Synthesizing Highly Expressive SQL Queries from Input-Output Examples

MohammadAmin Raeisi

July 11, 2023

1 Introduction

This project is an effort to implement the 2017 paper of Wang et al. for synthesizing SQL queries from I/O examples given by user. We tried to implement the paper's method from scratch. Our tool takes a list of input tables I, an output table T_{out}, and a list of constants C from the user, and returns a list of queries Q which for every $q \in Q$ we have $q(I) = T_{out}$ with the additional constraint that all constants used in predicates in q must come from C.

In the second section, we will briefly describe the method used in the paper. In the third section, we will explain our implementation details. In the fourth section, we will present the results of our tool on several benchmarks. And in the last part, we will explain our main challenges and limitations.

The paper is available at https://scythe.cs.washington.edu/media/scythe-pldi.pdf and our Implementations are available at https://github.com/mhmdaminraeisi/synthesis

2 Approach

The paper's key insight is to design a language of abstract queries to break down the synthesis process. Abstract queries resemble SQL queries except that they can contain uninstantiated filter predicates in the form of holes. This language lets us decompose the original synthesis problem into the following two subproblems that can be efficiently solved:

- Synthesizing all abstract queries that can be instantiated into queries satisfying the given I/O examples and pruning away the rest.
- Searching for predicates to fill holes in these abstract queries, instantiating them into concrete ones, and determining which ones are consistent with the I/O examples.

2.1 Abstract Query Synthesis

This part searches for all abstract queries that can potentially be instantiated into queries that satisfy the given Input-Output examples via enumerative search and pruning away the rest. The enumerative search approach can be applied efficiently in this part since the size of the abstract query space is much smaller than that of the concrete query space, as all predicates are kept as holes. The algorithm of this part is complete as it recognizes all abstract queries which contain the output.

The grammar for abstract queries resembles the SQL grammar (shown in Figure 7 of the paper), except that all filter predicates (in Where, Having, On clauses) are replaced with holes "$\square$". As in SQL, abstract queries can also be composed (as in the case of Join). Evaluating abstract queries is similar to evaluating SQL queries. For instance, an abstract Select or Join query is evaluated as a SQL query with its predicate hole replaced with $True$. Formal evaluation rules are defined in Figure 9 of the table. All evaluation rules satisfy the following over-approximation property: assume $\tilde{q}$ is an abstract query; then, for any concrete query q instantiated from $\tilde{q}$, i.e., with all holes replaced with any syntactically valid predicates, the result of q is contained in the result of $\tilde{q}$. Thus, any abstract query whose result does not contain the output example will not lead to a valid query and can be pruned.

2.2 Predicate Synthesis

Once candidate abstract queries are identified, the synthesizer synthesizes predicates to instantiate each of them. This search is highly challenging since (1) the number of candidate predicates is huge and (2) evaluating and memoizing intermediate tables remain expensive. Two optimizations have been used to address these issues.

- *Locally grouping candidate predicates*: First, for each subquery of the given abstract query, the synthesizer groups its candidate predicates into equivalence classes. The idea is that if two candidate predicates behave the same on the evaluation result of a given abstract subquery, their behaviors on the whole abstract query remain identical, no matter how other holes in the query are instantiated. Hence, it must retain only one predicate to represent the equivalence class.

- *Encoding tables using bit-vectors*: The second optimization encodes intermediate tables using bit-vectors to improve search efficiency. The insight stems from the over-approximation property of evaluating abstract queries. Because of that, when searching for the instantiation of an abstract query $\tilde{q}$, it can use its abstract evaluation result $\tilde{T}$ together with a bit-vector β to represent the evaluation result of every instantiation of $\tilde{q}$: the size of β is same as the number of rows in $\tilde{T}$, and the *i-th* bit in β represents whether the row i in $\tilde{T}$ appears in T. Encoding tables into bit-vectors has two benefits. First, the intermediate results of the predicate search process can be fully represented using bit-vectors to reduce the memorization overhead since the tables evaluated from abstract queries are shared among many, and bit-vectors are cheap to memorize. Second, it can optimize operators on queries into bit-vectors operators benefiting from this representation; since many bit-vector operators require no materialization of tables, the computation overhead is also significantly reduced.

3 Implementation Descriptions

In the following, there are explanations related to the packages and classes that we have implemented.

- All the models that the program uses, such as tables, rows, etc., are implemented in package model.
- There is a benchmarks folder where all the benchmarks and their results are placed inside this folder to test the program. The implementations related to reading the benchmark files and converting the input data taken by the user to the data defined in the program model are in the class FileScanner.
- Since our problem is the synthesis of SQL programs, it was necessary to use a database to execute and test the queries that were generated during the execution of the program. For this purpose, we have used the MySQL database and connected and used it with Java jdbc. The implementations related to running the generated queries on the database, converting its results to the tables defined in the program model, and storing them in the database are included in the class SqlManager. In fact, SqlManager class is the interface between the program and the database.
- There is a grammar package in which all the grammar defined in the paper (Figures 7 and 8) are implemented in this package. The query subpackage contains an abstract class *Query* that has two abstract methods. The first one is the *evaluateAbstract* method which includes the implementations related to rules of evaluation abstract queries that are included in Figure 9 inside the paper. The other is the $bitVectorDFS$ method, which consists of the implementations of the bit-vector search algorithms explained in Algorithm 7 in the paper. Any class (like *Select*, *Join*, *Aggr*, etc.) that inherits from the *Query* must implement these two methods according to the descriptions and details defined in the article.
- Operators related to bit vectors along with encode and decode functions which are defined in Figure 10 of the paper are implemented in the $BitVector$ class inside package model.
- Inside the synthesizer package, three classes have been implemented. The first one is the class *AbstractQuerySynthesizer*. The *synthesisAbstractQueries* method of this class takes a *depth* value as input and returns all candidate abstract queries up to that depth as output. For each of the types of queries, there is a method to generate the abstract queries of that type, which according to the limitations of each type, they have implemented rules to generate the abstract queries. For example, the Select abstract query can only have a NamedTable sub-query. Or the Aggr abstract query does not have Join or Aggr sub-queries, etc. These restrictions have been placed to reduce the search space of abstract queries and make searching easier on queries. The second class is the *PredicateSynthesizer*, whose *synthesisPredicate* method takes a list of candidate abstract queries and fills the holes of each by calling the $bitVectorDFS$ method. And finally, it returns related queries of each of the found bit-vectors whose decoding is the same as the output table as output. The last class is class Synthesizer, where all the steps of the algorithm are executed in order in this part. First, it generates candidate abstract queries with the help of the class AbstractQuerySynthesizer, and then it obtains the solutions with the help of the class PredicateSynthesizer.

4 Evaluation

We run our tool on 16 benchmarks, 10 of them were selected from Stackoverflow posts, and six were created manually for testing new features. Benchmarks and its results are completely in the GitHub repository benchmarks folder. We show our results briefly in the following table. Visited abstract queries and candidate abstract queries columns indicate how many abstract queries have been seen in the enumerative search algorithm and how many of them contain output table as candidate abstract queries. The solutions column indicates how many solutions were found by filling the holes of candidate abstract queries. The time column shows the algorithm's running time, and the depth column indicates how deep the solutions were found.

benchmark	visited abs queries	candidate abs queries	solutions	running time(s)	depth
1	67	2	1	4.245	3
2	2	1	1	6.137	2
3	221	9	3	6.343	3
4	67	2	1	7.608	3
5	221	11	2	6.494	3
6	81	4	1	1.829	3
7	7	1	1	0.218	1
8	1	1	1	0.062	1
9	4	1	1	1.128	2
10	4	1	1	0.529	2
11	19	1	1	64.816	2
12	18	1	1	1.115	2
13	318	4	1	8.94	3
14	219	14	4	10.945	3
15	221	11	3	11.8	3
16	221	14	3	23.377	3

In the following, we mention some points about the above table:

- There are some benchmarks like benchmark 2, which despite having fewer visited abstract queries, their algorithm running time is more than some other benchmarks like benchmark 1. The reason is that benchmark 2 uses the SUM aggregator, which takes a lot of time to evaluate the abstract table. More details about the high running time of the SUM aggregator come in Section 5.
- All the experiments are conducted on a machine with Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz 2.90GHz CPU and 12 GB of memory, running the windows 10 operating system.

5 Limitations and Challenges

- The algorithm of synthesizing abstract queries has one bottleneck: the number of tables enumerated at each stage is exponential to the maximum column size of input tables since the algorithm enumerates all possible grouping by columns for each aggregation query and also enumerates all possible column projections to ensure completeness. In order for the program execution time not to increase exponentially for problems that do not have these features, the user must specify in the input config whether he wants projection or grouping of all columns or not. Also, if the user wants the synthesizer to use these two features, the tables defined in the input should have a small number of columns so that the program execution time does not increase too much.
- The authors of the paper have run their tool on 193 benchmarks and have included the results in the article, but to implement the idea of the paper, we first tried to implement all the steps of the algorithm on only the example that was mentioned in the paper and after that we tried to add more features to our project by adding and testing on more benchmarks. But by adding each new benchmark and feature, we would spend time refactoring the code so that the previous benchmarks don't get into trouble, and that's why we only had time to test a few benchmarks. But these benchmarks that have been tested have produced perfect results.
- Due to time constraints, we could not add the two Left-Join and Rename queries to the project. Despite the simplicity of Rename query and the similarity between Left-Join and Join, we prioritized testing on more benchmarks with the same queries as before. And since it was very

time-consuming, we did not have time to add these two queries to the project and test them. For the same reason, we also could not add Avg, Concat, and Count Distinct aggregators.

- To reduce the search space of the programs, we have included only operator $<$ from the binary operators $<$ and $\leq$ ($>$ and $\geq$ are similar). Because for example, predicates $a \leq 49$ and $a < 50$ are equivalent, and the user can enter 50 instead of 49 and get the same result.
- Among the aggregators, SUM and COUNT have a very long running time. As it is clear in the explanation related to the evaluation of the Aggr abstract query in Figure 9, the evaluation time of these queries is exponential to the number of rows of evaluation of its abstract subquery $\tilde{q}$. The evaluation of any abstract query must represent all possible states of hole-filling. But since we don't know how the internal holes will be filled in the future, we consider all possible subsets of rows and apply the corresponding aggregator to each of them, and finally collect all the results and We delete duplicate rows. For this reason, the execution time of evaluation of Aggr abstract queries with SUM and COUNT aggregators is very high. But aggregators like MIN and MAX don't need to try all subsets of rows. Because regardless of how the internal predicates are filled in any case, their values come from the values of the original table, so in these two aggregators, we only need to delete the duplicate rows of the main table.
- Since the paper did not mention the order of the rows, we did not consider the order of the table rows, and two tables with equal rows and different permutations are considered equivalent.