prune the search space by blocking multiple incorrect programs at once.

NEO [10] is a synthesizer that uses conflict-driven learning to understand its past mistakes and improve search time by continually reducing the search space. Figure 3.2 illustrates the high-level architecture of NEO. We continue with a brief explanation the three components of this synthesizer.

3.3.1 Decide

In this phase, NEO starts with an empty partial program with holes that need to be filled. At each step, the decide component chooses a hole to be filled and determines how to fill it using the DSL constructs. The assignments made in this phase must satisfy any lemmas that have been added to the knowledge base. This knowledge base keeps track of any useful lemmas learned during the execution of the algorithm.

3.3.2 Deduce

For each partial program generated, the deduce component checks, using the knowledge base, if this is a feasible partial program, i.e., it checks the SMT formula's satisfiability that is the conjunction of the specification provided by the user and the partial program's specification. This verification can result in the possibility of the partial program to be further extended without violating the specification or that it is not possible to further extend the partial program without violating the specification, i.e., a conflict is detected. In the first case, the tool returns to the decide phase with no new lemmas learned, or it can deduce some holes with non-terminal symbols using the specific rules from the DSL. In this case, a new partial program with these changes is returned to the decide phase. On the other hand, if a conflict is detected, the control is passed to the analyze conflict phase.

3.3.3 Analyze Conflict

This phase aims to identify the root cause of failure and use it to learn new lemmas that are added to the knowledge base. These lemmas are additional conditions that a partial program must satisfy in order for there to exist a feasible concretization of that program. With this, the synthesizer gains the ability to learn from previous mistakes, preventing the algorithm from making the same bad decisions in the future. Therefore, this technique can increase the search speed throughout the search space by enabling a more aggressive pruning.

3.4 SQL Synthesizers

In this section, we describe SQL synthesizers ordered by their release date in order to enable a better understanding of how the state of the art evolved throughout the years. Finally, we summarize this section by comparing these synthesizers' main advantages and disadvantages.

3.4.1 SQLSynthesizer

SQLSynthesizer's [38] main features are that it is fully automated, in contrast with previous SQL synthesizers, such as Query By Synthesis [6] and Query-by-Example [39] and it supports a larger range of SQL constructs such as table joins, aggregations and the `GROUP BY`, `ORDER BY` and `HAVING` clauses. This tool is composed of three main phases: Skeleton Creation, Query Completion, and Candidate Ranking.

Definition 2 (Query Skeleton)**.** *A Query Skeleton is a partial SQL query with a defined structure that has holes which need to be completed.*

Skeleton Creation By analyzing the input-output examples given, SQLSynthesizer creates a set of query skeletons (see Definition 2) that capture the underlying structure of the intended query. To do so, SQLSynthesizer uses three heuristics:

1. Determining query tables: If the same column from an input table appears more than once in the output table, the input table will likely be used multiple times in the query. SQLSynthesizer replicates that input table N times in the query table set. With N being the number of times the column appeared in the output table.

2. Determining join conditions: Instead of exhaustively enumerating all the different possibilities for joining the queried tables, SQLSynthesizer uses two rules and applies them repeatedly. The first is to join tables on columns with the same name and data type. If such columns do not exist, it uses the second rule that joins tables on columns with the same data type.

3. Determining projection columns: If there exists a column in the input table with the same name as a column in the output table, SQLSynthesizer uses the first matched column from the input table as the projection column, i.e., a column the query will return. Otherwise, it infers that the column in the output must be the result of an aggregation.

Query Completion Using a machine learning algorithm, SQLSynthesizer infers the missing parts in each query skeleton and builds a list of candidate queries that satisfy the given input-output examples. To do so, it divides the problem into three substeps:

1. Inferring Query Conditions - The result tuples created by possible join conditions inferred during skeleton creation constitute the search space. In this substep, the goal is to correctly divide the search space into a positive (tuples that contain the output table) and a negative part (the remaining tuples). Usually, tuple values from the input table do not have enough information about the relations between tables. To mitigate this problem, two additional features were added. The first is aggregation features, which state that the synthesizer groups all tuples by value for each column in the joined table and then applies every possible aggregator to all of the other columns and stores the aggregation's result. The second is comparison features, which state that for every tuple, the system compares the values between two type-compatible columns and stores the results as 1 or 0. In order to learn a set of rules as query conditions, SQLSynthesizer uses a variant of the decision tree algorithm, called PART [11] that has a divide and conquer strategy. This enables a faster and less memory consuming process than the original decision tree algorithm.

2. Searching for aggregations - If a column in the output table does not match with a column in the input tables, SQLSynthesizer searches for an aggregate in the input table columns that produce the output table column. To speed-up this process, two sound heuristics were implemented. The

first is to only apply aggregators to type-compatible columns, which means that the returned type of the aggregation must be consistent with the value type of the output column. Furthermore, the second checks if each value in the output column exists in the input column. If not, then we know that `MAX` or `MIN` aggregate cannot be used. These heuristics help in pruning the search space.

3. Searching for columns for the `ORDER BY` clause - If the data values of an output table column are sorted, SQLSynthesizer appends that column name to the `ORDER BY` clause. The authors do not reference how they handle multiple columns in the `ORDER BY` clause.

Candidate Ranking When there is more than one SQL query that satisfies the given input-output examples, the queries are ranked using the Occam's razor principle, which states that the correct explanation is usually the simplest one. This ranking means that the simplest queries will be ranked higher than the complex ones.

In the end, if the output query does not satisfy the user's intent, the user can provide more specific examples and reapply SQLSynthesizer.

3.4.2 Scythe

Scythe [34] is considered a state-of-the-art SQL synthesizer. Besides receiving only one input-output example, Scythe can also receive a set of constants that must appear in the generated query.

Definition 3 (Abstract Queries)**.** *Abstract queries are syntactically similar to SQL queries. However, instead of the filter predicates, they contain holes, □, that can be instantiated with any valid predicate.*

Example 6. *Simple abstract query:*

```
SELECT name
FROM contacts
WHERE □
```

This approach introduces the language of abstract queries (see Definition 3) and decomposes the SQL synthesis problem into two phases.

The first phase consists of synthesizing all abstract queries that can potentially be instantiated into concrete SQL queries and that are consistent with the input-output examples provided by the user, thus pruning away the rest. This synthesis is done using an enumerative search.

Once all candidate abstract queries are identified, the second phase starts by searching for predicates to fill the holes in these abstract queries, instantiating them into concrete ones, and determines which ones are consistent with the examples. Scythe implements two optimizations in this search process. The first consists of locally grouping candidate predicates into equivalence classes (e.g., if we have an integer attribute id that does not contain the value 5 (five), then "$id > 5$" and "$id >= 5$" are

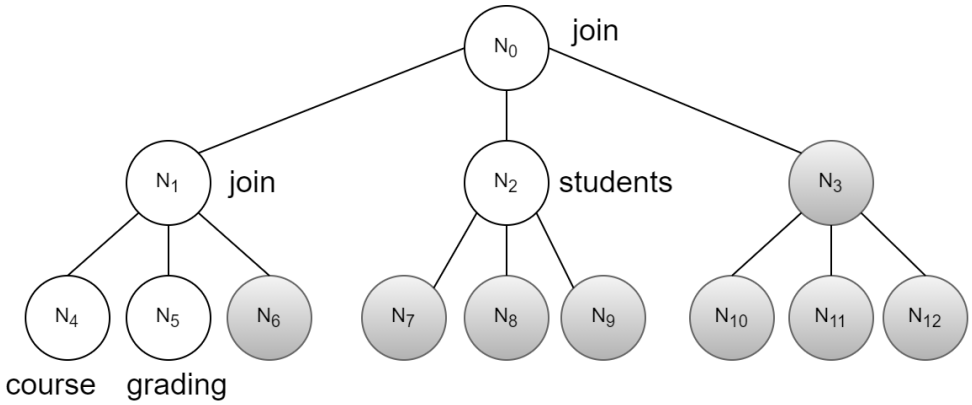


Figure 3.3: K-tree representation example

equivalent). The second optimization is to encode tables using bit-vector where the *n*-th bit of a bit-vector represents whether the row *n* in the candidate query appears in the corresponding input table. These bit-vectors are then used to improve efficiency by allowing the tool to verify more quickly if a set of rows are part of a given table.

In the end, it ranks the candidate queries according to their simplicity, predicate naturalness, and constant coverage, and it returns the top candidates to the user. If none of the candidates satisfy the user's intent, the user can rerun the system by providing new examples or aggregation functions for the query to use.

This solution has the bottleneck that the number of tables enumerated at each stage is exponential regarding the input tables' size.

3.4.3 SQUARES

SQUARES [22] is built on top of the Trinity framework [21]. As a specification, the user has to provide an input-output example. However, this synthesizer can also receive as input extra information regarding the intended query, including:

- a list of aggregation functions;
- a list of constants;
- columns names that must be used as arguments to aggregators.

A common approach to enumerate all feasible programs is a tree-based encoding, which uses a k-tree, i.e., a tree where each node has precisely k children, except the leaves, where k is also the largest number of parameters of any given component of the grammar. This k-tree is then encoded into an SMT formula in order to generate the candidate programs. In Figure 3.3 we have an example of the k-tree

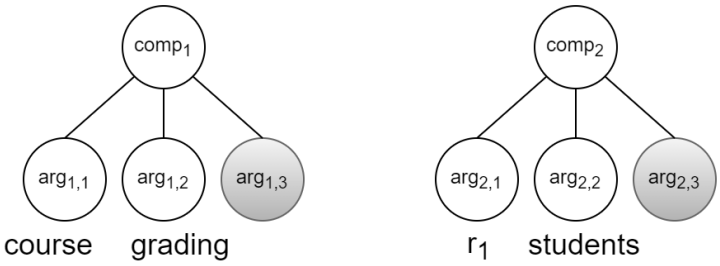


Figure 3.4: Line-based representation example

representation of the query $join(join(course, grading), students)$. The problem with this encoding is that it grows the number of nodes exponentially with the tree's maximum depth, which quickly becomes intractable to generate larger programs. To mitigate this intractability, SQUARES introduces the concept of line-based representation [23].

In line-based representation, a program is represented as a sequence of lines where each line corresponds to a DSL operation. Instead of using only one k-tree, as in tree-based encoding, the goal is to represent a program where each line corresponds to a tree of depth of 1. With this implementation, the authors' goal is to reduce the overhead of SMT solving. In Figure 3.4 we can see the line-based representation for the previous example where the nodes reduction is notorious.

Let l be the number of lines of code in a program and k the largest number of parameters of any given component of the DSL. Then we can analyze the complexity of the number of nodes in both encodings.

$$\frac{k^{l+1}-1}{k-1} \tag{3.1}$$

$$(k+1)*l \tag{3.2}$$

We can observe that in tree-based encoding, the number of nodes increases exponentially with the number of lines of code (see Equation 3.1), which contrasts with the linear increase of the line-based encoding (see Equation 3.2). This simplification is because the line-based encoding uses l trees, with $k+1$ nodes each, to represent a program with l lines, while in tree-based enumeration, a k-tree of depth $l+1$ is used. It was also expected that, besides the number of nodes, the number of necessary variables and constraints of the respective SMT encodings would also be significantly smaller.

The final results showed that this implementation could solve a higher number of benchmarks in less time than the tree-based encoding and was far less complex in terms of the number of nodes. It is also relevant to note that an optimization was done to block symmetric programs when a given program is determined as not being a solution by the synthesizer. However, for the benchmarks used to evaluate the solution, this optimization did not have a significant impact on the results.

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

Figure 3.5: DSL used by the CUBES synthesizer

3.4.4 CUBES

CUBES [5] was built on top of SQUARES, so every aspect described in the previous subsection, from the specification to the generation of the trees, also applies to CUBES. The key benefit of this implementation is not only to take advantage of modern-age computers' multi-core processing to speed-up the synthesis process but also greatly improve the performance of SQUARES with the introduction of bit-vectors.

This synthesizer has three different versions: a sequential and two parallel solutions. Regarding the parallel solutions, one uses a portfolio technique, and the other uses a divide-and-conquer technique. We further explain each of these implementations.

Sequential Synthesis

CUBES-Seq is a sequential SQL synthesizer. It extends the original SQUARES DSL in order to be more expressive, thus supporting a wider range of queries. It also removes the `SELECT` component from the DSL introducing it as a post-processing step in order to improve efficiency. An example of the Cubes DSL is given in Figure 3.5.

CUBES also introduces a new form of pruning that annotates all component arguments with a pair of sets of columns in bit-vectors. These new formulas are added to the SMT solver in the form of constraints, forcing consistency regarding the columns. This optimization was necessary since, with the previous encoding, many valid SQL queries could be generated that, for example, performed a natural join of tables without matching columns. With the new encoding, if a query is generated using some table T and given some constraint on column C, then, if T does not contain C, that query is removed,

Name	Id
John Smith	1
Mary Burns	2
Dale Cornish	3
Arian Millar	4
Peter Mayers	5

Table 3.1: Table Students

Id	Grade
1	18
2	15
3	14
4	16
5	17

Table 3.2: Table Grading

as shown in Example 7. This optimization helps to limit the search space and, consequently, improve the search time.

Example 7. *Consider Table 3.1, and Table 3.2, we could never generate a program with the operation* $filter(Students, Grade == 18)$ *since Table 3.1 does not contain the column "Grade", only the Table 3.2 does, so this program would be immediately pruned.*

The authors observed that, given the same amount of time, the number of instances solved by CUBES-Seq was much higher than other previous state-of-the-art synthesizers such as Scythe and SQUARES.

Parallel Synthesis

To improve CUBES-Seq solving time, the authors tried to use parallel programming, thus taking advantage of the several cores that modern computers have. Two parallel SQL Synthesizers were implemented: CUBES-Port, which uses portfolio solving, and CUBES-DC, which uses a divide and conquer technique.

Portfolio solving The goal of CUBES-Port is to diversify the exploration of the search space. To do so, it tries to force each thread to explore the same search space in different ways, as the portfolio technique states, and as soon as a solution is found, the search ends. The diversification of search space exploration can be achieved by using the same solver but with different configurations or applying a set of solvers that use different search techniques.

Divide and Conquer The main idea in CUBES-DC is to split the problem into smaller sub-problems, cubes, that can be solved by each of the processors. To achieve this, it divides the search space into different cubes that constitute a smaller search space to iterate over. Each cube corresponds to particular sequences of operations from the DSL in which the arguments for those operations are still undetermined. Then, in order to take advantage of the multi-core architecture, the cubes are distributed over

different threads. In the following paragraphs, we explain two different techniques that were implemented for generating the cubes.

Static Cube Generation Technique where the cubes are constructed using a static heuristic. This heuristic follows a predetermined order in which the operations are chosen. It selects the next operation to be executed, taking into account which operations were already selected.

Dynamic Cube Generation Technique that is inspired by Natural Language Processing (NLP) techniques. In this technique, each operation is given a likelihood based on the immediately preceding operation. These likelihoods are updated as candidate queries are evaluated. The operations in a given candidate query are scored higher if all the values in the expected output example occur in the query's output table.

Two measures were implemented in order to avoid biases in the cube generation induced by the scoring system. The first is that each time a new query is generated, all scores are multiplied by n, with $n \in [0, 1[$. This allows the tool to gradually forget about past information. The second measure is to solve randomly generated cubes by a fixed number of processes in order to diversify the search process.

To overcome the overhead introduced by complex operations, CUBES splits the available processes into two sets: one only attempting to solve programs that contain at least one of the `inner_join` or `cross_join` operations and the other attempting to solve programs as these two operations did not exist.

CUBES enumerates programs in increasing size so, if it knows that the intended query requires at least n lines, it starts the search by evaluating candidate queries that are at least of size n to avoid wasting time. However, by using parallel solving, CUBES splits the search space, and, consequently, it can first find a solution of size $n+1$ while the search for smaller solutions has not yet been completed. For this reason, CUBES provides two options: do an optimal or a non-optimal synthesis. The first continues to search for solutions smaller than n while the latter simply returns the first solution found.

Since the CUBES implementation uses SMT solvers, it can also take advantage of UNSAT cores to prune the search space even further. Every time a cube fails by not producing at least one valid candidate query, the UNSAT core is used to prune all the other cubes that would also fail for the same reason, according to that core.

3.4.5 Summary

To conclude this analysis of SQL synthesizers, we summarize the improvements of each of the presented tools in order to better understand its advantages and disadvantages.

SQLSynthesizer was the first fully automated synthesizer in this area and introduced queries that include table joins, aggregations, and the `GROUP BY`, `ORDER BY` and `HAVING` clauses.

On the other hand, Scythe also supports `LEFT JOIN`, `UNION` and `EXISTS` clauses and free-form subquery nesting, thus supporting a wider range of SQL features used in practical scenarios. It also scales better due to the two-phase synthesis algorithm.

SQUARES can solve as many instances as Scythe, yet, since it uses a line-based encoding instead of a decision tree algorithm, the memory space needed is much smaller than previous encodings. The generated queries are also easier to read and understand.

CUBES uses bit-vectors to increase the synthesizer's efficiency. It also supports `INNER JOIN`, `CROSS JOIN` and `SEMI JOIN` clauses, although its predominant feature is the fact that it supports parallel programming, thus making the best use of the multi-core architecture of modern computers.

Chapter 4

Sketch-Driven SQL Synthesis

In this chapter, we describe SKEL, a **Ske**tch-driven SQ**L** synthesizer developed on top of the sequential version of CUBES [5]. SKEL's goal is to generate an SQL query given an input-output example and an incomplete sketch of the query we want to generate. A sketch consists of a partial program with holes (missing information) that represent details that need to be filled by the synthesizer.

In an initial approach, our goal was to support sketches written in an SQL format with holes. However, after analyzing CUBES design, we concluded that it would not be a viable option to implement the sketches in that format. This is because the CUBES design has the DSL written in a format similar to R [19] (as shown in Figure 3.5), and doing the translation of an SQL sketch to R was too cumbersome to implement and was out of the scope for this work. Therefore we decided to change our approach to have a sketch that followed the CUBES' DSL. This sketch format is described in detail in section 4.1.

SKEL was implemented in Python 3.8 and R version 3.6.3. R was used to validate the program generated, using the input-output example, and then translate it to a program in SQL. SKEL also uses the Z3 SMT solver [8] to solve the SMT formulas generated by the synthesizer.

Figure 4.1 shows an overview of our tool's architecture. We can see that SKEL receives as specification a file that must contain the input-output tables, and also a sketch in the correct format. Similarly to CUBES, the specification can also contain constants, aggregate functions, columns, and Lines of Code (LOC), but SKEL provides a configuration parameter named `generate_sketch_dsl` that enables the grammar to be generated from the DSL according to the sketch productions, i.e., every sketch rule. This configuration is shown in the architecture and is later explained in more detail in section 4.2 where we also describe how we generate our sketch structure.

After parsing the sketch and generating the grammar, we can then map the DSL identifiers to the sketch productions. In section 4.3 we explain how this mapping is performed and the optimizations done in order to improve it.

Next, a tree representation of the sketch is provided to the Tree Enumerator that generates all pos-

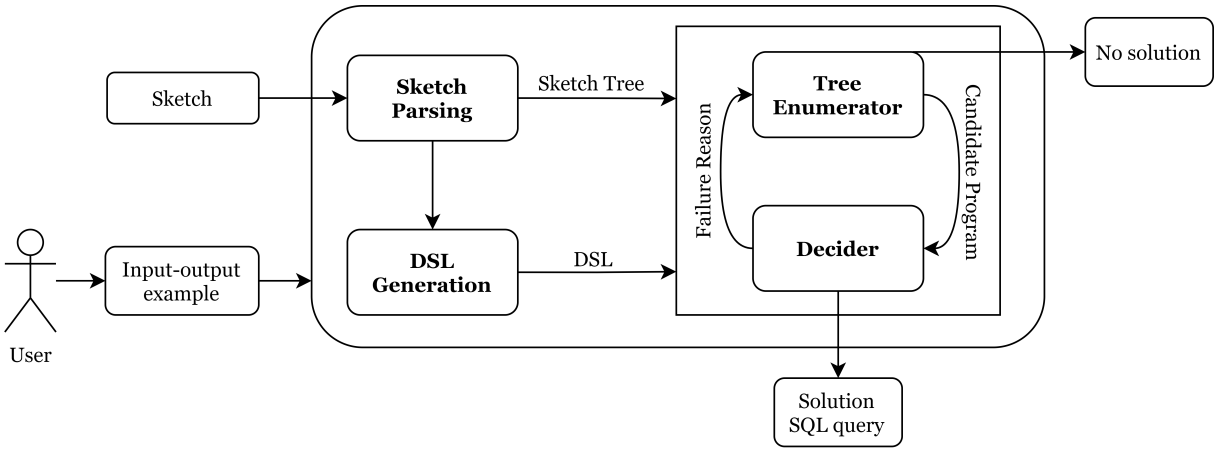


Figure 4.1: SKEL's architecture

sible programs according to the sketch structure. We describe this phase in more detail in section 4.4 where we specify the constraints generated by the synthesizer in order to comply with the given sketch.

In the next step, the synthesizer sends the generated program to the decider to test if the program satisfies the input-output example. If it does, then a solution has been found; if not, the decider provides information about the failure reason, such as the incorrect number of rows in the solution, and keeps enumerating programs with the newly added information.

4.1 Sketch Format

In order to run SKEL, the user needs to provide a specification file similar to the one that is provided in CUBES, augmented with an additional input parameter: the sketch. In this section, we thoroughly describe the format of the sketches that must be provided in the specification and illustrate all the different types of holes supported by the synthesizer.

The sketch is composed of one or more lines that can be either a hole or take the form $line\_name = Function$. Each line that is not a hole has a $line\_name$, i.e., a name with which that line could be referenced, and a $Function$, i.e., the name of the operation we want to perform (parent) and the corresponding input arguments (children). In Figure 4.2 we represent the functions and types of children the sketch supports, where the optional input arguments are represented in brackets. To avoid ambiguity, no column and table can have the same name. For future reference, we note that a $summariseCondition$ is the same as an aggregate function, such as `MAX`, `MIN`, `COUNT`, `SUM`, `AVG`.

It is also relevant to note that if we have more than one condition or column, these must be inside parenthesis and separated by a comma as shown in bold in Example 8.

$\texttt{natural\_join}(table, table, [table], [table])$
$\texttt{inner\_join}(table, table, joinCondition)$
$\texttt{anti\_join}(table, table, [column(s)])$
$\texttt{left\_join}(table, table)$
$\texttt{union}(table, table)$
$\texttt{intersect}(table, table, column)$
$\texttt{semi\_join}(table, table)$
$\texttt{cross\_join}(table, table, crossJoinCondition)$
$\texttt{filter}(table, filterCondition)$
$\texttt{summarise}(table, summariseCondition, [column(s)])$
$\texttt{mutate}(table, summariseCondition)$

Figure 4.2: Sketch Functions

Example 8. *Multiple columns:*

$$\texttt{summarise}\ (T1,\ max\_grade = max(grade),\ \boldsymbol{(id, name)})$$

If the user has information about the final SELECT columns of the query it intends to generate, then the final line of the sketch must represent the SELECT operation.

The table that is an input of this SELECT operation is always the final table generated by the synthesizer, i.e., the table with LOC identifier.

This optional final line takes $distinct$ and/or $orderby$ as optional arguments and selects the columns used in the output table:

$$out = \texttt{select}\ [\texttt{distinct}]\ (Column(s))\ [\texttt{order by}\ (Column(s))]$$

An important note is that this final line is optional, depending on the information the user has.

Finally, before introducing the several holes that the sketch supports, we present in Example 9 a possible representation of a query in our sketch language. This is going to be the basis for the following examples to demonstrate the types of sketch holes supported.

Example 9. *Representation of the SQL query (left) into a full sketch (right), i.e. a sketch with no holes:*

```
SELECT name, MAX(grade)
FROM students
WHERE id > 3
GROUP BY id
```

$$T1 = \texttt{filter}\ (students,\ id > 3)$$
$$T2 = \texttt{summarise}\ (T1,\ max\_grade = max(grade),\ id)$$
$$out = \texttt{select}\ (name,\ max\_grade)$$

Now that we have introduced the format of our sketch language, we describe which functionalities such sketches support in terms of holes. The main symbol that we use to represent a hole is $??$. The following variations are also supported: $??*$ corresponds to zero or more holes and $??+$ corresponds to one or more holes. These holes can be represented in the following ways:

- **Line hole** - It consists of a whole line replaced by a hole. The only exception is the `SELECT` line that must be completely omitted if the user cannot provide any information about it. If $??$ is used, it means that there is exactly one line in the respective position. However, if the user is unsure of the exact number of lines, he can also use the previously defined $??*$ or $??+$ variations. If the user knows that there is definitely a line in that position, he can also name the line to reference it in the sketch lines that follow. Example 10 shows two different uses of line holes. The first one (on the left side) corresponds to when the user references an unknown line, and the second one (on the right) to when he does not know the exact number of lines.

  Example 10. *Line hole:*

  $T1 =$ **??**
  $T2 =$ `summarise` $(T1,\ max\_grade = max(grade),\ id)$
  $out =$ `select` $(name,\ max\_grade)$

  $T1 =$ `filter` $(students,\ id > 3)$
  ??+
  $out =$ `select` $(name,\ max\_grade)$

- **Previous line hole** - It is also important to mention that in case the user knows that some line in the output query must use a previous sketch line, but there is no name for the line (due to the usage of, for example, $??+$), the user has a special semantics $T??$. This semantic represents a hole that must be filled by previously generated lines, thus giving the user more flexibility while writing a sketch. In Example 11 we show a possible usage of this type of hole.

  Example 11. *Previous line hole:*

  $??+$
  $T2 =$ `summarise` $(\boldsymbol{T??},\ max\_grade = max(grade),\ id)$
  $out =$ `select` $(name,\ max\_grade)$

- **Children holes** - Assuming that all children have a known parent function, then the user will also be aware of the type and format of the input arguments for that function. This way, the user can put holes, $??$, in the place of the input arguments he does not have information about, as shown in Example 12.

Example 12. *Children hole:*

$$T1 = \texttt{filter}\ (students,\ \texttt{??})$$
$$T2 = \texttt{summarise}\ (T1,\ max\_grade = max(grade),\ \texttt{??})$$
$$out = \texttt{select}\ (name,\ max\_grade)$$

- **Parent and children hole** - All parents can be replaced by a hole, $??$, which implies that the user may have less knowledge about the number of children for that line.

  If a parent is replaced by a hole, then we assume that the user does not have information about the order of the input arguments. To address this, we always consider the children to be out of order so that the user can freely provide any children that he has information on, despite not knowing the parent function.

  For this hole type, there are several possible scenarios:

  – The user knows all the children. In this case, the parent is represented with a $??$, and the user inserts the children regardless of the order.

  – The user knows how many children exist in the unknown function, but not all of their concrete expressions, and thus he replaces the ones he does not know with $??$ without taking the children's order into account. This scenario is illustrated in the first part of Example 13.

  – The user only knows some of the children without knowing how many exist in total. This is the most likely scenario, and it involves the user giving information about the children he knows, regardless of the order, and then inserting a $??+$ or $??*$ at the end, depending on the information he has. This is shown in the second part of Example 13.

Example 13. *Parent hole:*

$$T1 = \texttt{filter}\ (students,\ id > 3)$$
$$T2 = \texttt{??}\ (id,\ \texttt{??},\ \texttt{??})$$
$$out = \texttt{select}\ (name,\ max\_grade)$$

$$T1 = \texttt{filter}\ (students,\ id > 3)$$
$$T2 = \texttt{??}\ (max\_grade = max(grade),\ \texttt{??}+)$$
$$out = \texttt{select}\ (name,\ max\_grade)$$

- **Select hole** - The SELECT line is the optional last line of the sketch. Assuming this line exists in the sketch its holes follow particular rules:

  – The columns given to the SELECT function have to be exactly the columns the user intends its output to have; otherwise $??$ should be provided in the SELECT function.

- If the `distinct` option is not provided, the synthesizer assumes that it should not be used, but if the user is unsure, then $??$ may be provided in place of `distinct`.
- The `order by` option follows the same rule as the `distinct` option. However, in this case, when the `order by` is present, the user can also specify the columns or a hole, $??$, in case he does not know which columns to use to perform the ordering.

In Example 14 we illustrate some of these uses.

Example 14. *Select hole:*

$$T1 = \texttt{filter}\ (students,\ id > 3)$$
$$T2 = \texttt{summarise}\ (T1,\ max\_grade = max(grade),\ id)$$
$$out = \texttt{select}\ \textbf{??}\ (name,\ max\_grade)\ \texttt{order by}\ (\textbf{??})$$

- **Alternative hole** - It may be the case that the user knows that a given hole should be filled by one production of a small subset of parents or children. This can be done by providing a set of productions inside brackets, [ ], thus giving the synthesizer more information than it would have if the user only specified a hole with no additional details. In Example 15 we demonstrate a possible use of this functionality given three tables - stock, supplier, and parts.

Example 15. *Alternative hole:*

$$T1 = [\texttt{left\_join}, \texttt{semi\_join}]\ (stock, [supplier, parts])$$
$$out = \texttt{select distinct}\ (PartName)$$

4.2 Sketch Parsing

After receiving a sketch as a specification from the user in the format described in section 4.1, we start parsing it by creating the tree structure of the sketch. In this section, we will explain the tree structure and also how the synthesizer generates the grammar from the DSL using the sketch information.

4.2.1 Sketch Structure

Let l be the number of lines in the sketch, and k the maximum number of parameters of any given DSL component. The sketch structure consists of l trees of depth 1 with k leaves each. Figure 4.3 illustrates the tree structure for the sketch in Example 12.

Before describing more in-depth each one of the tree components we list the four main types of lines the sketch can have:

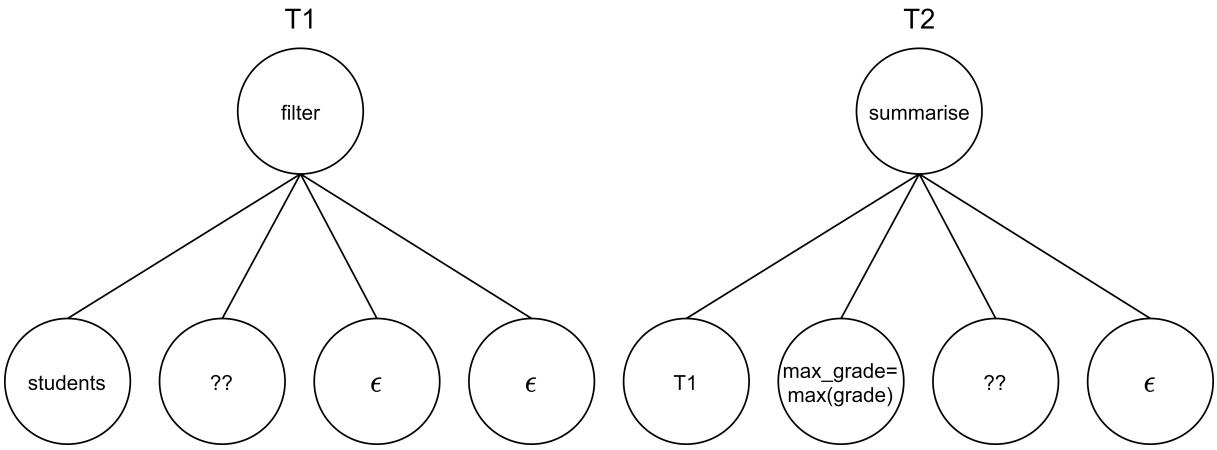


Figure 4.3: Sketch Tree Example

- **Normal line** - a line without a parent hole;
- **Incomplete line** - a line for which the function name is a hole, and we do not know the exact number of children or their order. For example, $T1 = ??\ (students,\ ??+)$ would be an incomplete line since it can have two or more children;
- **Unordered line** - a line for which the function name is a hole, but we know the exact number of children, although not their order. For example, $T1 = ??\ (students,\ ??)$ would be an unordered line since we know it has exactly two children but not their order since we do not know the parent function. The example $T1 = ??\ (id > 3,\ students)$ would also be an unordered line since, as the function is not specified, we cannot infer the order of the input arguments;
- **Empty line** - a line that represents an empty tree that is named. This way, the tree can be referenced in subsequent lines. For example, $T1 = ??$ would be an empty line with a tree named T1 that can be referenced in the following lines.

We now describe more in-depth each one of the tree components: the root and the leaves. Every tree root contains:

- the name of the function production(s) that can be used or a hole;
- the name of the line, so that it can be referenced in subsequent lines;
- the position of the line in the sketch;
- the type of line.

Currently, SKEL is using the same DSL as CUBES. Therefore, every tree has four leaves since the largest DSL production has four parameters. Each leaf contains information about:

- the name(s) of the production(s) allowed for that child, or a hole;
- the child type;

Besides the child types described in section 4.1, the sketch structure also allows as types the ”line type” or the ”unknown type”: The ”line type” corresponds to a table generated in a previous line, i.e., a previous root identifier. A child has an ”unknown type” when we do not know the function name, and we cannot infer the argument type while looking for matches in table names.

Select line In order to build the sketch structure, we start by parsing each line into this structure except the `SELECT` line. If a line starts with the keyword ”out”, it is the last line, i.e., the `SELECT` line. This last line, when present, is treated differently since the CUBES DSL does not contain the `SELECT` component, which was removed and introduced as a post-processing step to improve efficiency. To parse this line, we simply store which columns are present in the `SELECT` statement; if the `SELECT` statement must have a distinct; and if the table rows must be arranged in a specific order. Afterwards, we store this information in the specification, which is used by the decider component of the synthesizer to test the possible programs in order to find a valid solution.

LOC value In the sketch, we also consider the maximum and minimum LOC possible. SKEL follows the Occam's Razor principle, starting by generating trees using the minimum LOC for the given sketch and pruning all other smaller solutions. We then increase the LOC number by one after exhausting all the solutions for the current value. We do this until a solution is found or the maximum LOC is reached.

In the simplest cases, the LOC is the number of sketch lines. However, if a line has a $??+$ hole, the minimum LOC is increased by one, and the maximum LOC is removed since we do not know what the maximum number of lines the final solution will have. If a line has a $??*$ hole, then we only remove the value of the maximum LOC. If we do not have a maximum LOC, then all of the lines that succeed the hole are considered a free line since we do not know their exact position in the solution. The only guide on those lines is their id, which tells the synthesizer that it has to appear before or after another line.

4.2.2 DSL

After parsing the specification and constructing the sketch structure, the synthesizer generates the grammar from the DSL. This grammar will be used during synthesis. The DSL follows the same format as the CUBES DSL (see Figure 3.5). Each production of the DSL has a non-negative identifier, $id \mapsto IN_0$. It is relevant to note that the production with the 0 identifier represents the $empty$ production which is assigned to empty leaves. For example, if we consider the DSL used, the largest production has an arity of 4, so if a function only has three input arguments, the final leaf must have the $empty$ production

assigned since the trees are always generated with 4 children. This is later explained in more detail in section 4.3.

By making a change in the configuration, we can opt between having the grammar generated according to the constants, aggregates, and columns given in the specification (similarly to CUBES), or having the grammar generated from the sketch productions, with the option `generate_sketch_dsl`.

We provide the user with these two options since it is unfeasible to generate all possible productions that can be applied to a specific input-output example without severely compromising the efficiency. To improve efficiency, we limit the number of productions generated.

The option also used in CUBES can bring benefits when a user knows that he/she needs to use the aggregate function such as `MAX`, for example, but is unsure which column to perform that action. In this case, he cannot convey this information using a sketch since this is a limitation of the sketch language, and it does not allow a line to be, for example, $summarise(??, ?? = max(??), ??)$. Hence, the option used in CUBES becomes necessary.

To generate the grammar using the sketch productions, we store every aggregate, constant, and column while parsing the sketch. If a line contains the called function's name, this process is trivial since all the children have their respective types identified, and we can easily distinguish between aggregates, columns, and constants. However, if the function is unknown (a hole), the synthesizer has to analyze each child individually and infer which type of production it is.

The productions that correspond to the table and column names are always generated, so if a child's type is a table or a column, no further action is necessary. On the other hand, if the child type is not a table or a column, we first check if it contains a Boolean operation like "$==$" or "$>=$" in the child. If it does, then the child is either of the type $crossJoinCondition$ or $filterCondition$, and we add that information to later generate the grammar with these productions. If the child does not contain a Boolean operation, it can either be a $joinCondition$ or a $summariseCondition$. To distinguish between these, we check if any of the supported aggregate functions appear in the child. If so, then it is a $summariseCondition$; if not, the child is a $joinCondition$. This information is stored so that the synthesizer can later generate the grammar accordingly.

4.3 Sketch Filling

After having generated the grammar, in this phase, the synthesizer maps every known production in the sketch structure to its corresponding DSL production identifier.

Each DSL production has a different non-negative identifier to which we match the productions in the sketch. In this section, we explain how we match these productions in order to complete the sketch structure.

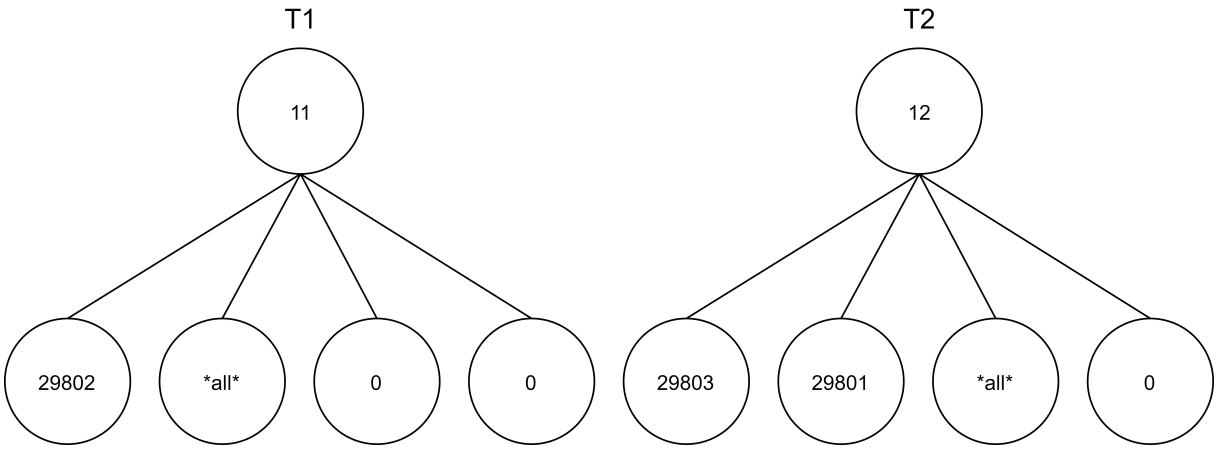


Figure 4.4: Filled Sketch Tree Example

In order to assign the DSL identifiers to the sketch structure, for every line, we first check if the root is a hole or not. If the root is a hole, we add to the structure the information that it can be any function. However, if it is not a hole, then we assign its corresponding DSL function production identifier or the multiple possible identifiers in case of an alternative hole.

Afterwards, for every child in the sketch structure, we check if the child contains information regarding the name and type, in which case SKEL assigns the corresponding production identifier from the DSL.

If we do not know the child type, we search in all the generated productions for a match and then assign those DSL identifiers to the child in the sketch structure. This search must be performed because some productions have equal content but a different type. For example, a column type production can be equal to a $joinCondition$ type production. If the child is a hole, no extra information is added to the sketch.

If we know the exact number of children in a line, we can assign the $empty$ DSL identifier to the excess leaves. For example, the `left_join` condition has two children. Therefore, considering that all trees have four leaves, the last two leaves will have the $empty$ identifier (0) assigned to them.

After this phase, the sketch tree example in Figure 4.3 will be filled with the DSL identifiers as shown in Figure 4.4. These numbers were retrieved from an extensive DSL with almost 30,000 productions. We can observe that the second leaf of the first tree and the third leaf of the second tree are filled with all the possible children productions, which is not very efficient.

Finally, the sketch structure now has information on each of its productions identifiers. The synthesizer then uses the sketch structure to generate the trees with the additional sketch information.

It is relevant to note that this sketch filling process is performed only once. The synthesizer can then use the sketch structure to generate new constraints every time the LOC is increased in order to search for more sizable solutions.

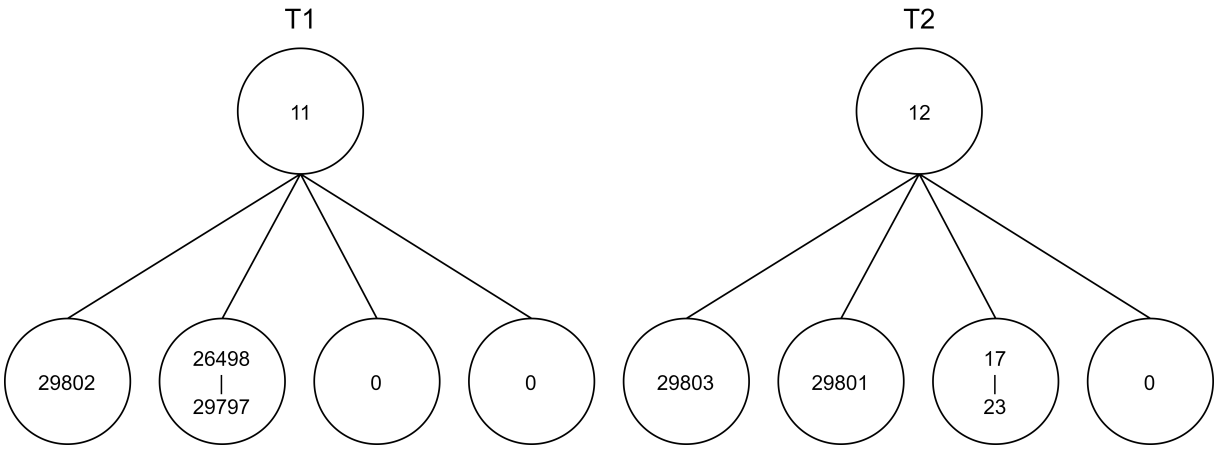


Figure 4.5: Filled Sketch Tree Example Optimized

Optimizations

We implemented two main optimizations when assigning the DSL identifiers to the sketch structure.

The first optimization applies when we have a parent hole. In this case, we can deduce a subset of functions that cannot be assigned to the parent's hole by analyzing the respective children. This deduction helps to prune solutions that would not satisfy the sketch. To do this, we analyze the type of each leaf and how many leaves there exist in the tree that are not $empty$.

For every unordered line, we add the DSL identifier of every function with arity equal to the number of non-empty leaves. For every incomplete line, we only add the DSL functions identifiers that have more input arguments than the known leaves in that sketch tree. We then check, for each function, if the children's types are consistent with the types of the known tree leaves in the sketch structure. If not, we remove that DSL function identifier from the possible sketch structure root productions.

The other optimization applies when we do not know the child's name but know its type. If this is the case, we only add the DSL identifiers that have the respective type. This optimization results in fewer productions allowed for the respective hole, thus pruning many invalid solutions. After applying these optimizations to the example in Figure 4.4, we obtain an optimized tree shown in Figure 4.5. In the new example, we can see that the tree contains a much smaller set of different DSL identifiers allowed for the leaves when compared with the previous sketch filling method.

4.4 Sketch Encoding

After having a complete sketch structure, the synthesizer will enumerate programs with the help of an SMT solver. For these generated programs to comply with the sketch structure, we encode this representation in an SMT formula and use it to enumerate valid programs. This way, we are able to

guide the synthesis process more efficiently.

Firstly, instead of building LOC trees with every production in every node (as CUBES does), we add the sketch constraints to each node, i.e., the set of productions allowed for each node is restricted according to the sketch. Subsequently, we implement new constraints that ensure the synthesizer follows the sketch correctly and optimize already existent CUBES constraints.

Let D denote the DSL, $Func(D)$ the set of functions in the DSL and $Prod(D)$ the set of children productions in the DSL.

Now, for the sketch notations, let $SRoot(i)$ represent all the possible function productions for the root in tree i of the sketch and $SLeaf(i, j)$ represent all the possible children productions for the j-th leaf of the tree i of the sketch.

As previously mentioned, each symbol of the DSL has a different non-negative identifier. We use $id(p)$ to denote the identifier of a given production p.

Finally, let n be the number of lines of the program the synthesizer is trying to generate, with a maximum operator arity of k, then we have the following integer variables:

- $\{op_i : 1 \leq i \leq n\}$: where each variable op_i denotes the production function used in tree i;
- $\{a_{ij} : 1 \leq i \leq n,\ 1 \leq j \leq k\}$: where each variable a_{ij} denotes the child production corresponding to argument j in tree i.

Root The function assigned to each tree must follow the sketch structure. If there is no information about that specific root in the structure, then we have the following constraints:

$$\forall\, 1 \leq i \leq n : \bigvee_{f \in Func(D)} op_i = id(f) \tag{4.1}$$

But if we have information on the DSL identifiers in the sketch structure for a specific root, then that tree has the following constraints:

$$\forall\, 1 \leq i \leq n : \bigvee_{f \in SRoot(i)} op_i = id(f) \tag{4.2}$$

From now on, let $Root(i)$ represent the possible function productions for the i-th tree.

Leaf The productions assigned to each leaf must also follow the sketch structure. If there is no information about that specific leaf in the structure, then we have the following constraints:

$$\forall\, 1 \leq i \leq n,\ 1 \leq j \leq k : \bigvee_{p \in Prod(D)} a_{ij} = id(p) \tag{4.3}$$

But if we have information on the DSL identifiers in the sketch structure for a specific child, then we have the following constraints:

$$\forall\, 1 \leq i \leq n,\, 1 \leq j \leq k : \bigvee_{p \in SLeaf(i,j)} a_{ij} = id(p) \tag{4.4}$$

From now on, let $Leaf(i, j)$ represent the possible children productions for the j-th leaf of the i-th tree. There are also specific constraints for the leaves of a line that is incomplete or unordered.

Unordered and incomplete lines In this case, we need to ensure that all the children must appear at least once in any of the leaves for that line (instead of only a specific leaf, like in the previous constraint). Let UL represent the set of indices of unordered lines and NC_i the number of children of line i, i.e., if line i is incomplete, then the number of children is going to be k, but if the line is unordered, then we use the information about its number of children given in the sketch. Let $S\_all\_children(i)$ denote all the productions known for each children in the sketch for line i. Finally, let c_j represent all the possible productions for the j-th child. As a result, we have the following constraints:

$$\forall\, i \in UL : \bigwedge_{c \in S\_all\_children(i)} \bigvee_{1 \leq j \leq NC_i} \bigvee_{p \in c_j} a_{ij} = id(p) \tag{4.5}$$

Type matching Besides the leaf constraints where each leaf is restricted to a subset of allowed DSL identifiers, we also have another set of constraints to ensure that the functions arguments format is followed. To do so, we traverse all function productions allowed for a given root and add two constraints. The first constraint ensures that the number of children is equal to the arity of the function, assigning the $empty$ production to all excess children:

$$\forall\, 1 \leq i \leq n,\ p \in Root(i),\, arity(p) < j \leq k :\ (op_i = id(p)) \implies (a_{ij} = id(empty)) \tag{4.6}$$

The second constraint ensures that for every possible function of a given root, every input argument of that function (leaf) has the correct type of productions. Let $Type(t)$ denote the type of production t and $Type(p, j)$ denote the type of parameter j of production p. If in the set of possible leaf productions for children j of line i there exist no productions of the correct type, i.e., $s = \emptyset$, then we only add the constraint in Equation 4.7. On the other hand, if there exist productions with the correct type, then the constraint in Equation 4.8 is added.

$$\forall\, 1 \leq i \leq n,\, p \in Root(i),\, 1 \leq j \leq k : \\ \neg(op_i = id(p)) \tag{4.7}$$

$$(op_i = id(p)) \implies \left(\bigvee_{s \in \{t \in leaf(i,j) : Type(t) = Type(p,j)\}} a_{ij} = id(s) \right) \tag{4.8}$$

Free lines If we have one or more free lines, we must ensure that those lines appear in the generated programs even if we do not know their exact position. Let FL be the set of free lines in the sketch (the lines we do not know the exact position), and ET the set of indices of trees in the enumerator, which we do not have any information on. Then, we have the following constraints:

$$\forall\, h \in FL : \bigvee_{i \in ET} \left[\left(\bigvee_{p \in SRoot(h)} op_i = id(p) \right) \wedge \left(\bigwedge_{1 \leq j \leq k} \bigvee_{p \in SLeaf(h,j)} a_{ij} = id(p) \right) \right] \tag{4.9}$$

After adding these constraints, the SMT solver will try to return a valid solution, with all the remaining holes filled. If it exists, this valid solution is then given to the decider in order to check if the program satisfies the input-output examples. If so, we have found a solution. Otherwise, the enumerator must generate another program. If we exhaust all possible solutions for that size, and if the sketch does not have a fixed maximum number of LOC then the number of LOC is increased. Otherwise, the program terminates with no solution found.

It is important to note that, if the sketch does not have fixed LOC, each time we increase the LOC we have to rebuild the trees for the new size, but the DSL and the filled sketch structure stay the same as previously stated.

Chapter 5

Experimental Evaluation

In this chapter, we evaluate the performance of SKEL, which was described in Chapter 4. Firstly, in section 5.1, we explain how the benchmarks were generated and each type of sketch used. Then, in section 5.2, we explain the methods that we use to perform the evaluation. Finally, in section 5.3, we compare different configurations of our approach and analyze them according to our metrics.

5.1 Benchmark Generation

To evaluate SKEL, we took the benchmarks used in CUBES [5] and prepared a subset of them to include several sketches, each one with different properties, resulting in a total of 100 benchmarks.

In Figure 5.1 we summarize the number of instances that we have evaluated for different numbers of LOC. With 29 instances having only 1 line of code, 35 with 2 lines of code, 26 with 3, and 11 instances with 4 lines of code. We also have examples with 5, 6, and 8 lines of code, each with only one instance, which helps us begin to understand how the program performs in more extensive scenarios. It is also relevant to note that there were a total of 48 instances with filter conditions and 49 instances with Aggregates. The set of benchmarks used includes:

- 29 instances extracted from exercises of the textbook *Database Management Systems* [25];
- 22 + 39 instances from the benchmarks used in Scythe [34], collected from recent and top-rated posts from StackOverflow[1], respectively;
- 9 instances from Spider [37], a dataset of SQL queries and respective natural language descriptions. For each instance, the SQL solution query and the sample database contents were used to create an input-output example for PBE synthesizers without manual intervention.

[1] https://stackoverflow.com

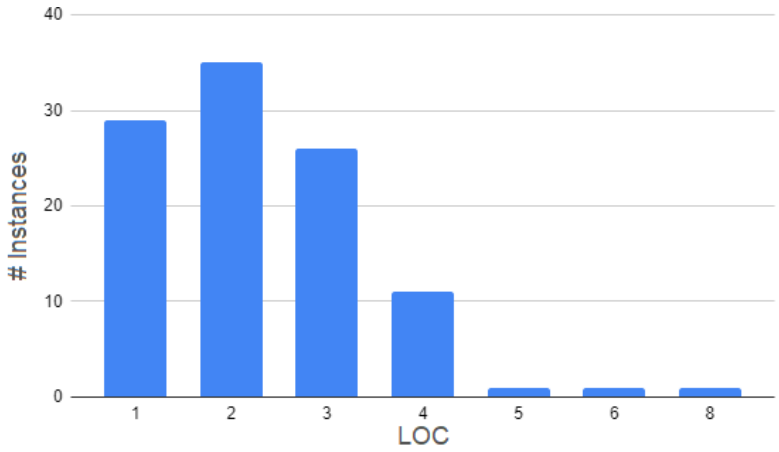


Figure 5.1: Instances LOC

To generate these benchmarks, we analyzed each of them and converted the SQL ground truth solution provided with each instance into our sketch format without any holes, a full sketch. We then proceeded to replace parts of the solution with holes. These sketches can be split into the following categories:

- Sketch with **no root**, i.e, sketches where the parent is replaced by a hole, as shown in Example 16.

  Example 16. *Sketch with no root:*

  $$T1 = ??\ (students,\ id > 3)$$
  $$T2 = ??\ (T1,\ max\_grade = max(grade),\ id)$$

- Sketch with **no children**, i.e, sketches where the children are replaced by holes, as shown in Example 17.

  Example 17. *Sketch with no children:*

  $$T1 = filter\ (??,\ ??)$$
  $$T2 = summarise\ (??,\ ??,\ ??)$$

- Sketch with **no root and no filter** where the parents and the filter production are replaced by a hole, as shown in Example 18.

  Example 18. *Sketch with no root and no filter:*

  $$T1 = ??\ (students,\ ??)$$
  $$T2 = ??\ (T1,\ max\_grade = max(grade),\ id)$$

- Sketch with **no root and no aggregate** where the parents and the aggregate production are replaced by a hole, as shown in Example 19.

  Example 19. *Sketch with no root and no aggregate:*

  $$T1 = ??\ (students,\ id > 3)$$
  $$T2 = ??\ (T1,\ ??,\ ??)$$

- Sketch with **no children except filter** where the sketch children are all replaced by holes except the filter production, as shown in Example 20.

  Example 20. *Sketch with no children except filter:*

  $$T1 = filter\ (??,\ id > 3)$$
  $$T2 = summarise\ (??,\ ??,\ ??)$$

- Sketch with **no children except aggregate** where the sketch children are all replaced by holes except the aggregate production, as shown in Example 21.

  Example 21. *Sketch with no children except aggregate:*

  $$T1 = filter\ (??,\ ??)$$
  $$T2 = summarise\ (??,\ max\_grade = max(grade),\ id)$$

5.2 Evaluation Method

In this section, we explain the different evaluation metrics used to analyze SKEL. We consider the following metrics, which are further explained in the rest of this section:

1. Efficiency (runtime to return a solution);
2. Number of attempted programs;
3. Accuracy;
4. Precision.

The first evaluation metric considers the program's efficiency, i.e., the time required by the synthesizer to return a solution. This metric allows us to compare the impact of using each different type of sketch. We also compare the different sketch types with a baseline using no sketch at all.

Then, we measure the number of programs generated by the synthesizer before returning a solution. We try to understand the impacts that using a sketch has on this metric and if it correlates with the efficiency of each type of sketch.

Another metric is the number of instances solved and if these were solved correctly according to the ground truth. It is essential to note the difference between the terms valid solution and correct solution. The first refers to any solution returned by the synthesizer, while the latter only refers to solutions with the same semantics as the ground truth.

We calculate the accuracy and the precision of each type of sketch and report rates of true positives, false positives, and false negatives. For this evaluation, a true positive is when the synthesizer returns a solution with the same semantics as the ground truth. A false positive is when the synthesizer returns a solution that does not implement the same semantics as the ground truth. Furthermore, a false negative is when the synthesizer could not produce a solution, i.e., timed out. It is important to note that the notion of true negative does not apply in this evaluation since all instances have a solution.

The accuracy metric is to understand how many instances the synthesizer could solve correctly according to the ground truth. The metric's values range between 0 and 100. An accuracy of $X\%$ indicates that the synthesizer could solve $X\%$ of all instances correctly, i.e., consistent with the ground truth. The formula for this metric is represented in equation 5.1:

$$Accuracy = \frac{true\_positives}{true\_positives + false\_positives + false\_negatives} \tag{5.1}$$

Finally, we use the precision metric to understand the percentage of the instances solved according to the ground truth. Similarly to the accuracy, the metric's values range between 0 and 100. A precision of $X\%$ indicates that the synthesizer returned a correct solution for $X\%$ of the instances it could solve. The formula for this metric is represented in equation 5.2:

$$Precision = \frac{true\_positives}{true\_positives + false\_positives} \tag{5.2}$$

5.3 Experimental Results

In this section, we evaluate the impact of different sketch types on the synthesizer's performance and understand how the obtained results correlate with our expectations.

First, we performed a simple analysis using a baseline where no sketch is used, and then we compared each of the different sketch types.

The results were obtained using an Intel® Xeon® CPU E5-2630 v2 @ 2.60GHz, with 64GB of RAM. We also used runsolver [26] to run the benchmarks with a limit of 10 minutes and 56GB of RAM.

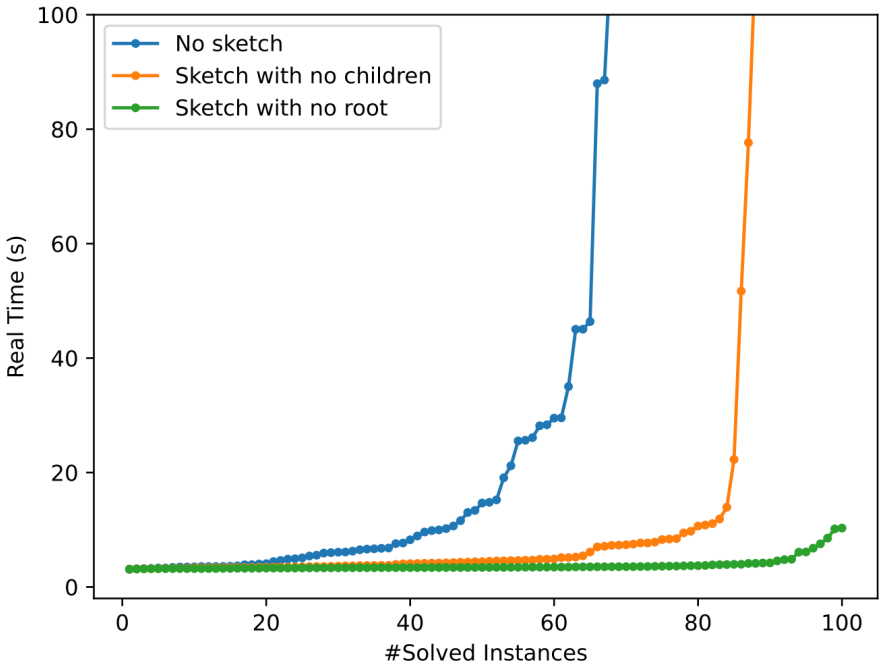


Figure 5.2: Baseline efficiency

5.3.1 Baseline Analysis

We start the evaluation by comparing two of the most simple sketch formats with a baseline using no sketch in Figure 5.2. The synthesizer solved 72 instances when no sketch was provided, and we can see it started having some difficulties roughly after the 50th instance. Since we do not give the synthesizer any additional information regarding the expected solution, this result is expected.

The benchmarks where sketches with no children are used appear to result in some difficulties roughly after the 84th instance. The synthesizer can solve a total of 92 instances without timing out, which is a considerable improvement compared with the baseline.

Finally, we ran the benchmarks with a sketch that had no root, and the synthesizer was able to solve all of the 100 benchmarks with a significant boost in performance. It is expected that when we give the synthesizer a *sketch with no children*, it performs worse than a *sketch with no root*. This performance difference is because only 15 productions are required in order to fill a hole in the *sketch with no root*, while the sketches with no children require an increased number of productions for each hole that the synthesizer has to use to find a solution, thus it has less information overall.

To sum up, we can clearly see that the synthesizer performs better with every sketch type compared to the baseline and is able to solve more instances when a sketch is given.

We then analyzed the number of programs generated by the synthesizer before a solution is found, as shown in Figure 5.3. In the plot, we can observe that, by the 70th instance, the benchmarks with no sketch were already exceeding 2000 tested programs before returning a solution. On the other hand, at

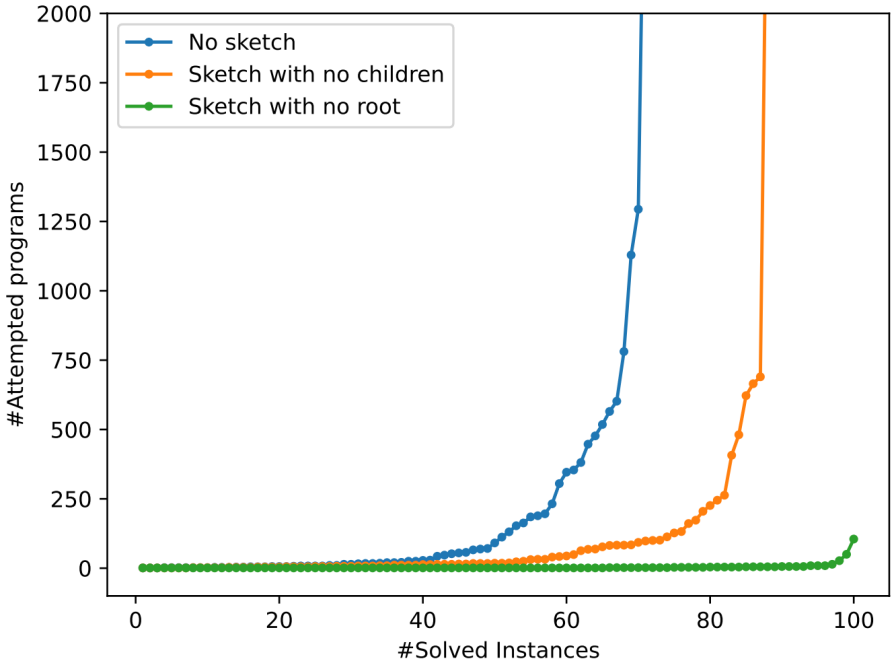


Figure 5.3: Baseline attempted programs

the 70th instance, the benchmarks where a sketch was given are still only generating roughly around a hundred solutions in the worst scenario, that is the *sketch with no children*.

Regarding the *sketch with no children* we start seeing a more notable increase by the 85th instance, solving all the previous instances with ease. Since we could solve benchmarks with more lines of code or more complex, with holes that could be filled with more productions, it is expected that the number of attempted programs also increases. After further analysis, we concluded that the increase in attempted programs is mainly because the instances have more filter conditions to choose from since the solution has to have two filters, which impacts the number of attempts.

Finally, we can see that the number of attempted programs of the instances where a *sketch with no root* was given is reasonably low. This number is expected because the synthesizer has more information than previous sketches, requiring fewer attempted programs to return a valid solution. The slight increase we see at the end is from an instance for which the respective solution has 8 LOC, thus increasing the number of attempted programs to 105.

Figure 5.4 shows a bar chart where it becomes clear the increase in solved and correct instances throughout the different sketches.

When using no sketch, the synthesizer was only able to solve 72 instances, with only 52 of them being correct according to the ground truth. On the other hand, if we provide a *sketch with no children* to the synthesizer, the number of solved instances increases to 92, with 69 of them correctly solved. Out of the 72 solved instances when using no sketch, by using the *sketch with no children* we were able to

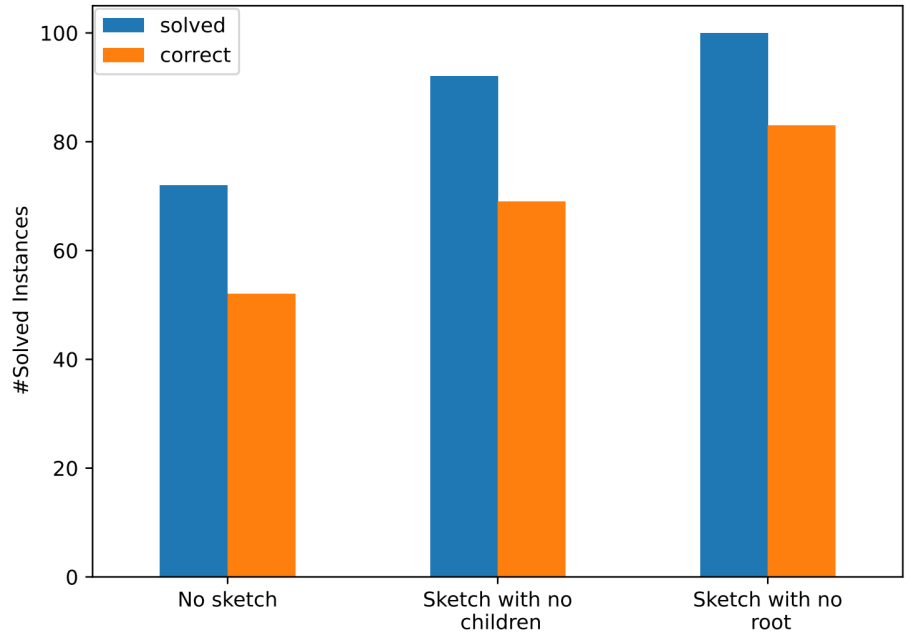


Figure 5.4: Baseline solutions

Sketch Type	Accuracy (%)	Precision (%)	Average Time (s)	#Solved Instances
No sketch	52,00	72,22	40,01	72
Sketch with no children	69,00	75,00	6,37	92
Sketch with no root	83,00	83,00	3,48	100

Table 5.1: Baseline metrics

solve 61 of them correctly. In the best case, i.e., when a *sketch with no root* is given, the synthesizer is able to solve all instances, and 83 of them correctly.

In Table 5.1, we can observe that the accuracy and precision metrics increase with the more information the sketch has, as expected. In the baseline where no sketch is provided, we can see an accuracy of 52% meaning that, of all the instances, this benchmark could only solve a little more than half of them correctly. On the other hand, the accuracy when a *sketch with no children* or a *sketch with no root* is provided corresponds to 69% and 83%, respectively.

Nonetheless, this increase was not so notorious when we compare the precision of the benchmarks with no sketch with the ones that have a *sketch with no children*. This precision means that although the synthesizer can solve more instances, many of those instances are not correctly solved consistently with the ground truth. This slight increase is expected since the sketches with no children do not contain much more information than no sketch.

For filter conditions, this type of sketch performs very poorly since, depending on the input-output examples specification, there can be some filters that satisfy the specification but are not what the user

intended. For example, consider the Table 1.1 in the motivating example. Suppose that a user wanted to get from that table all the bridges with a length higher than 65m. As we can see, 65m is the minimum length presented in the table. Due to this ambiguity, both the $length > 65$ and the $length != 65$ filter conditions would be valid, but only the first would correspond to the user intent.

In the best-case scenario, i.e., when a *sketch with no root* is provided, we can conclude that 83% of the instances solved were consistent with the ground truth. On the other hand, 17% of the instances were not solved correctly. These are mostly cases where parents that should be classified as `natural_join` are instead classified as `left_join` or `semi_join`. This misclassification is mainly due to the ambiguity of the input-output examples provided.

We can also analyze the average time that, for the same set of solved solutions, the synthesizer took to return a valid solution.

We can also analyze the average time that the synthesizer took to return a valid solution. It is important to note that this average is done considering only the benchmarks solved by every sketch type. If no sketch is given, the synthesizer takes an average time of 40,01 seconds to return a solution. In the benchmarks where a sketch is provided, we can see a substantial time improvement. With a sketch with no children, the time the synthesizer takes to return a solution drops to 6,37 seconds, and with no root, to almost half of that, 3,48 seconds.

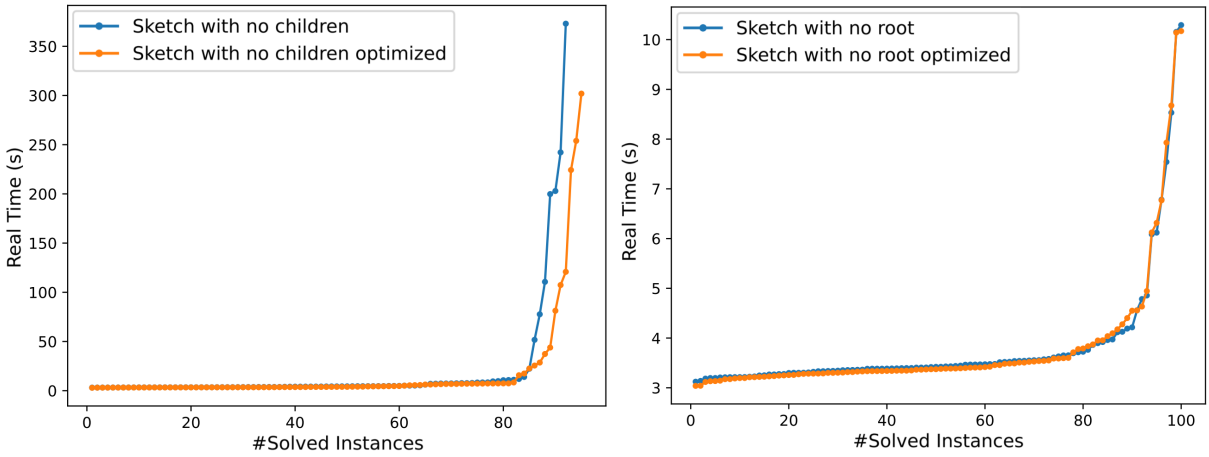
With this information, we can already see the positive impact of giving the synthesizer a sketch. The sketches result in a significant boost in performance since, in most cases, a solution can be found faster and more accurately.

5.3.2 Optimization Analysis

To analyze the optimization described in section 4.3, we consider two sets of the benchmarks in subsection 5.3.1: *sketch with no children* and *sketch with no root*.

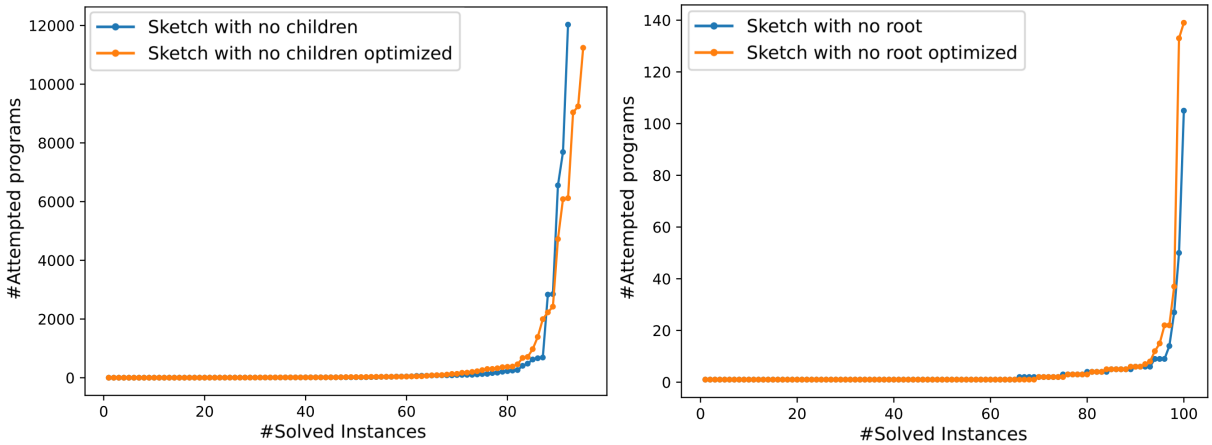
In Figure 5.5a, we can observe in the benchmarks that use a *sketch with no children* that the optimization has a positive impact on the efficiency. The optimized version is able to solve 95 out of the 100 instances, while the non-optimized version only managed to solve 92 instances and required overall more time to return an answer. On average, for the same set of solved benchmarks, the optimized version took approximately 12,29 seconds to return a solution, while the non-optimized version took 16,55 seconds, confirming the efficiency improvements previously stated.

In the benchmarks that ran with a *sketch with no root* in Figure 5.5b, we cannot clearly see an improvement in efficiency in this plot, with the optimized version being very similar to the non-optimized and both solving the 100 instances. On the other hand, if we analyze, for the same set of solved benchmarks, the time it took for each version to return a solution, we can see a slight increase in performance while using the optimized version, which returns a solution in an average of 3,59 seconds,



(a) Sketch with no children **(b)** Sketch with no root

Figure 5.5: Optimized version efficiency



(a) Sketch with no children **(b)** Sketch with no root

Figure 5.6: Optimized version attempted programs

while the non-optimized version takes 3,63 seconds. This difference in efficiency is not as relevant as the difference using the *sketch with no children*, but it should be mentioned nonetheless.

Next, we analyze the plots in Figure 5.6. This analysis helps us better understand the optimization's impact in terms of the number of programs that had to be tested until a solution was found.

In Figure 5.6a we can see that, in general, both of the sketches are side by side in terms of attempted programs, but the optimized version can solve more instances using less generated programs.

The benchmarks with the *sketch with no root* in Figure 5.6b have slight increase of attempted programs in the optimized version. This increase goes against our expectations but, after further analysis, this is possible if the accuracy increases since the synthesizer will need to generate more programs to find the correct solution instead of returning a solution that is not consistent with the ground truth due to the ambiguity of the specification. Nonetheless, this increase in attempted programs does not

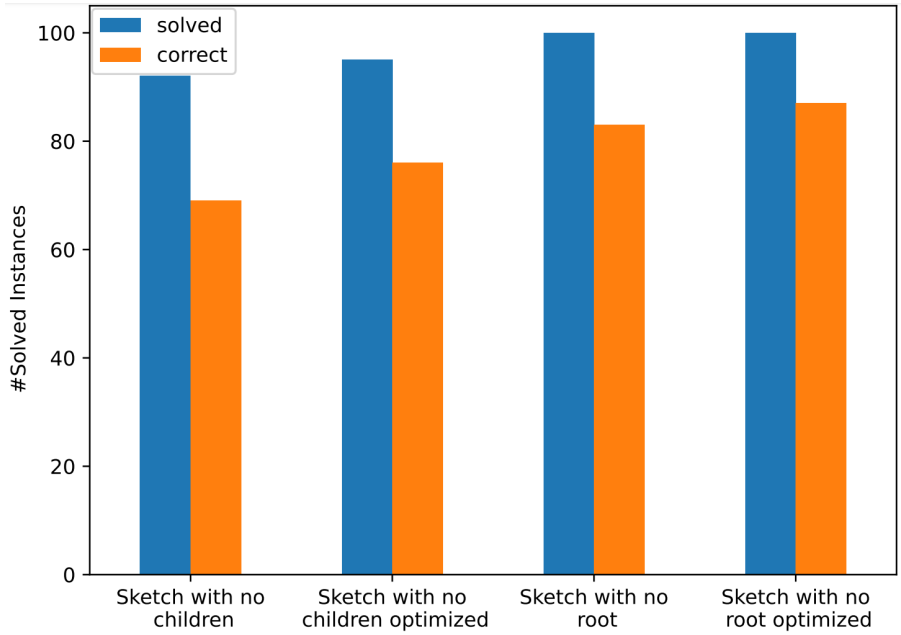


Figure 5.7: Optimized version solutions

Sketch Type	Accuracy (%)	Precision (%)	Average Time (s)	#Solved Instances
Sketch with no children	69,00	75,00	16,55	92
Sketch with no children optimized	76,00	80,00	12,29	95
Sketch with no root	83,00	83,00	3,63	100
Sketch with no root optimized	87,00	87,00	3,59	100

Table 5.2: Optimized version metrics

compromise the efficiency as we have seen before.

We now analyze the bar chart in Figure 5.7 where we can clearly see an improvement in the number of correctly solved instances while using the optimization.

If we run the synthesizer with the optimization and the *sketch with no children* we notice that we can solve 3 more instances (from 92 without optimization to 95 with optimization) and that the number of correctly solved instances increases from 69 to 76 instances.

Using the *sketch with no root*, we notice that the number of solved instances remains the same since it already solved all the instances, but the number of correctly solved instances increases from 83 to 87.

In Table 5.2 we can observe that the accuracy and precision metrics increase for the optimized versions, as expected. Without the optimization, the benchmarks ran with a *sketch with no children* had an accuracy of 69% while the optimized version was able to achieve an accuracy of 76%. The precision increased from 75% with no optimization to 80% with the optimization.

The benchmarks with a *sketch with no root* follow a pattern similar to that of the version without

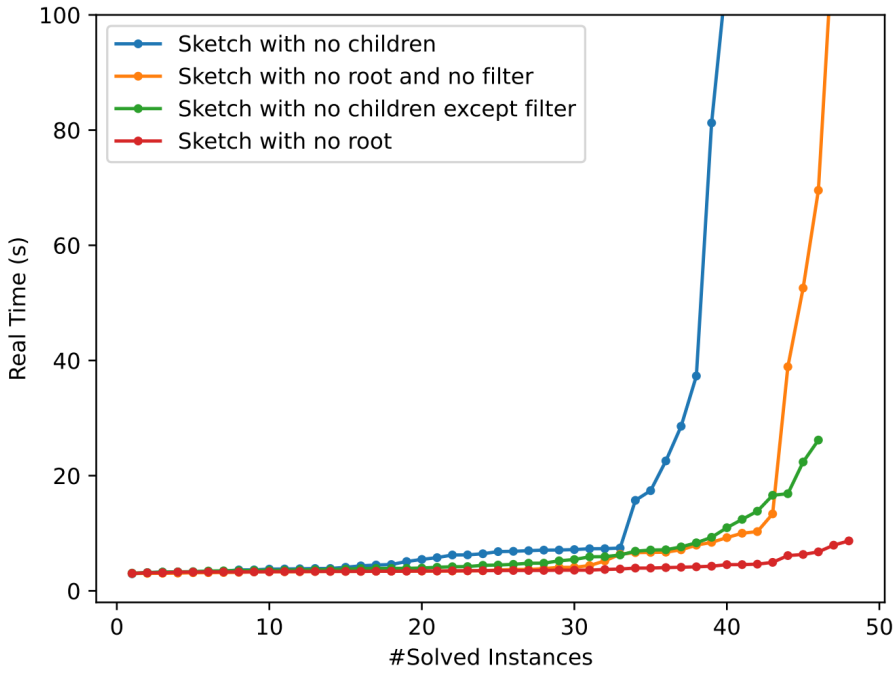


Figure 5.8: Filter sketches efficiency

optimization, resulting in an accuracy of 83% without optimization and 87% with. This means that, although the efficiency of both approaches is very similar, we still observed an increase in accuracy with this new optimization which is what we intended. This also explains why the number of attempted programs might have been slightly higher for the optimized version since this one was searching for a correct solution that, in terms of the order, was not one of the first solutions the synthesizer attempted.

Comparing the precision for each benchmark, we can see that using the *sketch with no children* the optimized version also performed better with a precision of 80% versus only 75% for the non-optimized version. Since, when given the *sketch with no root*, the synthesizer solved all 100 instances then, the precision is exactly like the accuracy, and they hold the same conclusions.

To sum up, the optimization proved to be relevant for our work, not only slightly increasing the efficiency but, most importantly, increasing the accuracy of the solutions returned.

It is important to note that we only use the optimized version for the rest of this section since this one proved to return solutions more accurately and, in most cases, be more efficient than the non-optimized version. Therefore, the *sketch with no children optimized* and the *sketch with no root optimize* are now referred to as simply *sketch with no children* and *sketch with no root*.

5.3.3 Sketches with Filters

In this subsection, we analyze the impact of different sketch types when considering the filter productions. We used the 48 instances for which the respective ground truth contains at least one filter for this

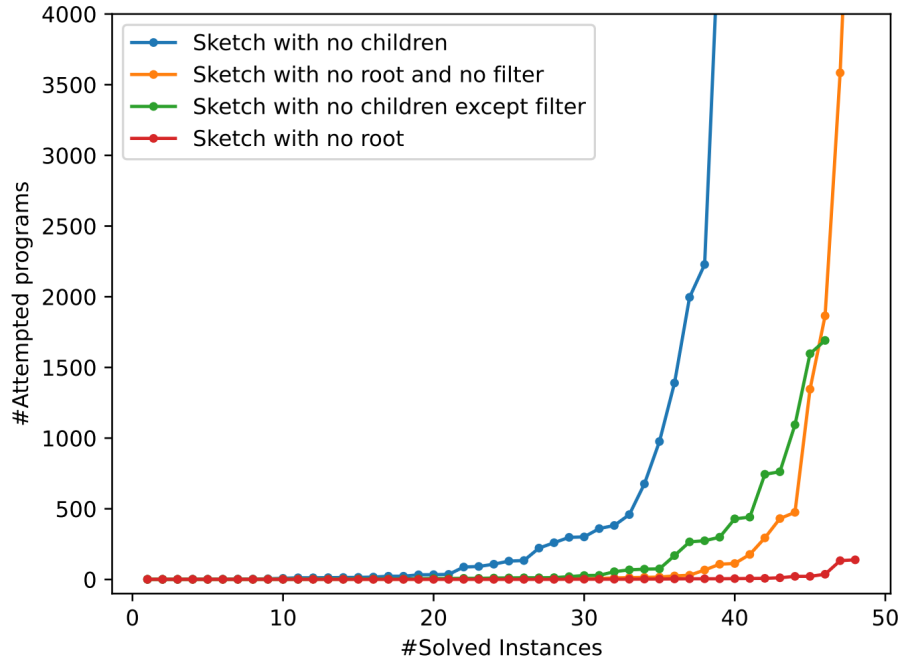


Figure 5.9: Filter sketches attempted programs

evaluation. We compare the *sketch with no children* and the *sketch with no root* with two new sketches, namely the *sketch with no root and no filter* and the *sketch with no children except filter*.

Looking at Figure 5.8 we can see that the two new sketches perform better than the *sketch with no children*. This improvement is expected since both contain more information than the aforementioned type of sketch. It is also expected that both of these new sketches perform worse than the *sketch with no root* since, contrary to the previous case, the new sketches have less information than this type of sketch.

Between the *sketch with no root and no filter* and the *sketch with no children except filter*, we can see that the latter performs better, which is according to our expectations. This improvement is because there are many different filter productions in the DSL, so if we immediately give the synthesizer the correct production, it does not need to test all of the filter productions, thus decreasing the search time. We can also observe that it is not able to solve two instances that the *sketch with no root and no filter* can solve, but we address this issue later in this subsection while discussing the accuracy and precision of the different sketch types.

To sum up, out of the 48 instances, the *sketch with no root* and the *sketch with no root and no filter* were able to solve all of them with an average time of 3,77 and 7,99 seconds, respectively. The *sketch with no children except filter* only solved 46 instances with an average time of 6,07 seconds, and the *sketch with no children* performed the worse with 44 instances solved and an average time of 31,30 seconds.

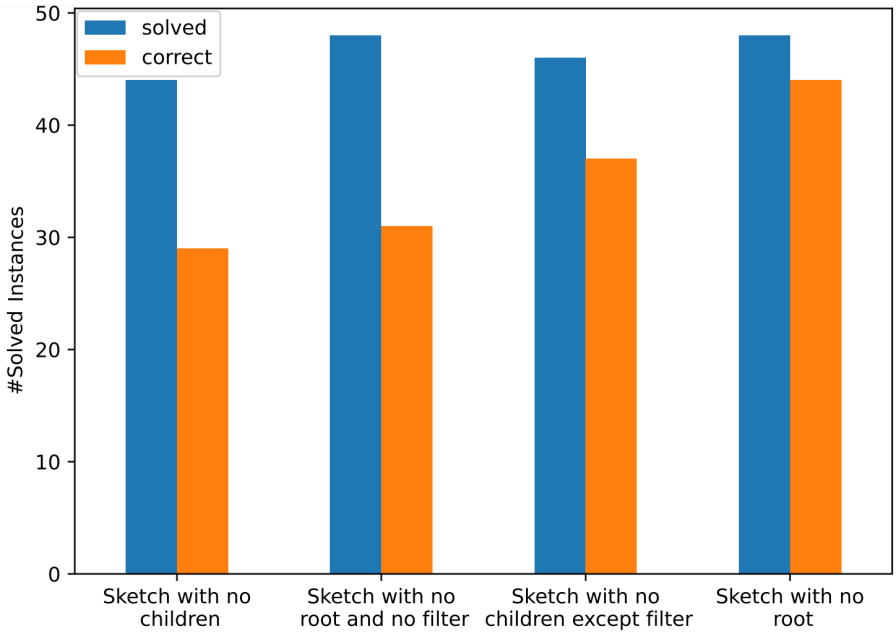


Figure 5.10: Filter sketches solutions

In Figure 5.9 we can see that the *sketch with no children except filter* generally attempts more programs than the *sketch with no root and no filter* before returning a solution. A possible explanation could be an increase in accuracy since the synthesizer could have to generate more programs in order to return the correct solution. We address this later in this subsection.

On average, for the same solved instances, the *sketch with no root and no filter* generates 150 programs while the *sketch with no children except filter* generates 151, although these results start to escalate for instances that require more complex solutions. Nonetheless, the *sketch with no children* generated more programs on average, 1291, than these two new filter sketches, and the *sketch with no root* generated fewer programs (6) before returning a solution.

In the bar chart in Figure 5.10, we can clearly see an improvement in the number of instances that are solved correctly. With the *sketch with no children*, we see that only 29 instances were correctly solved, and with the *sketch with no root and no filter*, only 31 instances were solved correctly as well. These results show the importance of the filters in the sketches since the sketches with no root have the highest correctly solved instances (44), but if we remove only the filter condition(s), those numbers drop significantly. On the other hand, if we add only the filter condition to the sketches with no children, the number of correctly solved programs increases to 37.

Looking at Table 5.3, we can see how the accuracy increases when we provide a filter in the sketch. This increase is expected since it is hard for the synthesizer to distinguish between, for example, a condition with a ”greater or equal then” sign, $>=$, or with only a ”greater than” sign, $>$, if the respective corner case is not included in the input-output example specification. Using a *sketch with no root and*

Sketch Type	Accuracy (%)	Precision (%)	Average Time (s)	#Solved Instances
Sketch with no children	60,42	65,91	31,30	44
Sketch with no root and no filter	64,58	64,58	7,99	48
Sketch with no children except filter	77,08	80,43	6,07	46
Sketch with no root	91,67	91,67	3,77	48

Table 5.3: Filter sketches metrics

no filter, we obtain an accuracy of 64,58%, meaning that it could solve more instances correctly than the *sketch with no children*, which resulted in an accuracy of 60,42%.

However, we can see that the precision of the *sketch with no root and no filter*, 64,58%, is lower than the precision of the *sketch with no children*, 65,91%. This difference means that, although all the children (except the filter) are provided, the *sketch with no children* returns more often a correct solution for the instances that it is able to solve. One explanation for this is the fact that `natural_joins` are often mistaken by `left_joins` or `semi_joins` by the synthesizer, and since we do not provide the roots, this results in additional ambiguity for a sketch that already does not have the filter condition.

It is also relevant to note that we had 4 instances where the *sketch with no root and no filter* returned a wrong root even though the *sketch with no root* returned the correct one. This can be justified if we consider that the synthesizer does not necessarily follow the same search patterns when more or fewer constraints are given. This could explain why the *sketch with no root* had the correct solution for these 4 instances, while the new type of sketch did not.

Considering the results of the *sketch with no children except filter*, we can see a substantial improvement with an accuracy of 77,08% and a precision of 80,43%. This improvement means that only providing the filter condition and no other children is enough to increase the accuracy by almost 17%. This is according to our expectations since there are many possible filter productions and so, by providing the correct filter that has to be used, we can significantly improve the accuracy.

We should also note that when the synthesizer is given a *sketch with no children except filter*, it is not able to solve two instances (due to time-out) that could solve with a *sketch with no root and no filter*. After further analysis, we could see that with the *sketch with no root and no filter* the two instances were not solved correctly in respect to the ground truth. Thus, although the synthesizer was able to return a solution before the time-out, this solution did not correspond to the user's intent. Another essential piece of information is that these instances were the ones with 6 LOC and 8 LOC which means that there was still a large number of children that had to be discovered besides the filter condition. It is also interesting to note that the query with 5 LOC was the one that took longer to solve (approximately 26 seconds).

The *sketch with no root* still has the best performance, as expected, with a 91,67% of accuracy and precision since it solved all instances. These new types of sketches perform worse than the *sketch with*

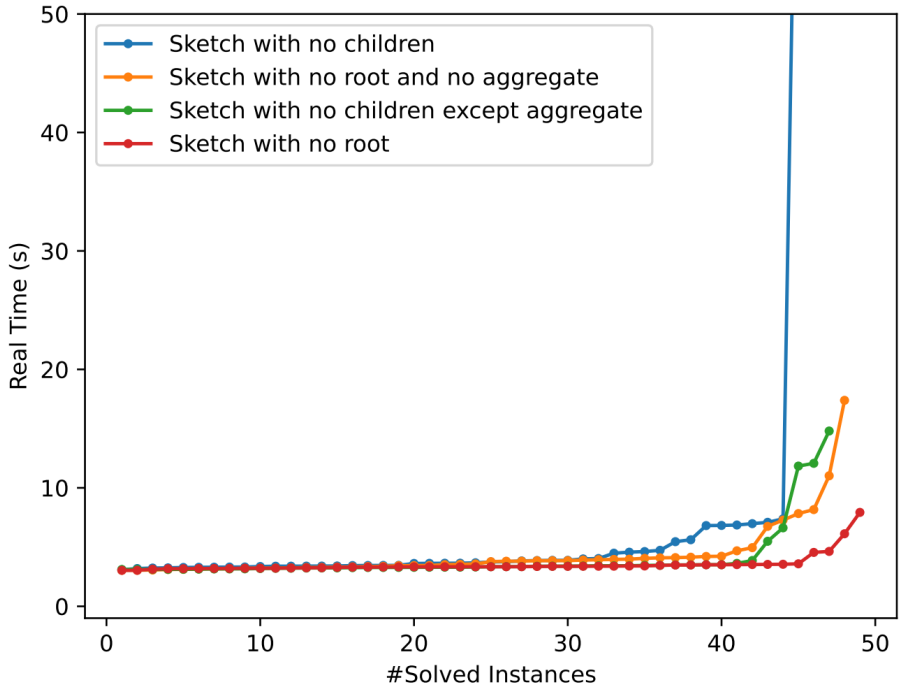


Figure 5.11: Aggregate sketches efficiency

no root since both have less information.

To sum up, we concluded that the filter conditions significantly impact the efficiency and accuracy of the synthesizer. Thus, if the user is able to provide the filter condition to the sketch, the synthesizer will yield much better results.

5.3.4 Sketches with Aggregates

Similarly to the previous subsection, we analyze the impact of different types of sketches with and without aggregate productions. For this evaluation, we used 49 instances for which the respective ground truth contains an aggregate. We introduce two new types of sketches, namely the *sketch with no root and no aggregate* and the *sketch with no children except aggregate*.

In Figure 5.11 we can see that there is an improvement in the performance of all sketches when compared to the *sketch with no children* which takes an average time of 11,29 seconds to return a solution for the instances solved by all sketches.

The two new sketches perform very similarly in terms of efficiency and do not perform as expected when compared with each other. We expected that the *sketch with no root and no aggregate* performed worse than the *sketch with no children except aggregate* since it would have less information about the aggregate function which would result in a longer search time.

For the same set of instances that returned a solution using both sketches, we can see that the

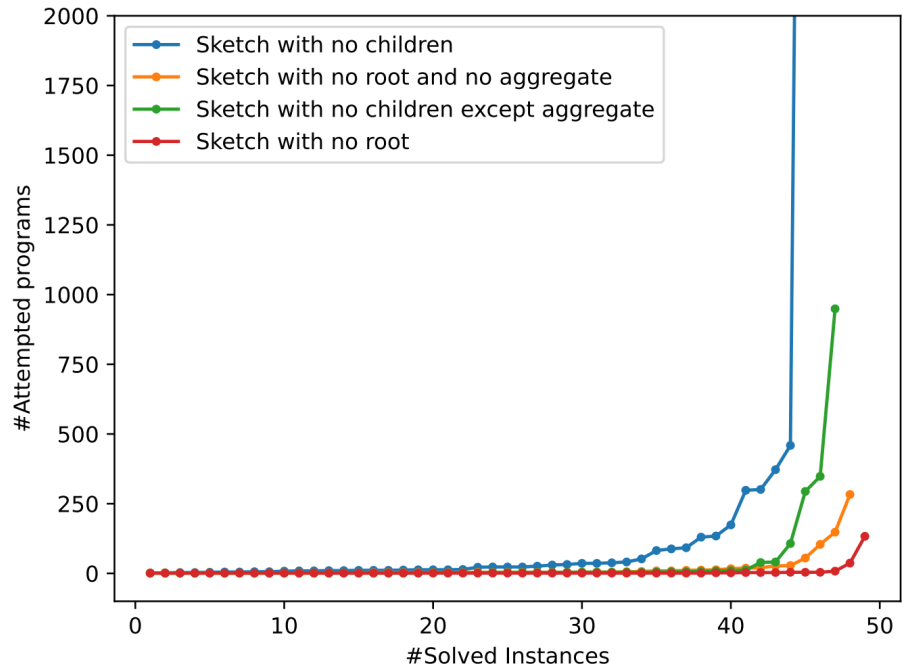


Figure 5.12: Aggregate sketches attempted programs

sketch with no root and no aggregate takes on average longer (4,25 seconds) to return a solution than the *sketch with no children except aggregate* which takes on average 3,81 seconds to return a solution.

After further analysis, we understood that the reason for this was that SKEL does not generate an extensive amount of aggregate productions in the DSL. The only thing the synthesizer has to consider is the columns that will be used in the `GROUP BY` clause. This substantially reduces the number of productions the synthesizer must consider before returning a solution, which could explain the obtained results.

As expected, the *sketch with no root* outperforms all other sketches with an average time of only 3,42 seconds for the instances that all sketches could solve.

Looking at Figure 5.12, we can see that the results are consistent with what we previously observed. The *sketch with no children* generates the most programs before returning a valid solution with an average of 387 programs for the same set of instances solved by all sketches.

The *sketch with no children except aggregate* takes an average of 21 programs to return a solution while the *sketch with no root and no aggregate* takes an average of 9 attempts. These results are consistent with the differences between the average times for the same instances that we previously mentioned.

The *sketch with no root* requires the fewest attempts before returning a solution with only an average of 2 programs for the same set of instances solved by all sketches.

Now we analyze each sketch and understand which returned more correct solutions and why. In

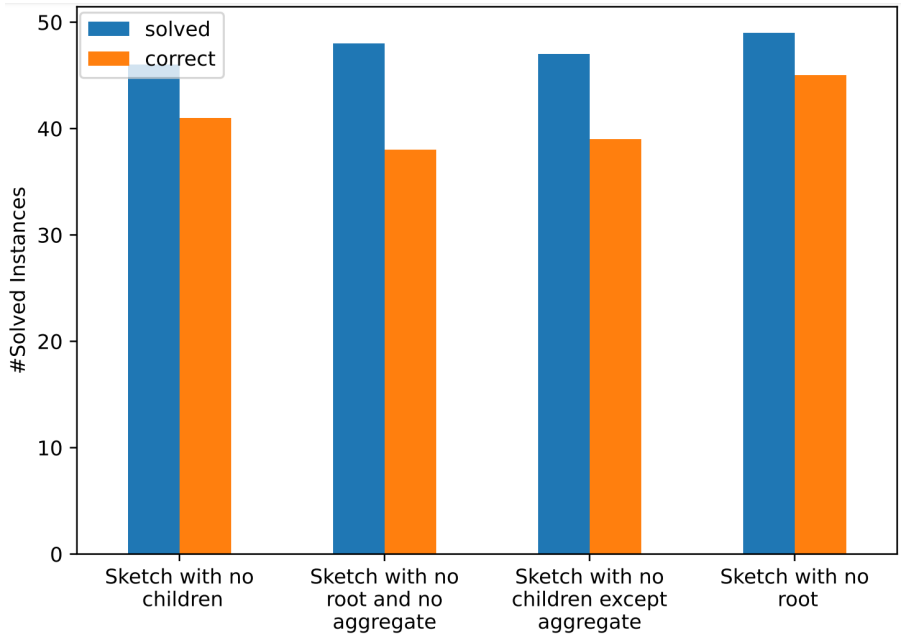


Figure 5.13: Aggregate sketches solutions

Figure 5.13 we can see that, compared to the previous bar charts, *sketch with no children* finds the correct solution more often than the remaining sketch types for this subset of instances. We can see that for a total of 49 instances, this type of sketch returned 40 correct solutions. As previously mentioned, this is probably due to the fact that the synthesizer does not have an extensive number of productions that it needs to try in order to return a solution with an aggregate. This allows the *sketch with no children* to achieve a greater accuracy since for the aggregate children, it does not need to do an extensive search, and it is not as prone to make mistakes due to the ambiguity of the specification. Overall this means that the instances with an aggregate will likely not need much information in order to yield correct results.

When looking at the *sketch with no root and no aggregate* we can see that more instances were solved (48), but fewer of them were solved correctly (only 38). Of the 10 instances that it could not solve correctly, most of them were new mistakes introduced by the lack of the aggregate production. It makes sense that the number of correctly solved instances dropped comparatively with the *sketch with no root* since it has less information than this sketch type, but we did not expect a result worse than the *sketch with no children*.

The *sketch with no children except aggregate* also does not improve the number of correctly solved instances, only solving 39 instances correctly. Most of the instances that it was not able to solve correctly contained an incorrect filter condition.

The *sketch with no root* keeps performing the best, with 45 instances solved correctly. The instances that the synthesizer could not solve correctly using this sketch are mostly mistakes that we previously

Sketch Type	Accuracy (%)	Precision (%)	Average Time (s)	#Solved Instances
Sketch with no children	83,67	89,13	11,29	46
Sketch with no root and no aggregate	77,55	79,17	4,25	48
Sketch with no children except aggregate	79,59	82,98	3,81	47
Sketch with no root	91,84	91,84	3,42	49

Table 5.4: Aggregate sketches metrics

discussed where a `natural_join` is mistaken with a `left_join` or a `semi_join` due to the ambiguity of the specification.

If we analyze Table 5.4, we can clearly see that the two new sketches yield worse results than expected. The accuracy and the precision of the *sketch with no children* are 83,67% and 89,13%, respectively, both better than the results obtained using the new sketches.

For the *sketch with no root and no aggregate* we can see that an accuracy of 77,55% and a precision of 79,17% were obtained. Both are worse than using a *sketch with no children except aggregate*, which resulted in an accuracy of 79,59% and a precision of 82,98%. This behavior is the expected between these two sketches, and it can explain the results obtained in the Figures 5.11 and 5.12 since the synthesizer could have been taking longer to return a solution so that it could return the correct solutions instead of only a valid one.

We did not expect the accuracy of the *sketch with no children except aggregate* to be so low. After a deeper analysis of the results, we concluded that the instances that were not solved correctly were mainly due to an incorrect filter condition. The reason that this sketch performed worse than the *sketch with no children* is probably that the synthesizer traversed the search space in a different order and thus found another solution consistent with the input-output example that uses a filter that does not satisfy the user's intent. Since it is much more likely to return a wrong filter condition than an aggregate due to the difference in the number of possible filter and aggregate productions generated, adding this additional information does not improve SKEL's performance. Therefore, we conclude that by generating the grammar from the DSL in this way, the aggregate production is not crucial to yield better results in terms of accuracy.

Finally, the *sketch with no root* continues to outperform the others by having the most information. For this specific set of instances, it resulted in an accuracy and precision of 91,84%.

5.3.5 Overall Discussion

From this evaluation, we can clearly see how a sketch enables the synthesizer to increase its performance and accuracy. The optimizations described in section 4.3 also improved the accuracy of the

synthesizer, and in most cases, also improved performance.

Regarding the different types of sketches, we concluded that some of the production types have more impact when searching for a solution, like filter productions, and others have less impact, like, for example, the aggregate productions.

Overall the fact that a sketch increases the performance and accuracy of the synthesizer is consistent across the different sketch types. Although among these different sketch types it is necessary to consider the impact of giving specific information by analyzing the number of productions generated for each production type.

Chapter 6

Conclusion

In this thesis, we introduced the topics of program synthesis and sketching and discussed state-of-the-art techniques used in these fields. We then proposed a new query synthesizer, SKEL, that had the previously existing synthesizer CUBES as a starting point. SKEL can receive sketches as a specification to help in guiding the synthesis process more efficiently and accurately. We introduce a new sketch format with a large variety of hole options such as line holes, children holes, root holes, et cetera. This format allows the user to provide a sketch that fits different situations, thus enhancing the performance and accuracy of the synthesizer.

We then evaluated SKEL. Firstly we compared it with a baseline, namely the CUBES-Seq version, while using simple sketches. The increase in performance was notorious as expected, there was an increase in accuracy of 17% in the sketches with no children and of 31% using the sketches with no root, which had the most information. The optimizations performed in our synthesizer also proved to increase the efficiency but mainly the accuracy of the synthesizer. We had an increase of 7% in accuracy using the sketches with no children and 4% for the sketches with no root.

Regarding the different sketch types, we concluded that the filter condition is very relevant in a sketch since it dramatically increases the synthesizer's performance. Nonetheless, if we only have a rough structure of the query the user wants to generate, the impact in efficiency and accuracy is already noticeable if we compare it with no sketch being provided.

We can conclude that SKEL boosts the performance and the accuracy even if the sketch provided has only a limited amount of information.

6.1 Future Work

For future work, it would be interesting to explore several changes in the DSL now that SKEL results in improving performance when compared to other synthesizers.

Since the sketch format is based on the CUBES [5] DSL, our work is not friendly for users who are not familiar with that DSL. In the future, it would be interesting to implement a parser that converts an SQL query into a language similar to this DSL, thus allowing the improved synthesizer to be more intuitive for users that already have some SQL knowledge.

We can also perform a more extensive evaluation with other metrics, such as considering the size of the solution to return or the number of joins the query must have. For example, in the sketches with aggregates, it would be interesting to use benchmarks with only the aggregate and no filters or joins to understand how the sketch performs without other variables.

As previously stated, SKEL has a significant impact on performance, and so the DSL could also be extended to include more productions, such as always generating aggregate functions for every column of the input tables. This way, the user does not need to explicitly specify in the specification that a specific aggregate function needs to be used in order for it to be generated in the grammar. The problem with this approach is that more filter productions will also need to be generated for the new possible aggregate columns, so it is crucial to understand the bottleneck of this extension.

Bibliography

[1] ALUR, R., BODIK, R., JUNIWAL, G., MARTIN, M. M. K., RAGHOTHAMAN, M., SESHIA, S. A., SINGH, R., SOLAR-LEZAMA, A., TORLAK, E., AND UDUPA, A. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design* (2013), pp. 1–8.

[2] ALUR, R., SINGH, R., FISMAN, D., AND SOLAR-LEZAMA, A. Search-based program synthesis. *Communications of the ACM 61*, 12 (2018), 84–93.

[3] BARRETT, C., AND TINELLI, C. Satisfiability modulo theories. In *Handbook of Model Checking*. Springer, 2018, pp. 305–343.

[4] BORNHOLT, J., TORLAK, E., GROSSMAN, D., AND CEZE, L. Optimizing Synthesis with Metasketches. *SIGPLAN Not. 51*, 1 (2016), 775–788.

[5] BRANCAS, R. *CUBES: A New Dimension in Query Synthesis From Examples (MSc thesis)*. 2020.

[6] CHEUNG, A., SOLAR-LEZAMA, A., AND MADDEN, S. Optimizing Database-Backed Applications with Query Synthesis. *SIGPLAN Not. 48*, 6 (2013), 3–14.

[7] DAVID, C., AND KROENING, D. Program synthesis: challenges and opportunities. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 375*, 2104 (2017), 20150403.

[8] DE MOURA, L., AND BJØRNER, N. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 2008), C. R. Ramakrishnan and J. Rehof, Eds., Springer Berlin Heidelberg, pp. 337–340.

[9] DONG, L., AND LAPATA, M. Coarse-to-Fine Decoding for Neural Semantic Parsing. *arXiv preprint arXiv:1805.04793* (2018).

[10] FENG, Y., MARTINS, R., BASTANI, O., AND DILLIG, I. Program Synthesis Using Conflict-Driven Learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2018), PLDI 2018, Association for Computing Machinery, p. 420–435.

[11] FRANK, E., AND WITTEN, I. H. Generating Accurate Rule Sets Without Global Optimization. In *Proceedings of the Fifteenth International Conference on Machine Learning* (San Francisco, CA, USA, 1998), ICML '98, Morgan Kaufmann Publishers Inc., p. 144–151.

[12] GULWANI, S. Dimensions in Program Synthesis. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming* (2010), pp. 13–24.

[13] GULWANI, S. Automating String Processing in Spreadsheets Using Input-Output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2011), POPL '11, Association for Computing Machinery, p. 317–330.

[14] GULWANI, S., POLOZOV, A., AND SINGH, R. *Program Synthesis*, vol. 4. NOW, 2017.

[15] GUO, J., ZHAN, Z., GAO, Y., XIAO, Y., LOU, J.-G., LIU, T., AND ZHANG, D. Towards Complex Text-to-SQL in Cross-Domain Database with Intermediate Representation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics* (Florence, Italy, 2019), Association for Computational Linguistics, pp. 4524–4535.

[16] KALASHNIKOV, D. V., LAKSHMANAN, L. V. S., AND SRIVASTAVA, D. FastQRE: Fast Query Reverse Engineering. In *Proceedings of the 2018 International Conference on Management of Data* (New York, NY, USA, 2018), SIGMOD '18, Association for Computing Machinery, p. 337–350.

[17] KALYAN, A., MOHTA, A., POLOZOV, O., BATRA, D., JAIN, P., AND GULWANI, S. Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples, 2018.

[18] KOZA, J. R. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing 4*, 2 (1994), 87–112.

[19] MAILUND, T. Manipulating Data Frames: dplyr. In *R Data Science Quick Reference: A Pocket Guide to APIs, Libraries, and Packages*. Apress, Berkeley, CA, 2019, pp. 109–160.

[20] MANNA, Z., AND WOLPER, P. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS) 6*, 1 (1984), 68–93.

[21] MARTINS, R., CHEN, J., CHEN, Y., FENG, Y., AND DILLIG, I. Trinity: An extensible synthesis framework for data science. *Proceedings of the VLDB Endowment 12*, 12 (2019), 1914–1917.

[22] ORVALHO, P. *SQUARES: A SQL Synthesizer Using Query Reverse Engineering (MSc thesis).* 2019.

[23] ORVALHO, P., TERRA-NEVES, M., VENTURA, M., MARTINS, R., AND MANQUINHO, V. Encodings for Enumeration-Based Program Synthesis. In *Principles and Practice of Constraint Programming* (Cham, 2019), T. Schiex and S. de Givry, Eds., Springer International Publishing, pp. 583–599.

[24] PNUELI, A., AND ROSNER, R. On the Synthesis of a Reactive Module. In *POPL* (1989), ACM, pp. 179–190.

[25] RAMAKRISHNAN, R., AND GEHRKE, J. *Database Management Systems*, 2nd ed. McGraw-Hill, Inc., USA, 2000.

[26] ROUSSEL, O. Controlling a Solver Execution: the runsolver Tool. *JSAT 7* (2011), 139–144.

[27] RYMER, J. R., KOPLOWITZ, R., LEADERS, S. A., MENDIX, K., ARE LEADERS, S., SERVICENOW, G., PERFORMERS, S., MATSSOFT, W., AND ARE CONTENDERS, T. The forrester wave™: Low-code development platforms for AD&D professionals, q1 2019, 2019.

[28] SHI, X., AND WANG, C. Support Aggregate Analytic Window Function over Large Data by Spilling. *arXiv preprint arXiv:2007.10385* (2020).

[29] SOLAR-LEZAMA, A. *Program Synthesis by Sketching*. PhD thesis, University of California at Berkeley, USA, 2008.

[30] SOLAR-LEZAMA, A. The Sketching Approach to Program Synthesis. In *Programming Languages and Systems* (Berlin, Heidelberg, 2009), Z. Hu, Ed., Springer Berlin Heidelberg, pp. 4–13.

[31] SOLAR-LEZAMA, A., TANCAU, L., BODIK, R., SESHIA, S., AND SARASWAT, V. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2006), ASPLOS XII, Association for Computing Machinery, p. 404–415.

[32] TRAN, Q. T., CHAN, C.-Y., AND PARTHASARATHY, S. Query reverse engineering. *The VLDB Journal 23*, 5 (2014), 721–746.

[33] VINCENT, P., IIJIMA, K., DRIVER, M., WONG, J., AND NATIS, Y. Magic quadrant for enterprise low-code application platforms. *Gartner report* (2019).

[34] WANG, C., CHEUNG, A., AND BODIK, R. Synthesizing Highly Expressive SQL Queries from Input-Output Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2017), Association for Computing Machinery, p. 452–466.

[35] WANG, Y., DONG, J., SHAH, R., AND DILLIG, I. Synthesizing Database Programs for Schema Refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2019), PLDI 2019, Association for Computing Machinery, p. 286–300.

[36] YAGHMAZADEH, N., WANG, Y., DILLIG, I., AND DILLIG, T. Type- and content-driven synthesis of SQL queries from natural language. *arXiv preprint arXiv:1702.01168* (2017).

[37] YU, T., ZHANG, R., YANG, K., YASUNAGA, M., WANG, D., LI, Z., MA, J., LI, I., YAO, Q., ROMAN, S., ZHANG, Z., AND RADEV, D. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-{SQL} Task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing* (Brussels, Belgium, 2018), Association for Computational Linguistics, pp. 3911–3921.

[38] ZHANG, S., AND SUN, Y. Automatically synthesizing SQL queries from input-output examples. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (11 2013), pp. 224–234.

[39] ZLOOF, M. M. Query-by-Example: the Invocation and Definition of Tables and Forms. In *Proceedings of the International Conference on Very Large Data Bases, September 22-24, 1975, Framingham, Massachusetts, USA* (1975), D. S. Kerr, Ed., ACM, pp. 1–24.