Towards Reliable SQL Synthesis: Fuzzing-Based Evaluation and Disambiguation

Ricardo Brancas$^{1[0000-0001-7006-9829]}$, Miguel Terra-Neves$^{2[0000-0003-4089-7206]}$, Miguel Ventura$^{2[0000-0002-4233-1348]}$, Vasco Manquinho$^{1[0000-0002-4205-2189]}$, and Ruben Martins$^{3[0000-0003-1525-1382]}$

[1] INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, Portugal
[2] OutSystems, Portugal
[3] Carnegie Mellon University, USA

Abstract In recent years, more people have seen their work depend on data manipulation tasks. However, many of these users do not have the background in programming required to write complex programs, particularly SQL queries. One way of helping these users is automatically synthesizing the SQL query given a small set of examples. Several program synthesizers for SQL have been recently proposed, but they do not leverage multicore architectures.
This paper proposes Cubes, a parallel program synthesizer for the domain of SQL queries using input-output examples. Since input-output examples are an under-specification of the desired SQL query, sometimes, the synthesized query does not match the user's intent. Cubes incorporates a new disambiguation procedure based on fuzzing techniques that interacts with the user and increases the confidence that the returned query matches the user intent. We perform an extensive evaluation on around 4000 SQL queries from different domains. Experimental results show that our parallel approach can scale up to 16 processes with super-linear speedups for many hard instances, and that our disambiguation approach is critical to achieving an accuracy of around 60%, significantly larger than other SQL synthesizers.

1 Introduction

In the age of digital transformation, many people are being reassigned to tasks that require familiarity with programming or database usage. However, many users lack the technical skills to build queries in a language such as Structured Query Language (SQL). Hence, several new systems have been proposed for automatically generating SQL queries for relational databases [32,20,30,33]. The goal of *query synthesis* is to automatically generate an SQL query that corresponds to the user's intent. For instance, the user can specify their intent using natural language [30,33] or examples [28,32,20,27]. Our work targets query synthesis using examples, where an example consists of a database and an output table that results from querying the database. The problem of synthesizing SQL queries from input-output examples is known as Query Reverse Engineering [29].

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

(a) The `Grades` table.

CourseID	CourseName
10	Programming
11	Algorithms
12	Databases

(b) The `Courses` 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.

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:

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

Observe that, for a person with limited database training, it is often easier to define one or more examples than to learn how to write the desired SQL query.

Even though query synthesis tools using examples [28,32,20,27] have seen a remarkable improvement in recent years, they still suffer from scalability problems with respect to the size of the input tables and the complexity of the synthesized queries. Nowadays, multicore processors have become the predominant architecture for common laptops and servers. However, none of the previous query synthesis tools take advantage of the parallelism available in these architectures. In this work, we present CUBES, the *first parallel synthesizer* for SQL queries. CUBES is built on top of an open-source sequential query synthesizer [20], which we further improved by extending the language of queries supported by CUBES and by adding pruning techniques that can prevent incorrect programs from being enumerated. To take advantage of parallel architectures, we extend CUBES by using *divide-and-conquer*. In this approach, each process searches a smaller sub-problem until it either finds a solution or exhausts that subspace and chooses another sub-problem to solve. We present a novel approach to create sub-problems based on considering different subsets of the domain-specific language for each process.

To evaluate our tool, we collected benchmarks from previous works [32,28,27,20]. Also, we created a new dataset by extending existing query synthesis problems using natural language [35] to use examples instead. In the end, we collected around 4000 instances that will be publicly available and can be used by other researchers when evaluating query synthesis tools.

We perform an exhaustive comparison between Cubes and state-of-the-art SQL synthesizers based on examples [32,20,27]. Our evaluation shows that current SQL synthesizers can synthesize many SQL queries that satisfy the examples but do not match the user intent. We observe that *all* state-of-the-art SQL synthesizers return fewer than 50% of queries that match the user intent, i.e., even though they satisfy the example given by the user they do not match the query that the user had in mind. Cubes addresses this challenge by using parallelism to find multiple solutions and interact with the user to *disambiguate* the query that matches the user intent. To disambiguate the queries, we use fuzzing to produce new examples that result in a different output for the possible synthesized queries. We select one of these examples and ask the user if the output is correct for these new input tables. If the user responds affirmatively, we can discard all queries that do not match this new output. Otherwise, if the user responds negatively, we can discard the queries that match the new output. We repeat this process until we are confident that we found the query the user intended.

To summarize, this paper makes the following key contributions:

– a divide-and-conquer procedure for SQL synthesis (section 2).
– a new procedure that uses fuzzing to disambiguate a set of queries that satisfies the initial example (section 3).
– a new large dataset for SQL synthesis using examples with around 4000 instances (section 5).
– a new open-source SQL synthesis tool called Cubes whose parallel version with 16 processes outperforms the sequential version by solving more instances and having a median speedup of around 15× on hard instances (section 5).
– a first study that analyses the accuracy of queries returned by SQL synthesizers showing that more than 55% of the queries do not match the user intent. Our disambiguation procedure improves the accuracy of Cubes to 60% and significantly outperforms other example-based synthesizers (section 5).

2 SQL Synthesis

In this work, we propose Cubes, a divide-and-conquer query synthesizer that builds upon the open-source SQL synthesizer Squares [20]. Squares is a sequential synthesizer based on enumeration that uses operations from the R programming language as its Domain Specific Language (DSL)[4]. R is more expressive than SQL and allows a more compact representation for database queries. Since Squares is modular and open-source, it is easy to modify and extend to a parallel setting. Cubes splits the synthesis problem into disjoint sub-problems to be solved in parallel by each of the available processes. Hence, each process focuses solely on a particular area of the search space.

[4] A detailed description of the DSL is available in the extended version of this paper [3].

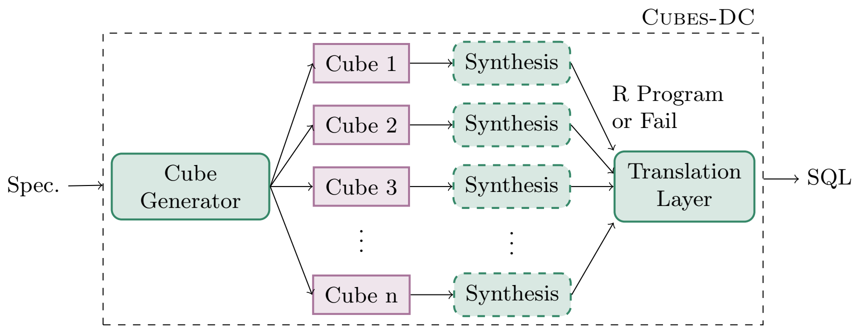

Figure 2: Cubes' architecture for divide-and-conquer.

In our context, each sub-problem is represented by a *cube*: a sequence of operations from Cubes' 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 operations, where the first is a `filter` (equivalent to a `WHERE` in SQL) and the second is a `natural_join`.

The overall architecture of Cubes is illustrated in Figure 2. The Cube Generator component is responsible for generating cubes in increasing size (i.e., first the cubes with one operation, then with two operations, and so forth), building a FIFO queue. Observe that since each cube corresponds to a distinct sequence of operations, there is no intersection in the search space of the different cubes. Then, each process receives a specific cube and checks if it is possible to fill in the missing arguments (e.g., columns, tables, filter conditions) to satisfy the input-output examples. Whenever a process finds a solution, the translation layer transforms the R program into SQL. Otherwise, if a cube cannot be extended into a complete program that satisfies the user specification, the process gets a new cube from the Cube Generator queue.

Dynamic Cube Generation. One approach for a cube generation heuristic is to define a static order of operations to be explored. Although a static heuristic can be effective on some specific domains, it is very unlikely that it generalizes to new instances. Therefore, Cubes uses a dynamic cube generator inspired by natural language techniques. Since candidate programs are constructed as a sequence of operations, a bigram prediction model can be used to decide the next operation to be chosen in a given sequence. Therefore, when choosing the next operation, the operation immediately preceding it is used to compute an expectation of which of the possible choices will lead to the desired program.

Program scoring. The initial scores of the bigram can be improved during the search by using information from programs that do not satisfy the examples. For a given program p, we compute the score of the program p as the percentage of elements of the expected output (according to the provided example) that appear in the output of p. A score of 1 indicates that all the expected values occur in the output, and as such, filtering or restructuring might lead to a correct program. On the other hand, a value of 0 means that the candidate program is likely very far from a correct solution.

For each evaluated program, the score, $score(p)$, is used to update the bigram scores. A high score for a given program, p, means that Cubes will generate new cubes similar to the one that originated the program p. On the other hand, a low score means that Cubes will try to diversify the search in the future.

DSL Splitting. Besides the splitting of the search space using cubes, Cubes also splits the DSL operations among the processes. The motivation for this additional split is that some DSL operations have more possible argument completions than others. For instance, there are many more ways to complete an `inner_join` operation than, for example, a `filter` operation. If the program to be synthesized does not require some of the complex operations, then we can solve this program more quickly with a smaller DSL. To ensure that Cubes can always find the correct program, at least one process always runs with the entire DSL while the other processes may contain only subsets of the DSL.

3 Accuracy and Disambiguation

An essential issue in program synthesis is knowing if the returned program corresponds to the user intent. To determine the accuracy of the synthesis tools, we call the query that the user wishes to obtain the *ground truth* query. Observe that SQL synthesis tools that use input-output examples return a query that satisfies the user's examples. However, these examples are an under-specification, and as such, the returned query might not satisfy the true user intent.

Cubes may find multiple queries that satisfy the examples. However, unless these queries are equivalent, only one of them matches the user's intent. To address this challenge, we create new examples with different input-output pairs for the synthesized queries and interact with the user to disambiguate the correct query. Next, we describe how to use fuzzing to create new examples and our disambiguation procedure to improve Cubes's accuracy and meet the user intent.

3.1 Fuzzing

Given a set of synthesized queries, our goal is to determine which one matches the user intent. Since some of them may be equivalent, multiple queries may be correct. One approach is to use query equivalence tools to check the equivalence of these queries and only consider a representative query of each equivalence class. Although recent work in query equivalence tools [6,38,5] has advanced the state-of-the-art, these tools remain incomplete, not supporting many complex queries present in our datasets. To overcome this limitation, we use a fuzzing-based approach to determine the approximate equivalency of different queries.

Consider a synthesis problem with an input-output example (I, O) and let Q_1 and Q_2 be two queries that satisfy this example. Fuzzing consists of taking the input I, slightly modifying it, and producing I'. Next, we apply both Q_1 and Q_2 to I' producing the outputs O'_1 and O'_2, respectively. If the outputs differ $(O'_1 \neq O'_2)$, then Q_1 and Q_2 are surely distinct. However, if the outputs are equal $(O'_1 = O'_2)$, we cannot conclude that the queries are equivalent. Hence, we perform several rounds of fuzzing, generating and testing different inputs, with each round increasing the confidence in our answer.

In order to produce fuzzed input-output examples, we use the Semantic Evaluation suite [37]. Consider a table, $T \in I$. In order to generate a fuzzed version of this table, $T' \in I'$, the suite starts by randomly selecting the number of rows of the new table. Then, to fill the cells of T', three sources are used: (1) values sampled from a uniform distribution for the given type (i.e., for integers a uniform distribution on $[-2^{63}, 2^{63} - 1]$), (2) values taken from the corresponding columns on the original table, T, and closely related values (i.e., if “Alice” is in T then both “Alice” and “Alicegg” might be considered for T'), and (3) values taken from the queries we are comparing, and closely related values. The reason why the suite takes into account values from the queries themselves is to increase code coverage (e.g., making it more likely to find off-by-one errors). Finally, all foreign keys are respected so that the semantics of the database are preserved.

3.2 Disambiguation

Cubes is able to return multiple queries that satisfy the user specification. However, if the example provided is an under-specification of the true user intent, those queries will most likely have slightly different semantics. In order to ease the burden on the user of selecting a correct query, we propose a disambiguation algorithm, shown in Algorithm 1.

Cubes starts by synthesizing all possible solutions under a given time limit. The goal of the disambiguation is then to ask the user questions in order to iteratively discard queries until we find one that satisfies the user intent. Our procedure attempts to minimize the number of questions as much as possible, by trying to discard approximately half of the queries each time we ask a question.

To do this, we start by generating a new input database I' through fuzzing. Next, we execute each of the synthesized queries on this new input I' and group them according to the output they produce. In each disambiguation step, we generate 16 new input databases, by performing fuzzing 16 times, and selecting the input-output example that is closest to splitting the set of queries in half.

Figure 3 shows a real-world disambiguation interaction. Initially, we have 7 queries found by Cubes that satisfy the original input-output example. In this case, we generate a new input I' such that 1 of the 7 queries provides the output table A', 3 queries provide as output table B', and 3 others provide an output C'. Then, we ask the user if the new input-output example (I', B') is correct. If the user answers yes, then the solution is one of the 3 queries. Otherwise, the solution should be one of the 4 remaining queries. Since the user answered yes, then 3 queries remain to disambiguate. The disambiguation procedure terminates

Algorithm 1: Disambiguation method

```
Input: $\mathcal{S}$, the set of synthesized queries, $I$, input database,
       $O$, output table, $R$, number of fuzzing rounds
Result: a query considered to be the most likely solution
Disambiguate($\mathcal{S}, I, O, R$)
1  bestSplit ← $\emptyset$;
2  for $i \leftarrow 1$ to $R$ do
3      $I'$ ← Fuzz($I$, $\mathcal{S}$);
4      split ← GroupByOutput($\mathcal{S}$, $I'$);
5      if BetterSplit(bestSplit, split) then
6          bestSplit ← split;
   end
7  if bestSplit = $\emptyset$ then
8      return First($\mathcal{S}$);
9  ($I', \mathcal{S}_A, O'_A, \mathcal{S}_B$) ← bestSplit;
10 if AskUserIfExampleIsCorrect($I'$, $O'_A$) then
11     return Disambiguate($\mathcal{S}_A$, $I$, $O$, $R$);
12 else
13     return Disambiguate($\mathcal{S}_B$, $I$, $O$, $R$);
```

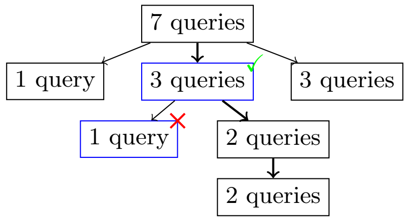

Figure 3: Example disambiguation process from a problem that generated 7 possible queries. Blue boxes represent the input-output example given to the user.

when either there is only one query remaining or the fuzzing procedure is unable to find a new example to distinguish the remaining queries. In the latter case, the remaining queries are deemed equivalent and the first one found by CUBES during the search is returned to the user. Notice that CUBES enumerates queries in increasing order of the number of operators. Hence, the first queries to be found by CUBES have the fewest operations and should be more general.

4 Methods and Data

This section describes the benchmark sets used to evaluate CUBES and compare it to other synthesizers, as well as two distinct methods to perform that comparison: simple evaluation and fuzzy-based evaluation.

Data. We use five different benchmark sets, divided into two groups. The first group, consisting of the benchmarks `recent-posts`, `top-rated-posts`, `textbook`

Algorithm 2: Query checker using fuzzing

```
   Input: $q$, the synthesized query, $Q$, the ground truth query,
          $I$, input database, $R$, number of fuzzing rounds
   Result: a Boolean representing if a distinguishing input was not found
FuzzyCheck($q, Q, I, R$)
1  if Execute($Q$, $I$) $\neq$ Execute($q$, $I$) then
2      return False;
3  for $i \leftarrow 1$ to $R$ do
4      $I' \leftarrow$ Fuzz($I$, $Q$);
5      if Execute($Q$, $I'$) $\neq$ Execute($q$, $I'$) then
6          return False;
   end
7  return True;
```

and `kaggle` refers to benchmarks that were previously used in other example-based SQL synthesis papers [32,36,20,27]. The second group consists of a single benchmark set: `spider`. We adapted the instances in `spider` from a very large and diverse dataset of queries used for SQL synthesis from Natural Language (NL) descriptions (also known as text-to-SQL) [35]. Overall, we used 176 instances from previously established benchmark sets, and created 3690 new instances.

Simple Evaluation. In this setting, we are simply interested in checking if a synthesizer can produce a query that satisfies the specification given by the user. That is, when executed, the query should produce an output table that is equal to the one specified by the user. Furthermore, we do not take into account the row order of the output table. This method has been extensively used in the past to measure the performance of SQL synthesizers [32,36,20,27]. The problem with simple evaluation is that, in the case of an ambiguous example, it does not address whether the synthesized query actually satisfies the user intent or not.

Fuzzy-based Evaluation. In this setting, we check if the synthesized queries satisfy the true intent of the user and not just the input-output example. The motive for this distinction is that the input-output example might be an under-specification of the query the user wishes to obtain. That is, several queries can satisfy the example, but they do not have the same semantics.

Algorithm 2 shows how we use fuzzing, as introduced in subsection 3.1, to determine if two queries are likely to have the same semantics. We start by sanity checking if the synthesized query, q, and the ground truth query, Q, produce the same output for the provided input database, I (lines 1-2). Then, we perform R rounds of fuzzing (line 3), where for each round, we generate a new input database, I', and check if the two queries still produce the same output table (lines 5-6). If all rounds pass successfully, we consider the queries equivalent (line 7). When comparing two tables, we perform a very lax comparison that: (1) ignores row order – tables are seen as a multiset of rows, (2) ignores column names, and (3) tries to convert the datatypes of columns – if two columns contain the same data but one as a number and the other as a string, they are considered equivalent. Note that several rounds might be needed to find an input that distinguishes the queries. The parameter R controls the maximum number of fuzzing rounds until the algorithm deems the queries equivalent.

5 Evaluation

The evaluation presented next aim to answer the following research questions:

Q1. How does the sequential version of Cubes, Cubes-Seq, compare with other state-of-the-art SQL synthesizers when using the simple evaluation metric? (subsection 5.2)
Q2. What are the speedups obtained by using the divide-and-conquer approach, Cubes-DC, when using the simple evaluation metric? (subsection 5.3)
Q3. How do Cubes and the other SQL synthesizers perform when using the fuzzy-based evaluation metric? (subsection 5.4)
Q4. What is the impact of program disambiguation in Cubes' fuzzy-based evaluation metric? (subsection 5.4)

All results were obtained on a dual socket Intel® Xeon® Silver 4210R @ 2.40GHz, with a total of 20 cores and 64GB of RAM. Furthermore, a limit of 10 minutes (wall-clock time) and 56GB of RAM was imposed on all synthesizers (sequential or parallel). All limits were strictly imposed using `runsolver` [22].

5.1 Implementation

Cubes is implemented on top of the Trinity [15] framework, using Python 3.8.3. Candidate programs are evaluated by translating the DSL operations into equivalent R instructions. In particular, the `tidyverse`[5] family of packages is used to implement table manipulations. Once a correct R program is found, the `dbplyr`[6] package (version 1.4.4) is used to translate that program to an equivalent SQL query. In the parallel synthesizer, inter-process communication is achieved using a message-passing approach through Python's `multiprocessing` pipes. All source code, instance files, and execution logs are made publicly available.[7]

We use the fuzzing framework developed by Zhong et al. [37] in our disambiguation module to perform accuracy analysis. Furthermore, queries are executed using the SQLAlchemy[8] library (version 1.3.20), and row order is ignored when comparing tables. The original implementation of the fuzzing framework is non-deterministic, so we modified it in two important ways: (1) we added proper seeding for Python's pseudo-random number generator, and (2) we replaced all

[5] `https://www.tidyverse.org/`
[6] `https://dbplyr.tidyverse.org/`
[7] `https://doi.org/10.5281/zenodo.10492998`
[8] `https://www.sqlalchemy.org/`

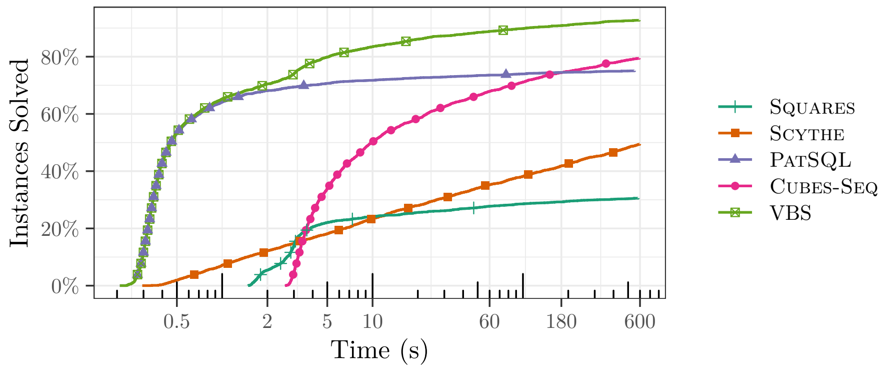

Figure 4: Percentage of instances solved by each tool at each point in time. A mark is placed every 150 solved instances.

usages of the `set` data structure with `OrderedSet` (sets backed with a list so that the iteration order is deterministic). This change was needed so that both the accuracy results presented in the paper and Cubes' disambiguation process are deterministic. The modified framework is also included in Cubes' source files.

5.2 Sequential Performance using Simple Evaluation

We start by evaluating the performance of Cubes-Seq, the sequential version of Cubes, and perform a comparison with other state-of-the-art SQL Programming by Example (PBE) tools: Squares [20], Scythe [32] and PatSQL [27]. Figure 4 shows the percentage of instances solved by each synthesizer as a function of time when using the simple evaluation method. Overall, Squares was able to solve 30.6% of the instances within the time limit of 10 minutes, while Scythe solved 49.5% and PatSQL solved 75.1%. Cubes-Seq was able to solve 79.4%.

Figure 4 also shows the Virtual Best Solver (VBS) for these four synthesizers. The VBS can be seen as the result of running the four synthesizers in parallel, or, equivalently, having an oracle that predicts which synthesizer is the best for a given instance and using it. The VBS is able to solve more instances than any of the other synthesizers (92.7% vs. the 79.4% for Cubes). This shows two things: (1) not all synthesizers solve the same instances, and (2) it is advantageous to run multiple synthesizers in parallel if the user has the resources for it. Furthermore, if we consider a VBS with only the top-performing synthesizers, PatSQL and Cubes, the percentage of solved instances is 90.5% (vs. 92.7% with the four synthesizers), meaning that using two synthesizers in parallel results in 10%+ extra instances solved compared to just using Cubes.

One interesting difference between these synthesizers is the minimum time in which they can return a solution for any of the instances, with Scythe and PatSQL at around 0.3 seconds, while Squares and Cubes only solve the first instance at 2 to 3 seconds. The most likely explanation for this difference is the

Table 1: Overall results for 10 seconds and 10 minutes grouped by benchmark. The best tool for each time-limit/benchmark pair is highlighted in **bold**.

Run	kaggle	recent-posts	top-rated-posts	spider	textbook	All	Median Speedup
			10 seconds				
Squares	21.2%	3.9%	5.3%	24.7%	28.6%	24.1%	
Scythe	0.0%	**49.0%**	**66.7%**	22.5%	28.6%	23.4%	
PatSQL	**57.6%**	41.2%	64.9%	72.5%	**62.9%**	71.7%	
Cubes-Seq	15.2%	11.8%	33.3%	51.5%	34.3%	50.3%	
Cubes-DC4	24.2%	11.8%	59.6%	70.0%	48.6%	68.5%	
Cubes-DC8	27.3%	15.7%	63.2%	73.2%	54.3%	71.8%	
Cubes-DC16	24.2%	19.6%	63.2%	**75.4%**	51.4%	**73.8%**	
			10 minutes				
Squares	21.2%	7.8%	22.8%	31.0%	40.0%	30.6%	
Scythe	3.0%	**66.7%**	**80.7%**	49.1%	54.3%	49.5%	
PatSQL	**63.6%**	45.1%	66.7%	75.8%	68.6%	75.1%	
Cubes-Seq	39.4%	25.5%	66.7%	80.9%	57.1%	79.4%	(1 ×)
Cubes-DC4	45.5%	31.4%	73.7%	88.4%	71.4%	86.9%	8.4 ×
Cubes-DC8	54.5%	39.2%	73.7%	89.6%	68.6%	88.2%	12.8 ×
Cubes-DC16	51.5%	39.2%	75.4%	**90.4%**	**77.1%**	**89.0%**	15.5 ×

startup time for the programming languages used by the synthesizers. PatSQL and Scythe both use Java, while Squares and Cubes use Python and also need to initialize the R execution environment. Figure 4 also shows that both Scythe and Cubes-Seq are able to solve more problem instances when we increase the time limit, while PatSQL and Squares seem to reach a plateau.

Table 1 shows the results for each benchmark set with virtual time limits of 10 seconds (top half) and 10 minutes (bottom half). We can see that Cubes-Seq is able to solve more instances than Squares in all benchmarks sets while solving more instances than Scythe in 3 out of 5 benchmark sets. When comparing with PatSQL, the results shown in Figure 4 are confirmed since although PatSQL solves more instances with a shorter time limit, Cubes-Seq is able to solve more instances in one benchmark set (`spider`) with a larger time limit.

5.3 Parallel Performance using Simple Evaluation

Considering the sequential version Cubes-Seq as our baseline, we now evaluate the performance of the parallel version using divide-and-conquer (Cubes-DC).

Table 1 shows the results for the divide-and-conquer strategy Cubes-DC with 4, 8, and 16 processes. Notice that divide-and-conquer tools improve upon the sequential version, from 79.4% up to 89.0% when using 16 processes. Moreover, within a limit of 10 seconds, the parallel versions are able to solve 68.5%,

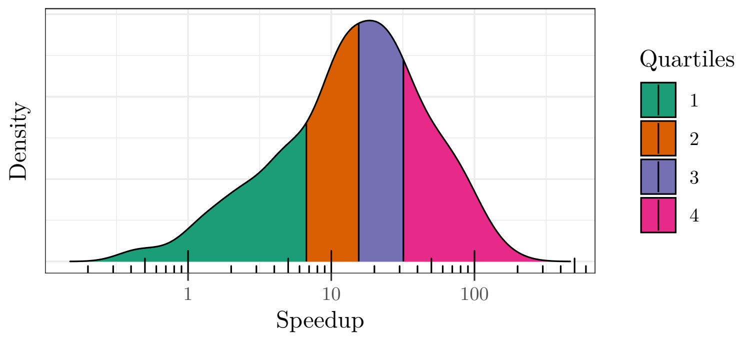

Figure 5: Instance speedup distribution for Cubes-DC16.

71.8%, and 73.8% of the instances when using, respectively, 4, 8, and 16 processes. This contrasts with the sequential version that only solves 50.3% of the instances. Hence, there is a significant speedup when using the divide-and-conquer strategy, especially for shorter time limits. Observe that even within the time limit of 10 seconds, Cubes-DC is the best-performing solver.

Formally, the speedup of method A in relation to method B is defined as the time needed to execute method B divided by the time needed to execute method A, and is a measure of how fast an implementation is compared to another. The last column of Table 1 shows the speedup obtained by each parallel version of Cubes in relation to the sequential version Cubes-Seq for instances where Cubes-Seq needed 1 minute (or more) to solve. We focus this analysis on the harder instances for the sequential tool since higher speedups in these instances have a higher impact on the end user's experience.

We can see that most configurations have a median speedup greater than the number of processes used. This is called a super-linear speedup and occurs because programs are enumerated in a different order when using our parallel versions. Figure 5 shows the full speedup distribution for Cubes-DC16 along with the distribution quartiles. We can see that more than 50% of instances have a speedup greater than 10 when using 16 processes, while more than 25% of instances have a speedup greater than 30.

5.4 Results using Fuzzing-based Evaluation

In this section we analyze the number of instances solved by Cubes when using the more thorough fuzzy-based evaluation, as well as comparing it with other program synthesis tools. Furthermore, we also evaluate the program disambiguator introduced in section 3.

Figure 6 shows the results when using the fuzzy-based evaluation method instead of the simple evaluation. For this evaluation, we used 16 fuzzing rounds ($R = 16$). The "`FuzzyCheck` Timeout" label in the plot represents instances for which the fuzzing evaluation timed out and not a timeout of the synthesizer

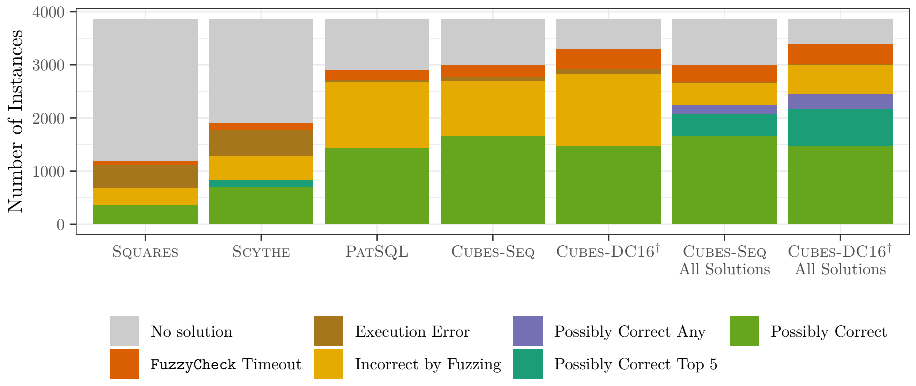

Figure 6: Results of the fuzzy-based evaluation for each synthesizer.

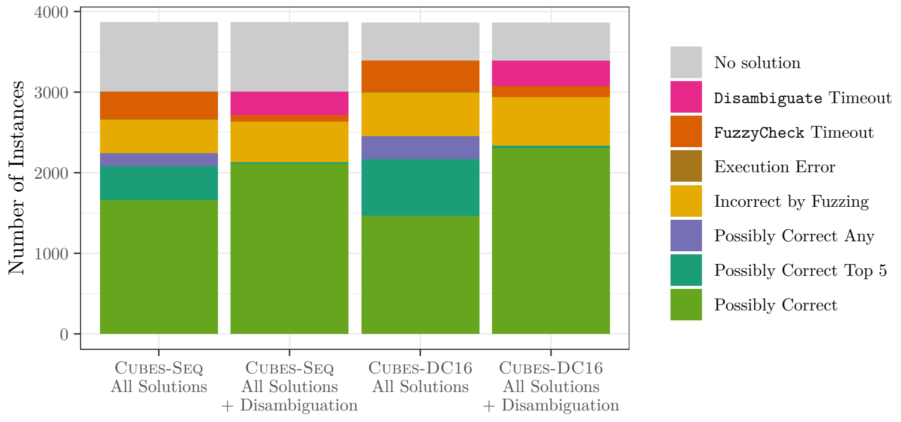

Figure 7: Fuzzy-based evaluation results before and after disambiguation.

used. We used a time limit of 60 seconds per fuzzing round ($16 \times 60s = 960s$). Furthermore, some of the synthesized queries failed to execute (labelled as "Execution Error"). This happens for two reasons: (1) some synthesized queries are incompatible with the SQLite dialect, and (2) some of the synthesized queries contain syntax problems.

We label instances for which we could not find a distinguishing input from the ground truth as "Possibly Correct", while instances for which we did find such input are labelled as "Incorrect by Fuzzing". Furthermore, for synthesizers that return multiple solutions, "Possibly Correct Top 5" means that there was a query in the top-5 returned queries for which we did not find a distinguishing input from the ground truth. Similarly, "Possibly Correct Any" means that the

Table 2: Comparison of the fuzzy-based evaluation with the simple evaluation.

	Scythe	Squares	PatSQL	Cubes-Seq All Solutions	Cubes-DC16 All Solutions
Solved (simple eval.)	49.5%	30.6%	75.1%	79.5%	90.2%
Possibly Correct[a]	21.6%	9.2%	37.1%	58.0%	63.3%
as % of Solved instances	43.6%	30.0%	49.4%	73.0%	70.2%
Incorrect by Fuzzing	11.6%	8.4%	32.3%	10.7%	14.1%
as % of Solved instances	23.4%	27.5%	43.0%	13.5%	15.6%
Inconclusive	16.2%	13.1%	5.7%	8.9%	10.2%
as % of Solved instances	32.7%	42.8%	7.6%	11.2%	11.3%

[a] Includes instances in Possibly Correct Top 5 and Possibly Correct Any.

synthesizer returned a query for which we could not distinguish it from the ground truth.

Previous tools all suffer from fairly low accuracy rates, staying under 45%, as do Cubes-Seq and Cubes-DC16 if we only consider the first solution returned. However, if we consider all solutions returned under 10 minutes, then Cubes generates a correct (using fuzzy-based evaluation) solution on around 63% of the instances, as shown in Table 2.

In order to be able to give that correct solution to the user, as opposed to giving them all the solutions generated, we developed a query disambiguator. Figure 7 shows the results of using that disambiguator on Cubes-Seq and Cubes-DC16. We can see that the disambiguator can almost always identify the correct query if such a query exists in the set of queries synthesized. Note that small differences in the exact number of queries deemed correct using the fuzzy-based evaluation may be due to different fuzzed inputs being generated.

It is also worth noting that a very small number of instances are labeled as "Possibly Correct Top 5". As explained in Section 3, Cubes returns the earliest synthesized query when we reach a set of queries that we cannot distinguish from one another. This means that, for those instances, a correct query was in the final set of queries selected by the disambiguation, but it was not the first one generated by Cubes. This happens because while the accuracy test has access to the ground truth and can thus generate better-fuzzed inputs, the disambiguator is limited to using values from the queries it is trying to disambiguate. Even so, the fact that this only occurs in a very small number of queries indicates that the approach is valid and seems to be able to both correctly disambiguate most queries and catch the cases where the disambiguation fails.

We show that if we only consider the first solution, Cubes' performance is similar to other existing tools. The main improvement comes from (1) synthesizing many possible queries for a given problem and (2) having a program disambiguator to choose the right query. This first point is directly influenced by our parallel approach to program synthesis, which allows us to synthesize more programs that satisfy the examples under the chosen time limit.

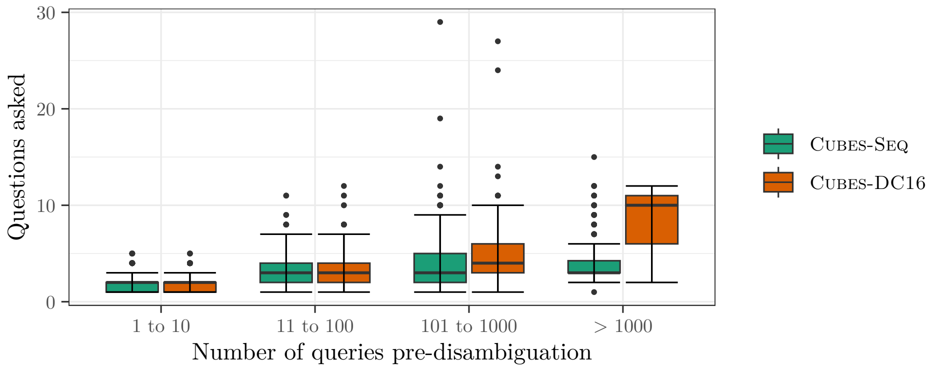

Figure 8: Number of questions that need to be asked to the user in order to perform disambiguation, as a function of the number of queries synthesized.

Finally, we analyze how many questions are asked to the user to disambiguate the queries produced by Cubes. Figure 8 shows this data as a function of the number of queries synthesized. Consider the first bar of the second group, relating to instances where Cubes-Seq generated 11 to 100 queries. The plot shows that to disambiguate those queries, we need at least 1 question, at most 11 questions, and on average 3 questions.

For Cubes-Seq the average number of questions needed to disambiguate up to 1000 queries is 2.31, while for Cubes-DC16 it is 2.69. As stated in Section 3, our goal with the disambiguation strategy is to discard half the queries with each question asked. Thus, we would expect that the number of questions needed to disambiguate a given set of queries scales logarithmically with the size of that set. Figure 8 shows that this behavior is, in fact, observed in practice.

6 Discussion

Here we discuss the main threats to validity of this work and some challenges that were raised during the experimental evaluation.

Benchmarks. Our evaluation uses a large set of benchmarks from different domains. However, they may not be representative of tasks commonly performed by users or may have a bias towards a specific synthesis tool. To mitigate this, we included benchmarks from several previous synthesis tools and also extended a large dataset from query synthesis using NLP to use examples instead. In the end, we have around 4000 instances but they are dominated by the `spider` dataset [35]. Nevertheless, since this dataset has been extensively used in other domains and was not created by us, we believe that it is more general and less prone to bias.

Parallelism. The divide-and-conquer approach already shows scalability for hard instances when using 4 and 8 processes in a multicore architecture with super-linear speedups. However, when increasing the number of processes to 16 the gains are reduced. When the number of processes increases, there is an increase of contention for memory accesses that can slow down the performance of each process. To address this issue, it would be interesting to evaluate Cubes in a distributed setting. Note that the overhead of going from multicore to distributed should be small since the inter-process communication is already done using message-passing techniques, and no shared memory is used. Exchanging information between processes is another source of improvement that would be worth exploring in future work.

Cube generation. One way to further improve the divide-and-conquer approach is to consider other cube generation strategies. For instance, we could learn from data and use machine learning techniques such as pre-trained bigram scores or using neural networks to predict the most likely cubes. We could also explore other techniques similar to the ones used in SAT solvers, such as restarting the search after n programs/cubes have been attempted.

Fuzzy-based Evaluation. Even though query synthesis tools are becoming more efficient and can find a query that satisfies the input-output example given by the user, they may not find the query that the user intended. To the best of our knowledge, this is the first study where fuzzing was used to evaluate if the query returned by the synthesizer matches the user's intent. Even though fuzzing is not a precise measurement of correctness since it may return that some queries are equivalent when they may not be, it is an upper bound on the accuracy of these tools. With the continuous improvement of SQL equivalence tools [6,38,5], it may be possible to have an exact accuracy measurement in the future. However, even with the current results, we already observe that all synthesis tools return many answers that do not match the desired behavior.

Disambiguation. Interacting with the user to perform query disambiguation is essential to increase the accuracy of SQL synthesizers based on examples. However, the questions that we asked the user may be too hard to answer, or the user may answer them incorrectly. To mitigate the difficulty of the questions, we only ask yes or no questions and present examples based on fuzzing that are often similar to the initial example provided by the user. With this approach, we hope that the user can quickly answer these questions. We currently automate the disambiguation procedure and use the ground truth to answer the questions, but a user study could be done in the future to confirm our hypothesis that the questions are easy for users to answer. In this work, we assume that the user never answers the questions incorrectly. However, considering this scenario could open new research directions and is in line with recent work on program synthesis with noisy data [11] where the examples may be incorrect.

7 Related Work

SQL Synthesis. In recent years, several tools for query synthesis have been proposed using input-output examples to specify user intent [28,36,7,32,15,20]. Solving approaches vary from using decision trees with fixed templates [28,36] to abstract representations of queries that can potentially satisfy the input-output examples [32]. Another approach is to use SMT-based representations of the search space [7,19] such that each solution to the SMT formula represents a possible candidate query to be verified. The CUBES framework proposed in this paper is also based on SMT-based representations, but it extends prior work in several dimensions: (i) extends the language in the programs to be synthesized, (ii) proposes pruning techniques that can be directly encoded into SMT, and (iii) it is the first parallel tool for query synthesis.

In this paper, we compare CUBES with three other SQL Synthesis tools that use input-output examples: SCYTHE [32], SQUARES [20] and PATSQL [27]. SCYTHE and PATSQL use sketch-based enumeration, where first a skeleton program with missing parts is generated, and then, if the skeleton satisfies a preliminary evaluation, the synthesizer tries to complete the sketch to obtain a complete program. SQUARES, on the other hand, uses Satisfiability Modulo Theories (SMT)-based enumeration where complete programs are obtained by iterating the possible solutions of an SMT formula. Both SCYTHE and SQUARES have limited DSLs and thus are not as well suited for complex tasks. Furthermore, SCYTHE's ability to solve a given instance is severely limited by the size of its input tables. Although PATSQL has a comparatively more expressive DSL, it is still not able to outperform CUBES.

Another approach for specifying user intent is using natural language [33,30]. However, these approaches often need a large training data set from the query's domain. Recently, several techniques have been proposed that try to better generalize to cross-domain data [34,24]. Although many improvements have been attained in finding the structure of the query through effective semantic table parsing, defining the details (e.g., specific filter conditions) is usually hard, particularly in more complex queries. The use of natural language for query synthesis is complementary to our approach, and a combination of both strategies could improve the accuracy of program synthesizers at the cost of more input from the user, namely examples and a natural language description of the task.

Program Disambiguation. Current synthesizers focus primarily on generating programs that satisfy the user's specifications. However, in many situations, the produced program does not satisfy the true user intent [16,26]. Previous work has shown that this shortcoming can be solved without recurring to complete specifications by introducing a program disambiguator. This component is responsible for interacting with the user and choosing between several possible solutions. Mayer et al. [16] describe two types of user interaction for program disambiguation: in the first approach, users select the correct program among a set of returned solutions, which are presented in a way that allows easy navigation. The second approach is described as *conversational clarification*, where the system iteratively asks questions to the user, further refining the original specification until just one candidate program is left [8,21,14,31,13,17]. In Cubes, we use conversational clarification to improve the confidence in produced solutions while still keeping the complexity for the user low.

Parallel Solving. Solving logic formulas in parallel has been the subject of extensive research work [10,9,1,2], both using memory-shared [25] and distributed approaches [18]. One of the techniques used to explore the search space is called divide-and-conquer [12]. In this approach, the search space is split into disjoint areas such that there is no intersection between the areas explored by each process. In this case, work-stealing techniques [23] are commonly used to avoid starvation since the search space can be unevenly split among the processes. Although we adapt techniques from parallel automated reasoning, the parallelization in the Cubes framework is not done at solving logic formulas but at a more abstract level. In our case, logic formulas continue to be solved sequentially. Moreover, starvation is avoided by producing additional work, i.e., increasing the number of operations from the DSL in the programs to be enumerated.

8 Conclusions

This work introduces Cubes, a new enumeration-based framework for query synthesis from examples. A new robust tool is proposed that is able to synthesize an extensive range of SQL queries. Additionally, Cubes also takes advantage of the current multicore processor architectures, providing the first parallel query synthesizer from examples using a divide-and-conquer approach. The splitting of the program space is done by providing different sequences of operations to each thread, as well as performing DSL splitting among threads.

An in-depth experimental evaluation is also carried out, comparing Cubes with other state-of-the-art query synthesizers in a wide variety of benchmark sets. Experimental results show the effectiveness and robustness of Cubes, being able to successfully synthesize SQL queries for a larger range of problem instances than other tools. Moreover, the parallel versions of Cubes have superlinear speedups for many hard instances and, when using 16 processes, provide a median speedup of 15× over the sequential version of the tool.

Finally, an accuracy analysis of the produced queries is also performed using fuzzing techniques. Results show that the queries produced by current synthesizers often differ from the user intent, and more than 50% of the queries returned to the user do not match the expected behavior the user had in mind. To increase the trust and reliability of SQL synthesizers, we advocate the need to use a fuzzing-based evaluation that can more precisely measure the accuracy of SQL synthesizers. Using this methodology together with the large dataset that we collected will make it easier for other researchers to evaluate their SQL synthesis tools in the future.

Since examples are imprecise specifications, increasing the trust and reliability of SQL synthesizers is essential. To improve the reliability of Cubes, we propose an interactive procedure with the user that can disambiguate among all queries found by CUBES that satisfy the original input-output example. After the disambiguation procedure, the accuracy of CUBES in providing the user intent query is significantly increased from around 40% to 60%. Other synthesizers can use similar disambiguation approaches, and it is also expected to improve their accuracy with respect to the user intent.

Data-Availability Statement

The CUBES SQL synthesizer, our dataset and the experimental results presented in this work are available in our suplemental artifact [4].

Acknowledgments

This work was partially supported under National Science Foundation (NSF) Grant No. CCF-1762363, an Amazon Research Award, and by OutSystems and by Portuguese national funds through FCT, under projects UIDB/50021/2020 (DOI: 10.54499/UIDB/50021/2020), PTDC/CCI-COM/2156/2021 (DOI:10.544-99/PTDC/CCI-COM/2156/2021) and 2022.03537.PTDC (DOI: 10.54499/202-2.03537.PTDC). Support was also provided by FCT through the Carnegie Mellon Portugal Program under Grant PRT/BD/152086/2021.

References

1. Aigner, M., Biere, A., Kirsch, C.M., Niemetz, A., Preiner, M.: Analysis of portfolio-style parallel SAT solving on current multi-core architectures. In: Berre, D.L. (ed.) POS-13. Fourth Pragmatics of SAT workshop, a workshop of the SAT 2013 conference, July 7, 2013, Helsinki, Finland. EPiC Series in Computing, vol. 29, pp. 28–40. EasyChair (2013). https://doi.org/10.29007/73N4
2. Balyo, T., Sanders, P., Sinz, C.: Hordesat: A massively parallel portfolio SAT solver. In: Heule, M., Weaver, S.A. (eds.) Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9340, pp. 156–172. Springer (2015). https://doi.org/10.1007/978-3-319-24318-4_12
3. Brancas, R., Terra-Neves, M., Ventura, M., Manquinho, V., Martins, R.: CUBES: A parallel synthesizer for SQL using examples. CoRR **abs/2203.04995** (2022). https://doi.org/10.48550/ARXIV.2203.04995
4. Brancas, R., Terra-Neves, M., Ventura, M., Manquinho, V., Martins, R.: Towards reliable SQL synthesis: Fuzzing-based evaluation and disambiguation (2024). https://doi.org/10.5281/zenodo.10492998
5. Chu, S., Murphy, B., Roesch, J., Cheung, A., Suciu, D.: Axiomatic foundations and algorithms for deciding semantic equivalences of SQL queries. Proc. VLDB Endow. **11**(11), 1482–1495 (2018). https://doi.org/10.14778/3236187.3236200
6. Chu, S., Wang, C., Weitz, K., Cheung, A.: Cosette: An automated prover for SQL. In: 8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings. www.cidrdb.org (2017), http://cidrdb.org/cidr2017/papers/p51-chu-cidr17.pdf

7. Feng, Y., Martins, R., Van Geffen, J., Dillig, I., Chaudhuri, S.: 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. pp. 422–436. PLDI 2017, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3062341.3062351
8. Ferreira, M., Terra-Neves, M., Ventura, M., Lynce, I., Martins, R.: FOREST: an interactive multi-tree synthesizer for regular expressions. In: Groote, J.F., Larsen, K.G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12651, pp. 152–169. Springer (2021). https://doi.org/10.1007/978-3-030-72016-2_9
9. Gent, I.P., Miguel, I., Nightingale, P., McCreesh, C., Prosser, P., Moore, N.C.A., Unsworth, C.: A review of literature on parallel constraint solving. Theory Pract. Log. Program. **18**(5-6), 725–758 (2018). https://doi.org/10.1017/S1471068418000340
10. Hamadi, Y., Sais, L. (eds.): Handbook of Parallel Constraint Reasoning. Springer (2018). https://doi.org/10.1007/978-3-319-63516-3
11. Handa, S., Rinard, M.C.: Inductive program synthesis over noisy data. In: Devanbu, P., Cohen, M.B., Zimmermann, T. (eds.) Proc. ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 87–98. ACM (2020). https://doi.org/10.1145/3368089.3409732
12. Heule, M.J.H., Kullmann, O., Biere, A.: Cube-and-conquer for satisfiability. In: Hamadi, Y., Sais, L. (eds.) Handbook of Parallel Constraint Reasoning, pp. 31–59. Springer (2018). https://doi.org/10.1007/978-3-319-63516-3_2
13. Ji, R., Liang, J., Xiong, Y., Zhang, L., Hu, Z.: Question selection for interactive program synthesis. In: Donaldson, A.F., Torlak, E. (eds.) Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020. pp. 1143–1158. ACM (2020). https://doi.org/10.1145/3385412.3386025
14. Li, H., Chan, C., Maier, D.: Query from examples: An iterative, data-driven approach to query construction. Proc. VLDB Endow. **8**(13), 2158–2169 (2015). https://doi.org/10.14778/2831360.2831369
15. Martins, R., Chen, J., Chen, Y., Feng, Y., Dillig, I.: Trinity: An Extensible Synthesis Framework for Data Science. Proc. VLDB Endow. **12**(12), 1914–1917 (Aug 2019). https://doi.org/10.14778/3352063.3352098
16. Mayer, M., Soares, G., Grechkin, M., Le, V., Marron, M., Polozov, O., Singh, R., Zorn, B.G., Gulwani, S.: User interaction models for disambiguation in programming by example. In: Latulipe, C., Hartmann, B., Grossman, T. (eds.) Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology, UIST 2015, Charlotte, NC, USA, November 8-11, 2015. pp. 291–301. ACM (2015). https://doi.org/10.1145/2807442.2807459
17. Narita, M., Maudet, N., Lu, Y., Igarashi, T.: Data-centric disambiguation for data transformation with programming-by-example. In: Hammond, T., Verbert, K., Parra, D., Knijnenburg, B.P., O'Donovan, J., Teale, P. (eds.) IUI '21: 26th International Conference on Intelligent User Interfaces, College Station, TX, USA, April 13-17, 2021. pp. 454–463. ACM (2021). https://doi.org/10.1145/3397481.3450680
18. Ngoko, Y., Cérin, C., Trystram, D.: Solving sat in a distributed cloud: A portfolio approach. Int. J. Appl. Math. Comput. Sci. **29**(2), 261–274 (2019). https://doi.org/10.2478/amcs-2019-0019

19. Orvalho, P., Terra-Neves, M., Ventura, M., Martins, R., Manquinho, V.: Encodings for Enumeration-Based Program Synthesis. In: Schiex, T., de Givry, S. (eds.) Principles and Practice of Constraint Programming. pp. 583–599. Lecture Notes in Computer Science, Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-30048-7_34
20. Orvalho, P., Terra-Neves, M., Ventura, M., Martins, R., Manquinho, V.: SQUARES: A SQL synthesizer using query reverse engineering. Proceedings of the VLDB Endowment **13**(12), 2853–2856 (Aug 2020). https://doi.org/10.14778/3415478.3415492
21. Ramos, D., Pereira, J., Lynce, I., Manquinho, V.M., Martins, R.: UNCHARTIT: an interactive framework for program recovery from charts. In: 35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020. pp. 175–186. IEEE (2020). https://doi.org/10.1145/3324884.3416613
22. Roussel, O.: Controlling a Solver Execution with the runsolver Tool: System description. Journal on Satisfiability, Boolean Modeling and Computation **7**(4), 139–144 (Nov 2011). https://doi.org/10.3233/SAT190083
23. Schubert, T., Lewis, M.D.T., Becker, B.: Pamira - A parallel SAT solver with knowledge sharing. In: Abadir, M.S., Wang, L. (eds.) Sixth International Workshop on Microprocessor Test and Verification (MTV 2005), Common Challenges and Solutions, 3-4 November 2005, Austin, Texas, USA. pp. 29–36. IEEE Computer Society (2005). https://doi.org/10.1109/MTV.2005.17
24. Shi, P., Ng, P., Wang, Z., Zhu, H., Li, A.H., Wang, J., dos Santos, C.N., Xiang, B.: Learning contextual representations for semantic parsing with generation-augmented pre-training. In: Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021. pp. 13806–13814. AAAI Press (2021). https://doi.org/10.1609/AAAI.V35I15.17627
25. Shinano, Y., Heinz, S., Vigerske, S., Winkler, M.: Fiberscip - A shared memory parallelization of SCIP. INFORMS J. Comput. **30**(1), 11–30 (2018). https://doi.org/10.1287/ijoc.2017.0762
26. Shriver, D., Elbaum, S.G., Stolee, K.T.: At the end of synthesis: Narrowing program candidates. In: 39th IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track, ICSE-NIER 2017, Buenos Aires, Argentina, May 20-28, 2017. pp. 19–22. IEEE Computer Society (2017). https://doi.org/10.1109/ICSE-NIER.2017.7
27. Takenouchi, K., Ishio, T., Okada, J., Sakata, Y.: PATSQL: efficient synthesis of SQL queries from example tables with quick inference of projected columns. Proc. VLDB Endow. **14**(11), 1937–1949 (2021). https://doi.org/10.14778/3476249.3476253
28. Tran, Q.T., Chan, C., Parthasarathy, S.: Query by output. In: Çetintemel, U., Zdonik, S.B., Kossmann, D., Tatbul, N. (eds.) Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009. pp. 535–548. ACM (2009). https://doi.org/10.1145/1559845.1559902
29. Tran, Q.T., Chan, C.Y., Parthasarathy, S.: Query reverse engineering. VLDB J. **23**(5), 721–746 (2014). https://doi.org/10.1007/s00778-013-0349-3
30. Wang, B., Shin, R., Liu, X., Polozov, O., Richardson, M.: RAT-SQL: relation-aware schema encoding and linking for text-to-sql parsers. In: Jurafsky, D., Chai,

J., Schluter, N., Tetreault, J.R. (eds.) Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020. pp. 7567–7578. Association for Computational Linguistics (2020). https://doi.org/10.18653/v1/2020.acl-main.677

31. Wang, C., Cheung, A., Bodík, R.: Interactive query synthesis from input-output examples. In: Salihoglu, S., Zhou, W., Chirkova, R., Yang, J., Suciu, D. (eds.) Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017. pp. 1631–1634. ACM (2017). https://doi.org/10.1145/3035918.3058738
32. Wang, C., Cheung, A., 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. pp. 452–466. PLDI 2017, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3062341.3062365
33. Yaghmazadeh, N., Wang, Y., Dillig, I., Dillig, T.: SQLizer: Query Synthesis from Natural Language. Proc. ACM Program. Lang. **1**(OOPSLA), 63:1–63:26 (Oct 2017). https://doi.org/10.1145/3133887
34. Yu, T., Wu, C., Lin, X.V., Wang, B., Tan, Y.C., Yang, X., Radev, D.R., Socher, R., Xiong, C.: Grappa: Grammar-augmented pre-training for table semantic parsing. In: 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021. OpenReview.net (2021), `https://openreview.net/forum?id=kyaIeYj4zZ`
35. Yu, T., Zhang, R., Yang, K., Yasunaga, M., Wang, D., Li, Z., Ma, J., Li, I., Yao, Q., Roman, S., Zhang, Z., Radev, D.R.: Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In: Riloff, E., Chiang, D., Hockenmaier, J., Tsujii, J. (eds.) Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018. pp. 3911–3921. Association for Computational Linguistics (2018). https://doi.org/10.18653/V1/D18-1425
36. Zhang, S., Sun, Y.: Automatically synthesizing SQL queries from input-output examples. In: Denney, E., Bultan, T., Zeller, A. (eds.) 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013. pp. 224–234. IEEE (2013). https://doi.org/10.1109/ASE.2013.6693082
37. Zhong, R., Yu, T., Klein, D.: Semantic evaluation for text-to-sql with distilled test suites. In: Webber, B., Cohn, T., He, Y., Liu, Y. (eds.) Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020. pp. 396–411. Association for Computational Linguistics (2020). https://doi.org/10.18653/v1/2020.emnlp-main.29
38. Zhou, Q., Arulraj, J., Navathe, S.B., Harris, W., Xu, D.: Automated verification of query equivalence using satisfiability modulo theories. Proc. VLDB Endow. **12**(11), 1276–1288 (2019). https://doi.org/10.14778/3342263.3342267