SQUARES : A SQL Synthesizer Using Query Reverse Engineering

Pedro Orvalho
INESC-ID, Lisboa, Portugal
pmorvalho@sat.inesc-id.pt

Miguel Terra-Neves
OutSystems, Lisboa, Portugal
miguel.neves@outsystems.com

Miguel Ventura
OutSystems, Lisboa, Portugal
miguel.ventura@outsystems.com

Ruben Martins
CMU, Pittsburgh, USA
rubenm@cs.cmu.edu

Vasco Manquinho
INESC-ID, IST - Universidade de Lisboa, Portugal
vmm@sat.inesc-id.pt

ABSTRACT

Nowadays, many data analysts are domain experts, but they lack programming skills. As a result, many of them can provide examples of data transformations but are unable to produce the desired query. Hence, there is an increasing need for systems capable of solving the problem of Query Reverse Engineering (QRE). Given a database and output table, these systems have to find the query that generated this table. We present SQUARES, a program synthesis tool based on input-output examples that can help data analysts to extract and transform data by synthesizing SQL queries, and table manipulation programs using the R language.

PVLDB Reference Format:
Pedro Orvalho, Miguel Terra-Neves, Miguel Ventura, Ruben Martins and Vasco Manquinho. SQUARES : A SQL Synthesizer Using Query Reverse Engineering. *PVLDB*, 13(12): xxxx-yyyy, 2020.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

1. INTRODUCTION

Due to the Big Data revolution, many people with expertise in their respective domains have become data analysts. As a result, there is a growing population of non-expert database end-users that have limited programming knowledge [18]. Although most users know how to make a description of what they want or what the task should do, sometimes they do not know how to express it in a query language, such as SQL. On that account, many systems were proposed in order to help end-users query a relational database [13, 18]. This area of Program Synthesis became known as Query Synthesis, where the goal is to automatically generate the query desired by the user [13].

The user's intent can be specified using different approaches such as input-output examples [13, 17] or a natural language description [16]. In both approaches, the user provides an input database. However, in the first approach, the user also provides the desired query's output table, while in the second the user provides a query description in natural language. Query Synthesis from input-output examples is also known as Query Reverse Engineering (QRE) where the user provides a database $\mathcal{D}$ with schema graph G and an output table $\mathcal{Q}(\mathcal{D})$, which is the result of running some unknown query $\mathcal{Q}$ on $\mathcal{D}$. Given the pair $(G, \mathcal{Q}(\mathcal{D}))$, the goal of QRE is to produce the query $\mathcal{Q}$ whose result is $\mathcal{Q}(\mathcal{D})$.

Nowadays, not knowing which SQL query generated some table can easily happen, due to changes in the software or just by changing the data analyst. Furthermore, consider that a data analyst wrote a query in SQL, but she is unaware whether there exists a similar query that produces the same result with lower complexity regarding the number of table joins and conditions. If a QRE system searches for a query in a growing number of table joins and conditions, it can return the query that is consistent with the output table and has the lowest complexity possible [17]. Finally, we note that QRE is not specific to SQL and other languages can be used such as SPARQL [1]. Moreover, other program synthesis tools have been developed for table manipulation using the R language [3].

This paper presents SQUARES, a new program synthesis tool based on input-output examples that is able to synthesize SQL queries, as well as table manipulation operations using the R language. We start by providing a general description of the tool in Section 2. Next, Section 3 demonstrates how to use our framework. Section 4 briefly reviews related systems and the paper concludes in Section 5.

2. SQUARES

SQUARES, **A** **S**QL **S**ynthesizer **U**sing **Q**uery **R**everse **E**ngineering, is a new enumeration-based programming by example system developed on top of a state-of-the-art synthesis framework, Trinity [6]. SQUARES is implemented in Python and uses the Z3 SMT solver [2] to check the satisfiability of constraints generated by the program enumerator.

Figure 1 illustrates the architecture of SQUARES. As other synthesis tools, SQUARES receives as input a set of input-output examples and its architecture can be divided into two main components: enumerator and decider. Given a Domain-Specific Language (DSL, see Figure 2), the enumerator is responsible for enumerating all possible programs up

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

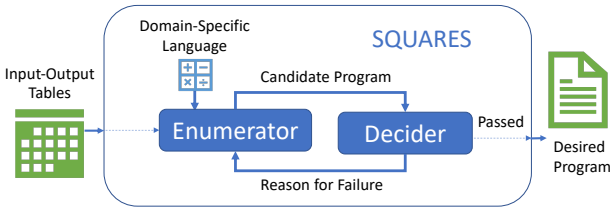

Figure 1: Architecture of SQUARES

$$
\begin{array}{lcl}
table & \rightarrow & input \mid inner\_join(table,\ table) \mid \\
 & & inner\_join3(table,\ table,\ table) \mid \\
 & & inner\_join4(table,\ table,\ table,\ table) \mid \\
 & & filter(table,\ filterCondition) \mid \\
 & & filters(table,\ filterCondition, \\
 & & filterCondition,\ op) \mid \\
 & & summariseGrouped(table, \\
 & & summariseCondition,\ Cols) \mid \\
 & & anti\_join(table,\ table) \mid left\_join(table,\ table) \\
 & & bind\_rows(table,\ table) \mid intersect(table,\ table) \\
tableSelect & \rightarrow & select(table,\ selectCols,\ distinct) \\
op & \rightarrow & Or \mid And \\
distinct & \rightarrow & true \mid false
\end{array}
$$

Figure 2: Domain-Specific Language of SQUARES.

to a given number of operations. Each production rule in our DSL has a direct translation to the R language [1] [7]. SQUARES' DSL uses four distinct types: *table, tableSelect, op* and *distinct*. The input's type is *table* and the output's type is *tableSelect*. Regarding the variety of production rules, we use the basic operations offered by R's *dplyr* [2] library (e.g. `inner_join, filter, summarise, left_join`). The terminal symbols belonging to *filterCondition, summariseCondition, Cols, selectCols* (Figure 2), are computed on the fly because they differ with the input-output examples, as well as, the number of input tables. Currently, our DSL already supports a considerable portion of SQL [7], however, the DSL could be further extended in order to synthesize more complex queries i.e., queries with other SQL operators.

For each program $\mathcal{P}$, the decider checks if $\mathcal{P}$ satisfies the input-output examples provided by the user. Let $\mathcal{P}_R$ denote the translation of program $\mathcal{P}$ to R. The enumerator executes $\mathcal{P}_R$ on the input examples and the decider compares if the output matches the expected one. If the output of $\mathcal{P}_R$ does not match, the decider produces a reason for failure. As in Neo [3], the enumerator prunes all equivalent unfeasible programs from the search space. Afterwards, the next candidate program is enumerated. Otherwise, if the output of $\mathcal{P}_R$ matches the expected one, the synthesizer translates $\mathcal{P}_R$ to SQL and returns both the SQL query and the R program. Note that each production rule in R (e.g. `anti_join`) can be easily translated into several operators in SQL[3] (e.g. `anti_join` → `SELECT ... FROM...WHERE NOT EXISTS ...`). Therefore, we can generate programs with more SQL productions rules if we use a DSL for R and then translate the desired program to SQL, instead of generating a program directly from a SQL grammar.

[1] `https://www.r-project.org`
[2] `https://cran.r-project.org/web/packages/dplyr/vignettes/dplyr.html`
[3] `https://dbplyr.tidyverse.org/articles/sql-translation.html`

Input	
Id	**Grade**
101	8
102	11
103	15
104	7
105	18
106	10

Output
Id
102
103
105
106

Figure 3: Input table: Students' ids and grades. Output table: id of students whose grade is positive ($grade >= 10$).

Trinity [6] by default uses tree-based enumeration to search for programs. The number of nodes used by Trinity's encoding grows exponentially with the number of production rules in a program. Therefore, SQUARES uses a new line-based encoding [10], that scales better than the tree-based approach. The interested reader is referred to the literature [7, 10] for more details about SQUARES' encoding.

SQUARES starts by searching for programs with one production rule and iteratively increases this bound until a program that satisfies all input-output examples is found. Therefore, SQUARES returns the first and also the smallest query in terms of SQL production rules that is consistent with the input-output examples provided by the user.

As most of the Programming-By-Example state-of-the-art synthesizers [3, 4, 6], SQUARES takes as input a set of examples and any constants or aggregate functions (e.g., `sum`, `mean`) that the query may need. Overall, the user can provide four types of information: input-output examples, constants, aggregates and attributes. This knowledge is useful to express the user intent and to guide the search through the program space.

- *Input-Output Examples.* The user specifies the input tables and the output table. The input and output examples are mandatory.
- *Constants (Optional).* The user can also provide constants to be used in the program. Note that only constants provided by the user will be considered when enumerating programs in SQUARES.
- *Aggregates (Optional).* The user can provide the name of aggregate functions that might appear in the desired program. Aggregates function such as `max, min, count (n), mean, like`.
- *Attributes (Optional).* When the user specifies a constant or an aggregate, she needs to specify which attribute is supposed to be compared against the constant or used in the aggregate function. This way, the synthesizer can generate valid conditions that use the constants and aggregates provided by the user.

The user can provide this information to our framework and run SQUARES in three different environments:

- with a Python script (locally),
- using Jupyter notebooks (locally),
- through Google Colab (online).

To an experienced programmer, the best way to use SQUARES is to download it from GitHub [4] and run it using Python. In this environment, the user can easily provide

[4] `https://github.com/squares-sql/squares`

Class	
C_name	**F_key**
Potions	f1
Charms	f2
Dark Arts	f1
Herbology	f3
Transfiguration	f4

Faculty	
F_key	**F_name**
f1	Snape
f2	Flitwick
f3	Sprout
f4	McGonagall

Student		
S_key	**S_name**	**Level**
S1	Harry	JR
S2	George	SR
S3	Ron	JR
S4	Fred	SR
S5	Hermione	JR

Output
n
2

Enrolled	**S_key**	S1	S2	S3	S3	S4	S4	S5
	C_name	Potions	Potions	Charms	Transfiguration	Charms	Herbology	Dark Arts

Figure 4: Four input tables: Class, Enrolled, Faculty and Student. One output table.

large input-output examples (tables) in the form of text files (.csv). In contrast, when using Jupyter notebooks or Google Colab, the user needs to write these tables as strings. Hence, these environments are more suited for small examples.

The Jupyter notebook and Google Colab environments are similar to use. The main difference is that the user has to install every dependency locally to run the Jupyter notebook. On the other hand, in Google Colab, all dependencies are automatically installed whenever a new session is started. In the next section, we focus on using SQUARES with Jupyter notebooks, as we did in the available video demonstration [8]. Nevertheless, the interface in Google Colab is analogous.

3. DEMONSTRATIONS

This section provides two useful demonstrations of SQUARES. Using a Jupyter notebook, a non-programmer user can synthesize the desired SQL query and/or an R program by following the instructions provided in the console.

Demonstration 1. Consider the scenario where a professor graded an exam with scores ranging from 0 to 20. [5] She does this every year and wants to know which students have grades greater than or equal to 10. To generate the desired query, she provides information from the previous year to SQUARES. In particular, she provides an input-output example with a table containing all the student's identification and respective grades (input table, Figure 3) and an output table containing the identifiers of students with positive grades (output table in Figure 3). To reinforce the user intent and to aid the program synthesizer, she can think of one constant (10) and one attribute (`grade`) as possibilities to be included in the query. Hence, giving this information to SQUARES the query generated using our DSL would be:

```
select(filter(input, grade >= 10),id,true)
```

Translating from our DSL to R, SQUARES generates the following R query:

```
df1 <- input %>% filter(grade >= 10)
df2 <- df1 %>% select(id) %>% distinct()
```

The following SQL program is translated from the previous R query:

```
SELECT DISTINCT `id`
FROM `input`
WHERE (`grade` >= 10.0)
```

[5] http://sat.inesc-id.pt/~pmorvalho/squares-vldb-1

Demonstration 2. We chose a more complex example to explain our framework's capabilities [6]. The following example is inspired by exercise 5.1.1 from a classic textbook on databases [11]. Consider the following four tables: Student, Class, Faculty and Enrolled. With schema: *Student(S_key: integer, S_name: string, level: string),Class(C_name: string, F_key: integer),Faculty(F_key: integer, F_name: string), Enrolled(S_key: integer, C_name: string)*. These tables are presented in Figure 4. Now, imagine a user wants to count (`COUNT()`) how many Juniors (`Level = "JR"`) are enrolled in a class taught by Professor Snape (`F_name = "Snape"`). First, the user should write these tables, input and output, as strings and provide them to SQUARES. Afterwards, the user should consider which hints (constants, aggregates, and attributes (see Section 2)) to be provided to SQUARES.

Constants. The user desires to know how many Juniors (`"JR"`) are enrolled in a class taught by Professor Snape (`"Snape"`). Therefore, she can think of two constants (`"JR"` and `"Snape"`) that may be used and introduce them in the Jupyter notebook as `const="JR, Snape"`.

Aggregates. Since the user wants to know how many students are enrolled in Professor `Snape`'s class, she should provide the `COUNT` operator to SQUARES. In this case, the user should write `"n"` to count the number of students in the field that corresponds to SQUARES' aggregates. The user only needs to indicate which SQL operators she thinks might be included in the desired program (e.g. MAX, MIN, AVG, SUM, LIKE, CONCAT or COUNT).

Attributes. In terms of attributes, the user should provide an attribute for each constant and aggregate provided. Hence, in this case, the user should provide three attributes: Faculty name `F_name` (to be compared with constant `"Snape"`), `Level` (to be compared with `"JR"`) and student identifier `S_key` (to be counted).

After introducing all this information the user should run the Jupyter notebook. After only a few seconds, SQUARES returns the desired query both in SQL and R. The following queries (R and SQL) are the ones produced by SQUARES.

```
df1 <- inner_join(inner_join(
inner_join(Faculty, Class),
Enrolled), Student)
df2 <- df1 %>% group_by(F_name, level)
%>% summarise(n = n())
df3 <- df2 %>% filter(F_name == "Snape"
& Level == "JR")
df4 <- df3 %>% select(n)
```

[6] http://sat.inesc-id.pt/~pmorvalho/squares-vldb-2

```
SELECT `n`
FROM
  (SELECT `F_name`, `Level`,
          COUNT() AS `n`
   FROM
     (SELECT *
      FROM
        (SELECT *
         FROM
           (SELECT *
            FROM `Faculty`
            INNER JOIN `Class`)
         INNER JOIN `Enrolled`)
      INNER JOIN `Student`)
   GROUP BY `F_name`, `Level`)
WHERE (`F_name` = 'Snape'
       AND `level` = 'JR')
```

4. RELATED SYSTEMS

Query Reverse Engineering has numerous applications like database usability, data analysis, and data security [13]. In the last decade, several systems were developed trying to solve QRE. TALOS [13], STAR [17] and FastQRE [5] are frameworks specialized in generating Select-Project-Join queries. The focus of SQLSynthesizer [18] and REGAL [12] is to generate more complex queries with aggregates. Morpheus [4] was originally designed for table transformations in R and has limited support for SQL queries. Neo [3] and Trinity [6] also perform an enumeration-based search of possible queries, they use a tree-based encoding to enumerate programs, which grows exponentially with the number of production rules used in a query [10].

Scythe [15, 14] is a state-of-the-art SQL synthesizer and has good performance for queries using small tables but has issues with memory usage when using larger tables. We compared SQUARES against Scythe, on several QRE instances from a classic database textbook [11] and observed that SQUARES has similar performance on small examples (both solved 19 out of 28 instances) [7]. However, when considering QRE instances from real-world queries from a low-code programming framework, tables with millions of entries, SQUARES solved 20 out of 20 instances whereas Scythe was only able to solve 2 instances [7]. The main reason for SQUARES' performance is that our approach, unlike Scythe, is independent of the size of the input-output tables.

5. CONCLUSION

In the last few years, companies have collected massive amounts of data. As a result, domain experts with little programming skills have become responsible for performing data analysis. Hence, a good and scalable Query Reverse Engineering (QRE) system is more important than ever to improve their productivity.

In this paper, we presented SQUARES, a novel open-source enumeration-based synthesizer for SQL and R. SQUARES can be used in three different ways: using Python (locally), Jupyter notebooks (locally) or Google Colab (online). To see SQUARES in action, we refer the reader to our demonstration video at YouTube [9] and to SQUARES' website [8], where the user can find links to examples of Query Reverse Engineering problems on Google Colab.

6. ACKNOWLEDGMENTS

This work was partially supported by NSF award number CCF-1762363 and national portuguese funds through FCT, under projects UIDB/50021/2020, DSAIPA/AI/0044/2018 and project ANI 045917 financed by FEDER and FCT.

7. REFERENCES

[1] E. Abramovitz, D. Deutch, and A. Gilad. Interactive inference of SPARQL queries using provenance. In *ICDE*, pages 581–592, 2018.

[2] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[3] Y. Feng, R. Martins, O. Bastani, and I. Dillig. Program synthesis using conflict-driven learning. In *PLDI*, pages 420–435, 2018.

[4] Y. Feng, R. Martins, J. V. Geffen, I. Dillig, and S. Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. In *PLDI*, pages 422–436, 2017.

[5] D. V. Kalashnikov, L. V. S. Lakshmanan, and D. Srivastava. Fastqre: Fast query reverse engineering. In *SIGMOD*, pages 337–350, 2018.

[6] R. Martins, J. Chen, Y. Chen, Y. Feng, and I. Dillig. Trinity: An extensible synthesis framework for data science. *PVLDB*, 12(12):1914–1917, 2019.

[7] P. Orvalho. SQUARES : A SQL Synthesizer Using Query Reverse Engineering. Master's thesis, Instituto Superior Técnico, Lisboa, Portugal, 2019.

[8] P. Orvalho, M. Terra-Neves, M. Ventura, R. Martins, and V. Manquinho. SQUARES video demonstration. `https://www.youtube.com/watch?v=ZJQcoWw-l14`. Accessed: 2020-03-22.

[9] P. Orvalho, M. Terra-Neves, M. Ventura, R. Martins, and V. Manquinho. SQUARES webpage. `https://squares-sql.github.io/`. Accessed: 2020-03-22.

[10] P. Orvalho, M. Terra-Neves, M. Ventura, R. Martins, and V. M. Manquinho. Encodings for enumeration-based program synthesis. In *CP*, pages 583–599, 2019.

[11] R. Ramakrishnan and J. Gehrke. *Database management systems (3. ed.)*. McGraw-Hill, 2003.

[12] W. C. Tan, M. Zhang, H. Elmeleegy, and D. Srivastava. Reverse engineering aggregation queries. *PVLDB*, 10(11):1394–1405, 2017.

[13] Q. T. Tran, C. Y. Chan, and S. Parthasarathy. Query reverse engineering. *PVLDB*, 23(5):721–746, 2014.

[14] C. Wang, A. Cheung, and R. Bodík. Interactive query synthesis from input-output examples. In *SIGMOD*, pages 1631–1634, 2017.

[15] C. Wang, A. Cheung, and R. Bodík. Synthesizing highly expressive SQL queries from input-output examples. In *PLDI*, pages 452–466, 2017.

[16] N. Yaghmazadeh, Y. Wang, I. Dillig, and T. Dillig. Sqlizer: query synthesis from natural language. *PACMPL*, 1(OOPSLA):63:1–63:26, 2017.

[17] M. Zhang, H. Elmeleegy, C. M. Procopiuc, and D. Srivastava. Reverse engineering complex join queries. In *SIGMOD*, pages 809–820, 2013.

[18] S. Zhang and Y. Sun. Automatically synthesizing SQL queries from input-output examples. In *ASE*, 2013.