Sketch-Driven SQL Synthesis

Viviana de Brito Bernardo
vivianabernardo@tecnico.ulisboa.pt
Instituto Superior Técnico, Universidade de Lisboa

ABSTRACT

Nowadays, the amount of data that needs to be analyzed and manipulated is rapidly increasing. Performing these actions is often a complex and cumbersome task. OutSystems is a Low-Code Development Platform whose goal is to simplify the development of many applications with most of the tasks automated by the platform itself. This automation is closely related to the program synthesis problem, which consists in automatically generating a program from a specification.

This work 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 SKEL 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.

1 INTRODUCTION

In recent years, Low-Code Development Platforms (LCDP) have become a crucial tool in the web/mobile application development domain. Forrester [15] and Gartner [18] also estimate a significant market increase for LCDP 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.

Nowadays, many users who have to manipulate large amount of data lack the knowledge on how to perform such manipulations. CUBES [2] is a state-of-the-art SQL synthesizer. A weak point of CUBES is the time it takes to generate an SQL query and the accuracy of these queries since not all correspond to the user intent. To tackle this issue, we introduce SKEL, a synthesizer that receives a sketch as part of the specification. We created our sketch format that contains information about the structure of the query the user intends to generate. By using sketches, our goal is to increase the efficiency of the synthesizer and the overall accuracy of the queries generated.

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

© 2021 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.

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: Table Bridge

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

Table 2: Table Architect

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

Table 3: Output Table

Motivation Example

Suppose that the user wants to generate an SQL query that manipulates Table 1 and Table 2 (input example) into Table 3 (output example). It is possible to provide a sketch that could represent this implementation, guiding the synthesis process more efficiently and accurately. A possible sketch is represented in Example 1.1:

Example 1.1. 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. The intended solution to this problem instance would be the query in Example 1.2.

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.

Example 1.2. 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
```

Contributions This work 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. 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].

This document is organized as follows. Firstly in section 2, we introduce background concepts necessary to understand this documents. In section 3, we analyze techniques and state-of-the-art synthesizers that are relevant for our developed work. Next, in section 4, we describe our new sketch format and the SKEL tool implementation in detail, followed by section 5, where we evaluate this implementation, discuss the results obtained. Finally, we conclude in section 6 with some final remarks about our work and insights into future work.

2 FUNDAMENTAL CONCEPTS

In this section, we present fundamental concepts necessary to understand the rest of the document.

2.1 Satisfiability Modulo Theories

A theory is a set of axioms in the underlying logic. 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 Theories (SMT) problem [1] 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.

2.2 Program Synthesis

Program synthesis consists of automatically finding a program, written in a given Domain Specific Language (DSL) (see Definition 2.1), that satisfies a given specification (user intent) [7]. This well-studied problem is mostly characterized by three main components: intent specification, program space, and search technique [6].. Program synthesis has been a rapidly growing research area since the sixties and is considered the "Holy Grail" of Computer Science [7] and "one of the most central problems in the theory of programming" [12] as Amir Pnueli described it.

Definition 2.1. The DSL represents the language in which synthesized programs are written, defining its syntax and semantics.

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

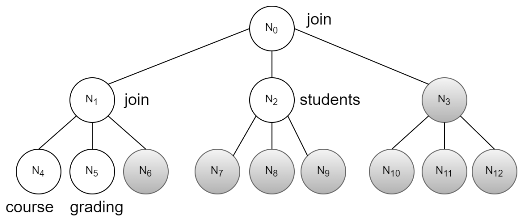

Figure 1: K-tree representation example

2.3 Programming by Example

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.

3 RELATED WORK

The SQL synthesis problem is an active research area. In recent year, many SQL synthesis tools were developed [2, 3, 5, 10, 19, 20, 22, 23]. In this section, we describe state-of-the-art techniques proposed in the literature for program and query synthesis.

3.1 Sketching

Sketching was introduced by Solar-Lezama [16, 17] and it allowed users to provide as a specification a skeleton that expresses the high-level specification about a program they want to create.

Definition 3.1. 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.

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. The SKETCH syntax is very similar to the C language with the additional symbol ?? representing the holes in the sketch. In this Example 3.2, the synthesizer will eventually replace the ?? symbol with a 2, thus satisfying the assertion statement.

Example 3.2. Basic sketch structure:

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

3.2 SQUARES

SQUARES [10] is built on top of the Trinity framework [9]. As a specification, the user has to provide an input-output example. However, this synthesizer can also receive as input some extra information regarding the intended query, including:

- a list of aggregation functions;
- a list of constants;
- columns 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

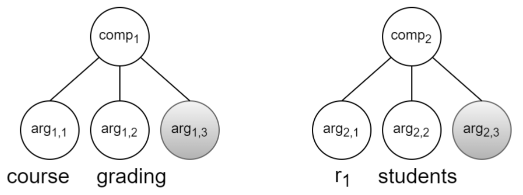

Figure 2: Line-based representation example

is also the largest number of parameters of any given grammar component. In Figure 1 is an example of the k-tree representation of the query $join(join(course, grading), students)$. This encoding grows the number of nodes exponentially with the tree's maximum depth. To mitigate this intractability, SQUARES introduces the concept of line-based representation [11].

In line-based representation, a program is represented as a sequence of lines where each line corresponds to a DSL operation. The goal is to represent a program where each line corresponds to a tree of depth 1. With this implementation, the authors' goal is to reduce the overhead of SMT solving. In Figure 2 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. We observe that in tree-based encoding, the number of nodes increases exponentially with the number of lines of code (see Equation 1), which contrasts with the linear increase of the line-based encoding (see Equation 2).

$$\frac{k^{l+1}-1}{k-1} \quad (1) \qquad\qquad (k+1)*l \quad (2)$$

The results showed that this encoding 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.

3.3 CUBES

CUBES [2] was built on top of SQUARES, so every aspect described in the previous subsection, 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. CUBES has three different versions: a sequential and two parallel solutions.

Sequential Synthesis. CUBES-Seq extends the 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.

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

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

Figure 3: DSL used by the CUBES synthesizer

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

Table 4: Table Students

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

Table 5: Table Grading

T does not contain C, that query is removed, as shown in Example 3.3. This optimization helps to limit the search space and, consequently, improve the search time.

Example 3.3. Consider Table 4, and Table 5 since Table 4 does not contain the column "Grade", we could never generate a program with the operation $filter(Students, Grade == 18)$, so this program would be immediately pruned.

Since the CUBES implementation uses SMT solvers, it can also take advantage of UNSAT cores to prune the search space even further. 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 SQUARES[10] and Scythe [19].

Parallel Synthesis. Two parallel SQL Synthesizers were implemented: CUBES-Port, which uses portfolio solving, and CUBES-DC, which uses a divide and conquer technique.

* Portfolio solving CUBES-Port diversifies 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 CUBES-DC splits the problem into smaller sub-problems, cubes, that can be solved by each of the processors. It divides the search space into different cubes, each corresponding to particular sequences of operations of the DSL in which the arguments for those operations are still undetermined. The cubes are then distributed over different threads.

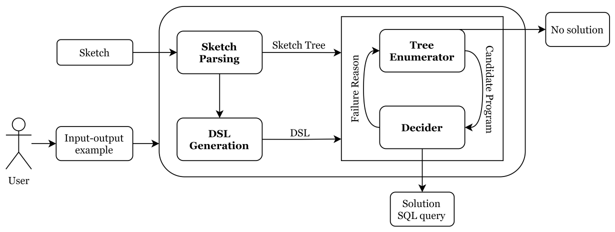

Figure 4: SKEL's architecture

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

Figure 5: Sketch Functions

4 SKETCH-DRIVEN SQL SYNTHESIS

In this section, we describe SKEL, a **Ske**tch-driven SQ**L** synthesizer developed on top of the sequential version of CUBES [2]. 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. SKEL was implemented in Python 3.8 and R version 3.6.3. R was used to validate the program generated [8], using the input-output example, and then translate it to a program in SQL. SKEL also uses the Z3 SMT solver [4] to solve the SMT formulas generated by the synthesizer.

Figure 4 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).

4.1 Sketch Format

In this subsection, 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 5 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 4.1.

Example 4.1. 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. An important note is that this final line is optional, depending on the information the user has. This optional final line takes *distinct* and/or *orderby* as optional arguments and selects the columns used in the output table:

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

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 4.2 shows two different uses of line holes. The first one corresponds to when the user references an unknown line, and the second one to when he does not know the exact number of lines.

  Example 4.2. Line hole:

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

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

- **Previous line hole** - 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 4.3 we show a possible usage of this type of hole.

  Example 4.3. Previous line hole:

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

- **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 4.4.

Example 4.4. Children hole:

$T1 = \texttt{filter}\ (students,\ ??)$

$T2 = \texttt{summarise}\ (T1,\ max\_grade = max(grade),\ ??)$

$out = \texttt{select}\ (name,\ max\_grade)$

- **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 consider the children to be out of order so that the user can freely provide any children that he has information on. 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 their concrete expressions, 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 4.5.
  – The user only knows some of the children without knowing how many exist in total. This 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 4.5.

Example 4.5. Parent hole:

$T1 = \texttt{filter}\ (students,\ id > 3)$

$T2 = ??\ (id,\ ??,\ ??)$

$out = \texttt{select}\ (name,\ max\_grade)$

$T1 = \texttt{filter}\ (students,\ id > 3)$

$T2 = ??\ (max\_grade = max(grade),\ ??+)$

$out = \texttt{select}\ (name,\ max\_grade)$

- **Select hole** - The SELECT line, if exists, is the last line of the sketch and its holes follow particular rules:
  – The columns given to the SELECT function have to be exactly the ones the output must 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 is similar to 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, as shown in Example 4.6.

Example 4.6. Select hole:

$T1 = \texttt{filter}\ (students,\ id > 3)$

$T2 = \texttt{summarise}\ (T1,\ max\_grade = max(grade),\ id)$

$out = \texttt{select}\ ??\ (name,\ max\_grade)\ \texttt{order by}\ (??)$

- **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. In Example 4.7 we demonstrate a possible use of this functionality given three tables - stock, supplier, and parts.

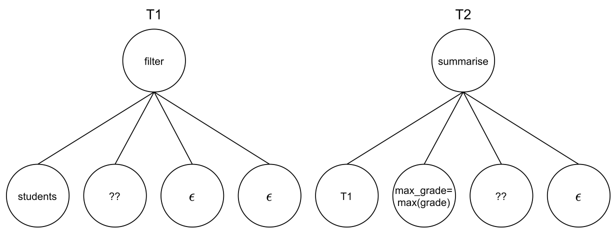

Figure 6: Sketch Tree Example

Example 4.7. 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 subsection 4.1, we start parsing it by creating the tree structure of the sketch. In this subsection, 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 6 illustrates the tree structure for the sketch in Example 4.4. The four main types of lines are:

- **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 a named empty tree. 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) 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:

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

Besides the child types described in subsection 4.1, the sketch structure also allows 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. This last line, when present, is treated differently since the CUBES DSL does not contain the `SELECT` component. 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.

* LOC value In the sketch, we also consider the maximum and minimum LOC possible. 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. 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 that will be used during synthesis. The DSL follows the same format as the CUBES DSL (see Figure 3). Each production of the DSL has a non-negative identifier, $id \mapsto IN_0$. By making a change in the configuration, we can opt between having the grammar generated as 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.

To generate the grammar from 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. This information is stored so that the synthesizer can later generate the grammar accordingly.

4.3 Sketch Filling

After having generated the grammar, the synthesizer maps every known production in the sketch structure to its corresponding DSL production identifier. In this subsection, we explain how this matching is done in order to complete the sketch structure.

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. 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.

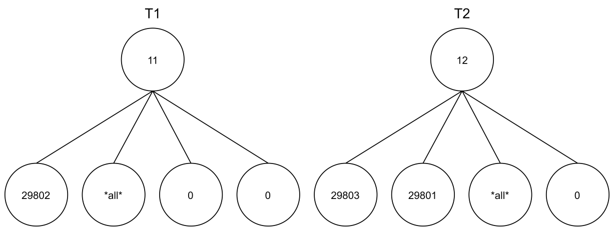

Figure 7: Filled Sketch Tree Example

After this phase, the sketch tree example in Figure 6 will be filled with the DSL identifiers as shown in Figure 7. 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. 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.

The synthesizer then uses this sketch structure to generate the trees with the additional sketch information.

Optimizations. We implemented two 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 7, we obtain an optimized tree shown in Figure 8 where we can see a much smaller set of different DSL identifiers allowed for the leaves when compared to the non-optimized sketch filling method.

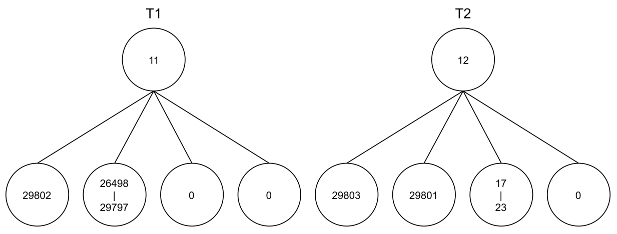

Figure 8: Filled Sketch Tree Example Optimized

4.4 Sketch Encoding

After having a complete sketch structure, the synthesizer will enumerate programs with the help of an SMT solver. 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. 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:

$$\forall\, 1 \leq i \leq n : \bigvee_{f \in Func(D)} op_i = id(f) \tag{3}$$

If we have information on the DSL identifiers in the sketch structure for a specific root, we have the following constraints:

$$\forall\, 1 \leq i \leq n : \bigvee_{f \in SRoot(i)} op_i = id(f) \tag{4}$$

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:

$$\forall\, 1 \leq i \leq n,\ 1 \leq j \leq k : \bigvee_{p \in Prod(D)} a_{ij} = id(p) \tag{5}$$

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

$$\forall\, 1 \leq i \leq n,\ 1 \leq j \leq k : \bigvee_{p \in SLeaf(i,j)} a_{ij} = id(p) \tag{6}$$

From now on, let $Leaf(i, j)$ represent the possible children productions for the j-th leaf of the i-th tree.

* Unordered and incomplete lines 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:

$$\forall\, i \in UL : \bigwedge_{c \in S\_all\_children(i)} \bigvee_{1 \leq j \leq NC_i} \bigvee_{p \in c_j} a_{ij} = id(p) \tag{7}$$

* Type matching We must ensure the functions arguments format is followed. 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:

$$\begin{gathered}\forall\, 1 \leq i \leq n,\ p \in Root(i),\ arity(p) < j \leq k : \\ (op_i = id(p)) \implies (a_{ij} = id(empty))\end{gathered} \tag{8}$$

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 9. On the other hand, if there exist productions with the correct type, then the constraint in Equation 10 is added.

$$\begin{gathered}\forall\, 1 \leq i \leq n,\ p \in Root(i),\ 1 \leq j \leq k : \\ \neg(op_i = id(p))\end{gathered} \tag{9}$$

$$\begin{gathered}(op_i = id(p)) \implies \\ \left(\bigvee_{s \in \{t \in leaf(i,j) : Type(t) = Type(p,j)\}} a_{ij} = id(s) \right)\end{gathered} \tag{10}$$

* Free lines If we have 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:

$$\begin{gathered}\forall\, h \in FL : \bigvee_{i \in ET} \left[\left(\bigvee_{p \in SRoot(h)} op_i = id(p) \right) \wedge \right. \\ \left. \left(\bigwedge_{1 \leq j \leq k} \bigvee_{p \in SLeaf(h,j)} a_{ij} = id(p) \right) \right]\end{gathered} \tag{11}$$

The SMT solver will then try to return a valid solution, with all the remaining holes filled. If it exists, the solution is given to the decider to check if the program is satisfiable. 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.

5 EXPERIMENTAL RESULTS

In this section we evaluate SKEL. The results were obtained using an Intel® Xeon® CPU E5-2630 v2 @ 2.60GHz, with 64GB of RAM. We ran the benchmarks using runsolver [14] with a 10 minutes limit. We took the benchmarks used in CUBES and prepared a subset of them to include several sketches, each one with different properties, resulting in a total of 100 benchmarks:

- 29 instances extracted from exercises of the textbook *Database Management Systems* [13];
- 22 + 39 instances from the benchmarks used in Scythe [19], collected from recent and top-rated posts from StackOverflow[3], respectively;
- 9 instances from Spider [21], a dataset of SQL queries and respective natural language descriptions.

We analyzed each instance and converted its SQL ground truth solution into our sketch format without any holes, a full sketch. We then proceeded to replace parts of the solution with holes. The sketches can be split into the following categories:

- Sketch with **no root**, i.e., sketches where the parent is replaced by a hole;
- Sketch with **no children**, i.e., sketches where the children are replaced by holes;
- Sketch with **no root and no filter** where the parents and the filter production are replaced by a hole;
- Sketch with **no root and no aggregate** where the parents and the aggregate production are replaced by a hole;
- Sketch with **no children except filter**, i.e., the sketch children are replaced by holes except the filter production;
- Sketch with **no children except aggregate**, i.e., sketch children are replaced by holes except the aggregate;

We use the following evaluation metrics to analyze SKEL:

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

5.1 Baseline Analysis

We start the evaluation by comparing two of the most simple sketch formats with a baseline using no sketch in Figure 9. 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. Using sketches with no children, the synthesizer can solve a total of 92 instances, which is a considerable improvement compared with the baseline. With a sketch that had no root, the synthesizer was able to solve all 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*, since it has less information. We can also see that the time, the synthesizer took to return a valid solution decreased significantly when a sketch was provided.

We then analyzed the number of programs generated by the synthesizer before a solution is found. Around the 70th instance, the benchmarks with no sketch were already exceeding 2000 tested programs before returning a solution. Regarding the *sketch with no children* we start seeing a more notable increase in attempted programs by the 85th instance. After further analysis, we concluded that this increase 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. Regarding the *sketch with no root* the number of attempted programs is very low which is expected because the synthesizer has more information than previous sketches.

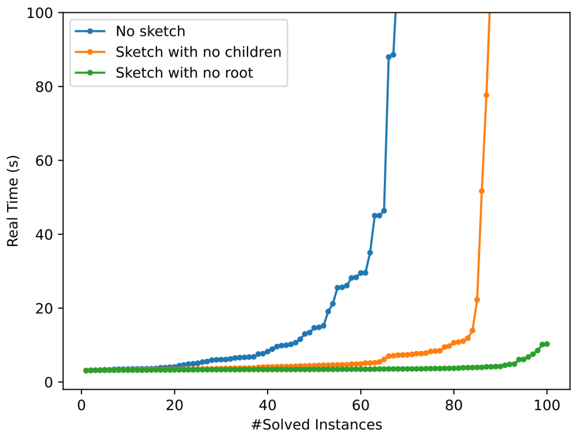

Figure 9: Baseline efficiency

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

Table 6: Baseline metrics

In Table 6, we can observe that the accuracy and precision metrics increase with the more information the sketch has, as expected. When 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 solved correctly. The slight increase is expected since the sketches with no children do not contain much more information than no sketch. When a *sketch with no root* is provided, we see that 83% of the instances solved were consistent with the ground truth. The instances that were not correct are mostly cases where parents that should be classified as `natural_join` are instead classified as `left_join` or `semi_join`, mainly due to the ambiguity of the specification.

With this information, we can see the positive impact of giving the synthesizer a sketch, which results in a boost in performance since a solution can be found faster and more accurately.

5.2 Optimization Analysis

In this subsection we analyze the impacts of the optimizations described in subsection 4.3. In Figure 10, we observe that the optimization has a positive impact on the efficiency. With the

[3]https://stackoverflow.com

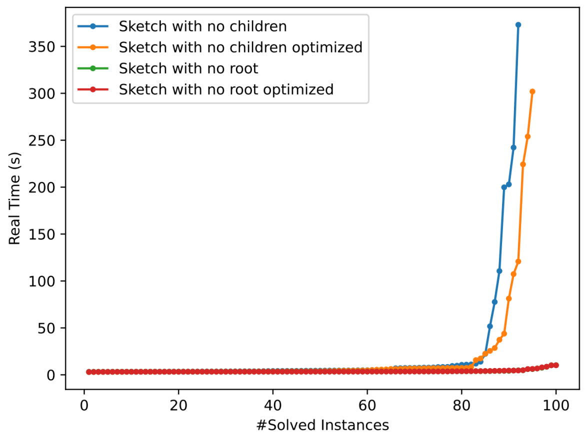

Figure 10: Optimized efficiency

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

Table 7: Optimized version metrics

sketch with no children being 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. Regarding the *sketch with no root*, we cannot clearly see an improvement in efficiency, with both sketches solving the 100 instances. 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.

In Table 7 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 optimization, resulting in an accuracy and precision of 83% without optimization and 87% with. Although the efficiency of both approaches is very similar, we still observed an increase in accuracy and precision.

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. For the rest of this section we will only use the optimized version.

5.3 Sketches with Filters

We used a subset of 48 instances for which the respective ground truth contains at least one filter for this evaluation. In Figure 11 we can see that the two new sketches perform better than the *sketch with no children*. It is also expected that both of these new sketches perform worse than the *sketch with no root* since 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

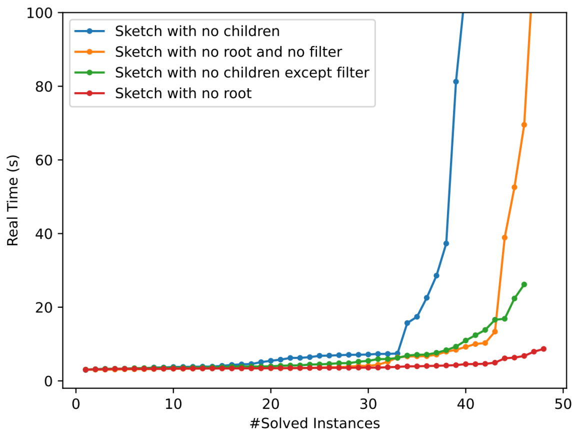

Figure 11: Filter sketches efficiency

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

Table 8: Filter sketches metrics

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.

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.

Looking at Table 8, 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 ">=" or with only ">" if the respective corner case is not included in the specification. Using a *sketch with no root and 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, the precision of this sketch is lower than the precision of the *sketch with no children*. One explanation could be 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. 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%. The *sketch with no root* still has best performance, as expected, with a 91,67% of accuracy and precision.

We concluded that the filter conditions significantly impact the efficiency and accuracy of the synthesizer.

5.4 Sketches with Aggregates

We used a subset of 49 instances for which the respective ground truth contains an aggregate. In Figure 12 we see that there is an improvement in the performance of all sketches when compared to the *sketch with no children*. The two new sketches perform

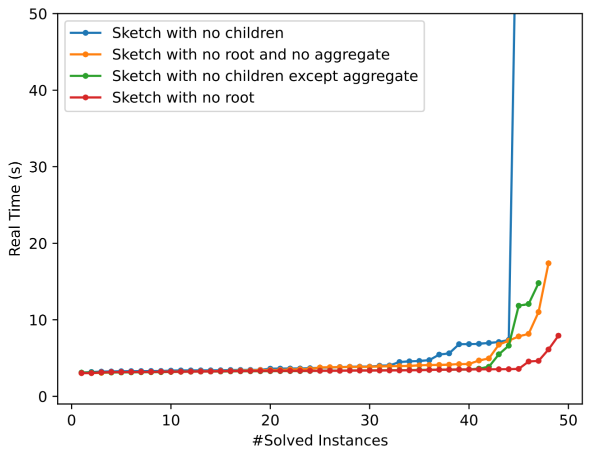

Figure 12: Aggregate sketches efficiency

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

Table 9: Aggregate sketches metrics

very similarly in terms of efficiency and do not perform as expected when compared with each other. After further analysis, we understood the reason for this was that SKEL does not generate an extensive amount of aggregate productions in the DSL. This substantially reduces the number of productions the synthesizer must consider before returning a solution, could explain the obtained results. The *sketch with no children* generates the most programs before returning a solution followed by the two new sketches. The *sketch with no root* requires the fewest attempts.

In Table 9, we 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 using the new sketches. This is can be because the synthesizer does not have an extensive number of productions that needs to try for returning 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 is not as prone to make mistakes due to the ambiguity of the specification. Overall giving the synthesizer an aggregate will likely not yield better results.

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*. The behavior between the two sketches is the expected, and it can explain the results obtained in the efficiency and the attempted programs since the synthesizer could have been taking longer so that it could return the correct one.

The instances that were not solved correctly by the *sketch with no children except aggregate* were mainly due to an incorrect filter condition. This sketch also performed worse than the *sketch with no children* probably because the synthesizer traversed the search space in a different order. Since it is much more likely to return a wrong filter condition than an aggregate due to the difference in the number of possible productions generated, adding this additional information does not improve SKEL's performance. The *sketch with no root* continues to outperform the others by having the most information.

6 CONCLUSION

In this work, we proposed a new query synthesizer, SKEL, that receives 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 allowing the user to provide a sketch that fits different situations.

We then evaluated SKEL comparing it with a baseline, namely the CUBES-Seq version, while using simple sketches. The increase in performance was notorious as expected. Nonetheless, there was also an increase in accuracy. The optimizations performed in our synthesizer also proved to increase the efficiency but mainly the accuracy of the synthesizer. We can conclude that SKEL boosts the performance and the accuracy even if the sketch provided has only a limited amount of information.

Since the sketch format is based on the CUBES [2] 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.

For future work, 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. Since SKEL has a significant impact on performance, 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 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.

REFERENCES

[1] Barrett, C., and Tinelli, C. Satisfiability modulo theories. In *Handbook of Model Checking*. Springer, 2018, pp. 305–343.
[2] Brancas, R. *CUBES: A New Dimension in Query Synthesis From Examples (MSc thesis)*. 2020.
[3] Cheung, A., Solar-Lezama, A., and Madden, S. Optimizing Database-Backed Applications with Query Synthesis. *SIGPLAN Not. 48*, 6 (2013), 3–14.
[4] 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.
[5] Dong, L., and Lapata, M. Coarse-to-Fine Decoding for Neural Semantic Parsing. *arXiv preprint arXiv:1805.04793* (2018).
[6] 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.
[7] Gulwani, S., Polozov, A., and Singh, R. *Program Synthesis*, vol. 4. NOW, 2017.
[8] 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.
[9] 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.
[10] Orvalho, P. *SQUARES: A SQL Synthesizer Using Query Reverse Engineering (MSc thesis)*. 2019.
[11] 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.
[12] Pnueli, A., and Rosner, R. On the Synthesis of a Reactive Module. In *POPL* (1989), ACM, pp. 179–190.

[13] Ramakrishnan, R., and Gehrke, J. *Database Management Systems*, 2nd ed. McGraw-Hill, Inc., USA, 2000.

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

[15] 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.

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

[17] 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.

[18] Vincent, P., Iijima, K., Driver, M., Wong, J., and Natis, Y. Magic quadrant for enterprise low-code application platforms. *Gartner report* (2019).

[19] 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.

[20] 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).

[21] 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.

[22] 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.

[23] 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.