UDF to SQL Translation through Compositional Lazy Inductive Synthesis

GUOQIANG ZHANG, North Carolina State University, United States
YUANCHAO XU, North Carolina State University, United States
XIPENG SHEN, North Carolina State University, United States
IŞIL DILLIG, University of Texas at Austin, United States

Many data processing systems allow SQL queries that call *user-defined functions (UDFs)* written in conventional programming languages. While such SQL extensions provide convenience and flexibility to users, queries involving UDFs are not as efficient as their pure SQL counterparts that invoke SQL's highly-optimized built-in functions. Motivated by this problem, we propose a new technique for translating SQL queries with UDFs to pure SQL expressions. Unlike prior work in this space, our method is not based on syntactic rewrite rules and can handle a much more general class of UDFs. At a high-level, our method is based on counterexample-guided inductive synthesis (CEGIS) but employs a novel compositional strategy that decomposes the synthesis task into simpler sub-problems. However, because there is no universal decomposition strategy that works for all UDFs, we propose a novel *lazy inductive synthesis* approach that generates a sequence of decompositions that correspond to increasingly harder inductive synthesis problems. Because most realistic UDF-to-SQL translation tasks are amenable to a fine-grained decomposition strategy, our lazy inductive synthesis method scales significantly better than traditional CEGIS.

We have implemented our proposed technique in a tool called CLIS for optimizing Spark SQL programs containing Scala UDFs. To evaluate CLIS, we manually study 100 randomly selected UDFs and find that 63 of them can be expressed in pure SQL. Our evaluation on these 63 UDFs shows that CLIS can automatically synthesize equivalent SQL expressions in 92% of the cases and that it can solve 2.4× more benchmarks compared to a baseline that does not use our compositional approach. We also show that CLIS yields an average speed-up of 3.5× for individual UDFs and 1.3× to 3.1× in terms of end-to-end application performance.

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, Yuanchao Xu, Xipeng Shen, and Işıl Dillig. 2021. UDF to SQL Translation through Compositional Lazy Inductive Synthesis. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 112 (October 2021), 26 pages. https://doi.org/10.1145/3485489

1 INTRODUCTION

As the most popular language for querying relational data, SQL provides a convenient and efficient mechanism for interacting with data stored in relational databases. Beyond providing the familiar relational algebra operators, SQL also allows programmers to perform computations on data

Authors' addresses: Guoqiang Zhang, North Carolina State University, Raleigh, United States, gzhang9@ncsu.edu; Yuanchao Xu, North Carolina State University, Raleigh, United States, yxu47@ncsu.edu; Xipeng Shen, North Carolina State University, Raleigh, United States, xshen5@ncsu.edu; Işıl Dillig, University of Texas at Austin, Austin, United States, isil@cs.utexas.edu.

2475-1421/2021/10-ART112
https://doi.org/10.1145/3485489

through SQL's highly optimized built-in functions as well as through *user-defined functions (UDFs)*. For example, Apache's Spark SQL allows programmers to write UDFs in a variety of conventional programming languages, such as Scala, Python, and Java. While such UDF-based extensions to SQL provide more flexibility to programmers as opposed to only using SQL's built-in functions, invoking UDFs from SQL queries typically incurs substantial performance overhead for the following reasons:

- First, since the SQL optimizer does not know the semantics of UDFs, they largely appear as black boxes to the query optimizer. As a result, the SQL compiler cannot perform any meaningful optimizations when the query involves UDFs.
- Second, executing a query with UDFs often requires expensive cross-language type conversion.
- Finally, SQL's built-in functions are highly-optimized, whereas user-defined functions tend to be much less efficient compared to their built-in counterparts.

Motivated by this problem, prior work [Gupta et al. 2020; Ramachandra and Park 2019; Ramachandra et al. 2017a] has proposed techniques for automatically translating UDFs to SQL expressions. For example, Froid [Ramachandra et al. 2017a] and BlackMagic [Ramachandra and Park 2019] translate *loop-free* user-defined functions to relational algebra expressions, and Aggify [Gupta et al. 2020] extends these techniques to deal with UDFs that contain so-called *cursor loops* (i.e., loops over query results). However, because these techniques are based on syntactic rewrite rules, their applicability is limited (e.g., *loop-free*, *cursor loops*); many practical user-defined functions are beyond their scope.

In this paper, we propose an alternative and more expressive approach based on *inductive program synthesis* for converting queries with UDFs to semantically equivalent pure SQL expressions over built-in functions. Unlike prior work in this space, our approach does not rely on rewrite rules over specific language constructs and can therefore convert a broader class of UDFs to SQL expressions. At its core, our approach is based on the standard *counterexample-guided inductive synthesis (CEGIS)* paradigm [Alur et al. 2013] and tries to learn the target SQL expression by generalizing from input-output examples provided by an equivalence checking engine as counterexamples. However, because pure inductive synthesis suffers from scalability problems, our method addresses this challenge by using a novel *compositional* approach.

At a high level, the main idea underlying our method is to break apart the given UDF into independent pieces, synthesize small SQL expressions for each piece using standard CEGIS, and then stitch these pieces together in a syntax-directed manner. In general, if we have a very fine-grained decomposition, the resulting inductive synthesis problems are much easier, allowing the technique to scale well to large UDFs. However, a key problem is that we do not know *a priori* what the right decomposition strategy is for an arbitrary UDF. For instance, consider a piece of code that is the sequential composition of three statements S_1, S_2, and S_3. If each statement S_i has an equivalent SQL expression, we can then potentially decompose our original synthesis problem into three independent, much simpler sub-problems, one for each S_i. On the other hand, even though the code $S_1; S_2; S_3$ may have an equivalent SQL expression, each S_i may not be individually expressible in pure SQL (e.g. node D of Figure 3b). Thus, it is unclear how to decompose a UDF to simpler synthesis sub-problems without placing syntactic restrictions on the class of UDFs that the method can handle.

Our method overcomes this challenge using a new paradigm that we dub *lazy inductive synthesis* that is illustrated schematically in Figure 1. At a high level, the key idea is to construct a sequence of (progressively coarser) decompositions of the original synthesis task based on the *dataflow graph (DFG)* representation of the UDF and solve each sub-problem using CEGIS. For a given decomposition, if every sub-problem can be solved using inductive synthesis, these solutions can be composed together to obtain a solution for the original task. Furthermore, because our method starts

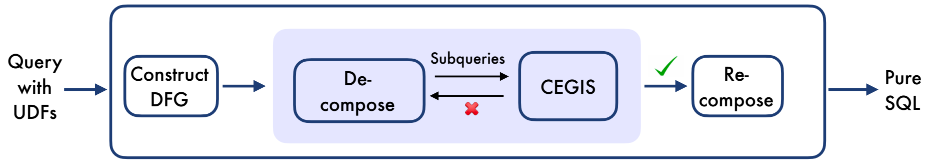


Fig. 1. Schematic overview of our approach

with the most fine-grained decomposition, the initial inductive synthesis tasks are easy to solve (i.e., they need to consider a small search space). However, due to limitations of the available SQL APIs, the inductive synthesizer may fail to find a solution for the sub-problems even though a solution exists for the original problem. Our method deals with this challenge by gradually coarsening the decomposition through merging of nodes in the data flow graph. We refer to this approach of gradually constructing harder and harder synthesis problems as *lazy inductive synthesis*, and we refer to our entire synthesis framework as *compositional lazy inductive synthesizer* (CLIS).

We evaluate CLIS in the context of UDF optimizations for Spark SQL programs. For a given Spark SQL program containing Scala UDFs, CLIS produces an equivalent Spark SQL program with some UDFs replaced with expressions composed of pure built-in SQL functions. Our evaluation on 63 real-world benchmarks collected from public repositories shows that CLIS can successfully synthesize equivalent SQL expressions for 92% of the cases, which is 2.8× more than prior solutions can solve within a given time limit. Furthermore, the optimizations performed by CLIS result in significant speedups in terms of run-time performance.

Overall, this work features two important novelties: (i) To the best of our knowledge, CLIS is the first general solution for transforming a broad class of UDFs into SQL; and (ii) it is based on a new synthesis paradigm that we call compositional lazy inductive synthesis. While UDF-to-SQL translation provides a useful application of this paradigm, we believe that our proposed approach may potentially be useful in other synthesis and code translation/optimization tasks as well.

In summary, this paper makes the following key contributions:

- We propose a new technique for translating SQL queries with UDFs to pure SQL expressions. Compared to prior work, our method can handle a broader class of user-defined functions.
- We show how to achieve a good trade-off between expressiveness and scalability using a technique that we dub *lazy inductive synthesis*. Our method uses a form of deductive reasoning to decompose the problem, but it solves each sub-problem using counterexample-guided inductive synthesis.
- We evaluate our implementation, CLIS, on real-world queries written in Spark SQL. Our evaluation shows that CLIS is effective at translating UDFs to SQL expressions and that it significantly improves query performance.

The rest of the paper is organized as follows: Section 2 gives a high-level overview of our method by running through a motivating example, and Section 3 presents our formal problem definition. Next, Section 4 introduces some background knowledge, and Section 5 presents our core synthesis algorithm. The following three sections (Sec. 6-8) describe our implementation and evaluation and discuss related work. Finally, we discuss limitations and conclude.

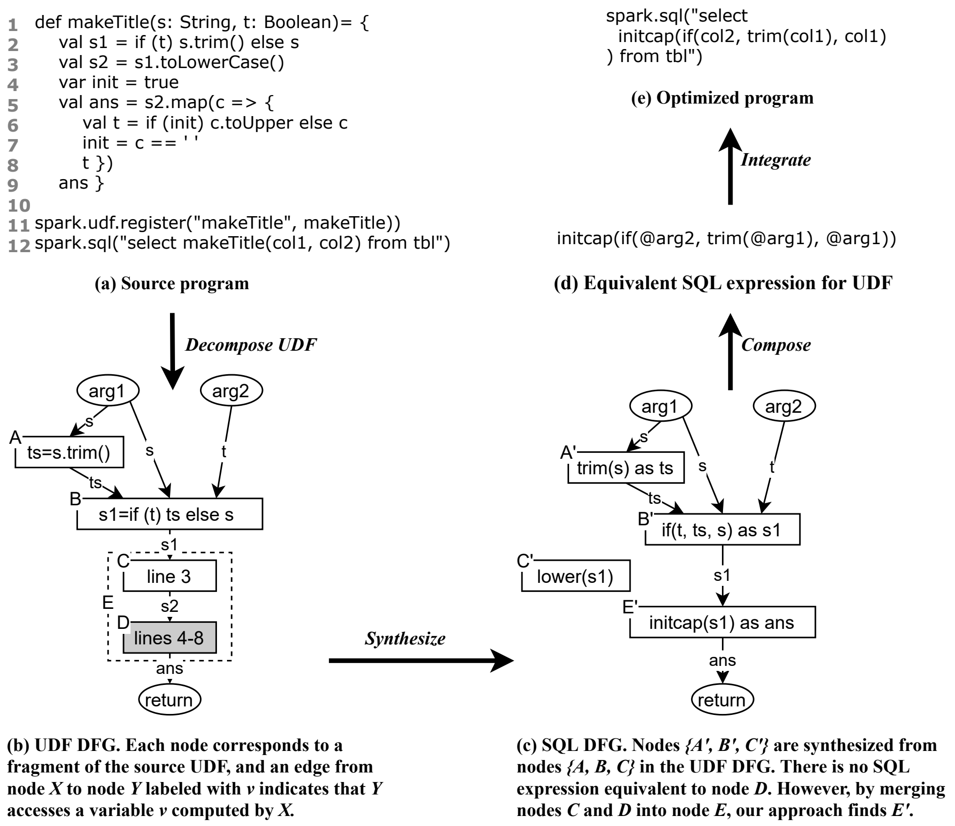
2 OVERVIEW

In this section, we give an overview of our method with the aid of the motivating example shown in Figure 2 that implements a SQL query in the Spark framework. Here, the query involves a user-defined function called `makeTitle` implemented in Scala. This function converts a string into

```
def makeTitle(s:String, trim:Boolean) = {
  val s1 = if(trim) s.trim() else s
  val s2 = s1.toLowerCase()
  var init = true
  val ans = s2.map(c => {
      val t = if (init) c.toUpper else c
      init = c == ‘ ‘
      t })
  ans }

spark.udf.register(“makeTitle”, makeTitle)
spark.sql(“select makeTitle(col1, col2) from tbl”)
```

Fig. 2. A Spark query containing a UDF



```
1  def makeTitle(s: String, t: Boolean)= {
2      val s1 = if (t) s.trim() else s
3      val s2 = s1.toLowerCase()
4      var init = true
5      val ans = s2.map(c => {
6          val t = if (init) c.toUpper else c
7          init = c == ' '
8          t })
9      ans }
10
11 spark.udf.register("makeTitle", makeTitle))
12 spark.sql("select makeTitle(col1, col2) from tbl")
```

(a) Source program

```
spark.sql("select
  initcap(if(col2, trim(col1), col1)
) from tbl")
```

(e) Optimized program

```
initcap(if(@arg2, trim(@arg1), @arg1))
```

(d) Equivalent SQL expression for UDF

(b) UDF DFG. Each node corresponds to a fragment of the source UDF, and an edge from node *X* to node *Y* labeled with *v* indicates that *Y* accesses a variable *v* computed by *X*.

(c) SQL DFG. Nodes *{A′, B′, C′}* are synthesized from nodes *{A, B, C}* in the UDF DFG. There is no SQL expression equivalent to node *D*. However, by merging nodes *C* and *D* into node *E*, our approach finds *E′*.

Fig. 3. Illustration of our approach on running example

a “title” format, in which the initial letter of each word is in upper case. The second argument of the function indicates if the input string needs to be trimmed. While this UDF is implemented correctly, the same functionality can actually be implemented using the following pure SQL expression:

```
spark.sql(“select initcap(if(col2, trim(col1), col1)) from tbl”)
```

which is 1.98 times faster compared to the original query containing the UDF.[1] We now explain how our technique automatically synthesizes the SQL query shown above given the original Spark program.

Our method starts by constructing the UDF's dataflow graph (DFG) (see Figure 3b). Here, each node corresponds to a fragment of the source UDF, and an edge from node X to node Y labeled with v indicates that Y accesses a variable v computed by X. Initially, our algorithm tries to synthesize a separate SQL statement for each DFG node by invoking a CEGIS-based synthesizer. In this case, it can find equivalent SQL expressions for nodes (A), (B), and (C); the result is shown in Figure 3c as nodes (A'), (B'), and (C'). However, synthesis fails for node (D) as there is no built-in SQL function that captures the computation performed in lines 4-8.

In the next iteration, our technique tries a less aggressive decomposition by merging nodes (C) and (D) in the original data-flow graph into a single node (E) (see Figure 3b). Note that the new DFG corresponds to a less fine-grained decomposition in that node (E) represents a bigger code fragment, so the CEGIS-based synthesizer needs to consider a larger search space. However, despite this larger search space, the target SQL expression is quite small, and the synthesizer returns the SQL expression `initcap(s1) as ans` as a solution to this sub-problem.

At this point, the algorithm has found a decomposition where each component is individually synthesizable. As a last step, we compose together each of the synthesis sub-results into the SQL expression of Figure 3d with the help of the DFG from Figure 3c. After eliminating intermediate variables and integrating this expression into the original query, we finally obtain the desired result.

Based on this example, we highlight the following salient features of our approach:

- Even though the final SQL query is quite large, the result of each synthesis sub-problem is small. Thus, decomposition allows our technique to scale to search spaces that cannot be traditionally handled by inductive synthesizers.
- Both the initial decomposition and its subsequent coarsening are guided by the dataflow graph, and the DFG representation is crucial for composing together the results of each subproblem.
- In contrast to prior work [Ramachandra et al. 2017a], our method can utilize built-in SQL functions such as `trim` and `initCap`, and it can also handle higher-order combinators like `map`.

3 PROBLEM DEFINITION

In this section, we formally define the synthesis problem addressed in the remainder of the paper. At a high level, our goal is to convert a SQL program containing user-defined functions (UDFs) to one without any UDFs.

As shown in Figure 4, our source language consists of a set of UDF definitions written in Scala, followed by a SQL query. [2] A UDF can contain arbitrary Scala expressions, including but not limited to loops and conditional statements. A query Q is a standard SQL query which may refer to temporary tables (views) defined using the `with` syntax in the grammar. Both projection operations Π and expressions E can involve built-in SQL functions/operators f as well as user-defined functions. In the grammar, we use the notation $\mathsf{UDF_s}$ to represent a UDF with a scalar return value and $\mathsf{UDF_m}$ to denote a UDF that returns multiple values (i.e., tuple).

Definition 1. **(UDF elimination problem.)** *Given a program P conforming to the grammar from Figure 4, our problem is to find a SQL query Q* without *any user-defined functions such that P is equivalent to Q (denoted $P \simeq Q$) — i.e., given any input data, P and Q produce the same output.*

[1]Tested on randomly generated 10 million rows on a Linux machine with an Intel Core i5-4570 CPU (four 3.2GHz cores).
[2]Because our implementation targets the Apache Spark framework, we consider UDFs written in Scala; however, our technique can, in principle, be applied to UDFs written in other programming languages as well.

$$\begin{array}{lll}
\text{Program } P & \longrightarrow & D_1; ...D_n; Q \\
\text{UDF Defn. } D & \longrightarrow & \texttt{def f}(a_1, ..., a_n) = \langle \mathit{ScalaExpr} \rangle \\
\text{SQL Query } Q & \longrightarrow & \sigma \mid \texttt{with } V_1; \ldots; V_n\ \sigma \\
\text{View } V & \longrightarrow & \text{TblName as } (\sigma) \\
\text{Select } \sigma & \longrightarrow & \texttt{select } \Pi \texttt{ from } \mathrm{T}_1, ..., \mathrm{T}_\mathrm{n}\ \ [\texttt{where } E] \\
 & & [\texttt{group by } \Pi]\ \ [\texttt{order by } \Pi] \\
\text{Projection } \Pi & \longrightarrow & \mathrm{UDF}_\mathrm{m}(C_1, ..., C_n) \mid C_1, ..., C_n \\
\text{Column } C & \longrightarrow & E \mid E \text{ as ColName} \\
\text{Expression } E & \longrightarrow & \text{Const} \mid \text{T.ColName} \mid F(E_1, ..., E_n) \\
\text{Function } F & \longrightarrow & \mathrm{UDF}_\mathrm{s} \mid f \in \text{BuiltInFuncs}
\end{array}$$

Fig. 4. The grammar of our source language.

In this paper, we solve this problem by translating each UDF definition D to an equivalent projection Π without UDFs and then inlining the call to this UDF by its equivalent projection. Thus, we can alternatively state our synthesis problem as the following UDF-to-SQL translation task:

Definition 2. **(UDF-to-SQL translation)** *Given a Scala function definition D with arguments $\overline{a}$, the UDF-to-SQL translation problem is to find a SQL projection expression* Π without *UDFs such that* $D \simeq \lambda\overline{a}.\ \Pi$.

4 PRELIMINARIES

In this section, we provide some background information on dataflow graphs and counterexample-guided inductive synthesis.

4.1 Background on Dataflow Graphs

In the rest of the paper, we represent UDFs as dataflow graphs defined as follows:

Definition 3. **(Dataflow graph (DFG))** *A dataflow graph for a function f is a directed acyclic graph $G = (N_A, N_S, N_R, E, L)$ where:*

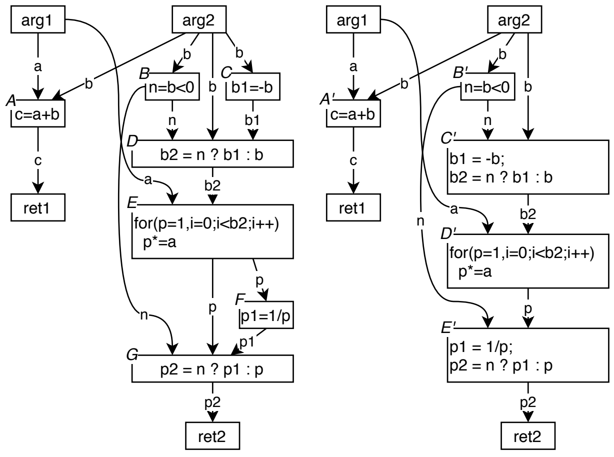
- *N_A is a list of nodes representing f's arguments in order.*
- *N_S is a set of nodes representing statements (i.e., code snippets) in f's body.*
- *N_R is a list of return nodes representing f's return values in order.*
- *$E \subseteq \{(n, n', v) \mid n \in N_A \cup N_S,\ n' \in N_S \cup N_R,\ v \in \mathsf{Vars}(f)\}$ is a set of directed edges (arcs).*
- *L is a mapping from nodes to labels. In particular, we have:*
  - *For $n \in N_A$, we have $L(n) \in \{\mathsf{arg}_k \mid 1 \leq k \leq \mathsf{Size}(N_A)\}$*
  - *For $n \in N_R$, we have $L(n) \in \{\mathsf{ret}_k \mid 1 \leq k \leq \mathsf{Size}(N_R)\}$*
  - *For $n \in N_S$, we have $L(n) \subseteq \mathsf{Stmts}(f)$*

Semantically, an edge (n, n', v) indicates that the computation performed in node n' depends on the computation from node n through variable v (i.e., n' reads from variable v that is modified in n). As expected, *argument nodes $n \in N_A$* do not have incoming edges, and *return nodes $n \in N_R$* do not have outgoing edges. The labeling function L assigns each argument (resp. return) node $n \in N_A$ (resp. N_R) to its sequential number in the list. Finally, for *statement nodes $n \in N_s$*, the label of n corresponds to some code fragment in f.

Example 4.1. Figure 5a shows a function called `addPower`, and Figure 5b shows a dataflow graph for this function. Since the UDF takes two arguments, the DFG has two argument nodes labeled

```
function addPower(a, b) {
    c = a + b;
    n = b < 0;
    if(n) b1 = -b;
    b2 = n ? b1 : b;
    for (p=1,i=0;i<b2;i++)
        p *= a;
    if(n) p1 = 1/p;
    p2 = n ? p1 : p;
    return c, p2;
}
```

(a) An example UDF



(b) Dataflow graph G (c) Dataflow graph G'

Fig. 5. An example UDF and its two possible DFGs

$arg1$ and $arg2$. It also has two return nodes labeled $ret1$ and $ret2$ because the UDF returns a pair. The statement node E represents the loop and has two outgoing edges (labeled with p) to nodes F and G: this indicates a data dependence through variable p between E and F as well as E and G. Observe that a given code fragment does not have a unique DFG representation. Figure 5c shows another valid DFG representation of the same addPower function.

Because a program has many possible DFG representations, we define a *refinement relation* between two DFGs as follows:

Definition 4. **(DFG refinement)** *Let* $G = (N_A, N_S, N_R, E, L)$ *and* $G' = (N_A, N'_S, N_R, E', L')$ *be two valid DFGs for the same function. We say that* G' *is a refinement of* G*, written* $G' \preceq G$*, iff:*

$$\forall n \in N_S.\ \big(\exists n_1 \in N'_S., \ldots, \exists n_k \in N'_S.\ \ L(n) \equiv L(n_1); \ldots ; L(n_k)\big)$$

where $L(n_1); \ldots ; L(n_k)$ *indicates the sequential composition of the statements labeling nodes* $n_1, \ldots, n_k$*.*

In other words, dataflow graph G' is a refinement of G if it “splits" one or more of the nodes in G into multiple nodes.

Example 4.2. For the two DFGs in Figure 5, we have $G \preceq G'$ because $L(A') = L(A)$, $L(B') = L(B)$, $L(D') = L(E)$, and

$$L(C') \ = \ L(C); L(D) \quad L(E') \ = \ L(F); L(G)$$

As we will see in Section 5.1, if $G \preceq G'$, our approach prefers G over G'.

4.2 Background on CEGIS

Since our synthesis technique is powered by *counterexample-guided inductive synthesis* (CEGIS) [Alur et al. 2013], we provide a brief overview of this paradigm. Given a specification ϕ, the goal of CEGIS is to generate a program that satisfies ϕ through a sequence of interactions between an *inductive synthesizer* and a *verifier* (see Figure 6). At any point in time, the inductive synthesizer has access to a set E of input-output examples (i.e., test cases), and it searches for a program that “passes” E. Once it finds such a program P, the verifier is used to check whether P actually satisfies

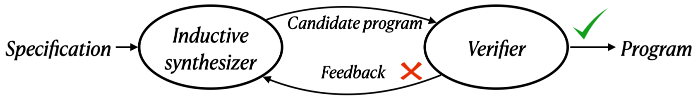


Fig. 6. Illustration of the CEGIS paradigm

ϕ. If so, P is returned as the solution; otherwise, the verifier provides feedback in the form of a new input-output example that P does not satisfy but that any valid solution *should* satisfy. This process continues until a valid solution is found. Note that, since the verification problem is, in general, undecidable, almost all instantiations of the CEGIS paradigm use a *bounded verifier* [Biere et al. 2003] instead of a full verifier.

In the context of our problem, the specification to the CEGIS problem is a reference implementation in the form of a Scala UDF, and the verifier checks equivalence between the Scala UDF and the proposed SQL query. If the verifier disproves equivalence, it also provides an input I on which the reference implementation (i.e., Scala UDF) and the proposed SQL query differ. In the next iteration, the inductive synthesizer needs to produce a SQL query that is consistent with $(I, f(I))$ where $f(I)$ is the output of the reference implementation on input I. Thus, the search problem solved by the inductive synthesizer becomes harder and harder over time as the verifier provides more and more counterexamples.

Since inductive synthesis is ultimately a search problem, almost all instantiations of the CEGIS paradigm impose an upper bound on the size of the search space considered by the inductive synthesizer. As standard, we assume that size of the synthesized program is proportional to the size of the reference implementation [Schkufza et al. 2013; Van Geffen et al. 2020; Wang et al. 2019]. That is, in our context, the larger the input UDF, the larger the search space that the inductive synthesizer needs to consider.

5 SYNTHESIS ALGORITHM

In this section, we describe our lazy inductive synthesis algorithm for translating user-defined functions to SQL expressions. We first give a high-level overview of the algorithm and then describe the details of the core procedures.

5.1 Top-Level Algorithm

Algorithm 1 shows the outer loop of our synthesis algorithm called UDF2SQL. This procedure takes as input a user-defined function F and returns a SQL expression E that is semantically equivalent to F (or null if no equivalent expression is found). To simplify presentation and avoid naming collisions, we assume that the UDF has been converted to static single assignment (SSA) form [Cytron et al. 1991]. At a high level, the UDF2SQL procedure maintains two data structures:

- A worklist W contains a set of decompositions of the UDF, where each decomposition is represented by a dataflow graph.
- A map Ω of partial results, mapping each code snippet S in function F to its corresponding SQL expression E. In other words, Ω is used to memoize partial synthesis results.

In the beginning of the algorithm (line 3), the worklist contains the initial DFG of the input function — i.e., output of ConstructDFG. While the details of the ConstructDFG procedure are not important for the synthesis algorithm, it is worth noting that our implementation constructs this DFG in a way that minimizes the size of the code snippets labeling each node. In other words, the initial DFG corresponds to the *most aggressive* decomposition of the UDF.

Algorithm 1 Top-level synthesis algorithm

```
1: function UDF2SQL($F$)
2:     Input: A user-defined function $F$
3:     $W \leftarrow \{\text{ConstructDFG}(F)\}$
4:     $\Omega \leftarrow \varnothing$
5:     while $W \neq \varnothing$ do
6:         $G \leftarrow \text{PickMostPromising}(W, \preceq)$
7:         $(E, n) \leftarrow \textsc{SynthFromDecomp}(G, \Omega)$
8:         if $E \neq \text{null}$ then return $E$
9:         $W \leftarrow W \cup \textsc{CoarsenDecomp}(G, n)$
10:    return null
```

Algorithm 2 Compositional synthesis from DFG

```
1: function SynthFromDecomp($G, \Omega$)
2:     Input: DFG $G = (N_A, N_S, N_R, E, L)$
3:     Input: Mapping $\Omega$ containing partial synthesis results
4:     for each $n \in N_S$ do
5:         $S \leftarrow L(n)$
6:         if $S \notin \text{Domain}(\Omega)$ then
7:             $\Omega[S] \leftarrow \textsc{Cegis}(S, \text{InLabels}(n), \text{OutLabels}(n))$
8:         if $\Omega[S] = \text{null}$ then return $(\text{null}, n)$
9:     $E \leftarrow \textsc{CodeGen}(G, \Omega)$
10:    return $(E, \text{null})$
```

After initialization, the algorithm enters a loop that terminates either when it finds an equivalent SQL expression or exhausts all possible decompositions of the initial DFG. Specifically, it dequeues the most promising decomposition from the worklist using the PickMostPromising function, which returns the most fine-grained decomposition in W. Since the search space of the inductive synthesizer is proportional to the size of the input program (recall Section 4.2), picking the most fine-grained decomposition translates into solving simpler inductive synthesis problems first. More formally, PickMostPromising returns a dataflow graph $G \in W$ with the following property:

$$\forall G' \in W.\, G \neq G' \Rightarrow G' \npreceq G$$

Next, given a decomposition strategy represented by G, UDF2SQL invokes the SynthFromDecomp procedure to synthesize an equivalent SQL expression using the chosen decomposition. If synthesis succeeds (e.g., returned expression E is not null), the algorithm terminates and returns E as a solution. Otherwise, SynthFromDecomp populates the map Ω with its partial synthesis results and produces a node $n \in G$ for which synthesis failed. Finally, the CoarsenDecomp procedure coarsens the current decomposition G by merging the failed node n with one of its neighboring nodes and produces a set of new decompositions that are added to the worklist at line 9. In the remainder of this section, we explain the SynthFromDecomp and CoarsenDecomp procedures in more detail.

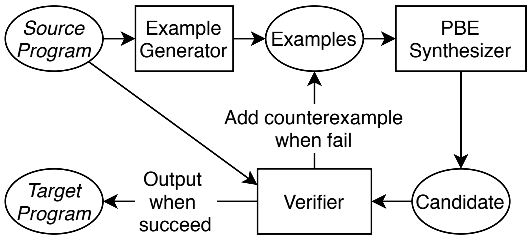


Fig. 7. Our instantiation of the CEGIS paradigm

5.2 Synthesis from Dataflow Graph

Our algorithm for synthesizing SQL code for a given decomposition (i.e., dataflow graph) is presented in Algorithm 2. The procedure SynthFromDecomp takes two inputs:

- A dataflow graph $G = (N_A, N_S, N_R, E, L)$ representing the current decomposition,
- Partial synthesis results Ω.

The idea behind SynthFromDecomp is as follows: For each statement node $n \in N_S$ of the current decomposition, it uses counterexample-guided inductive synthesis (line 7) to find an equivalent SQL expression for $L(n)$. If CEGIS is successful for every node $n \in N_S$, it then stitches together the resulting SQL expressions by calling CodeGen at line 9. However, if synthesis is unsuccessful for any DFG node n, the algorithm returns n as a "problem" node (line 8). It is worth noting that SynthFromDecomp uses partial results Ω to avoid re-synthesizing statements that have already been synthesized previously: In particular, if S is already in the domain of Ω (line 6), our algorithm does not invoke CEGIS and simply reuses the previous synthesis result.

Next, we explain how to instantiate the CEGIS paradigm in our setting as well as how to generate a full SQL query from individual synthesis results.

5.2.1 CEGIS Instantiation. Figure 7 illustrates how we instantiate the CEGIS paradigm in our context. Specifically, our CEGIS instantiation takes the following inputs:

- A code snippet S for which we want to synthesize an equivalent SQL expression
- The set of input variables $\overline{x}$ — these correspond to the label of incoming edges of the corresponding DFG node
- The set of output variables $\overline{y}$ (i.e., edge labels for outgoing edges of the DFG node)

Given these inputs, our CEGIS instantiation works as follows. First, we generate an initial set of inputs by choosing random values of $\overline{x}$ and executing S on these inputs to get corresponding values for outputs $\overline{y}$. We then use an off-the-shelf programming-by-example (PBE) engine [Martins et al. 2019] to find a SQL expression S' that satisfies these input-output examples. Next, we translate both the synthesized SQL expression S' and the original code snippet S to C code in a syntax-directed way and invoke an off-the-shelf model checker [Clarke et al. 2004] to test equivalence between S and S'. If they are not equivalent, we obtain an arbitrary counterexample from the model checker and invoke the PBE engine with this additional example.

Example 5.1. Consider the following code snippet in a DFG node:

```
var isPalindrome = true
for (i <- 0 until s.size)
   if (s(i) != s(s.size-i-1)) {
      isPalindrome = false
      break
```

Algorithm 3 Code Generation for Each DFG Node

```
1:  function CodeGen($n, \Omega$)
2:      Input: A DFG node $n$
3:      Input: Synthesis results $\Omega$
4:      $E \leftarrow \Omega(n)$
5:      for each $e \in$ InEdges($n$) do
6:          $(n', v) \leftarrow$ (Source($e$), Label($e$))
7:          if IsArg($n'$) then
8:              subExps[$v$] $\leftarrow L(n')$
9:          else
10:             parentRes $\leftarrow$ CodeGen($n', \Omega$)
11:             subExps[$v$] $\leftarrow$ parentRes[$v$]
12:     return map($E, \lambda t.$(Var($t$), replace(Def($t$), subExps)))
```

```
    }
```

This piece of code tests if a string is a palindrome. The input variable is `s` and the output variable is `isPalindrome`. We start by choosing a few random inputs, say "a" and "xy", and then execute the code on these inputs to get the following examples E:

$$E = \{\text{“}a\text{”} \mapsto true, \text{“}xy\text{”} \mapsto false\}$$

Given these examples, the PBE engine returns the candidate expression $\lambda x.x == \text{“}a\text{”}$. This is clearly wrong, and the verifier returns a counterexample, say "b". We then execute the reference implementation on this input and obtain the following examples:

$$E' = \{\text{“}a\text{”} \mapsto true, \text{“}xy\text{”} \mapsto false, \text{“}b\text{”} \mapsto true\}$$

Next, the PBE engine returns another incorrect expression $\lambda x.\ length(x) == 1$, so the verifier finds a new counter-example "aba". Our new examples now become:

$$E'' = \{\text{“}a\text{”} \mapsto true, \text{“}ab\text{”} \mapsto false, \text{“}b\text{”} \mapsto true, \text{“}aba\text{”} \mapsto true\}$$

Finally, in the third iteration, the PBE engine returns the expression $\lambda x.\ reverse(x) == x$ which can be proven equivalent to the reference implementation, so the CEGIS loop terminates.

5.2.2 Code Generation. Once we obtain the synthesis results for each node n in the dataflow graph, we generate the final SQL expression by traversing the DFG and substituting synthesis results for intermediate variables. In particular, Algorithm 3 describes our code generation procedure for each node in the DFG. The idea behind CodeGen is as follows: For a given node n, we first recursively generate code for all of its parents, which corresponds to obtaining SQL expressions for intermediate variables used in $\Omega(n)$ (lines 5–11). We then substitute these intermediate variables used in $\Omega(n)$ with their corresponding SQL expressions obtained through the recursive call (line 12). In more detail, consider a synthesis result such as (e_1 as $t_1, \ldots,$ e_n as t_n), where each e_i refers to intermediate columns defined in its parent block [3]. Line 12 of Algorithm 3 returns the mapping $[t_1 \mapsto e'_1, \ldots, t_n \mapsto e'_n]$ where each e'_i is obtained by substituting intermediate variables in e_i with a SQL expression over the arguments. The following example illustrates this code generation process.

Example 5.2. Consider the DFG and synthesis results shown in Figure 3(c). The CodeGen procedure generates SQL expressions for each node in the DFG as follows.

[3]For a SQL expression $e' \equiv e$ as t, we use the notation Var(e') to refer to t and Def(e') to refer to e.

Algorithm 4 Coarsen decomposition

```
1: function CoarsenDecomp($G, n$)
2:     Input: DFG $G = (N_A, N_S, N_R, E, L)$
3:     Input: $n \in N_S$, the node for which synthesis failed
4:     Output: A new set of dataflow graphs
5:     $S \leftarrow \emptyset$
6:     for each $n' \in N_S$ do
7:         if isNeighbor($n, n'$) then
8:             $G' \leftarrow$ MergeNodes($G, n, n'$)
9:             if isAcyclic($G'$) then
10:                $S \leftarrow S \cup G'$
11:    return $S$
```

- For node A', we generate the SQL expression trim(arg1).
- For node B', we generate the expression if(arg2, trim(arg1), arg1)) where the second argument is obtained through a recursive call.
- For node E', we generate the expression:

  initcap(if(arg2, trim(arg1), arg1))

  This is also the return value of CodeGen for the whole DFG.

5.3 Coarsening the Decomposition

Recall that our main synthesis procedure (Algorithm 1) performs lazy inductive synthesis by gradually coarsening the initial decomposition. In this section, we explain the CoarsenDecomp procedure, shown in Algorithm 4, for generating new decompositions when inductive synthesis fails. This procedure takes two inputs:

- A failing decomposition represented by dataflow graph G
- A DFG node n for which inductive synthesis fails

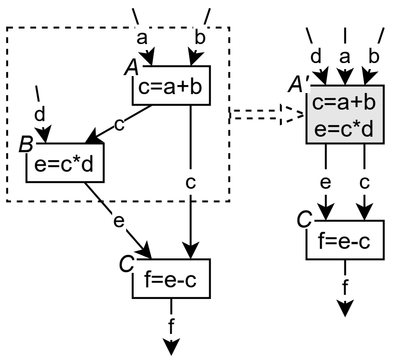
The idea behind CoarsenDecomp is quite simple: For each neighbor n' of n, we generate a new dataflow graph that is the same as G except that nodes n and n' have been merged. To merge nodes n and n', we follow the following steps:

- Generate a new node m with label $L(n); L(n')$
- The incoming (resp. outgoing) edges of m become the incoming (resp. outgoing) edges of both n and n' (except for those edges between n, n')
- G' includes m and all nodes of G except n, n'
- G' includes all edges of G except those involving n or n' as well as all new edges involving m

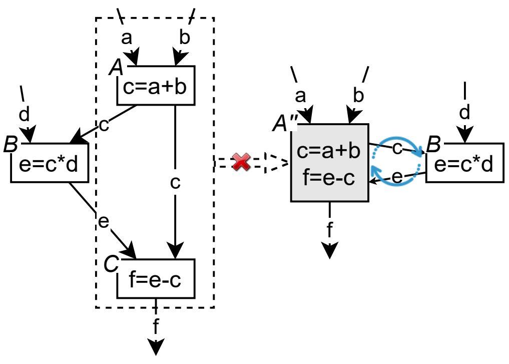
Note that line 9 of Algorithm 4 checks whether the resulting graph G' is acyclic before adding G' to the worklist because a valid dataflow graph is required to be acylic.

Example 5.3. Figure 8a shows a new DFG obtained by merging nodes A and B in the left DFG. Figure 8b shows the graph obtained by merging nodes A and C. However, this graph is rejected since it is cyclic and therefore an *invalid* DFG.

As mentioned earlier, coarsening the decomposition is useful after an unsuccessful synthesis attempt because the previous decomposition strategy may have been too aggressive. That is, while a statement S may not have an equivalent SQL expression, it is possible that a larger code fragment $S'; S$ does have an SQL equivalent, as illustrated in Section 2. Of course, the bigger the code fragment,



(a) A valid merge



(b) An invalid merge due to cyclic edges

Fig. 8. Merging two nodes

the larger the search space that needs to be considered by the CEGIS engine. Thus, coarsening the decomposition in a *demand-driven way* allows the overall synthesis algorithm to be more scalable in practice, as we discuss next.

5.4 Complexity Analysis

We first analyze the time complexity of the Cegis procedure in Algorithm 2. Let N be the number of statements in a DFG node, so the size of the target program is bounded by αN for some constant α. In the worst case, CEGIS explores the entire search space, so its worst-case time complexity is $O(F^{\alpha N})$, where F is the number of built-in SQL functions that the synthesizer supports.

Next, we analyze the time complexity of lazy inductive synthesis. In the best case, no coarsening is required, so each statement in the UDF is mapped to a DFG node. In this case, the time complexity of our technique is $O(SF^{\alpha})$. Since F and α are constants, this is $O(S)$, in which S is the number of statements in the UDF. Hence, the complexity of our technique can be exponentially better than standard CEGIS. In general, if there are C coarsening passes, and the largest coarsened node has N_C statements, then the time complexity is $O(CF^{\alpha N_C})$. In practice, we observe that the number of coarsening steps is small (approximately 2 on average in our evaluation); thus, our proposed lazy inductive synthesis technique works significantly better than standard CEGIS in practice.

6 IMPLEMENTATION AND OPTIMIZATIONS

We have implemented our proposed method in a tool called CLIS targeting the Spark framework [Zaharia et al. 2010]. We choose Spark as the test-bed due to its popularity and extensive use of UDFs. CLIS aims to accelerate Spark programs by replacing user-defined functions written in Scala with built-in SQL expressions whenever possible. Thus, CLIS can be viewed as a super-optimizer targeting Spark SQL programs.

In the rest of this section, we discuss some optimizations over the basic synthesis algorithm from Section 5.

6.1 Parallelization

The compositional synthesis technique of Algorithm 2 is embarrassingly parallel in that each DFG node can be translated independently. Thus, CLIS creates a new thread to run the CEGIS synthesizer for each DFG node. If there is no idle CPU, CLIS waits until one thread is finished before creating a new thread. If any thread exits with synthesis failure, then CLIS abandons the current DFG and coarsens its decomposition.

6.2 Type-Based Pruning

Our implementation performs one important optimization over the algorithm from Section 5. In our description of the UDF2SQL algorithm, the inductive synthesizer does not consider the input-output types of the code fragment to be synthesized. However, in practice, UDFs often involve intermediate types that are not supported by SQL. For instance, consider the following UDF:

```
def regexMatch(str: String, pattern: String): Boolean = {
  val regex: Regex = pattern.r
  return regex.matches(str)
}
```

Here, each of the two lines can be represented using separate nodes in the DFG. However, the data dependence between these nodes is through a variable of type `Regex`, which is not supported by SQL. Since it is impossible to find a SQL expression that returns a value of type `Regex`, we are guaranteed that the inductive synthesizer will fail on these DFG nodes. Our method uses this observation to rule out failing decompositions by eagerly merging DFG nodes that “return" a value whose type is not supported by SQL.

6.3 UDF-Level Transformations

CLIS performs a few transformations on the input UDF in order to make subsequent analysis easier.

Handling global variables. While UDFs can access global variables, SQL expressions cannot. But, in practice, many UDFs only access global variables without modifying them; so it is, in fact, possible to translate them into an equivalent SQL expression with some additional book-keeping. To enable the conversion of such UDFs to SQL expressions, we first append each global variable accessed by the UDF to its argument list. Then, if synthesis is successful, executing the synthesized SQL expression requires the table to contain a column whose value is equal to the value of the global variable. To facilitate this, we explicitly add such columns for global variables. For instance, consider the Spark program:

```
spark.udf.register("addx", (c:Int)=>c+x)) // x is global
spark.sql("select addx(c) from t")
```

Here, we first rewrite the original UDF into:

```
(c:Int,x:Int) => c+x
```

for which CLIS is able to find the equivalent SQL expression `@arg1 + @arg2`. To integrate the synthesized SQL expression into the query, we generate the following code:

```
spark.table("t").withColumn("x",lit(x)).registerTempTable("t1")
spark.sql("select c+x from t1")
```

The code snippet above first creates a temporary table `t1` that contains an additional column `x` that contains the value of global variable `x`. Then, the SQL query on the second line uses this temporary table `t1` instead of the original table `t`.

Flattening structs. While Spark allows columns of type struct, our synthesis engine does not support field accesses. To convert UDFs with field accesses to equivalent SQL expressions, we first flatten structs into multiple arguments. For instance, consider the following simple UDF:

```
(x:(Int,Int)) => x._1+x._2
```

In a pre-processing step, this UDF is rewritten to `(x1:Int,x2:Int) => x1+x2`. Then, once CLIS finds an equivalent SQL expression, we perform a post-processing step to map the flattened columns to the original fields.

6.4 PBE Back-End

Our implementation leverages the Trinity programming-by-example framework [Martins et al. 2019] for finding an expression that is consistent with a given set of input-output examples. To specialize Trinity for a given domain, one needs to specify (1) syntax of the DSL in which programs are to be synthesized, and (2) an optional abstract semantics of this DSL (which is used for pruning the search space). In our case, the DSL syntax corresponds to SQL projection expressions as defined in Figure 4, and we include a total of 73 built-in SQL functions. To help the programming-by-example engine, we also provide coarse-grained abstract semantics (i.e., logical specifications) for 33 of the 73 commonly used built-in functions. For instance, for the string concatenation function, our specification states that the length of the output is the sum of the lengths of the input strings. For constants, we restrict the search space of the synthesizer to a fixed set of commonly occurring values, such as 1, 0, and -1 for integers.

In addition, we employ an optimization called *incremental grammar generation* introduced in Casper [Ahmad and Cheung 2018]. Specifically, CLIS accelerates synthesis using two different grammars: The first (complete) grammar contains all 73 built-in SQL functions, but the second grammar contains only the 15 most-commonly used ones. For each DFG node, the synthesizer first tries to find an equivalent SQL expression using the partial grammar; and, if none is found, it runs the PBE engine using the complete grammar.

6.5 Checking Equivalence

We leverage the CBMC [Clarke et al. 2004] bounded model checker for checking equivalence between a Scala UDF and a SQL expression. However, since CBMC only works for programs written in C, we first convert both the source UDF and target SQL expression to C code. This is achieved by modeling Scala and SQL types and functions in C. For instance, a Scala `String` is modeled in C using a struct with two fields, namely `size` and `buffer`. Similarly, the '+' operation on two Scala `Strings` is modeled by a C function `String_concat()`. Our implementation of the Scala-to-C (resp. SQL-to-C translator) is a prototype and does not yet handle all Scala (resp. SQL) features.

7 EVALUATION

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

RQ1. How often do programmers write user-defined functions for computations that can be expressed using SQL expressions with existing built-in functions?
RQ2. How effective is CLIS at synthesizing SQL expressions for those UDFs that have a SQL equivalent?
RQ3. How much does decomposition help with synthesis? (i.e., how effective is the idea of *lazy inductive synthesis*?)

Table 1. Statistics about the benchmark set

Total # of UDFs	100
LOC per UDF	3 - 35, avg 5
AST size per UDF	9 - 235, avg 38
# UDFs with loops	78
# UDFs that have SQL equivalent	63

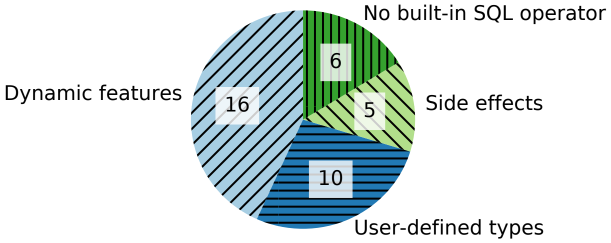


Fig. 9. Types of UDFs that do not have SQL equivalent

RQ4. How much of a performance improvement do we obtain by translating UDFs to equivalent SQL expressions?

RQ5. How much more expressive is CLIS compared to prior work in this space?

7.1 Benchmarks

To answer the research questions listed above, we collected a benchmark set consisting of 100 Spark UDFs using the following methodology. First, we downloaded all Github repositories that are (a) written in Scala, (b) use the Spark framework, and (c) contain at least one star. Next, we implemented a tool to extract from these projects all UDFs that conform to CLIS's requirements (e.g., no unmodeled Scala features like exceptions). Then we filtered out trivial UDFs that contain at most one operator. Finally, among the remaining UDFs, we randomly sampled 100 to use for our evaluation. Table 1 gives statistics about these UDFs in terms of their AST size, number of lines of Scala code, and whether they contain loops.

7.2 Manual Study

In order to answer RQ1, we manually inspected all 100 UDFs in our benchmark set and determined that 63 of them have an equivalent SQL expression. For the remaining 37 benchmarks, Figure 9 summarizes why these UDFs cannot be converted to an equivalent SQL expression. We identified four different reasons that prevent these UDFs from being expressible in SQL:

- **Dynamic features:** Some of these UDFs use dynamic Scala features such as reflection or virtual dispatch that cannot be expressed in SQL. These account for 43% of all UDFs that cannot be converted to SQL.
- **User-defined types:** 10 of the 37 UDFs use a custom Scala class defined by the user.
- **Side effects:** 5 of the 37 UDFs modify global variables.
- **No built-in SQL operator:** 6 of the 27 UDFs use Scala functions (e.g., matrix multiplication, AES encryption) for which there are no built-in SQL operators.

Table 2. Main synthesis results

Total convertible UDFs	63
Median synthesis time	9 seconds
Synthesized in 1 minutes	46 (73%)
Synthesized in 5 minutes	54 (86%)
Synthesized in one hour	58 (92%)
Decomposition coarsened	22 (35%)
Max top-level iterations	12
Avg top-level iterations	1.97

Result #1: 63% of the UDFs in our benchmark set have equivalent SQL expressions. The remaining ones cannot be expressed in SQL due to side effects, dynamic language features, custom types, and use of operators that do not have a built-in SQL equivalent.

7.3 Effectiveness of CLIS

In this section, we present the results of running CLIS on the 63 UDFs that have SQL equivalents. The experiments described here are performed on a Linux machine with an Intel Core i5-4570 CPU (four 3.2GHz cores) with a time limit set to one hour.

As summarized in Table 2, CLIS can successfully synthesize equivalent SQL expressions for 92% of these benchmarks (58 out of 63) within the given time limit of one hour. Furthermore, the median synthesis time across all benchmarks 9 seconds; about three quarters of benchmarks (73%) can be synthesized within one minute, and 86% of the benchmarks can be synthesized within 5 minutes. Also, as reported in Table 2, the initial decomposition is unsuccessful for 35% of the benchmarks, meaning that the decomposition needs to be coarsened in order for synthesis to be successful. The average number of iterations (i.e., decomposition coarsening steps) across all benchmarks is approximately 2; however, some benchmarks require up to 12 iterations.

Result #2: Among the 63 UDFs that have SQL equivalents, CLIS can successfully synthesize SQL expressions for 58 benchmarks (92%) within a time limit of one hour. Furthermore, synthesis terminates within 9 seconds for half of the benchmarks and within one minute for 73% of the UDFs.

7.4 Benefits of Decomposition and Laziness

In this section, we perform comparisons and ablation studies[4] to answer RQ3. In particular, we compare CLIS against an ablated version that does not perform lazy inductive synthesis. In particular, this baseline tries to find a SQL expression that is equivalent to the *entire* UDF rather than decomposing it into small snippets. In addition, we also compare the performance of CLIS with the synthesis technique used in Casper [Ahmad and Cheung 2018]. However, since Casper is designed for translating Java programs to MapReduce programs, we cannot directly use it in our setting. Thus, we re-implement Casper's key techniques for our problem, including incremental grammar generation and the CEGIS synthesizer backed by Sketch [Solar-Lezama et al. 2006].

The results of this comparison are included in Figure 10. Here, the x-axis shows the number of benchmarks (sorted according to synthesis time), and the y-axis shows cumulative running time in seconds. In this plot, the blue line (with dots) shows the results for CLIS running on four

[4]Ablation study is to compare a system against its variants generated by removing one or more parts of the system.

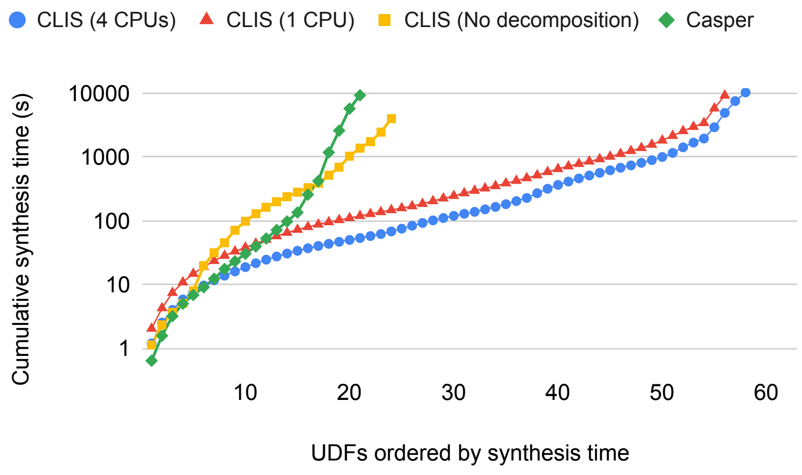


Fig. 10. Comparing cumulative synthesis time of different settings and techniques

CPU cores, the red line (with triangles) corresponds to the results for CLIS without parallelization, the yellow line (with squares) represents the ablated version, and the green line (with diamonds) corresponds to Casper. Overall, CLIS (running on 4 CPUs) solves 2.4× more benchmarks than the non-compositional (ablated) version and solves 2.8× as many benchmarks as Casper. The parallel version of CLIS is slightly faster than the non-parallel version and solves two additional benchmarks. Overall, these results demonstrate that the lazy inductive synthesis idea proposed in this paper is effective and that our DFG-based decomposition is highly beneficial in practice.

Result #3: The lazy inductive synthesis approach allows CLIS to solve 2.4× more benchmarks within the one hour time limit compared to an ablation that does not use this idea. Furthermore, CLIS solves 2.8× as many benchmarks compared to the Casper approach.

7.5 Performance Improvement

In this section, we evaluate RQ4 by measuring the performance improvement obtained by converting UDFs to SQL. We measure performance improvement both for individual UDFs and TPC-H workloads. We also consider two case studies for understanding end-to-end improvement in terms of overall application performance.

7.5.1 Individual UDFs. In this section, we compare the performance of the 58 original Spark UDFs that can be solved by CLIS against their SQL versions. To perform this comparison, we generate a synthetic dataset containing 10 million rows for each benchmark. The results of this experiment are summarized in Table 3. For all except two benchmarks, converting the UDF to a SQL expression results in a speed-up, with the geometric mean speed-up being 3.5×. Furthermore, for 71% of the benchmarks, we observe a speed-up exceeding 2×. Overall, these results confirm the hypothesis that converting UDFs to SQL expressions is quite beneficial in terms of performance.

7.5.2 TPC-H Workloads. In this section, we evaluate our method on TPC-H workloads [TPC 2005]. Since the original TPC-H queries do not contain UDFs, we rewrite the queries by replacing expressions with UDFs as done in previous work [Ramachandra et al. 2017b]. While the queries

Table 3. Improvement on individual UDFs

Total solved UDFs	58 (92% of total convertible UDFs)
# UDFs with speedups	56
Speedup factors	1.3-16.3×, geometric mean 3.5×
# UDFs of 1-2× speedups	15 (26%)
# UDFs of 2-3× speedups	15 (26%)
# UDFs of 3-5× speedups	10 (17%)
# UDFs of 5-10× speedups	11 (19%)
# UDFs of >10× speedups	5 (9%)
# UDF with slowdowns	2 (0.7× and 0.9 ×)

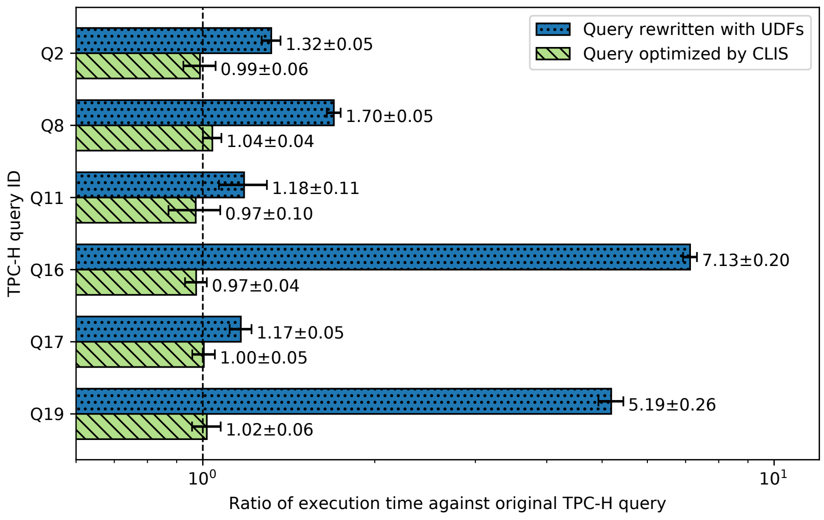


Fig. 11. Improvement of TPC-H queries with UDFs. Since the original TPC-H queries do not contain UDFs, we rewrite the queries by replacing expressions with UDFs as done in previous work [Ramachandra et al. 2017b]. Error bars are 95% confidence intervals.

are synthetic, the workloads of TPC-H represent real-world workloads. The rewritten queries are similar to those presented in the extended version of the Froid paper [Ramachandra et al. 2017b], except for that our versions have no subqueries inside UDFs because they are not supported by Spark SQL. We then compare performance between the original TPC-H queries and their rewritten versions on a 10GB dataset. In this experiment, six of the rewritten queries show significant (> 10%) performance degradation; thus, we use CLIS to optimize these six queries. Figure 11 shows the execution time of the three variants (original, rewritten with UDFs, and optimized by CLIS) for these queries. CLIS improves execution time of all six queries by $1.2 - 7.4\times$, with the geometric mean speed-up being 2.2× compared to the version containing UDFs. Furthermore, the queries optimized using CLIS execute as fast as the original TPC-H queries.

7.5.3 Case Study on Real Spark Applications. In this section, we evaluate the performance benefits of converting UDFs to SQL expressions in terms of end-to-end application running time. To perform this case study, we identify two relevant Spark applications using the following methodology. First, we collect a set of Github applications by searching for relevant keywords (namely, “SparkSession",

Table 4. Summary of applications

	WordCount	**CcyAggregator**
# Spark API calls	7	19
# API calls with UDF	2	11
# convertible UDFs	2	9
# UDFs CLIS can handle	1	9

Table 5. Four different test settings

	WordCount input	**CcyAggregator input**	**# Unique keys**	**Cluster size**
Setting A	463 MB	233 MB	10,000	8
Setting B	925 MB	466 MB	10,000	8
Setting C	463 MB	233 MB	1,000,000	8
Setting D	463 MB	233 MB	10,000	16

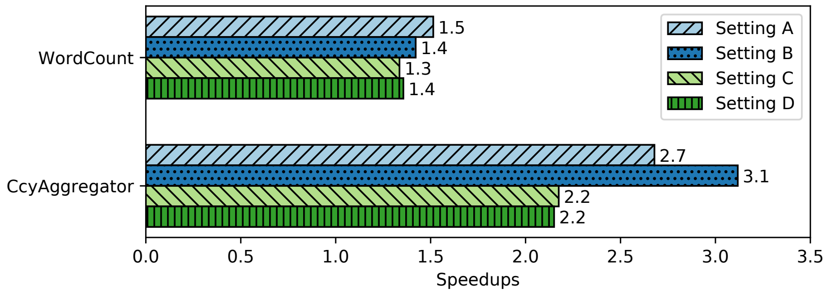


Fig. 12. End-to-end improvement for real applications

“read", “groupByKey”). Among the first 10 search results, 5 of them are Spark applications, and, based on our manual study, only two of those (WordCount and CcyAggregator) seem to consume a significant portion of their running time executing UDFs. In the remainder of this section, we describe our experience with using CLIS to optimize these applications.

Case study 1: WordCount. This application counts the number of occurrences of each word in a text file and contains a total of 7 Spark API calls. As shown in Table 4, two of these calls involve UDFs, and both of them have an equivalent SQL expression. However, converting one of these UDFs to SQL requires using a built-in function that CLIS currently does not support. CLIS is able to optimize the other UDF used in this application.

To measure the end-to-end performance improvement obtained by converting this UDF to SQL, we use the four different workloads/configurations summarized in Table 5. The workloads are input files with randomly generated contents. As shown in Figure 12, converting this single UDF to SQL improves end-to-end running time of the application by 30-50% depending on the setting.

Case study 2: CcyAggregator. We also evaluate the performance improvement on CcyAggregator, which is used for aggregating different currency rates. This application contains 19 Spark API calls,

Table 6. Comparison between CLIS and closely related work

Name*	Technique	Source program	Target program	Loops handled	# Applicable benchmarks†
CLIS	Program synthesis	Spark SQL with UDF	UDF-free Spark SQL	Arbitrary	58
Froid	Syntax-driven	T-SQL with UDF	UDF-free T-SQL	None	≤ 11
UDF Decorrelation	Syntax-driven	SQL with UDF	UDF-free SQL	Cursor loops	≤ 11
DBridge	Syntax-driven	Java ORM application	SQL	Cursor loops	≤ 11
QBS	Program synthesis	Java ORM application	SQL	Arbitrary	≤ 21
Casper	Program synthesis	Java program	MapReduce program	Arbitrary	21‡

* Citations: Froid [Ramachandra et al. 2017a], UDF Decorrelation [Simhadri et al. 2014], DBridge [Emani et al. 2017], QBS [Cheung et al. 2013], Casper [Ahmad and Cheung 2018]

† The number of collected benchmarks that can be solved within one hour. Upper bounds are conservatively estimated.

‡ The technique used by Casper is re-implemented to translate UDF to SQL.

11 of which contain UDFs. Among these eleven, nine have equivalent SQL expressions, and CLIS is able to convert all nine to SQL.

We evaluate the performance improvement of the optimized application using the same four workloads/configurations reported in Table 5. For this application, we observe much larger speed-ups, ranging from 2.2× to 3.1×, depending on which setting is used. These larger speed-ups compared to WordCount are expected, as CcyAggregator contains more UDFs that can be optimized.

Result #4: At individual UDF level, we observe an average 3.5× speed-up by converting UDFs to equivalent SQL expressions. At query level, we see 1.2 – 7.4× speed-up by using CLIS to convert TPC-H queries with UDFs to pure SQL. In terms of end-to-end application performance, CLIS enables significant improvements ranging from 1.3× to 3.1× for two representative Spark applications and four different test settings.

7.6 Expressiveness Compared to Prior Work

Table 6 lists prior techniques for translating code to SQL expressions and reports the maximum number of benchmarks that can be solved by each technique. As mentioned earlier, Froid [Ramachandra et al. 2017a] addresses a very similar problem setting but is only applicable to loop-free UDFs. Since our benchmark suite[5] contains exactly 11 loop-free benchmarks, an upper bound on the number of benchmarks Froid can handle is 11. The next two techniques from Table 6 can handle loops, but they are restricted to cursor loops that iterate over query results. Since Spark SQL prohibits cursor loops, these techniques cannot handle the types of loops that appear in our benchmark suite. Thus, 11 is also an upper bound on the number of benchmarks that can be handled by DBridge and the UDF decorrelation technique. Among the 11 loop-free benchmarks, CLIS solves 10 of them within 20 seconds, and times out for only one benchmark, in which there is a statement corresponding to a long SQL expression.

Next, we consider QBS [Cheung et al. 2013] and Casper [Ahmad and Cheung 2018], where the latter can be viewed as an improvement over the former. As mentioned earlier, we have re-implemented Casper's synthesis approach approach for our setting and showed that it can only solve 21 out of the 63 benchmarks within a 1 hour time limit. Thus, this number also serves as an upper bound on the number of benchmarks that can be solved by QBS and Casper. For only one out of the 21 benchmarks solved by Casper, CLIS times out because it takes too long time to find a good decomposition through coarsening.

[5]Here, we only focus on 63 UDFs that have SQL equivalents

Result #5: CLIS can handle more than 5.3× benchmarks compared to prior syntax-driven techniques, as they cannot deal with the loops found in Spark UDFs. Compared to prior work based on program synthesis, CLIS can solve at least 2.8× as many benchmarks within a one-hour time limit.

8 RELATED WORK

Among prior techniques for translating code to SQL, the most related one to ours is the Froid tool [Ramachandra et al. 2017a], which is now a feature of the Microsoft SQL Server [Ramachandra and Park 2019] for translating T-SQL UDFs to declarative SQL. Froid focuses on a loop-free subset of UDFs and generates equivalent SQL expressions using rewrite rules. In contrast, CLIS can handle a much wider class of UDFs using (lazy) inductive synthesis instead of term rewriting. However, since the Froid approach neither involves search nor requires an equivalence checker, it can be more efficient on the class of UDFs that it does handle. Simhadri et al. propose a query optimization technique [Simhadri et al. 2014] to decorrelate UDFs, but this technique can only handle so-called cursor loops that iterate over query results.

Another closely related tool is QBS [Cheung et al. 2013], which also uses the CEGIS paradigm to optimize ORM applications. The observation behind this work (as well as other papers [Emani et al. 2017, 2016; Wiedermann and Cook 2007; Wiedermann et al. 2008] that focus on the *query extraction* problem) is that ORM applications encourage developers to implement complex relational operations in application code, so the idea is to optimize such applications by generating more efficient SQL queries. An improvement over QBS is Casper, which augments CEGIS with incremental grammar generation. As described in Section 6, CLIS also uses the same optimization proposed in Casper; however, as demonstrated empirically in Section 7, that technique is not sufficient to solve many of our benchmarks. Finally, the more recent DBridge work [Emani et al. 2017, 2016] focuses on the same query extraction problem; however, similar to Froid, they solve this problem using a (more efficient, but less general) rewrite-based approach. Aggify [Gupta et al. 2020] converts cursor loops in UDFs or ORM applications to custom aggregates, which are also known as user-defined aggregate functions. Similar to Froid and DBridge, Aggify also uses a rewrite-based approach and inherits the same general trade-offs between generality and efficiency.

Optimizing code with UDFs. There have been several recent efforts to optimize UDFs used in data flow (MapReduce-style) programs. For instance, PeriSCOPE [Guo et al. 2012] tries to optimize UDFs by breaking UDF code into pieces that can be moved around in the execution plan. The Pact programming model [Hueske et al. 2013] reorders UDFs based on semantic information obtained from static code analysis. Sofa [Rheinländer et al. 2014] introduces UDF annotations to help optimization, and the Tupleware [Crotty et al. 2015] compiler architecture unifies workflow and UDF in the LLVM IR to perform optimization. Flare [Essertel et al. 2018] unifies UDFs and queries in a generative programming framework [Rompf and Odersky 2012] to enable native code generation of the whole query. Biltz [Schlaipfer et al. 2017] optimizes queries in an interestingly contrary direction to our work: Instead of converting UDF to SQL, it synthesizes UDFs to replace certain expensive subqueries. Finally, the technique proposed in [Sousa et al. 2014] optimizes dataflow programs by re-using computations across different UDFs that operate on the same data.

Compositional program synthesis. The technical approach used in this paper is related to other techniques that try to decompose a program synthesis problem into simpler sub-problems. For example, λ^2 [Feser et al. 2015] and FlashMeta [Polozov and Gulwani 2015] perform compositional programming-by-example by inferring input-output examples for sub-problems, and Synquid [Polikarpova et al. 2016] performs compositional synthesis from refinement types using so-called *round-trip type checking* and abduction. CPS [Raza et al. 2015] leverages natural language descriptions (in addition to examples) to decompose the synthesis problem into smaller sub-problems. Domino [Sivaraman et al. 2016], a DSL designed for programmable network switches, performs compilation using decomposition and program synthesis: the compiler first decomposes a loop-free Domino program to a pipeline and then employs a CEGIS synthesizer to generate code for each pipeline stage. Dexter [Ahmad et al. 2019] translates image processing libraries written in C++ to Halide (an image processing DSL) and exploits domain-specific knowledge to decompose the source program. Recent work by [Huang et al. 2020] proposes a compositional synthesis framework that leverages both deductive and inductive to solve SyGuS problems [Alur et al. 2013]. However, in contrast to these techniques, our decomposition is guided by a dataflow graph, and it is iteratively coarsened to perform inductive synthesis in a lazy manner.

9 LIMITATIONS

Despite enabling the conversion of many Spark UDFs to SQL expressions, CLIS has a number of limitations. First, CLIS relies on a verification oracle to check equivalence between a source UDF and a target SQL expression. Like all CEGIS-based synthesis tools, we implement this verification oracle using a bounded model checker, which only guarantees equivalence up to some bound (where the bound is the number of times loops are unrolled). Thus, in principle, it is possible for CLIS to produce a SQL expression that is not actually equivalent to the original UDF. However, we have confirmed through manual inspection that this is not the case for any of the benchmarks in our evaluation. This limitation of CLIS can be overcome by using heavier-weight equivalence checking tools such as SymDiff [Lahiri et al. 2012] or Coeus [Chen et al. 2019].

Second, CLIS does not support all Scala features that may be used in a UDF. Most of these limitations are inherited from shortcomings of other tools that CLIS leverages. For example, the CBMC tool used as our verification oracle has limited support for floating point arithmetic; thus UDFs containing floating point operations may fail to be verified.

Third, CLIS focuses on scalar Spark UDFs written in Scala. Beyond scalar UDFs that produce one output row from one input row, there are also other kinds of UDFs, such as custom aggregate functions that aggregate multiple rows into one row or table-valued functions that produce many rows from one row. While our current implementation does not support such UDFs, we see no fundamental problem in applying the lazy inductive synthesis method proposed in this paper to other types of UDFs, SQL engines, or other languages beyond Scala.

10 CONCLUSION

We have proposed a new technique, and its implementation in a tool called CLIS, for converting Spark UDFs (written in Scala) to equivalent SQL expressions. Our method is based on a new technical idea call *compositional lazy inductive synthesis*, where the goal is to reduce the search space of the inductive synthesizer through speculative decomposition and then lazily coarsen the decomposition if synthesis fails. Both the decomposition and gradual coarsening are driven by the dataflow graph representation of the UDF.

To evaluate CLIS, we collected 100 scalar Spark UDFs written in Scala and manually confirmed that 63 of them can be converted to SQL. Among these 63 UDFs, CLIS can successfully synthesize 58 of them within a time limit of one hour, and its median synthesis time is 9 seconds. We also performed an ablation study to evaluate the effectiveness of the proposed approach and showed that CLIS can solve 2.4× more benchmarks compared to an ablation that does not leverage DFG-guided decomposition. Finally, we demonstrated that CLIS yields significant speeds-ups (average speed-up of 3.5× per UDF and between $1.3 - 3.1\times$ in terms of end-to-end running time for two representative applications and several workloads).

Although our focus in this work is on UDF to SQL translation in this work, our proposed approach is more generally expected to be useful in settings where the source program can be decomposed into small translatable pieces. For example, another use case for this technique could be translating linear algebra or data cleansing code written in the basic Java language to expressions in specialized APIs targeting these domains. We leave it to future work to explore the applicability of the CLIS approach in other domains.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation (NSF) under Grants No. : CCF-1811865, CCF-1712067, CCF-1762299, and 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 and Alvin Cheung. 2018. Automatically Leveraging MapReduce Frameworks for Data-Intensive Applications. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1205–1220. https://doi.org/10.1145/3183713.3196891

Maaz Bin Safeer Ahmad, Jonathan Ragan-Kelley, Alvin Cheung, and Shoaib Kamil. 2019. Automatically Translating Image Processing Libraries to Halide. *ACM Trans. Graph.* 38, 6, Article 204 (Nov. 2019), 13 pages. https://doi.org/10.1145/3355089.3356549

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. 1–8. https://doi.org/10.1109/FMCAD.2013.6679385

Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. 2003. Bounded model checking. (2003).

Jia Chen, Jiayi Wei, Yu Feng, Osbert Bastani, and Isil Dillig. 2019. Relational Verification Using Reinforcement Learning. 3, OOPSLA, Article 141 (Oct. 2019), 30 pages. https://doi.org/10.1145/3360567

Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing Database-Backed Applications with Query Synthesis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/2491956.2462180

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

Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Cetintemel, and Stan Zdonik. 2015. An Architecture for Compiling UDF-Centric Workflows. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1466–1477. https://doi.org/10.14778/2824032.2824045

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (Oct. 1991), 451–490. https://doi.org/10.1145/115372.115320

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

K. Venkatesh Emani, Karthik Ramachandra, Subhro Bhattacharya, and S. Sudarshan. 2016. Extracting Equivalent SQL from Imperative Code in Database Applications. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1781–1796. https://doi.org/10.1145/2882903.2882926

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)*. USENIX Association, Carlsbad, CA, 799–815. https://www.usenix.org/conference/osdi18/presentation/essertel

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-Output Examples. *SIGPLAN Not.* 50, 6 (June 2015), 229–239. https://doi.org/10.1145/2813885.2737977

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 *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, USA, 121–133.

Surabhi Gupta, Sanket Purandare, and Karthik Ramachandra. 2020. Aggify: Lifting the Curse of Cursor Loops Using Custom Aggregates. 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, 559–573. https://doi.org/10.1145/3318464.3389736

Kangjing Huang, Xiaokang Qiu, Peiyuan Shen, and Yanjun Wang. 2020. Reconciling Enumerative and Deductive Program Synthesis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1159–1174. https://doi.org/10.1145/3385412.3386027

Fabian Hueske, Mathias Peters, Aljoscha Krettek, Matthias Ringwald, Kostas Tzoumas, Volker Markl, and Johann-Christoph Freytag. 2013. Peeking into the optimization of data flow programs with MapReduce-style UDFs. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 1292–1295. https://doi.org/10.1109/ICDE.2013.6544927

Shuvendu K Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *International Conference on Computer Aided Verification*. Springer, 712–717. https://doi.org/10.1007/978-3-642-31424-7_54

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.

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 522–538. https://doi.org/10.1145/2908080.2908093

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. *SIGPLAN Not.* 50, 10 (Oct. 2015), 107–126. https://doi.org/10.1145/2858965.2814310

Karthik Ramachandra and Kwanghyun Park. 2019. BlackMagic: Automatic Inlining of Scalar UDFs into SQL Queries with Froid. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 1810–1813. https://doi.org/10.14778/3352063.3352072

Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, César Galindo-Legaria, and Conor Cunningham. 2017a. Froid: Optimization of Imperative Programs in a Relational Database. *Proc. VLDB Endow.* 11, 4 (Dec. 2017), 432–444. https://doi.org/10.1145/3186728.3164140

Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, César A. Galindo-Legaria, and Conor Cunningham. 2017b. Optimization of Imperative Programs in a Relational Database. *CoRR* abs/1712.00498 (2017). arXiv:1712.00498 http://arxiv.org/abs/1712.00498

Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. 2015. Compositional Program Synthesis from Natural Language and Examples. In *Proceedings of the 24th International Conference on Artificial Intelligence* (Buenos Aires, Argentina) *(IJCAI'15)*. AAAI Press, 792–800.

Astrid Rheinländer, Martin Beckmann, Anja Kunkel, Arvid Heise, Thomas Stoltmann, and Ulf Leser. 2014. Versatile Optimization of UDF-Heavy Data Flows with Sofa. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 685–688. https://doi.org/10.1145/2588555.2594517

Tiark Rompf and Martin Odersky. 2012. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. *Commun. ACM* 55, 6 (June 2012), 121–130. https://doi.org/10.1145/2184319.2184345

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 305–316. https://doi.org/10.1145/2451116.2451150

Matthias Schlaipfer, Kaushik Rajan, Akash Lal, and Malavika Samak. 2017. Optimizing Big-Data Queries Using Program Synthesis. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 631–646. https://doi.org/10.1145/3132747.3132773

Varun Simhadri, Karthik Ramachandra, Arun Chaitanya, Ravindra Guravannavar, and S. Sudarshan. 2014. Decorrelation of user defined function invocations in queries. In *2014 IEEE 30th International Conference on Data Engineering*. 532–543. https://doi.org/10.1109/ICDE.2014.6816679

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

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. *SIGPLAN Not.* 49, 6 (June 2014), 554–564. https://doi.org/10.1145/2666356.2594305

TPC. 2005. TPC-H Benchmark Specification. http://www.tpc.org

Jacob Van Geffen, Luke Nelson, Isil Dillig, Xi Wang, and Emina Torlak. 2020. Synthesizing JIT Compilers for In-Kernel DSLs. In *Computer Aided Verification*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 564–586. https://doi.org/10.1007/978-3-030-53291-8_29

Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. 2019. Synthesizing Database Programs for Schema Refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 286–300. https://doi.org/10.1145/3314221.3314588

Ben Wiedermann and William R. Cook. 2007. Extracting Queries by Static Analysis of Transparent Persistence. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Nice, France) *(POPL '07)*. Association for Computing Machinery, New York, NY, USA, 199–210. https://doi.org/10.1145/1190216.1190248

Ben Wiedermann, Ali Ibrahim, and William R. Cook. 2008. Interprocedural Query Extraction for Transparent Persistence. *SIGPLAN Not.* 43, 10 (Oct. 2008), 19–36. https://doi.org/10.1145/1449955.1449767

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