CUBES: A New Dimension in Query Synthesis From Examples

Ricardo Miguel Bacala Brancas
Instituto Superior Técnico, Universidade de Lisboa

ABSTRACT

In recent years, more and more people are seeing their work depend on data manipulation tasks. However, many of these users do not have the background in programming required to write complex programs, and, in particular, SQL queries. The solution for this problem is Program Synthesis: the task of automatically deriving a program from a specification. Even though in the past decade many advances have been made in program synthesizers, current tools do not take advantage of the increase in number of cores per processor.

We propose CUBES, a parallel program synthesizer for the domain of SQL queries using input-output examples. We use SQUARES as a starting point, and modify it by extending its Domain Specific Language, changing how programs are enumerated and introducing new forms of pruning. We then use this new synthesizer, CUBES-SEQ, as a building block for the development of a parallel SQL synthesizer. In particular, we explore portfolio and divide-and-conquer approaches, which we implement in CUBES-PORT and CUBES-DC, respectively.

Finally, we perform an extensive analysis of CUBES, comparing it with previous state of the art, on on around 4000 queries from different domains. We show that CUBES-SEQ is generally faster than SQUARES and SCYTHE. We also show, that using parallelism provides a significant performance improvement and that CUBES-DC scales better with the number of available processors than CUBES-PORT.

1 INTRODUCTION

In the age of digital transformation, more and more people are being re-assigned to tasks that require familiarity with programming or database usage. However, many users have limited knowledge in these areas [30]. A crucial tool for accelerating this digital transformation are Low-Code Development Platforms (LCDPs), which according to Gartner, will account for more than 65% of application development activity by 2024 [10]. These platforms allow users with very little programming knowledge to quickly and easily develop digital solutions. However, one area that is still lacking is the implementation of custom domain logic. In the particular case of database manipulation, it is common that new data analysts using these tools are domain experts, but lack the technical skills to build queries in a language such as SQL. As a result, several new systems have been proposed in order to automatically handle table manipulations in R or generate SQL queries for relational databases [7, 8, 19, 27].

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

Figure 1 illustrates an input-output example with two input tables (Courses and Grades) and an output table. The output table corresponds to counting the number of grades in each course. In this example, the goal is to synthesize the following SQL query:

CourseID	CourseName
10	Programming
11	Algorithms
12	Databases

(a) The `Courses` table.

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

(b) The `Grades` table.

CourseName	GradeCount
Programming	4
Algorithms	2
Databases	3

(c) The output table.

Figure 1: Two input tables: `Courses` and `Grades`. Output table: number of grades per course.

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

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

In this work, we introduce CUBES, a novel parallel synthesizer for Structured Query Language (SQL) queries. We start by extending an existing synthesizer, SQUARES, in order to improve its performance and expand the range of queries it supports. Next, we use that new synthesizer as a building block for the development of parallel synthesis algorithms. We implement three modes of operation in CUBES:

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

This document is organized as follows. Section 2 discusses sequential state of the art tools for the synthesis of SQL queries that are relevant to our work. Following that, in section 3, we present our new sequential synthesizer, CUBES-SEQ, that improves and extends

the Squares synthesizer. In section 4, we propose Cubes-Port and Cubes-DC, two new parallel synthesizers that adapt techniques used in Parallel Constraint Reasoning solvers to the field of Program Synthesis. Next, in section 5 we evaluate the different configurations of our solver and compare them to previous state of the art in SQL synthesis. Finally, we conclude with some final remarks in section 6.

2 RELATED WORK

In recent years, many SQL synthesis tools have been proposed. These tools vary greatly in the types of specification they require, with some using natural language [14, 23, 28, 29], some using input-output examples [5, 7, 8, 15, 19, 24, 25, 27, 30], and others using multi-modal specifications [4]. In this chapter we focus on two tools, Scythe [27] and Squares [5, 19], which use input-output examples as their specification. We choose these tools because: (i) examples are often readily available to users and are easy to understand even with limited technical knowledge, (ii) they have expressive Domain Specific Languages (DSLs) which cover many common queries, (iii) they are very efficient when compared with other SQL synthesis tools, (iv) Squares' DSL, in particular, is very easily extended, because Squares is built on top of the Trinity framework [16], and (v) their source code is available online and can be integrated in our tool.

2.1 Scythe

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

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

In the second phase, after the possibly correct abstract queries have been enumerated, Scythe tries to instantiate them until a correct concrete query is found. Two optimizations are proposed that make the second phase more efficient:

(1) Equivalence classes are used to group programs so that the number of filter conditions that must be evaluated is reduced;
(2) Using the over-approximation rule, we know that all rows in the output table of a query q that results from the instantiation of an abstract query $\tilde{q}$, must also be present in the output table of that abstract query. As such, scythe represents intermediate tables as a tuple $(\tilde{T}, b)$ where $\tilde{T}$ is the output of the corresponding over-approximation of the abstract query and b is a bitvector with as many bits as there are rows in $\tilde{T}$, and where bit i represents if row i of $\tilde{T}$ is present in the intermediate table. This allows Scythe to reduce memory requirements and execution time [27]. Even so, Scythe's memory usage depends heavily on the size of the input and output tables.

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

2.2 Squares

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

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

Squares uses a Domain Specific Language (DSL) to specify the space of possible programs. This DSL is inspired by the operations available in the popular R data-manipulation library, `dplyr`, from `tidyverse` [2]. When using a DSL for enumeration, programs must be translated into some programming language in order to be evaluated and returned to the user. In Squares, the DSL is translated into R for evaluation. However, Squares is also a SQL synthesizer and as such an automated translation layer is used to convert the generated R program into a SQL query when presenting the final answer.

Figure 2 shows that the synthesizer itself is composed of the Program Enumerator and the Program Verifier. It receives a specification from the user and, if the synthesis is successful, returns an R program that satisfies the specification. This R program is then automatically translated into an equivalent SQL query.

2.2.1 Program Enumeration. At the core of Squares is a Program Enumerator. The purpose of the Program Enumerator is to continuously generate new candidate programs based on the specification. Programs are enumerated with the help of an Satisfiability Modulo Theories (SMT) solver, using the line-based representation introduced by Orvalho et al. [18]. Programs are also enumerated in increasing number of lines of code.

2.2.2 Program Verification and Translation. In order to evaluate the candidate programs and check if they satisfy the user's specification, Squares translates them into R. Then, the R program is executed using the input example that the user provided. The output is then compared with the expected output according to the example. This comparison treats tables as a multi-set of rows, meaning that row

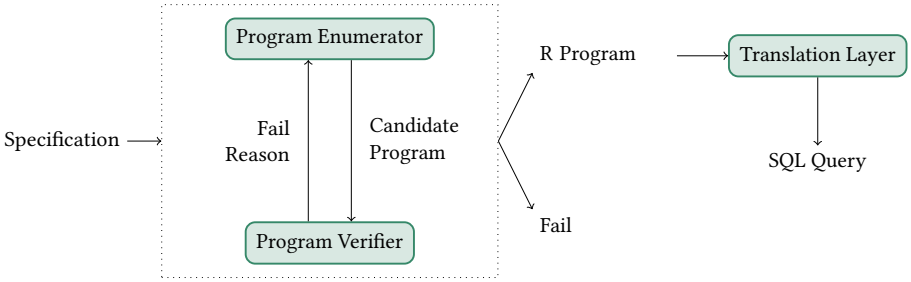

Figure 2: Diagram of Squares architecture.

$table \rightarrow input$ | `natural_join`($table$, $table$)
| `natural_join3`($table$, $table$, $table$)
| `natural_join4`($table$, $table$, $table$, $table$)
| `left_join`($table$, $table$)
| `inner_join`($table$, $table$, $joinCondition$)
| `cross_join`($table$, $table$, $crossJoinCondition$)
| `filter`($table$, $filterCondition$)
| `summarise`($table$, $summariseCondition$, $cols$)
| `mutate`($table$, $summariseCondition$)
| `anti_join`($table$, $table$, $cols$) | `semi_join`($table$, $table$)
| `union`($table$, $table$) | `intersect`($table$, $table$, col)

Figure 3: DSL used by the Cubes synthesizer. New components are highlighted in bold.

order is ignored. If the tables match, a solution to the problem has been found.

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

3 SEQUENTIAL SYNTHESIS

This work uses Squares as a starting point for creating a parallel SQL synthesizer. In this section we describe the changes made to Squares that are not directly related to multiprocessing. In particular, how the range of supported programs was extended and how new forms of pruning were introduced in order to improve synthesis performance. From now on, we refer to the improved version of Squares as Cubes-Seq.

3.1 Extending the Domain Specific Language

In order to support a wider range of programs, Squares' DSL was modified to be more expressive. In this section, we will describe those changes. The new DSL is presented in Figure 3.

Regarding DSL components, the follow components have been altered:

- the `select` component was removed from the DSL and introduced as a post-processing step;
- the `inner_join*` components have been renamed to `natural_join*`, while `bind_rows` was renamed to `union`;
- two complex join operations have been added: `inner_join` and `cross_join`;
- the `filters` component has been removed, while the `filter` component was changed so that it supports compound filter conditions, while removing redundant combinations;
- the `mutate` and `semi_join` have been added to the DSL;
- the semantics of `intersect` and `anti_join` have been modified.

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

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

3.2 Quantifier-Free Finite Domain Theory

When using constraint solvers, a lower-level encoding is typically more efficient (provided that the conversion from high-level to low-level is polynomial). Since all the variables used in Cubes-Seq's encoding are either bounded integers or Boolean variables, a possible way to improve performance is to use bit-blasting. Bit-blasting means converting all variables in the SMT formula to Boolean variables and all constraints to Conjunctive Normal Form (CNF). To do this, the integer variables are first converted to bit vectors, and then the bit-vectors are converted to sets of Boolean variables. The constraints are updated to reflect these changes and then transformed into CNF. The result is a propositional logic formula in CNF that can be solved using an off-the-shelf Propositional Satisfiability (SAT) solver.

The SMT solver used by Cubes-Seq, Z3 [6], implements a theory that performs all these steps automatically, including using an internal SAT solver to solve the resulting formula. This theory is called Quantifier-Free Finite Domain (QF_FD).

3.3 Deducing Invalid Programs

One way to improve a program synthesizer is to reduce the number of incorrect programs that must be tested before finding a solution.

In the case of Squares, a common example are programs where at some point a column that does not exist in the current context is referenced. Consider the following program, which uses the tables from Figure 1 and the DSL from Figure 3:

```
df1 = filter(Courses, grade == 'A')
```

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

We introduce a new form of pruning that eliminates these invalid programs. All component arguments are annotated with a pair of sets of columns. These annotations are then used to further constrain the set of programs that can be return by the program enumerator. In Example 3.1 we show how these annotations can be used to force all `filter` lines to always be valid. Note that the second annotation is only needed for some argument types, in order to record extra information. One such case is presented in Example 3.2.

Example 3.1. Consider again the previous program, which contains only one line: `filter(Courses, grade == 'A')`. That line takes two arguments: `Courses` and `grade == 'A'`.

We annotate all arguments of type *table* with the columns they contain, so in this case `Courses` would be annotated with {`CourseID`, `CourseName`}. Furthermore, we annotate filter condition arguments with the columns they require to be present in order do produce a valid program. In this case `grade == 'A'` would be annotated with {`grade`}. Finally, we encode that all filter operations must be such that all the columns in the annotation of the second argument appear in the annotation of the first argument in order to be valid.

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

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

In Figure 4 we show the inference rules for all the components of our DSL. Using these rules, we can infer from the arguments of a given operation what columns would be present in the output table if the line were executed. By extension, we can also determine invalid lines because no rule will be applicable to them. The contents of each annotation are described in Figure 5.

Example 3.2. Consider the following *summariseCondition*: `maxStudentID = max(StudentID)`. The first annotation of a *summariseCondition* corresponds to the columns that are "used", that is, the columns that must be present in order for the condition to be

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

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

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

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

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

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

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

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

$$\frac{summariseCondition' \subseteq table'_1 \qquad cols' \subseteq table'_1 \qquad (cols' \cap summariseCondition'') = \emptyset}{output' = summariseCondition'' \cup cols'}\ \textsc{Summarise}$$

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

Figure 4: Inference rules used to determine valid programs. A' denotes the first annotation of element A, while A'' denotes the second annotation. Where not mentioned, it is assumed that the second annotation is $= \emptyset$.

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

Figure 5: Description of the semantics of each annotation. A' denotes the first annotation of element A, while A'' denotes the second annotation.

applicable. In this case the first annotation would be {`StudentID`}. The second annotation corresponds to the columns that are generated by the *summariseCondition*, in this case: {`maxStudentID`}.

When this condition is used in a `mutate` operation, rule Mutate from Figure 4 states that if all of the required columns (first annotation) are present in the input, then we can conclude that the output table will be comprised of all columns that were already present in the input table, along with the generated columns (second annotation).

The rules in Figure 4 are implemented directly as SMT constraints, which means the corresponding invalid programs are never generated.

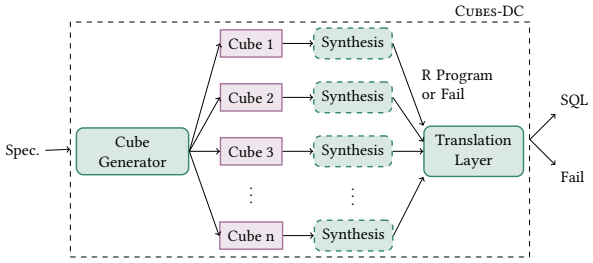

Figure 6: Diagram of Cubes' architecture when using divide and conquer.

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

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

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

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

Figure 7: Order in which operations are chosen when using Static Cube Generation.

4 PARALLEL SYNTHESIS

In this section we discuss how techniques used in Parallel Constraint Reasoning solvers can be adapted in order to create a Parallel Program Synthesizer. In the first subsection, we introduce Cubes-Port, the portfolio mode of Cubes, which takes advantage of a portfolio of synthesizers in order to produce faster results. Next, in subsection 4.2, we introduce Cubes-DC, the divide-and-conquer mode of Cubes, that divides the synthesis problem into several sub-problems and then solves those sub-problems in parallel.

4.1 Portfolio

In the last decade, the portfolio approach has been successfully applied to several decision problems [12]. In this technique, as soon as one of the processes finds a solution, the search ends and there is no need to completely explore the rest of the search space. Therefore, the main goal of a portfolio is to diversify the exploration of the search space by making each thread explore the same search space in different ways.

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

Internally, Cubes uses an SMT formula to enumerate candidate programs. Hence, one can devise a portfolio by providing the same SMT formula to each process, but using different configurations of the Z3 SMT solver [6] in order to diversify the search. A complementary option is to change the active techniques from Cubes in each process, thus changing the learned constraints in the SMT formula and the subsequent search. Another complementary alternative is to use different synthesizers in parallel. Each synthesizer such as Squares or Scythe uses different techniques, thus increasing the diversity in the exploration of the search space.

4.2 Divide and conquer

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

Inspired by previous work in solving Propositional Satisfiability formulas [26], we present a strategy to split the Program Synthesis search space in many sub-problems that should be easy to solve. The overall architecture is illustrated in Figure 6. In our context, each sub-problem is represented by a cube: a sequence of operations from the DSL, such that the arguments for the operations are still to be determined. Consider the following cube as an example: `[filter, natural_join]`, which represents the section of the search space composed by programs with two lines, where the first operation is a `filter` and the second operation is a `natural_join`. Each process receives a specific cube to be filled in and determines if a solution can be reached for that particular cube. If the cube cannot be completed such that it satisfies the input-output examples, then the cube is deemed unsatisfiable and the process requests a new cube to explore. Observe that each cube corresponds to a particular sequence of operations, and as such, there is no intersection in the search space of each process.

This approach is very similar to using guiding paths in Parallel SAT [11, 17]. A guiding path is an assignment to a subset of the variables of the formula that defines a partition of the search space. The task is then to generate several (disjoint) guiding-paths that can be solved in parallel. It is also similar to using 2-step enumeration [8, 9, 21, 27, 29]. In this case, the first step would consist in generating the cubes using a graph-based algorithm, and the second step would consist in filling in each cube using SMT-based enumeration.

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

4.2.1 Static Cube Generation. In static cube generation, cubes are constructed using a static heuristic. However, the sequence of operations to be tried first is not purely a predetermined order to be followed. Instead, the heuristic, presented in Figure 7, selects the operation to be executed next in a given sequence depending on the already selected operations. For instance, if the first operation in a given cube is a `natural_join`, it is unlikely that applying a `natural_join` next will lead to a solution. Therefore, a cube that uses a `natural_join` followed by an `inner_join` is generated before a cube that applies two `natural_join` in sequence.

4.2.2 Dynamic Cube Generation. Considering that the static generation heuristic was empirically designed based on available benchmark instances, its behavior might not be adequate for new instances. Therefore, Cubes also includes a cube generator inspired on Natural Language Processing (NLP) techniques. Since candidate programs are constructed as a sequence of operations, a bigram prediction model can be used to decide the operation to be placed next in a given sequence. Therefore, when choosing the operation for a given position in the sequence, the operation immediately preceding it is used to compute the likelihood that each of the possible choices will lead to the desired program. That is, for each pair of operations (a, b) there is a score, $S_{\mathsf{a,b}}$, that represents the likelihood that using a b operation after an a operation will lead to the desired program. Scores are updated as programs are evaluated in the following way:

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

$$score(p) = \frac{|\textbf{unique}(\texttt{output}) \cap \textbf{unique}(\texttt{expected})|}{|\textbf{unique}(\texttt{expected})|} \quad (1)$$

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

Score updates. For each evaluated program, p, the score, $score(p)$, is used to update the bigram scores. Consider that program p uses the following components: `filter`, `natural_join`, `summarise` (in that order). Then, the scores for the bigrams that appear in the program will be updated as follows:

$$S_{\emptyset,\texttt{filter}} \mathrel{+}= score(p) \quad (2)$$

$$S_{\texttt{filter},\texttt{natural\_join}} \mathrel{+}= score(p) \quad (3)$$

$$S_{\texttt{natural\_join},\texttt{summarise}} \mathrel{+}= score(p) \quad (4)$$

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

$$S_{\emptyset,\texttt{filter}} \mathrel{+}= {}^1\!/_1 \cdot score(p) \quad (5)$$

$$S_{\emptyset,\texttt{natural\_join}} \mathrel{+}= {}^1\!/_4 \cdot score(p) \quad (6)$$

$$S_{\emptyset,\texttt{summarise}} \mathrel{+}= {}^1\!/_9 \cdot score(p) \quad (7)$$

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

Cube selection. Cubes are constructed by adding operations to a sequence. Suppose that the last selected operation is op (in case of the first operation, op is the empty symbol $\emptyset$). In order to decide which operation should follow, the scores for that prefix, S_{op}, are retrieved, normalized and smoothed, using Laplace smoothing [13]. These steps result in a list of probabilities that correspond to the likelihood of each operation. The operation for the current line is then chosen from a distribution using those probabilities. This is done until we have a program of the desired length. A compact tree structure is used to keep track of already generated cubes, as to avoid repetition.

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

- Each time a new program is generated, all scores are multiplied by a number smaller than one, δ, by default 0.99999. This is done so that past information can be gradually forgotten, in order to increase diversification in exploring the search space. These updates are done in batches, in order to not overwhelm inter-process communication.
- A fixed number of processes, by default 2, always solve randomly generated cubes (as long as not previously generated), in order to diversify the search process.

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

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

4.2.3 Optimal and Non-Optimal Solving. As explained in subsubsection 2.2.1, Squares, and by extension Cubes, enumerates programs in increasing size. The same is true for cube generation. However, when splitting the search space, it is common for some processes to finish searching the final cubes for the current program size, while others are still trying candidate programs. Cubes allows these processes to start searching cubes of the next size, so that they do not stall. However, this means that a solution of size n can be found before all programs of size $n-1$ have been explored (and therefore a shorter solution might exist). Cubes allows for the user to choose between:

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

4.2.4 Learning from Unfeasible Cubes. Cubes are implemented by adding supplemental constraints to the SMT solver. A cube stating that the first line should be a `filter` and the second line should be a `summarise` would be implemented as $line_1$ = `filter` $\wedge$ $line_2$ = `summarise`. We can take advantage of UNSAT cores, a capability of SMT solvers, to further prune the search space.

An UNSAT core (unsatisfiable core) is a subset of constraints that by themselves make a formula unsatisfiable. In Z3, the SMT solver used by Cubes, UNSAT cores can be obtained by labeling relevant constraints and then asking Z3 which of those labels are part of the UNSAT core. Suppose that for the cube represented by the constraint $line_1$ = `filter` $\wedge$ $line_2$ = `summarise`, Z3 determines that there is an UNSAT core composed by just the first part, $line_1$ = `filter`. That means that even if we tried to use a different component for the second line, it would always fail, as just the first constraint is enough to make the formula unsatisfiable. This information can then be used to prune those other cubes, as they will surely not produce a solution for the problem.

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

5 EVALUATION

In order to test and compare our tool with other state of the art SQL synthesizers we took the set of benchmarks used in Squares and expanded it. Table 1 summarizes the benchmarks used for evaluation. All results were obtained on a dual socket Intel® Xeon® Silver 4110 @ 2.10GHz, for a total of 16 cores/32 threads, with 64GB of RAM. Furthermore, using `runsolver` [22], a limit of 10 minutes (wall-clock time) and 56GB of RAM was imposed on all solvers.

The set of benchmarks used is:

- `texbook`: 37 instances extracted from exercises from the popular database textbook, *Database Management Systems* [20];
- `55-tests`: 55 instances derived from the `textbook` benchmark;
- `recent-posts`, `top-rated-posts`: 55+51 instances collected from recent and top-rated posts, respectively, on the StackOverflow [1] website;
- `spider`: 3765 instances generated from a very large and diverse benchmark of NLP instances for SQL synthesizers. For each original instance, the SQL solution query was used, along with the sample database contents, to create an input-output example that could be used in PBE synthesizers. Instances were transformed without intervention.

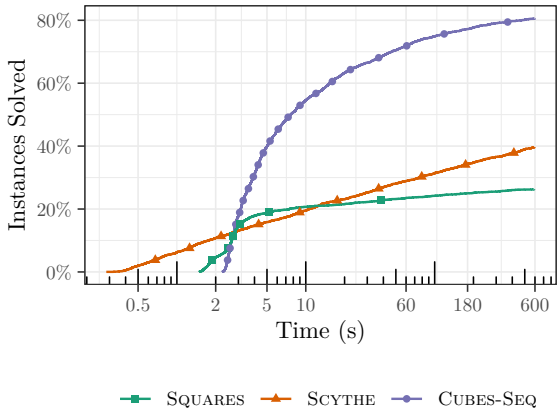

Figure 8: Percentage of instances solved by each synthesizer at each point in time. A mark is placed every 150 solved instances.

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

5.1 Sequential Results

In this section we evaluate the performance of Cubes-Seq, the sequential version of Cubes. As a comparison point, we also present the results for Squares and Scythe. Figure 8 shows the percentage of instances solved by each of these tools at each point in time. Note that the time axis is in log-scale. Overall, Squares was able to solve 26.3% of the instances in 10 minutes, while Scythe solved 39.7%. Cubes-Seq was able to solve 80.6%. Cubes-Seq solved three times more instances than Squares, and two times more instances than Scythe.

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

5.1.1 Squares and Scythe. Figure 9 compares the time taken to solve each instance when using Squares and Scythe. Each mark in the plot represents a single instance; marks above the diagonal line mean that Squares solved that instance faster, while marks below the line mean the opposite. Finally, marks positioned on the dashed lines represent a timeout for the corresponding synthesizer. Only instances solved by at least one of the synthesizers are shown. Figure 9 is similar to Figure 9, except that it compares Scythe and Cubes-Seq.

Looking at Figure 9 we can see that there is a great disparity between the set of instances solved by Squares and the set solved by Scythe (that is, most instances lie on one of the timeout lines).

Table 1: Summary of the benchmarks used for evaluation and comparison.

Benchmark	Source	# Instances
`textbook`	*Database Management Systems* [20]	37
`55-tests`	Squares [5]	55
`recent-posts`	Scythe [27]	51
`top-rated-posts`	Scythe [27]	57
`spider`	Spider [3]	3765
Total		3965

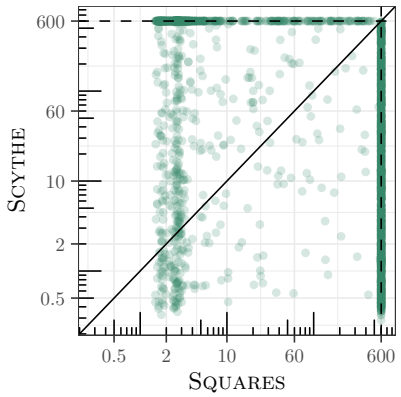

Figure 9: Scatter plot comparing the performance of Squares and Scythe.

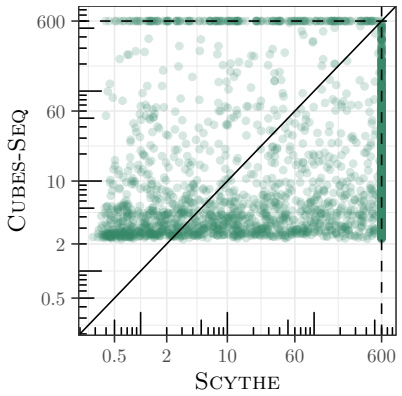

Figure 10: Scatter plot comparing the performance of Scythe and Cubes-Seq.

This can be explained by the fact that these synthesizers operate in very different ways.

Furthermore, the great majority (78.7%) of instances solved by Squares are solved in the first 10 seconds, while the same is not true for Scythe (only 49.1%). This can also be seen in Figure 8 where although Scythe is generally faster, Squares actually comes ahead in the 3 to 10 seconds time-frame.

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

5.1.2 Cubes-Seq. Looking at Table 2, we can see that the number of solved instances improved on all benchmarks when comparing with Squares, while when comparing with Scythe the number of solved instances improved on the benchmarks from Squares and Spider, and decreased on both benchmarks from Scythe. The difference in solved instances between Scythe and Cubes-Seq can be seen in Figure 10.

By default, Cubes-Seq is configured with: (i) the QF_FD SMT Theory enabled, (ii) the new DSL components introduced in subsection 3.1 enabled, and (iii) the Invalid Program Deduction enabled.

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

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

5.1.3 Portfolio. Looking at Figure 11, which shows the percentage of solved instances for each of the configurations considered,

Table 2: Overall results for 10 seconds and 10 minutes, for all configurations tested, grouped by benchmark. The best configuration for each time-limit/benchmark pair is highlighted in bold.

	Run	55-tests	recent-posts	top-rated-posts	spider	textbook	All
10 sec	Squares	32.7%	4.0%	3.5%	20.9%	30.6%	20.7%
	Scythe	38.2%	**45.1%**	61.4%	18.2%	27.8%	19.5%
	Cubes-Seq	50.9%	19.6%	47.4%	55.4%	35.1%	54.6%
	Cubes-Port4	70.9%	21.6%	56.1%	60.8%	40.5%	60.2%
	Cubes-Port8	74.5%	21.6%	56.1%	60.9%	45.9%	60.4%
	Cubes-Port16	69.1%	21.6%	56.1%	59.1%	43.2%	58.5%
	Cubes-DC4	72.7%	21.6%	61.4%	70.8%	51.4%	69.9%
	Cubes-DC8	80.0%	25.5%	63.2%	73.4%	**54.1%**	72.6%
	Cubes-DC16	**83.6%**	29.4%	**66.7%**	**75.2%**	**54.1%**	**74.4%**
10 min	Squares	70.9%	6.0%	26.3%	25.7%	41.7%	26.3%
	Scythe	70.9%	**62.7%**	71.9%	38.3%	52.8%	39.7%
	Cubes-Seq	80.0%	35.3%	63.2%	81.8%	51.4%	80.6%
	Cubes-Port4	90.9%	41.2%	70.2%	85.3%	59.5%	84.4%
	Cubes-Port8	92.7%	43.1%	73.7%	85.8%	64.9%	84.9%
	Cubes-Port16	92.7%	43.1%	73.7%	85.5%	62.2%	84.6%
	Cubes-DC4	90.9%	39.2%	**75.4%**	86.1%	**73.0%**	85.3%
	Cubes-DC8	92.7%	45.1%	**75.4%**	87.1%	70.3%	86.3%
	Cubes-DC16	**94.5%**	47.1%	**75.4%**	**88.6%**	70.3%	**87.8%**

we can see that Cubes-Port4 constitutes a modest improvement over Cubes-Seq, solving 84.4% vs 80.6% of benchmarks. However, increasing the number of portfolio processes in a way that diversifies the search is no straightforward task. With that in mind, it comes at no surprise that the improvements going from Cubes-Port4 to Cubes-Port8 and Cubes-Port16 processes are not as significant, with Cubes-Port8 solving 84.9% of the instances and Cubes-Port16 solving 84.6% of the instances. In particular, the diversity gained from the extra configurations considered in Cubes-Port16 is not enough the overcome the performance penalty of using 16 cores in the system architecture used for testing.

5.1.4 Divide and Conquer. In Figure 12, we present the results for the divide-and-conquer approach for 4 processes (Cubes-DC4), 8 processes (Cubes-DC8) and 16 processes (Cubes-DC16). In the plot we can see that going from Cubes-Seq to Cubes-DC4, Cubes-DC8 and Cubes-DC16, provides small improvements to the number of instances solved: 80.6%, 85.3%, 86.3% and 87.8%, respectively. If limited to 10 seconds, the difference becomes slightly larger with Cubes-Seq solving 54.6% of the instances, Cubes-DC4 solving 69.9%, Cubes-DC8 solving 72.6% and Cubes-DC16 solving 74.4%. In Table 2, we can see that Cubes-DC16 is best overall configuration for both 10 minutes and 10 seconds. Furthermore, it is also the best configuration for all benchmarks except `recent-posts` under 10 seconds, and all benchmarks except `recent-posts` and `textbook` for 10 minutes.

Next, we analyze the effectiveness of the work splitting technique in Cubes-DC. For each instance, we compute the equivalent number

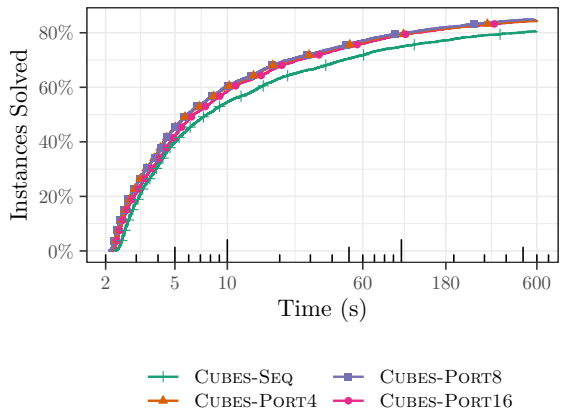

Figure 11: Performance comparison of different portfolio configurations.

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

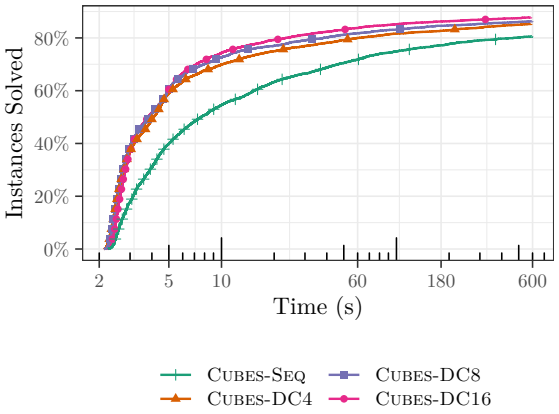

Figure 12: Performance impact of using different numbers of processes in Cubes-DC. Cubes-Seq also shown as a comparison point.

We compute this metric for Cubes-DC16, ignoring instances that took less than 20 seconds to be solved, as the inherently sequential initialization procedure distorts the metric for these instances, and conclude that 75.5% of the instances have an equivalent process $\geq$ 15, that is, it would require at least 16 processors to do the same work in the same amount of time even if it were perfectly distributed.

Cubes-DC is non-deterministic, which means that, if run several times, it does not always produce the same solutions nor solve the same benchmarks. We chose a subset of the benchmarks and executed Cubes-DC16 for each of them 10 times, in order to count the number of different outcomes. These tests were executed using 8 processes, with a 5 minute time limit. We randomly selected 1% of instances, which amounts to 38 benchmarks. Of these, Cubes-DC solved 33 of them on all 10 executions, while 2 benchmarks were solved only once, 1 benchmark was solved in 3/10 executions and 2 benchmarks were not solved in any execution. Furthermore, the median number of different solutions was 2, while the average was 2.05, the mode was 1, and the maximum was 6.

6 CONCLUSION AND FUTURE WORK

In this work, we explored the topic of Parallel Program Synthesis. We propose a new sequential program synthesizer, Cubes-Seq, which is based on Squares and constitutes an improvement to the state of the art in sequential SQL synthesis. We then use Cubes-Seq as a building block, and propose Cubes-Port and Cubes-DC, two parallel synthesizers for SQL, using techniques inspired by parallel constraint solvers.

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

6.1 Future Work

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

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

It would also be interesting to add an option for Cubes-DC to be more deterministic (at the cost of performance). Proposed changes include: (i) updating the bigram scores in batches and in a deterministic way, (ii) solve cubes in batches so that processes stay synchronized — this would require that cubes be of approximately the same difficulty in order to reduce stalls, and (iii) either find a deterministic way to assign generated cubes to the available processes or disable some optimizations with are process-local and depend on the order the received cubes.

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

REFERENCES

[1] 2020. *Stack Overflow - Where Developers Learn, Share, & Build Careers.* Retrieved 2020-09-20 from https://stackoverflow.com/

[2] 2020. *Tidyverse.* Retrieved 2020-09-09 from https://www.tidyverse.org/

[3] LILY Group at Yale University. 2020. *Spider: Yale Semantic Parsing and Text-to-SQL Challenge.* Retrieved 2020-09-03 from https://yale-lily.github.io/spider

[4] Yanju Chen, Ruben Martins, and Yu Feng. 2019. Maximal Multi-Layer Specification Synthesis. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019).* ACM, New York, NY, USA, 602–612. https://doi.org/10.1145/3338906.3338951

[5] Pedro Miguel Orvalho Marques da Silva. 2019. *SQUARES: A SQL Synthesizer Using Query Reverse Engineering.* Master's thesis. Instituto Superior Técnico, Universidade de Lisboa.

[6] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08).* Springer-Verlag, Berlin, Heidelberg, 337–340.

[7] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program Synthesis Using Conflict-Driven Learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018).* ACM, New York, NY, USA, 420–435. https://doi.org/10.1145/3192366.3192382

[8] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-Based Synthesis of Table Consolidation and Transformation Tasks from Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017).* ACM, New York, NY, USA, 422–436. https://doi.org/10.1145/3062341.3062351

[9] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017. Component-Based Synthesis for Complex APIs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017).* ACM, New York, NY, USA, 599–612. https://doi.org/10.1145/3009837.3009851

[10] Gartner, Inc. 2019. *Magic Quadrant for Enterprise Low-Code Application Platforms.* Technical Report ID G00361584.

[11] Youssef Hamadi and Lakhdar Sais (Eds.). 2018. *Handbook of Parallel Constraint Reasoning*. Springer International Publishing. https://doi.org/10.1007/978-3-319-63516-3

[12] Marijn J. H. Heule, Oliver Kullmann, and Armin Biere. 2018. Cube-and-Conquer for Satisfiability. In *Handbook of Parallel Constraint Reasoning*. Springer, 31–59.

[13] Dan Jurafsky and James H. Martin. 2009. *Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition, 2nd Edition*. Prentice Hall, Pearson Education International.

[14] Fei Li and H. V. Jagadish. 2014. NaLIR: an interactive natural language interface for querying relational databases. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 709–712. https://doi.org/10.1145/2588555.2594519

[15] Hao Li, Chee-Yong Chan, and David Maier. 2015. Query From Examples: An Iterative, Data-Driven Approach to Query Construction. *Proc. VLDB Endow.* 8, 13 (2015), 2158–2169. https://doi.org/10.14778/2831360.2831369

[16] Ruben Martins, Jia Chen, Yanju Chen, Yu Feng, and Isil Dillig. 2019. Trinity: An Extensible Synthesis Framework for Data Science. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 1914–1917. https://doi.org/10.14778/3352063.3352098

[17] Ruben Martins, Vasco Manquinho, and Inês Lynce. 2012. An Overview of Parallel SAT Solving. *Constraints* 17, 3 (July 2012), 304–347. https://doi.org/10.1007/s10601-012-9121-3

[18] Pedro Orvalho, Miguel Terra-Neves, Miguel Ventura, Ruben Martins, and Vasco Manquinho. 2019. Encodings for Enumeration-Based Program Synthesis. In *Principles and Practice of Constraint Programming (Lecture Notes in Computer Science)*, Thomas Schiex and Simon de Givry (Eds.). Springer International Publishing, Cham, 583–599. https://doi.org/10.1007/978-3-030-30048-7_34

[19] Pedro Orvalho, Miguel Terra-Neves, Miguel Ventura, Ruben Martins, and Vasco Manquinho. 2020. SQUARES: A SQL Synthesizer Using Query Reverse Engineering. *Proceedings of the VLDB Endowment* 13, 12 (Aug. 2020), 2853–2856. https://doi.org/10.14778/3415478.3415492

[20] Raghu Ramakrishnan and Johannes Gehrke. 2002. *Database Management Systems* (third ed.). McGraw-Hill, Inc., USA.

[21] Daniel Rosa Ramos. 2019. *Program Synthesis from Noisy Tabular Data*. Master's thesis. Instituto Superior Técnico, Universidade de Lisboa.

[22] Olivier Roussel. 2011. Controlling a Solver Execution with the Runsolver Tool: System Description. *Journal on Satisfiability, Boolean Modeling and Computation* 7, 4 (Nov. 2011), 139–144. https://doi.org/10.3233/SAT190083

[23] Jaydeep Sen, Chuan Lei, Abdul Quamar, Fatma Özcan, Vasilis Efthymiou, Ayushi Dalmia, Greg Stager, Ashish R. Mittal, Diptikalyan Saha, and Karthik Sankaranarayanan. 2020. ATHENA++: Natural Language Querying for Complex Nested SQL Queries. *Proc. VLDB Endow.* 13, 11 (2020), 2747–2759. http://www.vldb.org/pvldb/vol13/p2747-sen.pdf

[24] Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. 2009. Query by output. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul (Eds.). ACM, 535–548. https://doi.org/10.1145/1559845.1559902

[25] Quoc Trung Tran, Chee Yong Chan, and Srinivasan Parthasarathy. 2014. Query reverse engineering. *VLDB J.* 23, 5 (2014), 721–746. https://doi.org/10.1007/s00778-013-0349-3

[26] Peter van der Tak, Marijn Heule, and Armin Biere. 2012. Concurrent Cube-and-Conquer - (Poster Presentation). In *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings (Lecture Notes in Computer Science)*, Alessandro Cimatti and Roberto Sebastiani (Eds.), Vol. 7317. Springer, 475–476. https://doi.org/10.1007/978-3-642-31612-8_42

[27] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing Highly Expressive SQL Queries from Input-Output Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 452–466. https://doi.org/10.1145/3062341.3062365

[28] Huajie Wang, Lei Chen, Mei Li, and Mengnan Chen. 2020. GuideSQL: Utilizing Tables to Guide the Prediction of Columns for Text-to-SQL Generation. In *2020 International Joint Conference on Neural Networks, IJCNN 2020, Glasgow, United Kingdom, July 19-24, 2020*. IEEE, 1–7. https://doi.org/10.1109/IJCNN48605.2020.9206700

[29] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: Query Synthesis from Natural Language. *Proc. ACM Program. Lang.* 1, OOPSLA (Oct. 2017), 63:1–63:26. https://doi.org/10.1145/3133887

[30] Sai Zhang and Yuyin Sun. 2013. Automatically synthesizing SQL queries from input-output examples. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, Ewen Denney, Tevfik Bultan, and Andreas Zeller (Eds.). IEEE, 224–234. https://doi.org/10.1109/ASE.2013.6693082