Automated Translation of Functional Big Data Queries to SQL

GUOQIANG ZHANG, North Carolina State Univeristy, United States
BENJAMIN MARIANO, The University of Texas at Austin, United States
XIPENG SHEN, North Carolina State Univeristy, United States
IŞIL DILLIG, The University of Texas at Austin, United States

Big data analytics frameworks like Apache Spark and Flink enable users to implement queries over large, distributed databases using functional APIs. In recent years, these APIs have grown in popularity because their functional interfaces abstract away much of the minutiae of distributed programming required by traditional query languages like SQL. However, the convenience of these APIs comes at a cost because functional queries are often less efficient than their SQL counterparts. Motivated by this observation, we present a new technique for automatically transpiling functional queries to SQL. While our approach is based on the standard paradigm of counterexample-guided inductive synthesis, it uses a novel *column-wise decomposition* technique to split the synthesis task into smaller subquery synthesis problems. We have implemented this approach as a new tool called RDD2SQL for translating Spark RDD queries to SQL and empirically evaluate the effectiveness of RDD2SQL on a set of real-world RDD queries. Our results show that (1) most RDD queries can be translated to SQL, (2) our tool is very effective at automating this translation, and (3) performing this translation offers significant performance benefits.

CCS Concepts: • **Software and its engineering → Automatic programming**; • **Information systems → Query optimization**.

Additional Key Words and Phrases: program synthesis, source-to-source compiler, query optimization

ACM Reference Format:
Guoqiang Zhang, Benjamin Mariano, Xipeng Shen, and Işıl Dillig. 2023. **Automated Translation of Functional Big Data Queries to SQL**. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 95 (April 2023), 29 pages. https://doi.org/10.1145/3586047

1 INTRODUCTION

Big data analytics frameworks like Spark [Zaharia et al. 2010] and Flink [Carbone et al. 2015] have become increasingly popular for expressing queries over large amounts of data. These frameworks provide functional-style programming interfaces incorporating both higher-order APIs (e.g., map and filter) as well as first-order user-defined functions (UDFs). This style of programming is appealing because the higher-order components enable concise expression of computations over large amounts of data, while the first-order UDFs allow writing parts of the query in a familiar general-purpose programming language like Scala.

However, queries written in such functional APIs often turn out to be less performant than their pure SQL counterparts [Armbrust et al. 2015; Begoli et al. 2018], making it desirable to automatically transpile them to SQL. In fact, recognizing a similar problem for *imperative* query APIs, a number of automated translators have been proposed for generating SQL code from an

Authors' addresses: Guoqiang Zhang, North Carolina State Univeristy, United States, gzhang9@ncsu.edu; Benjamin Mariano, The University of Texas at Austin, United States, bmariano@cs.utexas.edu; Xipeng Shen, North Carolina State Univeristy, United States, xshen5@ncsu.edu; Işıl Dillig, The University of Texas at Austin, United States, isil@cs.utexas.edu.

2475-1421/2023/4-ART95
https://doi.org/10.1145/3586047

imperative implementation [Cheung et al. 2013; Emani et al. 2017; Noor and Fegaras 2020]. However, these approaches are ill-equipped to tackle the functional API translation problem because (1) they rely on intermediate representations specific to imperative APIs (e.g., Java ORM programs and array loops), and (2) they often rely on hand-crafted syntactic translation rules which limit their scope to a small set of stylized coding patterns.

Recently, Zhang et al. proposed a technique dubbed CLIS for automatically translating UDFs embedded in SQL queries to pure SQL expressions using a lazy inductive synthesis approach [Zhang et al. 2021]. Unlike the imperative translators mentioned before, CLIS does not rely on specific intermediate representations or hand-crafted syntactic translation rules, so it can, in principle, translate any UDF to SQL. However, CLIS only solves half of the problem: while CLIS is powerful for translating *embedded* UDFs to SQL expressions, it effectively degrades to brute-force search when translating *entire* functional queries with higher-order operators like map and filter. Because such higher order combinators are ubiquitious in functional APIs like Spark and Flint, CLIS is unable to handle most query transpilation tasks of interest in this paper (as demonstrated in Section 7.6).

To address this limitation, we propose a new technique for automatically translating functional big data queries (including higher-order operators) to semantically equivalent SQL versions. At a high level, our solution, like CLIS, is based on inductive program synthesis, so it can be applied to a wide class of programs without requiring hand-crafted translation rules to handle specific coding patterns. However, an obvious potential down-side of such a synthesis-based approach is scalability: Because all inductive synthesizers are based on *some* form of search, they tend to scale much more poorly compared to rule-based translation techniques. Furthermore, because we intend to solve a strictly harder synthesis problem than the one considered by CLIS, techniques from that paper are not sufficient for solving the full functional translation problem (as we explore in detail in Section 7.6).

The technique we propose in this paper aims to achieve the best of both worlds in terms of generality and scalability by combining inductive synthesis with a novel *query decomposition* technique. Unlike existing compositional program synthesis techniques [Alur et al. 2015, 2017; Guria et al. 2021; Smith and Albarghouthi 2016], our approach leverages a key insight from the database query domain: A query that produces a table with N columns can be decomposed into N different queries each of which produces one column of the output table. Our technique, which we call *column-wise decomposition*, leverages this insight by decomposing the full synthesis problem into several smaller synthesis problems (one for each column of the output) and *efficiently* merges the results into the desired SQL query. Because our decomposition strategy results in simpler synthesis problems compared to the original one, the proposed solution helps alleviate many of the scalability problems that are typically associated with inductive synthesis.

The key challenge in realizing this approach is that merging *arbitrary* queries is impractical, as composing them is often just as hard as synthesizing the full query. To overcome this challenge, we restrict each of the smaller queries to be instances of a shared *query sketch* that is amenable to efficient merging. In particular, our algorithm *automatically* generates suitable query sketches and uses them to decompose the top-level synthesis task into simpler sub-problems that can be solved using an off-the-shelf counterexample-guided inductive (CEGIS) approach. Critically, our decomposition strategy does not sacrifice the completeness of our overall approach – that is, if there exists an equivalent SQL expression, our technique is guaranteed to eventually find it.

In principle, our proposed synthesis algorithm can be applied to a broad class of query translation problems; however, our implementation focuses on input programs written in the Spark framework (one of the most common big data analytics frameworks) and using its functional RDD API (one of Spark's most popular APIs). Specifically, we have evaluated our tool, called RDD2SQL, on a set of 100 benchmarks collected from Github and find that most of these functional queries (79%)

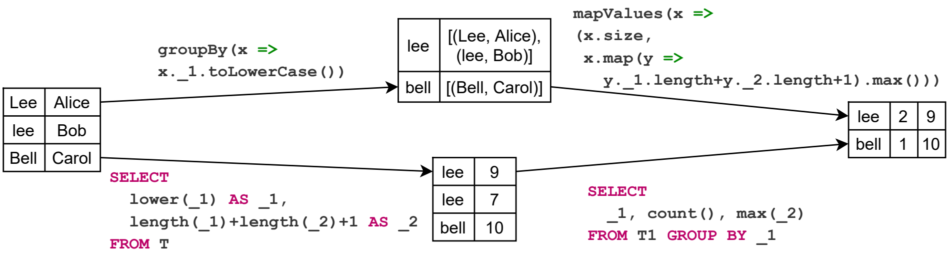
```
// T: RDD[(String, String)]
val T1 = T.groupBy(x => x._1.toLowerCase())
val O = T1.mapValues(x => (
    x.size,
    x.map(y => y._1.length + y._2.length + 1).max()
))
```

(a) Apache Spark Program

```
-- T: String | String
SELECT _1, count(), max(_2) FROM (
    SELECT lower(_1) AS _1,
           length(_1) + length(_2) + 1 AS _2
    FROM T
) GROUP BY _1
```

(b) SQL Query

Fig. 1. Apache Spark Program and Equivalent SQL Query



Lee	Alice
lee	Bob
Bell	Carol

```
groupBy(x =>
x._1.toLowerCase())
```

lee	[(Lee, Alice), (lee, Bob)]
bell	[(Bell, Carol)]

```
mapValues(x =>
 (x.size,
  x.map(y =>
    y._1.length+y._2.length+1).max()))
```

```
SELECT
  lower(_1) AS _1,
  length(_1)+length(_2)+1 AS _2
FROM T
```

lee	9
lee	7
bell	10

```
SELECT
  _1, count(), max(_2)
FROM T1 GROUP BY _1
```

lee	2	9
bell	1	10

Fig. 2. Semantic difference between functional APIs and SQL

can be translated to SQL and that RDD2SQL can automate this translation process for 96% of the functional queries it is evaluated on. Our evaluation also shows that the performance benefits of this translation are substantial: SQL queries are, on average, 2× faster than their RDD counterparts and up to 7× faster in some cases. Furthermore, we show that our technique significantly outperforms prior work: In particular, CLIS (extended to the functional translation problem) can solve way fewer benchmarks compared to RDD2SQL. Finally, our evaluation shows that the idea of column-based decomposition is crucial for the practicality of our approach. Without this decomposition strategy, nearly half of the benchmarks cannot be transpiled within a 60-minute time limit.

In summary, this paper makes the following contributions:

(1) We propose the first automated approach for translating functional big data queries to SQL.
(2) We describe a novel decomposition technique leveraging sketch-based program synthesis to split the full query synthesis problem into a set of smaller single-column synthesis problems.
(3) We prove that our column-wise decomposition technique is both sound and complete.
(4) We evaluate our implementation, RDD2SQL, on a benchmark of real-world Spark RDD programs and find that it can translate the vast majority of RDD programs to SQL and that this translation leads to significant performance benefits.

2 OVERVIEW

In this section, we give a high-level overview of our method using the Apache Spark program shown in Fig. 1a. Given a table with two `String` columns corresponding to last and first names respectively, the program computes a table with three columns: the first contains the unique last names (in lower-case), the second contains the number of occurrences of that last name, and the third records the length of the longest complete name (first and last) among people of that last name. As shown in Fig. 1a, this program computes the output table `O` by using higher-order functional operators like `groupBy` and `mapValues`. The SQL query Q shown in Fig. 1b computes the same output table but instead relies on relational operators like `SELECT`. In fact, not only are the operators different, but even the semantics of how they produce the desired output are different. For example, Fig. 2 shows the intermediate tables produced when executing both the Spark and SQL programs on a given input. As shown in this figure, although both expressions produce the same output, their intermediate results are very different. These syntactic and semantic differences between the source and target programs make the problem of translating the Spark program into an equivalent SQL query quite challenging.

As mentioned in Section 1, our approach addresses this translation problem using a form of *counterexample-guided inductive synthesis* (CEGIS). By way of background, the idea behind CEGIS is to inductively generalize from a set of input-output examples to conjecture a program P (in our case, a SQL expression) and check whether P satisfies the specification (in our case, whether P is equivalent to its functional version). If the second verification step fails, the CEGIS approach adds new examples and tries to perform inductive generalization from this larger set. This process continues until the verifier proves (or at least fails to disprove) equivalence.

As is evident from the above discussion, CEGIS uses the source program merely as a black-box for generating input-output examples, so it can be applied to a wide class of input programs [Ahmad et al. 2019; Sivaraman et al. 2016; Zhang et al. 2021]. However, this generality comes at the cost of scalability compared to best-effort rule-based translation approaches. In fact, from a practical standpoint, a straightforward application of CEGIS does not yield satisfactory results when transplating real-world functional queries to SQL.

Our approach alleviates the scalability bottleneck of CEGIS by leveraging the *column-based decomposition* idea mentioned in Section 1. To recap, the basic idea is as follows: rather than generating a SQL expression to construct the *entire* output table, we instead synthesize N different SQL expressions, one for each of the N columns, and merge them into a single SQL query for constructing the full output table. In particular, by carefully restricting the shape of these so-called *column queries* $Q_1, \ldots, Q_N$ to be instances of the *same query sketch* Q_s, it is possible to syntactically consolidate the N column queries into the desired SQL query in an efficient way.

Going back to our running example, consider the following query sketch Q_s:

Q_s

```
SELECT ?1 FROM (
    SELECT lower(_1) AS _1 ?2 FROM T
) GROUP BY _1
```

Here, the symbols $?_1$ and $?_2$ (in red) correspond to unknowns ("*holes*") which can be filled with an arbitrary expression. Observe that, by instantiating the holes of this sketch in different ways, we can obtain different columns of the desired output table. In particular, consider the SQL expressions Q_1, Q_2, Q_3 shown in Fig. 3. If we execute each column query Q_i on the input table from Fig. 2, we obtain exactly the i'th column of the output table shown on the right side of Fig. 2. Furthermore, by concatenating each of the three instantiations of the holes, we obtain precisely the desired SQL query in Fig. 1b.

As illustrated by this example, our transpilation approach works as follows: First, it constructs a query sketch Q_s with certain key properties that we describe in Section 5.2. Then, given the source program P, it (syntactically) extracts programs $P_1, \ldots, P_N$, where each program P_i produces the i'th column of the output. Next, it generates N different synthesis problems wherein the goal is to find a SQL query Q_i (referred to as a *column query*) that (1) is an instantiation of the query sketch Q_s and (2) is semantically equivalent to P_i. Because each of the N synthesis problems is often much simpler to solve than the original one, these sub-problems can be solved using a standard CEGIS approach. Finally, given solutions $Q_1, \ldots, Q_N$ to the synthesis sub-problems, our algorithm merges them in an efficient, syntax-directed way to obtain the final synthesis result Q. Because of the properties of our query sketches, the algorithm can guarantee that the produced SQL query Q is semantically

Q_1

```
SELECT _1 FROM (
    SELECT lower(_1) AS _1 FROM I
) GROUP BY _1
```

Q_2

```
SELECT count() FROM (
    SELECT lower(_1) AS _1 FROM T
) GROUP BY _1
```

Q_3

```
SELECT max(_2) FROM (
    SELECT
        lower(_1) AS _1,
        length(_1) + length(_2) + 1 AS _2
    FROM T
) GROUP BY _1
```

Fig. 3. Column queries for example

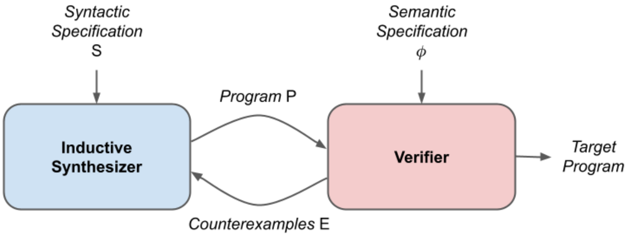


Fig. 4. CEGIS Overview

equivalent to the input program P as long as each sub-query Q_i is semantically equivalent to the extracted program P_i for producing the i'th column of the output.

3 BACKGROUND ON CEGIS

Since our solution is based on counterexample-guided inductive synthesis (CEGIS) [Solar-Lezama et al. 2006], we now give a brief overview of this method for readers who are new to program synthesis. As shown in Fig. 4, the CEGIS approach takes as input a semantic specification ϕ which describes the desired behavior of the output program as well as a syntactic specification $\mathcal{S}$ that constrains the high-level syntactic structure of the output program. Given these inputs, the goal of CEGIS is to produce a program satisfying both the semantic and syntactic constraints.

Overview. At a high level, the CEGIS paradigm involves a series of interactions between an *inductive synthesizer* and a *verifier* (see Fig. 4). The inductive synthesizer takes as input a set of input-output examples $\mathcal{E}$ (initialized to $\emptyset$) and the syntactic specification $\mathcal{S}$, and produces a program P whose semantics is consistent with all examples in $\mathcal{E}$ and whose syntax conforms to $\mathcal{S}$. The proposed candidate program P is then passed to a verifier, which is responsible for checking whether P actually satisfies the full semantic specification ϕ. If it does, P is returned as the solution; otherwise, the verifier generates a new input-output example that P does not satisfy but that any correct program *should* satisfy. This *counterexample* is added to the example set $\mathcal{E}$ and the inductive synthesizer is invoked again. This back and forth process continues until a program is found which matches the semantic specification.

Functional Query	P	$\longrightarrow$	$D_1, ..., D_n, C$
UDF Definition	D	$\longrightarrow$	def f$(a_1, ..., a_n) = \langle ScalaExpr \rangle$
Functional Call	C	$\longrightarrow$	$\langle API \rangle (C_1, ..., C_n, F_1, ..., F_n) \mid$
			$\langle input \rangle$
UDF	F	$\longrightarrow$	$\langle UDFName \rangle \mid \langle lambdaExpr \rangle$

Fig. 5. Functional source language grammar

Syntactic specification. The syntactic specification in the CEGIS paradigm consists of (1) the grammar of the target programming language (in our case, SQL), and (2) an (optional) program sketch [Solar-Lezama 2008] that further constrains the general structure of the target program. While the exact nature of the sketch depends on the application domain, a sketch contains *holes* that stand for unknown expressions to be synthesized. Generally, a sketch is useful both for constraining the search space and also for acting as a regularizer; so, the effectiveness of the inductive synthesizer often depends on the quality of the provided sketch. As discussed in Section 5.2, some of the contributions of this work include (a) the design of a suitable sketch language for our context, and (b) a technique for effectively generating useful sketches to feed to the inductive synthesizer.

Semantic specification. The CEGIS approach supports a variety of different semantic specifications, including logical specifications (e.g., expressed in first-order or temporal logic) as well as reference implementations written in a different programming language than the target program. Since our goal in this work is to transpile functional queries to SQL, our semantic specifications take the form of a reference implementation (i.e., Spark RDD query [Zaharia et al. 2012]).

Inductive synthesizer. In general, there are a number of ways to implement an inductive synthesizer, including search-based techniques and constraint solving [Gulwani et al. 2017]. However, most inductive synthesizers perform some form of enumeration, where candidate programs conforming to the syntactic specification are iteratively sampled from the target language and checked for compliance against the example set. Our proposed approach does not rely on a specific choice of inductive synthesizer; however, the one used in our implementation is described in Section 6.

Verifier. As the synthesizer performs inductive generalization from a set of input-output examples, the use of a verifier is crucial for ensuring the correctness of the synthesized program. In addition, the verifier is also responsible for constructing useful counterexamples that the inductive synthesizer can generalize from. Due to this counterexample requirement, verifiers that are based on *over-approximate* static analysis techniques like abstract interpretation [Cousot and Cousot 1977] are not suitable in this context. Hence, almost all instantiations of the CEGIS paradigm use techniques like symbolic execution or (bounded) model checking in order to produce valid counterexamples. As discussed in Section 6, we also adopt a similar approach.

4 PROBLEM STATEMENT

The problem that we address in this paper is an instance of *transpilation*, where we want to automatically translate an expression written in one programming language to an expression in another language. Specifically, Fig. 5 shows the syntax of our source (functional) language, and Fig. 6 shows that of the target (declarative) language.

Source language. A functional query in the source language consists of a list of user-defined function (UDF) definitions followed by a call to a functional API. A UDF defines a function f as an arbitrary Scala expression over its arguments $a_1, \ldots, a_n$. The functional API call can invoke any

$$
\begin{array}{llcl}
\text{SQL Query} & Q & \rightarrow & \Pi \mid \sigma \mid \bowtie \mid \cup \mid \mathcal{G} \mid \langle inputTable \rangle \mid ?_Q \\
\text{Project} & \Pi & \rightarrow & \texttt{Project}(Q, E_1, ..., E_n) \mid ?_\Pi \\
\text{Select} & \sigma & \rightarrow & \texttt{Select}(Q, E) \mid ?_\sigma \\
\text{Join} & \bowtie & \rightarrow & \texttt{Join}(Q_1, Q_2) \mid ?_\bowtie \\
\text{Union} & \cup & \rightarrow & \texttt{Union}(Q_1, Q_2) \mid ?_\cup \\
\text{Aggregate} & \mathcal{G} & \rightarrow & \texttt{Aggregate}(Q, \{C_1, ..., C_n\}, E_1, ..., E_n) \mid ?_\mathcal{G} \\
\text{Expression} & E & \rightarrow & C \mid \langle func \rangle (E_1, ..., E_n) \ \texttt{AS} \ \langle alias \rangle \mid ?_E \\
\text{Column} & C & \rightarrow & \langle columnName \rangle \mid \langle alias \rangle \mid ?_C
\end{array}
$$

Fig. 6. Target relational language grammar (with extended sketch grammar in blue)

of the 20 different API calls we support, including operators like map, filter, flatMap, groupBy, and aggregate (see Appendix A in the supplementary material for a complete list of supported APIs). In general, these APIs can take as input nested API calls, UDFs, and lambda expressions. Furthermore, the number and order of these arguments can vary by API. An input to a functional query can be either a list of scalars (i.e., a single-column table) or a list of tuples, which we view as a multi-column table. The output of the query can be either a list of scalars, a list of tuples, or a non-list (scalar/tuple). If the output of a query is a scalar (e.g., the return value of count() API) or a tuple, we treat it as a single-row table.

Target language. Our target language, shown in Fig. 6, is a SQL-like language that is trivial to translate into standard SQL in a syntax-directed way. We use the target language shown in Fig. 6 rather than standard SQL to simplify our presentation; however, the interested reader can find the translation rules between our target language and standard SQL in Appendix B (under supplementary material). In more detail, a relational query in the target language consists of a collection of common relational operators over subqueries and/or the input table. Unlike standard relational algebra that manipulates unordered sets or bags, our relational language processes and outputs ordered lists to match the data types in the source language. In addition, the join operator implicitly uses the first columns of both input tables to perform an inner join. Aggregate groups the input table by a set of columns and then aggregates each group by a list of aggregate expressions. The result of an Aggregate operation consists of the 'group-by' columns followed by the columns of aggregate results. Note that expressions in this grammar can introduce new columns by naming the result of a function call using the AS construct. Given an expression E, we use the notation Def(E) to refer to the name of the column defined via this AS construct. Conversely, we write Refs(E) to denote the column names referenced (rather than defined) in E.

Problem statement. Our goal in this work is to find a SQL query (defined by Fig. 6) that is equivalent to the given functional Spark program (defined by Fig. 5). Thus, in order to formalize our problem, we first need to state what it means for two such programs to be equivalent. For this purpose, we assume two operational semantics $[\![\cdot]\!]_{SQL}$ and $[\![\cdot]\!]_{Spark}$ which, given an input table T and program P, produces an output table T'.

Definition 1. **(Program equivalence)** *Given a functional program P and SQL query Q, we say that P is equivalent to Q, denoted $P \equiv Q$, if, for* all *input tables T, $[\![P]\!]_{SQL}(T) = [\![Q]\!]_{Spark}(T)$.*

In other words, we consider a target program to be equivalent to the provided source program if it always produces the same output table given the same input table.

Definition 2. **(Functional-to-SQL translation)** *Given a functional program P, the functional-to-SQL translation problem is to find a SQL query Q, such that $P \equiv Q$.*

Algorithm 1 Transpilation from functional query to SQL

```
function Spark2SQL($P$)
    Input: A functional program $P$
    Output: SQL query $Q$ equivalent to $P$ or $\perp$
    while true do
        $Q_s \leftarrow$ GetNextQuerySketch( )
        if $Q_s = \perp$ then return $\perp$
        $done \leftarrow$ true
        for each $i \in [1, ..., n]$ do
            $P_i \leftarrow P.\mathsf{map}(x \Rightarrow x.\_i)$
            $Q_i \leftarrow \mathrm{CEGIS}(P_i, Q_s)$
            if $Q_i = \perp$ then
                $done \leftarrow$ false
                break
        if $done$ then
            return Merge($Q_s, Q_1, \ldots, Q_n$)
```

5 TRANSLATION ALGORITHM

In this section, we describe our synthesis algorithm for translating functional programs to SQL queries. First, we introduce our main SQL query synthesis algorithm based on the idea of column-wise decomposition (Section 5.1). Then, we introduce dependency free query sketches (Section 5.2) and show how we can use them for efficient SQL query synthesis (Section 5.3). Finally, we prove that our algorithm is sound and complete and discuss its time complexity (Section 5.4).

5.1 Column-Wise Compositional Synthesis

In this subsection, we describe our SQL query synthesis technique based on the idea of column-wise composition. Our transpilation procedure is shown in Algorithm 1 and takes as input a functional program P and outputs a SQL query that is semantically equivalent to P if one exists (and $\perp$ if it fails to find one).

As stated earlier, our column-wise compositional synthesis approach hinges on the following key observation: a column query Q_i that only outputs the i'th column of Q is usually simpler than Q and hence easier to synthesize. However, to leverage this observation, we need a way of efficiently constructing the full query from a given set of column queries. To address this problem, our approach only considers column queries that are instantiations of the *same* query sketch, defined as follows:

Definition 3. **(Query sketch)** *A query sketch is a SQL expression that can contain holes. More formally, a query sketch is a string in the extended sketch grammar of Fig. 6 (in blue) which augments the grammar of the target language by allowing unknowns (or* holes*) $?_N$, where N is the nonterminal which produces that unknown.*

At a high level, the synthesis algorithm works by iterating over the space of all possible query sketches, and, for each query sketch, the algorithm attempts to complete that sketch for each column of the desired output. In particular, given a query that produces a table with N columns, the basic idea is to synthesize a so-called *column query* Q_i for each column. Each Q_i is an instantiation of the same query sketch Q_s and is equivalent to $P_i = P.\mathsf{map}(x \Rightarrow x.\_i)$. In other words, for every input table T, the synthesized column query is guaranteed to produce the i'th column of $P(T)$. The

DFQS	Q_s	$\rightarrow$	$\texttt{Project}(Q, ?)$
Sub sketch	Q	$\rightarrow$	$\Pi \mid \sigma \mid \bowtie \mid \cup \mid \mathcal{G} \mid \langle \mathit{inputTable} \rangle$
Project	Π	$\rightarrow$	$\texttt{Project}(Q, E_1, ..., E_n, ?)$
Select	σ	$\rightarrow$	$\texttt{Select}(Q, E)$
Join	$\bowtie$	$\rightarrow$	$\texttt{Join}(Q_1, Q_2)$
Union	$\cup$	$\rightarrow$	$\texttt{Union}(Q_1, Q_2)$
Aggregate	$\mathcal{G}$	$\rightarrow$	$\texttt{Aggregate}(Q, \{C_1, ..., C_n\}, E_1, ..., E_n, ?)$
Expression	E	$\rightarrow$	$C \mid \langle \mathit{func} \rangle (E_1, ..., E_n)$ AS $\langle \mathit{alias} \rangle$
Column	C	$\rightarrow$	$\langle \mathit{columnName} \rangle \mid \langle \mathit{alias} \rangle$

Fig. 7. DFQS Grammar

synthesis of column queries is performed using the standard CEGIS technique (see Section 6 for more details) and utilizing Q_s as the sketch and P_i as the specification. If synthesis of any column query fails (lines 11-13), then the algorithm moves on to the next query sketch. On the other hand, if it can successfully synthesize all N column queries based on Q_s, then it constructs the full SQL query by calling Merge at line 15.

The key challenge in this algorithm is to efficiently generate a query sketch Q_s such that (1) each desired column query $Q_i \equiv P_i$ is a completion of Q_s and (2) all column queries can be efficiently merged into the desired query. Achieving both of these goals is nontrivial. In particular, if we leave query sketches unrestricted (i.e. GetNextQuerySketch() can return any query sketch from Figure 6), we can easily find a query sketch Q_s such that each Q_i is a completion of Q_s (e.g., $Q_s = ?_Q$); however, there is no guarantee that the produced column queries Q_i from Q_s can be *efficiently* merged. On the other hand, if we restrict the space of query sketches too much, we might make the merging task very easy, albeit at the cost of completeness.

5.2 Dependence-Free Query Sketches

To address the key challenge outlined in the previous subsection, we introduce so-called *dependence-free query sketches*, a restricted set of query sketches that admit an efficient merging algorithm while not sacrificing completeness:

Definition 4. **(DFQS)** *A* dependence-free query sketch (DFQS) *is a query sketch Q_s with the following two restrictions:*

(1) **(Syntactic)** *Holes in Q_s can only appear as arguments of projection and aggregation operators. In particular, Q_s must adhere to the grammar shown in Figure 7.*
(2) **(Semantic)** *For every hole-free subexpression E of Q_s, E must evaluate to the same value when executing any instantiation Q_i of Q_s on the same input.*

As stated in the above definition, our synthesis algorithm imposes both syntactic and semantic restrictions on the query sketches it considers. In particular, the syntactic restriction makes it possible to efficiently merge different column queries in linear time, while the semantic restriction ensures the correctness of this merge procedure. Furthermore, as we state formally in Section 5.4, these syntactic and semantic restrictions on the query do not affect the completeness of our synthesis algorithm. Intuitively, this is the case because any target query Q can be expressed as an instantiation of the sketch $Q_s \equiv \textsc{Project}(Q, ?)$, which conforms to the syntactic and semantic restrictions in the DFQS definition.

Algorithm 2 Procedure for consolidating column queries

```
function MERGE($Q_s, Q_1, \ldots, Q_n$)
    Input: DFQS $Q_s$
    Input: Completions $Q_i$ of $Q_s$
    Output: A SQL query
    $\sigma \leftarrow \{(?_i, []) \mid ?_i \in \text{Domain}(Q_s)\}$
    for each $i \in [1, \ldots, n]$ do
        $\sigma_i \leftarrow \text{ExtractMapping}(Q_s, Q_i)$
        for each $?_i \in \text{Domain}(\sigma)$ do
            $\sigma[?_i] \leftarrow \sigma[?_i] \cdot \sigma_i[?_i]$
    return $Q_s[\sigma]$
```

Example 5.1. Consider the following query sketch where T is the input table and x is a column of table T:

$$\text{Project}(\text{Aggregate}(\text{Project}(T, MOD(x, 2)\ AS\ c_1), \{c_1\}), ?_1)$$

This sketch is dependence-free because the values of concrete subexpressions are not affected by how $?_1$ is instantiated. Now, consider the following sketch obtained by replacing the expression '$MOD(x, 2)\ AS\ c_1$' in the previous sketch with a hole:

$$\text{Project}(\text{Aggregate}(\text{Project}(T, ?_2), \{c_1\}), ?_1)$$

This new sketch is *not* dependence-free because there exist instantiations $\sigma_1 = \{?_2 \mapsto [1\ AS\ c_1], ...\}$ and $\sigma_2 = \{?_2 \mapsto [2\ AS\ c_1], ...\}$ that feed different values to c_1 in Aggregate.

We conclude this section by formally defining the instantiation of a DFQS:

DEFINITION 5. **(DFQS instantiation)** *We say that a query Q is an instantiation of DFQS Q_s if it can be obtained from Q_s through a substitution $\sigma = \{?_1 \mapsto L_1, \ldots ?_n \mapsto L_n\}$ where (1) $\{?_1, \ldots, ?_n\}$ is exactly the set of holes in Q_s and (2) each L_i is a (possibly empty) list of concrete expressions $E_1, \ldots, E_n$. In this case, we write $Q = Q_s[\sigma]$.*

We illustrate the above definition through an example:

Example 5.2. The queries Q_1, Q_2, and Q_3 in Fig. 3 are instantiations of Q_s in Section 2 through the following substitutions respectively:

$$\begin{aligned}
&\sigma_1 = \{?_1 \mapsto [\texttt{\_1}], ?_2 \mapsto []\} \\
&\sigma_2 = \{?_1 \mapsto [\texttt{count()}], ?_2 \mapsto []\} \\
&\sigma_3 = \{?_1 \mapsto [\texttt{max(\_2)}], ?_2 \mapsto [\texttt{length(\_1)} + \texttt{length(\_2)} + 1\ \texttt{AS \_2}]\}
\end{aligned}$$

As the above example illustrates, a sketch instantiation can map a hole in the sketch to nothing (i.e., an empty list). In the next example, we illustrate that a sketch could also be instantiated by mapping a hole to a list with multiple elements:

Example 5.3. The query in Fig. 1b corresponds to $Q_s[\sigma]$ where σ is the following substitution:

$$\begin{aligned}
\sigma \quad = \quad &\{?_1 \mapsto [\texttt{\_1}, \texttt{count()}, \texttt{max(\_2)}], \\
&\ ?_2 \mapsto [\texttt{length(\_1)} + \texttt{length(\_2)} + 1\ \texttt{AS \_2}]\}
\end{aligned}$$

5.3 Efficient Column-Wise Compositional Synthesis with DFQS

As mentioned previously, the main benefit of DFQS for column-wise synthesis is that they permit an efficient merging algorithm. In this section, we will first describe this merging algorithm, prove it correct, and then describe an efficient algorithm for enumerating dependence-free query sketches.

5.3.1 Efficient Merging with DFQS. Algorithm 2 gives a linear-time algorithm for consolidating the column query completions of a DFQS. Given a set of column queries $Q_1, \dots, Q_n$ that are instantiations of the same DFQS Q_s, Merge produces a query Q with the following property: For any table T such that $Q_i(T) = T_i$ where T_i is a single column, then $Q(T)$ produces a table with columns $T_1, \dots, T_n$. Thus, assuming that each column query Q_i is equivalent to program P_i from line 9 of Algorithm 1, the result of calling Merge produces a query Q that is equivalent to the input query P in the source language.

Due to the dependence-freedom assumption and restrictions on where holes can appear in a DFQS, our technique can efficiently merge the column queries into a full query in linear time. In particular, the Merge algorithm works as follows: for each column query Q_i, we first extract a substitution σ_i such that $Q_i = Q_s[\sigma_i]$ (recall Definition 5). Then, we generate a new substitution σ such that, for any hole $?_j$, $\sigma[?_j]$ is the *concatanation* of all $\sigma_i[?_j]$'s. Finally, we obtain the merged query as $Q_s[\sigma]$. In other words, the merging of column queries simply boils down to concatenating the instantations of the holes in each column query. Furthermore, as formalized by the following theorem, this simple merge procedure is guaranteed to return a correct query:

Theorem 5.4. **(Merge correctness).** *Let Q be the result of calling* Merge *on column queries $Q_1, \dots, Q_n$ that are instantiations of sketch Q_s. Then, for any input table T, $\Pi(Q(T), i) = Q_i(T)$.*

Proof. See Appendix C in the supplementary material. □

5.3.2 Efficient DFQS Enumeration. Our synthesis algorithm from the previous subsection assumes that we can iterate over the space of all possible DFQSs. However, this is difficult in practice because it is unclear how to effectively enumerate all possible dependence-free query sketches. Furthermore, even if we had a reasonable enumeration strategy, many of the enumerated sketches would likely be useless, so this strategy would be inefficient in practice.

To deal with this concern, we propose an optimized version of Algorithm 1 that does not require explicitly enumerating all DFQSs. Instead, the idea is to perform synthesis for a single column first (without using any sketch), then extract a sketch Q_s from the synthesized column query, and finally perform synthesis for the remaining columns based on Q_s. This approach makes our algorithm more practical because (1) we do not need to consider useless sketches that cannot be instantiated for any column and (2) we can extract the sketch from the column query in such a way that the resulting sketch is guaranteed to be dependence-free.

Algorithm 3 shows the optimized version of our synthesis algorithm based on the above idea. The key difference from Algorithm 1 is that the call GetNextQuerySketch is replaced by an invocation of CEGIS and ExtractDFQS in lines 6–7. Specifically, we first synthesize a column query Q_1 for the first column (line 6) by calling CEGIS with an empty sketch (indicated by $\perp$), Then, the call at line 7 to ExtractDFQS (presented in Fig. 8 and discussed later) extracts a query sketch from Q_1. This query sketch Q_s is then used when synthesizing column queries for the remaining columns in lines 10–15. As in Algorithm 1, the loop terminates when queries for all n columns have been synthesized, and line 17 consolidates these n column queries into a single one using the same Merge procedure from Algorithm 2.

The idea behind the ExtractDFQS procedure, shown in Fig. 8, is quite simple: Given a query Q, it generates *the most general sketch* Q_s such that (1) Q is an instantiation of Q_s, and (2) Q_s satisfies the dependence-freedom assumption. The basic idea is to keep *exactly* those concrete expressions

Algorithm 3 Transpilation with DFQS extraction

```
function Spark2SQL-Opt($P$)
    Input: A functional program $P$
    Output: SQL Query $Q$ equivalent to $P$ or $\perp$
    while true do
        $P_1 \leftarrow P.\mathsf{map}(x \Rightarrow x.\_1)$
        $Q_1 \leftarrow \text{CEGIS}(P_1, \perp)$
        $Q_s \leftarrow \textsc{Extract}\text{DFQS}(Q_1)$
        if $Q_s = \perp$ then return $\perp$
        $done \leftarrow \mathsf{true}$
        for each $i \in [2, ..., n]$ do
            $P_i \leftarrow P.\mathsf{map}(x \Rightarrow x.\_i)$
            $Q_i \leftarrow \text{CEGIS}(P_i, Q_s)$
            if $Q_i = \perp$ then
                $done \leftarrow \mathsf{false}$
                break
        if $done$ then
            return $\textsc{Merge}(Q_s, Q_1, \ldots, Q_n)$
```

$$\textsc{Extract}\text{DFQS}(Q) = \texttt{Project}(Q', ?) \text{ where } \varnothing \vdash Q \rightsquigarrow Q'$$

$$\textsc{Project}\qquad \frac{\begin{array}{c}\mathcal{E}' = \{E \mid E \in \mathcal{E} \wedge \mathsf{Def}(E) \in C\} \\ \mathsf{Refs}(\mathcal{E}') \vdash Q \rightsquigarrow Q'\end{array}}{C \vdash \texttt{Project}(Q, \mathcal{E}) \rightsquigarrow \texttt{Project}(Q', \mathsf{Concat}(\mathcal{E}', ?))}$$

$$\textsc{Select}\qquad \frac{\begin{array}{c}C' = C \cup \mathsf{Refs}(\{E\}) \\ C' \vdash Q \rightsquigarrow Q'\end{array}}{C \vdash \texttt{Select}(Q, E) \rightsquigarrow \texttt{Select}(Q', E)}$$

$$\textsc{Join}\qquad \frac{\begin{array}{cc}C_1 = C \cup \{\mathsf{FirstColumn}(Q_1)\} & C_2 = C \cup \{\mathsf{FirstColumn}(Q_2)\} \\ C_1 \vdash Q_1 \rightsquigarrow Q_1' & C_2 \vdash Q_2 \rightsquigarrow Q_2'\end{array}}{C \vdash \texttt{Join}(Q_1, Q_2) \rightsquigarrow \texttt{Join}(Q_1', Q_2')}$$

$$\textsc{Union}\qquad \frac{C \vdash Q_1 \rightsquigarrow Q_1' \quad C \vdash Q_2 \rightsquigarrow Q_2'}{C \vdash \texttt{Union}(Q_1, Q_2) \rightsquigarrow \texttt{Union}(Q_1', Q_2')}$$

$$\textsc{Aggregate}\qquad \frac{\begin{array}{c}\mathcal{E}' = \{E \mid E \in \mathcal{E} \wedge \mathsf{Def}(E) \in C\} \\ C'' = C' \cup \mathsf{Refs}(\mathcal{E}') \\ C'' \vdash Q \rightsquigarrow Q'\end{array}}{C \vdash \texttt{Aggregate}(Q, C', \mathcal{E}) \rightsquigarrow \texttt{Aggregate}(Q', C', \mathsf{Concat}(\mathcal{E}', ?))}$$

Fig. 8. Rules for extracting DFQS from a query.

in the input query Q that are necessary for satisfying dependence-freedom and replacing the remaining expressions with holes. Intuitively, this sketch generation procedure allows reusing the

Table 1. Example of extracting DFQS from a query (Example 5.5)

C	Query	Rule	Intermediate values	Sketch
$\varnothing$	$Q = \text{Project}(Q_1, c_2)$	Project	$\mathcal{E} = \{c_1\}$ $\mathcal{E}' = \varnothing$ $\text{Refs}(\mathcal{E}') = \varnothing$	$Q_s = \text{Project}(Q'_1, ?)$
$\varnothing$	$Q_1 = \text{Select}(Q_2, c_1 > 0)$	Select	$E = c_1 > 0$ $\text{Refs}(\{E\}) = \{c_1\}$ $C' = \{c_1\}$	$Q'_1 = \text{Select}(Q'_2, c_1 > 0)$
$\{c_1\}$	$Q_2 = \text{Project}(T, E_1 \text{ AS } c_1, E_2 \text{ AS } c_2)$	Project	$\mathcal{E} = \{E_1 \text{ AS } c_1, E_2 \text{ AS } c_2\}$ $\mathcal{E}' = \{E_1 \text{ AS } c_1\}$	$Q'_2 = \text{Project}(T, E_1 \text{ AS } c_1, ?)$

general structure of the synthesized column query while allowing maximal generalization for the remaining column queries without violating dependence-freedom.

In more detail, the ExtractDFQS procedure is presented in Fig. 8 using judgments of the following form:

$$C \vdash Q \rightsquigarrow Q_s$$

Here, C refers to the set of column names referenced in the parent query of Q. The meaning of this judgment is that, under the assumption that Q's parents reference columns C, the most general sketch that generalizes Q without violating dependence-freedom is Q_s. Intuitively, the column names on the left-hand-side of the entailment are used for determining which expressions in Q can be replaced while respecting the dependence-freedom requirement.

Since many of these rules in Fig. 8 are similar to each other, we only explain the Project rule to give the reader some intuition. Consider a sub-query of the form `Project`$(Q, \mathcal{E})$ where the parent query references columns C. The expressions $\mathcal{E}'$ in the generated sketch only include those expressions in $\mathcal{E}$ that define a column referenced by the parent query; the rest of the expressions are replaced by a hole. Note that expressions in $\mathcal{E}'$ *cannot* be further generalized (i.e., replaced by a hole) without violating the dependency-freedom requirement, because, otherwise, the result of the parent query would be dependent on how the hole is instantiated. In addition to replacing *some* of the expressions in the query by holes, the Project rule also (recursively) generalizes the sub-query Q to a DFQS Q'. Since the columns referenced in Q's parent query are the definitions in $\mathcal{E}'$, we use $\text{Refs}(\mathcal{E}')$ as the "context" when generalizing from sub-query Q to a query sketch Q'.

Example 5.5. Consider the following query Q:

$$\text{Project}(\text{Select}(\text{Project}(T, E_1 \text{ AS } c_1, E_2 \text{ AS } c_2), c_1 > 0), c_2)$$

Table 1 presents a step-by-step evaluation of the ExtractDFQS procedure on Q. In the table, each row applies a rule for the current relational operation and recursively uses the next row to extract a sub-sketch from its subquery. The resulting sketch Q_s is:

$$\text{Project}(\text{Select}(\text{Project}(T, E_1 \text{ AS } c_1, ?), c_1 > 0), ?)$$

We conclude this section with a theorem stating that any query sketch extracted via ExtractDFQS satisfies Definition 4:

Theorem 5.6. **(Correctness of ExtractDFQS)** *For any query Q,* ExtractDFQS(Q) *yields a dependence-free query sketch.*

Proof. See Appendix D in the supplementary material. □

5.4 Properties of the Algorithm

Thus far, we have given an algorithm for RDD-to-SQL transpilation. In this section, we prove that our algorithm is sound and complete and show that the proposed column-wise decomposition idea leads to an exponential reduction of the search space.

5.4.1 Soundness. Assuming the soundness of the underlying CEGIS procedure, our proposed column-wise decomposition technique preserves the soundness of the overall appraoch, as stated by the following theorem:

Theorem 5.7. **(Soundness)** *Let Q be the output of Algorithm 3 on input query P. Then, if $Q \neq \bot$, we have $P \equiv Q$.*

Proof. See Appendix E in the supplementary material. □

5.4.2 Completeness. Our algorithm is also complete in the sense that the column-wise decomposition idea does not cause us to miss any transpilation opportunities. More formally, we can state the completeness result as follows:

Theorem 5.8. **(Completeness.)** *Under the assumption that the underlying CEGIS procedure is complete and that there exists a SQL query that is equivalent to the input program P, Algorithm 3 will return a query Q such that (1) $Q \neq \bot$, and (2) $Q \equiv P$.*

Proof. See Appendix F in the supplementary material. □

While we leave the full proof to the appendix, we now provide a high-level sketch of the proof here to provide the reader with some intuition. The proof relies on a key observation: for any input program P and equivalent query Q, any column query Q_i of Q can be expressed as $\Pi(Q, c_i)$, i.e., the projection of the full desired query Q onto the i^{th} column. Thus, assuming CEGIS is complete, CEGIS($P_1, \bot$) (line 6 of Algorithm 1) will eventually return $Q_1 = \Pi(Q, c_1)$ and ExtractDFQS(Q_1) (line 7) will produce the DFQS $Q_s = \Pi(Q', ?)$, where Q' is obtained by replacing some of the expressions of Q with holes according to the rules in Figure 8. Because Q is a completion of Q' and each Q_i is a completion of $\Pi(Q, ?)$, CEGIS(P_i, Q_s) is guaranteed to (eventually) return $Q_i = \Pi(Q, c_i)$. Finally, an equivalent query can always be obtained from Merge($Q_s, Q_1, \ldots, Q_n$) as shown in Theorem 5.4. To further explain this idea, consider the following example:

Example 5.9. Let us assume a source functional program P that unions two tables T_1 and T_2 that both have columns c_1 and c_2. In this case, P is equivalent to the query $\mathsf{Union}(T_1, T_2)$. As discussed above, let us suppose CEGIS($P_1, \bot$) returns the first column query $Q_1 = \mathsf{Project}(\mathsf{Union}(T_1, T_2), c_1)$. Calling *ExtractDFQS* on column query Q_1 produces the sketch $Q_s = \mathsf{Project}(\mathsf{Union}(T_1, T_2), ?)$ and the call CEGIS(P_2, Q_s) produces $Q_2 = \mathsf{Project}(\mathsf{Union}(T_1, T_2), c_2)$. Finally, Merge($Q_s, Q_1, Q_2$) produces $\mathsf{Project}(\mathsf{Union}(T_1, T_2), c_1, c_2)$ which is equivalent to P.

As shown in the example above, while our synthesis procedure is complete, it is not guaranteed to return the simplest equivalent program, just some equivalent program. However, as we examine in more detail in Section 7.7, modern SQL engines can effectively optimize away redundant code produced by the synthesizer (like the example above).

5.4.3 Benefits of Columnwise Decomposition on Time Complexity. We now briefly discuss the time complexity benefits of our proposed column-wise decomposition idea.

First, because our approach relies on an underlying CEGIS solver, the complexity benefits of column-wise decomposition are dependent on the runtime complexity of CEGIS. In general, inductive synthesis requires searching through a space that is exponential in the size of the target program in the worst case; hence, in the following discussion, we model the complexity of CEGIS as $O(r^k)$ where r is the number of productions in the grammar of the target language and k is the (AST) size of the program to be synthesized. We will use the notation $|P|$ to denote the AST size of program P.

Now, let Q be the target query which produces a table with N columns. Thus, a baseline approach that uses standard CEGIS without column-wise decomposition would have time complexity:

$$O(r^{|Q|}) \tag{1}$$

Now, to reason about the complexity of the method *with* column-wise decomposition, recall that Algorithm 3 (1) first synthesizes a query Q_1 for the first column using CEGIS, (2) then extracts a sketch Q_s from Q_1 in linear time, (3) then performs synthesis of the remaining column queries $Q_2, \ldots, Q_N$ using CEGIS (going back to step (1) upon failure), and (4) finally merges these queries. Thus, the overall complexity is the time to perform steps (2)-(4) *multiplied* by the time to perform step (1). [1] Furthermore, note that (2) and (4) are both linear time, so we can exclude those steps from our analysis. For step (1), the time complexity is $O(r^{|Q_1|})$. For step (3), let H be the number of holes in Q_s, Q_h^i be the completion of hole h in column query i, and L be the maximum AST size of any Q_h^i. We can now express the complexity of step (3) as $O(r^{|Q_1|} \times (N-1) \times r^{H \times L})$. Hence, the total worst-case time complexity is:

$$O(r^{|Q_1|} \times (N-1) \times r^{H \times L}) = O((N-1) \times r^{|Q_1|+H \times L}) \tag{2}$$

However, in practice, we find that backtracking from step (3) to step (1) never happens — i.e., the query sketch extracted from the first Q_1 is always sufficient for overall synthesis to succeed, so under this assumption, the complexity simplifies to:

$$O(r^{|Q_1|} + (N-1) \times r^{H \times L}) \tag{3}$$

Now, to understand the benefits of column-wise decomposition, let us compare Equations 1 and 3. Clearly, the benefits of decomposition depend upon the relationship between $|Q|$, $|Q_1|$, and $H \times L$. In practice, we find that $|Q_1| < |Q|$ and $H \times L < |Q|$; thus, under this assumption, the column-wise decomposition technique exponentially reduces the running time of the algorithm. In Section 7.4, we empirically compare the values of $|Q|$, $|Q_1|$, $H \times L$ observed in our benchmarks to further justify the benefits of our proposed column-wise decomposition technique. We also evaluate the number of backtracking steps to justify the assumption that allows us to simplify Equation 2 to Equation 3.

6 IMPLEMENTATION AND OPTIMIZATIONS

We implemented our proposed algorithm in a new tool called RDD2SQL that targets Scala programs written using the Spark RDD API [Zaharia et al. 2010], which is the functional query API for Apache Spark. We believe that Spark RDD is a good application for our approach because it is the de facto functional query engine. Our tool is implemented in Python and uses the Trinity [Martins et al. 2019] program synthesis tool as the inductive synthesis backend. This section describes salient implementation details of RDD2SQL as well as some important optimizations that allow it to scale to real-world translation tasks.

6.1 Source Program Analysis

Our implementation leverages the source code of the input RDD query to further speed up synthesis. This optimization is based on the following key observation: although the high-level structures of the source and target queries are quite different, they nonetheless often share related subexpressions. For instance, consider the expression `length(_1)+length(_2)+1` from the target SQL query in

[1] As standard, this analysis assumes that synthesis of Q_1 is incremental, meaning that we do not start synthesis from scratch every time, but rather continue from where the inductive synthesizer left off.

Fig. 1. Even though this expression is not *exactly* identical to any expression in the source program, it is quite closely related to the source expression `y._1.length+y._2.length+1` in Fig. 1.

To exploit this observation, we leverage CLIS [Zhang et al. 2021], a tool from prior work that can synthesize an equivalent SQL expression for a given user-defined function. Note that, unlike this work, CLIS can only translate UDFs to SQL expressions with *no relational operators* (like Project and Join). However, we can nonetheless use CLIS to learn equivalent SQL expressions (with no relational operators) for subexpressions of UDFs appearing in the source program and then add these translated subexpressions to the grammar of the target language, as done in prior work [Pailoor et al. 2021]. In particular, our algorithm uses static analysis to extract relevant UDFs (i.e., UDFs translatable by CLIS) from the source program and then invokes CLIS on each of these to learn a SQL expression equivalent to the UDF. Then, for each SQL expression discovered in this manner, it adds new rules to the grammar of the target language. For example, invoking CLIS on `y._1.length+y._2.length+1` results in the SQL expression `length(_1)+length(_2)+1`, so we add productions like $E \rightarrow$ `length(_1)` + `length(_2)` + 1 and $E \rightarrow$ `length(_1)` to the grammar of the target language. This grammar augmentation strategy allows us to re-use partial synthesis results obtained by invoking CLIS on UDFs used in the source query.

Furthermore, we can tailor this process for the synthesis of each column-query individually. To do this, we build off the observation that some expressions are only used in the calculation of particular output columns. For example, notice that the expression `y._1.length+y._2.length+1` is only used in the calculation of the values for the third column. We mechanize this idea by performing data-flow analysis to determine which expressions in the source program contribute to which output columns and construct a separate grammar for each column-query.

6.2 API-Level Decomposition

In addition to our novel column-wise decomposition idea, RDD2SQL also employs other types of decomposition strategies proposed in prior work [Zhang et al. 2021]. Specifically, RDD2SQL initially decomposes the source program into a data flow graph (DFG) where each node contains only one functional API call and tries to perform translation for each node independently. If RDD2SQL fails to find an equivalent SQL query for a specific node, it merges the node with an adjacent node and retries synthesis. This try-and-merge process continues until all nodes in the data flow graph are successfully translated. Finally, RDD2SQL composes the results together to obtain the target SQL query using the same technique described in [Zhang et al. 2021]. This DFG-based decomposition can be viewed as an optimization of the underlying CEGIS solver for the case where the semantic specification takes the form of a reference implementation.

6.3 Inductive Synthesizer

Recall that the underlying CEGIS solver consists of an *inductive synthesizer* and a *verifier*. Our underlying CEGIS solver leverages Trinity [Martins et al. 2019], an extensible framework for synthesizing programs that are consistent with a given set of input-output examples. Trinity is parametrized over the (1) the grammar of the target language, as well as (2) the (optional) *abstract semantics* [Cousot and Cousot 1977] of the target language. In particular, the abstract semantics are provided as (overapproximate) logical specifications of constructs in the language and are used for pruning the search space. To instantiate Trinity in our setting, we use the SQL grammar from Fig. 6 and write logical specifications for reasoning about the *number* of rows/columns in the table as well as the *types* of those columns. As mentioned in prior literature [Feng et al. 2017; Wang et al. 2017], reasoning about table dimensions provides good pruning power without adding too much overhead to the inductive synthesizer.

6.4 Equivalence Checking

Since our specification is in the form a reference implementation, the verifier underlying the CEGIS solver is a program equivalence checker based on symbolic model checking. Specifically, our equivalence checker works as follows: First, we create a harness program that invokes both the input functional query and the target SQL query on a *symbolic* input table and then asserts that the query outputs are the same for any arbitrary input. We then feed this harness program to CBMC [Clarke et al. 2004], a state-of-the-art symbolic model checker targeting the C language. In order to leverage CBMC for this purpose, we compile both the functional and SQL queries to the C programming language by modeling a table as an array of structs and implementing C functions to model SQL operators as well as those provided by the Spark RDD API.

6.5 Optimizing the CEGIS Loop

In the standard CEGIS paradigm, recall that the inductive synthesizer starts with an empty example set, which is augmented with additional examples provided by the verifier in each iteration. However, since each CEGIS iteration can be expensive, our implementation performs an optimization to reduce the number of interactions between the verifier and the inductive synthesizer. Specifically, rather than starting with an empty set of examples, we instead seed the CEGIS loop with carefully chosen test cases that help reduce the number of iterations. In our implementation, we first tried seeding the CEGIS loop with a set of random inputs; however, we found that this strategy provides insufficient path coverage. Inspired by techniques in coverage-guided fuzzing [Beyer et al. 2013; Holzer et al. 2010], our implementation instead inserts `assert(false)` statements in the original query and then uses CBMC to generate inputs that reach these failing assertions. We found that this model-checker-guided test generation strategy allows us to seed the CEGIS loop with a good set of initial examples from which effective inductive generalization can be performed.

7 EVALUATION

In this section, we present the results of our evaluation that is designed to answer the following research questions:

- **RQ1.** How often can functional big data queries be rewritten to semantically equivalent SQL?
- **RQ2.** How effective is our method at translating functional big data queries to SQL?
- **RQ3.** How important is the proposed column-wise decomposition idea?
- **RQ4.** What is the impact of the optimizations from Section 6 on synthesis time?
- **RQ5.** How does our technique compare to CLIS [Zhang et al. 2021]?
- **RQ6.** What are the limitations of our technique?
- **RQ7.** How much performance benefit is achieved by by translating functional big data queries to SQL using our method?

All experiments were run on a Linux machine with Intel Xeon Silver 4114 CPU @ 2.20GHz.

7.1 Benchmarks

To answer these research questions, we collected a set of 100 functional queries written in the Spark RDD API. To collect these benchmarks, we downloaded all Github repositories that have at least one star and that contain calls to the Spark RDD API. We then extracted all non-trivial Spark RDD programs (e.g., containing at least three RDD API calls) and randomly sampled 100 to use for our evaluation. Table 2 gives statistics about these functional queries in terms of the number of functional API calls, LOC, and AST size.

Table 2. Statistics about the benchmark set

Total # of functional queries	100
# API calls per query	3-14, avg 5
LOC per query	4-28, avg 9
AST size per query	39-342, avg 104

Table 4. Main synthesis results

Total supported benchmarks	57
Synthesized in one hour	55 (96%)
Median synthesis time	97 seconds
Average synthesis time	263 seconds

7.2 Manual Study

To assess how many big-data queries can be rewritten to SQL, we performed a manual study, summarized in Table 3, of the 100 benchmarks. We found that 79 of the 100 benchmarks do have equivalent SQL queries. For most of the benchmarks that do not have equivalent SQL queries (14 of 21), the reason is that they contain API calls and/or UDFs that access dynamic values (e.g., information from configuration files) which are not expressible in SQL. The other 7 programs cannot be translated because they define a custom column type or contain a UDF that modifies the global state.

Additionally, there are 22 benchmarks involving language features not yet supported by RDD2SQL but with no fundamental limitation in their ability to be translated. These features include SQL window functions (e.g., running total), non-scalar column types (e.g., list), 'ORDER BY' operator, random sampling functions, and Scala features not handled by our verifier. We leave extension of RDD2SQL to these features as future development.

Table 3. Manual study results

Total benchmarks	100
Supported	57
Total non-translatable	21
Dynamic	14
User-defined types	6
Side-effect	1
Total not supported	22
Window functions	7
Non-scalar column types	5
'ORDER BY' operator	3
High order functions	2
Random sampling	2
Not modeled Scala features	3

> **Result for RQ1:** Most RDD queries in our benchmark set (79%) are expressible in SQL, and a majority of those (72%) are supported for translation by RDD2SQL.

7.3 Effectiveness of Synthesis

To assess the effectiveness of our method at translating functional queries to SQL, we evaluated RDD2SQL on the 57 benchmarks which are both translatable to SQL and that are also supported by our tool. Fig. 9 gives an overview of the results when running RDD2SQL on these benchmarks with a time limit of 1 hour. The x-axis corresponds to the time (per benchmark) and the y-axis corresponds to the percentage of benchmarks solved in that time-limit. As we can see, almost all benchmarks (55 out of 57) are successfully translated within the hour time limit. Furthermore, most of them (88%) can be translated within 7 minutes. Table 4 gives more detailed statistics about the performance of RDD2SQL. As

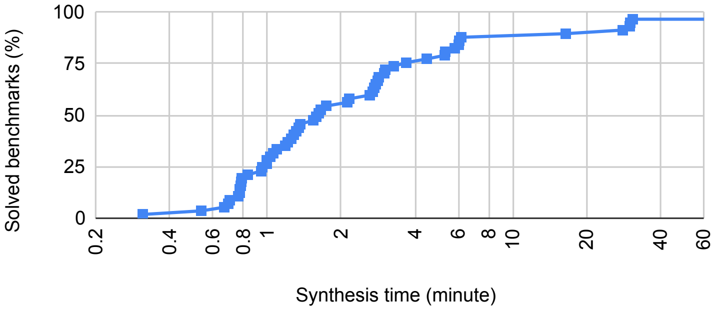


Fig. 9. Cumulative distribution of synthesis time

shown in this table, the median synthesis time of RDD2SQL is just over one and half minutes, and the average synthesis time is less than 5 minutes. Since RDD2SQL is meant to be used in an off-line manner, we believe that the time to transpile functional queries to SQL at compile time is worth the run-time benefits (as we show in Section 7.8).

Result for RQ2: RDD2SQL can successfully translate 96% of the supported RDD queries within an hour, and the vast majority (88%) in under 7 minutes.

7.4 Evaluating Column-Wise Decomposition

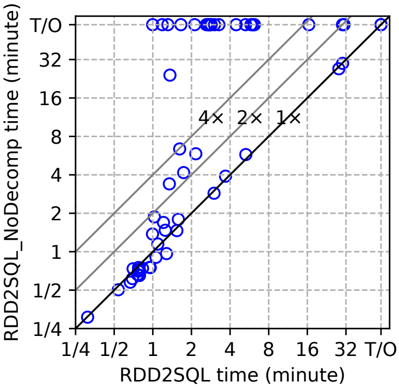


Fig. 10. Comparing synthesis time of RDD2SQL and RDD2SQL_NoDecomp

Since the key technical novelty of this work is the idea of column-wise decomposition, we perform an ablation study to evaluate the impact of this idea. Specifically, we compare RDD2SQL against an ablated version, called RDD2SQL_NoDecomp that does not perform column-wise decomposition. In other words, RDD2SQL_NoDecomp corresponds to our implementation of CEGIS with all optimizations described in Section 6 enabled.

Fig. 10 shows the results of this experiment as a scatter plot. Each circle in the figure represents a benchmark. The x-axis corresponds to the synthesis time of RDD2SQL and the y-axis corresponds to that of RDD2SQL_NoDecomp. The label 'T/O' indicates the time limit of 1 hour. As we can see from this scatter plot, RDD2SQL_NoDecomp translates fewer benchmarks than RDD2SQL within the time limit. Specifically, RDD2SQL_NoDecomp translates only 33/57 (58%) benchmarks while RDD2SQL translates 55/57 (96%) within an hour. If we consider a shorter time limit of 5 minutes, RDD2SQL_NoDecomp translates 27/57(47%) while RDD2SQL translates 44/57(77%). For some simple benchmarks, especially those that require less than one minute for both versions to solve, RDD2SQL_NoDecomp is slightly faster, because RDD2SQL needs to check the correctness of all N column queries while RDD2SQL_NoDecomp only needs to check one time for the complete query. This overhead is cancelled out by the benefits of column-wise decomposition for more complicated queries.

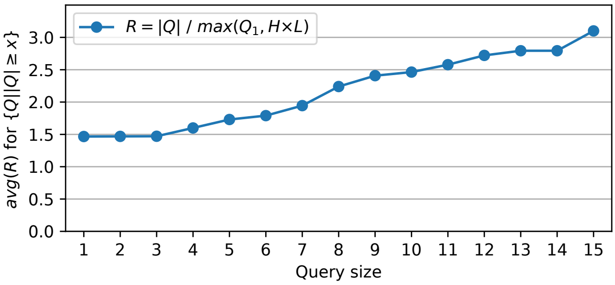


Fig. 11. Evaluating asymptotic complexity with and without column-wise decomposition. The x-axis shows query size s and the y-axis shows the average ratio $R = |Q|/max(|Q_1|, H \times L)$ for queries of at least size s.

Understanding asymptotic complexity. To gain further intuition about the scalability differences between RDD2SQL and RDD2SQL_NoDecomp, we also empirically analyze the parameters that affect the time complexity of RDD2SQL and its ablated version without decomposition. Recall from Section 5.4 that the complexity of CEGIS depends on the size $|Q|$ of the target query. In contrast, the complexity of our approach depends on (1) the number of backtracking steps (N) in Algorithm 3, (2) the size of the first column query ($|Q_1|$), and (3) $H \times L$, the number H of holes multiplied by the maximum AST size over all hole instantiations. In the remainder of this section, we look at the values of these parameters in our benchmark set in more detail.

First, we find that N is always 1 for all of our benchmarks, so there is effectively no backtracking in the synthesis algorithm in practice. In particular, it turns out that the query sketch extracted from the first column query is sufficient to synthesize all remaining column queries. This result indicates that our technique for extracting a query sketch from the first column query is extremely effective and justifies the assumption behind Equation 3 from Section 5.4.

Next, Fig. 11 shows the ratio $R = |Q|/max(|Q_1|, H \times L)$ as we vary query size. In particular, the x-axis shows the size s of the query, and the y-axis shows the average R value for all queries with *at least* size s. To interpret this plot, recall that a ratio R that is greater than 1 corresponds to an exponential improvement in terms of time complexity. As we can see from Fig. 11, the average of R for all queries is 1.5. More importantly, R increases as the target query size grows, exceeding 3 for queries of size 15. Hence, Fig. 11 provides further evidence about the effectiveness of column-wise decomposition in improving the scalability of our proposed approach to larger target queries.

Result for RQ3: Our column-wise decomposition idea has a big impact on the effectiveness of RDD-to-SQL translation in practice. Without this compositional approach, 42% of the benchmarks cannot be solved within the time limit.

7.5 Evaluating Optimizations

In this section, we describe the results of another ablation study to evaluate the impact of the optimizations described in Section 6. In particular, we consider the following ablations:

- RDD2SQL_NoSC: This is the variant of RDD2SQL that does not utilize the source program to augment the grammar with new productions. In other words, this variant does not perform the optimization described in Section 6.1.
- RDD2SQL_NoDfg: This is the variant of RDD2SQL that does not perform the optimization described in Section 6.2 (i.e., DFG-based decomposition adapted from prior work [Zhang et al. 2021]).

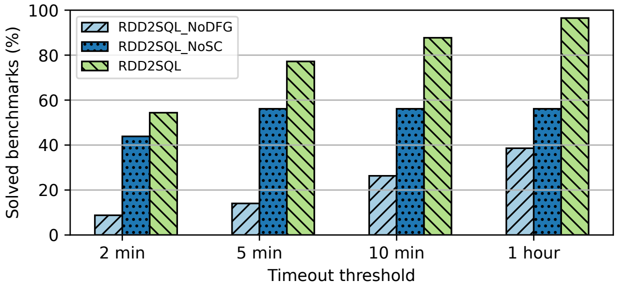


Fig. 12. Comparing complete and ablated versions

The results of this ablation study are presented in Fig. 12 as a bar graph. The x-axis shows the time limit, and the y-axis shows the percentage of benchmarks solved within that time limit for RDD2SQL and the two ablations. As we can see from this figure, both ablations solve fewer benchmarks compared to RDD2SQL, but the impact of the DFG-based optimization from Section 6.2 is even more pronounced. Furthermore, the benefit of the grammar augmentation optimization from Section 6.1 is more substantial for more complicated target queries.

Result for RQ4: Both grammar augmentation and DFG-based decomposition are important for the effectiveness of RDD-to-SQL translation. Without both of these optimizations enabled, nearly half of the benchmarks cannot be solved within the 1 hour time limit.

7.6 Comparison against CLIS

While there is no prior work that addresses the same problem that we tackle in this paper, the closest baseline is CLIS [Zhang et al. 2021]. However, as mentioned earlier, CLIS can only handle UDFs embedded inside SQL queries and cannot translate functional queries containing higher-order combinators. Thus, in order to use CLIS as a baseline for comparison against RDD2SQL, we need to extend it to handle higher-order combinators, as *all* of our benchmarks contain them. To that end, we implemented the following two extensions of CLIS:

- CLIS-Ext1: This extension performs enumerative search over higher-order combinators and invokes CLIS for all remaining parts of the query.
- CLIS-Ext2: As mentioned in Section 6.2, the lazy inductive synthesis idea of CLIS can be extended further to perform API-level decomposition. We thus applied a good-faith extension of the CLIS approach to high-order combinators. This extension of CLIS essentially is the same as RDD2SQL_NoDecomp from Section 7.4

The results of this comparison are shown in Fig. 13. In the figure, the x-axis shows the number of benchmarks, and the y-axis indicates cumulative synthesis time. As we can see, both CLIS-Ext1 and CLIS-Ext2 solve significantly fewer benchmarks than RDD2SQL within the one-hour time limit. In particular, while RDD2SQL solves 55 benchmarks, CLIS-Ext1 solves only 14 benchmarks and CLIS-Ext2 solves only 33 benchmarks. Note that CLIS-Ext2 is slightly faster than RDD2SQL for simple problems, because RDD2SQL needs to check the correctness of all N column queries while CLIS-Ext2 only needs to check one time for the complete query.

Result for RQ5: Of the benchmarks solved by RDD2SQL, CLIS-Ext1 solves only about one quarter, and CLIS-Ext2 solves only 60%.

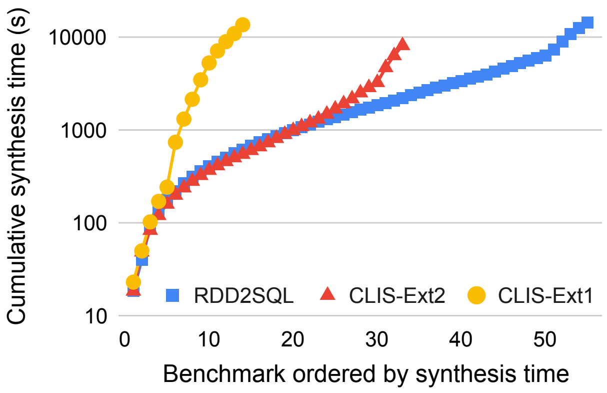


Fig. 13. Comparing RDD2SQL and CLIS

7.7 Failure Analysis and Limitations

In this section, we report some qualitative findings about both successful and failed synthesis tasks. To gain intuition about the shortcomings of RDD2SQL, we first perform a manual inspection of the benchmarks on which RDD2SQL fails and report our findings. We also manually inspect the SQL queries successfully synthesized by RDD2SQL and present some observations about their quality.

RDD2SQL fails to translate only two out of the 57 benchmarks within the one hour timeout. For one of these benchmarks, the root cause of failure is a very large UDF in the source program. In particular, during the program analysis phase (see Section 6.1), we invoke CLIS on each subexpression of this large UDF, which is both very expensive and also results in a very large grammar for RDD2SQL. From this single large UDF, our algorithm extracts 417 subexpressions and then augments them into the grammar. Note that disabling grammar augmentation also does not help with this benchmark: synthesis still fails because the UDF corresponds to a very large expression in the target query. For the second failed benchmark, the target SQL query is very large compared to the other queries, so RDD2SQL is unable to synthesize it within the one hour time limit.

Of the 55 correctly synthesized benchmarks, 33 of them contain RDD APIs that have no direct correspondence with SQL operators, such as `mapPartitions()`, `combineByKey()`, and `foldByKey()`, meaning RDD2SQL effectively addresses the semantic discrepancy between RDD and SQL demonstrated in Figure 2. Furthermore, we find that RDD2SQL is able to synthesize a number of complicated SQL queries from large functional queries: the largest synthesized query contains 12 nested subqueries and 115 nodes in its AST. On average, synthesized queres have 5 nested subqueries and 44 AST nodes.

To gain some intuition about the synthesis results, we also manually inspected the SQL queries returned by RDD2SQL. Note that, while our column-wise decomposition idea does not sacrifice completeness, it is *not* guaranteed to generate the simplest query. Our manual inspection indeed revealed some examples where the generated SQL query is slightly more complex than what a human would have written. In particular, we found two common types of redundancies: The first redundancy is the introduction of an unnecessary top-level Project operator; we provide an example of such a redundancy in Example 5.9. The second type of redundancy is caused by common sub-expressions that are not hoisted by RDD2SQL. To illustrate this redundancy, consider a query sketch $Q_s = \text{Project}(t, ?)$ with the following two instantiations for the column queries:

$$Q_1 = \text{Project}(t, split(c_1, \text{“,”})[0]);\;\; Q_2 = \text{Project}(t, split(c_1, \text{“,”})[1])$$

After merging these, we obtain the following synthesis result:

$$Q = \text{Project}(t, split(c_1, ``, ")[0], split(c_1, ``, ")[1])$$

This query contains two occurrences of $split(c_1, ``, ")$, and can be simplified to:

$$\text{Project}(\text{Project}(t, split(c_1, ``, ")\ AS\ x), x[0], x[1])$$

While RDD2SQL does not perform this type of simplification, we found that existing SQL engines can effectively optimize away both types of redundancies.

Result for RQ6: The two benchmarks that RDD2SQL failed to synthesize are very large, consisting of more than 72 AST nodes. Some of the queries successfully synthesized by RDD2SQL do contain redundancies, but these can be optimized away by existing SQL query engines.

7.8 Performance Benefits of SQL Translation

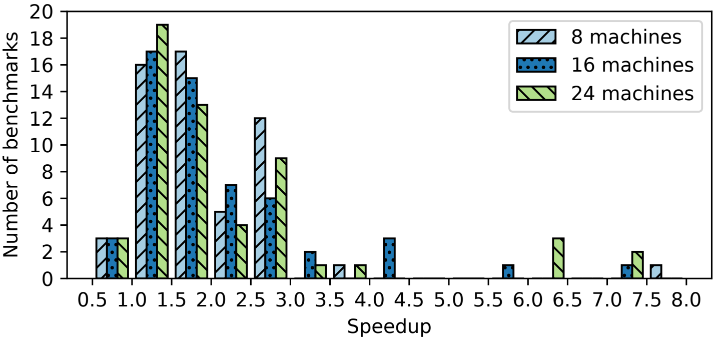


Fig. 14. Distribution of speedups. The baselines are the original functional queries processed on 8, 16, or 24 machines respectively.

A key motivation for translating functional queries to SQL is the potential to improve performance. Thus, we also conduct an experiment to evaluate the performance benefit offered by translating functional queries to SQL. To this end, we compare the runtimes of the original Spark RDD queries and their translated SQL version for the 55 benchmarks that RDD2SQL was able to transpile.

To perform this evaluation, we run each source and target query on Spark 3.2.0 clusters of 8, 16, and 24 machines with AMD Opteron Processor 6128 CPUs. We use randomly sampled in-memory data as query inputs and run each query 10 times for each input. The same input data set is used for both source and target queries. Fig. 14 shows the average speedup achieved by the SQL query as compared to the original functional version. In particular, the x-axis shows the magnitude of speedup (calculated as average runtime of source query divided by average runtime of SQL) and the y-axis shows the number of benchmarks which achieve that speedup. Across all benchmarks and all cluster sizes, all but 3 SQL translations are as fast or faster than their spark equivalents. As we can see, most programs achieve a 1.5 to 3× speedup.

Table 5 gives some additional statistics about the comparison. Across all cluster sizes, the SQL query is over 7× faster than its Spark equivalent in the best case. On average, a SQL query was approximately 2× faster, with the average speedup increasing as more machines are added to the cluster. For three benchmarks, the SQL query is slower than its original functional version (1.6×

Table 5. Summary of speedups

	# faster	Max	# slower	Min	Average
8 machines	52	7.57	3	0.74	1.92
16 machines	52	7.04	3	0.64	2.07
24 machines	52	7.43	3	0.63	2.23

slower in the worst case). On manual inspection, we found these three benchmarks contain string manipulation functions that are inefficiently executed by the Spark SQL engine.

Result for RQ7: Translating Spark RDD programs to SQL using RDD2SQL conferred an average speedup of 2× and a maximum speedup of over 7× across a variety of cluster sizes and randomly sampled inputs.

8 RELATED WORK

Translating other languages to SQL. To the best of our knowledge, our work is the first to focus on translating functional big data queries to SQL. There are some prior papers that translate from various languages into SQL. The most closely related is CLIS [Zhang et al. 2021], which converts UDFs used in SQL queries to pure SQL expressions. The DFG-based decomposition in Section 6.1 was inspired by CLIS; however, in contrast to this work, CLIS can only translate UDFs to SQL expressions without any *relational operators*. Allowing relational operators significantly increases the complexity of the problem space, which motivates the invention of *column-wise* decomposition in this work.

Another related work in this space is QBS [Cheung et al. 2013], which introduces a technique for automatically identifying translatable fragments of Java ORM programs and synthesizing their SQL equivalents. It depends on an intermediate representation of the source program tailored for ORM applications, not applicable to our domain. Similarly, DBridge [Emani et al. 2017] also synthesizes SQL from Java ORM programs, and SQLgen [Noor and Fegaras 2020] translates array loops to SQL. Both approaches rely on a hand-crafted set of translation rules which handle only a predefined set of commonly occurring coding patterns.

Compositional Program Synthesis. We are not the first to propose compositional program synthesis. Given a partial program with unknowns, a number of works [Feser et al. 2015; Osera and Zdancewic 2015; Polikarpova et al. 2016; Smith and Albarghouthi 2016] propose deductive techniques for decomposing a top-level synthesis specification into individual specifications for each unknown in the partial program which can then be synthesized independently. For example, Synquid [Polikarpova et al. 2016] and Bigλ [Smith and Albarghouthi 2016] infer type specifications for unknowns given a top-level type specification, while λ^2 [Feser et al. 2015] and Myth [Osera and Zdancewic 2015] infer input-output specifications for unknowns given a top-level inductive specification. In these approaches, results are merged together by enumerating completions of the partial program using the synthesized subprograms. In contrast, our approach uses dependence-free query sketches to ensure that synthesized subqueries can be *efficiently* merged using a simple, linear-time algorithm. Another line of work [Alur et al. 2015, 2017; Guria et al. 2021] decomposes the inductive synthesis task *per example* in the specification and merges the results by synthesizing suitable distinguishing predicates. In contrast to these techniques, our approach does not decompose by example but by column of the output and uses query sketches to guarantee efficient merging. Solis [Mariano et al. 2020] uses a decompositional technique for synthesizing loop summaries in smart contracts which synthesizes summaries for each variable in the loop independently and merges the results syntactically; however, their approach is particular to imperative loop summaries and is not complete.

Synthesis of SQL queries. This paper is related to a line of inductive synthesis work on learning SQL queries from input-output examples. For instance, Scythe [Wang et al. 2017], Query-By-Output [Tran et al. 2009], Trinity [Martins et al. 2019], SQLSynthesizer [Zhang and Sun 2013], SqlSol [Cheng 2019], SQUARES [Orvalho et al. 2020], EGS [Thakkar et al. 2021], and PATSQL [Takenouchi et al. 2021] all employ some form of inductive synthesis to learn SQL queries. While any of these tools could, in principle, be used as an inductive synthesis backend in our CEGIS implementation, we choose Trinity [Martins et al. 2019] due to its extensible nature. These prior efforts differ from our approach in that they neither perform synthesis from a reference implementation nor employ the idea of *column-wise* decomposition.

In a broader scope, program synthesis has been widely employed in many DB-related domains. For instance, SQLizer [Yaghmazadeh et al. 2017] synthesizes SQL from natural language; Foofah [Jin et al. 2017], Mitra [Yaghmazadeh et al. 2018], Hades [Yaghmazadeh et al. 2016], Dynamite [Wang et al. 2020], and Morpheus [Feng et al. 2017] all synthesize programs from examples in order to automate transformations between different structures and/or schemas. FlashFill [Singh and Gulwani 2012] and BlinkFill [Singh 2016] automatically fill spreadsheet columns by synthesizing string manipulation programs from examples. HYB [Raza and Gulwani 2020] utilizes program-by-example to synthesize programs for Web data extraction. Finally, [Singh et al. 2017] employs CEGIS to learn entity matching rules for detecting records that refer to the same object.

Optimizing Spark programs. The Spark [Zaharia et al. 2010] framework executes programs written in its functional RDD API [Zaharia et al. 2012] using a traditional volcano [Graefe and McKenna 1993] query execution model. Due to the extensive use of UDFs, further optimization of Spark RDD programs is hard. To the best of our knowledge, there is no other work on optimizing Spark RDD programs, although a number of efforts try to optimize UDF-rich queries in other big data frameworks. Notably, PeriSCOPE [Guo et al. 2012] and Niijima [Xu et al. 2019] optimize SCOPE [Chaiken et al. 2008] queries by eliminating, moving, and merging code in UDFs. PeriSCOPE reports an average speedup of 1.7× on 8 studied cases, and Niijima accelerates 21 queries by 1.24×; however, it is unclear whether these optimizations are effective for Spark. In contrast, our method achieves an average 2× speedup through *column-wise* decomposition. Stratosphere [Alexandrov et al. 2014] and Tupleware [Crotty et al. 2015] perform static analysis on UDFs (e.g., cost estimation) to enable query optimization such as operator reordering. Their techniques rely on new programming models and are not applicable to Spark. Another work [Sousa et al. 2014] avoids redundant computations in UDFs across multiple queries; in contrast, our work focuses on optimizing a single query.

Other efforts focus on accelerating Spark programs by improving runtime components, including distributed job scheduling [Sidhanta et al. 2016; Wang et al. 2019], communication [Nguyen et al. 2018], native code generation [Essertel et al. 2018; Navasca et al. 2019], and memory management [Maas et al. 2015; Nguyen et al. 2015; Shi et al. 2019]. These techniques have the potential to improve the performance of both Spark RDD programs (source program of RDD2SQL) and Spark SQL (target program of RDD2SQL). These approaches are orthogonal to ours and can be profitably combined with RDD2SQL to further improve performance.

9 CONCLUSION AND FUTURE WORK

Motivated by the performance advantages of SQL over functional big data queries, this work proposes the first method for transpiling functional queries to SQL. Based on the insight that SQL queries are column-wise decomposable, our method first finds a SQL query for each output column through program synthesis and then merges all column queries to a complete query. Our method utilizes the novel idea of *dependence-free query sketches (DFQS)* and automatically generates useful query sketches with this dependence-freedom property.

We implemented our method as a tool called RDD2SQL and tested it on real Spark RDD programs collected from Github. Our results show that RDD2SQL can translate 88% functional programs to SQL in under 7 minutes and 96% of them within one hour. The SQL queries that RDD2SQL produces are 2× faster than original functional queries on average, with a maximum speedup of 7×.

While our implementation targets the specific setting of translating functional Spark queries to SQL, the key contribution of this paper, namely the technique of *column-wise decomposition*, is applicable in any setting where the goal is to synthesize SQL queries (or, more broadly, programs that operate over tables). Hence, an interesting direction for future work is to apply this idea to a broader class of source and target languages.

Another interesting direction for future work is to perform cost-guided synthesis to guarantee that the synthesized queries are optimal or near-optimal. In this work, we do not consider the performance of the SQL queries, as we found that existing SQL query optimizers can effectively handle the types of queries that we generate.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation (NSF) under Grant No. CCF-1703487. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

REFERENCES

Maaz Bin Safeer Ahmad, Jonathan Ragan-Kelley, Alvin Cheung, and Shoaib Kamil. 2019. Automatically translating image processing libraries to halide. *ACM Transactions on Graphics (TOG)* 38, 6 (2019), 1–13. https://doi.org/10.1145/3355089.3356549

Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, et al. 2014. The stratosphere platform for big data analytics. *The VLDB Journal* 23, 6 (2014), 939–964. https://doi.org/10.1007/s00778-014-0357-y

Rajeev Alur, Pavol Černý, and Arjun Radhakrishna. 2015. Synthesis Through Unification. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 163–179. https://doi.org/10.1007/978-3-319-21668-3_10

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *Tools and Algorithms for the Construction and Analysis of Systems*, Axel Legay and Tiziana Margaria (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 319–336. https://doi.org/10.1007/978-3-319-21668-3_10

Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1383–1394. https://doi.org/10.1145/2723372.2742797

Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 221–230. https://doi.org/10.1145/3183713.3190662

Dirk Beyer, Andreas Holzer, Michael Tautschnig, and Helmut Veith. 2013. Information Reuse for Multi-goal Reachability Analyses. In *Programming Languages and Systems*, Matthias Felleisen and Philippa Gardner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 472–491. https://doi.org/10.1007/978-3-642-37036-6_26

Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 38, 4 (2015).

Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. 2008. Scope: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1265–1276. https://doi.org/10.14778/1454159.1454166

Lin Cheng. 2019. SqlSol: An accurate SQL Query Synthesizer. In *Formal Methods and Software Engineering*, Yamine Ait-Ameur and Shengchao Qin (Eds.). Springer International Publishing, Cham, 104–120. https://doi.org/10.1007/978-3-030-32409-4_7

Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing database-backed applications with query synthesis. *ACM SIGPLAN Notices* 48, 6 (2013), 3–14. https://doi.org/10.1145/2499370.2462180

Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004) (Lecture Notes in Computer Science, Vol. 2988)*, Kurt Jensen and Andreas Podelski (Eds.). Springer, 168–176. https://doi.org/10.1007/978-3-540-24730-2_15

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California) *(POPL '77)*. Association for Computing Machinery, New York, NY, USA, 238–252. https://doi.org/10.1145/512950.512973

Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur Çetintemel, and Stanley B Zdonik. 2015. Tupleware:" Big" Data, Big Analytics, Small Clusters.. In *CIDR*.

K. Venkatesh Emani, Tejas Deshpande, Karthik Ramachandra, and S. Sudarshan. 2017. DBridge: Translating Imperative Code to SQL. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1663–1666. https://doi.org/10.1145/3035918.3058747

Gregory Essertel, Ruby Tahboub, James Decker, Kevin Brown, Kunle Olukotun, and Tiark Rompf. 2018. Flare: Optimizing apache spark with native compilation for scale-up architectures and medium-size data. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 799–815.

Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. *ACM SIGPLAN Notices* 52, 6 (2017), 422–436. https://doi.org/10.1145/3140587.3062351

John K Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. *ACM SIGPLAN Notices* 50, 6 (2015), 229–239. https://doi.org/10.1145/2813885.2737977

G. Graefe and W.J. McKenna. 1993. The Volcano optimizer generator: extensibility and efficient search. In *Proceedings of IEEE 9th International Conference on Data Engineering*. 209–218. https://doi.org/10.1109/ICDE.1993.344061

Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119. https://doi.org/10.1561/2500000010

Zhenyu Guo, Xuepeng Fan, Rishan Chen, Jiaxing Zhang, Hucheng Zhou, Sean McDirmid, Chang Liu, Wei Lin, Jingren Zhou, and Lidong Zhou. 2012. Spotting code optimizations in data-parallel pipelines through periscope. In *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 121–133.

Sankha Narayan Guria, Jeffrey S. Foster, and David Van Horn. 2021. RbSyn: Type- and Effect-Guided Program Synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 344–358. https://doi.org/10.1145/3453483.3454048

Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. 2010. How Did You Specify Your Test Suite. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering* (Antwerp, Belgium) *(ASE '10)*. Association for Computing Machinery, New York, NY, USA, 407–416. https://doi.org/10.1145/1858996.1859084

Zhongjun Jin, Michael R. Anderson, Michael Cafarella, and H. V. Jagadish. 2017. Foofah: Transforming Data By Example. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 683–698. https://doi.org/10.1145/3035918.3064034

Martin Maas, Tim Harris, Krste Asanović, and John Kubiatowicz. 2015. Trash day: Coordinating garbage collection in distributed systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*.

Benjamin Mariano, Yanju Chen, Yu Feng, Shuvendu K Lahiri, and Isil Dillig. 2020. Demystifying Loops in Smart Contracts. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, New York, NY, USA, 262–274. https://doi.org/10.1145/3324884.3416626

Ruben Martins, Jia Chen, Yanju Chen, Yu Feng, and Isil Dillig. 2019. Trinity: An extensible synthesis framework for data science. *Proceedings of the VLDB Endowment* 12, 12 (2019), 1914–1917. https://doi.org/10.14778/3352063.3352098

Christian Navasca, Cheng Cai, Khanh Nguyen, Brian Demsky, Shan Lu, Miryung Kim, and Guoqing Harry Xu. 2019. Gerenuk: Thin Computation over Big Native Data Using Speculative Program Transformation. In *Proceedings of the 27th*

ACM Symposium on Operating Systems Principles (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 538–553. https://doi.org/10.1145/3341301.3359643

Khanh Nguyen, Lu Fang, Christian Navasca, Guoqing Xu, Brian Demsky, and Shan Lu. 2018. Skyway: Connecting managed heaps in distributed big data systems. *ACM SIGPLAN Notices* 53, 2 (2018), 56–69. https://doi.org/10.1145/3296957.3173200

Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, Jianfei Hu, and Guoqing Xu. 2015. Facade: A compiler and runtime for (almost) object-bounded big data applications. *ACM SIGARCH Computer Architecture News* 43, 1 (2015), 675–690. https://doi.org/10.1145/2786763.2694345

Md Hasanuzzaman Noor and Leonidas Fegaras. 2020. Translation of Array-Based Loops to Spark SQL. In *2020 IEEE International Conference on Big Data (Big Data)*. 469–476. https://doi.org/10.1109/BigData50022.2020.9378136

Pedro Orvalho, Miguel Terra-Neves, Miguel Ventura, Ruben Martins, and Vasco Manquinho. 2020. SQUARES: a SQL synthesizer using query reverse engineering. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2853–2856. https://doi.org/10.14778/3415478.3415492

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. *ACM SIGPLAN Notices* 50, 6 (2015), 619–630. https://doi.org/10.1145/2813885.2738007

Shankara Pailoor, Yuepeng Wang, Xinyu Wang, and Isil Dillig. 2021. Synthesizing Data Structure Refinements from Integrity Constraints. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 574–587. https://doi.org/10.1145/3453483.3454063

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices* 51, 6 (2016), 522–538. https://doi.org/10.1145/2980983.2908093

Mohammad Raza and Sumit Gulwani. 2020. Web Data Extraction Using Hybrid Program Synthesis: A Combination of Top-down and Bottom-up Inference. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1967–1978. https://doi.org/10.1145/3318464.3380608

Xuanhua Shi, Zhixiang Ke, Yongluan Zhou, Hai Jin, Lu Lu, Xiong Zhang, Ligang He, Zhenyu Hu, and Fei Wang. 2019. Deca: a garbage collection optimizer for in-memory data processing. *ACM Transactions on Computer Systems (TOCS)* 36, 1 (2019), 1–47. https://doi.org/10.1145/3310361

Subhajit Sidhanta, Wojciech Golab, and Supratik Mukhopadhyay. 2016. OptEx: A Deadline-Aware Cost Optimization Model for Spark. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. 193–202. https://doi.org/10.1109/CCGrid.2016.10

Rishabh Singh. 2016. Blinkfill: Semi-supervised programming by example for syntactic string transformations. *Proceedings of the VLDB Endowment* 9, 10 (2016), 816–827. https://doi.org/10.14778/2977797.2977807

Rishabh Singh and Sumit Gulwani. 2012. Learning Semantic String Transformations from Examples. *Proc. VLDB Endow.* 5, 8 (apr 2012), 740–751. https://doi.org/10.14778/2212351.2212356

Rohit Singh, Venkata Vamsikrishna Meduri, Ahmed Elmagarmid, Samuel Madden, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Armando Solar-Lezama, and Nan Tang. 2017. Synthesizing entity matching rules by examples. *Proceedings of the VLDB Endowment* 11, 2 (2017), 189–202. https://doi.org/10.14778/3149193.3149199

Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet Transactions: High-Level Programming for Line-Rate Switches. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 15–28. https://doi.org/10.1145/2934872.2934900

Calvin Smith and Aws Albarghouthi. 2016. MapReduce program synthesis. *Acm Sigplan Notices* 51, 6 (2016), 326–340.

Armando Solar-Lezama. 2008. *Program synthesis by sketching*. University of California, Berkeley.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. *SIGARCH Comput. Archit. News* 34, 5 (oct 2006), 404–415. https://doi.org/10.1145/1168919.1168907

Marcelo Sousa, Isil Dillig, Dimitrios Vytiniotis, Thomas Dillig, and Christos Gkantsidis. 2014. Consolidation of queries with user-defined functions. *ACM SIGPLAN Notices* 49, 6 (2014), 554–564. https://doi.org/10.1145/2666356.2594305

Keita Takenouchi, Takashi Ishio, Joji Okada, and Yuji Sakata. 2021. PATSQL: Efficient Synthesis of SQL Queries from Example Tables with Quick Inference of Projected Columns. *Proc. VLDB Endow.* 14, 11 (2021), 1937–1949. http://www.vldb.org/pvldb/vol14/p1937-takenouchi.pdf

Aalok Thakkar, Aaditya Naik, Nathaniel Sands, Rajeev Alur, Mayur Naik, and Mukund Raghothaman. 2021. Example-Guided Synthesis of Relational Queries. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1110–1125. https://doi.org/10.1145/3453483.3454098

Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. 2009. Query by Output. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (Providence, Rhode Island, USA) *(SIGMOD '09)*. Association for Computing Machinery, New York, NY, USA, 535–548. https://doi.org/10.1145/1559845.1559902

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

Kewen Wang, Mohammad Maifi Hasan Khan, Nhan Nguyen, and Swapna Gokhale. 2019. Design and implementation of an analytical framework for interference aware job scheduling on apache spark platform. *Cluster Computing* 22, 1 (2019), 2223–2237. https://doi.org/10.1007/s10586-017-1466-3

Yuepeng Wang, Rushi Shah, Abby Criswell, Rong Pan, and Isil Dillig. 2020. Data Migration using Datalog Program Synthesis. *Proc. VLDB Endow.* 13, 7 (2020), 1006–1019. https://doi.org/10.14778/3384345.3384350

Guoqing Harry Xu, Margus Veanes, Michael Barnett, Madan Musuvathi, Todd Mytkowicz, Ben Zorn, Huan He, and Haibo Lin. 2019. Niijima: Sound and Automated Computation Consolidation for Efficient Multilingual Data-Parallel Pipelines. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 306–321. https://doi.org/10.1145/3341301.3359649

Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. 2016. Synthesizing transformations on hierarchically structured data. *ACM SIGPLAN Notices* 51, 6 (2016), 508–521. https://doi.org/10.1145/2980983.2908088

Navid Yaghmazadeh, Xinyu Wang, and Isil Dillig. 2018. Automated migration of hierarchical data to relational tables using programming-by-example. *Proceedings of the VLDB Endowment* 11, 5 (2018), 580–593. https://doi.org/10.1145/3177732.3177735

Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: query synthesis from natural language. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–26. https://doi.org/10.1145/3133887

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*. 15–28.

Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.

Guoqiang Zhang, Yuanchao Xu, Xipeng Shen, and Işıl Dillig. 2021. UDF to SQL translation through compositional lazy inductive synthesis. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–26. https://doi.org/10.1145/3485489

Sai Zhang and Yuyin Sun. 2013. Automatically synthesizing SQL queries from input-output examples. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 224–234. https://doi.org/10.1109/ASE.2013.6693082

Received 2022-10-28; accepted 2023-02-25