ABSTRACT

ZHANG, GUOQIANG. Cross-Boundary Optimization of Data Analytics Programs. (Under the direction of Xipeng Shen).

Performance is critical to data analytics programs. To address the performance issue, people have developed many third-party components (TPCs), e.g., Numpy, Spark, and SQL engines, with various performance optimization techniques, including data parallelism, lazy evaluation, and query planning. Most of these traditional optimization techniques only focus on the TPC that they are embedded in, but do not consider the code outside the boundary of the TPC. This in-boundary view misses a lot of optimization opportunities across the boundary of TPC.

We thus propose cross-boundary optimization, a type of optimization that focuses on optimizing code around the TPC boundaries. The core challenge of cross-boundary optimization is to retain the unified semantics of the code on both sides of the boundary, which involves unifying different design patterns, different abstractions, and different languages.

In this thesis, we identify three types of TPC boundaries respectively between TPC and host code, between TPC and user-defined extension, and between different TPCs. For each type of boundary, we identify optimization opportunities around it and propose a novel method to optimize them. Experiments on these methods show that they can significantly accelerate data analytics programs.

Cross-Boundary Optimization of Data Analytics Programs

by
Guoqiang Zhang

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina
2022

APPROVED BY:

Isil Dillig External Member	Xu Liu
Guoliang Jin	Huiyang Zhou

Xipeng Shen
Chair of Advisory Committee

DEDICATION

To Jin.

ACKNOWLEDGEMENTS

I am very grateful to many people who have backed my dissertation research. If there must be one most important person, it is my wife, Jin. Without her understanding and support, I would not be pursuing a graduate degree.

Much of my academic success can be attributed to two professors: Dr. Xipeng Shen and Dr. Isil Dillig. As my advisor, Dr. Shen is the person who always motivates, enlightens, and encourages me. And Dr. Dillig helped me a lot throughout my research, especially on my last two projects.

I am also thankful to other professors on my advisory committee: Dr. Xu Liu, Dr. Guoliang Jin, and Dr. Huiyang Zhou. They provided much valuable advice and are willing to endorse my dissertation.

I owe thanks to my family. My parents' support is essential to me. They suffer the pain of separation when I study in the other hemisphere of the earth, especially during this pandemic hardship. And it is my children who bring me joy and soothe my heart.

I need to say thanks to my co-author, Benjamin Mariano. He contributed many genius ideas to the work of RDD2SQL. Moreover, his adept writing skill polishes the paper a lot.

Finally, I would like to express appreciation to many other professors and co-workers who helped me. Professor George Rouskas has been my 'secondary' advisor. Professor Ben Watson endorsed my written preliminary exam. It is professor Min Chi who leads me into the world of research. In addition, many talented people in the PICTure research group offered me help. They are Dr. Hui Guan, Dr. Chencheng Ye, Dr. Shuai Yang, Dr. Lin Ning, Yuanchao Xu, Weijie Zhou, Zifan Nan, Jou-An Chen, Hsin-Hsuan Sung, Weiqi Sun, and Dong Xu.

TABLE OF CONTENTS

List of Tables . . . . . **vii**

List of Figures . . . . . **viii**

Chapter 1 INTRODUCTION **1**
1.1 Motivations of Works 3
1.1.1 Across the Boundary between TPC and Host Code 3
1.1.2 Across the Boundary between TPC and User-Defined Extension 4
1.1.3 Across the Boundary between TPCs 4

Chapter 2 Background **6**
2.1 Third-Party Components Studied 6
2.1.1 Numpy 6
2.1.2 Pandas 7
2.1.3 Spark RDD 7
2.1.4 Spark SQL 8
2.2 Counter-Example Guided Inductive Synthesis 8

Chapter 3 Best-Effort Lazy Evaluation **11**
3.1 Introduction 11
3.2 Overview 15
3.3 Dependencies between Operations 16
3.4 Best-Effort Lazy Evaluation (BELE) 19
3.4.1 Minimum Interference Runtime Watching (MIN-Watch) 19
3.4.1.1 Overview of MIN-Watch 19
3.4.1.2 Watch Protocol 20
3.4.1.3 Watching Relatives 22
3.4.1.4 Unwatching Objects via Deferred Flag Resetting 24
3.4.2 Lazy Value Propagation 24
3.4.3 Overhead Control 26
3.4.4 Additional Complexities 26
3.5 Intermediate Representation 27
3.6 Optimizers 27
3.6.1 Reducing Temporary Variables in NumPy 28
3.6.2 Adaptive Caching for PySpark 29
3.6.3 From NumPy to WeldNumpy 29
3.6.4 From Pandas to Spark 31
3.7 API Redirection 32
3.8 Efforts in Applying Cunctator 33
3.9 Evaluation 33
3.9.1 Optimizers 34

3.9.1.1 NumPy . . . 34
3.9.1.2 Spark . . . 37
3.9.1.3 Pandas . . . 37
3.9.2 Overheads . . . 38
3.9.3 Threats to Validity . . . 39
3.10 Related Work . . . 39
3.11 Conclusion . . . 40

Chapter 4 Compositional Lazy Inductive Synthesis . . . 41
4.1 Introduction . . . 41
4.2 Overview . . . 44
4.3 Problem Definition . . . 46
4.4 Background on Dataflow Graphs . . . 48
4.5 Synthesis Algorithm . . . 50
4.5.1 Top-Level Algorithm . . . 50
4.5.2 Synthesis from Dataflow Graph . . . 51
4.5.2.1 CEGIS Instantiation . . . 52
4.5.2.2 Code Generation . . . 54
4.5.3 Coarsening the Decomposition . . . 55
4.5.4 Complexity Analysis . . . 57
4.6 Implementation and Optimizations . . . 57
4.6.1 Parallelization . . . 57
4.6.2 Type-Based Pruning . . . 58
4.6.3 UDF-Level Transformations . . . 58
4.6.4 PBE Back-End . . . 59
4.6.5 Checking Equivalence . . . 60
4.7 Evaluation . . . 60
4.7.1 Benchmarks . . . 61
4.7.2 Manual Study . . . 61
4.7.3 Effectiveness of CLIS . . . 62
4.7.4 Benefits of Decomposition and Laziness . . . 63
4.7.5 Performance Improvement . . . 64
4.7.5.1 Individual UDFs . . . 64
4.7.5.2 TPC-H Workloads . . . 65
4.7.5.3 Case Study on Real Spark Applications . . . 65
4.7.6 Expressiveness Compared to Prior Work . . . 68
4.8 Related Work . . . 69
4.9 Conclusion . . . 71

Chapter 5 Column-Wise Decomposition . . . 72
5.1 Introduction . . . 72
5.2 Overview . . . 74
5.3 Problem Statement . . . 77

5.4 Translation Algorithm . . . 79
5.4.1 Dependence-Free Query Sketches . . . 79
5.4.2 Compositional Synthesis . . . 81
5.4.3 Efficient DFQS Enumeration . . . 84
5.4.4 Properties of the Algorithm . . . 89
5.4.4.1 Soundness . . . 89
5.4.4.2 Completeness . . . 89
5.4.4.3 Benefits of Columnwise Decomposition on Time Complexity 90
5.5 Implementation and Optimizations . . . 91
5.5.1 Source Program Analysis . . . 92
5.5.2 API-Level Decomposition . . . 93
5.5.3 Inductive Synthesizer . . . 93
5.5.4 Equivalence Checking . . . 93
5.5.5 Optimizing the CEGIS Loop . . . 94
5.6 Evaluation . . . 94
5.6.1 Benchmarks . . . 95
5.6.2 Manual Study . . . 95
5.6.3 Effectiveness of Synthesis . . . 96
5.6.4 Evaluating Column-Wise Decomposition . . . 97
5.6.5 Evaluating Optimizations . . . 99
5.6.6 Performance Benefits of SQL Translation . . . 101
5.7 Related Work . . . 102
5.8 Conclusion . . . 104

Chapter 6 CONCLUSIONS AND FUTURE WORKS . . . 105
6.1 Overarching Conclusion . . . 105
6.2 Future Work . . . 106

APPENDICES . . . 125
Appendix A Supported Spark RDD APIs . . . 126
Appendix B Translating Target Language to SQL . . . 127

LIST OF TABLES

Table 1.1 Domains and involved boundaries studied in this thesis 3

Table 3.1 Summary of optimizers 34
Table 3.2 Descriptions of collected benchmarks 35
Table 3.3 Benchmark results 35
Table 3.4 Overhead 38

Table 4.1 Statistics about the benchmark set 61
Table 4.2 Main synthesis results 62
Table 4.3 Improvement on individual UDFs 65
Table 4.4 Summary of applications 67
Table 4.5 Four different test settings 67
Table 4.6 Comparison between CLIS and closely related work 68

Table 5.1 Example of extracting DFQS from a query (Example 9) 87
Table 5.2 Statistics about the benchmark set 95
Table 5.3 Manual study results 96
Table 5.4 Main synthesis results 96
Table 5.5 Summary of speedups 101

Table B.1 Example of translating target language to SQL (Example 10) 128

LIST OF FIGURES

Figure 1.1 Types of boundaries enclosing a third-party component (TPC) 2

Figure 2.1 CEGIS Overview 9

Figure 3.1 NumPy example and WeldNumpy variants 12
Figure 3.2 Architecture of Cunctator 15
Figure 3.3 Running example 15
Figure 3.4 Three types of dependencies 18
Figure 3.5 The architecture of MIN-watch 21
Figure 3.6 A high-level illustration of how MIN-watch works for API *numpy.add(a,b)* 21
Figure 3.7 Default implementation of watch protocol 21
Figure 3.8 Reducing redundant temporary variables in NumPy 28
Figure 3.9 Adding cache operation in Spark 29
Figure 3.10 Translating NumPy to WeldNumpy 30
Figure 3.11 Pandas to Spark 31
Figure 3.12 Speedups 36

Figure 4.1 Schematic overview of our approach 43
Figure 4.2 A Spark query containing a UDF 45
Figure 4.3 Illustration of our approach on running example 45
Figure 4.4 The grammar of our source language. 47
Figure 4.5 An example UDF and its two possible DFGs 49
Figure 4.6 Our instantiation of the CEGIS paradigm 53
Figure 4.7 Merging two nodes 56
Figure 4.8 Types of UDFs that do not have SQL equivalent 62
Figure 4.9 Comparing cumulative synthesis time of different settings and techniques 64
Figure 4.10 Improvement of TPC-H queries with UDFs 66
Figure 4.11 End-to-end improvement for real applications 67

Figure 5.1 Apache Spark Program and Equivalent SQL Query 74
Figure 5.2 Semantic difference between functional APIs and SQL 75
Figure 5.3 Column queries for example 76
Figure 5.4 Functional source language grammar 77
Figure 5.5 Target relational language grammar 78
Figure 5.6 The grammar of query sketch 79
Figure 5.7 Rules for extracting DFQS from a query. 86
Figure 5.8 Cumulative distribution of synthesis time 97
Figure 5.9 Comparing synthesis time of RDD2SQL and RDD2SQL-NoDecomp 98
Figure 5.10 Evaluating asymptotic complexity with and without column-wise decomposition 99
Figure 5.11 Comparing complete and ablated versions 100

Figure 5.12 Distribution of speedups 100

Figure B.1 Rules for translating target language (Fig. 5.5) to SQL 128

CHAPTER

1

INTRODUCTION

Data have been a driving force of modern society. People rely on insights produced by data analytics programs to make all kinds of decisions. To obtain timely and economic data insight, data analytics programs need to be efficient. However, as the amount of data to be analyzed is ever-growing, high-performance data analysis has been an increasingly challenging problem.

To address the performance challenge, people have developed for data analytics many third-party components (TPC), including libraries (e.g., Numpy [WCV11]), frameworks (e.g., Spark [Zah+10]), and DSL runtimes (e.g., SQL [GWO99] engines). These TPCs have various kinds of performance optimization techniques built-in. Some widely employed techniques are data parallelism [HS86], lazy evaluation [Mor+11; FTA14; CMSL16], and query planning [Cha98].

Most of these traditional optimization techniques share one common trait: Their optimization scopes are all within the TPC boundaries. In other words, they are only effective when data flowing through the TPC that they are embedded in, but have no control on how data are processed in other components as well as how data are fed into the TPC. This in-boundary optimization strategy is a reasonable choice, because it is generally hard to obtain knowledge about other components.

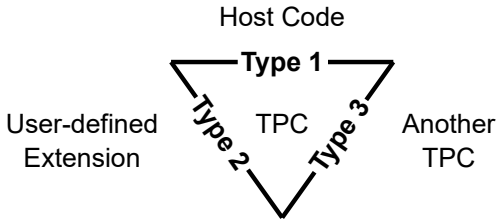


Figure 1.1: Types of boundaries enclosing a third-party component (TPC)

However, how a TPC cooperates with other components is important to the performance of data analytics programs. Traditional TPC-embedded techniques cannot effectively optimize inefficient cooperation, as they either are not aware of other components or simply treat other components as black boxes. Thus we start from a holistic perspective of data analytics programs and propose novel methods that focus on optimizing cooperation of multiple program components. We call such methods cross-boundary optimization.

From a high-level view, cross-boundary optimization involves two challenging tasks: (1) to understand the unified semantics of the code on both sides of the boundary, such that the program can be safely optimized; and (2) to rewrite the code to achieve better performance. Both tasks are challenging because they need to unify boundary-separated components, which involve different design patterns, abstractions, and languages.

As shown in Figure 1.1, we classify the boundaries of TPCs into three types. First, the boundary type 1 is between the TPC and the code hosting it. This is usually materialized as invocations of TPC programming interfaces in the host code. Second, the boundary type 2 is between the TPC and a user-defined extension. Many TPCs allow users to implement modules that can extend their built-in functionalities. For instance, a SQL query engine often allows calling user-defined functions (UDF) written in general procedural languages (e.g., Scala). Finally, the boundary type 3 is between two TPCs. In this thesis, we focus on the case of exploiting existing optimization techniques in one TPC (e.g., Spark SQL [Arm+15]) to optimize programs written in API of another TPC (e.g., Spark RDD [Zah+12]). Unlike type 1 and type 2 boundaries which have explicit programming interfaces, type 3 boundaries are implicit as there is no direct communication between involved TPCs. In case that one TPC calls API of another TPC, or one TPC is an extension of another TPC, the boundary belongs to type 1 or type 2 correspondingly rather than type 3.

Table 1.1: Domains of data analytics programs and involved boundaries studied in this thesis

Chapter	Domain	Boundary		
		Type	TPC	The other component
3	Python programs	1	TPC with Python APIs	Host code in Python
4	SQL queries with UDFs	2	SQL engine	UDF
5	Functional queries	3	Functional query engine	SQL engine

For each boundary type, we identify an important domain of data analytics programs and propose a novel cross-boundary optimization method to optimize it. Table 1.1 lists these domains and involved boundaries. More details can be found in Section 1.1.

The remaining content of this thesis is organized as follows. Section 1.1 introduces the motivations of the three works (one for each boundary type) included in this thesis. Chapter 2 gives common backgrounds of the works. Chapter 3, 4, and 5 present the details of each work. Finally, Chapter 6 concludes the thesis and discusses future works.

1.1 Motivations of Works

1.1.1 Across the Boundary between TPC and Host Code

To study the optimization opportunity around the boundary between TPC and host code, we focus on TPCs with Python APIs, as Python has been a popular hosting language for many influential data analytics libraries and frameworks, including NumPy [WCV11], pandas [McK11], Spark [Zah+10], Tensorflow [Aba+16], and PyTorch [Pas+19].

A fundamental requirement for program optimization is to understand the data dependency between program parts, such that the optimizer can adjust the program without changing its semantics. When it comes to the case of hosting API calls in Python, the data dependency between API calls and host code is generally hard to analyze because of the dynamic data types of Python.

We thus propose best-effort lazy evaluation (BELE), a method to dynamically capture data dependence between Python API calls and the host code during execution of the program. Based on BELE, we implement a framework called Cunctator that enables holistically optimizing the host code and API calls. Supported by Cunctator, we develop four proof-of-concept optimizers for NumPy, Pandas, and Spark. We evaluate the optimizers on real programs collected from Github and observe significant speedups. Details can be found in Chapter 3.

1.1.2 Across the Boundary between TPC and User-Defined Extension

For cross-boundary optimization with user-defined extensions, we focus on the user-defined functions of SQL. Many SQL execution engines allow users to register and call their own functions defined using general procedural languages (e.g., C, Java, and Scala). While user-defined functions (UDFs) improve the flexibility of SQL, queries with UDFs are often inefficient, mainly because most of query optimizers treat UDFs as black boxes and thus do not holistically optimize them with other parts of SQL queries.

In fact, many UDFs can be translated to SQL expressions using built-in SQL functions. A previous work Froid [Ram+17a] leverages this fact to translate an SQL query with UDFs to a UDF-free SQL query before further optimizing the query, such that the computations in UDFs can be optimized together with other parts of the query. Froid performs the translation using syntactic rewrite rules, and cannot handle loops. However, in modern big data analytics programs that utilize SQL (e.g., Spark SQL [Arm+15]), UDFs with loops are common.

To translate UDFs with loops to SQL expressions, we turn to a program synthesis method. Program synthesis means searching for a target program that satisfies a given specification. In our case, the specification is a UDF, and the target program is an SQL expression that is semantically equivalent to the UDF. However, traditional program synthesis methods do not scale well, and take too long time to find complicated target programs. To address the scalability issue, we propose compositional lazy inductive synthesis (CLIS) that exploits the composiblity of both source UDFs and target SQL expressions. Evaluation on real UDFs collected from Github shows that CLIS can translate most of UDFs to SQL expressions if a translation exists. After the translation, the programs' performance is significantly improved. See Chapter 4 for details.

1.1.3 Across the Boundary between TPCs

For optimization across multiple TPCs, we study a fundamental problem of translating a program written with one TPC to a program written with another TPC. If such a translation is feasible, we can generalize it to the cases of programs written with multiple TPCs, through translating program parts using different TPCs to parts using the same TPC. And once the program parts are unified with the same TPC, we could apply traditional in-boundary optimization techniques.

Particularly, we are interested in translating to SQL big data queries written in functional APIs, e.g., map(), filter(), and reduce() in Spark RDD [Zah+12] APIs. Functional APIs and SQL are two popular ways of writing data analytics programs. Moreover, such translation brings to functional APIs the advanced SQL optimization techniques developed in decades, which are

currently not applicable to functional APIs, because SQL optimization techniques require a theory foundation of relational algebra that functional APIs do not have.

We again seek for a program synthesis method, as such a method requires few assumptions on source programs. Initially, we employ a variant of the CLIS algorithm. However, preliminary results show that such an algorithm still suffers serious scalability problems: Many programs cannot be translated within reasonable amount of time.

Further examination of the problem leads to a novel algorithm that we call column-wise decomposition. The algorithm exploits the insight that a multi-column SQL query can usually be decomposed to several smaller queries, each of which corresponds to one column in the original query. Since the one-column queries are smaller, they take less time to synthesis. Thus the algorithm first synthesizes one-column queries and then compose them into a complete query. Evaluation on real Spark RDD programs collected from Github shows that column-wise decomposition can translate most of Spark RDD programs into SQL if a translation exists. After the translation, the queries are significantly accelerated. Details can be found in Chapter 5.

CHAPTER

2

BACKGROUND

In this chapter, we first give background knowledge of the third-party components that we study in this thesis. They are Numpy (Section 2.1.1), Pandas (Section 2.1.2), Spark RDD (Section 2.1.3), and Spark SQL (Section 2.1.4). In addition, we introduce in Section 2.2 counter-example guided inductive synthesis (CEGIS) [SL+06] - the fundamental technique used by the two works presented in chapters 4 and 5.

2.1 Third-Party Components Studied

2.1.1 Numpy

Numpy [WCV11] is a Python library designed to manipulate large multidimensional arrays. It provides a large collections of high-performance mathematical functions, ranging from common arithmetic operations (e.g., `add`) to linear algebra functions (e.g., `matmul`).

The core of Numpy is `ndarray`, a data type for n-dimension arrays. Unlike Python's built-in `list` type, `ndarray` stores *homogeneous* values in *continuous* memory. While Numpy provides high-level Python interface to user, at low-level, `ndarray` and its mathematical functions are impelemented in C/C++ and Fortran. Moreover, Numpy leverage CPU SIMD

instructions to accelerate operations on `ndarray`. As a result, Numpy functions enjoy huge performance advantage over counterparts written in pure Python.

2.1.2 Pandas

Pandas [McK11] is another Python library for data analytics. It focuses on tabular data, also known as `Dataframe` - a data structure first introduced in the R programming language [Tea14]. Conceptually, a `Dataframe` is a table with labeled columns and rows. To manipulate `Dataframes`, Pandas provides APIs to import/export `Dataframes` from/to various data formats (e.g., csv and json), and allows a spectrum of operations on `Dataframes`, such as selecting, updating, and reshaping. Moreover, Pandas is based on Numpy and thus inherits Numpy's high performance. Powered by these features, Pandas has become a handy tool for data cleaning and data wrangling tasks.

2.1.3 Spark RDD

Spark [Zah+10] provides APIs that can be utilized to write data analytic programs running in parallel on a cluster through data partitioning. One important set of Spark APIs manipulates so-called resilient distributed data (RDD) [Zah+12].

RDD is defined as a generic type which represents a logical collection of data. For example, `RDD[Int]` represents a collection of integers. Usually, a Spark program does not need to care about how an RDD is materialized. Spark runtime tries to determine an RDD's best possible materialization, such as how the collection is partitioned and distributed to machines, and whether the data is stored on the disk or in the memory, or even in the register.

Since RDD is just a logical abstraction, it could be virtual, which means that the data it holds does not have to be materialized until the data is really needed, hence the *lazy evaluation* feature in Spark. For instance, for the following Spark program:

```
sc.textFile("foo").map(x => x.toInt).reduce((x,y) => x+y)
```

the file loading and the integer conversion in the `map` call do not get executed until the execution reaches the `reduce` call. This lazy evaluation strategy enables pipelining operations, and thus improve data locality.

As seen in the Spark program example above, many Spark RDD APIs (e.g., `map` and `reduce`) are second-order functions that take as arguments first-order *user-defined functions* (UDFs). While the second-order APIs enable automatic parallelization of the program, the

UDFs offer flexibility of using general-purpose programming languages (e.g. Scala) to express program-specific logic.

2.1.4 Spark SQL

In addition to RDD APIs, Spark also supports SQL [Arm+15]. Although Spark RDD and Spark SQL share Spark's scheduler for distributed processing, their execution paths above the scheduler are independent to each other, and thus we consider them as two different TPCs. They not only differs on programming interfaces, but also utilize different optimization techniques.

In general, Spark SQL enjoys better performance than Spark RDD, thanks to the sophisticated SQL optimization techniques developed in decades. Specifically, there are two main SQL language features that enable its performance advantage over Spark RDD. First, unlike RDD that can contain objects of polymorphic types, SQL requires the column types of involved tables to be pre-defined and concrete. Thus, while in-memory RDD are stored as lists of objects, Spark SQL stores in-memory data more compactly in continuous memory pages, leading to better data locality and less network load when exchanging data between cluster nodes. Secondly, SQL's builtin functions allow users to write queries without UDFs in most cases. Since built-in functions' semantics are known, the query optimizer can leverage them in many optimizations, such as cost-based query rewriting [Sel+89] and whole-stage code generation [Neu11]. On the other hand, semantics of UDFs are generally hard to decide.

2.2 Counter-Example Guided Inductive Synthesis

Since the two works presented in chapters 4 and 5 are based on counterexample-guided inductive synthesis (CEGIS) [SL+06], we now give a brief overview of this method for readers who are new to program synthesis. As shown in Figure 2.1, 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 Figure 2.1). 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

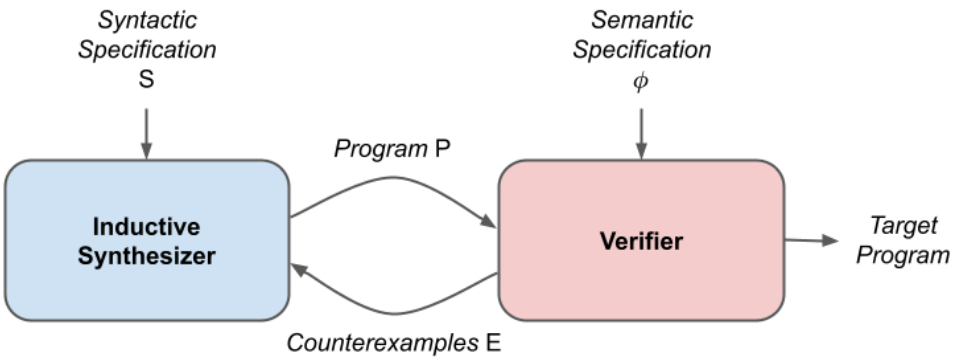


Figure 2.1: CEGIS Overview

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.

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 [SL08] 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.

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 thesis is to transpile UDFs or functional queries to SQL, our semantic specifications take the form of a reference implementation (i.e., Spark SQL [Arm+15] UDF and Spark RDD query [Zah+12]).

Inductive synthesizer. In general, there are a number of ways to implement an inductive synthesizer, including search-based techniques and constraint solving [GPS17]. 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. 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 the size of the synthesized program is proportional to the size of the reference implementation [Wan+19b; VG+20; SSA13].

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 [CC77] 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.

CHAPTER

3

BEST-EFFORT LAZY EVALUATION

3.1 Introduction

Modern software is built upon APIs. Although APIs typically encapsulate highly optimized code, suboptimal usage of APIs can cause large performance degradation. Such a problem is especially common in Python programs, as Python has become the host language of many popular libraries or domain-specific languages (DSL) targeting performance-demanding tasks, such as NumPy [WCV11], Pandas [McK11], PySpark [Zah+10], TensorFlow [Aba+16], and PyTorch [Pas+19].

Suboptimal usage of APIs typically involves a sequence of API calls. For illustration purpose, we show a simple example in Figure 3.1(a). While the code is simple, it suffers the performance flaw of a redundant temporary value: S1 creates an object and assigns it to `x`, but after `x` points to another object in S2, Python garbage collector (GC) releases the former object as it now has zero reference count. The program can be optimized by replacing the second statement with an in-place operation: `numpy.add(x, c, out=x)`. The argument `out=x` instructs `numpy.add` to reuse `x` to store the result. The optimization not only improves data locality, but also reduces memory usage.

```
S1: x = numpy.add(a,b)
S2: x = numpy.add(x,c)
```

(a)

```
S1: wa= weldarray(a)
S2: x = weldnumpy.add(wa,b)
S3: x = weldnumpy.add(x,c)
S4: x.evaluate()
```

(b)

```
S1: wa = weldarray(a)
S2: x = weldnumpy.add(wa,b)
S3: x = weldnumpy.add(x,c)
S4: a[0] = 0
S5: x.evaluate()
```

(c)

Figure 3.1: NumPy example and WeldNumpy variants

Existing work to tackle the problem of suboptimal API sequence relies on lazy evaluation. Several API sets, such as Spark [Zah+10], TensorFlow [Aba+16], and WeldNumpy [Pal+17], have been designed and implemented in that way. They designate some APIs as eager APIs and the rest as lazy APIs. Invocations of lazy APIs only log the APIs in a certain form rather than execute them. Once an eager API is encountered, the logged sequence of APIs will be optimized together and then executed. For instance, Figure 3.1b shows the WeldNumpy version of the code in Figure 3.1a; the two `add` operations are not evaluated until S4; before the evaluation happens, the WeldNumpy runtime optimizes the two add operations and avoids the unnecessary object creation for `x` in the second add operation.

A fundamental problem underlying the API-based lazy evaluation is the data dependence that arises between the invocations of the APIs and the host Python code. Figure 3.1c gives an illustration. Compared to Figure 3.1b, the difference is that a Python statement S4 updates the input of S1 before `evaluate()`. Python statements, by default, are eagerly evaluated. But as the `weldnumpy.add` API is lazily evaluated, S2 would end up using the wrong values of `a`.

Existing frameworks either leave the issue to the programmers (e.g., in WeldNumpy [Pal+17]), relying on them to put in eager APIs at the right places, or design the library such that any API that might incur dependencies with the host code is designated as an eager API, regardless of the context (e.g., in Spark [Zah+10] or TensorFlow [Aba+16]). The former increases programmers' burdens, while the latter often misses optimization opportunities due to its conservative design.

Listing 3.1 shows an example in *Spark*. It loads a text file (Line 1), splits the lines into words (Line 2), filters out illegal words (Line 3), counts the number of words (Line 4), sums the

Listing 3.1: A Spark program with performance issues that are hard to automatically optimize away

```
1 lines = sc.textFile("foo")
2 ws = lines.flatMap(lambda l: l.split())
3 ws = ws.filter(lambda x: re.match("^[\w]+$", x))
4 word_count = ws.count()
5 total_len = ws.map(lambda w: len(w)).sum()
6 avg_len = total_len / word_count
```

lengths of all words (Line 5), and finally outputs the average word length (Line 5). In *Spark*, the APIs `textFile`, `flatMap`, `filter`, and `map` are always lazily evaluated; both `count` and `sum` are always eagerly evaluated APIs because they return values to the host code and hence the value, in general, could potentially be operated on by the host code. When an eager API is invoked, Spark fuses relevant lazy APIs together into a pipeline; intermediate results are not cached. As there are two eager API calls, the lazy operations `textFile`, `filter`, and `flatMap` are evaluated twice at lines 4 and 5. The solution from *Spark* is to introduce extra APIs such that programmers can use them for caching. This "band-aid" solution further increases the burdens of programmers, who now need to be concerned of not only the usage of the many existing APIs but also the best places to use the caching APIs.

Our study (Section 3.9) shows that these limitations prevent existing frameworks from tapping into the full potential of lazy evaluations for Python+API programs, leaving up to 14X performance improvement yet to harvest.

The primary goal of this work is to create a solution that overcomes the limitations of the existing methods for enabling lazy evaluation for Python+API programming. The principles for developing our solution are two fold: (1) It should be automatic such that programmers do not need to worry about manually finding the best places in their code to insert APIs to trigger evaluations; (2) it should be effective in postponing API evaluations to places as late as possible to maximize API optimization opportunities.

The key to both principles is to effectively analyze data dependencies between the host code and the APIs in a Python program. The problem is challenging. Many features of Python, such as dynamic typing and reflection, make analysis of the host code difficult. The difficulty is exacerbated by the extra need to analyze library APIs and their interactions with the host code. The lack of such automatic data dependence analysis is plausibly one of the main reasons for the unsatisfying solutions being used today.

In this chapter, we address the challenge by developing a *minimum interference runtime watching scheme* (MIN-watch for short). The basic idea underlying MIN-watch is simple, tracking data accesses at runtime to detect data dependencies. The novelty is in how MIN-watch makes the tracking efficient and effective for sound dependence detection in the context of Python+API programs. MIN-watch does it by taking advantage of the characteristics of Python and the special needs in lazy evaluation for Python+API. It is resilient to Python language complexities. It minimizes runtime overhead through a focused tracking scope in data and an efficient runtime checking mechanism (bit-level flagging and deferred flag resetting). It meanwhile imposes near-zero burdens on programmers. MIN-watch is based on a dependence theorem we introduce to formulate the correctness of lazy evaluation in this host+API context (Section 3.3).

Based on MIN-watch, we further develop Cunctator, a software framework for materializing the extended lazy evaluation. Cunctator consists of an intermediate representation (lazy IR) for the deferred operations, a lazy IR evaluator, a class that delegates the results of deferred operations and postpones operations applied to itself, and a set of interfaces for redirecting API calls and registering optimizers. With these components together, Cunctator provides programmers the conveniences of enabling the automatic *Best-Effort Lazy Evaluation (BELE)* for a Python library and harvesting the optimization benefits.

To demonstrate the usefulness of Cunctator, we implement four optimizations enabled by BELE for three API packages (`Numpy, Spark, Pandas`). Experiments on 15 programs show that the optimizations generate 1.03-14.2X speedups. Stress testing shows that the overhead of Cunctator is no greater than 2.25% (in its default setting).

In summary, this work makes the following major contributions:

- It introduces the concept of Best-Effort Lazy Evaluation, and shows that MIN-watch is effective in enabling data dependence analysis for Python+API programs to support Best-Effort Lazy Evaluation.
- It develops the first software framework to support Best-Effort Lazy Evaluation for Python+API programs.
- It demonstrates the effectiveness of the proposed techniques in enabling optimizations of Python+API programs.

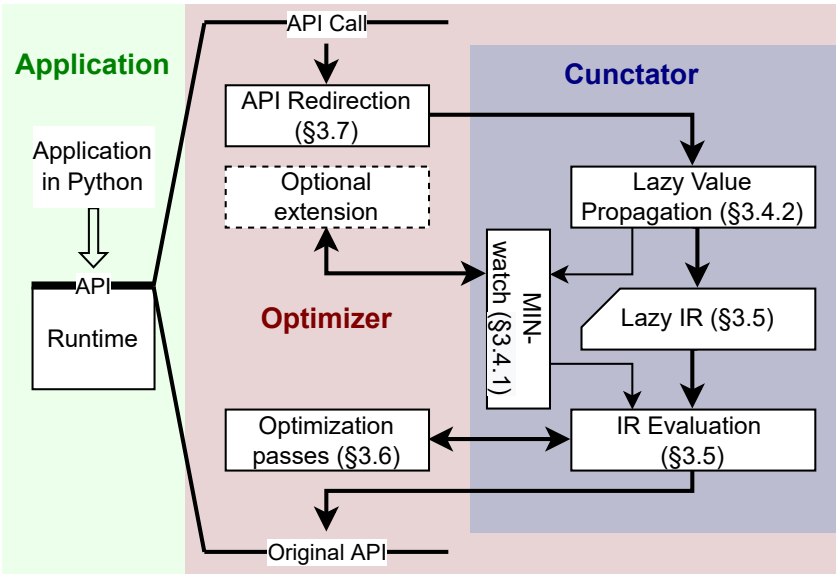


Figure 3.2: Architecture of Cunctator

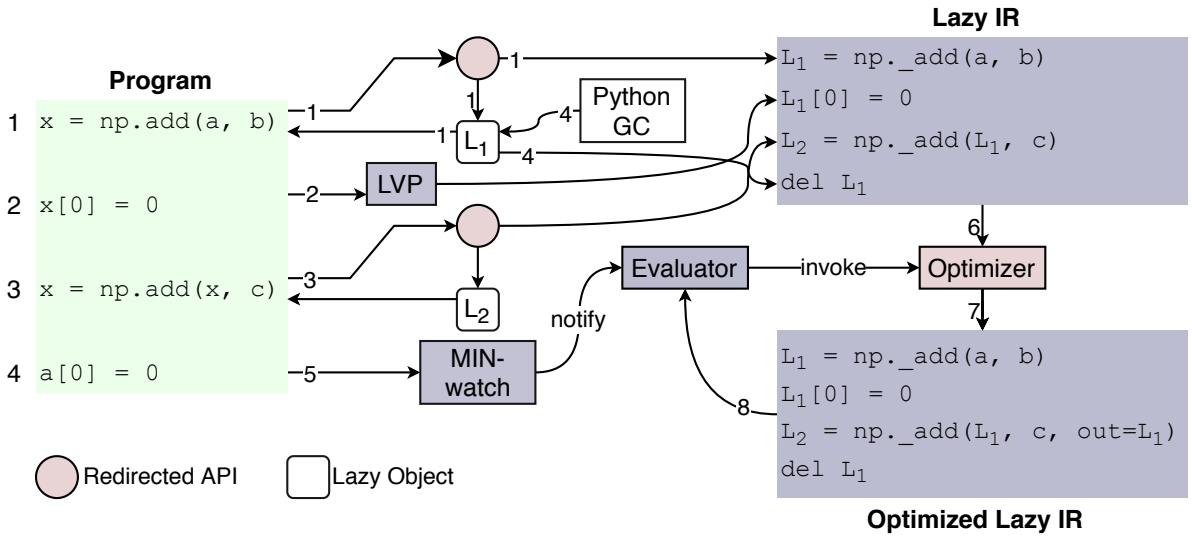


Figure 3.3: Running example. Numbers on directed edges indicate the order of actions.

3.2 Overview

Figure 3.2 illustrates Cunctator's architecture. When an application invokes a DSL API, the API call is redirected to a Cunctator optimizer. Instead of evaluating the API, the optimizer records the API in the form of Lazy IR (Section 3.5), and returns a *lazy object*. The lazy object supports *Lazy Value Propagation* (LVP, see Section 3.4.2), which tries to propagate a new lazy

object when an operation is applied to the lazy object. Cunctator employs MIN-watch (Section 3.4.1) to monitor accesses to objects related with the deferred operations. When MIN-watch encounters host statements or APIs that prevent further delays (based on dependence theorems in Section 3.3), it triggers the evaluation of the deferred operations. During the evaluation, the Cunctator optimizer applies optimization passes (Section 3.6) onto the IR, and then invokes the original DSL APIs for evaluation. To apply Cunctator to a domain, the developers of the DSL optimizer only needs to use Cunctator interfaces to specify redirections of the domain APIs, to support MIN-watch for some common types, and to write domain-specific optimizations. The extra work an application developer needs to do is just to import one or several modules.

Figure 3.3 shows the execution flow of a NumPy program with Cunctator. First, the `np.add` in line 1 is redirected to Cunctator optimizer, which records the API call as a lazy IR instruction and returns a lazy object L_1. Note, the assignment to x is not deferred but executed, and x now points to the lazy object L_1. The optimizer also sets up the two arguments, `a` and `b`, for watching. At line 2, because `x` is lazy, the Lazy Object class automatically captures and logs this operation and defers its execution. Line 3 is similar to line 1, and Cunctator defers and logs the operation. That assignment makes `x` point to L_2; L_1's reference count reduces to zero, which triggers Python's garbage collection on L_1. L_1's deconstructor, however, rather than deconstructs L_1, defers the deconstruction and inserts a `del` instruction into the IR. Line 4 tries to update `a`, which is captured by MIN-watch, which triggers the evaluation of all the deferred operations. The evaluator first invokes the optimizer, which reduces redundant temporary variables, and then evaluates the operations.

Before presenting the details of Cunctator, in the next section, we first define some terms and prove a dependence theorem that formulate Cunctator's correctness.

3.3 Dependencies between Operations

To ensure the correctness of Cunctator, one key aspect is to properly manage the dependencies between postponed API calls and eagerly executed statements. We first introduce a set of terms that are used in the following discussions.

Terminology Unless otherwise stated, an *object* denotes a Python object. An *operator* denotes a Python built-in operator. An *operation* denotes the process of applying an operator to its operands. For example, `foo.bar()` consists of two *operations*: The '.' *operator* is applied

to `foo` and `"bar"` to return a function object, which becomes the operand of the '()' *operator*. We in addition introduce the following terms.

- *Contents of an object*: All in-memory states that could be potentially accessed or updated *directly* through the object's fields and methods. Take the `list` object `["foo", "bar"]` as an example—its contents are the references to its two elements, the string objects, rather than the characters of the strings. By this definition, two objects could share contents, namely, their methods or attributes could access or update the same in-memory state.

- *Sealed object*: An object that shares no content with other objects. This means the contents of a sealed object can be accessed or updated only by its own attributes or methods.

- *Domestic object*: An object whose attributes and methods access no external resources (e.g., files and network) but only the memory space of the current process. We are interested in sealed and domestic objects (e.g., `list` objects).

- *Dependents of an object*: The object itself and the objects referred to in the contents of the object. For example, the dependents of a `list` are itself and its elements.

- *Relatives of an object*: Object R is a relative of object O if and only if there is a dependent chain $O \leftarrow ... \leftarrow R$, in which $x \leftarrow y$ denotes that object y is a dependent of object x.

- *Regular operation*: An operation is regular if the relatives of its operands and return value are all sealed and domestic, and it only accesses or updates the contents of its operands' relatives or newly created objects during the operation. In most cases, a DSL API call is a regular operation. One example of irregular operation is an API manipulating global variables.

Without noting otherwise, the following discussions assume regular operations and sealed and domestic objects and there is no exceptions. Section 3.4.4 discusses exceptions and other complexities.

Dependency types Based on the above definitions, we classify potential dependencies between an API call O_A and a statement O_B into three types:

- *Return-access*: The return value of O_A is accessed (read or written) by O_B, as illustrated by the top left example in Figure 3.4.

- *Input-update*: A relative of O_A's operands is updated in O_B, illustrated by the bottom left example in Figure 3.4.
- *Update-access*: O_A updates a relative of its operand I and O_B accesses that relative, illustrated on the right side in Figure 3.4. Cunctator uses a conservative version of this definition, which forgoes the requirement of the two relatives being the same. It simplifies runtime checking as shown later.

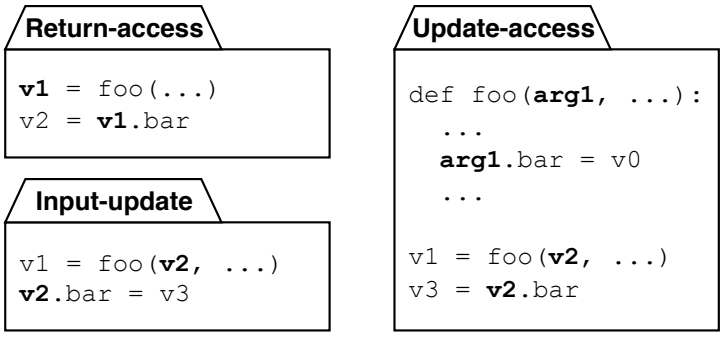


Figure 3.4: Three types of dependencies

Dependency Theorem. This part presents the dependence theorem governing the validity of lazy evaluation for APIs, which underpins the design of Best-Effort Lazy Evaluation.

Lemma 1. *For an API call A followed by a statement B, deferring the execution of A to a point after B does not change the data dependencies between them if there are no* return-access*,* input-update*, or* update-access *dependencies between them.*

The lemma comes from the observation that for the properties of sealed and domestic objects and regular operations, the three types of dependencies cover all possible data dependencies (true dependencies, anti-dependencies, output dependencies) [AK01] between two statements.

Theorem 1. *For an API call A followed by a sequence of statements S, deferring the execution of A to a point after S is valid if there are no* return-access*,* input-update*, or* update-access *dependencies between A and any of the statements in S.*

This theorem is derived from the classic *fundamental theorem of dependence* [AK01], which states the following: Any reordering transformation that preserves every dependence in

a program preserves the meaning of that program. Deferring executions is clearly a kind of reordering transformation. The deferring does not cause any dependence changes according to Lemma 1 for none of the three types of dependencies exist between A and S. The theorem hence holds.

Theorem 1 is essentially a variant of the *fundamental theorem of dependence* in the context of API lazy evaluation; the benefits of having it are however significant. It entails what types of dependencies are needed to consider during lazy evaluation, and what set of data objects are needed to watch, which lay the foundation for the design of MIN-watch and BELE in the next section.

3.4 Best-Effort Lazy Evaluation (BELE)

The purpose of BELE is to defer DSL API calls until the necessary moment. The central challenge that BELE confronts is to satisfy three mutually constrained requirements: First, BELE has to ensure correctness of the program. Second, the deferring period, or *laziness*, should be as long as possible to harvest optimization opportunities. Finally, the overhead should be low.

To address these challenges, we introduce minimum interference runtime watching (MIN-watch) in Section 3.4.1 to detect, with low overhead, *input-update* and *update-access* dependencies between deferred API calls and host code. In addition, Cunctator employs *lazy value propagation* (LVP) to manage *return-access* dependencies while ensuring enough laziness; see Section 3.4.2. The overheads of Cunctator and strategies to control them are discussed in Section 3.4.3. Finally, Section 3.4.4 describes how to handle special scenarios.

3.4.1 Minimum Interference Runtime Watching (MIN-Watch)

Based on Theorem 1, the key for BELE is in detecting data dependencies. MIN-watch takes the way of runtime object watching, which makes it not subject to the language complexities Python imposes on compilers or other static methods.

3.4.1.1 Overview of MIN-Watch

What makes MIN-watch distinctive over common runtime access tracking is the strategy it employs, which takes advantage of the characteristics of this problem setting and Python language properties, and uses a lightweight type-based scheme for non-intrusive implementation. Specifically, the design of MIN-watch draws on three observations: (1) In Python, most memory

accesses go through object interface with multiple layers of redirection and procedure abstractions, hence a much reduced sensitivity to runtime memory access tracking overhead compared to many other languages and settings. (2) The key to BELE is the dependence between API and host. So many data accesses that are irrelevant to such dependencies can be exempted from tracking. (3) Python object assignments and parameter passing are both through references; so to check dependencies related to an actual object, it is not necessary to track references to it, if the watching scheme is put onto that object.

Built on the observations, MIN-watch has the following features: (1) By focusing on API to host dependencies and Theorem 1, MIN-watch concentrates runtime watching on only relevant data objects. (2) It employs an efficient runtime checking mechanism (bit-level flagging and deferred flag resetting) via the Python interpreters to minimize interference to program executions. (3) It employs a type-based approach to enabling runtime object watching, but does it in a non-intrusive way such that application developers need to make no changes to the implementation of a data type for the approach to take effect. Moreover, the utilities in Cunctator simplify the work an optimizer developer[1] needs to do to enable MIN-watch (and BELE) for a domain DSL. The first two features make MIN-watch efficient, and the other features make it easy to use.

Figure 3.5 shows the architecture of MIN-watch, and Figure 3.6 uses `numpy.add(a, b)` as an example to illustrate at a high level how MIN-watch works. The API was overloaded such that when the API is called in a program, instead of doing the computation of arrays addition, it sets up objects a and b and their relatives for runtime watching via function `setupWatch`. Function `setupWatch` calls a function `findRelatives()` to go through each relative of an object, and calls `__set_watch__` of that object to set it up for runtime watching. The setup process flags some special bits in the object header such that the extended Python interpreter, when executing a statement, can recognize such objects and invoke Cunctator lazy evaluation listener to evaluate deferred operations.

We next explain MIN-watch in detail, starting with the basic watch protocol on a single object (Section 3.4.1.2), and moving on to describe the procedure in finding and watching all relatives (Section 3.4.1.3).

3.4.1.2 Watch Protocol

Cunctator adds the following method into the root class in Python:

```
def __set_watch__(self, watchUpdateOnly):
```

[1]Please note the differences between an application developer and an optimizer developer.

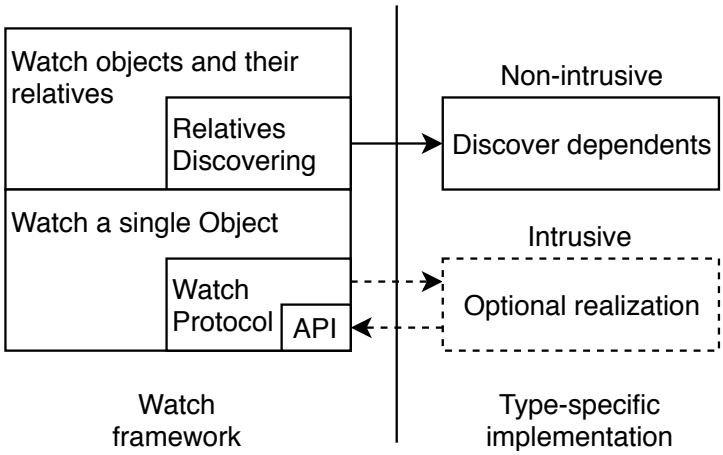


Figure 3.5: The architecture of MIN-watch. Arrows indicate invocation; dotted lines indicate optional implementation.

```
# in module numpy
# gIR: the global IR scratchpad
def add(a, b):
   setupWatch(a, True)
   setupWatch(b, True)
   id = gIR.add_call(...)
   return Lazy(gIR, id)

def setupWatch(obj, watchUpdateOnly):
   for r in findRelatives(obj):
      r.__set_watch__(watchUpdateOnly)
```

Figure 3.6: A high-level illustration of how MIN-watch works for API *numpy.add(a,b)*

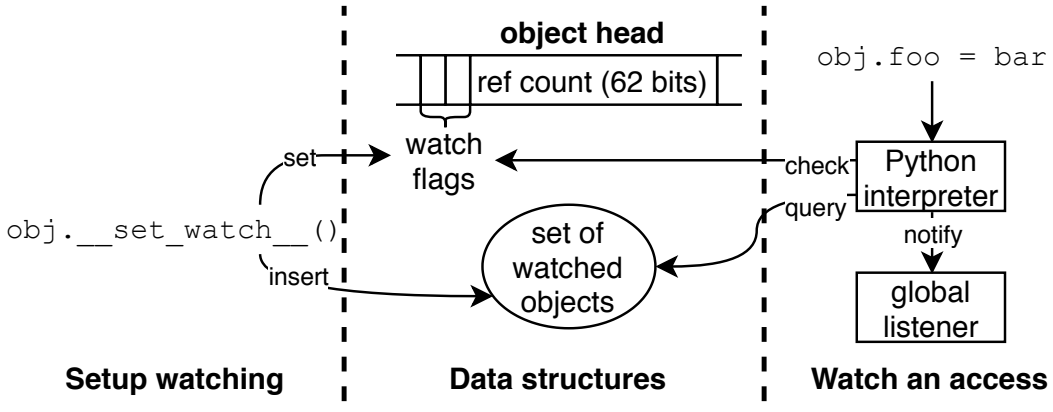


Figure 3.7: Default implementation of watch protocol

```
# pseudo-code
self.watch_flag = WATCH_UPDATE_ONLY
g_watch_set.insert(self)
```

The parameter *watchUpdateOnly* determines whether the object should be watched for read and write accesses (`false`) or just writes (`true`).

Figure 3.7 depicts the actions when the protocol takes place. In the setup time (e.g., when `numpy.add(a,b)` is called in the example in Figure 3.6), `__set_watch__` sets two bits in the object head to indicate whether the object is to be watched for update only (01), read/update (10), or nothing (00). These bits help the interpreter determine the corresponding action quickly. Our implementation borrows the first two bits of the reference count field of an object. That saves extra memory, and also helps ensure that most third-party binary libraries are still compatible by keeping the length of the head unchanged. The method `__set_watch__` in addition adds the object into a global set `watchSet`. It is for fast resetting at the time when the deferred operations are evaluated, which will be elaborated in Section 3.4.1.4.

We extend the Python interpreter such that it notifies Cunctator lazy evaluation listener when the content of an object that is being watched is accessed by a bytecode (e.g., `LOAD_ATTR` and `STORE_SUBSCR`).

The default implementation of `__set_watch__` ignores the `watchUpdateOnly` parameter (i.e., assuming it is false). It is because when just encountering the statement, for some data types, the interpreter sometimes cannot tell whether the access would update the object. (Note that it is legitimate in Python for a seemingly read-only (e.g., `foo.bar`) operation to update the object.) This conservative implementation may reduce the laziness but won't cause correctness issues. For a given data type, the optimizer developer can choose to customize its `__set_watch__()` method and other methods to enable a more precise treatment.

It is worth noting that if an object is not sealed, accesses or updates to the object's contents through other objects are not watched with the default implementation. This is fixed in the upper relatives discovering component by not supporting the specific type, which causes the watch process to fail and thus triggers eager evaluations of involved operations.

3.4.1.3 Watching Relatives

With the watch protocol, we can watch a single object. MIN-watch requires watching all the relatives of an object of interest as Figure 3.6 has shown. The watch framework holds a registry that can register user-defined procedures to discover dependents of specific types. Through recursively calling registered procedures, all relatives of an object can be found. For example,

a list A contains objects B, C, and D, and D is another list that contains E and F. Then, the dependent-discovering procedure registered for type `list` returns B, C, and D for object A, after which a recursive call of the procedure for D returns E and F. Typically, a type-specific dependent-discovering procedure is easy to implement. For example, the procedure for `list` is as simple as[2]:

```
def list_deps(l):
    for e in l:
        yield e
```

Algorithm 1 shows the process of setting up to watch an object's relatives. The SETWATCH procedure first checks existing watch flags and returns in two cases: The first case is that the object is watched for access, when the procedure returns disregarding the parameter `watchUpdateOnly`. The second case is that the object is watched for updates only and the parameter `watchUpdateOnly` is `True`. In other cases, the procedure sets up to watch the object through the watch protocol and then recursively calls SETWATCH for each of its dependents (except for the object itself). If the type of the object is not registered in the registry, the procedure raises an exception, which will be caught by Cunctator to trigger an eager evaluation of the involved operation.

Algorithm 1 Setting up to watch an object's relatives

```
1:  DepReg ← the registry for discovering dependents
2:  procedure SETWATCH(obj, watchUpdateOnly)
3:      if obj is watched for access then return
4:      if watchUpdateOnly ∧ obj is watched then return
5:      obj.__set_watch__(watchUpdateOnly)
6:      for all d ∈ DEPS(obj) do
7:          SETWATCH(d, watchUpdateOnly)
8:  procedure DEPS(obj)
9:      if type(obj) is registered in DepReg then
10:         return DepReg.getHandler(type(obj))(obj)
11:     else
12:         raise an exception
```

[2] `yield` is a Python construct that returns the next value in the next call of its enclosing function.

When Cunctator defers an operation, it invokes the SETWATCH procedure for all the operands except for lazy values, whose potential dependency is handled by lazy value propagation.

3.4.1.4 Unwatching Objects via Deferred Flag Resetting

After the deferred operations are triggered to get evaluated (or when a watch procedure is aborted because of an unsupported type, see Section 3.4.1.3), Cunctator would need to clear the watch flags of all watched objects. Otherwise, later accesses to them would trigger unnecessary evaluations. Going through all the objects could incur substantial overhead. Cunctator circumvents it by introducing a global *watchSet*. Recall in Figure 3.7, `__set_watch__()` puts an object to be watched into *watchSet* at setup time. That set is emptied once the evaluation of deferred operations is triggered. Python interpreter, when it encounters an object with watching bits set, would check whether that object is within *watchSet*. If not, it cleans the watch bits; otherwise, it invokes Cunctator lazy evaluation listener.

3.4.2 Lazy Value Propagation

Return-access dependency is easy to detect for lazily evaluated operations, since all subsequent visits to the return value fall to the actually returned lazy object, which can trigger the evaluation whenever it is used, similar to how the modifier `lazy` works in some other popular languages (e.g., Scala and Swift). However, too often, a lazy object is used shortly after it is returned. For example, in the statement `(a, b) = lazy_func()`, the lazy object is used to unpack its elements right after it is returned from `lazy_func()`. In such cases, an evaluate-when-used semantics of lazy objects results in short-lived laziness, and leaves no optimization opportunities. As a solution to ensure sufficient laziness, we enhance Python with lazy value propagation (LVP), which propagates new lazy values for most operations applied to existing lazy values. In this way, the return-access dependency is not violated, since the operation that uses the return value is deferred as well.

When a lazy value is being operated, LVP records the operation into the lazy IR and then returns a newly created lazy object. One problem that LVP has to solve is when the propagation should stop – in other words, when the true evaluation should be triggered. An evident scenario is when a lazy value is used to determine the execution branch (e.g., the `if` condition). Theoretically, we could explore all possible paths and collect the lazy IR in the form of computational tree logic (CTL) [CE82]. Such exploration, however, would introduce large

Listing 3.2: Lazy value propagation

```
class Lazy:
    def __init__(self, ir, v):
        self.__ir, self.__v = ir, v
    def __add__(self, r):
        if self.__ir.evaluated(self.__v):
            return self.__ir.value(self.__v) + r
        watch(r)
        newv = self.__ir.add_op2('+', self, r)
        return Lazy(self.__ir, newv)
    def __bool__(self):
        return bool(self.__ir.evaluate(self.__v))
    # More overwritten operations
    ...
```

overhead, while its benefit is unclear. Another situation of stopping LVP involves overhead control (see Section 3.4.3).

Cunctator implements LVP within the class of lazy objects through operator overwriting, as shown in Listing 3.2. A lazy object is bound with one lazy IR variable. The `__add__` function, for instance, overwrites the operation `lazy + r`. Their operands are set up for watching before the operation is recorded in the lazy IR. Other operations are overwritten in a similar way, except for the `bool()` operation, which triggers the evaluation, as the operation is invoked when a value is used for branch selection. Although the `bool()` operation does not necessarily imply branching, it is a good heuristic.

A special operation in Python is accessing an object's attribute. Commonly, the attribute is accessed by the '.' notation (e.g. `o.a`), which can be overwritten by `__setattr__()` and `__getattr__()`. But the special `__dict__` attribute can be used to access other attributes. For example, `o.__dict__["a"]` is equivalent to `o.a`. Cunctator extends the Python interpreter to invoke the lazy evaluation listener when the `__dict__` attribute is accessed.

It is worth noting that Cunctator chooses to implement LVP in pure Python for fast prototyping. We plan to re-implement it as a part of Python interpreter in the future. This built-in LVP will have lower overhead and know precisely when a value is used for branch selection.

3.4.3 Overhead Control

If there are too many inexpensive DSL API calls or too many propagated lazy values, generating and evaluating the lazy IR could introduce too much overhead. Although such cases never appear in our experiments, we still introduce a dynamic scheme to prevent it from happening in the extreme cases. Cuncator employs a parameter $\mathcal{N}_{IRPS}$ to control how many lazy IR instructions can be generated per second. Initially, Cunctator sets a variable $\mathcal{M}$ to $\mathcal{N}_{IRPS}$. When the total number of generated instructions is equal to $\mathcal{M}$, Cunctator evaluates recorded IR, then it sets $\mathcal{M}$ to $\mathcal{N}_{IRPS} * T$, in which T is the total elapsed time since the first API is deferred. If $\mathcal{M}$'s new value is smaller than its old value, it indicates that the program is an extreme case; Cunctator disables itself by avoding API redirection and LVP. In our experiments, we set $\mathcal{N}_{IRPS}$ to 1000.

3.4.4 Additional Complexities

Exception. Theorem 1 assumes that neither O_A nor O_B raises exceptions. Exceptions could direct the execution to their handlers. If there is no exception handler set up, which is the case for most of the DSL programs we encountered, any raised exception would cause the program to crash. Thus, Cunctator disregards potential exceptions of an operation when there is no installed exception handler. When the current context has exception handlers, Cunctator disables BELE, and thence, all operations are eagerly evaluated. Cunctator checks the currently installed exception handlers through an interface added to the Python interpreter.

External dependency. Theorem 1 assumes that O_A and O_B are not dependent on each other through external resources (e.g., one writes to a file, and the other reads the file). Cunctator considers that the information of whether lazily evaluated APIs access external resources as domain knowledge and relies on the optimizer's developer to provide the knowledge. If none of the lazily evaluated operations access external resources, there is no external dependency. Otherwise, a monitor that watches the program's system calls could notify Cunctator when the program tries to access external resources; Cunctator can then avoid deferring the operations.

Unwatchable objects. Although the watch framework works in most cases, there are objects that cannot be watched because they are not sealed or domestic. For example, if an object holds a segment of shared memory, an update to the shared memory in another process will not notify the listener. In addition, it is impractical to implement MIN-watch for all potential types; thus, some uncommon types may not support MIN-watch. Any kind of unwatchable object causes an involved operation to be eagerly evaluated.

Loss of seal. A sealed object may become not sealed at runtime. For example, invocation of `numpy.ones()` creates a sealed object *O*; however, `O.reshape()` may create a new object *P* that shares *O*' data buffer (not a Python object) through pointer alias, rendering that *O* is not sealed any more, and updates to the data buffer by operating *P* cannot be monitored by watching *O*. Therefore, if a type supports MIN-Watch, and there is a method of the type leaks the content, the method needs to mark the involved object as unwatchable by setting watch flags' value to 11 (see Section 3.4.1.2). Subsequent attempts to set watch on *O* will enforce eager evaluation.

3.5 Intermediate Representation

This section gives details on the design of the lazy IR in Cunctator. The lazy IR has a static single assignment (SSA) form. Each instruction is a 4-tuple:

$$< ID, OP, Operands, Annotation >$$

ID is a globally unique name, which represents the result of current instruction. OP is the operator, such as '+', '.' (attribute access), '[]' (array alike access), '()' (function calls). $Operands$ are stored as a list. $Annotation$ can be used to store any extra info that the optimizer may use. For an API call, for instance, it is logged as a `call` instruction (OP is '()'), the function pointer is stored in the $Operands$ field along with the function's arguments, and the API name is put into the $Annotation$ field.

An operand of an IR instruction could be either a lazy value or a non-lazy value. When an operand is a lazy value, the instruction stores its ID. For a non-lazy value, the instruction stores a reference to it. (In our discussion, L_x denotes a lazy value, N_x a non-lazy value, and V_x can be any value.)

Cunctator provides a simple interface for optimizer developers of a DSL to register optimization passes. Each optimization pass accepts a sequence of IR instructions as input, and outputs an optimized sequence. Registered optimization passes are chained in order. During an evaluation, the sequence of all recorded IR instructions since the last evaluation is passed down through all optimization passes.

3.6 Optimizers

Cunctator is an enabler. By enabling BELE, it paves the way for many optimizations that are not supported by existing DSL frameworks. We have implemented proof-of-concept DSL

optimizers for NumPy, Pandas, and Spark. These optimizations fall into two categories: *in-language* optimization and *cross-language* optimization. The in-language optimization tries to identify inefficient API uses and replace them with some other APIs of the DSL. The cross-language optimization tries to replace APIs of one DSL with APIs of another DSL. Two techniques for each category are illustrated in the following sections.

3.6.1 Reducing Temporary Variables in NumPy

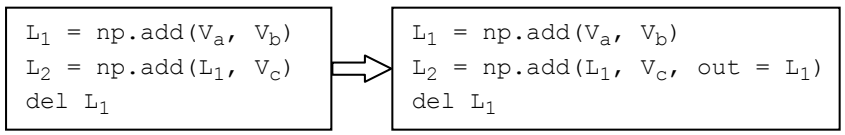


Figure 3.8: Reducing redundant temporary variables in NumPy

Redundant temporary variables are a performance issue in many NumPy programs. They impair performance in two ways. First, value assignment to a new variable has worse data locality than an in-place value update. Second, depending on the array size, a temporary variable can consume a lot of memory and thus increase peak memory usage.

When the API call trace is collected as lazy IR in Cunctator, an optimizer can easily optimize away a redundant temporary variable through pattern matching and IR rewriting. At the pattern matching stage, the optimizer locates a redundant temporary variable L_a if the following conditions are all satisfied:

- L_a's value is initialized from the result of an operation that generates a new value rather than performing in-place update.
- L_a participates in no in-place updating operations.
- L_a is passed to an operation O that generates a new value L_b, and O has a counterpart O' that performs an in-place update.
- After being used in operation O, L_a is deleted and participates in no other operations.

At the IR rewriting stage, the optimizer replaces the operation O with O', which saves the result to L_a. Figure 3.8 shows an example of this optimization technique.

3.6.2 Adaptive Caching for PySpark

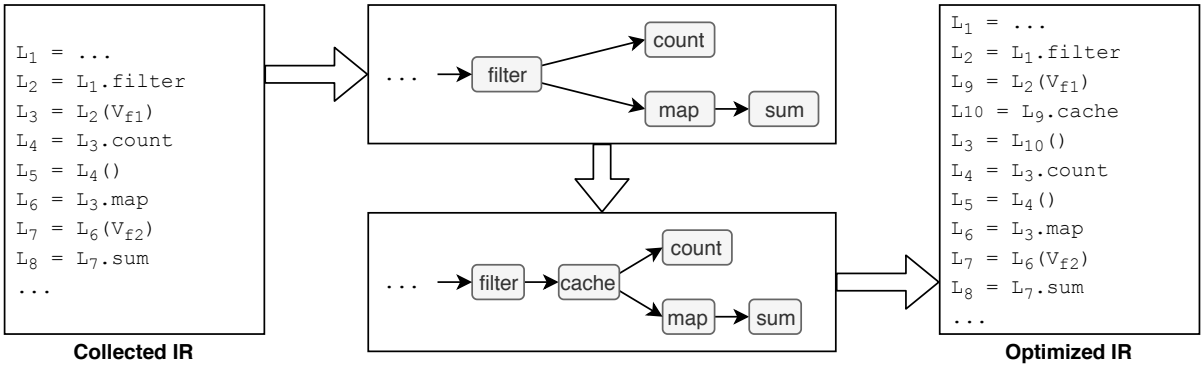


Figure 3.9: Adding cache operation in Spark

PySpark is Spark's Python programming interface. Although Spark's runtime employs lazy evaluation to optimize its API call sequences, it fails to handle performance flaws similar to that in Listing 3.1, because an eager API does not know whether the intermediate result of a lazy API will be used by a subsequent eager API.

With Cunctator, the performance problem in Listing 3.1 can be optimized away by adding `cache` operations for intermediate results used by more than one eager operation, as shown in Figure 3.9. The IR shown on the left side of the figure is collected by Cunctator. Note that `del` instructions are omitted for concision. Based on the collected IR, the optimizer constructs a data flow graph for all Spark operations. If two or more eager operations share a common ancestor, the optimizer inserts a `cache` operation at the fork.

Another similar performance problem involves unnecessary `cache` operations, namely, `cache` operations for intermediate results used by only one eager API. Such operations introduce unnecessary memory writing and consume a lot of memory. Based on the same graph analysis as was used for inserting `cache` operations, the optimizer can identify and remove unnecessary `cache` operations.

3.6.3 From NumPy to WeldNumpy

WeldNumpy [3] was developed as a replacement for NumPy with better performance, which was achieved via two main techniques. First, WeldNumpy exploits lazy evaluation instead

[3] https://www.weld.rs/weldnumpy/

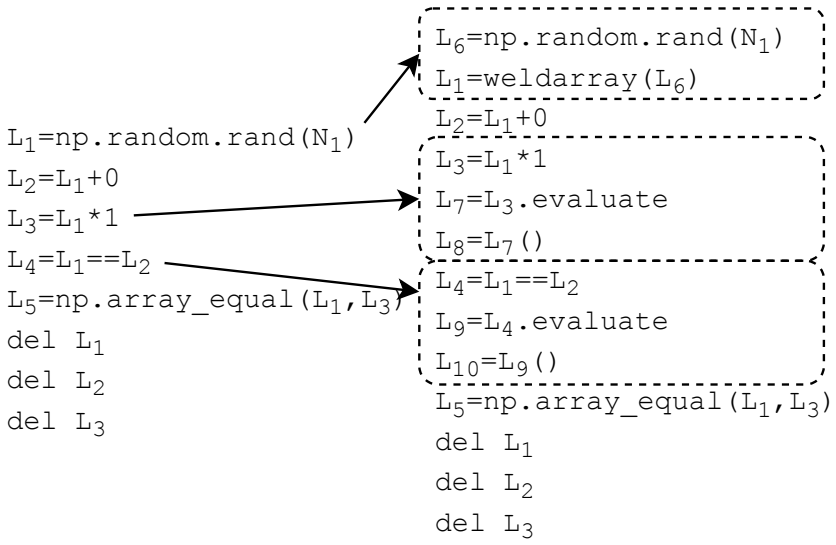


Figure 3.10: Translating NumPy to WeldNumpy

of eager evaluation, which is used in NumPy. Second, WeldNumpy implements its APIs using Weld IR [Pal+17], an intermediate representation designed for parallel data processing. Through lazy evaluation, the IR fragments of invoked APIs are combined into an IR program. During a true evaluation, the IR program is compiled and optimized for native hardware. Some major optimization techniques are loop fusion, loop tiling, and vectorization. WeldNumpy provides `weldarray`, a subclass of NumPy's `ndarray`. Thus, after an `ndarray` is converted to a `weldarray`, the new object supports most NumPy operations and enjoys improved performance.

However, as WeldNumpy is lazily evaluated, it requires users to explicitly call `evaluate()` when necessary. The `evaluate()` method should not be invoked too often; otherwise, the WeldNumpy runtime misses optimization opportunities and introduces overheads of compiling the Weld IR. Neither should it be too late as that would cause errors. Thus, a NumPy-to-WeldNumpy translator needs to figure out the appropriate positions to insert `evaluate()`.

The evaluating positions can be located by identifying *exposed lazy variables*. A variable is exposed if it is used beyond the DSL's APIs, which means the true value of the variable may be required, or it is alive at the end of the collected lazy IR, which allows potential external usage of the variable during subsequent execution. When a variable is exposed but lazy, it should be explicitly evaluated. Such variables can be identified within an one-pass scan of the lazy IR. The translator can thence insert `evaluate()` for these variables.

Figure 3.10 shows an example of translating NumPy to WeldNumpy. The translated IR first converts L_1, an `ndarray`, to a `weldarray`, such that L_2, L_3, and L_4 enjoy WeldNumpy's optimization. However, `np.array_equal()` is not supported by WeldNumpy; thus, operand L_3 has to be evaluated before being passed. While L_4 is explicitly evaluated because of potential exposure, L_2 remains lazy, since it is deleted and has no external use.

Such a translator leverages the laziness analysis enabled by Cunctator. It might be tempting to think that the translation could be done through a compiler without Cunctator. Note that that compiler would have to face the laziness analysis problem as Cunctator tackles; if it ignores that, its replacement of an eagerly evaluated NumPy API with a lazy evaluated WeldNumpy could cause errors. Doing the laziness analysis is difficult for a compiler for the many challenges (e.g., Python complexities, API-host interplay) mentioned in the introduction section.

3.6.4 From Pandas to Spark

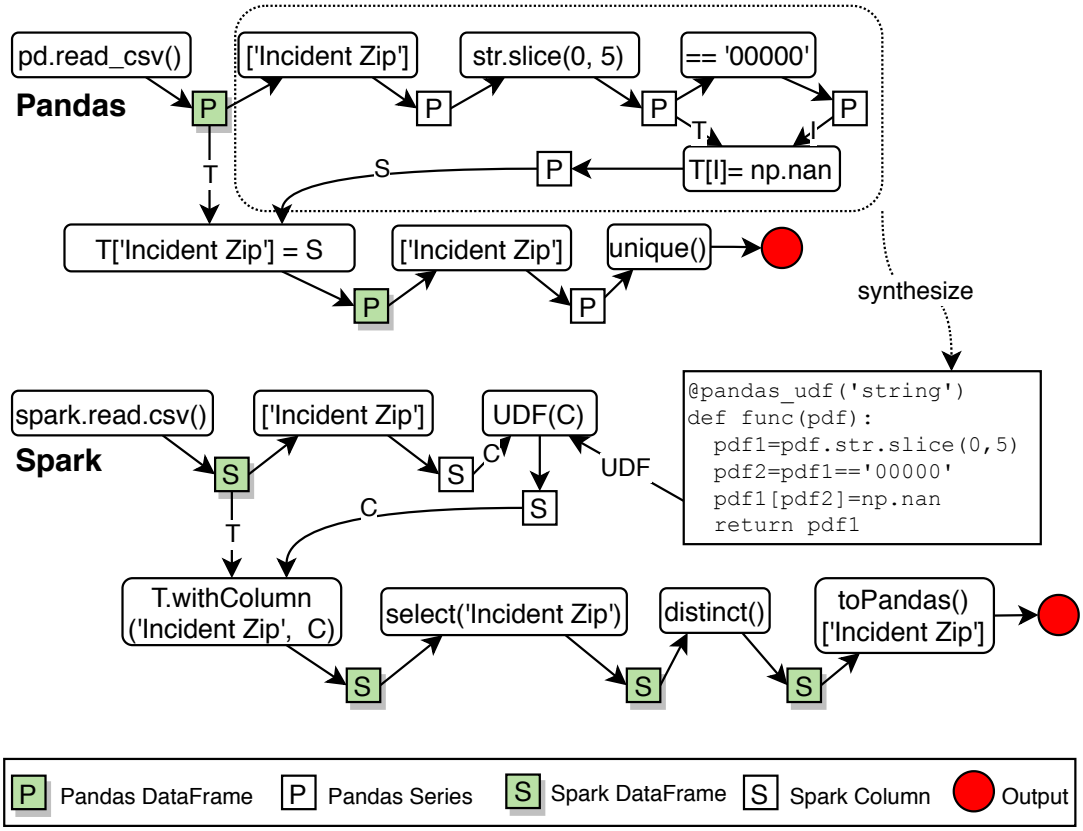


Figure 3.11: Pandas to Spark

Both Pandas and Spark provide a class called `DataFrame`. They both represent logical tables, which have named and typed columns. While Pandas' operations in `DataFrame` are eagerly evaluated, most of Spark's `DataFrame` methods are lazily evaluated. During a true evaluation, Spark employs a code generation technique [Neu11] to compile an operation sequence. Such technique renders the Spark DataFrame API a performant replacement of Pandas. In addition, Spark has native support for Pandas, including Pandas UDF [4], by which a user can apply Pandas operations to a Spark `Column`. Spark also contains type casting APIs that convert between Spark `DataFrame` and Pandas `DataFrame`. These features offers conveniences to translation of a Pandas program to a Spark program.

Similar to NumPy to WeldNumpy, the laziness analysis by Cunctator puts down the basis for the development of an automatic Pandas-to-Spark translator. Our prototype focuses on a common use pattern of Pandas: A program first loads a file as a `DataFrame`, then performs some operations on it, and finally outputs the result. In such a pattern, only one `DataFrame` object is involved, and no Pandas `DataFrame` object or `Series` (typically represents a column) object is exposed, so all instances of the two types only participate in Pandas operations. When such a pattern is matched, the translator tries to optimize it.

During the translation, the Pandas file loading function is replaced by a counterpart in Spark, thus creating a Spark `DataFrame`. Correspondingly, the `Series` objects selected from Pandas' `DataFrame` become Spark `Column` objects. If there is a sequence of operations on a `Series` that outputs another `Series`, the sequence is synthesized into a Pandas UDF for Spark, which is applied to the corresponding Spark `Column`. If a `Series` is assigned to the Pandas `DataFrame`, the corresponding `Column` is assigned to the Spark `DataFrame` as well. When an operation on a `Series` returns an object other than a `Series`, if the operation (e.g., `unique()`) has a counterpart in Spark, the `Column` is applied to the corresponding Spark operation, and then the result is converted to the expected type; otherwise (e.g., `diff()`), the `Column` is selected and converted to a `Series` before applying the operation. Figure 3.11 illustrates the translation for a Pandas program collected from the Pandas Cookbook [5].

3.7 API Redirection

If a DSL's runtime needs to leverage Cunctator to perform optimization, the optimizer developer needs to redirect the APIs in the DSL through renaming and rewriting. With the Cunctator framework, the process is made simple. For example, to redirect `numpy.add` in NumPy's

[4] https://databricks.com/blog/2017/10/30/introducing-vectorized-udfs-for-pyspark.html
[5] https://nbviewer.jupyter.org/github/jvns/pandas-cookbook/tree/v0.1/cookbook/

runtime, current implementation of `numpy.add` could be renamed to `numpy._add`; then, a new implementation of `numpy.add` will just record API calls as lazy IR instructions and returns a lazy object as shown in Figure 3.6.

To simplify the process, Cunctator offers some utilities. For the aforementioned example, what the optimizer developer needs to write to put the following into the module *numpy*:

```
def add(*args, **kwargs):
  return lazy_call("numpy.add", numpy._add, args, kwargs,
    kwargsToUpdate={"out"})
```

Method `lazy_call` is the utility interface that Cunctator offers. Its first argument is for the annotation field of a `call` instruction (see Section 3.5). The argument `kwargsToUpdate` specifies that `numpy._add` is going to update only its argument `out` (if there is one). The call to `lazy_call` in this example will essentially materialize the method shown in Figure 3.6.

3.8 Efforts in Applying Cunctator

There is some work needed from the library developers. This work needs to be done only once for a given library; the results can benefit all programs using that library. This one-time work includes: (1) redirecting some APIs that are important for performance (other APIs can be left alone, which will be treated in the same way as host Python code is); (2) supporting MIN-watch for some common types; and (3) implementing optimization passes. Table 3.1 shows our prototype optimizers' summary in this work. For a common programmer that uses a library, the only change she needs to make to her code is to insert one or several lines of code to import the optimizer.

We initially considered automatic library transformations, but found that it was difficult to do for the complexities of Python. It is, for instance, often impossible for static code analysis to tell whether an argument is subject to modifications, due to dynamic types, aliases, higher-level functions, and inter-procedural complexities. The design choice made in Cunctator is a choice for practicability.

3.9 Evaluation

In this section, we conduct a series of experiments to (1) demonstrate the usefulness of the four optimizations (Section 3.6) enabled by Cunctator, and (2) measure the runtime overhead

Table 3.1: Summary of optimizers

Optimizer	#APIs*	Supported types†	Opt pass LoC‡
NumPy	45	`ndarray`, `dtype`	50 (Section 3.6.1) 93 (Section 3.6.3)
Spark	24	`RDD`, `StorageLevel`	201 (Section 3.6.2)
Pandas	28	`DataFrame`, `Series`	436 (Section 3.6.4)

* The number of redirected APIs.

† The types that support dependent discovery.

‡ Lines of code for implementing the optimization passes described in Section 3.6.

of Cunctator. Time usage is collected by the *timeit* command of jupyter [6], which adaptively chooses a number of repetitions in favor of timing accuracy. Peak memory usage is collected by *memit* command extended by *memory-profiler* [7], which profiles a program's memory usage line by line. The test platform for NumPy and Pandas is a Linux machine with Intel Xeon Silver 4114 CPUs. Spark programs run on a cluster of eight Linux machines with AMD Opteron 6128 CPUs.

3.9.1 Optimizers

We collect 15 programs for the experiments that are relevant to the example optimizations described in the previous section; five for each of the three packages (NumPy, Spark, Pandas). Thirteen of them were collected from GitHub; the other two were the examples used in the earlier sections of this chapter—we included them to show the performance benefits for the described optimizations. Table 3.2 shows the descriptions and inputs of all benchmarks. Their source code can be found in the Docker image of Cunctator [8]. Figure 3.12 shows the speedups in different optimizer settings. Detailed results are presented in Table 3.3. Each program set is discussed separately in following subsections.

3.9.1.1 NumPy

The temporary variable reducer (abbr. *reduceTmp*) accelerates all benchmarks, with speedups ranging from 1.19X to 1.54X. The highest speedup is achieved on P1, because its operations are easy to compute and hence the cost of temporary variable is prominent. Besides time benefits,

[6] https://jupyter.org/

[7] https://pypi.org/project/memory-profiler/

[8] https://github.com/sangongs/Cunctator_docker

Table 3.2: Descriptions of collected benchmarks

	Description	Input
	NumPy	
P1	Program in Figure 3.1a	vectors of size 10^9
P2	Compute vibration energy	vectors of size $5{\times}10^8$
P3	Find least-squares solution	vectors of size $5{\times}10^8$
P4	Find log-likelihood of $\mathcal{N}(\mu, \sigma^2)$	vectors of size $5{\times}10^8$
P5	Compute Black-Scholes model	vectors of size 10^8
	Spark	
P1	Program in Listing 3.1	text file of 90MB
P2	Demultiplex a file to multiple files	xml file of 244MB
P3	Transform data format	json file of 62MB
P4	Intersect IDs in two tables	two csv files of 34MB
P5	Find counts of different words	text file of 460MB
	Pandas	
P1	Find names of median occurrence	csv file of 273MB
P2	Find top complaints	csv file of 526MB
P3	Find ratios of noise complaints	csv file of 526MB
P4	Find unique zip codes after data cleaning	csv file of 526MB
P5	Find top occupations wrt. male ratio	csv file of 240MB

Table 3.3: Benchmark results

	P1	P2	P3	P4	P5
NumPy time usage (mean ± std. dev.)					
baseline	12.5s±2.95ms	41.1s±75.4ms	27.4s±5.51ms	43.2s±57.6ms	39.1s±23ms
reduceTmp	8.14s±2.14ms	34s±15.3ms	22.9s±105ms	33.7s±26.9ms	32.9s±22.7ms
Weld 1T	6.38s±34.1ms	39.1s±87.6ms	21.8s±45ms	42.7s±144ms	22.6s±28.3ms
Weld 10T	995ms±9.24ms	27s±142ms	17.4s±60.3ms	15.6s±44.4ms	9.94s±35ms
NumPy peak memory usage (MB)					
baseline	38181	19131	19130	15316	9213
reduceTmp	30577	11503	15317	3873	6251
Spark time usage (mean ± std. dev.)					
baseline	31.9s±123ms	82s±698ms	38.5s±116ms	20.9s±36ms	49.1s±272ms
w/o opt	32.1s±215ms	81s±639ms	38.5s±178ms	21.1s±75.8ms	49.2s±392ms
optimized	17.1s±93.4ms	48.8s±348ms	28.8s±66.8ms	20.3s±317ms	47s±226ms
Pandas time usage (mean ± std. dev.)					
baseline	6.03s±50.3ms	9.65s±21.8ms	9.8s±13.4ms	9.72s±40.7ms	7.7s±37.4ms
Spark 1T	17.1s±100ms	5.51s±152ms	3.01s±127ms	5.76s±101ms	7.45s±241ms
Spark 10T	2.92s±203ms	951ms±51ms	690ms±26.9ms	1.3s±145ms	1.29s±59ms

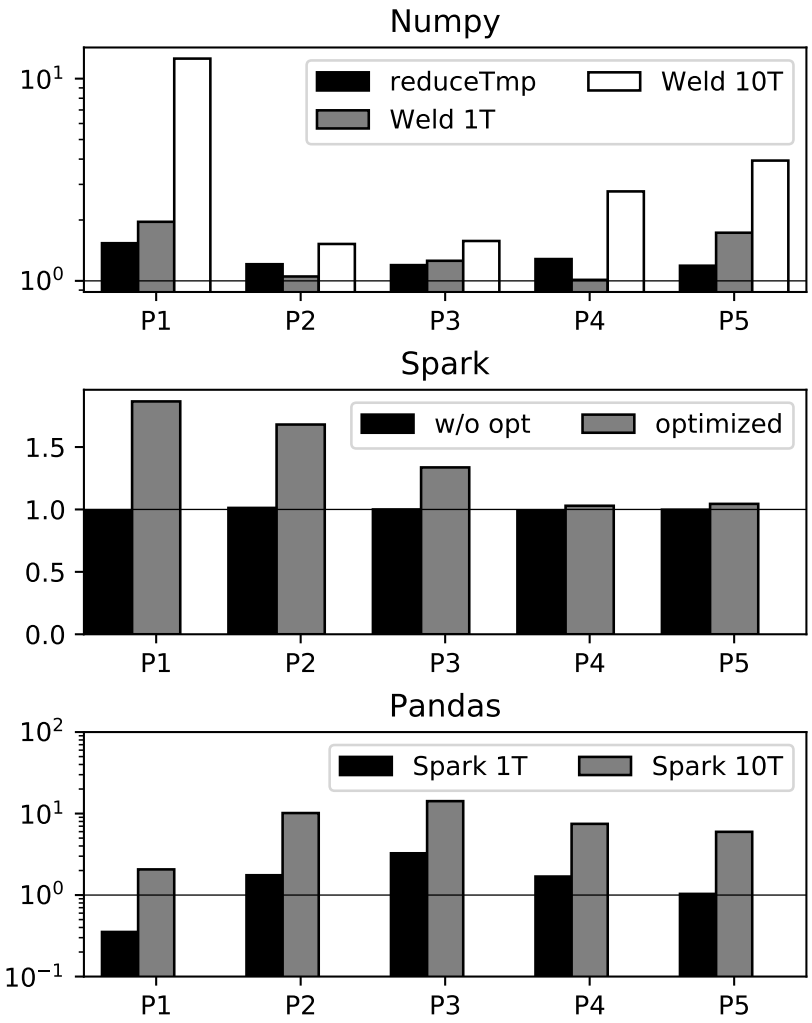


Figure 3.12: Speedups

reduceTmp also reduces peak memory usage. P4 highlights the reduction with a rate of 75%. The high rate is because of pipelined operations, which means each temporary variable is only used one time and then discarded.

For WeldNumpy converter, we test it with one thread (abbr. *Weld 1T*) and ten threads (abbr. *Weld 10T*) separately. *Weld 1T* shows speedups ranging from 1.01X to 1.95X. Because WeldNumpy currently supports only a limited number of NumPy APIs, for unsupported APIs, it needs to transform data from Weld format to NumPy format to perform the operations, and if necessary, the results need to be transformed back. As WeldNumpy evolves to support more APIs, *Weld 1T* is going to perform better. Moreover, with ten threads, WeldNumpy achieves significant speedups up to 12.5X. Note that Weld has built-in support for multi-threading but NumPy does not.

3.9.1.2 Spark

The Spark optimizer shows speedups ranging from 1.03X to 1.87X. Among the benchmarks, P1 and P2 lack `cache()`; P3 and P5 have unnecessary `cache()`; P4 has a `cache()` operation at a useless location, while the place that needs `cache()` does not have one. Our optimizer fixes them all. It adds `cache()` to P1 and P2, removes `cache()` from P3 and P5, and corrects P4 by removing the unnecessary `cache()` and adding one at the appropriate place.

In addition, we test the benchmarks with Cunctator enabled but optimizing pass disabled(abbr. `w/o opt`). The results show no performance degradation. This confirms that MIN-watch has almost no overhead for non-watched objects, as PySpark programs typically invoke user defined functions written in Python frequently.

3.9.1.3 Pandas

The Pandas-to-Spark optimizer is tested with one Spark thread (abbr. `Spark 1T`) and ten Spark threads (abbr. `Spark 10T`). Note that Spark supports multi-threading but Pandas does not. `Spark 1T` shows speedups on three programs. This is impressive because, while Pandas enjoys the high performance of SIMD instructions, Spark's query compiler emits Java bytecode. The slowdown on P1 is dominated by Spark's CSV loader, which performs much worse than Pandas' loader in this case. Nevertheless, `Spark 10T` enjoys speedups as high as 14.2X.

Table 3.4: Overhead (percentage of 10s program runs)

		CPS				
		50	500	1000	2000	10000
Threshold	50	0	0.85	0.85	1.7	0
	500	0.35	3.05	1.05	1.8	0
	1000	**0.25**	**2.35**	**2.25**	**1.7**	**0**
	2000	0.15	2.15	2.05	2.5	0.1
	10000	0.15	1.25	1.75	2.9	11.8

3.9.2 Overheads

For programs that cannot be optimized, a major concern is the overhead, which is highly related to the number of lazy IR instructions recorded. To investigate the overhead in different cases, we design an adversarial case for stress-testing:

```
def cps_simulator(M, N):
  for i in range(M):
    numpy.ones(N)
```

The program calls M times of `numpy.ones(N)`, which initializes a vector of size N. By tuning M and N, we can control the number of calls per second (CPS) and the total run time. We then combine some representative values of CPS and overhead control thresholds (see Section 3.4.3). For each combination, we run a ten-second experiment with Cunctator. By subtracting the results with the corresponding baseline results, we obtain an overhead matrix, shown as Table 3.4.

The overhead increases when CPS increases. When CPS exceeds the threshold, Cunctator disables itself for the later part of the run; the overhead drops. For the default threshold (1000), the worst overhead is 2.35%, which happens in the extreme case where there are 1000 function calls per second. In practice, a program is unlikely to have a stable CPS rate close to the threshold, thus the overhead is much lower. In addition, it is worth noting that Cunctator is mainly implemented in Python, except for the MIN-watch. If we reimplement some critical components in C, such as the lazy IR evaluator, a lower overhead is expected.

It is worth noting that, in the domains that we explored, the number of relatives per object is few, hence our benchmarks bear little overhead of finding relatives. For example, a NumPy array usually has no relative if its buffer belongs to itself, or only one relative if its buffer is from another object. For domains where deeply nested objects are common, the overhead control threshold can be adjusted to fit the need of the domains.

3.9.3 Threats to Validity

Cunctator is evaluated based on Python 3.7.3, NumPy 1.17.0, WeldNumpy 0.0.1, Pandas 0.25.0, and Spark 2.4.3. The APIs and implementation of these software packages may change after new versions are released. Thus the new releases may invalidate our optimization techniques and evaluation results. Nevertheless, new patterns of API misuses related to these new releases are likely to appear. Unless the new versions employ a technique similar to BELE, Cunctator can be leveraged to optimize the new patterns.

The soundness of a Cunctator-based optimizer relies on the correctness of the API knowledge provided by the optimizer's developer. Such knowledge includes how to discover the relatives of an object, which arguments of an API could be updated during the API call, and how to apply optimization passes onto the recorded lazy IR. If any of the knowledge is incorrect, programs optimized by Cunctator may yield unexpected results.

3.10 Related Work

Lazy evaluation has been studied extensively in functional programming [HM76; Joh84; BHY88; Lau93; Hud+07]. Scala [OR14] provides a `lazy` keyword to express the call-by-need semantics of a variable. However, Scala does not manage the potential side-effect of a thunk, the expression bound to the lazy variable; thus, the correctness of lazy evaluation relies on the programmer. Many hosted DSLs (e.g., Spark [Zah+10] and TensorFlow [Aba+16]) employ lazy evaluation; their limitations have been discussed in Section 3.1.

There are some studies on optimizing DSLs. Weld [Pal+17] and its limitations have been discussed and compared with. Delite [Suj+14] is a framework for developing Scala-hosted DSLs by leveraging generative programming [CE00]. Similarly to Cunctator, it lazily evaluates DSL operations and logs them as a form of IR, which will be optimized and executed at a certain point in time. However, Delite provides no mechanism to handle the dependencies between DSL operations and their host code.

There are several earlier studies (e.g., telescoping languages [Ken+05], Broadway [GL05]) that try to use manual annotations of libraries to help optimizations. They give no systematic considerations of the host-API dynamic dependencies. Numba [LPS15] is a JIT compiler of Python that targets optimizing manipulations of `ndarray` in NumPy. AutoGraph [Mol+19] employs static code conversion and generative programming to transform PyTorch-style programs to TensorFlow-style programs. All these methods and tools offer a closed set of optimization tech-

niques for specific program semantics. Cunctator does not include any optimization technique but provides a general framework to simplify the creation of a DSL optimizer.

Finally, the NumPy optimizer presented in Section 3.6.1 replaces list copies with in-place updates. In this sense, it is similar to deforestation, an optimization technique usually used in programming environments where referential transparency ends up being very costly[Wad88; KPP11; FSFP19; UM19].

3.11 Conclusion

This chapter introduces the concept of BELE, and describes MIN-watch, the first efficient runtime monitoring method tailored to data dependence analysis between host code and APIs for BELE. The chapter demonstrates the usefulness of Cunctator in enabling four optimizations that are not supported by existing frameworks, giving 1.03-14.2X speedups. While Cunctator targets Python-hosted DSLs, we believe the potentially applicability of the techniques goes much beyond Python.

CHAPTER

4

COMPOSITIONAL LAZY INDUCTIVE SYNTHESIS

4.1 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 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 [Ram+17a; RP19; GPR20] has proposed techniques for automatically translating UDFs to SQL expressions. For example, Froid [Ram+17a] and BlackMagic [RP19] translate *loop-free* user-defined functions to relational algebra expressions, and Aggify [GPR20] 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 chapter, 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 [Alu+13] 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 4.3b). 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.

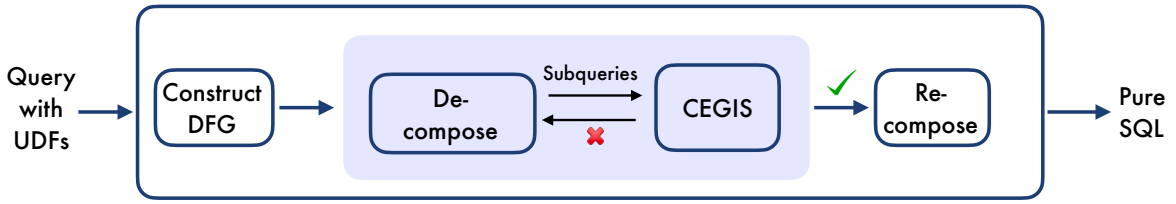


Figure 4.1: Schematic overview of our approach

Our method overcomes this challenge using a new paradigm that we dub *lazy inductive synthesis* that is illustrated schematically in Figure 4.1. 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 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 work 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 chapter is organized as follows: Section 4.2 gives a high-level overview of our method by running through a motivating example, and Section 4.3 presents our formal problem definition. Next, Section 4.4 introduces some background knowledge on dataflow graphs, and Section 4.5 presents our core synthesis algorithm. The following three sections (Sec. 4.6-4.8) describe our implementation and evaluation and discuss related work. Finally, we discuss limitations and conclude.

4.2 Overview

In this section, we give an overview of our method with the aid of the motivating example shown in Figure 4.2 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 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.

[1]Tested on randomly generated 10 million rows on a Linux machine with an Intel Core i5-4570 CPU (four 3.2GHz cores).

```
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”)
```

Figure 4.2: A Spark query containing a UDF

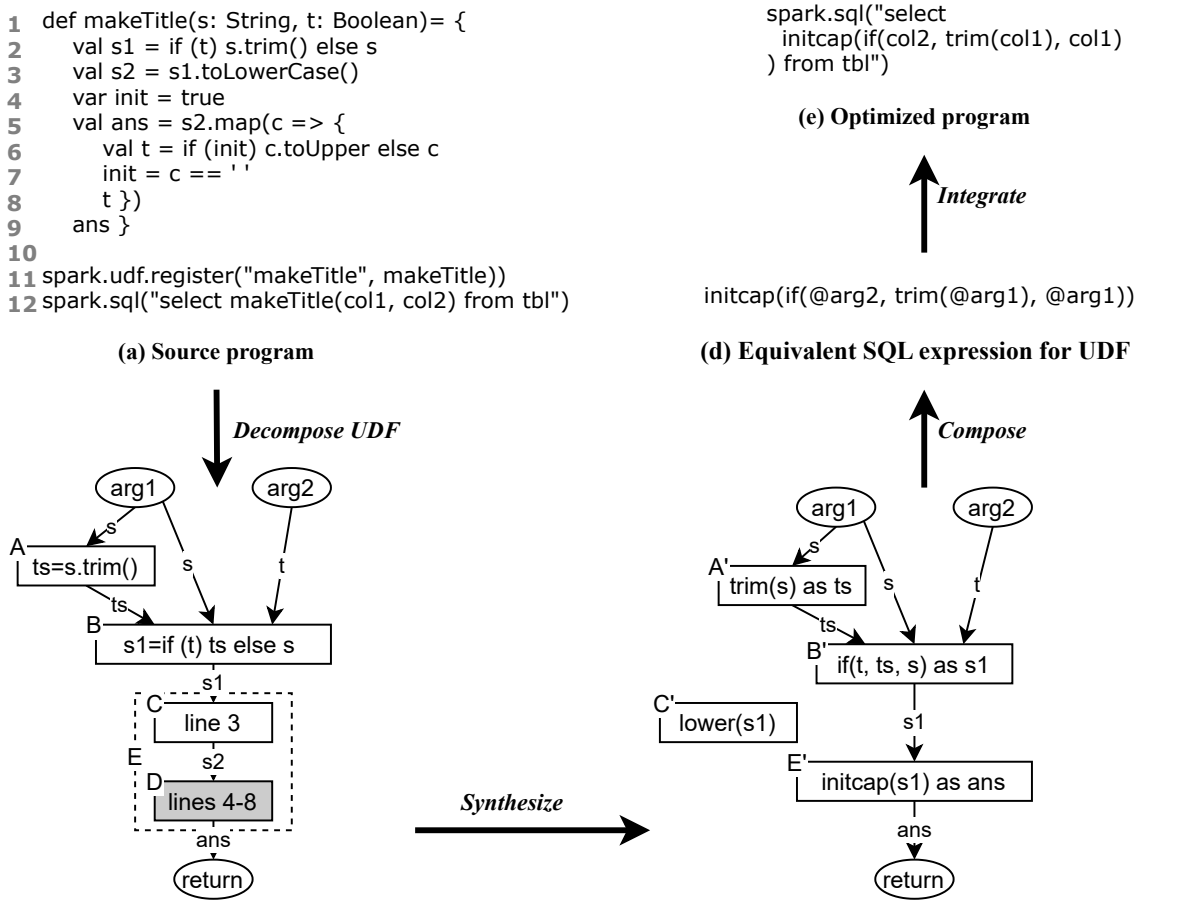


Figure 4.3: Illustration of our approach on running example

Our method starts by constructing the UDF's dataflow graph (DFG) (see Figure 4.3b). 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 4.3c 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 4.3b). 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 4.3d with the help of the DFG from Figure 4.3c. 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 [Ram+17a], our method can utilize built-in SQL functions such as `trim` and `initCap`, and it can also handle higher-order combinators like `map`.

4.3 Problem Definition

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

Program P	$\rightarrow$	$D_1; ...D_n; Q$
UDF Defn. D	$\rightarrow$	`def` f$(a_1, ..., a_n) = \langle$*ScalaExpr*$\rangle$
SQL Query Q	$\rightarrow$	σ | `with` $V_1; \ldots; V_n$ σ
View V	$\rightarrow$	TblName `as` (σ)
Select σ	$\rightarrow$	`select` Π `from` $\mathsf{T}_1, ..., \mathsf{T}_n$ [`where` E] [`group by` Π] [`order by` Π]
Projection Π	$\rightarrow$	$\mathsf{UDF}_\mathsf{m}(C_1, ..., C_n) \mid C_1, ..., C_n$
Column C	$\rightarrow$	$E \mid E$ `as` ColName
Expression E	$\rightarrow$	Const | T.ColName | $F(E_1, ..., E_n)$
Function F	$\rightarrow$	$\mathsf{UDF}_\mathsf{s} \mid f \in$ BuiltInFuncs

Figure 4.4: The grammar of our source language.

As shown in Figure 4.4, 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 UDF_s to represent a UDF with a scalar return value and 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 4.4, 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.*

In this work, 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 $D \simeq \lambda\overline{a}.\ \Pi$.*

[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.

4.4 Background on Dataflow Graphs

In the rest of the chapter, 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 $L(n) \subseteq \mathsf{Stmts}(f)$*

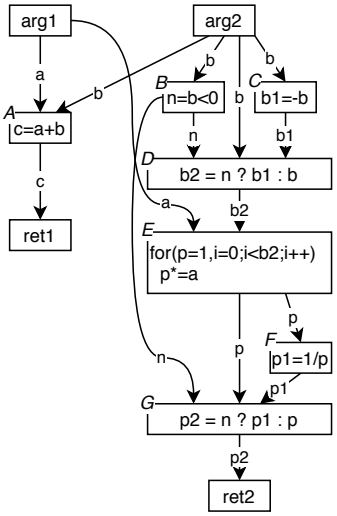
Semantically, an edge (n, n', v) indicates that the computation performed in node n' depends on the computation from node n through variable v (i.e., n' reads from variable v that is modified in n). As expected, *argument nodes* $n \in N_A$ do not have incoming edges, and *return nodes* $n \in N_R$ do 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 1. *Figure 4.5a shows a function called* `addPower`*, and Figure 4.5b shows a dataflow graph for this function. Since the UDF takes two arguments, the DFG has two argument nodes labeled $arg1$ 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 4.5c shows another valid DFG representation of the same* `addPower` *function.*

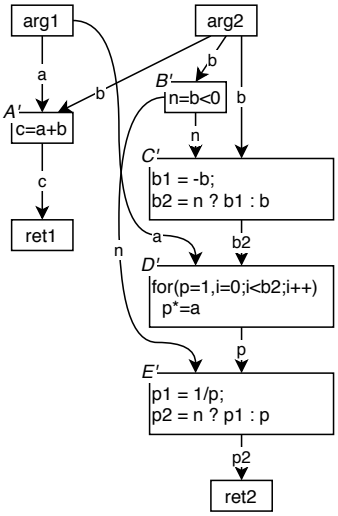
Because a program has many possible DFG representations, we define a *refinement relation* between two DFGs as follows:

```
1  function addPower(a, b) {
2     c = a + b;
3     n = b < 0;
4     if(n) b1 = -b;
5     b2 = n ? b1 : b;
6     for (p=1,i=0;i<b2;i++)
7        p *= a;
8     if(n) p1 = 1/p;
9     p2 = n ? p1 : p;
10    return c, p2;
11 }
```

(a) An example UDF



(b) Dataflow graph G



(c) Dataflow graph G'

Figure 4.5: An example UDF and its two possible DFGs

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:*

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

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 2. *For the two DFGs in Figure 4.5, 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 4.5.1, if $G \preceq G'$, our approach prefers G over G'.

Algorithm 2 Top-level synthesis algorithm

```
1:  function UDF2SQL(F)
2:      Input: A user-defined function F
3:      W ← {ConstructDFG(F)}
4:      Ω ← ∅
5:      while W ≠ ∅ do
6:          G ← PickMostPromising(W, ⪯)
7:          (E, n) ← SynthFromDecomp(G, Ω)
8:          if E ≠ null then return E
9:          W ← W ∪ CoarsenDecomp(G, n)
10:     return null
```

4.5 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.

4.5.1 Top-Level Algorithm

Algorithm 2 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 [Cyt+91]. 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 3 Compositional synthesis from DFG

```
1:  function SynthFromDecomp(G, Ω)
2:      Input: DFG G = (N_A, N_S, N_R, E, L)
3:      Input: Mapping Ω containing partial synthesis results
4:      for each n ∈ N_S do
5:          S ← L(n)
6:          if S ∉ Domain(Ω) then
7:              Ω[S] ← CEGIS(S, InLabels(n), OutLabels(n))
8:          if Ω[S] = null then return (null, n)
9:      E ← CodeGen(G, Ω)
10:     return (E, 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 2.2), 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:

$$\forall G' \in W.\, G \neq G' \Rightarrow G' \not\preceq G$$

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.

4.5.2 Synthesis from Dataflow Graph

Our algorithm for synthesizing SQL code for a given decomposition (i.e., dataflow graph) is presented in Algorithm 3. 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.

4.5.2.1 CEGIS Instantiation

Figure 4.6 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 [Mar+19] 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 [CKL04] 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 3. *Consider the following code snippet in a DFG node:*

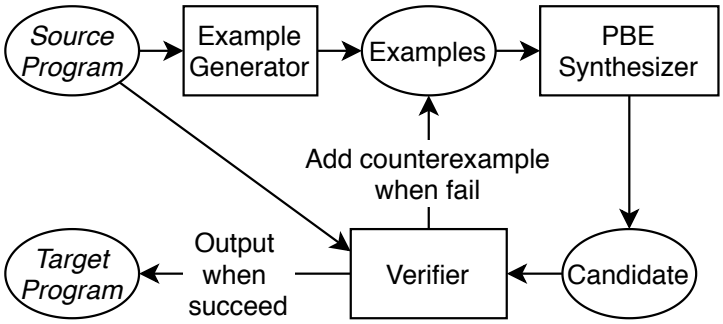


Figure 4.6: Our instantiation of the CEGIS paradigm

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

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*:*

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

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* $\text{“}b\text{”}$*. We then execute the reference implementation on this input and obtain the following examples:*

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

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:*

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

Algorithm 4 Code Generation for Each DFG Node

```
1:  function CodeGen(n, Ω)
2:      Input: A DFG node n
3:      Input: Synthesis results Ω
4:      E ← Ω(n)
5:      for each e ∈ InEdges(n) do
6:          (n′, v) ← (Source(e), Label(e))
7:          if IsArg(n′) then
8:              subExps[v] ← L(n′)
9:          else
10:             parentRes ← CodeGen(n′, Ω)
11:             subExps[v] ← parentRes[v]
12:     return map(E, λt.(Var(t), replace(Def(t), subExps)))
```

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.*

4.5.2.2 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 4 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 $(e_1\ \mathsf{as}\ t_1, \ldots,\ e_n\ \mathsf{as}\ t_n)$, where each e_i refers to intermediate columns defined in its parent block [3]. Line 12 of Algorithm 4 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 4. *Consider the DFG and synthesis results shown in Figure 4.3(c). The* CodeGen *procedure generates SQL expressions for each node in the DFG as follows.*

- *For node* A'*, we generate the SQL expression* $\mathsf{trim}(\mathsf{arg1})$.

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

- *For node B', we generate the expression* $\mathsf{if}(\mathsf{arg2}, \mathsf{trim}(\mathsf{arg1}), \mathsf{arg1})$*) where the second argument is obtained through a recursive call.*

- *For node E', we generate the expression:*

$$\mathsf{initcap}(\mathsf{if}(\mathsf{arg2}, \mathsf{trim}(\mathsf{arg1}), \mathsf{arg1}))$$

This is also the return value of CODEGEN *for the whole DFG.*

4.5.3 Coarsening the Decomposition

Recall that our main synthesis procedure (Algorithm 2) performs lazy inductive synthesis by gradually coarsening the initial decomposition. In this section, we explain the COARSENDECOMP procedure, shown in Algorithm 5, 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:

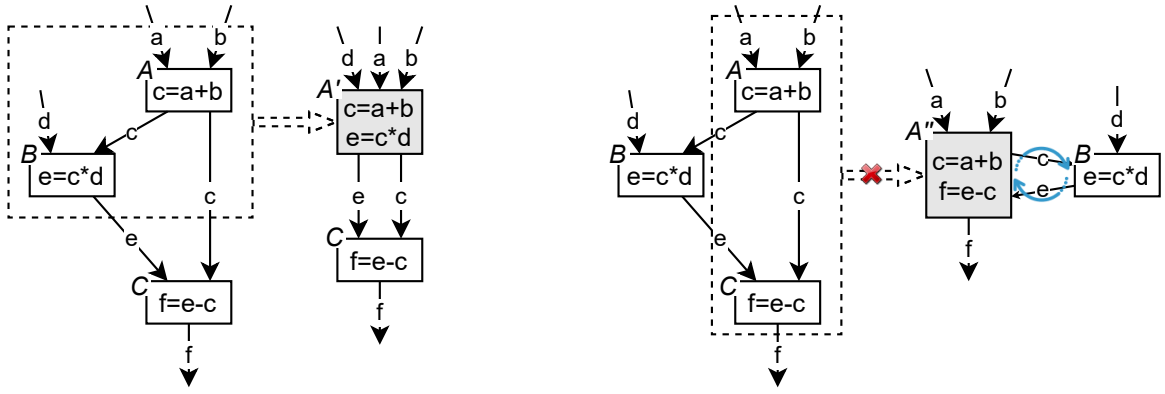
- Generate a new node m with label $L(n); L(n')$
- The incoming (resp. outgoing) edges of m become the incoming (resp. outgoing) edges of both n and n' (except for those edges between n, n')
- G' includes m and all nodes of G except n, n'
- G' includes all edges of G except those involving n or n' as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

Note that line 9 of Algorithm 5 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. *Figure 4.7a shows a new DFG obtained by merging nodes A and B in the left DFG. Figure 4.7b shows the graph obtained by merging nodes A and C. However, this graph is rejected since it is cyclic and therefore an* invalid *DFG.*

Algorithm 5 Coarsen decomposition

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



(a) A valid merge

(b) An invalid merge due to cyclic edges

Figure 4.7: Merging two nodes

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 4.2. Of course, the bigger the code fragment, 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.

4.5.4 Complexity Analysis

We first analyze the time complexity of the CEGIS procedure in Algorithm 3. 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.

4.6 Implementation and Optimizations

We have implemented our proposed method in a tool called CLIS targeting the Spark framework [Zah+10]. 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 4.5.

4.6.1 Parallelization

The compositional synthesis technique of Algorithm 3 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.

4.6.2 Type-Based Pruning

Our implementation performs one important optimization over the algorithm from Section 4.5. 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.

4.6.3 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.

4.6.4 PBE Back-End

Our implementation leverages the Trinity programming-by-example framework [Mar+19] 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 4.4, 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 [AC18]. 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.

4.6.5 Checking Equivalence

We leverage the CBMC [CKL04] 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.

4.7 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*?)

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?

Table 4.1: 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

4.7.1 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 4.1 gives statistics about these UDFs in terms of their AST size, number of lines of Scala code, and whether they contain loops.

4.7.2 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 4.8 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.

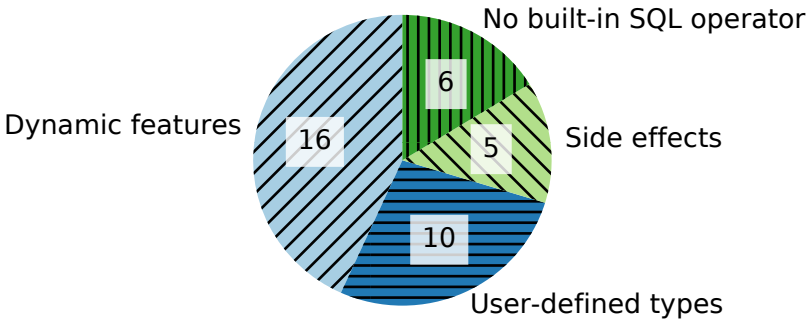


Figure 4.8: Types of UDFs that do not have SQL equivalent

Table 4.2: 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.

4.7.3 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 4.2, 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 4.2, 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.

4.7.4 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 [AC18]. 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 [SL+06].

The results of this comparison are included in Figure 4.9. 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 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$\times$ more benchmarks than the non-compositional (ablated) version and solves $2.8\times$ 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 chapter is effective and that our DFG-based decomposition is highly beneficial in practice.

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

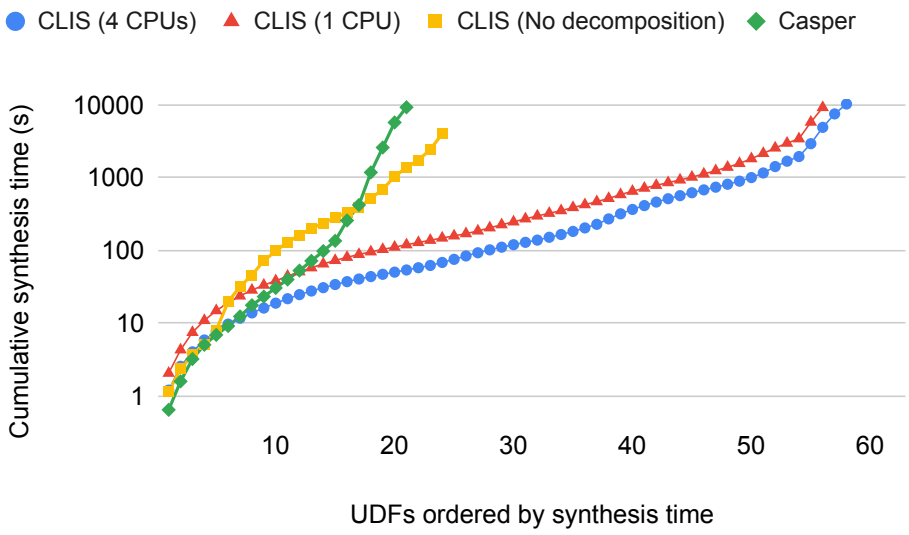


Figure 4.9: Comparing cumulative synthesis time of different settings and techniques

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.

4.7.5 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.

4.7.5.1 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 4.3. 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

Table 4.3: 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 ×)

benchmarks, we observe a speed-up exceeding $2\times$. Overall, these results confirm the hypothesis that converting UDFs to SQL expressions is quite beneficial in terms of performance.

4.7.5.2 TPC-H Workloads

In this section, we evaluate our method on TPC-H workloads [TPC05]. Since the original TPC-H queries do not contain UDFs, we rewrite the queries by replacing expressions with UDFs as done in previous work [Ram+17b]. While the queries 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 [Ram+17b], 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 4.10 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 $2.2\times$ compared to the version containing UDFs. Furthermore, the queries optimized using CLIS execute as fast as the original TPC-H queries.

4.7.5.3 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", “read", “groupByKey”). Among

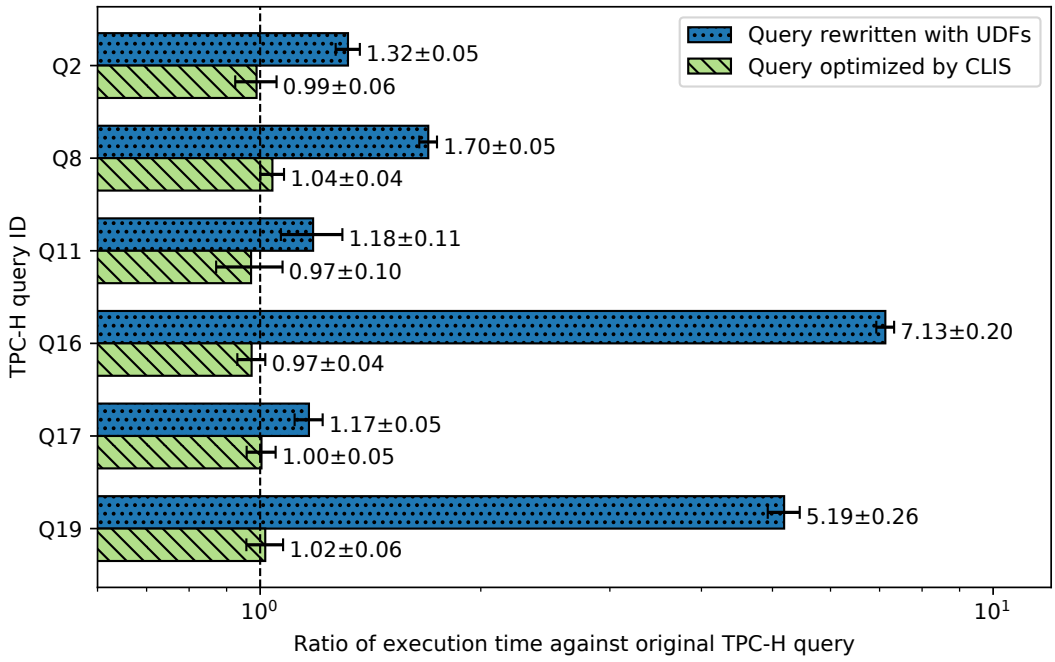


Figure 4.10: Improvement of TPC-H queries with UDFs. Since the original TPC-H queries do not contain UDFs, we rewrite the queries by replacing expressions with UDFs as done in previous work [Ram+17b]. Error bars are 95% confidence intervals.

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 4.4, 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 4.5. The workloads are input files with randomly generated contents. As shown in Figure 4.11, converting this single UDF to SQL improves end-to-end running time of the application by 30-50% depending on the setting.

Table 4.4: 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 4.5: 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

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, 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 4.5. For this application, we observe much larger speed-ups, ranging from $2.2\times$ to $3.1\times$, depending on which setting is used. These larger

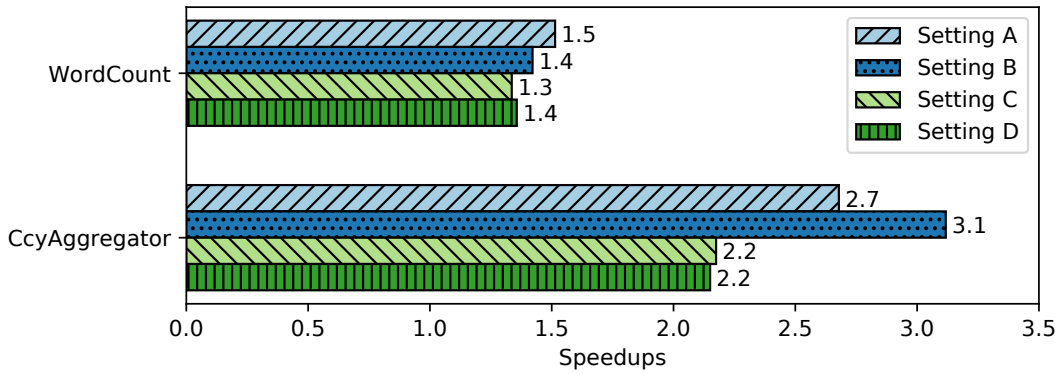


Figure 4.11: End-to-end improvement for real applications

Table 4.6: 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 [Ram+17a], UDF Decorrelation [Sim+14], DBridge [Ema+17], QBS [CSLM13], CASPER [AC18]
† 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.

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 $3.5\times$ speed-up by converting UDFs to equivalent SQL expressions. At query level, we see $1.2 - 7.4\times$ 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 $1.3\times$ to $3.1\times$ for two representative Spark applications and four different test settings.

4.7.6 Expressiveness Compared to Prior Work

Table 4.6 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 [Ram+17a] 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 4.6 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 [CSLM13] and CASPER [AC18], where the latter can be viewed as an improvement over the former. As mentioned earlier, we have re-implemented CASPER's

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

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.

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.

4.8 Related Work

Among prior techniques for translating code to SQL, the most related one to ours is the Froid tool [Ram+17a], which is now a feature of the Microsoft SQL Server [RP19] 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 [Sim+14] to decorrelate UDFs, but this technique can only handle so-called cursor loops that iterate over query results.

Another closely related tool is QBS [CSLM13], which also uses the CEGIS paradigm to optimize ORM applications. The observation behind this work (as well as other papers [WC07; WIC08; Ema+16; Ema+17] 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 4.6, CLIS also uses the same optimization proposed in CASPER; however, as demonstrated empirically in Section 4.7, that technique is not sufficient to solve many of our benchmarks. Finally, the more recent DBridge work [Ema+16; Ema+17] 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 [GPR20] 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+12] tries to optimize UDFs by breaking UDF code into pieces that can be moved around in the execution plan. The Pact programming model [Hue+13] reorders UDFs based on semantic information obtained from static code analysis. Sofa [Rhe+14] introduces UDF annotations to help optimization, and the Tupleware [Cro+15a] compiler architecture unifies workflow and UDF in the LLVM IR to perform optimization. Flare [Ess+18] unifies UDFs and queries in a generative programming framework [RO12] to enable native code generation of the whole query. Biltz [Sch+17] 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 [Sou+14] 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 work is related to other techniques that try to decompose a program synthesis problem into simpler sub-problems. For example, λ^2 [FCD15] and FlashMeta [PG15] perform compositional programming-by-example by inferring input-output examples for sub-problems, and Synquid [PKSL16] performs compositional synthesis from refinement types using so-called *round-trip type checking* and abduction. CPS [RGMF15] leverages natural language descriptions (in addition to examples) to decompose the synthesis problem into smaller sub-problems. Domino [Siv+16], 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 [Ahm+19] 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 [Hua+20] proposes a compositional synthesis framework that leverages both deductive and inductive to solve SyGuS problems [Alu+13]. 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.

4.9 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$\times$ 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 $3.5\times$ per UDF and between $1.3 - 3.1\times$ in terms of end-to-end running time for two representative applications and several workloads).

CHAPTER

5

COLUMN-WISE DECOMPOSITION

5.1 Introduction

Big data analytics frameworks like Spark [Zah+10] and Flink [Car+15] 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 [Arm+15; Beg+18], 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 imperative implementation [CSLM13; Ema+17; NF20]. However, these approaches are ill-equipped to tackle the functional API translation problem because (1) they rely on intermediate representations specific to the imperative APIs they consider (such as Java ORM programs and

array loops) which are not applicable to functional APIs and (2) they often rely on hand-crafted syntactic translation rules which limit their scope to a small set of stylized coding patterns.

In this chapter, we propose an automated technique for translating functional big data queries to semantically equivalent SQL versions. At a high level, our solution 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.

The technique we propose in this chapter 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 decompositional program synthesis techniques [AČR15; ARU17; SA16; GFVH21], 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 term *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, our 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 – i.e., 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 analystics 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%) 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

```
1  // T: RDD[(String, String)]
2  val T1 = T.groupBy(x => x._1.toLowerCase())
3  val O = T1.mapValues(x => (
4     x.size,
5     x.map(y => y._1.length + y._2.length +
          1).max()
6  ))
```

(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

Figure 5.1: Apache Spark Program and Equivalent SQL Query

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. 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 work makes the following contributions:

1. We propose the first automated approach for translating functional big data queries to SQL.
2. We describe a novel column-wise decomposition technique which leverages sketch-based program synthesis to decompose the full query synthesis problem into a set of smaller single-column synthesis problems.
3. We prove 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.

5.2 Overview

In this section, we give a high-level overview of our method using the Apache Spark program shown in Figure 5.1a. 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 Figure 5.1a, this program computes the output table `O` by

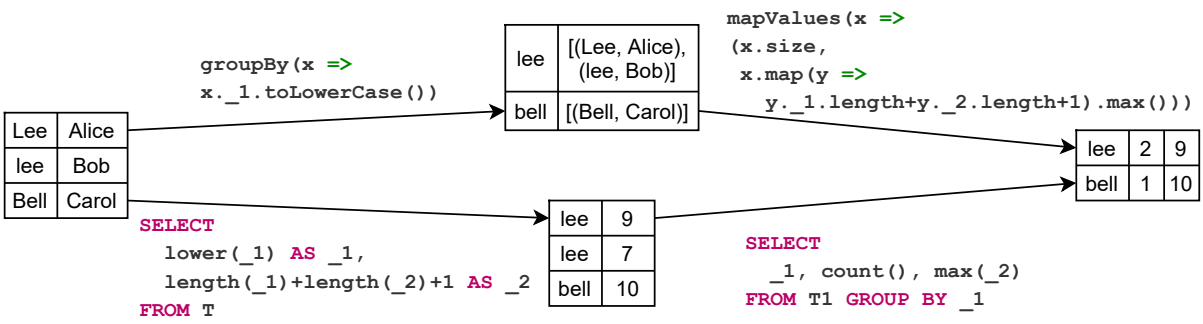


Figure 5.2: Semantic difference between functional APIs and SQL

using higher-order functional operators like `groupBy` and `mapValues`. The SQL query Q shown in Figure 5.1b 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, Figure 5.2 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 5.1, 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 [Siv+16; Ahm+19; Zha+21]. However, this generality comes at the cost of scalability compared to best-effort rule-based translation approaches. In fact, from a practical stand-point, 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 5.1. To recap, the basic idea is as follows: rather than generating a SQL expression to construct the *entire* output table, we instead synthesize N

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
```

Figure 5.3: Column queries for example

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 Figure 5.3. If we execute each column query Q_i on the input table from Figure 5.2, we obtain exactly the i'th column of the output table shown on the right side of Figure 5.2. Furthermore, by concatenating each of the three instantiations of the holes, we obtain precisely the desired SQL query in Figure 5.1b.

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 5.4.1. 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

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

Figure 5.4: Functional source language grammar

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, our algorithm can guarantee that the produced SQL query Q is semantically 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.

5.3 Problem Statement

The problem that we address in this work 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, Figure 5.4 shows the syntax of our source (functional) language, and Figure 5.5 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 of the 20 different API calls we support, including operators like map, filter, flatMap, groupBy, and aggregate (see Appendix A 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.

SQL Query	Q	$\rightarrow$	$\Pi \mid \sigma \mid \bowtie \mid \cup \mid \mathcal{G} \mid \langle \textit{inputTable} \rangle$
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	$\rightarrow$	$C \mid \langle \textit{func} \rangle(E_1, ..., E_n)$ AS $\langle alias \rangle$
Column	C	$\rightarrow$	$\langle columnName \rangle \mid \langle alias \rangle$

Figure 5.5: Target relational language grammar

Target language. Our target language, shown in Figure 5.5, 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 Figure 5.5 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. 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 $\mathsf{Def}(E)$ to refer to the name of the column defined via this AS construct. Conversely, we write $\mathsf{Refs}(E)$ 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 Figure 5.5) that is equivalent to the given functional Spark program (defined by Figure 5.4). 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 5. (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, $[\![P]\!]_{SQL}(T) = [\![Q]\!]_{Spark}(T)$.*

Query Sketch	Q_s	$\rightarrow$	$\texttt{Project}(Q, ?)$
Sub sketch	Q	$\rightarrow$	$\Pi \mid \sigma \mid \bowtie \mid \cup \mid \mathcal{G} \mid \langle \textit{inputTable} \rangle$
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	$\rightarrow$	$C \mid \langle \textit{func} \rangle (E_1, ..., E_n)$ AS $\langle alias \rangle$
Column	C	$\rightarrow$	$\langle columnName \rangle \mid \langle alias \rangle$

Figure 5.6: The grammar of query sketch

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

5.4 Translation Algorithm

In this section, we describe our synthesis algorithm for translating functional programs to SQL queries. First, we start by defining query sketches and discuss their key properties (Section 5.4.1). Then, we introduce our new SQL query synthesis algorithm based on the idea of column-wise decomposition (Section 5.4.2) and show how it can be made practical in Section 5.4.3. Finally, we prove that our algorithm is sound and complete and discuss its time complexity (Section 5.4.4).

5.4.1 Dependence-Free Query Sketches

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 7. (Query sketch) *A query sketch is a SQL expression that can contain holes. More formally, a query sketch is a word in the grammar of Figure 5.6, which augments the grammar*

of the target language by allowing ? *in certain production rules. We refer to the symbol* ? *in the sketch as a* hole.

Note that our query sketches (as defined in Figure 5.6) only allow holes as arguments of projection and aggregation operations. We also note that some sub-expressions in a query sketch may not contain unknowns; we refer to such hole-free subexpressions as *concrete expressions*. For example, consider the query sketch from Section 5.2 [1]:

Q_s

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

Here, `lower(_1)` is a concrete expression embedded inside Q_s.

Next, we say that a concrete query Q is an *instantiation* of query sketch Q_s if it can be obtained from Q_s by substituting every hole by a concrete expression. For example, concrete queries Q_1, Q_2, Q_3 from Figure 5.3 are all instantiations of Q_s. More formally, we define *sketch instantiation* as follows:

Definition 8. (Sketch instantiation) *We say that a query Q is an instantiation of sketch 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 6. *The queries Q_1, Q_2, and Q_3 in Figure 5.3 are instantiations of Q_s in Section 5.2 through the following substitutions respectively:*

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

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 7. *The query in Figure 5.1b corresponds to $Q_s[\sigma]$ where σ is the following substitution:*

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

[1] Note that query sketches in our system are expressed in the target language shown in Figure 5.5; however, for clarity, we will sometimes express the equivalent SQL sketch obtained via the translation rules in Appendix B.

Beyond restricting column queries to be instantiations of the same query sketch, our method further restricts the space of query sketches in order to ensure that the synthesized column queries can be *correctly and efficiently* merged into a full SQL query. We refer to this sub-class of query sketches as *dependence-free query sketches*, abbreviated as DFQS. Intuitively, a DFQS does *not* allow the result of a concrete expression to depend on how a given hole is instantiated, so different instantiations of the same DFQS can safely share concrete sub-expressions without affecting their semantics. As we prove later in this section, this dependence-freedom property makes it possible to correctly and easily consolidate column queries.

We make this notion of DFQS more precise in the following definition:

Definition 9. (DFQS) *Let E be a concrete subexpression in query sketch Q_s. We say that E is* dependence-free *in Q_s, if, for every input table T, E has the same value when executing* any *instantiation Q of Q_s on T. Then, a query sketch Q_s is said to be* dependence-free *if all concrete expressions that occur in it are dependence-free.*

We further clarify this notion of DFQS through examples and non-examples:

Example 8. *Consider the following query sketch where T is the input table and x is a column of table T:*

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

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:

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

This new sketch is not *dependence-free because there exist two different 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.

As we will see in later in this section, the concept of dependence-freedom is important for our synthesis algorithm and the correctness of our approach.

5.4.2 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 6 and takes as input a

Algorithm 6 Transpilation from functional query to SQL

```
1:  function SPARK2SQL(P)
2:      Input: A functional program P
3:      Output: SQL query Q equivalent to P or ⊥
4:      while true do
5:          Q_s ← GETNEXTDFQS( )
6:          if Q_s = ⊥ then return ⊥
7:          done ← true
8:          for each i ∈ [1, ..., n] do
9:              P_i ← P.map(x ⇒ x._i)
10:             Q_i ← CEGIS(P_i, Q_s)
11:             if Q_i = ⊥ then
12:                 done ← false
13:                 break
14:         if done then
15:             return MERGE(Q_s, Q_1, ..., Q_n)
```

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).

At a high level, the synthesis algorithm iterates over the space of all possible dependence-free query sketches and, in each iteration, tries to find a SQL query based on a specific DFQS Q_s (obtained via the call to GETNEXTDFQS in line 5 and discussed more in Section 5.4.3). 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 synthesis of column queries is performed using the standard CEGIS technique (see Section 5.5 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 MERGE procedure for consolidating the column queries is shown in Algorithm 7. Given a set of column queries $Q_1, \ldots, Q_n$ that are instantiations of the same sketch 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 $T_1, \ldots, T_n$. Thus, assuming that each column query Q_i is equivalent to program P_i from line 9 of Algorithm 6, the result

Algorithm 7 Procedure for consolidating column queries

```
1: function MERGE(Q_s, Q_1, ..., Q_n)
2:     Input: DFQS Q_s
3:     Input: Completions Q_i of Q_s
4:     Output: A SQL query
5:     σ ← {(?_i, []) | ?_i ∈ Domain(Q_s)}
6:     for each i ∈ [1, ..., n] do
7:         σ_i ← ExtractMapping(Q_s, Q_i)
8:         for each ?_i ∈ Domain(σ) do
9:             σ[?_i] ← σ[?_i] · σ_i[?_i]
10:    return Q_s[σ]
```

of calling MERGE produces a query Q that is equivalent to the input query P in the source language.

Due to our dependence-freedom assumption and restrictions on where holes can appear in the query sketch, 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 8). 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 2. (Merge correctness). *Let Q be the result of calling* MERGE *on column queries $Q_1, \ldots, Q_n$ that are instantiations of sketch Q_s. Then, for any input table T, $\Pi(Q(T), i) = Q_i(T)$.*

Proof. First, we assume that the expressions in $\sigma_1, ..., \sigma_n$ have unique aliases, such that there are no name conflicts after merge. This is always possible because any expression in σ_i can only be depended on by another expression in σ_i, thus if there are name conflicts, we can resolve them by renaming. Note that this is not true if Q_s is not dependence-free. Consider the following example:

$$Q_s = \texttt{Project}(\texttt{Project}(\texttt{Project}(Q, ?_1), C), ?_2)$$
$$\sigma_1 = \{?_1 \to [x \texttt{ AS } C], ...\}$$
$$\sigma_2 = \{?_1 \to [y \texttt{ AS } C], ...\}$$

We cannot rename C in σ_1 and σ_2 to unique names as C is referenced *concretely* by Q_s as the second argument of the second Project.

Next, we prove the theorem by induction. Let N be the number of relational operators in Q_s.

Base case. When $N = 1$, $Q_s = \texttt{Project}(t, ?)$, where t is the input table. We have $Q = \mathsf{Project}(t, \sigma_1[?], ..., \sigma_n[?])$, thus $\Pi(Q(T), i) = \mathsf{Project}(t, \sigma_i[?]) = Q_i(T)$.

Induction step. Assume the theorem is true for $N = k$, namely $\Pi(Q^k(T), i) = Q_i^k(T)$. Given Q_s^{k+1}, we pick an arbitrary leaf relational operation O in Q_s^{k+1} which has no sub query.

If there is no hole in O, let $t_o = O(T)$. Then we replace O in Q_s^{k+1}, Q^{k+1}, and Q_i^{k+1} with t_o to obtain Q_s^k, Q^k and Q_i^k, and augment T with t_o by $T' \leftarrow T \cup t_o$. We have $Q^{k+1}(T) = Q^k(T')$ and $Q_i^{k+1}(T) = Q_i^k(T')$. Thus $\Pi(Q^{k+1}(T), i) = \Pi(Q^k(T'), i) = Q_i^k(T') = Q_i^{k+1}(T)$.

If there is a hole in O, in case

$$O = \texttt{Aggregate}(t, \{C_1, ..., C_m\}, E_1, ..., E_l, ?)$$

, let

$$\begin{aligned} O_i &= O[\sigma_i] = \texttt{Aggregate}(t, \{C_1, ..., C_m\}, E_1, ..., E_l, \sigma_i[?]) \\ O_M &= \textsc{Merge}(O, O_1, ..., O_n) \\ &= \texttt{Aggregate}(t, \{C_1, ..., C_m\}, E_1, ..., E_l, \sigma_1[?], ..., \sigma_n[?]) \end{aligned}$$

We know that $t_M = O_M(T)$ is a super relation of $t_i = O_i(T)$, namely for all column $C \in t_i \rightarrow C \in t_M$. We then respectively replace O, O_M, and O_i in Q_s^{k+1}, Q_M^{k+1}, and Q_i^{k+1} with t_M to obtain Q_s^k, Q_M^k, and Q_i^k, as well as augment T with t_M by $T' \leftarrow T \cup t_M$. Similarly, we have $\Pi(Q^{k+1}(T), i) = \Pi(Q^k(T'), i) = Q_i^k(T') = Q_i^{k+1}(T)$.

For the case of $O = \texttt{Project}(t, E_1, ..., E_m, ?)$, similarly, let

$$\begin{aligned} O_i &= O[\sigma_i] = \texttt{Project}(t, E_1, ..., E_m, \sigma_i[?]) \\ O_M &= \textsc{Merge}(O, O_1, ..., O_n) \\ &= \texttt{Project}(t, E_1, ..., E_m, \sigma_1[?], ..., \sigma_n[?]) \end{aligned}$$

, and respectively replace O, O_M, and O_i in Q_s^{k+1}, Q_M^{k+1}, and Q_i^{k+1} with $t_M = O_M(T)$ to obtain Q_s^k, Q_M^k, and Q_i^k as well as let $T' = T \cup t_M$. Similarly, $\Pi(Q^{k+1}(T), i) = \Pi(Q^k(T'), i) = Q_i^k(T') = Q_i^{k+1}(T)$. □

5.4.3 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

Algorithm 8 Transpilation with DFQS extraction

```
1:  function SPARK2SQL-OPT(P)
2:      Input: A functional program P
3:      Output: SQL Query Q equivalent to P or ⊥
4:      while true do
5:          P_1 ← P.map(x ⇒ x._1)
6:          Q_1 ← CEGIS(P_1, ⊥)
7:          Q_s ← EXTRACTDFQS(Q_1)
8:          if Q_s = ⊥ then return ⊥
9:          done ← true
10:         for each i ∈ [2, ..., n] do
11:             P_i ← P.map(x ⇒ x._i)
12:             Q_i ← CEGIS(P_i, Q_s)
13:             if Q_i = ⊥ then
14:                 done ← false
15:                 break
16:         if done then
17:             return MERGE(Q_s, Q_1, ..., Q_n)
```

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 6 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 8 shows the optimized version of our synthesis algorithm based on the above idea. The key difference from Algorithm 6 is that the call GetNextDFQS 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 Figure 5.7 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 6, the loop terminates when queries for all

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

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

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

$$\textsc{Join}\quad \frac{\begin{array}{cc}\mathcal{C}_1 = \mathcal{C} \cup \{\mathsf{FirstColumn}(Q_1)\} & \mathcal{C}_2 = \mathcal{C} \cup \{\mathsf{FirstColumn}(Q_2)\} \\ \mathcal{C}_1 \vdash Q_1 \rightsquigarrow Q_1' & \mathcal{C}_2 \vdash Q_2 \rightsquigarrow Q_2'\end{array}}{\mathcal{C} \vdash \texttt{Join}(Q_1, Q_2) \rightsquigarrow \texttt{Join}(Q_1', Q_2')}$$

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

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

Figure 5.7: Rules for extracting DFQS from a query.

n columns have been synthesized, and line 17 consolidates these n column queries into a single one using the same MERGE procedure from Algorithm 7.

The idea behind the EXTRACTDFQS procedure, shown in Figure 5.7, is quite simple: Given a query Q, it essentially 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 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 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 Figure 5.7 using judgments of the following form:

$$\mathcal{C} \vdash Q \rightsquigarrow Q_s$$

Table 5.1: Example of extracting DFQS from a query (Example 9)

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

Here, $\mathcal{C}$ 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 $\mathcal{C}$, 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 Figure 5.7 are similar to each other, we only explain the PROJECT rule to give the reader some intuition. Let us consider a subquery of the form $\texttt{Project}(Q, \mathcal{E})$ where the parent query references columns $\mathcal{C}$. 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 $\mathsf{Refs}(\mathcal{E}')$ as the "context" when generalizing from sub-query Q to a query sketch Q'.

Example 9. *Consider the following query Q:*

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

Table 5.1 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:*

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

We conclude this section with a theorem stating that any query sketch extracted via EXTRACTDFQS satisfies Definition 9:

Theorem 3. (Correctness of ExtractDFQS) *For any query Q,* $\textsc{ExtractDFQS}(Q)$ *yields a dependence-free query sketch.*

Proof. According to Figure 5.7:

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

Since there is no concrete expression in the top `Project` operation, we only need to prove that Q' is dependence-free. To prove this, we will prove a slightly stronger property, namely that for any (potentially non-empty) $\mathcal{C}$, if $\mathcal{C} \vdash Q \rightsquigarrow Q_s$ then Q_s is dependence free. It will be beneficial in this proof to prove an additional property as well, namely that $\forall T \forall \sigma \forall c\ c \in \mathcal{C} \rightarrow \Pi(Q_s[\sigma](T), c) = \Pi(Q(T), c)$, i.e., for arbitrary input table T, a column $c \in \mathcal{C}$ has the same value when executing any instantiation of Q_s on T.

We prove these properties by induction. Let N be the number of relational operators in Q.

Base case. When $N = 0$, $Q_s = Q = t$ where t references an input table in T. Both properties trivially hold.

Induction step. Assume the two properties hold for any $N <= k$. Let Q^{k+1} be a query with $k + 1$ relational operators.

Case 1: $Q^{k+1} = \texttt{Aggregate}(Q^k, \mathcal{C}', \mathcal{E})$. After applying the AGGREGATE rule, we have

$$Q_s^{k+1} = \texttt{Aggregate}(Q_s^k, \mathcal{C}', \mathsf{Concat}(\mathcal{E}', ?))$$

where $\mathcal{E}' = \{E \mid E \in \mathcal{E} \wedge \mathsf{Def}(E) \in \mathcal{C}\}$ and $\mathcal{C}' \cup \mathsf{Refs}(\mathcal{E}') \vdash Q^k \rightsquigarrow Q_s^k$. Thus, for each $c \in C' \cup \mathsf{Refs}(\mathcal{E}')$, by the inductive hypothesis, we know that $\Pi(Q_s^k[\sigma](T), c) = \Pi(Q^k(T), c)$ for any input table T and completion σ. Note however, that $C' \cup \mathsf{Refs}(\mathcal{E}') = \mathsf{Refs}(C') \cup \mathsf{Refs}(\mathcal{E}')$ – in other words, $C' \cup \mathsf{Refs}(\mathcal{E}')$ are exactly the columns on which C' and $\mathcal{E}'$ depend. Thus, we can conclude the following for all completions σ and input tables T:

$$\forall c \in C' \cup \mathcal{E}'.\ \Pi(Q_s^k[\sigma](T), c) = \Pi(Q^k(T), c) \tag{5.1}$$

By the inductive hypothesis, we know Q_s^k is dependence free. Thus, the only expressions which could violate this property are $C' \cup \mathcal{E}'$. However, Equation 5.1 shows these do not violate the dependence free property, and thus Q_s^{k+1} is dependence free.

Furthermore, we can also use Equation 5.1 to prove the second property. In particular, we know that $C \subseteq C' \cup \mathcal{E}$ and by the definition of $\mathcal{E}'$ we know $\mathcal{E}' = \mathcal{E} \cap C$, and thus, Equation 5.1 proves exactly the property we want.

Cases for operators other than `Aggregate` are similar to approve. □

5.4.4 Properties of the Algorithm

In this section, we discuss some important properties of our proposed technique for transpiling functional queries to SQL. We first prove soundness and completeness of our algorithm. Then we show that our proposed column-wise decomposition idea leads to exponential reduction of the search space.

5.4.4.1 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 4. (Soundness) *Let Q be the output of Algorithm 8 on input query P. Then, if $Q \neq \perp$, we have $P \equiv Q$.*

Proof. If the algorithm fails before calling MERGE, $Q = \perp$. Otherwise, Q_s is a valid DFQS by Theorem 3, and $\forall i. Q_i \equiv P_i = P.\texttt{map}(x \Rightarrow x.\_i)$. According to Theorem 2, for any input table T and column i, $\Pi(Q, i)(T) = Q_i(T) = P(T).\texttt{map}(x \Rightarrow x.\_i)$. Therefore $Q \equiv P$. □

5.4.4.2 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. (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 8 will return a query Q such that (1) $Q \neq \perp$, and (2) $Q \equiv P$.*

Proof. To prove the completeness of our algorithm, we first will prove the following lemma:

Lemma 2. *If $\emptyset \vdash Q \rightsquigarrow Q'$, then there exists some σ such that $Q'[\sigma] = Q$.*

Proof. Let us introduce $?_i^{\mathcal{E}-\mathcal{E}'}$ for each hole introduced to Q' in the rules for Aggregate and Project, where $\mathcal{E} - \mathcal{E}'$ denotes the concrete expressions removed in that rule. Let $?^*$ denote the outermost hole argument to the added Project. Let us define σ as follows:

$$\sigma = \{?_i^{\mathcal{E}-\mathcal{E}'} \mapsto [e \mid e \in \mathcal{E} - \mathcal{E}'] \mid ?_i^{\mathcal{E}-\mathcal{E}'} \in Q'\} \cup \{?^* \mapsto [c_1, \ldots, c_n]\}$$

where n is the number of output columns in Q. Note, the order of expression arguments to Project and Aggregate is not preserved by this substitution. However, as all expressions are given a unique alias and all expressions are only referred to by their alias, ordering does not matter. □

Now, we continue with the proof of completeness. Let Q be a SQL query in $\mathcal{G}$ such that $Q \equiv P$ and let $Q_1 = \mathsf{Project}(Q, c_1)$. Due to the completeness of CEGIS, the call $\mathsf{CEGIS}(P_1, \bot)$ will eventually either (1) return some $Q_1' \neq Q_1$ which leads to a query sketch Q_s such that all column queries are successfully synthesized and the correct full query is returned or (2) return Q_1. In case (1) the theorem is satisfied, so we will focus on case (2). In this case, let $Q_s = \mathsf{ExtractDFQS}(Q_1) = \mathsf{Project}((Q', ?_1), ?_2)$ where $\emptyset \vdash Q \rightsquigarrow Q'$. By Lemma 2, there exists σ s.t. $Q'[\sigma] = Q$. Let $\sigma_i = Q_s[\sigma][?_1 \mapsto c_i][?_2 \mapsto c_i]$. This gives us completion σ_i of Q_s such that $Q_s[\sigma_i] = Q_i$. Thus, by the completeness of CEGIS, the call $\mathsf{CEGIS}(P_i, Q_s)$ will eventually yield Q_i. Therefore, by Theorem 2, we have $Q = \mathsf{Merge}(Q_s, Q_1, \ldots, Q_n)$. □

5.4.4.3 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:

$$O(r^{|Q|}) \tag{5.2}$$

Now, to reason about the complexity of the method *with* column-wise decomposition, recall that Algorithm 8 (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). [2] 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 finally, and let 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:

$$O(r^{|Q_1|} \times (N-1) \times r^{H \times L}) = O((N-1) \times r^{|Q_1| + H \times L}) \tag{5.3}$$

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:

$$O(r^{|Q_1|} + (N-1) \times r^{H \times L}) \tag{5.4}$$

Now, to understand the benefits of column-wise decomposition, let us compare Equations 5.2 and 5.4. 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 5.6.4, 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 5.3 to Equation 5.4.

5.5 Implementation and Optimizations

We implemented our proposed algorithm in a new tool called RDD2SQL that targets Scala programs written using the Spark RDD API [Zah+10], which is the functional query API for Apache Spark. We believe that Spark RDD is a good application for our approach because

[2] 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.

it is the de facto functional query engine.Our tool is implemented in Python and uses the Trinity [Mar+19] 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.

5.5.1 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 Figure 5.1. 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 Figure 5.1.

To exploit this observation, we leverage CLIS [Zha+21], 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 [Pai+21]. 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 $E \rightarrow \texttt{length}(\texttt{\_1}) + \texttt{length}(\texttt{\_2}) + 1$ and $E \rightarrow \texttt{length}(\texttt{\_1})$ 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.

5.5.2 API-Level Decomposition

In addition to our novel column-wise decomposition idea, RDD2SQL also employs other types of decomposition strategies proposed in prior work [Zha+21]. 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 [Zha+21]. 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.

5.5.3 Inductive Synthesizer

Recall that the underlying CEGIS solver consists of an *inductive synthesizer* and a *verifier*. Our underlying CEGIS solver leverages Trinity [Mar+19], 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* [CC77] 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 Figure 5.5 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 [Fen+17; WCB17], reasoning about table dimensions provides good pruning power without adding too much overhead to the inductive synthesizer.

5.5.4 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 [CKL04], 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.

5.5.5 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 iteration. 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 [Bey+13; Hol+10], 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.

5.6 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.** How much of an impact do the optimizations described in Section 5.5 have on synthesis time?
- **RQ5.** 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.

Table 5.2: 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

5.6.1 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 5.2 gives statistics about these functional queries in terms of the number of functional API calls, LOC, and AST size.

5.6.2 Manual Study

To assess how many big-data queries can be rewritten to SQL, we performed a manual study, summarized in Table 5.3, of the 100 benchmarks. Overall, we found that 21 of the 100 benchmarks cannot be translated to SQL, while the remaining 79 do have equivalent SQL queries. For most of the queries that cannot be translated (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 are not translatable 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.

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.

Table 5.3: 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

Table 5.4: Main synthesis results

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

5.6.3 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. Figure 5.8 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 5.4 gives more detailed statistics about the performance of RDD2SQL. As 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 5.6.6).

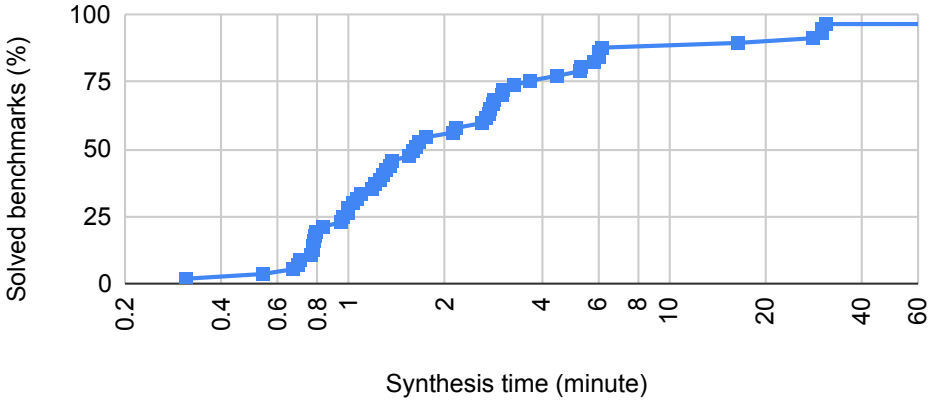


Figure 5.8: Cumulative distribution of synthesis time

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.

5.6.4 Evaluating Column-Wise Decomposition

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 5.5 enabled.

Figure 5.9 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 a 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.

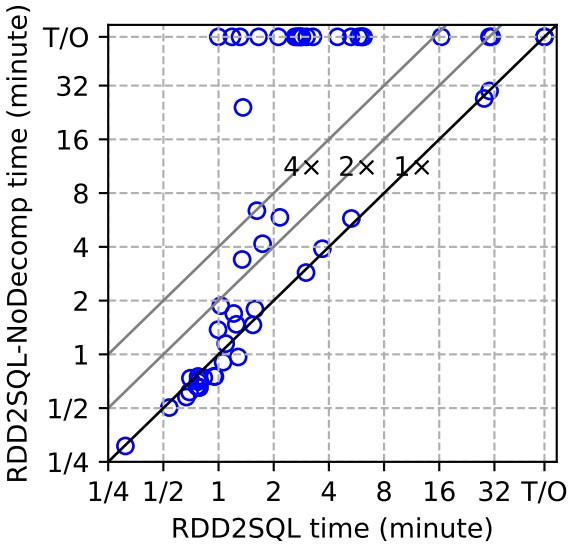


Figure 5.9: Comparing synthesis time of RDD2SQL and RDD2SQL-NoDecomp

5.6.4.0.1 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 5.4.4 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 8, (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 5.4 from Section 5.4.4.

Next, Figure 5.10 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

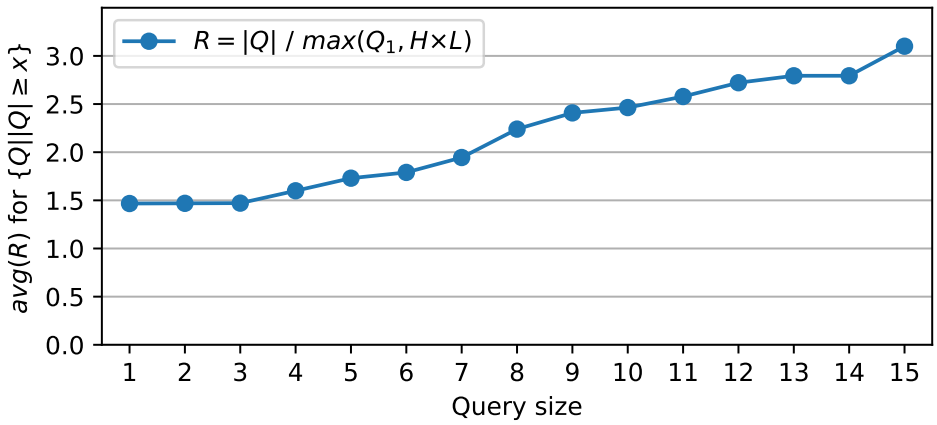


Figure 5.10: Evaluating asymptotic complexity with and without column-wise decomposition. The x-axis shows query size s and the y-axis shows the average ratio $R = |Q|/max(|Q_1|, H \times L)$ for queries of at least size s.

from Figure 5.10, 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, Figure 5.10 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.

5.6.5 Evaluating Optimizations

In this section, we describe the results of another ablation study to evaluate the impact of the optimizations described in Section 5.5. 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 5.5.1.
- RDD2SQL-NoDFG: This is the variant of RDD2SQL that does not perform the optimization described in Section 5.5.2 (i.e., DFG-based decomposition adapted from prior work [Zha+21]).

The results of this ablation study are presented in Figure 5.11 as a bar graph. The x-axis shows the time limit, and the y-axis shows the percentage of benchmarks solved within that time

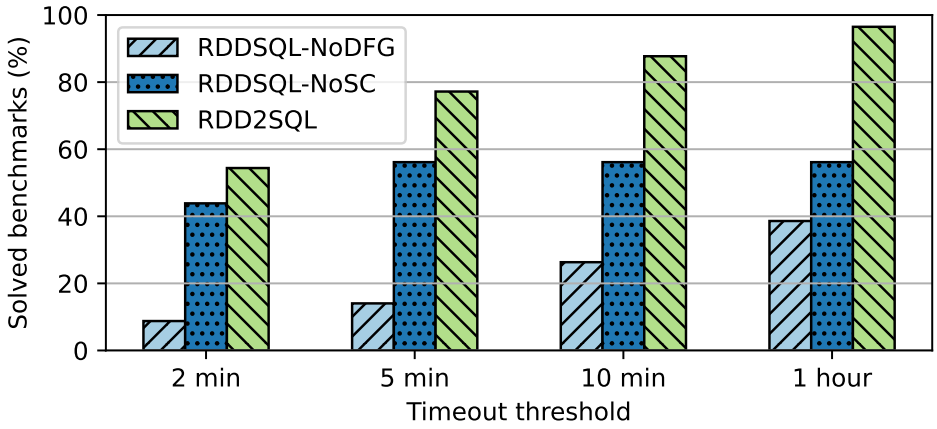


Figure 5.11: Comparing complete and ablated versions

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 5.5.2 is even more pronounced. Furthermore, the benefit of the grammar augmentation optimization from Section 5.5.1 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. Nearly half of the benchmarks cannot be solved within a time limit of 1 hour without either of these optimizations.

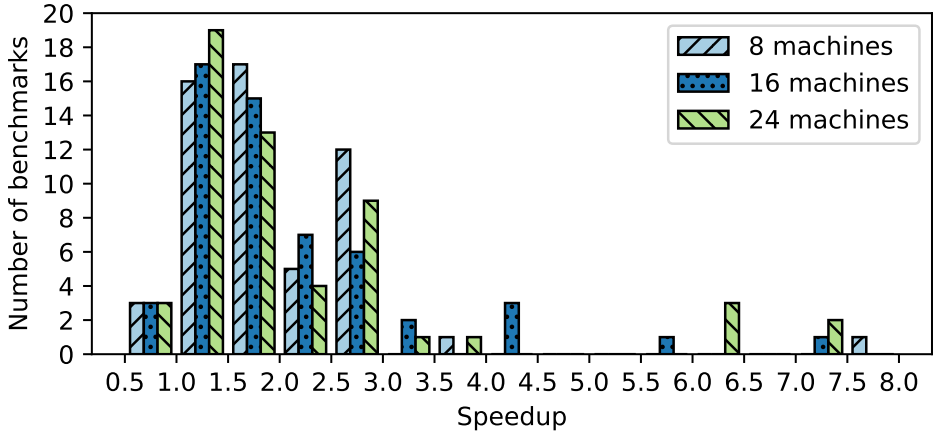


Figure 5.12: Distribution of speedups. The baselines are the original functional queries processed on 8, 16, or 24 machines respectively.

5.6.6 Performance Benefits of SQL Translation

A key motivation behind 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. Figure 5.12 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$\times$ speedup.

Table 5.5: 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

Table 5.5 gives some additional statistics about the comparison. Across all cluster sizes, the SQL query is over 7$\times$ faster than its Spark equivalent in the best case. On average, a SQL query was approximately 2$\times$ 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$\times$ 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 RQ5: Translating Spark RDD programs to SQL using RDD2SQL conferred an average speedup of 2$\times$ and a maximum speedup of over 7$\times$ across a variety of cluster sizes and randomly sampled inputs.

5.7 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 [Zha+21], which converts UDFs used in SQL queries to pure SQL expressions. The DFG-based decomposition in Section 5.5.1 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 [CSLM13], 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 [Ema+17] also synthesizes SQL from Java ORM programs, and SQLgen [NF20] 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 [PKSL16; SA16; FCD15; OZ15] 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 [PKSL16] and BIGλ [SA16] infer type specifications for unknowns given a top-level type specification, while λ^2 [FCD15] and MYTH [OZ15] 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 [AČR15; ARU17; GFVH21] 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 [Mar+20] 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 work is related to a line of inductive synthesis work on learning SQL queries from input-output examples. For instance, Scythe [WCB17], Query-By-Output [TCP09], Trinity [Mar+19], SQLSynthesizer [ZS13], SqlSol [Che19], SQUARES [Orv+20], EGS [Tha+21], and PATSQL [Tak+21] 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 [Mar+19] 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 [Yag+17] synthesizes SQL from natural language; Foofah [Jin+17], Mitra [YWD18], Hades [Yag+16], Dynamite [Wan+20], and Morpheus [Fen+17] all synthesize programs from examples in order to automate transformations between different structures and/or schemas. FlashFill [SG12] and BlinkFill [Sin16] automatically fill spreadsheet columns by synthesizing string manipulation programs from examples. HYB [RG20] utilizes program-by-example to synthesize programs for Web data extraction. Finally, [Sin+17] employs CEGIS to learn entity matching rules for detecting records that refer to the same object.

Optimizing Spark programs. The Spark [Zah+10] framework executes programs written in its functional RDD API [Zah+12] using a traditional volcano [GM93] 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+12] and Niijima [Xu+19] optimize SCOPE [Cha+08] queries by eliminating, moving, and merging code in UDFs. PeriSCOPE reports an average speedup of $1.7\times$ on 8 studied cases, and Niijima accelerates 21 queries by $1.24\times$; however, it is unclear whether these optimizations are effective for Spark. In contrast, our method achieves an average $2\times$ speedup through the innovative *column-wise* decomposition. Stratosphere [Ale+14] and Tupleware [Cro+15b] 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 [Sou+14] avoids redundant computations in UDFs across multiple queries; in contrast, our work focuses on optimizing single query.

Other efforts focus on accelerating Spark programs by improving runtime components, including distributed job scheduling [Wan+19a; SGM16], communication [Ngu+18], native code generation [Ess+18; Nav+19], and memory management [Shi+19; Ngu+15; Maa+15]. 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.

5.8 Conclusion

Motivated by the performance advantages of SQL over functional big data queries, this chapter 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 $2\times$ faster than original functional queries on average, with a maximum speedup of $7\times$.

CHAPTER

6

CONCLUSIONS AND FUTURE WORKS

6.1 Overarching Conclusion

Motivated by the problem that many inefficient code across the boundaries of data analytics TPCs cannot be effectively optimized by existing techniques, we propose cross-boundary optimization. In particular, we identify three types of TPC boundaries and propose a novel method to optimize programs of a specific domain for each boundary type:

- **Type 1: Boundary between TPC and host code**. We focus on the domain of data analytics programs written in Python. We first propose MIN-watch to dynamically determine data dependency between TPC API calls and host code, and then create an optimization framework called *Cunctator* featuring best-effort lazy evaluation enabled by MIN-watch to optimize inefficient sequences of API calls.
- **Type 2: Boundary between TPC and user-defined extension**. We target the problem of optimizing UDFs in SQL. We address it by translating UDFs into expressions of SQL built-in functions through compositional lazy inductive synthesis (*CLIS*), a novel program synthesis method that exploits composibility of both source and target programs to dramatically improve scalability.

- **Type 3: Boundary between TPCs**. We study how to apply optimization techniques of existing SQL engines to functional big-data queries. Our strategy is to translate functional queries into SQL querie before applying SQL optimization techniques. We fulfill the strategy through our novel SQL synthesis method that leverages column-wise composibility of SQL queries to address the scalability issue. The method is implemented in a tool named *RDD2SQL*.

To evaluate the effectiveness of our methods, we implement our methods in tools to optimize real-world benchmarks collected from Github. Results demonstrate that our methods are applicable to most of benchmarks and significantly accelerate applicable ones:

- *Cunctator* accelerates all 15 benchmarks through 4 optimizers by 1.03-14.2×.
- *CLIS* translates 58 out of 63 benchmarks. After translation, 56 benchmarks gain speedups averaged 3.5×.
- *RDD2SQL* solves 55 out of 57 benchmarks, in which 52 of them are accelerated. On average, the speedup is approximately 2×.

6.2 Future Work

We now discuss the future works. We classify them into three directions: improvement, generalization, and unification.

Improvement. In all three works presented in this thesis, there are potential improvements left for future works. For Cunctator, currently optimizer developers need to completely implement the details of optimization passes as Cunctator only provides deferred API call sequence. As an optimization framework, it will be helpful to provide APIs featuring commonly used program optimization techniques, such as data-flow analysis and common sub-expression elimination. Moreover, due to the overhead, Cunctator currently disables itself for programs calling more than 1000 APIs per second. This could be improved by reducing Cunctator's overhead, e.g., through re-implementing most parts of Cunctator using the C programming language. For CLIS, one important unanswered question is whether we can further improve the scalability of the algorithm through smarter decomposition or merge, rather than most-aggressive decompostion and enumerative DFG node merge that the algorithm currently utilize. For RDD2SQL, the synthesis time takes minutes and thus prohibits online application of the algorithm. A possible

way to reduce the synthesis time is to efficiently predict the query sketch using machine learning techniques. Finally, for both CLIS and RDD2SQL, the algorithms do not consider the performance of synthesized programs as empirical results show significant speedups after translation. We could further improve the performance through employing cost-guided synthesis to produce optimal or near optimal programs.

Generalization. Another important future direction is to generalize our method to more domains. For instance, we can generalize Cunctator to languages other than Python, such as Julia. Similarly, we can generalize CLIS to transpile arbitrary programs from one programming language to another programming language. 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. Finally, we can leverage column-wise decomposition to accelerate the synthesis of SQL in general (e.g., inductive SQL synthesizer), or even other programs operating over tables (e.g., Pandas programs).

Unification. The methods proposed in this paper respectively targets one type of TPC boundary. It is interesting to study the synergy of these methods. In other words, are there optimization opportunities across multiple types of runtime boundaries that require a fusion of these methods?

BIBLIOGRAPHY

[Aba+16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. "TensorFlow: A System for Large-Scale Machine Learning". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. USA: USENIX Association, 2016, pp. 265–283. ISBN: 978-1-931971-33-1.

[AC18] Maaz Bin Safeer Ahmad and Alvin Cheung. "Automatically Leveraging MapReduce Frameworks for Data-Intensive Applications". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD'18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 1205–1220. ISBN: 978-1-4503-4703-7. DOI: 10.1145/3183713.3196891.

[AČR15] Rajeev Alur, Pavol Černý, and Arjun Radhakrishna. "Synthesis Through Unification". In: *Computer Aided Verification*. Ed. by Daniel Kroening and Corina S. Păsăreanu. Cham: Springer International Publishing, 2015, pp. 163–179. ISBN: 978-3-319-21668-3. DOI: 10.1007/978-3-319-21668-3_10.

[Ahm+19] Maaz Bin Safeer Ahmad, Jonathan Ragan-Kelley, Alvin Cheung, and Shoaib Kamil. "Automatically Translating Image Processing Libraries to Halide". In: *ACM Trans. Graph.* 38.6 (2019). ISSN: 0730-0301. DOI: 10.1145/3355089.3356549.

[AK01] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001. ISBN: 978-1-55860-286-1.

[Ale+14] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, et al. "The stratosphere platform for big data analytics". In: *The VLDB Journal* 23.6 (2014), pp. 939–964. DOI: 10.1007/s00778-014-0357-y.

[Alu+13] 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. "Syntax-guided synthesis". In: *2013 Formal*

Methods in Computer-Aided Design. 2013, pp. 1–8. ISBN: 978-0-9835678-3-7. DOI: 10.1109/FMCAD.2013.6679385.

[Arm+15] 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. "Spark SQL: Relational Data Processing in Spark". In: SIGMOD'15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 1383–1394. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2742797.

[ARU17] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. "Scaling Enumerative Program Synthesis via Divide and Conquer". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Axel Legay and Tiziana Margaria. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 319–336. ISBN: 978-3-662-54577-5. DOI: 10.1007/978-3-662-54577-5_18.

[Beg+18] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. "Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources". In: SIGMOD'18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 221–230. ISBN: 978-1-4503-4703-7. DOI: 10.1145/3183713.3190662.

[Bey+13] Dirk Beyer, Andreas Holzer, Michael Tautschnig, and Helmut Veith. "Information Reuse for Multi-goal Reachability Analyses". In: *Programming Languages and Systems*. Ed. by Matthias Felleisen and Philippa Gardner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 472–491. ISBN: 978-3-642-37036-6. DOI: 10.1007/978-3-642-37036-6_26.

[BHY88] Adrienne G. Bloss, Paul Hudak, and Jonathan Young. "Code Optimizations for Lazy Evaluation". In: *LISP Symb. Comput.* 1.2 (1988), pp. 147–164. DOI: 10.1007/BF01806169.

[Car+15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. "Apache Flink™: Stream and Batch Processing in a Single Engine". In: *IEEE Data Eng. Bull.* 38.4 (2015), pp. 28–38.

[CC77] Patrick Cousot and Radhia Cousot. "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*. POPL'77. New York, NY, USA: Associ-

ation for Computing Machinery, 1977, pp. 238–252. ISBN: 978-1-4503-7350-0. DOI: 10.1145/512950.512973.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming - methods, tools and applications*. Addison-Wesley, 2000. ISBN: 978-0-201-30977-5.

[CE82] Edmund M. Clarke and E. Allen Emerson. "Design and synthesis of synchronization skeletons using branching time temporal logic". In: *Logics of Programs*. Ed. by Dexter Kozen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 52–71. ISBN: 978-3-540-39047-3. DOI: 10.1007/BFb0025774.

[Cha+08] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. "SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets". In: *Proc. VLDB Endow.* 1.2 (2008), pp. 1265–1276. ISSN: 2150-8097. DOI: 10.14778/1454159.1454166.

[Cha98] Surajit Chaudhuri. "An Overview of Query Optimization in Relational Systems". In: *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. PODS'98. New York, NY, USA: Association for Computing Machinery, 1998, pp. 34–43. ISBN: 978-0-89791-996-8. DOI: 10.1145/275487.275492.

[Che19] Lin Cheng. "SqlSol: An accurate SQL Query Synthesizer". In: *Formal Methods and Software Engineering*. Ed. by Yamine Ait-Ameur and Shengchao Qin. Cham: Springer International Publishing, 2019, pp. 104–120. ISBN: 978-3-030-32409-4. DOI: 10.1007/978-3-030-32409-4_7.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. "A Tool for Checking ANSI-C Programs". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Kurt Jensen and Andreas Podelski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 168–176. ISBN: 978-3-540-24730-2. DOI: 10.1007/978-3-540-24730-2_15.

[CMSL16] Alvin Cheung, Samuel Madden, and Armando Solar-Lezama. "Sloth: Being Lazy Is a Virtue (When Issuing Database Queries)". In: *ACM Transactions on Database Systems* 41.2 (2016). ISSN: 0362-5915. DOI: 10.1145/2894749.

[Cro+15a] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Cetintemel, and Stan Zdonik. "An Architecture for Compiling UDF-Centric Workflows". In: *Proc. VLDB Endow.* 8.12 (2015), pp. 1466–1477. ISSN: 2150-8097. DOI: 10.14778/2824032.2824045.

[Cro+15b] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur Çetintemel, and Stanley B Zdonik. “Tupleware: "Big" Data, Big Analytics, Small Clusters.” In: *CIDR*. 2015.

[CSLM13] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. “Optimizing Database-Backed Applications with Query Synthesis”. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI’13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 3–14. ISBN: 978-1-4503-2014-6. DOI: 10.1145/2491956.2462180.

[Cyt+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. “Efficiently Computing Static Single Assignment Form and the Control Dependence Graph”. In: *ACM Trans. Program. Lang. Syst.* 13.4 (1991), pp. 451–490. ISSN: 0164-0925. DOI: 10.1145/115372.115320.

[Ema+16] K. Venkatesh Emani, Karthik Ramachandra, Subhro Bhattacharya, and S. Sudarshan. “Extracting Equivalent SQL from Imperative Code in Database Applications”. In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD’16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 1781–1796. ISBN: 978-1-4503-3531-7. DOI: 10.1145/2882903.2882926.

[Ema+17] K. Venkatesh Emani, Tejas Deshpande, Karthik Ramachandra, and S. Sudarshan. “DBridge: Translating Imperative Code to SQL”. In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD’17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 1663–1666. ISBN: 978-1-4503-4197-4. DOI: 10.1145/3035918.3058747.

[Ess+18] Grégory M. Essertel, Ruby Y. Tahboub, James M. Decker, Kevin J. Brown, Kunle Olukotun, and Tiark Rompf. “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. USA: USENIX Association, 2018, pp. 799–815. ISBN: 978-1-931971-47-8.

[FCD15] John K. Feser, Swarat Chaudhuri, and Isil Dillig. “Synthesizing Data Structure Transformations from Input-Output Examples”. In: PLDI’15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 229–239. ISBN: 978-1-4503-3468-6. DOI: 10.1145/2737924.2737977.

[Fen+17] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. "Component-Based Synthesis of Table Consolidation and Transformation Tasks from Examples". In: PLDI 2017. New York, NY, USA: Association for Computing Machinery, 2017, pp. 422–436. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062351.

[FSFP19] Bruno Morais Ferreira, Britaldo Silveira Soares-Filho, and Fernando Magno Quintão Pereira. "The Dinamica EGO virtual machine". In: *Science of Computer Programming* 173 (2019). Brazilian Symposium on Programming Languages (SBLP '15+16), pp. 3–20. ISSN: 0167-6423. DOI: 10.1016/j.scico.2018.02.002.

[FTA14] Jose M. Faleiro, Alexander Thomson, and Daniel J. Abadi. "Lazy Evaluation of Transactions in Database Systems". In: SIGMOD'14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 15–26. ISBN: 978-1-4503-2376-5. DOI: 10.1145/2588555.2610529.

[GFVH21] Sankha Narayan Guria, Jeffrey S. Foster, and David Van Horn. "RbSyn: Type- and Effect-Guided Program Synthesis". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, pp. 344–358. ISBN: 978-1-4503-8391-2. DOI: 10.1145/3453483.3454048.

[GL05] Samuel Z. Guyer and Calvin Lin. "Broadway: A Compiler for Exploiting the Domain-Specific Semantics of Software Libraries". In: *Proc. IEEE* 93.2 (2005), pp. 342–357. DOI: 10.1109/JPROC.2004.840489.

[GM93] Goetz Graefe and William J McKenna. "The volcano optimizer generator: Extensibility and efficient search". In: *Proceedings of IEEE 9th International Conference on Data Engineering*. IEEE. 1993, pp. 209–218. ISBN: 978-0-8186-3570-0. DOI: 10.1109/ICDE.1993.344061.

[GPR20] Surabhi Gupta, Sanket Purandare, and Karthik Ramachandra. "Aggify: Lifting the Curse of Cursor Loops Using Custom Aggregates". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD'20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 559–573. ISBN: 978-1-4503-6735-6. DOI: 10.1145/3318464.3389736.

[GPS17] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. “Program Synthesis”. In: *Foundations and Trends® in Programming Languages* 4.1-2 (2017), pp. 1–119. ISSN: 2325-1107. DOI: 10.1561/2500000010.

[Guo+12] Zhenyu Guo, Xuepeng Fan, Rishan Chen, Jiaxing Zhang, Hucheng Zhou, Sean McDirmid, Chang Liu, Wei Lin, Jingren Zhou, and Lidong Zhou. “Spotting Code Optimizations in Data-Parallel Pipelines through PeriSCOPE”. In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI’12. USA: USENIX Association, 2012, pp. 121–133. ISBN: 978-1-931971-96-6.

[GWO99] James R Groff, Paul N Weinberg, and Andrew J Oppel. *SQL: The Complete Reference*. McGraw-Hill/Osborne, 1999. ISBN: 978-0-07-211845-2.

[HM76] Peter Henderson and James H. Morris. “A Lazy Evaluator”. In: *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages*. Ed. by Susan L. Graham, Robert M. Graham, Michael A. Harrison, William I. Grosky, and Jeffrey D. Ullman. POPL’76. New York, NY, USA: Association for Computing Machinery, 1976, pp. 95–103. ISBN: 978-1-4503-7477-4. DOI: 10.1145/800168.811543.

[Hol+10] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. “How Did You Specify Your Test Suite”. In: *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. ASE’10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 407–416. ISBN: 978-1-4503-0116-9. DOI: 10.1145/1858996.1859084.

[HS86] W. Daniel Hillis and Guy L. Steele. “Data Parallel Algorithms”. In: *Commun. ACM* 29.12 (1986), pp. 1170–1183. ISSN: 0001-0782. DOI: 10.1145/7902.7903.

[Hua+20] Kangjing Huang, Xiaokang Qiu, Peiyuan Shen, and Yanjun Wang. “Reconciling Enumerative and Deductive Program Synthesis”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1159–1174. ISBN: 978-1-4503-7613-6. DOI: 10.1145/3385412.3386027.

[Hud+07] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. “A History of Haskell: Being Lazy with Class”. In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. Ed. by Barbara G. Ryder and Brent Hailpern. HOPL III. New York, NY, USA: Association for Computing Machinery, 2007, pp. 1–55. ISBN: 978-1-59593-766-7. DOI: 10.1145/1238844.1238856.

[Hue+13] Fabian Hueske, Mathias Peters, Aljoscha Krettek, Matthias Ringwald, Kostas Tzoumas, Volker Markl, and Johann-Christoph Freytag. “Peeking into the Optimization of Data Flow Programs with MapReduce-style UDFs”. In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 2013, pp. 1292–1295. ISBN: 978-1-4673-4910-9. DOI: 10.1109/ICDE.2013.6544927.

[Jin+17] Zhongjun Jin, Michael R. Anderson, Michael Cafarella, and H. V. Jagadish. “Foofah: Transforming Data By Example”. In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD’17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 683–698. ISBN: 978-1-4503-4197-4. DOI: 10.1145/3035918.3064034.

[Joh84] Thomas Johnsson. “Efficient Compilation of Lazy Evaluation”. In: *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*. Ed. by Mary S. Van Deusen and Susan L. Graham. SIGPLAN’84. New York, NY, USA: Association for Computing Machinery, 1984, pp. 58–69. ISBN: 978-0-89791-139-9. DOI: 10.1145/502874.502880.

[Ken+05] Ken Kennedy, Bradley Broom, Arun Chauhan, Robert J. Fowler, John Garvin, Charles Koelbel, Cheryl McCosh, and John M. Mellor-Crummey. “Telescoping Languages: A System for Automatic Generation of Domain Languages”. In: *Proc. IEEE* 93.2 (2005), pp. 387–408. DOI: 10.1109/JPROC.2004.840447.

[KPP11] Georgios Korfiatis, Michalis A. Papakyriakou, and Nikolaos Papaspyrou. “A Type and Effect System for Implementing Functional Arrays with Destructive Updates”. In: *Federated Conference on Computer Science and Information Systems - FedCSIS 2011, Szczecin, Poland, 18-21 September 2011, Proceedings*. Ed. by Maria Ganzha, Leszek A. Maciaszek, and Marcin Paprzycki. 2011, pp. 879–886. ISBN: 978-83-60810-39-2.

[Lau93] John Launchbury. “A Natural Semantics for Lazy Evaluation”. In: *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming*

Languages. POPL'93. New York, NY, USA: Association for Computing Machinery, 1993, pp. 144–154. ISBN: 978-0-89791-560-1. DOI: 10.1145/158511.158618.

[LPS15] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. "Numba: A LLVM-Based Python JIT Compiler". In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. LLVM'15. New York, NY, USA: Association for Computing Machinery, 2015. ISBN: 978-1-4503-4005-2. DOI: 10.1145/2833157.2833162.

[Maa+15] Martin Maas, Tim Harris, Krste Asanovic, and John Kubiatowicz. "Trash Day: Coordinating Garbage Collection in Distributed Systems". In: *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*. HOTOS'15. USA: USENIX Association, 2015.

[Mar+19] Ruben Martins, Jia Chen, Yanju Chen, Yu Feng, and Isil Dillig. "Trinity: An Extensible Synthesis Framework for Data Science". In: *Proc. VLDB Endow.* 12.12 (2019), pp. 1914–1917. ISSN: 2150-8097. DOI: 10.14778/3352063.3352098.

[Mar+20] Benjamin Mariano, Yanju Chen, Yu Feng, Shuvendu K. Lahiri, and Isil Dillig. "Demystifying Loops in Smart Contracts". In: *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 262–274. ISBN: 978-1-4503-6768-4.

[McK11] Wes McKinney. "pandas: a Foundational Python Library for Data Analysis and Statistics". In: *Python for High Performance and Scientific Computing* 14 (2011).

[Mol+19] Dan Moldovan, James M. Decker, Fei Wang, Andrew A. Johnson, Brian K. Lee, Zachary Nado, D. Sculley, Tiark Rompf, and Alexander B. Wiltschko. "AutoGraph: Imperative-style Coding with Graph-based Performance". In: *SysML*. 2019.

[Mor+11] Kristi Morton, Magdalena Balazinska, Dan Grossman, and Christopher Olston. "The Case for Being Lazy: How to Leverage Lazy Evaluation in MapReduce". In: *Proceedings of the 2nd International Workshop on Scientific Cloud Computing*. ScienceCloud'11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 1–6. ISBN: 978-1-4503-0699-7. DOI: 10.1145/1996109.1996111.

[Nav+19] Christian Navasca, Cheng Cai, Khanh Nguyen, Brian Demsky, Shan Lu, Miryung Kim, and Guoqing Harry Xu. “Gerenuk: Thin Computation over Big Native Data Using Speculative Program Transformation”. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP’19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 538–553. ISBN: 978-1-4503-6873-5. DOI: 10.1145/3341301.3359643.

[Neu11] Thomas Neumann. “Efficiently Compiling Efficient Query Plans for Modern Hardware”. In: *Proc. VLDB Endow.* 4.9 (2011), 539–550. ISSN: 2150-8097. DOI: 10.14778/2002938.2002940.

[NF20] Md Hasanuzzaman Noor and Leonidas Fegaras. “Translation of Array-Based Loops to Spark SQL”. In: *2020 IEEE International Conference on Big Data (Big Data)*. IEEE. 2020, pp. 469–476. ISBN: 978-1-7281-6251-5. DOI: 10.1109/BigData50022.2020.9378136.

[Ngu+15] Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, Jianfei Hu, and Guoqing Xu. “FACADE: A Compiler and Runtime for (Almost) Object-Bounded Big Data Applications”. In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS’15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 675–690. ISBN: 978-1-4503-2835-7. DOI: 10.1145/2694344.2694345.

[Ngu+18] Khanh Nguyen, Lu Fang, Christian Navasca, Guoqing Xu, Brian Demsky, and Shan Lu. “Skyway: Connecting Managed Heaps in Distributed Big Data Systems”. In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 56–69. ISBN: 978-1-4503-4911-6. DOI: 10.1145/3173162.3173200.

[OR14] Martin Odersky and Tiark Rompf. “Unifying Functional and Object-Oriented Programming with Scala”. In: *Commun. ACM* 57.4 (2014), pp. 76–86. ISSN: 0001-0782. DOI: 10.1145/2591013.

[Orv+20] Pedro Orvalho, Miguel Terra-Neves, Miguel Ventura, Ruben Martins, and Vasco Manquinho. “SQUARES: A SQL Synthesizer Using Query Reverse Engineering”. In: *Proc. VLDB Endow.* 13.12 (2020), pp. 2853–2856. ISSN: 2150-8097. DOI: 10.14778/3415478.3415492.

[OZ15] Peter-Michael Osera and Steve Zdancewic. “Type-and-Example-Directed Program Synthesis”. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI’15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 619–630. ISBN: 978-1-4503-3468-6. DOI: 10.1145/2737924.2738007.

[Pai+21] Shankara Pailoor, Yuepeng Wang, Xinyu Wang, and Isil Dillig. “Synthesizing Data Structure Refinements from Integrity Constraints”. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 574–587. ISBN: 978-1-4503-8391-2. DOI: 10.1145/3453483.3454063.

[Pal+17] Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. “Weld: A common runtime for high performance data analytics”. In: *Conference on Innovative Data Systems Research (CIDR)*. 2017.

[Pas+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. NIPS’19. Red Hook, NY, USA: Curran Associates Inc., 2019, pp. 8026–8037.

[PG15] Oleksandr Polozov and Sumit Gulwani. “FlashMeta: A Framework for Inductive Program Synthesis”. In: *SIGPLAN Not.* 50.10 (2015), pp. 107–126. ISSN: 0362-1340. DOI: 10.1145/2858965.2814310.

[PKSL16] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. “Program Synthesis from Polymorphic Refinement Types”. In: PLDI’16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 522–538. ISBN: 978-1-4503-4261-2. DOI: 10.1145/2908080.2908093.

[Ram+17a] Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, César Galindo-Legaria, and Conor Cunningham. “Froid: Optimization of Imper-

ative Programs in a Relational Database". In: *Proc. VLDB Endow.* 11.4 (2017), pp. 432–444. ISSN: 2150-8097. DOI: 10.1145/3186728.3164140.

[Ram+17b] Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, César A. Galindo-Legaria, and Conor Cunningham. "Optimization of Imperative Programs in a Relational Database". In: *CoRR* abs/1712.00498 (2017). arXiv: 1712.00498.

[RG20] Mohammad Raza and Sumit Gulwani. "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*. SIGMOD'20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1967–1978. ISBN: 978-1-4503-6735-6. DOI: 10.1145/3318464.3380608.

[RGMF15] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. "Compositional Program Synthesis from Natural Language and Examples". In: *Proceedings of the 24th International Conference on Artificial Intelligence*. IJCAI'15. Buenos Aires, Argentina: AAAI Press, 2015, pp. 792–800. ISBN: 978-1-57735-738-4.

[Rhe+14] Astrid Rheinländer, Martin Beckmann, Anja Kunkel, Arvid Heise, Thomas Stoltmann, and Ulf Leser. "Versatile Optimization of UDF-Heavy Data Flows with Sofa". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD'14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 685–688. ISBN: 978-1-4503-2376-5. DOI: 10.1145/2588555.2594517.

[RO12] Tiark Rompf and Martin Odersky. "Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs". In: *Commun. ACM* 55.6 (2012), pp. 121–130. ISSN: 0001-0782. DOI: 10.1145/2184319.2184345.

[RP19] Karthik Ramachandra and Kwanghyun Park. "BlackMagic: Automatic Inlining of Scalar UDFs into SQL Queries with Froid". In: *Proc. VLDB Endow.* 12.12 (2019), pp. 1810–1813. ISSN: 2150-8097. DOI: 10.14778/3352063.3352072.

[SA16] Calvin Smith and Aws Albarghouthi. "MapReduce Program Synthesis". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI'16. New York, NY, USA: Association for

Computing Machinery, 2016, pp. 326–340. ISBN: 978-1-4503-4261-2. DOI: 10.1145/2908080.2908102.

[Sch+17] Matthias Schlaipfer, Kaushik Rajan, Akash Lal, and Malavika Samak. "Optimizing Big-Data Queries Using Program Synthesis". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP'17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 631–646. ISBN: 978-1-4503-5085-3. DOI: 10.1145/3132747.3132773.

[Sel+89] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. "Access Path Selection in a Relational Database Management System". In: *Readings in Artificial Intelligence and Databases*. Ed. by John Mylopolous and Michael Brodie. San Francisco (CA): Morgan Kaufmann, 1989, pp. 511–522. ISBN: 978-0-934613-53-8. DOI: 10.1016/B978-0-934613-53-8.50038-8.

[SG12] Rishabh Singh and Sumit Gulwani. "Learning Semantic String Transformations from Examples". In: *Proc. VLDB Endow.* 5.8 (2012), pp. 740–751. ISSN: 2150-8097. DOI: 10.14778/2212351.2212356.

[SGM16] Subhajit Sidhanta, Wojciech Golab, and Supratik Mukhopadhyay. "OptEx: A Deadline-Aware Cost Optimization Model for Spark". In: *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE. 2016, pp. 193–202. ISBN: 978-1-5090-2453-7. DOI: 10.1109/CCGrid.2016.10.

[Shi+19] Xuanhua Shi, Zhixiang Ke, Yongluan Zhou, Hai Jin, Lu Lu, Xiong Zhang, Ligang He, Zhenyu Hu, and Fei Wang. "Deca: A Garbage Collection Optimizer for In-Memory Data Processing". In: *ACM Trans. Comput. Syst.* 36.1 (2019). ISSN: 0734-2071. DOI: 10.1145/3310361.

[Sim+14] Varun Simhadri, Karthik Ramachandra, Arun Chaitanya, Ravindra Guravannavar, and S. Sudarshan. "Decorrelation of user defined function invocations in queries". In: *2014 IEEE 30th International Conference on Data Engineering*. 2014, pp. 532–543. ISBN: 978-1-4799-2555-1. DOI: 10.1109/ICDE.2014.6816679.

[Sin16] Rishabh Singh. "BlinkFill: Semi-Supervised Programming by Example for Syntactic String Transformations". In: *Proc. VLDB Endow.* 9.10 (2016), pp. 816–827. ISSN: 2150-8097. DOI: 10.14778/2977797.2977807.

[Sin+17] Rohit Singh, Venkata Vamsikrishna Meduri, Ahmed Elmagarmid, Samuel Madden, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Armando Solar-Lezama, and Nan Tang. "Synthesizing Entity Matching Rules by Examples". In: *Proc. VLDB Endow.* 11.2 (2017), pp. 189–202. ISSN: 2150-8097. DOI: 10.14778/3149193.3149199.

[Siv+16] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. "Packet Transactions: High-Level Programming for Line-Rate Switches". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM'16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 15–28. ISBN: 978-1-4503-4193-6. DOI: 10.1145/2934872.2934900.

[SL+06] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. "Combinatorial Sketching for Finite Programs". In: *SIGARCH Comput. Archit. News* 34.5 (2006), pp. 404–415. ISSN: 0163-5964. DOI: 10.1145/1168919.1168907.

[SL08] Armando Solar-Lezama. *Program synthesis by sketching*. University of California, Berkeley, 2008.

[Sou+14] Marcelo Sousa, Isil Dillig, Dimitrios Vytiniotis, Thomas Dillig, and Christos Gkantsidis. "Consolidation of Queries with User-Defined Functions". In: PLDI'14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 554–564. ISBN: 978-1-4503-2784-8. DOI: 10.1145/2594291.2594305.

[SSA13] Eric Schkufza, Rahul Sharma, and Alex Aiken. "Stochastic Superoptimization". In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS'13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 305–316. ISBN: 978-1-4503-1870-9. DOI: 10.1145/2451116.2451150.

[Suj+14] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. "Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages". In: *ACM Trans. Embed. Comput. Syst.* 13.4s (2014). ISSN: 1539-9087. DOI: 10.1145/2584665.

[Tak+21] Keita Takenouchi, Takashi Ishio, Joji Okada, and Yuji Sakata. “PATSQL: Efficient Synthesis of SQL Queries from Example Tables with Quick Inference of Projected Columns”. In: *Proc. VLDB Endow.* 14.11 (2021), pp. 1937–1949. ISSN: 2150-8097. DOI: 10.14778/3476249.3476253.

[TCP09] Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. “Query by Output”. In: SIGMOD’09. New York, NY, USA: Association for Computing Machinery, 2009, pp. 535–548. ISBN: 978-1-60558-551-2. DOI: 10.1145/1559845.1559902.

[Tea14] R Core Team. “R: A language and environment for statistical computing.” In: *MSOR connections* 1 (2014).

[Tha+21] Aalok Thakkar, Aaditya Naik, Nathaniel Sands, Rajeev Alur, Mayur Naik, and Mukund Raghothaman. “Example-Guided Synthesis of Relational Queries”. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 1110–1125. ISBN: 978-1-4503-8391-2. DOI: 10.1145/3453483.3454098.

[TPC05] TPC. *TPC-H Benchmark Specification*. 2005. URL: http://www.tpc.org.

[UM19] Sebastian Ullrich and Leonardo de Moura. “Counting Immutable Beans: Reference Counting Optimized for Purely Functional Programming”. In: *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages*. IFL’19. New York, NY, USA: Association for Computing Machinery, 2019. ISBN: 978-1-4503-7562-7. DOI: 10.1145/3412932.3412935.

[VG+20] Jacob Van Geffen, Luke Nelson, Isil Dillig, Xi Wang, and Emina Torlak. “Synthesizing JIT Compilers for In-Kernel DSLs”. In: *Computer Aided Verification*. Ed. by Shuvendu K. Lahiri and Chao Wang. Cham: Springer International Publishing, 2020, pp. 564–586. ISBN: 978-3-030-53291-8. DOI: 10.1007/978-3-030-53291-8_29.

[Wad88] Philip Wadler. “Deforestation: Transforming Programs to Eliminate Trees”. In: *ESOP ’88, 2nd European Symposium on Programming, Nancy, France, March 21-24, 1988, Proceedings*. Ed. by Harald Ganzinger. Vol. 300. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 344–358. ISBN: 978-3-540-38941-5. DOI: 10.1007/3-540-19027-9_23.

[Wan+19a] Kewen Wang, Mohammad Maifi Hasan Khan, Nhan Nguyen, and Swapna Gokhale. "Design and implementation of an analytical framework for interference aware job scheduling on apache spark platform". In: *Cluster Computing* 22.1 (2019), pp. 2223–2237. DOI: 10.1007/s10586-017-1466-3.

[Wan+19b] Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. "Synthesizing Database Programs for Schema Refactoring". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 286–300. ISBN: 978-1-4503-6712-7. DOI: 10.1145/3314221.3314588.

[Wan+20] Yuepeng Wang, Rushi Shah, Abby Criswell, Rong Pan, and Isil Dillig. "Data Migration Using Datalog Program Synthesis". In: *Proc. VLDB Endow.* 13.7 (2020), pp. 1006–1019. ISSN: 2150-8097. DOI: 10.14778/3384345.3384350.

[WC07] Ben Wiedermann and William R. Cook. "Extracting Queries by Static Analysis of Transparent Persistence". In: POPL'07. New York, NY, USA: Association for Computing Machinery, 2007, pp. 199–210. ISBN: 978-1-59593-575-5. DOI: 10.1145/1190216.1190248.

[WCB17] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. "Synthesizing Highly Expressive SQL Queries from Input-Output Examples". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. New York, NY, USA: Association for Computing Machinery, 2017, pp. 452–466. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062365.

[WCV11] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. "The NumPy Array: A Structure for Efficient Numerical Computation". In: *Computing in Science Engineering* 13.2 (2011), pp. 22–30. DOI: 10.1109/MCSE.2011.37.

[WIC08] Ben Wiedermann, Ali Ibrahim, and William R. Cook. "Interprocedural Query Extraction for Transparent Persistence". In: *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*. OOPSLA'08. New York, NY, USA: Association for Computing Machinery, 2008, pp. 19–36. ISBN: 978-1-60558-215-3. DOI: 10.1145/1449764.1449767.

[Xu+19] Guoqing Harry Xu, Margus Veanes, Michael Barnett, Madan Musuvathi, Todd Mytkowicz, Ben Zorn, Huan He, and Haibo Lin. “Niijima: Sound and Automated Computation Consolidation for Efficient Multilingual Data-Parallel Pipelines”. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP’19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 306–321. ISBN: 978-1-4503-6873-5. DOI: 10.1145/3341301.3359649.

[Yag+16] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. “Synthesizing Transformations on Hierarchically Structured Data”. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 508–521. ISBN: 978-1-4503-4261-2. DOI: 10.1145/2908080.2908088.

[Yag+17] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. “SQLizer: Query Synthesis from Natural Language”. In: *Proc. ACM Program. Lang.* 1.OOPSLA (2017). DOI: 10.1145/3133887.

[YWD18] Navid Yaghmazadeh, Xinyu Wang, and Isil Dillig. “Automated Migration of Hierarchical Data to Relational Tables Using Programming-by-Example”. In: *Proc. VLDB Endow.* 11.5 (2018), pp. 580–593. ISSN: 2150-8097. DOI: 10.1145/3177732.3177735.

[Zah+10] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. “Spark: Cluster Computing with Working Sets”. In: *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud’10. USA: USENIX Association, 2010, p. 10.

[Zah+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for in-Memory Cluster Computing”. In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. NSDI’12. USA: USENIX Association, 2012, p. 2.

[Zha+21] Guoqiang Zhang, Yuanchao Xu, Xipeng Shen, and Işıl Dillig. “UDF to SQL Translation through Compositional Lazy Inductive Synthesis”. In: *Proc. ACM Program. Lang.* 5.OOPSLA (2021). DOI: 10.1145/3485489.

[ZS13] Sai Zhang and Yuyin Sun. “Automatically synthesizing sql queries from input-output examples”. In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2013, pp. 224–234. ISBN: 978-1-4799-0215-6. DOI: 10.1109/ASE.2013.6693082.

APPENDICES

APPENDIX

A

SUPPORTED SPARK RDD APIS

Our tool RDD2SQL supports translating the following Spark RDD APIs to SQL: aggregate, combineByKey, count, countByValue, distinct, filter, flatMap, flatMapValues, foldByKey, groupBy, groupByKey, join, map, mapPartitions, mapValues, mean, reduceByKey, reduceByKeyLocally, union, and values.

We choose to support these APIs because they are commonly used. There is no fundamental difficulty to extend RDD2SQL to support other RDD APIs.

APPENDIX

B

TRANSLATING TARGET LANGUAGE TO SQL

Fig. B.1 presents the rules that we use to translate our target language (shown in Fig. 5.5) to SQL. The following example shows how the rules are applied during translation.

Example 10. *Consider the following query Q in our target language:*

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

Table B.1 presents a step-by-step translation of Q to SQL. In the table, each row applies a rule for the current relational operation and recursively uses the next row to translate its subquery. The resulting SQL S is

```
SELECT c2, sum(c1) FROM (
   SELECT * FROM (
      SELECT E1 AS c1, E2 AS c2 FROM (T)
   ) WHERE c1 > 0
) GROUP BY c2
```

$$\textsc{Project}\quad \frac{Q \rightsquigarrow S}{\texttt{Project}(Q, \mathcal{E}) \rightsquigarrow \texttt{SELECT}\ \mathcal{E}\ \texttt{FROM}\ (S)}$$

$$\textsc{Select}\quad \frac{Q \rightsquigarrow S}{\texttt{Select}(Q, E) \rightsquigarrow \texttt{SELECT} * \texttt{FROM}\ (S)\ \texttt{WHERE}\ E}$$

$$\textsc{Join}\quad \frac{\begin{array}{cc} Q_1 \rightsquigarrow S_1 & Q_2 \rightsquigarrow S_2 \\ C_1 = \mathsf{FirstColumn}(Q_1) & C_2 = \mathsf{FirstColumn}(Q_2) \end{array}}{\texttt{Join}(Q_1, Q_2) \rightsquigarrow \texttt{SELECT} * \texttt{FROM}\ (S_1)\ \texttt{AS x},\ (S_2)\ \texttt{AS y}\ \texttt{WHERE x}.C_1\ = \texttt{y}.C_2}$$

$$\textsc{Agg}\quad \frac{Q \rightsquigarrow S \quad \mathcal{E}' = \mathcal{C} \cup \mathcal{E}}{\texttt{Aggregate}(Q, \mathcal{C}, \mathcal{E}) \rightsquigarrow \texttt{SELECT}\ \mathcal{E}'\ \texttt{FROM}\ (S)\ \texttt{GROUP BY}\ \mathcal{C}}$$

$$\textsc{Union}\quad \frac{Q_1 \rightsquigarrow S_1 \quad Q_2 \rightsquigarrow S_2}{\texttt{Union}(Q_1, Q_2) \rightsquigarrow \texttt{SELECT} * \texttt{FROM}\ (S_1)\ \texttt{UNION}\ (S_2)}$$

Figure B.1: Rules for translating target language (Fig. 5.5) to SQL

Table B.1: Example of translating target language to SQL (Example 10)

Query	**Rule**	**SQL**
$Q = \mathsf{Aggregate}(Q_1, c_2, sum(c_1))$	AGG	$S = \texttt{SELECT}\ c_2, sum(c_1)\ \texttt{FROM}\ (S_1)\ \texttt{GROUP BY}\ c_2$
$Q_1 = \mathsf{Select}(Q_2, c_1 > 0)$	SELECT	$S_1 = \texttt{SELECT} * \texttt{FROM}\ (S_2)\ \texttt{WHERE}\ \texttt{c}_1 > 0$
$Q_2 = \mathsf{Project}(T, E_1\ AS\ c_1, E_2\ AS\ c_2)$	PROJECT	$S_2 = \texttt{SELECT}\ E_1\ \texttt{AS}\ c_1, E_2\ \texttt{AS}\ c_2\ \texttt{FROM}\ (T)$