



AMERICAN UNIVERSITY OF BEIRUT

COMPONENT AND TRANSFORMATION BASED  
FRAMEWORKS FOR BUILDING AND OPTIMIZING  
SPARK PROGRAMS

by  
ZEINAB HASAN SHMEISS

A thesis  
submitted in partial fulfillment of the requirements  
for the degree of Master of Science  
to the Department of Computer Science  
of the Faculty of Arts and Sciences  
at the American University of Beirut

Beirut, Lebanon

AMERICAN UNIVERSITY OF BEIRUT

COMPONENT AND TRANSFORMATION BASED  
FRAMEWORKS FOR BUILDING AND OPTIMIZING  
SPARK PROGRAMS

by  
ZEINAB HASAN SHMEISS

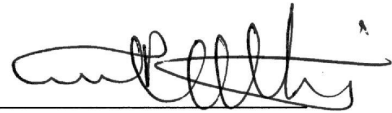
Approved by:

\_\_\_\_\_  
Dr. Mohamad Jaber, Assistant Professor  
Computer Science



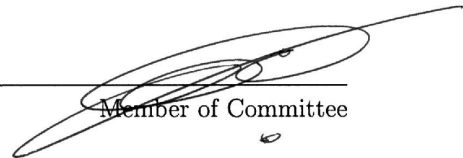
Advisor

\_\_\_\_\_  
Dr. Paul Attie, Professor  
Computer Science



Member of Committee

\_\_\_\_\_  
Dr. Mohamed Nassar, Assistant Professor  
Computer Science



Member of Committee

Date of thesis defense: February 8, 2018

# AMERICAN UNIVERSITY OF BEIRUT

## THESIS, DISSERTATION, PROJECT RELEASE FORM

Student Name: Shmeiss Zeinab Hasan  
Last First Middle

Master's Thesis       Master's Project       Doctoral Dissertation

I authorize the American University of Beirut to: (a) reproduce hard or electronic copies of my thesis, dissertation, or project; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.

I authorize the American University of Beirut, to: (a) reproduce hard or electronic copies of it; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes after: **One \_\_\_ year from the date of submission of my thesis, dissertation or project.**  
**Two \_\_\_ years from the date of submission of my thesis, dissertation or project.**  
**Three \_\_\_ years from the date of submission of my thesis, dissertation or project.**

  
Signature

12 Feb 2018  
Date

This form is signed when submitting the thesis, dissertation, or project to the University Libraries

## ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor Prof. Mohamad Jaber for his continuous support and patience, for feeding my hungry mind with his immense knowledge, for believing in me even when I lost the belief in myself, and most importantly for making me the strong person I am today. I could not imagine having a better advisor and mentor than him.

I would also like to thank my family: my parents, brother, and sister for providing me with the unfailing support and continuous encouragement throughout this thesis and my life in general. This accomplishment would not have been possible without them.

# AN ABSTRACT OF THE THESIS OF

Zeinab Hasan Shmeiss for Master of Science  
Major: Computer Science

Title: Component and Transformation Based Frameworks for Building and Optimizing Spark Programs

Spark is the leading platform for distributed large-scale data processing. It is designed with two main features: (1) an in-memory data engine that makes it uniquely faster than other systems (e.g., Hadoop MapReduce), and (2) a distributed programming model with an extensible, easy-to-use API supported by Scala, Java, R, and Python. Despite these features, writing efficient and complex Spark applications is still error-prone, time-consuming, and requires a clear and deep understanding of the inner-workings of Spark. For instance, (1) Spark does not support composition of distributively developed Spark applications; (2) it lacks automatic persisting/-caching of distributed data sets for reuse across several operations; and (3) the same task can be implemented in several different ways, with significantly different execution times. The contribution of the thesis is twofold. First, we propose a component-based framework for composing independently developed Spark applications. The framework takes as input a set of sub-Spark applications embedded with input/output interfaces for exchanging datasets, and a configuration file defining the dependencies between these interfaces. Then, it automatically merges them into a single monolithic Spark application. We support our framework with several automatic persisting strategies to optimize the execution of the produced Spark application. Second, we present TaBOS, a transformation-based optimizer for Spark programs. TaBOS takes a Spark program and generates a state-space of semantically equivalent programs by applying a set of rewrite rules. A single rewrite rule replaces a fragment in the program with a new one aiming at performance optimization while preserving its semantics. From the generated state-space, TaBOS selects one optimal program based on a predefined strategy. We introduce several selection strategies (e.g., applying maximum number of transformations, a program with minimum number of heavy operations, prune-search techniques) for identifying an optimal program from the generated state-space. We evaluate the effectiveness, robustness and speedup gain of our solutions on several case studies.

# Contents

ACKNOWLEDGEMENTS . . . . .	v
ABSTRACT . . . . .	vi
LIST OF FIGURES . . . . .	ix
LIST OF TABLES . . . . .	x
LIST OF ABBREVIATIONS . . . . .	xi
1 INTRODUCTION . . . . .	1
1.1 Problem Definition . . . . .	1
1.2 Our Approach . . . . .	3
1.3 Thesis Organization . . . . .	4
2 APACHE SPARK FRAMEWORK . . . . .	5
2.1 Overview . . . . .	5
2.2 Spark Architecture Overview . . . . .	5
2.3 Resilient Distributed Datasets . . . . .	7
2.4 Spark Execution Engine . . . . .	9
2.5 Fault Tolerance . . . . .	10
2.6 RDD Persisting . . . . .	10
3 COMPONENT BASED SPARK . . . . .	12
3.1 Overview . . . . .	12
3.2 Spark Composition Model . . . . .	13
3.2.1 Place Holder Instructions . . . . .	13
3.2.2 Sub-Spark Application . . . . .	13
3.2.3 Configuration . . . . .	14
3.3 Spark Composition Design . . . . .	15
3.3.1 Sub-Spark Program Structure . . . . .	15
3.3.2 DSL for the Configuration File . . . . .	16
3.3.3 Semantics and Code Generation . . . . .	17
3.4 Automatic Persisting of Output RDDs . . . . .	18
3.4.1 Persisting Strategy . . . . .	19
3.4.2 Persisting Task . . . . .	20
3.4.3 Benchmarks . . . . .	20

4	TRANSFORMATION BASED OPTIMIZER FOR SPARK . . . . .	26
4.1	Overview . . . . .	26
4.2	Modeling Spark Jobs . . . . .	27
4.3	Rewriting System . . . . .	28
4.3.1	Preliminaries . . . . .	28
4.3.2	Rewrite Rules . . . . .	29
4.3.2.1	Transformation Fusion . . . . .	30
4.3.2.2	Redundancy Elimination . . . . .	31
4.3.2.3	Transformations Reordering . . . . .	31
4.3.2.4	Transformation Action Reordering . . . . .	34
4.3.2.5	GroupBy-Aggregate . . . . .	34
4.4	Code Transformation . . . . .	35
4.4.1	Synthesis Phase . . . . .	36
4.4.2	Selection Phase . . . . .	36
4.4.2.1	Maximum Weight Strategy . . . . .	37
4.4.2.2	Minimum Cost Strategy . . . . .	37
4.5	Implementation . . . . .	39
5	EVALUATION OF TABOS . . . . .	42
5.1	Experimental Design . . . . .	42
5.2	Evaluation of Rewrite Rules . . . . .	43
5.3	Case Study . . . . .	51
5.3.1	State-Space Size . . . . .	52
5.3.2	Comparison of Selection Strategies . . . . .	53
6	RELATED WORK . . . . .	58
6.1	Spark Program Synthesis . . . . .	58
6.2	Data Flow Optimization . . . . .	59
6.3	MapReduce Programs Modeling and Optimization . . . . .	60
6.4	Spark Program Optimization . . . . .	60
6.5	Spark Programs Composition . . . . .	62
6.6	Spark RDD Automatic Checkpointing . . . . .	62
7	CONCLUSION AND FUTURE WORK . . . . .	64
7.1	Conclusion . . . . .	64
7.2	Future Work . . . . .	65



## List of Figures

2.1	Architecture View of the Components of a Spark Cluster . . . . .	6
2.2	A Diagram of the Data Processing Ecosystem Including Spark . . . . .	6
2.3	Spark Components . . . . .	7
2.4	Dependencies Between Partitions for Narrow v.s. Wide Transformations	8
3.1	Design of Component Based Spark . . . . .	17
3.2	Comparison of Several Persisting Strategies for Scenario 1 . . . . .	21
3.3	The Overlapping Between the Scopes of the Output RDDs . . . . .	21
3.4	Comparison of Several Persisting Strategies for Scenario 2 . . . . .	22
3.5	Comparison of Several Persisting Strategies for Scenario 2 on 8GB dataset . . . . .	22
3.6	Simulated Scenario 3 . . . . .	23
3.7	Graph Built Given the Simulated Scenario . . . . .	24
3.8	The Final Sequence <i>SA</i> of Scneario 3 . . . . .	25
3.9	Comparison of Several Persisting Strategies for Scenario 3 . . . . .	25
4.1	Synthesis Rules . . . . .	36
4.2	High Level Architecture of TaBOS . . . . .	40
5.1	Average Execution Time of Scenario 1 Under Various Input Data Sizes	44
5.2	Average Execution Time of Scenario 2 Under Various Input Data Sizes	45
5.3	Average Execution Time of Scenario 3 Under Various Input Data Sizes	46
5.4	Average Execution Time of Scenario 4 Under Various Input Data Sizes	48
5.5	Average Execution Time of Scenario 5 Under Various Input Data Sizes	49
5.6	Average Execution Time of Scenario 6 Under Various Input Data Sizes	51
5.7	Average Execution Time of the Case Study Under Various Input Data Sizes on Local Machine . . . . .	57
5.8	Average Execution Time of the Case Study for 50 GB Dataset Size on a Cluster . . . . .	57

## List of Tables

4.1	Alternative Transformation Sequences for Operations of the Iterator Class in Scala . . . . .	35
5.1	Count of Applied Rewrite Rules in the Case Study . . . . .	53

## List of Abbreviations

CBD .....	Component-Based Development
DAG .....	Directed Acyclic Graph
DSL .....	Domain Specific Language
GC .....	Garbage Collector
HDFS .....	Hadoop Distributed File System
IR .....	Intermediate Representation
RDD .....	Resilient Distributed Dataset
UDF .....	User Defined Function
YARN .....	Yet Another Resource Negotiator

# Chapter 1

## INTRODUCTION

### *Contents*

1.1	Problem Definition . . . . .	1
1.2	Our Approach . . . . .	3
1.3	Thesis Organization . . . . .	4

### 1.1 Problem Definition

The huge interest in Big Data has advanced the development of distributed computing frameworks that can abstract away parallelization, data distribution, fault tolerance, and load balancing. Google’s MapReduce programming paradigm [1] was the first contribution to efficiently process data on a cluster. It is based on two simple primitives: `map` and `reduce`. The `map` transforms a key-value pair to list of intermediate key-value pairs, which are shuffled through the network. Then the `reduce` takes an intermediate key and a list of all the intermediate values for that key and merge the values to produce the final output of the program.

Apache Hadoop [2, 3, 4] is an open source implementation of MapReduce, it is made of three building blocks: (1) the Hadoop distributed file system (HDFS) responsible for storing and replicating data on a cluster of machines, (2) YARN used for job’s scheduling and monitoring, and (3) the MapReduce API used to write map and reduce jobs. While Hadoop has its advantages, it still suffers from some major limitations. First, it is unable to support iterative algorithm because of the high latency introduced by the shuffling phase, which requires network communication and disk operations, between each `map` and `reduce`. Second, it lacks expressiveness, as it can only support two operations, `map` and `reduce`, and requires the developer to write a *Mapper* and *Reducer* classes for each program.

Apache Spark [5, 6] was introduced in 2010 to overcome the limitations of Hadoop. Since its release, it evolved to be one of the most active open-source Apache projects with a huge developers and users community. Spark is uniquely

fast, it supports the sharing of data in-memory between executions, which makes it well suited for the iterative algorithms used in machine learning and graph processing. Moreover, Spark has an extensible easy to use API, it provides richer and more composable operations than those of Hadoop. Despite all the facilities offered by Spark, there are still open potentials to automate programming tasks that can increase the productivity of the programmer. We mainly distinguish two directions to improve the development of Spark programs: (1) by allowing Spark programs to be decomposed and written by several programmers, which can be achieved through the use of component-based development, and (2) by automatically applying source-to-source transformation to optimize Spark programs.

Component-based development (CBD) [7] is a branch in software engineering that builds applications by composing components (black boxes) developed by distributed teams. This composition is based on the relationships between the externally visible properties of the components. CBD has several advantages when it comes to developing complex applications: (1) teams are able to work independently, (2) components are easier to understand and modify as they form smaller applications, (3) components can be re-used within different applications, and (4) the time spent in the integration phase is well reduced. By combining CBD concept with Spark, we can unleash programmers from the complications encountered while developing complex monolithic Spark applications.

On the other hand, implementing Spark programs can be extremely tricky, as you can have two different implementations of the same task, yet their execution time can vary drastically, even when executed under the same environment. Listing 1.1 shows an example of two different implementations for computing the number of trips and the amount of money spent by each individual customer for a flight company. On a dataset composed of 20 million record, the first implementation that uses `groupByKey` takes 15.48 sec, while the second implementation which uses `reduceByKey` only takes 4.65 sec. The difference in execution time falls behind the functionality of the `groupByKey` and the `reduceByKey` operations. The authors in [8] discusses many optimization techniques that can be used to write the optimal code for a given task, however, learning those techniques is a time-consuming and a hardly predictable task. To tackle this issue, we need to use program

optimization, a subscenario of program transformation [9, 10], that modifies the structure of a program in order to improve its runtime and/or its space performance.

```
case class Purchase(customerId: Int, price: Double)

val purchasesRDD: RDD[Purchase] = sc.textFile(...)

// using groupByKey
val purPerMonth1 = purchasesRDD.map(p => (p.customerId, p.price))
                                .groupByKey()
                                .map(p => (p._1, (p._2.size, p._2.sum)))
                                .count

// using reduceByKey
val purPerMonth2 = purchasesRDD.map(p => (p.customerId, (1, p.price)))
                                .reduceByKey((v1, v2) => {
                                    (v1._1 + v2._1, v1._2 + v2._2)
                                }).count
```

Listing 1.1: An Example of Two Spark Implementations for the Same Task

Briefly speaking, the contribution of the thesis is twofold. First, we propose a CBD framework for composing independently developed Spark applications. Second, we introduce a transformation-based system to optimize Spark programs.

## 1.2 Our Approach

In the first part of the thesis, we seek to combine CBD with Spark. A Spark application is built using a set of independently developed sub-Spark applications with predefined dependencies. A sub-Spark application is embedded with placeholder instructions that declare waiting or outsourcing a dataset. These instructions form the input/output interfaces and the only visible part of the application. Then, a configuration file is provided, which is written using a domain-specific language, to determine the dependencies between the interfaces of the sub-Spark applications. We describe the algorithm used to automatically merge the sub-Spark applications into one monolithic application. We further propose a technique for optimizing the execution of the produced application based on reusing datasets that were previously computed within the same execution.

As for the second part, we introduce TaBOS, an optimizer that automatically transforms a given Spark program to a more efficient and semantically equivalent one. We define a formal model that captures the structure of a Spark program. We then present a set of rewrite rules, which replaces a fragment in the program with a new one aiming at performance optimization while preserving its semantics. We formalize a rewrite system that applies the defined rules to transform a given Spark program into several optimized alternatives. We finally propose several strategies, to select from the alternatives, the most optimized version. We evaluate TaBOS and the rewrite rules defined on several non-trivial examples.

### **1.3 Thesis Organization**

The rest of the thesis is organized as follows. Chapter 2 introduces background information about Apache Spark. Chapter 3 defines component-based Spark’s model and design, along with the automatic (un)persisting method used. Then we shift in chapter 4 to present a detailed description of our optimizer (TaBOS), including: the formal model used for Spark program representation, the rewrite rules, the code transformation algorithm, and some implementation details. Chapter 5 evaluates the rewrite rules and algorithm used by TaBOS on six simple scenarios and two non-trivial case studies. Chapter 6 surveys some of the related works. Finally, Chapter 7 draws conclusion and presents future work.

# Chapter 2

## APACHE SPARK FRAMEWORK

### *Contents*

2.1	Overview . . . . .	5
2.2	Spark Architecture Overview . . . . .	5
2.3	Resilient Distributed Datasets . . . . .	7
2.4	Spark Execution Engine . . . . .	9
2.5	Fault Tolerance . . . . .	10
2.6	RDD Persisting . . . . .	10

### **2.1 Overview**

Apache Spark [5, 8] is a high-performance general-purpose distributed computing framework, initially developed by the University of California, Berkeley’s AMPLab, and was later denoted by the Apache Software Foundation. It extends the popular MapReduce model to efficiently support iterative and interactive applications. Spark’s ability to leverage in-memory computation of immutable data makes it particularly faster than other distributed data processing systems. It is also more expressive than the other systems, as it has a high level and relatively easy to use API composed of highly generalizable methods beyond the traditional map and reduce operations, providing the ability to develop applications in Scala, Java, Python, and R. To get the most out of Spark, it is important to understand Spark’s architecture and execution engine.

### **2.2 Spark Architecture Overview**

Spark framework has a master/slave architecture. The driver program (master) is responsible for running the main function of the Spark program. The executors, which are scattered over the worker nodes (slaves), are responsible for running the parallel computation tasks. When an application is launched, Spark’s driver sets up the executors on the worker nodes and sends them the application code, then a `SparkContext` object is created at the driver process in order to coordinate these



executors and to send them the computation tasks to be executed.

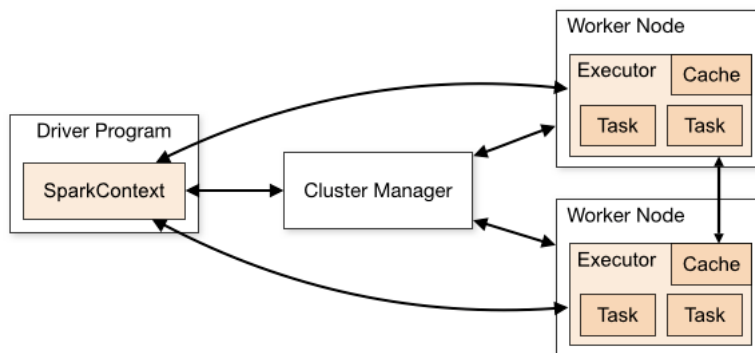


Figure 2.1: Architecture View of the Components of a Spark Cluster

On its own, Spark is not a data storage solution, nor a cluster manager. It is often used on top of an existing distributed storage system (e.g., HDFS, Amazon S3, Cassandra) that house the data to be processed, and in tandem with a cluster manager that is connected with the `SparkContext` object to arrange the distribution of the Spark applications across the cluster. There are currently three types of cluster managers option available: (1) Hadoop Yarn, (2) Apache Mesos, and (3) standalone cluster manager which is a simple cluster manager supplied with Apache Spark and requires the installation of Spark on every node in the cluster. Furthermore, Spark can run in local mode on a single machine that spawns all the execution components in the same single JVM.

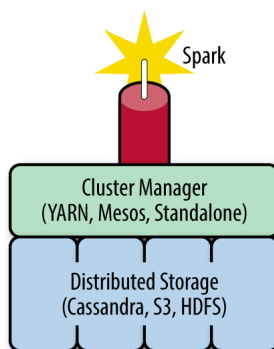


Figure 2.2: A Diagram of the Data Processing Ecosystem Including Spark

Spark supports a rich set of high-level components used for data processing. Spark core is the main one in the Spark ecosystem. It contains the main function-

alities of Spark, such as IO functionalities, task scheduling, memory management, fault recovery. It also provides the basic API for data manipulation. In this thesis, we will be concentrating on this specific component as it the base of the other ones. Other components in the Spark ecosystem that can be found on top of Spark Core are:

- **Spark SQL:** provides a high-level API for extracting and merging various structured and semi-structured data so that they are ready to use for machine learning.
- **Spark MLlib:** provides multiple types of machine learning algorithms including classification, regression, and clustering.
- **Spark Streaming:** a lightweight API that allows developers to perform batch processing and streaming of data with ease, in the same application.
- **GraphX:** a library for creating, manipulating and performing computations on graphs.

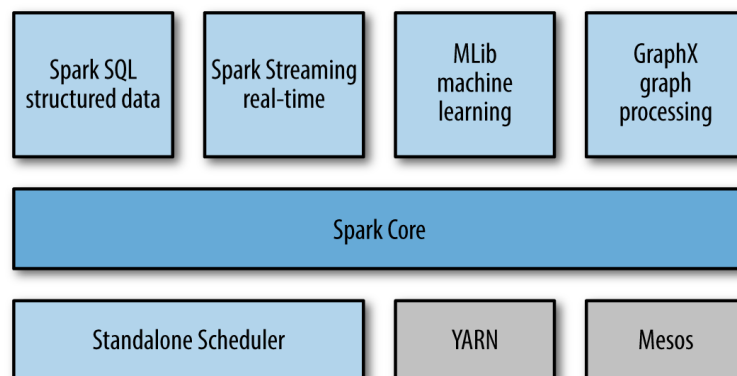


Figure 2.3: Spark Components

### 2.3 Resilient Distributed Datasets

Spark is based on two main abstractions for parallel programming: *resilient distributed datasets* [11] to represent large datasets, and the *parallel operations* on these datasets. A resilient distributed dataset (RDD) is immutable, lazily evaluated, distributed collection of objects partitioned across the cluster's machines with a network in between and operated on in parallel. An RDD can be constructed either by (1) reading a file from a stable storage (e.g., local, HDFS), (2) parallelizing

a Scala collection (e.g., an array) in the driver program, or (3) applying a parallel operation on an existing RDD (e.g., map, filter) which are often known as transformations. RDDs have a number of predefined “coarse-grained” parallel operations, which are simply high order functions that execute a user-defined function (UDF) in a particular manner. They can be distinguished into two categories:

- **Transformations:** are operations applied on an RDD and returns a new RDD. They are lazy evaluated and their result RDD is not immediately computed.
- **Actions:** are operations applied on an RDD and returns a non-RDD result, which is either returned to the driver processor or saved to an external storage system. They are eager operations and they trigger the evaluation of RDD transformations.

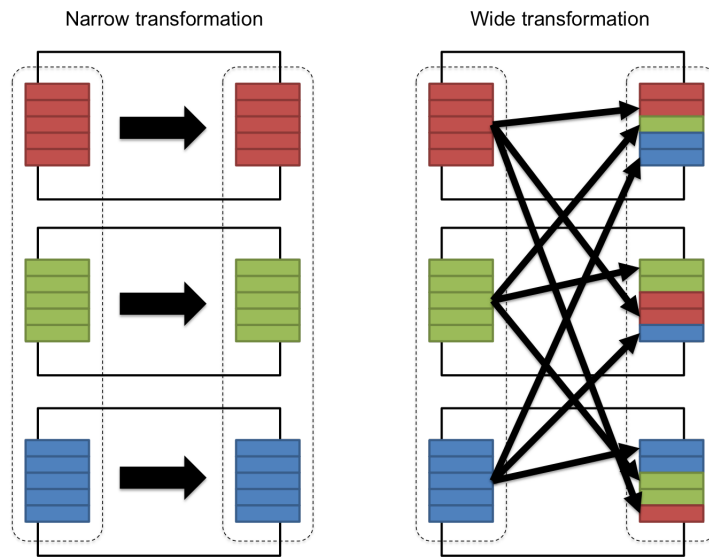


Figure 2.4: Dependencies Between Partitions for Narrow v.s. Wide Transformations

RDD transformations can be further classified into two sets:

- **Transformations with Wide dependencies:** are transformations, such as `groupByKey`, where the partitions in the child RDD (the resulting RDD) have dependencies on an arbitrary set of partitions in the parent RDD. These type of transformations require data to be shuffled and partitioned according to the operation (e.g., according to key). As such, downstream computation cannot

be computed before this transformation ends which can introduce latency to the execution time of the application.

- **Transformations with Narrow dependencies:** are transformations, such as map and filter, where each partition in the child RDD has simple and finite dependencies on partitions in the parent RDD. So narrow transformations can be executed on an arbitrary subset of the data without any information about the other partitions. Thus they can be pipelined and intermediate results can be kept in memory.

## 2.4 Spark Execution Engine

As we have already mentioned, Spark transformations are lazily evaluated and only executed once an action is encountered by the driver program. Meanwhile, Spark preserves the logical execution plan (aka. lineage), which is the sequence of transformations applied from a given data source to the current RDD. Once an action is called, the DAGScheduler, which is the scheduling layer of Spark that implements stage-oriented scheduling, build the physical execution plan, in the form of a directed acyclic graph (DAG), from the logical execution plan and launches a Spark Job.

**Spark Job:** is the highest element of Spark's execution hierarchy. It is a directed acyclic graph with nodes representing the computed RDDs and edges representing the dependencies between the partition in RDD transformations, thus an operation that returns something other than an RDD cannot have any children. Then the Spark job is breakdown into stages based on the wide transformation encountered.

**Stages:** Each stage corresponds to a shuffle dependency created by a wide transformation in the Spark program. Thus a stage is a sequence of narrow transformations that can be pipelined together. At a high level, it can also be seen as the set of computations (tasks) where each task can be computed on one executor without the need to communicate with other executors or the driver process. And since the stage boundaries require communication with the driver, the stages associated with the same job must be executed in sequence and not in

parallel unless they are used to compute different RDDs that are combined in a downstream transformation such as join.

**Task:** A task is the actual unit of execution in the physical plan and represents one local computation performed by only one executor. All the tasks that correspond to the same stage execute the same code on different partitions of the input RDD, thus the number of tasks performed per stage is equal to the number of partitions in the output RDD of that stage.

## 2.5 Fault Tolerance

Most distributed computing frameworks like Hadoop provide fault tolerance through logging or the replication of data across machines. However, Spark's unique method to achieve fault tolerance is by replaying the computation of the lost partition. With the dependency information preserved by the RDDs, Spark remembers how an RDD was built from some base dataset. This is often known as the RDD lineage, and recalculate the lost partition in case of failure of a host machine or the network. Moreover this calculation can be parallelized to make recovery faster.

## 2.6 RDD Persisting

According to the Spark's execution engine, each Spark job is handled and executed independently of the other jobs even if the two jobs compute the same RDD. Moreover and as we have mentioned, Spark tolerates event failure by recomputing the whole lineage of the failing RDD, which can slow down the execution of the job. For this, Spark offers the capability of persisting/caching an RDD in memory, so that the entire transformations proceeding this persisting are executed only once. There are many cases in which persisting can lead to huge performance gain, some of these cases:

- In iterative computations where a transformation uses the same parent RDD several times, like when performing a loop of joins to the same RDD.
- When multiple actions are performed on the same RDD. In other words, the same RDD is computed within several jobs.

- When the cost of RDD computation is enormous and the chances of downstream failure is high. In this case, it is preferable to cache the RDD outside of Spark's executor memory (`off_heap`).

Spark does not perform the RDD persisting automatically, it should rather be done by the developer using the `persist()` or `cache()` methods on the RDD. The reason why the developer should have a clear understanding of the cost and benefits of such persisting. Furthermore, the developer should also be aware of removing the RDDs from the memory when it is no longer used by the application using the `RDD.unpersist()` method. Deciding whether to persist an RDD or not can form a challenge to the user. First, it is space intensive to store data in memory, as it can take from the space used in downstream computation, increasing the garbage collector (GC) overhead and the risk of memory failure. Second, persisting is a pipeline breaker that prevents transformations with narrow dependencies from being combined into a single task. Therefore, in some situations recomputing the RDD is rather less expensive than storing and reading it. Moreover, there is one special case where Spark automatically persists intermediate RDDs, in case of stage boundaries where a wide transformation (e.g., `reduceByKey`) is applied and a shuffling is performed. It is done to avoid recomputing the entire input if a node fails during the shuffle.

# Chapter 3

## COMPONENT BASED SPARK

### *Contents*

3.1	Overview . . . . .	12
3.2	Spark Composition Model . . . . .	13
3.2.1	Place Holder Instructions . . . . .	13
3.2.2	Sub-Spark Application . . . . .	13
3.2.3	Configuration . . . . .	14
3.3	Spark Composition Design . . . . .	15
3.3.1	Sub-Spark Program Structure . . . . .	15
3.3.2	DSL for the Configuration File . . . . .	16
3.3.3	Semantics and Code Generation . . . . .	17
3.4	Automatic Persisting of Output RDDs . . . . .	18
3.4.1	Persisting Strategy . . . . .	19
3.4.2	Persisting Task . . . . .	20
3.4.3	Benchmarks . . . . .	20

### **3.1 Overview**

Spark applications are getting larger and more complex, calling for a shift from Spark programming to Spark system composing. For this, we present a component-based Spark framework (CBSpark), that integrates the CBD approach with Spark. In our framework, we consider a component to be an independently developed Spark application, we shall call it a sub-Spark application. A sub-Spark application has a well defined input/output interfaces that expresses exchanging computed datasets (e.g., RDD, file path). The framework takes these components as input, along with a configuration file defining the dependencies between their interfaces. Then, based on the defined dependencies, it automatically composes the sub-Spark applications to build a complete final Spark project.

We find this approach to be more efficient than using function calls for two main reasons. First, using functions limit the scope of all the intermediate RDDs,

produced while computing the desired output dataset, to the function’s scope, which makes the developer incapable of using any of them within a different computation in the same sub-Spark program. Second, component-based applications are easier to maintain and modify. For example, updating an input interface point to receive different output dataset only requires modifying the configuration file while keeping the input sub-Spark set unchanged. Furthermore, sub-Spark applications are (1) self-contained, (2) easier to understand, and (3) integrated easily to a Spark project as it only requires knowing the configuration file of the project without examining any of the previously developed code.

## 3.2 Spark Composition Model

### 3.2.1 Place Holder Instructions

A user writing a sub-Spark application can express in/outsourcing a dataset within a specific location of the application using a set of predefined instructions that represent an input/output interface. These instructions are called placeholder instructions and are defined as follows:

*Definition 1. (Place Holder Instructions) A place holder instruction describes waiting input, providing output, or declaring end of scope of input/output place holders:*

- *`val X = input(T)`, indicating that the application is waiting for a dataset of type  $T$  to be used in the upcoming code.*
- *`output(X)`, indicating that the application will provide dataset  $X$  resulting from previous computation to other sub-Spark applications.*
- *`endOfScope(X)`, indicating that the user is no longer using dataset  $X$  in his application<sup>1</sup>.*

### 3.2.2 Sub-Spark Application

Based on the place holder instructions, a sub-Spark application can be decomposed into sequence of computation blocks (set of instructions).

---

1. Note this can be removed by integrating static code analysis.



*Definition 2. (Sub-Spark Application)* A sub-Spark application  $SSA$  is defined by a sequence of blocks  $SSA = (\delta_1, \delta_2, \dots, \delta_n)$ , where each block  $\delta_i$  can either represent a place holder or a computation block.

For a  $n$ -tuple  $t = (t_1, \dots, t_n)$ , we define  $\mathbf{t2s}(t)$  the set containing the elements in tuple  $t$ , i.e.,  $\mathbf{t2s}(t) = \{t_1, \dots, t_n\}$ .

For a given sub-Spark application  $SSA$ , let  $SSA.ins$  (resp.  $SSA.outs$ ) denotes all the input (resp. output) place holder blocks in the sequence  $SSA$ . Formally,

- $SSA.ins = \{\delta_i \in \mathbf{t2s}(SSA) \mid \delta_i \text{ represents an input place holder block}\}$ .
- $SSA.outs = \{\delta_i \in \mathbf{t2s}(SSA) \mid \delta_i \text{ represent an output place holder block}\}$ .

### 3.2.3 Configuration

Given a user defined configuration, sub-Spark applications are composed by mapping input of sub-Spark applications to outputs of other applications.

*Definition 3. (Configuration)* Given a set of sub-Spark applications  $\{SSA_i\}_{i \in I}$ , a configuration  $C$  is a function defined by  $C : Input \rightarrow Output$ , where:

- $Input = \bigcup_{i \in I} SSA_i.ins$ ;
- $Output = \bigcup_{i \in I} SSA_i.outs$ .

For set of sub-Spark applications  $\{SSA_i\}_{i \in I}$  and a configuration  $C$  we define the directed graph  $G = (V, E)$ , where:

- $V = \bigcup_{i \in I} \mathbf{t2s}(SSA_i)$ , is the set of vertices representing the set of blocks for each of the sub-Spark application  $SSA_i$ ;
- $E = L \cup G$ , with:
  - $L = \bigcup_{i \in I} \{(\delta_k^{(i)}, \delta_{k+1}^{(i)}) \mid 1 \leq k < |SSA_i| \wedge SSA_i = (\delta_1^{(i)}, \dots, \delta_{|SSA_i|}^{(i)})\}$ , represent the local edges between blocks in the same sub-Spark application,
  - $G = \{(\delta^{out}, \delta^{in}) \mid i, j \in I \wedge i \neq j \wedge \delta^{out} \in SSA_i.outs \wedge \delta^{in} \in SSA_j.ins \wedge C(\delta^{in}) = \delta^{out}\}$  represent the global edges defined by the mappings in  $C$ .

A configuration  $C$  is said to be valid iff:

- Each input in  $\bigcup_{i \in I} SSA_i.ins$  is mapped to an output with respect to configuration  $C$ . Formally,  $\forall in \in \bigcup_{i \in I} SSA_i.ins : \exists out \in \bigcup_{i \in I} SSA_i.outs \wedge C(in) = out$ ;
- Inputs are mapped to outputs of different applications. Formally,  $\forall in \in SSA_i.ins \wedge out \in SSA_j.outs : C(in) = out \Rightarrow i \neq j$ ;
- The directed graph  $G$  obtained from  $\{SSA_i\}_{i \in I}$  and  $C$  does not contain cycles.

### 3.3 Spark Composition Design

#### 3.3.1 Sub-Spark Program Structure

A sub-Spark program follows the same structure of a typical Spark program. It starts with a header declaration (object and main method header), followed by the configuration of the Spark environment, creating a “SparkContext” object and giving it the corresponding parameters configuration. The rest of the file represents the program’s body, which consists of a sequence of instructions to be executed. Listing 3.1 depicts the grammar for a sub-Spark program. According to the grammar, the sub-Spark program’s body is composed of a sequence of blocks, as described in definition 2, where a block is either a place holder instruction or a set of other instructions representing specific computation.

SSA	→ MAINHEADER '{' SCDEF Body '}'
SCDEF	→ 'val sc = new SparkContext(' .+ ')
Body	→ (Block)+
Block	→ IBlock   OBlock   ESBLOCK   CBlock
IBlock	→ 'val' IDENTIFIER '= input(' Type ')
OBlock	→ 'output(' IDENTIFIER ')
ESBlock	→ 'endOfScope(' IDENTIFIER ')
CBlock	→ .*?
Type	→ RDD[.+]   FILEPATH

Listing 3.1: Sub-Spark File Syntax

### 3.3.2 DSL for the Configuration File

We define a Domain Specific Language (DSL) using JSON representation to describe the location and interfaces of sub-Spark applications and the configuration that connects them. Each sub-Spark application is identified by an identifier *id*, a path where the sub-Spark program exists *path*, number of inputs *ni*, and number of outputs *no*. Then, the configuration maps inputs to the output of other sub-Spark applications. Listing 3.2 depicts the general structure to specify a set of sub-Spark applications and a configuration. It mainly consists of two parts:

- The first part defines the set of sub-Spark applications with their corresponding interfaces (i.e., an identifier, location, number of inputs and number of outputs).
- The second part defines the configuration which connects the sub-Spark applications (i.e., connect inputs to outputs).

```
{ "spark-applications": [
  { "id": "id", "path": "path", "ni": "n", "no": "n" },
  { "id": "id", "path": "path", "ni": "n", "no": "n" },
  ...
  { "id": "id", "path": "path", "ni": "n", "no": "n" },
]}

{ "configuration": [
  { "id": "id",
    "i": ["i1", "i2", ...],
    "o": ["o1", "o2", ...],
  },
  { "id": "id",
    "i": ["i1", "i2", ...],
    "o": ["o1", "o2", ...],
  },
  ...
  { "id": "id",
    "i": ["i1", "i2", ...],
    "o": ["o1", "o2", ...],
  }
]}
}
```

Listing 3.2: General Structure of a Configuration File.

### 3.3.3 Semantics and Code Generation

Figure 3.1 shows the general design of our framework. The framework takes as input a set of sub-Spark application files and a configuration file. The sub-Spark files are transformed by the “Spark File Parser” to a set of sequences  $\{SSA_i\}_{i \in I}$ . Whereas, the “JSON File Parser” extracts the configuration  $C$ , containing the mappings between the input/output interfaces, from the configuration file. Then, the sub-Spark sequences  $\{SSA_i\}_{i \in I}$  and the configuration  $C$  are fed to the composition system that handles the corresponding merging into one single Spark sequence. Listing 3.3 depicts the algorithm used by CBSpark system, it consists of four main steps. The first step builds the graph  $G$  from  $\{SSA_i\}_{i \in I}$  and  $C$  according to the description in the previous section. Once  $G$  is build “`isValid(G)`” checks the validation of configuration  $C$ . If  $C$  passes the validation test,  $G$  is transformed by “`transformToSequence(G)`” into a sequence of blocks ordered by their topological order in  $G$ . Formally:

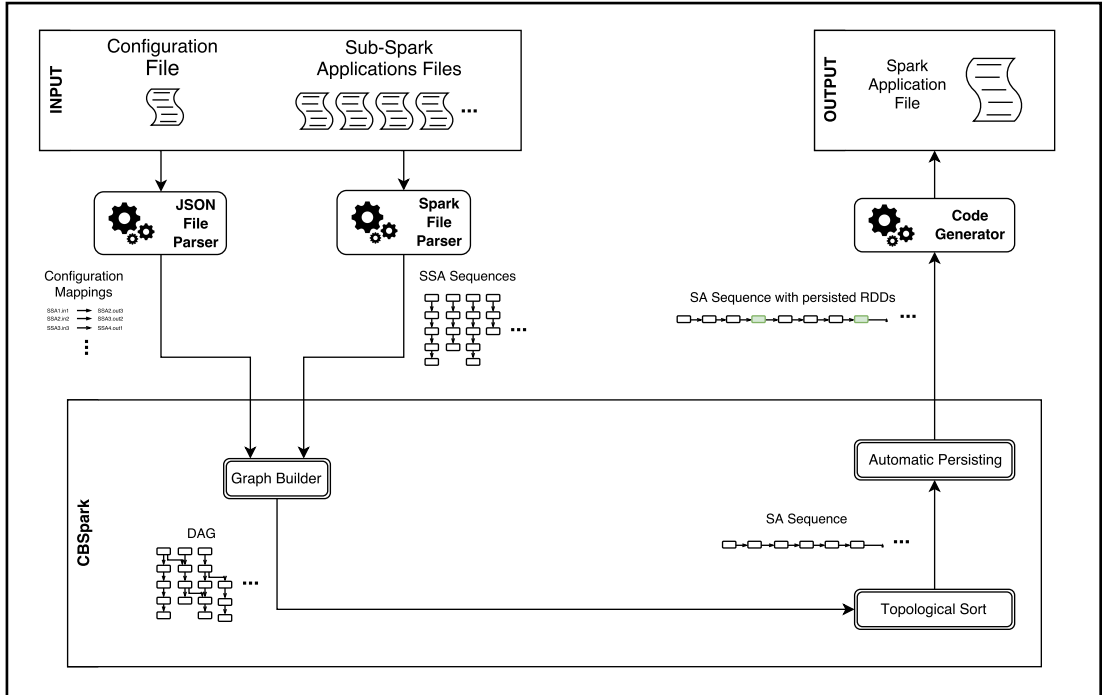


Figure 3.1: Design of Component Based Spark

*Definition 4.* Given a graph  $G$ , a sequence of blocks  $S = (\delta_1, \delta_2, \dots, \delta_n)$  is defined such that, if  $\exists$  a path from  $\delta_i$  to  $\delta_j$  then  $\delta_i$  will come before  $\delta_j$  in  $S$ .

```

G = buildGraph ( {SSAi}i∈I, C );
if( isValid(G) ) {
    SA = transformToSequence(G);
    applyPersistStrategy(SA);
}

```

Listing 3.3: Automatic Merging of Sub-Spark Applications

Notice that several possible sequences can be obtained from the same graph  $S$ , since two blocks may not have a path connecting them (i.e., are independent). However, all the sequences generated based on definition 4 are semantically equivalent, since two blocks that have no path in  $G$  performs computations independent from each other. Hence, the order by which they are executed will not affect the final result of the program.

In the final step of the algorithm, automatic (un)persisting of the output datasets is applied. The (un)persisting of outputs in  $SA$  is explained in details in the next section. Once the final Spark sequence is generated, the “Code Generator” transforms this sequence to the final Spark application file.

### 3.4 Automatic Persisting of Output RDDs

Within a configuration, a dataset that is outsourced by a sub-Spark application can have several inputs mapped to it. In addition, it may be used locally in further downstream computation. To this end, multiple actions can be called on a single output dataset, and since Spark recomputes an RDD each time an action is called on it, the overall execution time of the resulting Spark application can be very slow. In order to avoid such drawback in our composition, we suggest an optimization technique based on RDD reuse by persisting them in-memory, which can introduce a huge improvement in terms of speed. Throughout the rest of this section, we refer to the set of sub-Spark applications as  $\{SSA_i\}_{i \in I}$ , the configuration as  $C$ , the graph build from  $\{SSA_i\}_{i \in I}$  and  $C$  as  $G$ , and the final sequence of blocks for the resulting Spark application as  $SA$ .

Let  $SA.outs$  denotes all the output place holders of the given Spark application. Formally,  $SA.outs = \bigcup_{i \in I} SSA_i.outs$ . For each  $out \in SA.outs$ , we define two main attributes:

1.  $\text{count}(out)$  to denote the number of actions in the reachable computation blocks from the node  $out$  in Graph  $G$ . Formally,

$$\text{count}(out) = \sum_{i \in \text{reachable}(out) \wedge i \text{ is a computation block}} \text{count}(i, out),$$

where,  $\text{count}(i, out)$  returns the number of actions in the computation block  $i$  that includes (directly or indirectly)  $out$  in its lineage.

2.  $\text{ScopeEnd}(out)$  to denote the index of the last **endOfScope** block that corresponds to  $out$  or the inputs mapped to  $out$ . Formally,

$$\text{scopeEnd}(out) = \text{Max}_{i = out \vee C(i) = out} \text{endOfScopeIndex}(i),$$

where,  $\text{endOfScopeIndex}(i)$  returns the index (in the sequence  $SA$ ) of the **endOfScope** block that corresponds to  $i$  (input or output).

### 3.4.1 Persisting Strategy

A persisting strategy defines a criteria that decides for each  $out$  RDD,  $out \in SA.outs$ , whether to be persisted or not. We distinguish between three different persisting strategies:

- Persist All: a naive strategy where every output RDD is selected to be persisted;
- Persist “n” Top Used: in this strategy the “n”  $out \in SA.outs$ , that has the highest  $\text{count}(out)$  values are persisted;
- Persist “n” Top Used Within the Same Scope: this strategy is similar to the previous strategy, except that it takes the scope of the RDD into consideration. For this, the  $out \in SA.outs$ , that has the highest  $\text{count}(out)$  values are persisted, such that no more than “n” RDDs with overlapping scopes are selected at the same time. Therefore, it is possible to persist more than “n” output dataset in the entire  $SA$ .

### 3.4.2 Persisting Task

The automatic (un)persisting of RDDs is performed by injecting the sequence of blocks of  $SA$  with blocks having the `persist()` and `unpersist()` instruction calls corresponding to the selected output datasets. The `persist` action is added just after the output placeholder. Whereas, the `unpersist` action is added after the returned value of the `scopeEnd` function. To this end, the persisting task is defined as a function that takes an  $out \in SA.outs$ , and returns the corresponding index of the `persist` and `unpersist` action, in case  $out$  was selected to be persisted, otherwise it returns a pair of -1. Formally,

*Definition 5. (Persisting Task)* A persisting task is a function defined by  $P : SA.outs \rightarrow \mathbb{N} \times \mathbb{N}$ :

$$P(out) = \begin{cases} (index(out) + 1, scopeEnd(out) + 1), & \text{if } out \in \mathcal{P} \\ (-1, -1), & \text{otherwise} \end{cases}$$

where,  $\mathcal{P}$  represent the set of output RDD that satisfy the defined strategy.

### 3.4.3 Benchmarks

In order to evaluate the automatic persisting approach and the proposed strategies, we have conducted three experimental sets each with different scenario. The first two scenarios are simple scenarios in order to test wither automatic persisting can reduce the total execution time of the composed Spark application. Whereas the third scenario studies how the persisting strategies perform on real world example. All of the experiments in this section were performed using a local machine (Ubuntu 16.04.2) with 4 cores, running Spark 2.2.0, and on a “GeoNames” database [12] covering all countries with over eleven million place names.

**Scenario 1.** In our first scenario, several sub-Spark applications are composed to produce a Spark application with five output RDDs such that the actions invoked on these RDDs do no overlap. We compare the performance of no persisting with `persist` (1, 2, 3, and 4) top used RDDs and `persist all` strategy. Note that, we exclude here the `persist top used within same scope` strategy since we have no overlapping

between the RDDs’ scopes. Figure 3.2 shows the execution time of the Spark application while varying the number of output RDDs persisted and for datasets of sizes ranging from 250MB to 2GB. For all the datasets sizes, the execution time decreases as the number of persisted RDDs increase. This shows that persisting output RDDs can optimize the execution time of the final application produced by our framework.

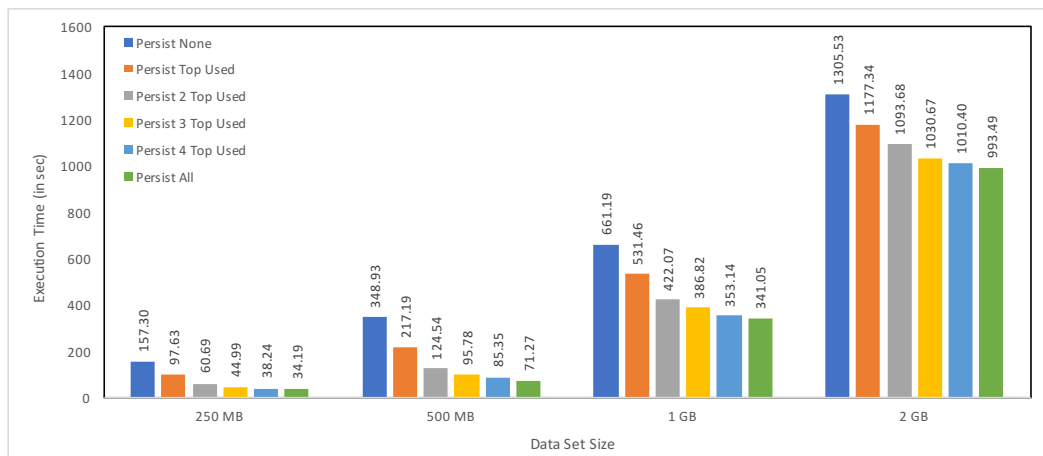


Figure 3.2: Comparison of Several Persisting Strategies for Scenario 1

**Scenario 2.** In this scenario, we modify the sub-Spark applications used in scenario 1 such that the scopes of the output RDDs in the produced Spark application overlap as shown in figure 3.3. In the comparison here, we add two extra strategies: persist top used within the same scope which persists the RDDs  $O_2$ ,  $O_4$ , and  $O_5$ , and persist 2 top used within the same scope which persists the RDDs  $O_1$ ,  $O_2$ , and  $O_3$ . Figure 3.4 shows the execution time of the Spark application for the several persist strategies. The persist all strategy had the best performance. Whereas the worst performance was for the persist top used within same scope strategy, this is because the amount of computation reduced by persisting the three RDDs is low compared to the overhead added in the other computations, i.e., the actions invoked on output  $O_1$ , executed while those three RDDs are in memory.

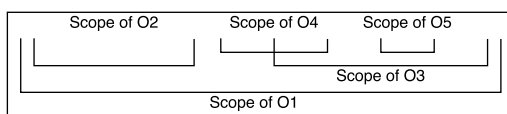


Figure 3.3: The Overlapping Between the Scopes of the Output RDDs



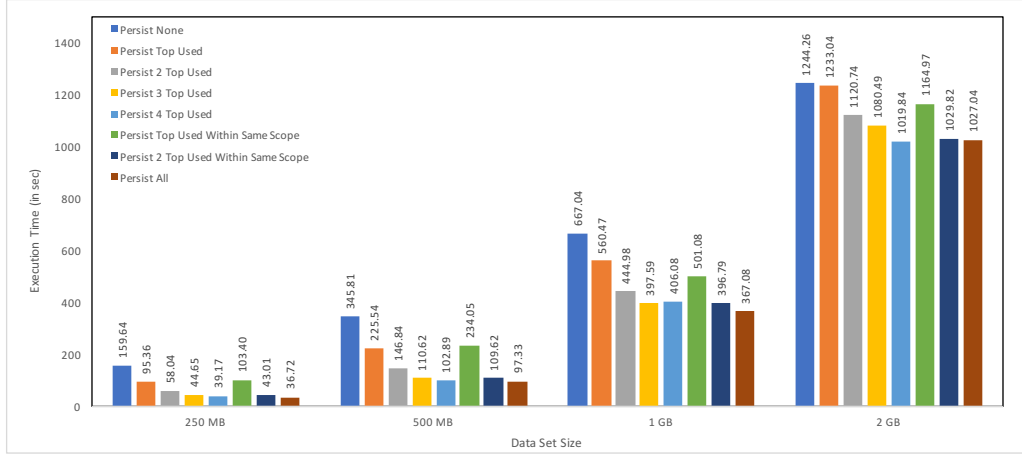


Figure 3.4: Comparison of Several Persisting Strategies for Scenario 2

To test more the influence of the automatic persisting on the application performance we used a bigger dataset (8GB). The resulting execution time is depicted in figure 3.5. The figure shows a change in the performance of the persisting strategies than in previous results. Some of the persisting strategies had higher execution time than that no RDD was persisted, and the persist all strategy no longer give the best performance. After investigating the results, we noticed that the garbage collector (GC) time, in some of the strategies that had a bad performance, was higher than that on smaller datasets. On the other hand, the persist 1 and 2 top used within scope strategy had more consistent results that of all the other strategies. To this end, we can deduce that the GC overhead is a parameter to be taken into consideration for the automatic persisting approach<sup>2</sup>.

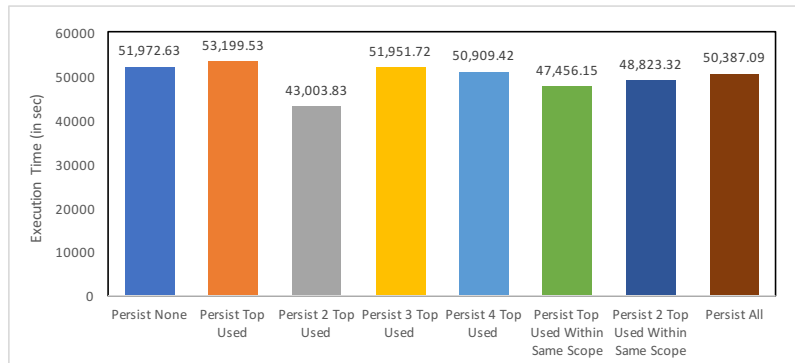


Figure 3.5: Comparison of Several Persisting Strategies for Scenario 2 on 8GB dataset

---

2. This can be fixed using dynamic approaches that monitor the garbage collector during execution.

**Scenario 3.** In our final scenario, we used a more complex example. The example consists of 13 sub-Spark application depicted in figure 3.6b with the configuration shown in Listing 3.6a. Figure 3.7 shows the graph built for the given scenario, the red edges correspond to the given mappings in 3.6a and the black edges correspond to the local ordering between blocks within the same sub-Spark application. Moreover, each output block is labeled with the count of reachable actions (i.e., `count`), and the block that represents the end of scope for this output ends (i.e., `scopeEnd`). For the sake of simplicity, we eliminate the placeholder blocks corresponding to the `endOfScope` instruction and replace it with labels of the directly preceding block. For example, the end of the scope of  $O1$  is block  $C2$ , hence if  $O1$  was selected to be persisted the unpersist instruction should be embedded directly after  $C2$ . Listing 3.8a represent the monolithic Spark program  $SA$  obtained from applying topological sort on the built graph<sup>3</sup>, and Listing 3.8b present the application  $SA$  after applying the persist 3 top used strategy.

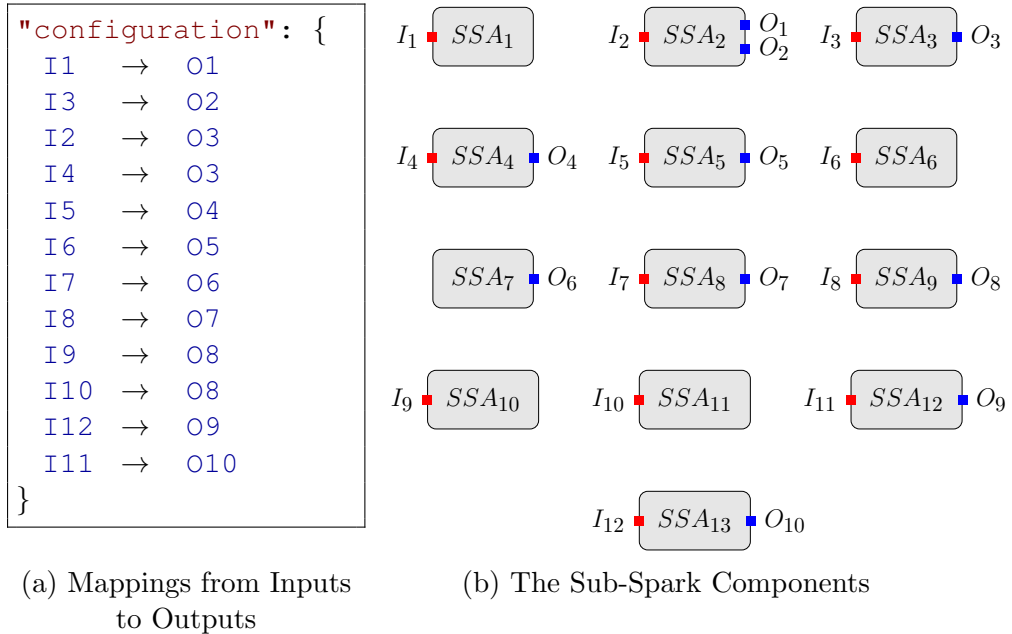


Figure 3.6: Simulated Scenario 3

---

3. Note that variable renaming may be needed to avoid any conflict.

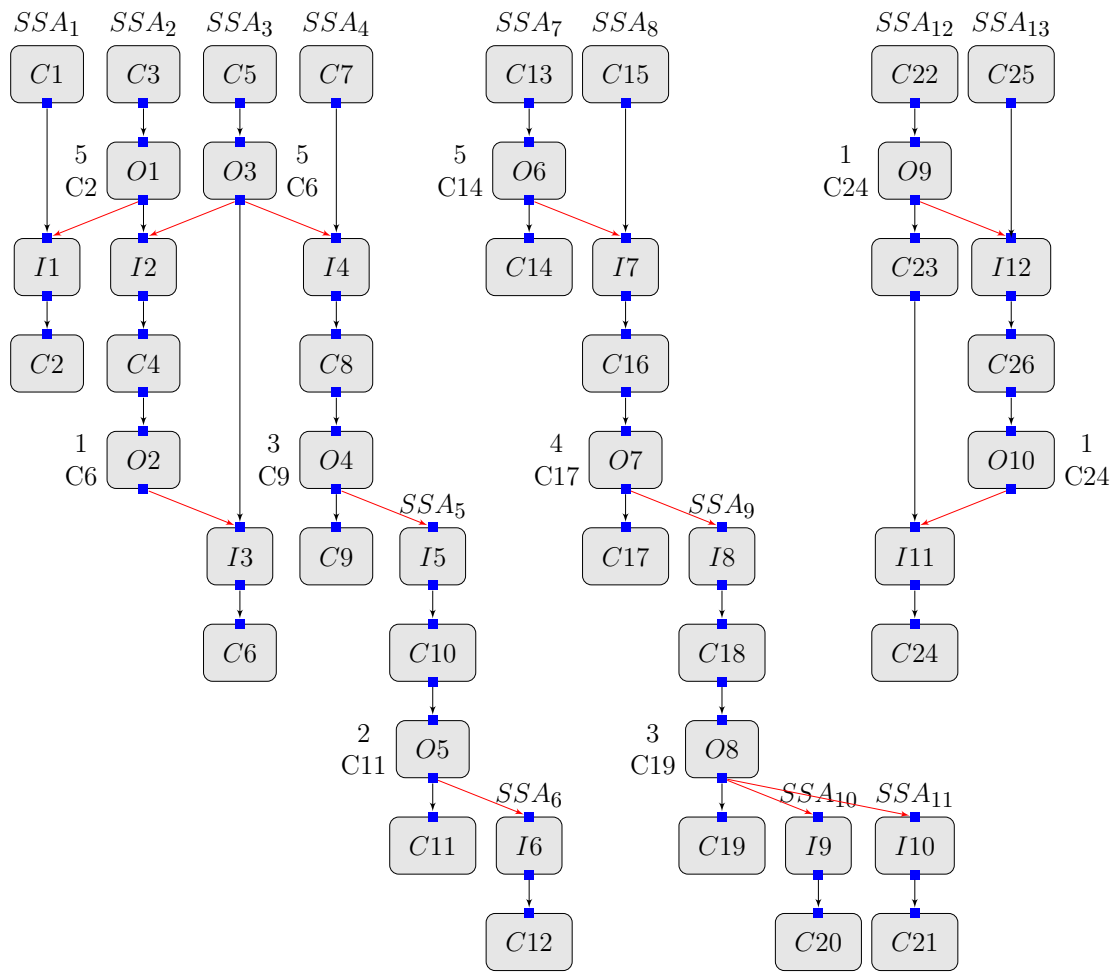


Figure 3.7: Graph Built Given the Simulated Scenario

```

C22 . O10 . C23 . C25 . I12 . C26 . O11 . I11 . C24 . C15 . C13 .
O7 . I7 . C16 . O8 . I8 . C18 . O9 . I10 . C21 . I9 . C20 . C19 .
C17 . C14 . C7 . C5 . O3 . I4 . C8 . O4 . I5 . C10 . O5 . I6 .
C12 . C11 . C9 . C3 . O1 . I2 . C4 . O2 . I3 . C6 . C1 . I1 . C2

```

(a) Before Persisting

```

C22 . O10 . C23 . C25 . I12 . C26 . O11 . I11 . C24 . C15 .
C13 . O7 . O7.persist() . I7 . C16 . O8 . I8 . C18 . O9 . I10 .
C21 . I9 . C20 . C19 . C17 . C14 . O7.unpersist() . C7 . C5 .
O3 . O3.persist() . I4 . C8 . O4 . I5 . C10 . O5 . I6 . C12 .
C11 . C9 . C3 . O1 . O1.persist() . I2 . C4 . O2 . I3 . C6 .
O3.unpersist() . C1 . I1 . C2 . O1.unpersist()

```

(b) After persisting the Three Top Used Output RDDs

Figure 3.8: The Final Sequence *SA* of Scenarion 3

We compare the performance of the resulting Spark application *SA* without persisting, when persisting the three top used outputs, when persisting the two top used output in the same scope, and when persisting all the outputs. Figure 3.9 shows the execution time of *SA* when applying each of the four scenarios while varying the dataset size. The obtained results confirm the previous deduction that the expensive persistence can deteriorate the performance of *SA* on big datasets and in complex applications. For instance, for 5GB dataset the persist all took 1018.90 sec whereas persisting none took 865 sec. The best performance in this scenario was the persist 3 top used strategy (801 sec on 5GB).

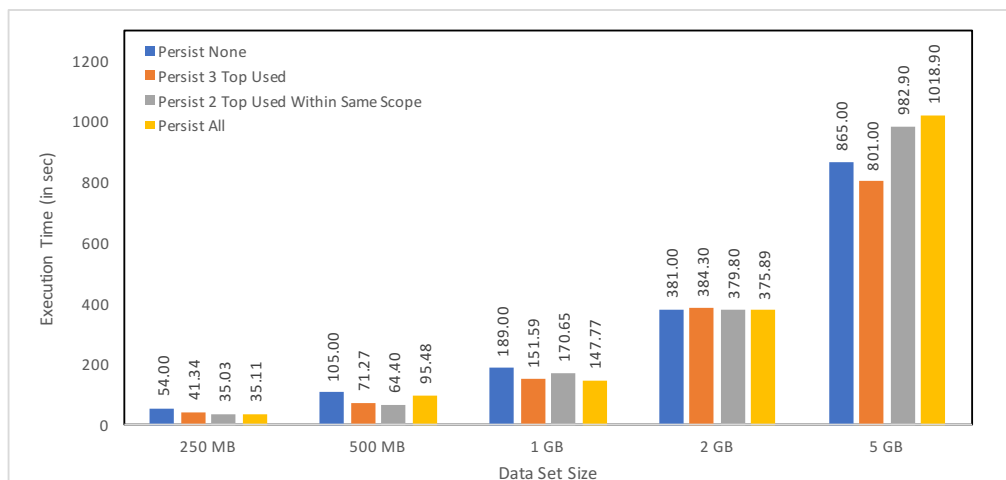


Figure 3.9: Comparison of Several Persisting Strategies for Scenario 3

## Chapter 4

# TRANSFORMATION BASED OPTIMIZER FOR SPARK

### *Contents*

4.1	Overview . . . . .	26
4.2	Modeling Spark Jobs . . . . .	27
4.3	Rewriting System . . . . .	28
4.3.1	Preliminaries . . . . .	28
4.3.2	Rewrite Rules . . . . .	29
4.3.2.1	Transformation Fusion . . . . .	30
4.3.2.2	Redundancy Elimination . . . . .	31
4.3.2.3	Transformations Reordering . . . . .	31
4.3.2.4	Transformation Action Reordering . . . . .	34
4.3.2.5	GroupBy-Aggregate . . . . .	34
4.4	Code Transformation . . . . .	35
4.4.1	Synthesis Phase . . . . .	36
4.4.2	Selection Phase . . . . .	36
4.4.2.1	Maximum Weight Strategy . . . . .	37
4.4.2.2	Minimum Cost Strategy . . . . .	37
4.5	Implementation . . . . .	39

### 4.1 Overview

Spark provides developers with an extensible easy to use API. Despite that, writing an efficient Spark program can be challenging, and requires a clear and deep understanding of the inner-workings of Spark. For this, we present TaBOS, a Spark job optimizer. For a given Spark job, TaBOS generates all the state space of alternative jobs by applying a set of rewrite rules. Then, it selects the most optimal state based on a predefined strategy. We present some strategies that can be defined for selecting an optimal state, along with a cost model for comparing two Spark jobs, i.e., states.

## 4.2 Modeling Spark Jobs

As a first step, we describe the grammar, depicted in Listing 4.1, used for modeling the structure of a Spark job. A Spark job ends with a unique **Action** preceded with a **Flow**. A **Flow** consists of branches of transformations each originating from a **DataSource**. The **DataSource** can be either a read operation (e.g., `textFile`) or a pre-existing RDD. Transformation are separated into two sets. Single transformation (**SingleTrans**) are operations that takes as input a single RDD and produces one output RDD (e.g., `map`). These are further sub-divided into four groups based on the parameters they take: (1) no parameters, (2) a single unary or binary user-defined function “UDF”, (3) an initial value and a UDF, or (4) an initial value and two UDFs. On the other hand, double transformations (**DoubleTrans**) are operations that receive two RDDs as an input and merge them into a new RDD (e.g., `join`). If a flow starts with a single transformation it is considered as a **SingleFlow**, otherwise, i.e., it starts with double transformation, it is considered as a **DoubleFlow**. The final operation in the Spark job is **Action** that takes an input RDD and produces a non-RDD result (e.g., `count`), which is the desired output of the job.

Within our grammar, we use “•” to denotes the composition of several flows, i.e. ‘‘A • B’’ denotes that the output of B is the input of A. We also use “;” to denote combining two terms, i.e., ‘‘A ; B’’ denotes that the outputs from both A and B are combined together. For instance, ‘‘A • (B ; C)’’ denotes that the outputs from B and C are both inputs for A.

Without loss of generality, we model a non-Pair RDD, which is a collection of objects, as a collection of tuples, since any object can be easily mapped to a tuple like structure. A tuple  $(v_1, v_2, \dots, v_n)$  have several variables each of which has a type (e.g., Integer, String, Array). On the other hand, a Pair RDD can be modeled, based on the context of use, either as the non-Pair RDD or as a collection of pairs consisting of a key and a tuple  $(k, (v_1, v_2, \dots, v_n))$ , where key  $k$  has a type as well.

Job	→ Action • Flow
Flow	→ SingleFlow   DoubleFlow   DataSource
SingleFlow	→ SingleTrans • Flow
DoubleFlow	→ DoubleTrans • ( Flow ; Flow )
SingleTrans	→ T1   T2( UDF )   T3( IDENTIFIER , UDF )   T4( IDENTIFIER , UDF, UDF )
DoubleTrans	→ union   join   zip   subtract   intersection
Action	→ collect   forEach   count   reduce( UDF )   take   first   max   min   sum
T1	→ groupByKey
T2	→ map   filter   flatMap   mapPartition   mapValues   flatMapValues   reduceByKey
T3	→ foldByKey
T4	→ aggregateByKey
UDF	→ ( IDENTIFIER ) => CODE   ( IDENTIFIER , IDENTIFIER ) => CODE
DataSource	→ textFile( STRING )   RDD

Listing 4.1: Spark Job Syntax

### 4.3 Rewriting System

TaBOS performs Spark job optimization using a rewrite system composed of a set of rewrite rules. Each one of these rules specifies a semantically valid modification to the structure of the Spark job aiming at performance gain.

#### 4.3.1 Preliminaries

We start this section by introducing some of the basic notations and concepts that are used by our rewrite system.

*Definition 6. (Rewrite Rule)* A Rewrite rule  $R$  denotes the transformation of a sequence of instructions matching pattern “ $A$ ” to the instantiation of “ $B$ ” under the context “ $C$ ”. It is expressed in the following form:

$$\frac{A}{B} \quad C$$

*Definition 7. (Redex)* A redex is subterm in job that matches a rewrite rule.

*Definition 8. (Normal Form)* A job is said to be in normal form if it has no redices.

We say  $job_1[r] = job_2$  to denote that  $job_1$  evaluates to  $job_2$  after applying the rewrite rules  $r$  to the first redex in  $job_1$ . If  $job_1$  is in normal form, then  $job_1[r] = \emptyset$ .

*Definition 9. (Identity Function)* A function  $f$  is said to be an identity function over the domain  $M$  if:  $\forall x \in M, f(x) = x$ .

*Definition 10. (Distributive Functions)* A function  $f_1$  is said to be distributive over another function  $f_2$  under the domain  $M$  if:  $\forall x, y \in M, f_2(f_1(x), f_1(y)) = f_1(f_2(x, y))$ .

We define the read set of function  $f$ , denoted as  $Read(f)$ , to be the indicies of all the attributes in the input tuple that can influence and change the output tuple of  $f$ . Formally,

*Definition 11. (Read Set)* For a function  $f : V \rightarrow U$ , we define  $Read(f) = \{i \in \mathbb{N}^+ \mid \exists x, y \in V, \forall i, j < |V| \wedge i \neq j : x_i \neq y_i \wedge x_j = y_j \Rightarrow f(x) \neq f(y)\}$

### **4.3.2 Rewrite Rules**

We introduce several rewrite rules where each of which performs a specific type of modification. Based on the modification’s type, we classify these rules into five distinct sets.



### 4.3.2.1 Transformation Fusion

The first set of rules, similar to those presented in [13] and inspired by the deforestation rules in [14], are concerned with transformations' composition. Deforestation (a.k.a. data structure fusion) is an optimization technique in functional languages, that aims to reduce the time consumed in constructing intermediate data structure, through the composition of several functions that produce and consume some temporary data structure. Since RDDs are considered as a data structure, applying deforestation on Spark's transformations can speed up the execution time of the job. For instance, consider the example in Listing 4.2, which computes  $\sum_{x \in \text{RDD}} (2 * x + 3)^2$ , using Spark job in 4.2b instead of that in 4.2a eliminates the time wasted on generating the intermediate RDD and requires a single pass over the records of RDD instead of two.

sum • map( $x \Rightarrow x^2$ ) • map( $x \Rightarrow 2 * x + 3$ ) • RDD
---

(a) Before Deforestation

sum • map( $x \Rightarrow (2 * x + 3)^2$ ) • RDD
--

(b) After Deforestation

Listing 4.2: Transformation Fusion Example

Below are the fusion rewrite rules defined in this set:

$$\frac{\text{map}(\text{udf}_2) \bullet \text{map}(\text{udf}_1)}{\text{map}(\text{udf}_2 \circ \text{udf}_1)} \quad (4.1)$$

$$\frac{\text{mapValues}(\text{udf}_2) \bullet \text{mapValues}(\text{udf}_1)}{\text{mapValues}(\text{udf}_2 \circ \text{udf}_1)} \quad (4.2)$$

$$\frac{\text{map}(\text{udf}_2) \bullet \text{mapValues}(\text{udf}_1)}{\text{map}(\text{udf}_2 \circ [(k, v) \Rightarrow (k, \text{udf}_1(v))])} \quad (4.3)$$

$$\frac{\text{mapValues}(\text{udf}_2) \bullet \text{map}(\text{udf}_1)}{\text{map}([(k, v) \Rightarrow (k, \text{udf}_2(v))] \circ \text{udf}_1)} \quad (4.4)$$

$$\frac{\text{flatMap}(\text{udf}_2) \bullet \text{map}(\text{udf}_1)}{\text{flatMap}(\text{udf}_2 \circ \text{udf}_1)} \quad (4.5)$$

$$\frac{\text{flatMap}(\text{udf}_2) \bullet \text{mapValues}(\text{udf}_1)}{\text{flatMap}(\text{udf}_2 \circ [(k, v) \Rightarrow (k, \text{udf}_1(v))])} \quad (4.6)$$

$$\frac{\text{flatMapValues}(\text{udf}_2) \bullet \text{mapValues}(\text{udf}_1)}{\text{flatMapValues}(\text{udf}_2 \circ \text{udf}_1)} \quad (4.7)$$

$$\frac{\text{filter}(\text{udf}_2) \bullet \text{filter}(\text{udf}_1)}{\text{filter}(\text{udf}_2 \ \&\& \ \text{udf}_1)} \quad (4.8)$$

where  $\circ$  denotes function composition,  $(f \circ g)(x) = f(g(x))$ .

#### 4.3.2.2 Redundancy Elimination

In this set, we define rules that eliminate redundant transformations. We consider a transformation to be redundant if removing it from the Spark job has no influence on the final result of the job. As an example of such transformations are those that take an identity function as an argument. This case can happen: (1) by mistake from the developer; (2) after applying one of the transformation fusion rules; or (3) after sequentially composing two flows. For example, applying rule 4.1 on the following flow: “ $\text{map}(x \Rightarrow x^2) \bullet \text{map}(x \Rightarrow \sqrt{x})$ ” will lead to “ $\text{map}(x \Rightarrow (\sqrt{x})^2)$ ”, which is equivalent to “ $\text{map}(x \Rightarrow x)$ ” over positive integers. Clearly, eliminating these transformations can drastically reduce the overall execution time of the Spark job. Formally,

$$\frac{\text{map}(\text{udf})}{\varepsilon} \quad \text{udf is an identity function} \quad (4.9)$$

#### 4.3.2.3 Transformations Reordering

As their name indicates, rules that belong to this set handles swapping transformations for better execution. The reordering between two transformations can

be established either based on their algebraic properties, or by reasoning about the read and write “conflicts” between them.

$$\frac{\text{reduceByKey}(udf_2) \bullet \text{map}(udf_1)}{\text{map}(udf_1) \bullet \text{reduceByKey}(udf_2)} \quad udf_1 \text{ is distributive over } udf_2 \quad (4.10)$$

In the first rule, if a `map` is followed with a `reduceByKey` transformation and its function distributes over the `reduceByKey`’s function, then the `map` can be safely reordered with the `reduceByKey`. For datasets with plenty of records per key, the following rule can lead to a huge performance gain. Listing 4.3 shows an example which computes for each key in RDD “ $\sum_{v \in \text{RDD}(k)} k * v$ ”, which is equivalent by the distributive property to “ $k * \sum_{v \in \text{RDD}(k)} v$ ”, where `RDD(k)` represent all the values in RDD that are associated with the key  $k$ . Now consider the case where we have an RDD with  $10^{32}$  record and only 10 keys, the job in figure 4.3a will have higher execution time than that in 4.3b, as it will perform “ $10^{31} - 10$ ” more multiplication operation.

<code>collect</code> • <code>reduceByKey((a, b) =&gt; a + b)</code> • <code>map((k, v) =&gt; (k, k * v))</code> • RDD
---

(a) Before Reordering

<code>collect</code> • <code>map((k, v) =&gt; (k, k * v))</code> • <code>reduceByKey((a, b) =&gt; a + b)</code> • RDD
---

(b) After Reordering

Listing 4.3: Example of Reordering `map` and `reduceByKey`

In the rest of the reordering rules, we consider pushing the `filter` transformation as close as possible to the data source, as early `filter` execution reduces the quantity of records processed in downstream transformations which can help reduce the overall execution time. In rule 4.11, a `filter` transformation is reordered with any narrow transformation if the read set of  $udf_2$  used by `filter` does not conflict with the changes made by  $udf_1$ , i.e. there is no read/write conflict between  $udf_1$  and  $udf_2$ . In the example of Listing 4.4, the `map` and `filter` transformations can be safely reordered as `Read((v1, v2) => v1 != 0) = {1}` and the `map`’s function does not change the value of  $v_1$ . Whereas, in the second example of Listing 4.5, it is not possible to have any reordering since the value of  $v_1$  was changed by the

map's function. In the other rules, the `filter` transformation is reordered with wide transformations. This can lead to a higher speedup in the total execution time of the Spark job, since applying the `filter` before the wide transformation decreases the number of records shuffled through the network. In rule 4.14, if the read set of  $udf_1$  used by the `filter` does not intersect with the attributes of the records produced by  $flow'$ , then the `filter` can be safely pushed to be composed with  $flow$ . The opposite is true in case of rule 4.13. On the other hand, if the `filter` operation is based only on the key value, then it should be executed on both of the `join`'s input flow before joining them. As for `union` and `intersection`, any `filter` transformation following them should be immediately pushed before them.

`collect • filter((v1, v2) => v1 ≠ 0) • map((v1, v2, v3) => (v1, v1 * v2)) • RDD`

Listing 4.4: Example where `map` and `filter` can be Reordered

`collect • filter((v1, v2) => v1 ≠ 0) • map((v1, v2, v3) => (v1 * v3, v2, v3)) • RDD`

Listing 4.5: Example where `map` and `filter` can not be Reordered

$$\frac{filter(udf_2) \bullet T(udf_1)}{T(udf_1) \bullet filter(udf_2)} \quad udf_1[(v_1, \dots, v_n) \Rightarrow (u_1, \dots, u_m)], \quad (4.11)$$

$$\forall i \in Read(udf_2) : i \leq m \wedge v_i = u_i$$

$$\frac{filter(udf_1) \bullet join(flow; flow')}{join(filter(udf_2) \bullet flow; flow')} \quad \begin{aligned} & udf_1[(k, v, v') \Rightarrow (k, v, v')], \\ & v' \notin Read(udf_1), \\ & udf_2[(k, v) \Rightarrow (k, v)] \end{aligned} \quad (4.12)$$

$$\frac{filter(udf_1) \bullet join(flow; flow')}{join(flow; filter(udf_2) \bullet flow')} \quad \begin{aligned} & udf_1[(k, v, v') \Rightarrow (k, v, v')], \\ & v \notin Read(udf_1), \\ & udf_2[(k, v') \Rightarrow (k, v')] \end{aligned} \quad (4.13)$$

$$\frac{filter(udf_1) \bullet join(flow; flow')}{join((filter(udf_3) \bullet flow); (filter(udf_2) \bullet flow'))} \quad \begin{array}{l} udf_1[(k, v, v') \Rightarrow (k, v, v')], \\ v, v' \notin Read(udf_1), \\ udf_2[(k, v') \Rightarrow (k, v')], \\ udf_3[(k, v) \Rightarrow (k, v)] \end{array} \quad (4.14)$$

$$\frac{filter(udf) \bullet T(flow_1; flow_2)}{T((filter(udf) \bullet flow_1); (filter(udf) \bullet flow_2))} \quad T \in \{union, intersection\} \quad (4.15)$$

#### 4.3.2.4 Transformation Action Reordering

The transformation action reorder rule is similar to rule 4.10, except that instead of reordering the `map` with the `reduce`, we move the function used by the `map` to be computed after the `reduce` operation. This rule guarantees to have a speedup on any dataset, since only one record is returned by the `reduce` action at the end.

$$\frac{reduce(udf_2) \bullet map(udf_1)}{udf_1 \bullet reduce(udf_2)} \quad udf_1 \text{ is distributive over } udf_2 \quad (4.16)$$

#### 4.3.2.5 GroupBy-Aggregate

`groupByKey` is one of the most expensive transformations in Spark. It can cause all dataset records to be shuffled around the network. Moreover, it combines records with the same key in memory as an iterator, which can't be distributed causing memory errors at scale. For this, we propose a rewrite rule that substitutes a flow composed of `groupByKey` followed with a `map/mapValues` by an alternative sequence of transformations that applies the same computation performed by the `map/mapValues` transformation. For simplicity, we will only handle computations that uses some of the predefined operations of the `Iterator` class in Scala<sup>1</sup>. Formally,

---

1. This can be developed in future work by using tools that translate imperative code to MapReduce form.

$$\frac{\text{map}([(k, v) \Rightarrow (k, \text{udf}(v))]) \bullet \text{groupByKey}())}{\text{Alt}(op_n) \bullet \dots \bullet \text{Alt}(op_2) \bullet \text{Alt}(op_1)} \quad \begin{array}{l} \text{udf} = op_n \circ \dots \circ op_2 \circ op_1 \\ op_i \in \mathcal{S} \quad \forall i \ 1 \leq i \leq n \end{array} \quad (4.17)$$

$$\frac{\text{mapValues}([v \Rightarrow \text{udf}(v)]) \bullet \text{groupByKey}())}{\text{Alt}(op_n) \bullet \dots \bullet \text{Alt}(op_2) \bullet \text{Alt}(op_1)} \quad \begin{array}{l} \text{udf} = op_n \circ \dots \circ op_2 \circ op_1 \\ op_i \in \mathcal{S} \quad \forall i \ 1 \leq i \leq n \end{array} \quad (4.18)$$

Where  $\mathcal{S}$  represent the set of operations of the Iterator class in Scala, and “ $\text{Alt}(op)$ ” represent alternative sequence of transformation for the operation  $op$ . Table 4.1 represent the alternative sequence  $\text{Alt}(op)$  for each operation  $op \in \mathcal{S}$ .

Iterator Operation	Alternative Sequence of Transformation
<code>map(f)</code>	<code>map((k, v) =&gt; (k, f(v)))</code>
<code>filter(p)</code>	<code>filter((k, v) =&gt; p(v))</code>
<code>flatMap(f)</code>	<code>flatMap((k, v) =&gt; (k, f(v)))</code>
<code>reduce(f)</code>	<code>reduceByKey(f)</code>
<code>aggregate(v, f, f')</code>	<code>aggregateByKey(v, f, f')</code>
<code>sum</code>	<code>reduceByKey((a, b) =&gt; a + b)</code>
<code>product</code>	<code>reduceByKey((a, b) =&gt; a * b)</code>
<code>size</code>	<code>reduceByKey((a, b) =&gt; a + b) \bullet map((k, v) =&gt; (k, 1))</code>
<code>max</code>	<code>reduceByKey((a, b) =&gt; Max(a, b))</code>
<code>min</code>	<code>reduceByKey((a, b) =&gt; Min(a, b))</code>
<code>maxBy(f)</code>	<code>reduceByKey((a, b) =&gt; Max(a, b)) \bullet map((k, v) =&gt; (k, f(v)))</code>
<code>minBy(f)</code>	<code>reduceByKey((a, b) =&gt; Min(a, b)) \bullet map((k, v) =&gt; (k, f(v)))</code>
<code>forall(p)</code>	<code>reduceByKey((a, b) =&gt; a \&amp;\&amp; b) \bullet map((k, v) =&gt; (k, p(v)))</code>

Table 4.1: Alternative Transformation Sequences for Operations of the Iterator Class in Scala

#### 4.4 Code Transformation

In this section, we present the algorithm used by TaBOS to generate for a given Spark job “*initJob*” a semantically equivalent optimized job using the defined set of rewrite rules “ $\mathcal{R}$ ”. The algorithm is composed of two cooperating phases: synthesis and selection phase. The synthesis phase recursively applies the rewrite rules to generates all the state space of alternative jobs. Then, the selection phase scavenges

the generated states to select the most optimal job based on a predefined strategy.

We represent the search space using a directed acyclic graph (DAG)  $G = (V, E)$ , with vertices  $V$  representing the alternative jobs obtained from applying one or more rewrite rules to  $initJob$ , and edges representing one step of transformation between two vertices.

#### 4.4.1 *Synthesis Phase*

The synthesis phase, presented in Figure 4.1 as a set of rules, populates the search space  $G$  with alternative jobs of  $initJob$  produced by applying one or more of the rewrite rules. The synthesis process is performed in a top-down fashion, it starts with the “INIT” rule that initializes  $G$  with  $initJob$ , then it exhaustively applies the “UPDATE” rule to generate all possible alternatives until it can longer be applied. Rule UPDATE traverses the representation structure of  $job \in V$  to find the first redex where a rule  $r \in \mathcal{R}$  can be applied. It applies the rule, and update  $G$  to include this new job. The algorithm terminates when all the reached vertices are in normal form. The final result of the synthesis phase is the least acyclic graph that can be generated from the synthesis rules (INIT and UPDATE) and originating from  $initJob$ .

$\frac{}{V = \{initJob\} \wedge E = \emptyset} \quad \text{INIT}$ $\frac{job \in V \wedge \exists r \in \mathcal{R} \wedge job[r] \neq \emptyset}{V = V \cup \{job[r]\} \wedge E = E \cup \{job \xrightarrow{r} job[r]\}} \quad \text{UPDATE}$
---

Figure 4.1: Synthesis Rules

#### 4.4.2 *Selection Phase*

Once the search space  $G$  is generated, the selection phase explores it to select one optimal alternative job. This exploration requires a mechanism for comparing solution based on some kind of strategy, either by comparing the entire paths of  $G$  or by comparing the nodes. We distinguish between two selection strategies:

#### 4.4.2.1 Maximum Weight Strategy

In this strategy, we select the job that has the maximum sum of weights for consecutive rewrite rules <sup>2</sup> applied to *initJob*. This can be obtained by finding the path with maximum weight that originates from the source and ends at any sink in  $G$ . Formally,

$$\operatorname{argmax}_{job} \{ d^w(\text{initJob}, job) \}$$

where  $d^w(v_1, v_2)$  denotes the weight of the path from vertex  $v_1$  to  $v_2$  in  $G$ .

#### 4.4.2.2 Minimum Cost Strategy

This strategy is based on cost estimation of each generated job, and the job with the minimum cost is selected. Formally,

$$\operatorname{argmin}_{job} \{ cost(job) \}$$

where  $cost(job)$  denotes the computed cost of *job*.

**Cost Model:** In order to compare Spark jobs, we propose a cost model that estimates the cost of a given job. It is a linear model that sums the unit-cost of all the operators (transformation or action) of the Spark *job*.

$$cost(job) = \sum_{i=1}^n cost(op_i)$$

The unit cost of a give operator  $op_i$  is the product of three main cost factors:

$$cost(op_i) = computation_i * ratioIO_i * communication_i$$

with

- $computation_i$ : representing the cost of the computation performed by the operator. It can be determined by the complexity of the UDF used, if any.

---

2. Weights can assigned to the defined rewrite rules in a greedy manner or based on the benchmarks described in the evaluation chapter.



- $ratioIO_i$ : this cost depends on how the operator influence the number of records ( $\#records$ ) in the input. It is estimated based on the following equation:

$$ratioIO_i = \begin{cases} \mathit{distance}(op_i) & \text{if } \#records \text{ increases} \\ \frac{1}{\mathit{distance}(op_i)} & \text{if } \#records \text{ decreases} \\ 1 & \text{otherwise} \end{cases}$$

where  $\mathit{distance}(op_i)$  represent the distance from  $op_i$  to the unique action of the Spark *job*. We base our equation on the observation that it is preferable to have early evaluation of operations that reduce the  $\#records$  of an RDD (e.g., `filter`). Therefore, such operations should be as close as possible to the data-source and as far as possible from the action operation. The opposite is true for operation that increase the  $\#records$  of an RDD (e.g., `flatMap`).

- $communication_i$ : depends on the cost of communicating either with the file system for read/write (e.g., `textFile` and `saveAsTextFile`), or with the network for shuffling the data (e.g., `reduceByKey` and `collect`). In case of a narrow transformation, this cost is set to 1 as no communication is required.

Furthermore, we have decided to multiply the cost factors instead of summing them to better express the changes made by the defined rewrite rules, mainly the fusion and reorder rules. Consider the example presented in Listing 4.6 for two semantically equivalent Spark jobs. The corresponding cost for both jobs is:

$$cost(j_1) = cost(\mathit{count}) + cost(\mathit{filter}(f_3)) + cost(\mathit{map}(f_2)) + cost(\mathit{filter}(f_1))$$

$$cost(j_2) = cost(\mathit{count}) + cost(\mathit{filter}(f_3 \ \&\& \ f_1)) + cost(\mathit{map}(f_2))$$

`sum • filter(f3) • map(f2) • filter(f1) • RDD`

(a) Spark Job  $j_1$

`sum • filter(f3 && f1) • map(f2) • RDD`

(b) Spark Job  $j_2$

Listing 4.6: Example of Two Semantically Equivalent Jobs

Let  $comp(f)$  denote the complexity of function  $f$ . In our example:

$$comp(f_3 \&\& f_1) = comp(f_3) + comp(f_1)$$

According to our model, where we multiply the factors to compute the unit-cost:

$$\begin{aligned} cost(j_1) - cost(j_2) &= cost(\mathbf{filter}(f_3)) + cost(\mathbf{filter}(f_1)) - cost(\mathbf{filter}(f_3 \&\& f_1)) \\ &= comp(f_3) * \frac{1}{1} * 1 + comp(f_1) * \frac{1}{3} * 1 - (comp(f_3 \&\& f_1) * \frac{1}{1} * 1) \\ &= comp(f_3) + \frac{1}{3} * comp(f_1) - (comp(f_3) + comp(f_1)) \\ &= -\frac{2}{3} * comp(f_1) \\ &< 0 \end{aligned}$$

Therefore,  $cost(j_1) < cost(j_2)$  which means that job  $j_1$  is better than job  $j_2$ , which is true.

Whereas, summing the factors instead results with:

$$\begin{aligned} cost(j_1) - cost(j_2) &= cost(\mathbf{filter}(f_3)) + cost(\mathbf{filter}(f_1)) - cost(\mathbf{filter}(f_3 \&\& f_1)) \\ &= comp(f_3) + \frac{1}{1} + 1 + comp(f_1) + \frac{1}{3} + 1 - (comp(f_3 \&\& f_1) + \frac{1}{1} + 1) \\ &= \frac{4}{3} \\ &> 0 \end{aligned}$$

Then,  $cost(j_1) > cost(j_2)$  which means that job  $j_2$  is better than job  $j_1$ , and this is false in reality.

## 4.5 Implementation

TaBOS implements all the ideas discussed in the previous sections. Figure 4.2 presents the overall architecture of TaBOS, it is basically composed of six major components:

- *Parser*: responsible for parsing the input file and converting the Spark job to an intermediate representation (IR) which is based on the model presented in section 4.2.
- *Configuration*: contains two main sub-components:
  - *Property Checker*: used for checking the conflicts and the algebraic properties included in the premises of the rewrite rules (e.g., check if read-

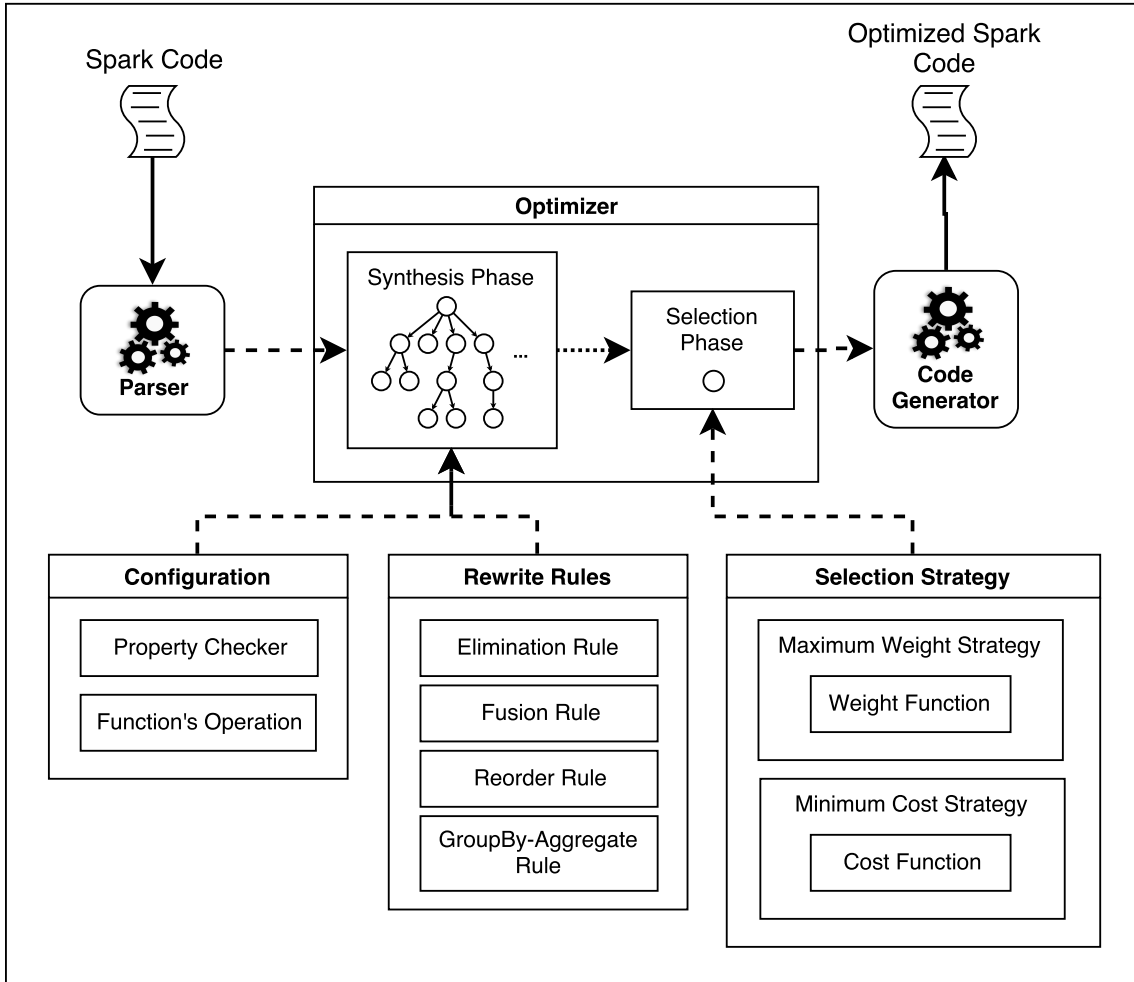


Figure 4.2: High Level Architecture of TaBOS

/write conflict exists between functions, check if a function is an identity function).

- *Function's Operation*: used for applying some of the functions' manipulation operations (e.g., composing functions, changing function's domain).

our system is not restricted for specific implementation of these two sub-components, as several solutions can exist.

- *Rewrite Rules*: includes the implementation of the rewrite rules defined in section 4.3.2. Each rule class contains two main functions: one the matches the rewrite rule with a given sub-term of the Spark job, and one that constructs the new sub-term to replace the old one. New rules can be easily integrated with TaBOS by simply extending the “Rule” interface.
- *Selection Strategy*: consist of the strategies used by the selection phase in

order to determine the most optimal alternative generated job. Our system currently support the two strategies discussed in section 4.4.2:

- *Maximum Weight Strategy*: takes a *Weight Function* that assign weights to the rewrite rules, and finds the path with the maximum weight in the generated search space using Dijkstra’s algorithm.
  - *Minimum Cost Strategy*: takes a *Cost Function* that compute the cost of a given Spark job, and compares all the generated alternative jobs to find the one with the minimum computed cost. There are currently two implementation for the *Cost Function*: one based on the length of operations used in the Spark job, and the other is based on the cost model discussed in section 4.4.2.2.
- *Optimizer*: it is the main component of TaBOS, it implements the algorithm presented in section 4.4, and consist of two phases:
    - *Synthesis Phase*: takes the intermediate representation of the Spark job generated by the *Parser*, along with a selected *Configuration* and a set of *Rewrite Rules*, then it applies the proposed synthesis algorithm to generate an acyclic graph of alternative Spark jobs.
    - *Selection Phase*: takes the generated graph from the *Synthesis Phase* and selects one optimal job based on a given *Selection Strategy* object.
  - *Code Generator*: translates the final selected Spark job to code that can be executed in Apache Spark engine.

TaBOS design is based on decoupling the *Configuration*, *Rewrite Rules*, and *Selection Strategy* modules from the internal core *Optimizer* in order to make it easy for user’s and third part libraries to extend it. Furthermore, within each module several extension points as offered to enable users to integrate their own implementations of: *Property Checker*, *Function’s Operation*, *Rewrite Rules*, *Selection Strategy*, *Weight Function*, and *Cost Function*.

# Chapter 5

## EVALUATION OF TaBOS

### *Contents*

5.1	Experimental Design . . . . .	42
5.2	Evaluation of Rewrite Rules . . . . .	43
5.3	Case Study . . . . .	51
5.3.1	State-Space Size . . . . .	52
5.3.2	Comparison of Selection Strategies . . . . .	53

### 5.1 Experimental Design

To evaluate the performance of our approach for Spark job optimization, we carry two sets of experiments. In the first, we test the performance gain for each individual rewrite rule. And in the second, we evaluate the synthesis algorithm and selections strategies supported by our tool on non-trivial case study. To perform our experiments we use several scenarios, each conducted under two environmental setups:

- **Local environment:** that uses a Linux machine (64-bit Ubuntu 16.04 LTS) with 32 core, Intel Xeon (R) processor, and 32 GB memory, running Spark 2.2.0. In this environment the applications are submitted locally on 5 cores using spark-submit.
- **Cluster environment:** that uses a small cluster of four Linux machines (64-bit Ubuntu 16.04 LTS), each with 8 cores, Intel Core i7-6700 processor, and 32 GB memory. The machines are connected with a 500 MBit Ethernet. Furthermore, we run Spark 2.2.0 using Yarn [15] cluster mode on top of Hadoop Distributed File System (HDFS) version 2.8.1 with 128 MB block setting. In this environment the application are submitted using spark-submit with client deploy mode, 30 GB driver memory, and 3 executors each with 5 cores and 25 GB executor memory.

For each experiment we report the average of 10 runs after dropping the highest and lowest time values.

Four datasets are used in the performance evaluation. Two synthesized datasets which we generate, one composed of doubles and the other composed of pairs of integers. The “GeoNames” database [12] available on Kaggle, covering all countries with over eleven million place names. And the “311 service requests” dataset [16] available by the NYC Open Data Project, containing calls monitored to the non-emergency service centers from 2010 to 2018.

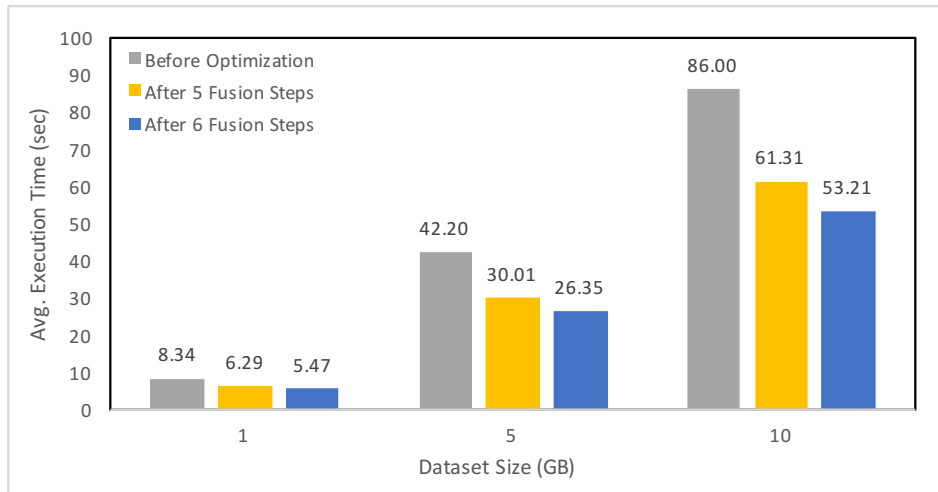
## 5.2 Evaluation of Rewrite Rules

This section assesses the independent optimization potential of our rewrite rules using six different scenarios. For each scenario, we report the average execution time of the Spark job before and after applying the rewrite rule. In order to have a better view of the actual performance gain of the rewrite rules, we report the execution time without the preprocessing time. The preprocessing time is the time consumed for computing the input RDD used by the redex that matches the rewrite rule.

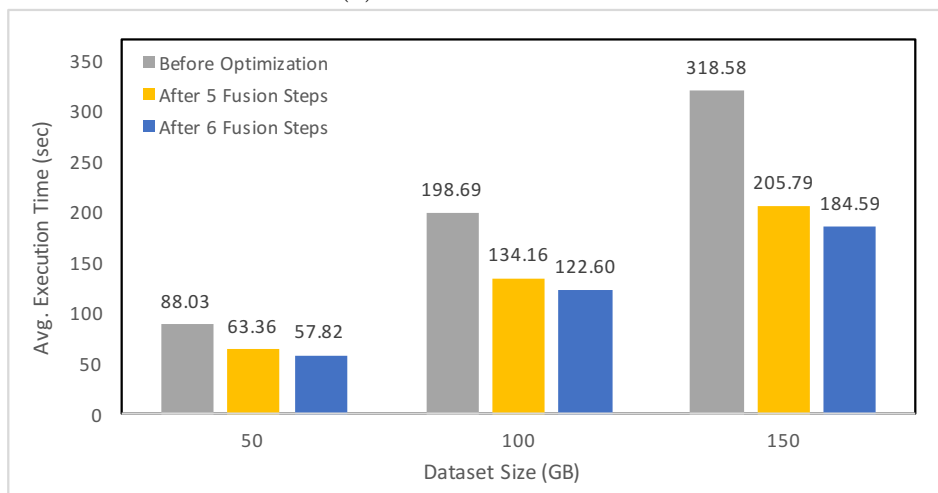
**Scenario 1.** In our first scenario, we test Rule 4.1 for `map` transformations’ fusion. The scenario consist of a Spark job that invokes seven `map` transformations on an “RDD” of double to compute the following result:

$$\sum_{x \in \text{RDD}} \frac{1}{1 - e^{x * 1.5046 - 4.0777}}$$

We apply rule 4.1 five times to reduce the given Spark job to two `map` transformations, then we apply an extra fusion step to further reduce it to one `map` transformation. Figure 5.1 presents the average execution time for the originally given Spark job compared to the reduced ones on the synthesized dataset of doubles with sizes ranging from 1 GB to 150 GB. For all sizes, applying the rewrite rule showed an improvement in the execution time, with speedup that increased from 1.3 to 1.5 as the dataset size increased. Furthermore, investigating the effect of a single fusion step showed a speedup  $\simeq 1.1$ .



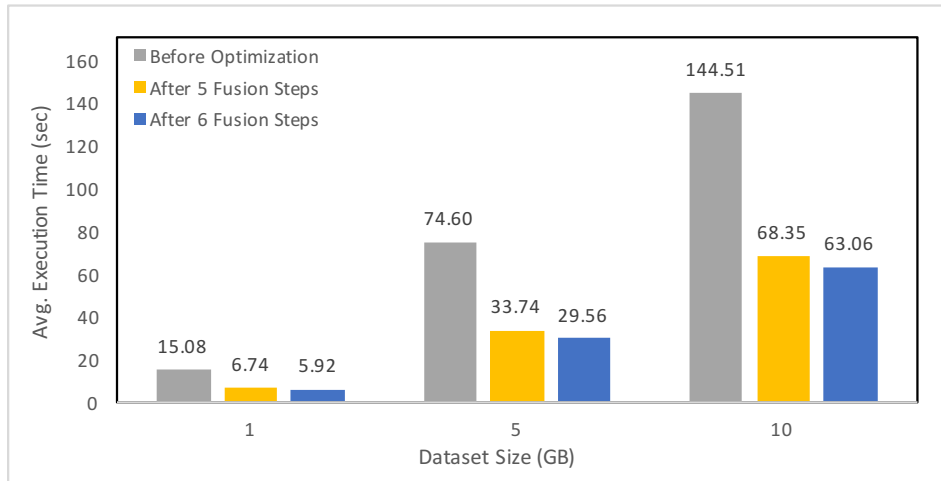
(a) Local Benchmarks



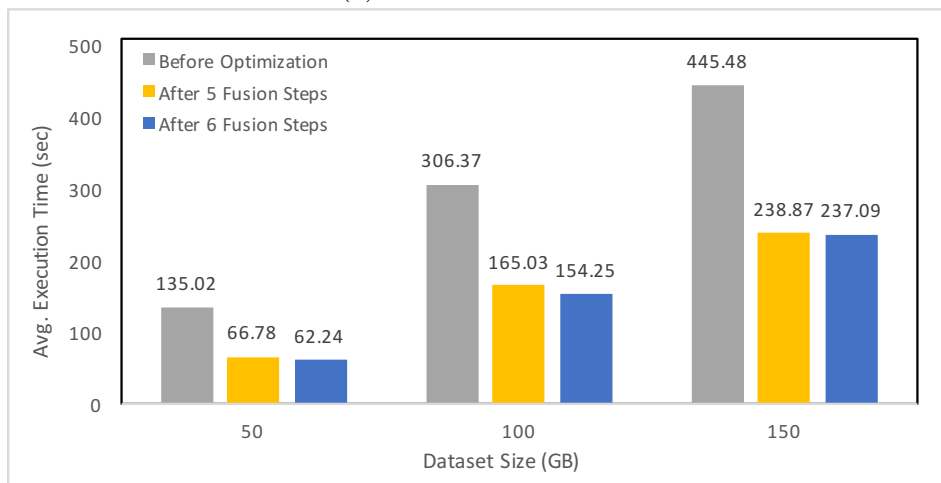
(b) Cluster Benchmarks

Figure 5.1: Average Execution Time of Scenario 1 Under Various Input Data Sizes

**Scenario 2.** The second scenario evaluates the performance of Rule 4.2 for `mapValues` fusion. We use an example similar to that used in scenario 1, expect that we replace the `map` transformation with `mapValues` transformation and we use a pair “RDD” of doubles  $(x, x^2)$ . The average execution time before and after applying the five and six steps of fusion are presented in figure 5.2. The results shows a speedup of two orders-of-magnitude for all dataset sizes, which is a higher speedup than that in scenario 1. This is expected as `mapValues` fusion reduces more time wasted on constructing intermediate data structures than `map` fusion. This is because the `mapValues` transformation has a higher complexity in constructing the resulting RDD than `map` transformation, as the results produced by the `mapValues`’s function have to be joined back with their corresponding keys.



(a) Local Benchmarks

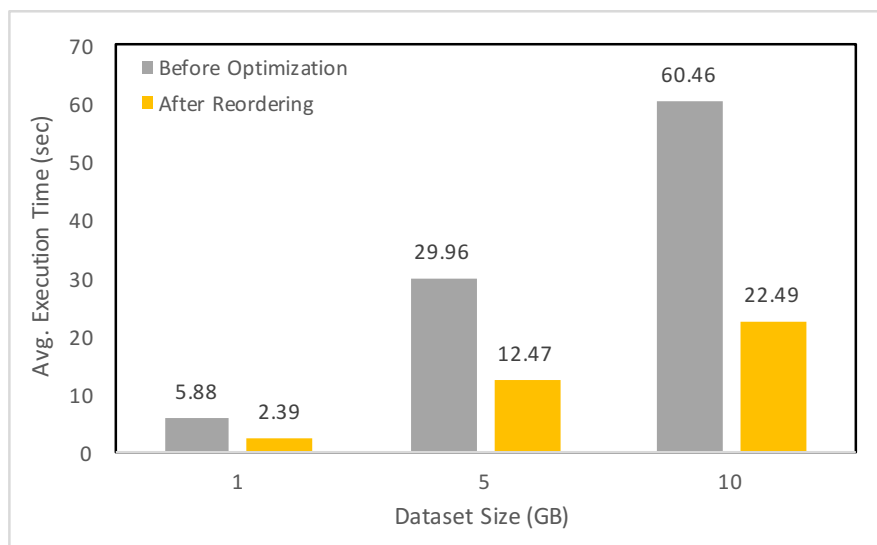


(b) Cluster Benchmarks

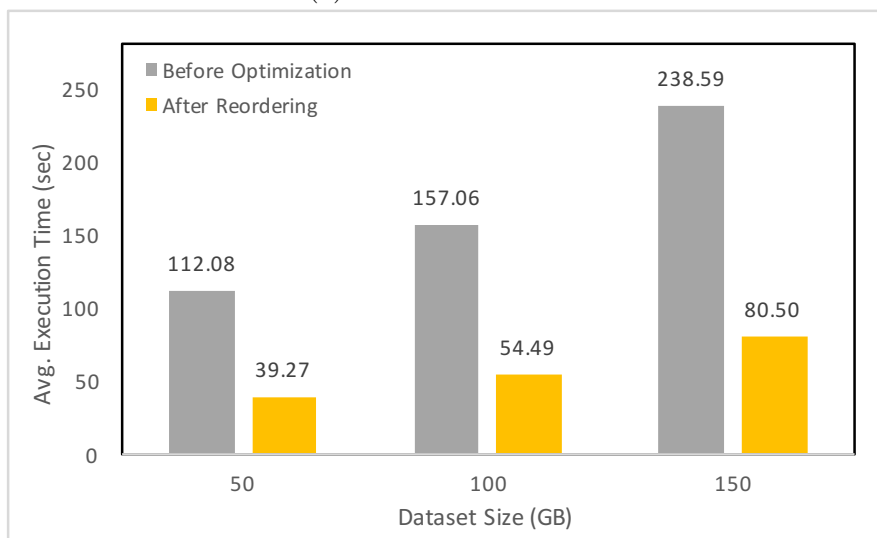
Figure 5.2: Average Execution Time of Scenario 2 Under Various Input Data Sizes



**Scenario 3.** Here we evaluate the effect of reordering `reduceByKey` with `map` transformation based on the distributive property of the transformation’s functions. Listing 5.1 shows the example used by this scenario. Since  $lcm(a, gcd(b, c)) = gcd(lcm(a, b), lcm(a, c))$  the `map` transformation in this example can be safely pushed past the `reduceByKey`. We measure the execution time before and after the reordering on a synthesized dataset of integer pairs with sizes ranging from 1 GB to 150 GB. The results are shown in figure 5.3. Based on the results, the reordering had a speedup increasing from 2.5 on 1 GB to 3 on 150 GB.



(a) Local Benchmarks



(b) Cluster Benchmarks

Figure 5.3: Average Execution Time of Scenario 3 Under Various Input Data Sizes

```
collect • reduceByKey((a, b) => gcd(a, b)) • map((k, v) => (k, lcm(k, v))) • RDD
```

(a) Before Reordering

```
collect • map((k, v) => (k, lcm(k, v))) • reduceByKey((a, b) => gcd(a, b)) • RDD
```

(b) After Reordering

Listing 5.1: Scenario 3

**Scenario 4.** In this scenario, we handle reordering `filter` and `map` transformations in the absence of read/write conflict between them. The example used in this scenario is depicted in Listing 5.2. In this example, the `filter` and `map` transformation can be safely reordered as the read set of the `filter`'s function ( $Read((k, v) => k \% 2 = 0) = \{1\}$ ) does not conflict with the values changed by the `map`'s function. We compare the average execution time before and after the reordering on a synthesized dataset of integer pairs. Figure 5.4 presents the corresponding results, the maximum speedup reached by this scenario was  $\simeq 1.5$  on the local benchmarks and  $\simeq 2.5$  on the cluster benchmarks.

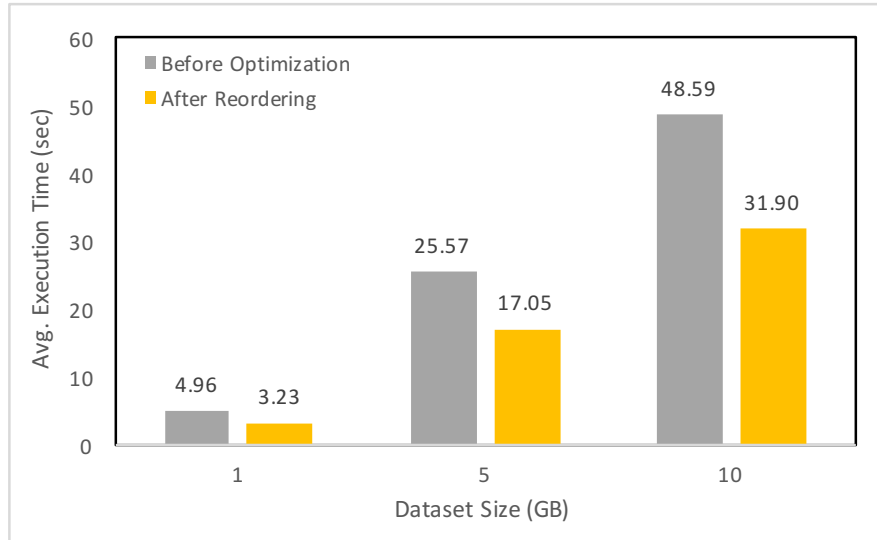
```
collect • filter((k, v) => k \% 2 = 0) • map((k, v) => (k, lcm(k, v))) • RDD
```

(a) Before Reordering

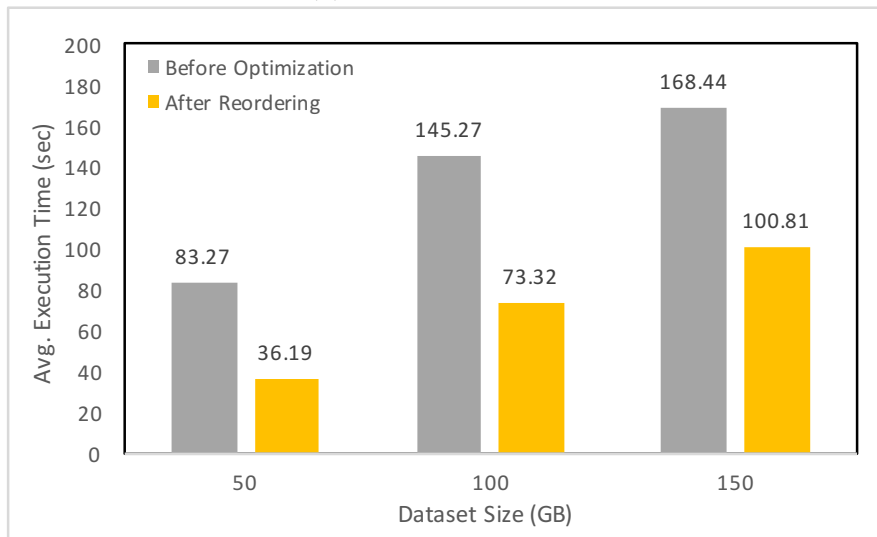
```
collect • map((k, v) => (k, lcm(k, v))) • filter((k, v) => k \% 2 = 0) • RDD
```

(b) After Reordering

Listing 5.2: Scenario 4



(a) Local Benchmarks

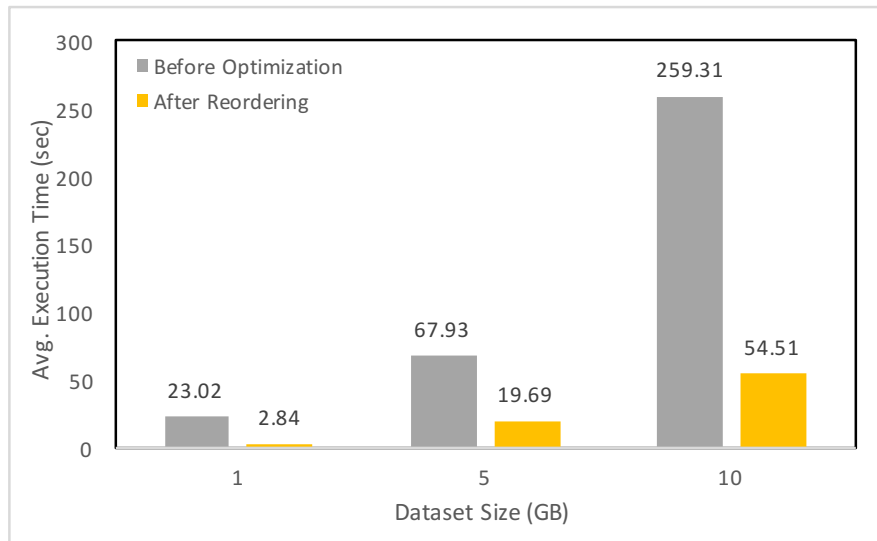


(b) Cluster Benchmarks

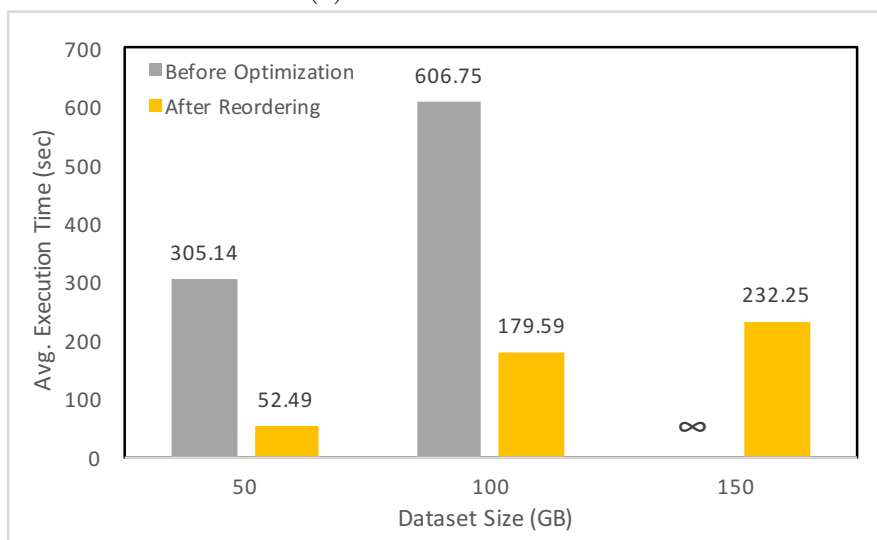
Figure 5.4: Average Execution Time of Scenario 4 Under Various Input Data Sizes

**Scenario 5.** This scenario tests Rules 4.12 and 4.13 for reordering the `filter` transformation to precede the `join`. For this evaluations, we use the GeoNames dataset with the example presented in Listing 5.3. Given two rdds: RDD1 consisting of the place id along with the country code, and RDD2 consisting of the place id with the distance of the place from a certain origin. The given example joins the two rdds, then filters the records using  $f_1$  to keep those with specific country code and  $f_2$  to keep those with distance below a certain threshold. According to the rules' premises, both `filter` transformations can be safely applied prior to `join`. Figure 5.5 shows the average execution time before and after applying the rewrite

rules. In the local benchmarks, the optimized job was 3-8 orders of magnitude faster. On the other hand, in the cluster benchmark, the optimized job was not only faster than the original job but also more efficient as the original job failed to execute in all the runs on 150GB, whereas the optimized version succeeds to finish with acceptable time. Although this rule guarantees to have a speedup, as it reduces the number of shuffled records during the `join` operation, there's is no correlation between the speedup and the dataset size, this is mainly because the speedup depends on the filtered ratio of records which is not a constant ratio and may vary with the dataset size.



(a) Local Benchmarks



(b) Cluster Benchmarks

Figure 5.5: Average Execution Time of Scenario 5 Under Various Input Data Sizes

```
count • filter((k, v1, v2) => f2(v2)) • filter((k, v1, v2) => f1(v1)) • join() • (RDD1 ; RDD2)
```

(a) Before Reordering

```
count • join() • ( filter((k, v1, v2) => f1(v1)) • RDD1 ; filter((k, v1, v2) => f2(v2)) • RDD2)
```

(b) After Reordering

Listing 5.3: Scenario 5

**Scenario 6.** The final scenario evaluates the substitution of `groupByKey` with a more efficient sequence of transformations. We use the example presented in Listing 5.4 with the GeoNames dataset to compute for a given RDD, made of a country code and the coordinates of a place that belongs to that country, the sum of the distances of all the places that belongs to the same country. We compare the average execution time of the example before and after the substitution. The results in figure 5.6 shows tremendous improvement in the execution time, the optimized job was able to outperforms the naive job with over 60 times faster.

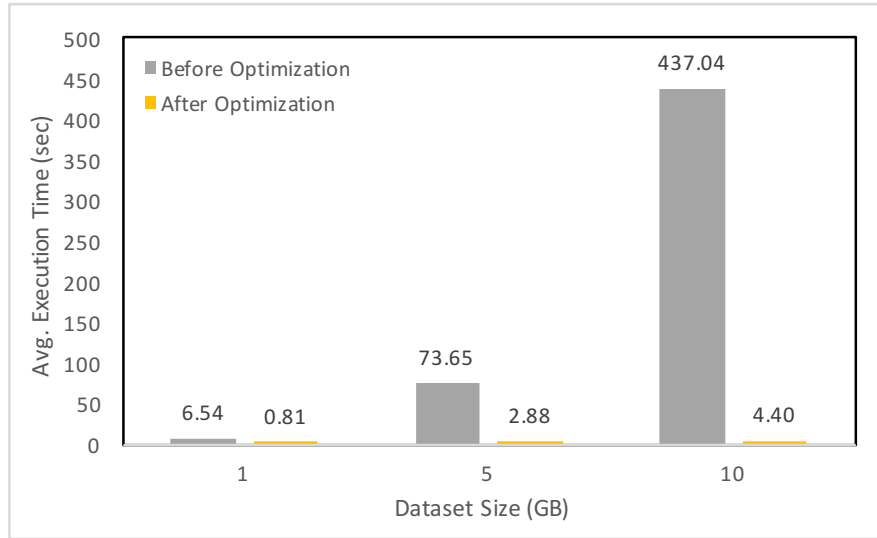
```
collect • mapValues(x => x.map((v1, v2) =>  $\sqrt{v_1^2 + v_2^2}$ ).sum) • groupByKey() • RDD
```

(a) Before Optimization

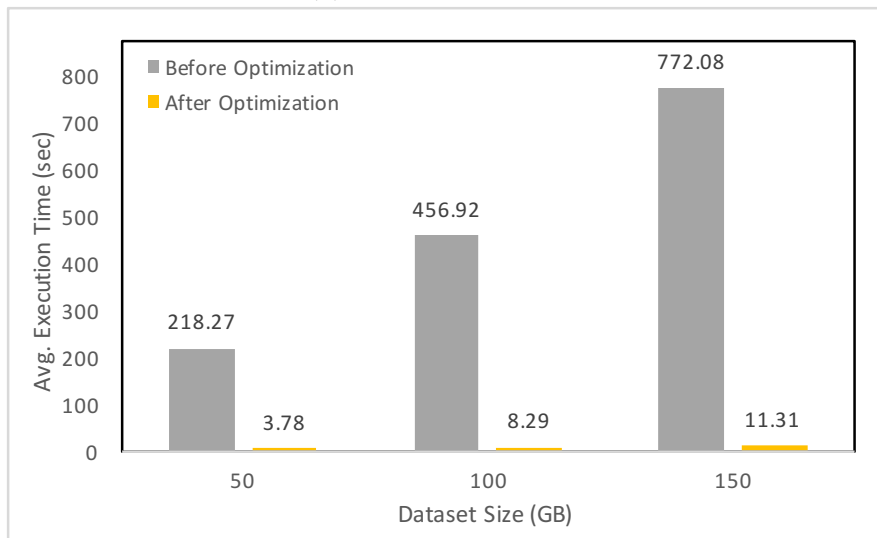
```
collect • reduceByKey((a, b) => a + b) • map((k, v1, v2) =>  $\sqrt{v_1^2 + v_2^2}$ ) • RDD
```

(b) After Substituting `GroupByKey`

Listing 5.4: Scenario 6



(a) Local Benchmarks



(b) Cluster Benchmarks

Figure 5.6: Average Execution Time of Scenario 6 Under Various Input Data Sizes

### 5.3 Case Study

In this section, we present a case study that evaluates the performance of TaBOS with respect to the size of the state-space generated, and the execution time of the different Spark jobs selected based on several strategies. Our case study leverages the “311 service requests” dataset containing call requests to the non-emergency service centers. Each request record consists of the following information: request id, created time, longitude and latitude of the incident location, and the responding agency. Consider that we want to compute for each agency the number of requests made during the busy hours and within a certain distance from the agency’s center.

The corresponding handwritten Spark implementation is presented in Listing 5.5. Lines 1-4 compute the number of requests made during each hour. Lines 6-17 identify for each request: the hour by which the request was made, the responding agency, and the distance from that agency. Finally, lines 19-26, join and filter the two resulting RDDs, then calculate for each agency the remaining number of requests.

```

1  val RDD1 = sc.textFile(...)
      .map(line => line.split(","))
3  .map(x => (format.parse(x(1)).getHours, 1))
      .reduceByKey((v1, v2) => v1 + v2)
5
6  val RDD2 = sc.textFile(...)
7  .map(line => line.split(","))
8  .map(x => {
9      (format.parse(x(1)).getHours, (x(2), (x(5).toDouble, x(6).toDouble)))
10     })
11 .map(x => (x._1, (x._2._1, (x._2._2._1 - agencyX(x._2._1),
12     x._2._2._2 - agencyY(x._2._1))))))
13 .map(x => {
14     (x._1, (x._2._1, (pow(x._2._2._1, 2), pow(x._2._2._2, 2))))
15     })
16 .map(x => (x._1, (x._2._1, x._2._2._1 + x._2._2._2)))
17 .map(x => (x._1, (x._2._1, sqrt(x._2._2))))
19 val result = RDD1.join(RDD2)
      .filter(x => x._2._2._2 < distEpsilon)
21 .filter(x => x._2._1 > hourEpsilon)
      .filter(x => x._1 != 0)
23 .map(x => x._2._2)
      .groupByKey()
25 .mapValues(x => x.size)
      .collect()

```

Listing 5.5: A Handwritten Spark job Example

### 5.3.1 State-Space Size

We first discuss the size of the state-space attained by our synthesis algorithm. For the given implementation in Listing 5.5, TaBOS was able to generate an acyclic graph made of **5130** nodes and **18066** edges. As such, using Spark, a single task can be written in plenty different possible ways. Moreover, we classify the edges in the generated graph according to the sets defined in section 4.3.2. Table 5.1 shows

the count of times each set of rewrite rules was used by our algorithm. According to the table, the transformation fusion rules had the highest count followed by transformation reordering rules.

<b>Rewrite Rule Set</b>	<b>Count of Used Times</b>
Transformation Fusion	9714
Redundancy Elimination	0
Transformation Reordering	6642
Transformation Action Reordering	0
GroupBy-Aggregate	1710

Table 5.1: Count of Applied Rewrite Rules in the Case Study

### 5.3.2 Comparison of Selection Strategies

In the second part of this section, we evaluate and compare the acceleration achieved by the different Spark jobs (i.e., states) selected from the generated state-space based on the following strategies:

1. *Max weight greedy*: In this strategy, all the rewrite rules are assigned equal weights, and the state reached by the longest path is selected.
2. *Max weight without pruning*: In this strategy, the rewrite rules are assigned weights based on the benchmarks obtained from the previous section. The weight assigned for each rewrite rule reflects its influence on the job’s execution time, those accompanied with the highest speedup are assigned the highest weights, and vice versa. Based on these weight, the strategy returns the state reached by the path with the maximum weight.
3. *Max weight with pruning*: It is similar to the previous strategy, however, from a given state, only the edges with the maximum assigned weights are traversed.
4. *Min cost greedy*: This strategy selects the state containing the Spark job with the minimum number of operations.
5. *Min cost without pruning*: This strategy selects the state that contains the Spark job with the minimum computed cost according to the proposed cost model in section 4.6.



6. *Min cost with pruning*: It is similar to the previous strategy, except that the search is driven by our cost model, and only the job with the current minimum weight is explored.

Three unique Spark jobs have been selected by TaBOS based on the above strategies. The first three strategies selected the Spark job depicted in Listing 5.6. Although the same Spark job was selected by the maximum weight strategy with and without pruning, using pruning examined only **989** state, which is five-time less than the total search space (5130 state). Listing 5.7 presents the Spark job selected by the minimum cost greedy strategy, and which consist of only 12 operations. On the other hand, the minimum cost strategy which uses the cost model presented in section 4.6, selected the job depicted in Listing 5.8. Also here, the number of states explored using pruning was **1207** state, which is also low compared to the whole number of states. Therefore, we can deduce that pruning can significantly reduce the generated state-space, thereby, decreasing the running time of the synthesis algorithm.

```

val RDD1 = sc.textFile(...)
    .map(line => {
        val record = line.split(",")
        (format.parse(record(1)).getHours, 1)
    })
    .filter(x => x._1 != 0)
    .reduceByKey((v1, v2) => v1 + v2)
    .filter(x => x._2 > hourEpsilon)

val RDD2 = sc.textFile(...)
    .map(line => {
        val x = line.split(",")
        (format.parse(x(1)).getHours, (x(2), (x(5).toDouble, x(6).toDouble)))
    })
    .filter(x => x._1 != 0)
    .map(x => {
        (x._1, (x._2._1, sqrt(pow(x._2._2._1 - agencyX(x._2._1), 2)
            + pow(x._2._2._2 - agencyY(x._2._1), 2))))
    })
    .filter(x => x._2._2 < distEpsilon)

val result = RDD1.join(RDD2)
    .map(x => (x._2._2._1, 1))
    .reduceByKey((v1, v2) => v1 + v2)
    .collect()

```

Listing 5.6: Spark Job Selected by the Max Weight Strategies

```

val RDD1 = sc.textFile(...)
    .map(line => {
        val x = line.split(",")
        (format.parse(x(1)).getHours, 1)
    })
    .filter(x => x._1 != 0)
    .reduceByKey((v1, v2) => v1 + v2)
    .filter(x => x._2 > hourEpsilon)

val RDD2 = sc.textFile(...)
    .map(line => {
        val x = line.split(",")
        (format.parse(x(1)).getHours, (x(2),
            sqrt(pow(x(5).toDouble - agencyX(x(2)),2) +
                pow(x(6).toDouble - agencyY(x(2)),2))))
    })
    .filter(x => x._2._2 < distEpsilon)

val result = RDD1.join(RDD2)
    .map(x => (x._2._2._1, 1))
    .reduceByKey((v1, v2) => v1 + v2)
    .collect()

```

Listing 5.7: Spark Job Selected by the Min Cost Greedy Strategy

```

val RDD1 = sc.textFile(...)
    .map(line => {
        val record = line.split(",")
        (format.parse(record(1)).getHours, 1)
    })
    .filter(x => x._1 != 0)
    .reduceByKey((v1, v2) => v1 + v2)
    .filter(x => x._2 > hourEpsilon)

val RDD2 = sc.textFile(...)
    .map(line => {
        val x = line.split(",")
        (format.parse(x(1)).getHours, (x(2),
            sqrt(pow(x(5).toDouble - agencyX(x(2)), 2) +
                pow(x(6).toDouble - agencyY(x(2)), 2))))
    })
    .filter(x => x._1 != 0)
    .filter(x => x._2._2 < distEpsilon)

val result = RDD1.join(RDD2)
    .map(x => (x._2._2._1, 1))
    .reduceByKey((v1, v2) => v1 + v2)
    .collect()

```

Listing 5.8: Spark Job Selected by the Min Cost With/Without Pruning Strategies

Since the “GroupBy-Aggregate” rules have shown a tremendous speedup in execution time compared to the other rewrite rules, and to efficiently view the acceleration obtained by applying all possible rewrite rules, we implemented and executed the original Spark job after applying only the “GroupBy-Aggregate” rule. Figures 5.7 and 5.8 shows the average execution time of the original Spark job, compared to that without the `groupByKey` operation and the three discussed Spark jobs generated by TaBOS, we vary the dataset size from 1 GB to 50 GB. Because of the high complexity of the case study and the limited cluster size available, all the implementations of the case studies failed to execute on 100 and 150 GB. Therefore, for the cluster benchmarks, we only reported those of 50 GB. The best execution time was for the Spark job selected by the minimum cost strategy with the cost model. The maximum weight strategy, had slightly higher execution time ( $\simeq 10$  sec). Whereas, the minimum cost strategy with greedy approach had the worst results, proving that using less operation does not always lead to better performance.

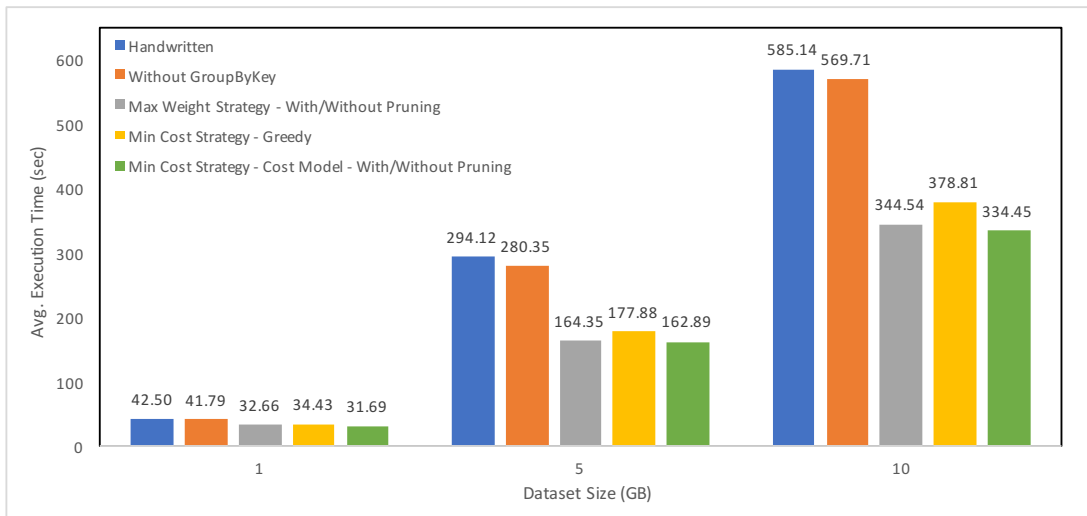


Figure 5.7: Average Execution Time of the Case Study Under Various Input Data Sizes on Local Machine

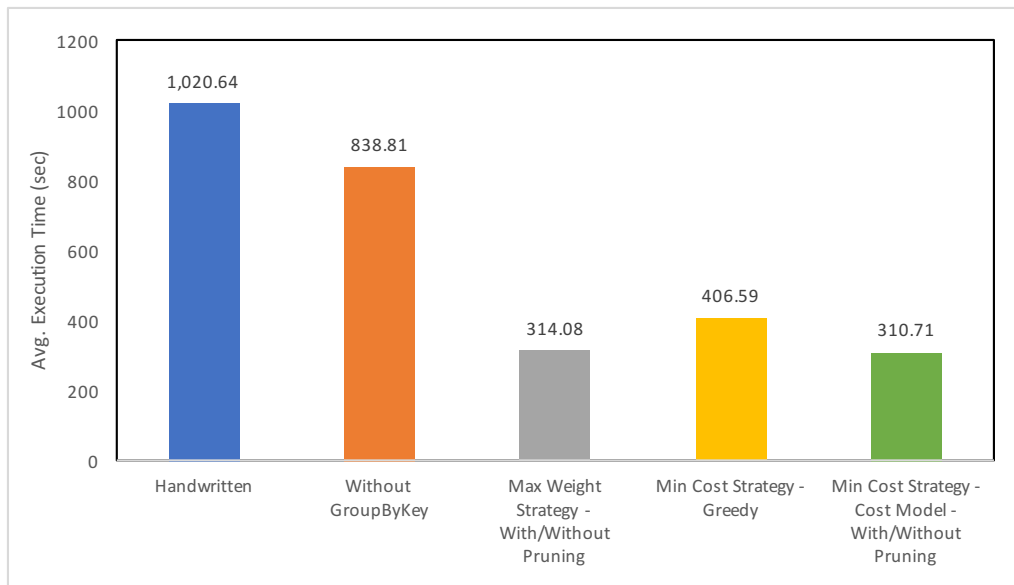


Figure 5.8: Average Execution Time of the Case Study for 50 GB Dataset Size on a Cluster

# Chapter 6

## RELATED WORK

### *Contents*

6.1 Spark Program Synthesis . . . . .	58
6.2 Data Flow Optimization . . . . .	59
6.3 MapReduce Programs Modeling and Optimization . . . . .	60
6.4 Spark Program Optimization . . . . .	60
6.5 Spark Programs Composition . . . . .	62
6.6 Spark RDD Automatic Checkpointing . . . . .	62

Throughout the recent years, several researches have been conducted in the direction of facilitating the process of writing efficient distributed data processing programs. In this chapter, we present some of the presented work for generating, optimizing, and composing spark and other data-parallel applications.

### **6.1 Spark Program Synthesis**

MOLD [17, 18] and BIG $\lambda$  [19] are tools proposed for automating MapReduce like program synthesis (mainly Spark) from inputs of specific type (e.g., sequential code, input-output examples). MOLD translates imperative Java code into parallel code targeting Apache Spark. It works by transforming the sequential code into functional intermediate representation (IR) based on  $\lambda$ -calculus. Then, using rewrite rules, it attempts to introduce parallelism to the intermediate code generating a broad space of equivalent MapReduce programs which is explored using a heuristic search based on a customizable cost function. After that, the final selected program is translated to Scala code that can run on Apache Spark. On the other hand, BIG $\lambda$  synthesizes data-parallel programs from input-output examples. It uses two sets for the synthesis, one composed of high-order sketches (HOS), which are sketches of Spark programs with wildcards in the place of the user-defined functions (UDFs), and another made of predefined components representing UDFs. The synthesis algorithm used by BIG $\lambda$  is composed of two cooperating phases:

1. **Synthesis Phase:** which looks up for all possible components to be matched with the wildcards in the HOS.
2. **Composition phase:** which produce all the possible programs by mapping the selected components with the HOS.

At the end,  $BIG\lambda$  generates an optimal Spark program with respect to a predefined weight function.  $BIG\lambda$  is still limited in the data-parallel operations used for the synthesized programs. Our domain here is different, we handle generating Spark programs which are optimized versions of input Spark programs using rewrite rules that address some of the tricks that can be missed by programmers.

## 6.2 Data Flow Optimization

There have been several attempts for addressing the optimization of data flow programs. Hueske et al. [20, 21] addressed the problem of automatic data flow reordering without full knowledge of the operators’ semantics. The reordering was based on read/write sets of the operators used. These sets were obtained via. static code analysis passes over the 3-address form of the first-order functions used by the operators. In particular, two subsequent operators may be safely switched if they have no read/write or write/write conflict on any of the attributes. Hueske also proposed a query optimizer which enumerates all alternative data flows that can be derived from this setting. However, the query optimizer only handles plan enumeration and does not identify the beneficial re-orderings that can lead to the optimal alternative. The reason that derived SOFA [22, 21], an optimizer for UDF-heavy data flows, to be developed. SOFA rewrites a given data flow into a semantically equivalent one with higher efficiency. Its rewriting system consists of two major components: an operator-property graph and rewrite templates. The operator-property graph is composed of nodes representing the target platform’s operators, manually annotated by the package developer with information describing their input/output behavior and their algebraic and semantic properties, and edges representing the relationships between them. Whereas, the rewrite templates specify the valid reordering, insertion, and deletion of operators. On the other hand, the optimization algorithm used by SOFA consists of two passes. The first pass infers all the precedence relationships between the operators in the data flow based on the operator-property graph and the rewrite templates defined. Once the precedence

graph is built, it generates all possible alternative data flows while performing cost-based pruning. Both optimizers were designed and implemented in the Stratosphere [23] system, a platform for big data analytics, and to our knowledge, there were no attempts to implement them in Spark. Moreover, the discussed optimizations are limited to operation's reordering ignoring other possible optimizations that we handled in our work.

### 6.3 MapReduce Programs Modeling and Optimization

Dörre et al. in [24] proposed a formal data-flow model for MapReduce programs presented in the functional language Haskell, and cost model that estimates and compares the performance of such programs. The cost model takes into consideration the input/output/intermediate data size, the cluster configuration parameter, and the program parameter. It computes the cost by summing two main factors: the startup cost needed to configure the cluster, and the processing cost based on data reading/writing, function computation, and intermediate data shuffling. Dörre also suggested two optimization rules formulated on top of the cost model. The first introduces parallelism and decomposes a sequential Reducer in the MapReduce program to two steps (a Combiner and a Reducer). The second fuses two parallel steps (Mapper and Combiner) into a single one (Mapper) discarding the intermediate results and reducing the communication overhead. The presented models and rules were evaluated and validated in the Apache Hadoop MapReduce framework. However, the proposed work cannot be generalized and does not fit into the Spark computing framework.

### 6.4 Spark Program Optimization

Some frameworks were proposed in the previous years tackling the problem of optimizing Spark programs. For example, Catalyst [25, 26] is an extensible query optimizer which transforms a query plan to Java bytecode that can run by Spark. It is based on representing user's program as trees (aka. logical plans) and applying transformation rules to manipulate these trees. Several sets of rules are defined on top of this framework, and each set handles one of the four phases of query execution: (1) analysis phase, (2) logical plan optimization, (3) physical planning, and (4) code generation. The first phase resolves the reference and types of the

expression in the unresolved logical plan constructed from the program to change it to a resolved logical plan. Then Catalyst applies a number of simplifications (e.g., constant folding, projection pruning, boolean expression simplification) directly on the logical plan to produce an optimized logical plan. Once the logical plan is optimized, several physical plans are generated using physical operators that match the Spark execution engine, and the most optimal plan is selected using a cost model. Also, in this phase, some rule-based physical optimizations, like predicate pushdown from the logical plan to the data source level, are applied. At the final phase, the Java bytecode is generated from the physical plan to be run on each machine. Catalyst leverage some features of the Scala programming language, such as pattern matching for applying the optimization rules, and quasiquotes for simplifying and speeding up the code generation phase. The main advantage of Catalyst is being extensible, it allows external developers to add optimization rules and extend the optimizer in an easy manner.

VEGA [27] instead optimizes a series of Spark programs, by reusing intermediate results that are materialized at the stage’s input and job’s final output of the previous executions of the program. When a new version of the program is executed, it automatically pushes down, using a set of rewrite rules, the modified operations for later data flow stages to enable using the materialized points instead of re-computing the upstream operations. Furthermore, it uses the incremental execution [14] to also avoid re-computing the downstream operations from scratch. VEGA proposes a library that is implemented in two modules, (1) VEGA SQL implemented at the level of Spark SQL by extending the Catalyst optimizer with their rewrite rules, and (2) VEGA RDD extends the Spark RDD abstraction with API that enables the rewrite rules to be applied, this API requires some information (e.g., the inverse function of the used user-defined function) to be provided by the programmer.

In a more recent work, a tool, HYLAS [13], was presented for optimizing Spark queries using semantic preserving transformation rules. The rules were inspired by the deforestation techniques [28] to eliminate intermediate superfluous data structures.



All of the introduced tools performs the rewriting at the compiler level, either directly on the abstract syntax tree (AST) like in HYLAS or on an intermediate tree representation that is translated to AST like in Catalyst and VEGA, which can consume from the total runtime of the application especially when the same application is run several times and the same optimizations have to be re-performed. Moreover, all the rewrite systems proposed are either specific to a certain module (e.g., Spark SQL in case of Catalyst) or limited to a specific optimization (e.g., eliminating intermediate data structures in case of HYLAS). VEGA, on the other hand, can only handle optimization after the second execution of the program, nevertheless, it is memory expensive. To our knowledge, we are the first to propose a framework that translates Spark code to more efficient and robust Spark code through rewrite rules that covers a wide range of optimization tricks.

## 6.5 Spark Programs Composition

Reconfigurable component-based Spark [29] is a high-level specification language that has been recently proposed to compose independently developed Spark applications with predefined dependencies. Given a set of dependent Spark application with input/output interfaces located at the extremities of the program and a configuration defining the mappings between them, a composition is performed by augmenting each application with the proper code to send and receive from/to other applications. Unlike their work, our composition is based on building a complete Spark application from the decomposed parts which deducts the synchronization and communication overhead from the program execution time. Furthermore, we increase the level of visibility of the component to enable input/output interfaces within the program, allowing more composition expressiveness.

## 6.6 Spark RDD Automatic Checkpointing

In [30] an automatic checkpointing algorithm for Spark RDD was proposed. The algorithm starts by selecting the candidate RDDs to be checkpointed, and then based on the utilization rate of JVM old generation heap space, the time by which checkpointing should be performed is decided. The proposed checkpointing algorithm has several drawbacks. First, the selection strategy is based on selecting all RDDs that are included in more than one Spark job. Second, their timing requires

knowing the utilization rate of the JVM heap space of the machines that are running the Spark program.

## Chapter 7

# CONCLUSION AND FUTURE WORK

### *Contents*

7.1 Conclusion . . . . .	64
7.2 Future Work . . . . .	65

### 7.1 Conclusion

Apache Spark is one of the key big data distributed processing frameworks that introduces the concept of resilient distributed datasets leveraging distributed memory. Its in-memory data operation makes it uniquely fast and well-suited for iterative algorithms such as those used in machine learning and graph processing. In addition to that, it provides a developer-friendly API that uses powerful distributed abstractions which are beyond the simple `map/ reduce`, and similar to those in functional programming (e.g., `filter`, `join`, and `collect`). While Spark have all of these advantages, developing complex and efficient Spark application remains to impose effort, burden, and complication on the programmer. Therefore, in this thesis, we propose two solutions that can increase the programmers productivity by automating some of the programming tasks in Spark, such as: (1) integrating independently developed Spark applications, and (2) automatically transform a given Spark program to a more efficient version.

First, we presented CBSpark, a component-based framework that builds a Spark application from independently developed sub-Spark components with input/output interfaces. The input/output interfaces are specified by the developer within a sub-Spark application using a pre-defined placeholder instructions which describes wither waiting or sending a computed dataset. Moreover, the composition of the sub-Spark applications is based on the dependencies between their interfaces. These dependencies are provided using an input configuration file, written by a Domain Specific Language (DSL) that we define, as mappings from input interfaces to output interfaces. Moreover, We have introduced several strategies to automatically (un)persist the output datasets, and compare the different strategies

on simple and real life scenarios.

Second, we propose TaBOS, a transformation based optimizer that can optimize a given Spark program using a set of rewrite rules. A rewrite rules transforms a sequence of operations into different sequence that can optimize the program performance while preserving its semantics. We define several rewrite rules that can accelerate the execution time of a given Spark program. And we further describe the algorithm used for applying the rewrite rules. The algorithm consists of two phases: a synthesis phase which generate a state-space of possible alternative Spark programs, and a selection phase which uses one of several defined strategies to select the most optimal program from the generated space. We evaluate the effectiveness of our system on several simple scenarios and one non-trivial case study. We show a speedup in execution time up to 60 times faster.

## 7.2 Future Work

Our future work goes in several directions:

- Improve the automatic (un)persisting module in CBSpark to be able to:
  - handle the persisting of any RDD and not only the output RDDs. This can be addressed by building a directed acyclic graph of the produced Spark application that represent the dependencies between the computed RDDs and then apply the strategies on the nodes with the several output edges.
  - take the garbage collector utilization into consideration. This can be achieved using dynamic approach that monitors the garbage collector during runtime.
- Expanding TaBOS to be able to:
  - optimize a complete Spark program instead of a single Spark job.
  - optimize Spark program written by any language and not just Scala.
  - use tools that can identify parallelism in imperative code, to be able to refine our “GroupBy-Key” rewrite rules so that it does not stay limited to using operation of the Iterator class in Scala.

- use module that can automatically identify the best selection strategy based on input dataset features (e.g., structure and size).
- A third direction would be to establish an integration between the two frameworks.

## References

- [1] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
- [2] T. White, *Hadoop: The definitive guide*. ” O’Reilly Media, Inc.”, 2012.
- [3] “Hadoop map/reduce tutorial,” [http://hadoop.apache.org/common/docs/r0.20.0/mapred\\_tutorial.html](http://hadoop.apache.org/common/docs/r0.20.0/mapred_tutorial.html), 2016.
- [4] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The hadoop distributed file system,” in *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. IEEE, 2010, pp. 1–10.
- [5] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster computing with working sets.” *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.
- [6] “Apache spark project,” <http://spark.apache.org>.
- [7] A. W. Brown, “Component-based development,” 2000.
- [8] H. Karau and R. Warren, *High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark*. O’Reilly Media, 2017. <https://books.google.com.lb/books?id=o0IIDwAAQBAJ>
- [9] E. Visser, “A survey of rewriting strategies in program transformation systems,” *Electronic Notes in Theoretical Computer Science*, vol. 57, pp. 109–143, 2001.
- [10] N. Dershowitz and J.-P. Jouannaud, “Rewrite systems,” in *Formal models and semantics*. Elsevier, 1990, pp. 243–320.
- [11] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. 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*. USENIX Association, 2012, pp. 2–2.
- [12] “Geonames database,” <https://www.kaggle.com/geonames/geonames-database>.

- [13] Z. A. Kocsis, J. H. Drake, D. Carson, and J. Swan, “Automatic improvement of apache spark queries using semantics-preserving program reduction,” in *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*. ACM, 2016, pp. 1141–1146.
- [14] A. Gupta, I. S. Mumick, and V. S. Subrahmanian, “Maintaining views incrementally,” *ACM SIGMOD Record*, vol. 22, no. 2, pp. 157–166, 1993.
- [15] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, “Apache hadoop yarn: Yet another resource negotiator,” in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.
- [16] “311 service requests dataset.” <https://data.cityofnewyork.us/Social-Services/311-Service-Requests-from-2010-to-Present/erm2-nwe9>.
- [17] C. Radoi, S. J. Fink, R. Rabbah, and M. Sridharan, “Translating imperative code to mapreduce,” *ACM SIGPLAN Notices*, vol. 49, no. 10, pp. 909–927, 2014.
- [18] S. Fink, R. Rabbah, C. A. Radoi, and M. Sridharan, “Automatic conversion of sequential array-based programs to parallel map-reduce programs,” Sep. 5 2017, uS Patent 9,753,708.
- [19] C. Smith and A. Albarghouthi, “Mapreduce program synthesis,” *ACM SIGPLAN Notices*, vol. 51, no. 6, pp. 326–340, 2016.
- [20] F. Hueske, M. Peters, M. J. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas, “Opening the black boxes in data flow optimization,” *Proceedings of the VLDB Endowment*, vol. 5, no. 11, pp. 1256–1267, 2012.
- [21] A. Rheinländer, U. Leser, and G. Graefe, “Optimization of complex dataflows with user-defined functions,” *ACM Computing Surveys (CSUR)*, vol. 50, no. 3, p. 38, 2017.
- [22] A. Rheinländer, A. Heise, F. Hueske, U. Leser, and F. Naumann, “Sofa: An extensible logical optimizer for udf-heavy data flows,” *Information Systems*, vol. 52, pp. 96–125, 2015.

- [23] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl *et al.*, “The stratosphere platform for big data analytics,” *The VLDB Journal*, vol. 23, no. 6, pp. 939–964, 2014.
- [24] J. Dörre, S. Apel, and C. Lengauer, “Modeling and optimizing mapreduce programs,” *Concurrency and Computation: Practice and Experience*, vol. 27, no. 7, pp. 1734–1766, 2015.
- [25] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, “Spark sql: Relational data processing in spark,” in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 1383–1394.
- [26] “Spark sql.” <https://spark.apache.org/sql/>.
- [27] M. Interlandi, S. D. Tetali, M. A. Gulzar, J. Noor, T. Condie, M. Kim, and T. Millstein, “Optimizing interactive development of data-intensive applications,” in *Proceedings of the Seventh ACM Symposium on Cloud Computing*. ACM, 2016, pp. 510–522.
- [28] P. Wadler, “Deforestation: Transforming programs to eliminate trees,” in *European Symposium on Programming*. Springer, 1988, pp. 344–358.
- [29] M. Jaber, M. Nassar, W. A. R. A. Orabi, B. A. Farraj, M. O. Kayali, and C. Helwe, “Reconfigurable and adaptive spark applications,” in *CLOSER 2017 - Proceedings of the 7th International Conference on Cloud Computing and Services Science, Porto, Portugal, April 24-26, 2017.*, 2017, pp. 84–91.
- [30] W. Zhu, H. Chen, and F. Hu, “Asc: Improving spark driver performance with automatic spark checkpoint,” in *Advanced Communication Technology (ICACT), 2016 18th International Conference on*. IEEE, 2016, pp. 607–611.



