On performance optimization and prediction of parallel computing frameworks in big data systems

Haifa AlQuwaiee
New Jersey Institute of Technology, haifa.alquwaiee@gmail.com

ABSTRACT

ON PERFORMANCE OPTIMIZATION AND PREDICTION OF PARALLEL COMPUTING FRAMEWORKS IN BIG DATA SYSTEMS

by
Haifa AlQuwaiee

A wide spectrum of big data applications in science, engineering, and industry generate large datasets, which must be managed and processed in a timely and reliable manner for knowledge discovery. These tasks are now commonly executed in big data computing systems exemplified by Hadoop based on parallel processing and distributed storage and management. For example, many companies and research institutions have developed and deployed big data systems on top of NoSQL databases such as HBase and MongoDB, and parallel computing frameworks such as MapReduce and Spark, to ensure timely data analyses and efficient result delivery for decision making and business intelligence.

This dissertation investigates and addresses two main challenges in such big data systems: i) performance optimization for distributed information composition, and ii) performance modeling and prediction of big data applications. To address the first challenge, analytical cost models are constructed to formulate a Distributed Information Composition problem in Big Data Systems, referred to as DIC-BDS, to aggregate multiple datasets stored as data blocks in Hadoop Distributed File System (HDFS) using a composition operator of specific complexity to produce one final output. DIC-BDS is rigorously proved to be NP-complete, and two heuristic algorithms are proposed. Extensive experiments are conducted with various composition operators of commonly considered degrees of complexity, and experimental results illustrate the performance superiority of the proposed solutions over existing methods. To address the second challenge, a class of regression-based machine learning models is proposed to predict the execution performance of Spark-HBase applications in Hadoop. Accurate performance modeling and prediction are critical to optimizing application performance through strategic resource allocation with suitable parameter settings and also to providing an effective recommendation of optimal system configurations to end users. An in-depth exploratory analysis is conducted to identify an exhaustive set of system parameters across multiple technology layers including Spark and HBase, and examine their effects on the execution time of Spark-HBase applications. Based on these analysis results, a subset of critical parameters is selected to develop a performance predictor using regression-based machine learning. Experimental results show that the resulted predictor achieves high accuracy with different algorithms in comparison.

ON PERFORMANCE OPTIMIZATION AND PREDICTION OF PARALLEL COMPUTING FRAMEWORKS IN BIG DATA SYSTEMS

by
Haifa AlQuwaiee

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Department of Computer Science

December 2021

APPROVAL PAGE

ON PERFORMANCE OPTIMIZATION AND PREDICTION OF PARALLEL COMPUTING FRAMEWORKS IN BIG DATA SYSTEMS

Haifa AlQuwaiee

Prof. Chase Wu, Dissertation Advisor — Date
Professor of Computer Science, NJIT

Prof. Ali Mili, Committee Member — Date
Professor of Computer Science, NJIT

Prof. Zhi Wei, Committee Member — Date
Professor of Computer Science, NJIT

Prof. Jing Li, Committee Member — Date
Assistant Professor of Computer Science, NJIT

Prof. Dantong Yu, Committee Member — Date
Associate Professor, Martin Tuchman School of Management, NJIT

BIOGRAPHICAL SKETCH

Author: Haifa AlQuwaiee

Degree: Doctor of Philosophy

Date: December 2021

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 2021
- Master of Science in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 2015
- Bachelor of Computer Science and Education,
  Imam Abdulrahman Bin Faisal University, Saudi Arabia, 2005

Major: Computer Science

Presentations and Publications:

H. AlQuwaiee, and C. Q. Wu "On Performance Modeling and Prediction for Spark-HBase Applications in Big Data Systems." *under review.*

H. AlQuwaiee, and C. Q. Wu "On Distributed Information Composition for Multi-source Big Media-data." *International Symposium on Media Big Data Intelligent Media*, Hangzhou, China, November 2019.

H. AlQuwaiee, S. He, C. Q. Wu, Q. Tang, and X. Shen "On Distributed Information Composition in Big Data Systems." *Proceedings of the 15th Conference on eScience (eScience)*, 168-177, San Diego, USA, Sep 24th-27th 2019.

To the Gallant Soul that departed us too soon;
the Loved, Cherished, Precious and Honourable One,

Thank you for everything ...

Rest In Peace and May Love Surround You Wherever You Are ...

ACKNOWLEDGMENT

In the name of Allah (God), the Most Gracious, the Most Merciful.

First of all, I would like to express my deepest and sincerest gratitude to Allah for supporting me with the strength and blessings to reach this point of my PhD journey; Alhamdulillah.

Second, I would like to thank my advisor, Prof. Chase Wu for his valuable guidance, support, and constant motivation during my doctoral journey at New Jersey Institute of Technology (NJIT). I truly appreciate that he allowed me to have the space, time and resources to ignite my passion for research, flourish my intention and fuel it to improve my research skills.

I am also very thankful to the committee members for being willing to serve in my dissertation committee Professors: Ali Mili, Zhi Wei, Jing Li, and Dantong Yu. Special thanks go to Professor Vincent Oria for believing in me and for his bright opinion about my abilities as a Researcher.

My sincere gratitude goes to the Saudi scholarship program for investing in me and giving me the opportunity to pursue my graduate studies.

Also, I would like to thank my lab mates: Songlin He, Qianwen Ye and Wuji Liu at the Center for Big Data (CBD) for their valuable discussion, kindness and help.

Finally, I would like to extend my gratitude to my lovely family and friends: my parents, my siblings, my niece and nephew, for their endless love, prayers and support.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

Chapter **Page**

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES
(Continued)

CHAPTER 1

INTRODUCTION

Nowadays, a wide spectrum of applications in various domains produce colossal amounts of data on a daily basis, which must be managed and processed in a timely and reliable manner for knowledge discovery [20]. This computing process is now commonly executed in big data systems exemplified by Hadoop based on parallel processing and distributed storage and management. The tasks in this process are typically represented as computing modules and arranged in a workflow structure[1] [62] for coordinated data processing and analysis.

In this dissertation, we investigate and address two main challenges in such big data systems: i) performance optimization for distributed information composition, and ii) performance modeling and prediction of big data applications.

Parallel computing has become a norm for big data processing and it is of great importance to optimizing the performance of such frameworks to meet stringent application requirements. For example, in workflow-based applications, there may exist multiple computing modules processing and generating data (intermediate or semi-final results) in parallel at different locations. Moreover, In big data systems, even for a single computing module implemented within parallel computing frameworks such as MapReduce, it may use multiple reducers to produce outputs stored as different files/data blocks in Hadoop Distributed File System (HDFS) [51]. Since each reducer processes a subset of (key, value) pairs depending on the associated key assigned to that reducer, it generally does not have access to all (key, value)

[1]In the context of workflows, these computing components are usually referred to as modules that represent either a serial computing program or a parallel processing job such as a MapReduce application in Hadoop.

pairs. However, in many big data applications, such distributed information must be aggregated to produce one final result.

To address this challenge, we construct analytical cost models and formulates a Distributed Information Composition problem in Big Data Systems, referred to as DICBDS, to aggregate multiple datasets stored as data blocks in HDFS using a composition operator of specific complexity. We prove DICBDS to be NP-complete, and propose two heuristic algorithms, namely, Fixed-window Distributed Composition Scheme (FDCS) and Dynamic-window Distributed Composition Scheme with Delay (DDCS-D). Extensive experiments are conducted in Google clouds with various composition operators of commonly considered degrees of complexity including $O(n)$, $O(n \log n)$, and $O(n^2)$, and experimental results illustrate the performance superiority of the proposed solutions over existing methods. Specifically, FDCS outperforms all other algorithms in comparison with a composition operator of complexity $O(n)$ or $O(n \log n)$, while DDCS-D achieves the minimum total composition time with a composition operator of complexity $O(n^2)$. These algorithms provide an additional level of data processing for efficient information aggregation in existing workflow and big data systems.

There are an increasing number of large-scale applications in business and scientific domains that require a combination of parallel computing with distributed data storage and management for big data processing. In fact, it has become a widely adopted practice in industry to deploy big data systems such as Hadoop on top of NoSQL databases (such as HBase [7] and MongoDB [8]), and employ parallel computing frameworks (such as MapReduce [18] and Spark [72, 71]) to ensure timely data processing and efficient delivery of analysis results in support of decision making and business intelligence.

We focus our research on Spark applications that run over Hadoop/HDFS [61] as a data storage system and HBase as a data management system, which are referred to as Spark-HBase applications in this context. The execution of such applications in big data systems typically has a life circle of computing that spans through several stages across multiple technology layers: submitting the application to YARN as a resource manager, assigning executors in Spark for data processing, coordinating with *RegionServers* in HBase to determine a logical data block, and accessing the actual data block stored in Hadoop Distributed File System (HDFS) [4]. Each of these layers has a large set of parameters for configuration, and it is crucial for any given Spark-HBase application to decide a subset of configurable parameters according to its computing needs and performance requirements, e.g., the number of executors and the number of cores for each executor in Spark as well as database operation API-related parameters in HBase. Deciding an appropriate setting of effective parameters is critical to understanding and optimizing application performance for end users, and also to maximizing the utilization of system resources for infrastructure providers. However, it is challenging for end users, who are primarily domain experts, to decide a satisfactory configuration for executing Spark-HBase applications in such complex computing systems. The execution complexity of Spark-HBase workflows compounded by system dynamics makes it a daunting task to select and configure a right set of parameters across different layers of the technology stack. In fact, in most of the existing big data systems, the parameters are typically set with default values, which, unfortunately, do not always lead to the best performance. These default values, once set, are oftentimes used for all applications of disparate types as end users generally do not have enough knowledge in computing to modify the system configuration. Even with the aid of certain knowledge, this problem is still largely unexplored and unresolved.

To address this challenge, we tackle a general problem of optimizing the execution performance of big data applications deployed on high-performance computing platforms. We investigate the problem of modeling and predicting the performance of Spark-HBase applications in big data systems by strategically selecting a subset of hyper parameters and setting their values using machine learning. Our goal is to achieve an accurate prediction of execution time using a performance-influence model that takes influential parameters as input features. Towards this goal, we start by conducting a large number of experiments to run various Spark-HBase applications with different parameter settings and collect their corresponding performance measurements. Such measurement data conveys informative knowledge and provides insights into the performance pattern and execution behavior of these applications under different configurations in big data systems, which facilitate performance optimization and configuration recommendation for such applications. We then conduct an in-depth comparison-based analytical study to investigate the effects of these parameters on application performance. Based on the data collected from the exploratory analysis and aided by domain knowledge, we design a class of regression-based prediction models to estimate the execution time of Spark-HBase applications, and illustrate the accuracy of such models using different performance metrics.

CHAPTER 2

ON DISTRIBUTED INFORMATION COMPOSITION IN BIG DATA SYSTEMS

2.1 Introduction

Nowadays, a wide spectrum of applications in science, engineering, and business domains are generating data of colossal amounts, which require big data computing systems for timely and efficient processing and analysis [20]. In many of these applications, various tasks for data generation, processing, visualization, and analysis are represented as computing modules and assembled in a workflow structure[1] [62]. Particularly, in the broad science community, workflow systems have been recognized as an important technology for mission-critical applications, allowing execution and management of complex computations on distributed resources [17, 19]. As the era of big data is widely emerging, workflow applications have been increasingly deployed in big data systems as exemplified by Hadoop [61, 51] using different computing frameworks such as MapReduce for batch parallel data processing [18], Spark for in-memory data processing [71] and Storm for streaming data processing [1]. In workflow-based applications, there may exist multiple computing modules processing and producing data (intermediate or semi-final results) in parallel at different locations, which must be aggregated to produce the final result. Some scientific workflows such as Montage [9, 31] and CyberShake [31] follow an aggregation approach to combine different results or data from different sub-workflows or components of a workflow. In big data computing systems, even for a single computing module implemented within distributed processing frameworks such as MapReduce, it may use multiple reducers to produce outputs stored as different files/data blocks in

[1]In the context of workflows, these computing entities are usually referred to as modules that represent either a serial computing program or a parallel processing job such as a MapReduce application in Hadoop.

Hadoop Distributed File System (HDFS) [51]. Since each reducer processes a subset of (key, value) pairs depending on the associated key assigned to that reducer, it generally does not have access to all (key, value) pairs. In the simplest case to identify the top n words with the highest use frequency in a large text document, it is generally insufficient to use a classical WordCount program as each reducer only outputs the number of occurrences for a subset of words, and another procedure is typically required to aggregate all these occurrences for a global sorting to determine the top word list as the final result. In this paper, analytical cost models are constructed to formulate a Distributed Information Composition problem in Big Data Systems, referred to as DIC-BDS, to aggregate multiple datasets stored as data blocks in Hadoop Distributed File System (HDFS) using a composition operator of specific complexity to produce one final output. DIC-BDS is rigorously proved to be NP-complete, and two heuristic algorithms are proposed: Fixed-window Distributed Composition Scheme (FDCS) and Dynamic-window Distributed Composition Scheme with Delay (DDCS-D). Extensive experiments are conducted in Google clouds with various composition operators of commonly considered degrees of complexity including $O(n)$, $O(n \log n)$, and $O(n^2)$, and compare the performance with existing methods in the literature in terms of execution time. The experimental results show the performance superiority of the proposed algorithms over existing methods. Specifically, FDCS achieves a performance improvement of about 31-61% and 44-65% on average with a composition operator of complexity $O(n)$ and $O(n \log n)$, respectively, and DDCS-D achieves a performance improvement of about 61-95% on average with a composition operator of complexity $O(n^2)$ over other algorithms in comparison. The proposed algorithms provide an additional level of data processing for efficient information aggregation in existing workflow and big data systems.

2.2 Related Work

In this section, we conduct a survey of related work on distributed information composition in different computing environments.

In [42], Mayer *et al.* formulated a set of partitioning and scheduling problems in TensorFlow and proved them to be NP-complete. In [68], Yun *et al.* studied a workflow optimization problem and designed an approach that integrates workflow mapping and on-node scheduling. Although these problems are discussed in the framework of TensorFlow or in a generic computing environment, they are conceptually similar to the problem under this study. However, the solutions proposed in their work cannot be directly applied to this problem since they require prior knowledge about the execution of a workflow, which is not always available in practice.

This work is focused on distributed information composition in big data systems such as Hadoop and provides an additional level of data processing to improve the performance of existing workflow engines and computing frameworks such as MapReduce. In particular, a significant number of efforts have been made to improve the performance of the MapReduce framework. In [21], Elteir *et al.* proposed asynchronous data-processing techniques to enhance the performance of MapReduce without considering data locality, which, however, is an important aspect in this work and has been extensively explored in many other methods [45, 10, 15, 2, 30, 29, 24, 69, 56].

Information integration or aggregation has also been studied in other contexts such as service-oriented computing [43] and image composition in volume visualization. Particularly, for the latter, several approaches have been proposed to decide the composition order of partial images to minimize the total image composition time on a cluster [64, 67, 46, 55]. These approaches consider minimizing the number of communication messages. Since this study focuses on the composition time that mainly depends on the data size and the complexity of the composition operator, the solutions originally designed for image composition are not directly applicable. For instance, Wu *et al.* proposed an optimized approach for image composition with a linear pipeline for efficient image delivery to a remote client [64]. However, the proposed algorithm does not consider the complexity of composing a segment in each step/phase. Moreover, they considered data transfer throughput over a wide-area network connection for remote visualization, which is out of the scope of this work.

In this work, we adopt two algorithms from the literature for performance comparison with our proposed algorithms. The first one follows a simple greedy procedure to process and compose distributed data, and the second one is inspired by a data aggregation method developed in the field of sensor networks [38, 66, 27, 41]. More specifically, in Periodic Sensor Networks (PSN) [25, 53], this method guides sensors to send data collected over a period of time to a Cluster Head (CH) through an aggregation tree.

2.3 Problem Formulation

In this section, we construct analytical cost models and defines formally a Distributed Information Composition problem in Big Data Systems.

2.3.1 Cost Models

Cluster Model As illustrated in Figure 2.1, a cluster is modeled as a tree of Physical Machines (PMs) connected via high-speed switches. Without loss of generality, two-level switches are considered. The top-level or root switch S_{root} has a capacity $C_{S_{root}}$, and connects other in-rack switches S_{in_rack}, each of which connects a number of PMs that are located in the same rack R. Each PM is associated with a resource profile that specifies the CPU frequency f_{CPU}, memory size s_{RAM}, I/O speed $r_{I/O}$, disk capacity c_{disk}, and a Network Interface Card (NIC) with uplink bandwidth BW_{up} and downlink bandwidth BW_{down}. Also, each PM provisions a number of

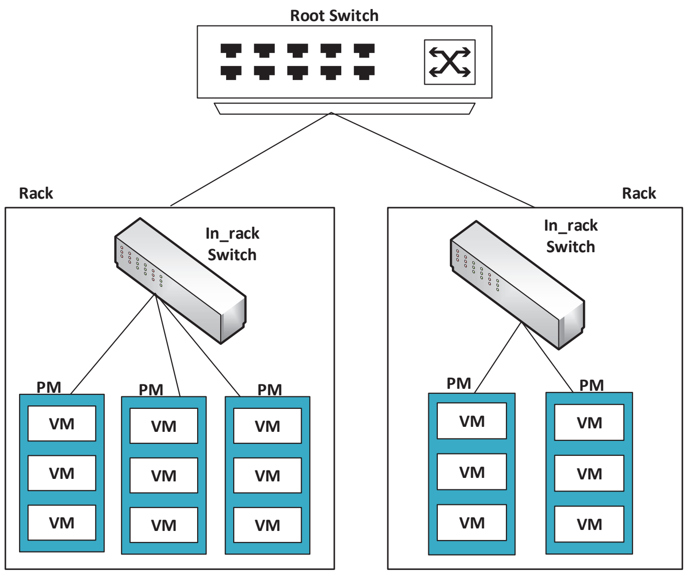

Figure 2.1 A general cluster structure.

Virtual Machines (VMs) and each VM is associated with a set of performance attributes including CPU frequency f'_{CPU}, I/O speed $r'_{I/O}$, and disk capacity c'_{disk} [62]. However, provisioning VMs on PMs is beyond the scope of this paper.

Composition Model We consider a generic scheme of information composition that can be applied to many scenarios such as aggregating the output from a workflow or the output of multiple reduce tasks in the MapReduce framework. Mainly, the composition model has two components: 1) datasets to be composed, and 2) a composition operator.

Datasets Suppose that there are n datasets (ds_1, ds_2, ..., ds_n) that have to be composed into one final output F. Also, each dataset is of different size s_{ds}, where s denotes the size of dataset ds in bytes. If Hadoop system is considered, the dataset or data block size ranges from 64 to 128 MB as widely implemented in HDFS [51].

In the MapReduce framework [18], such datasets could be the intermediate results (temporary files) after executing map tasks, or the output of reduce tasks. The intermediate data produced by a map task is generally stored locally on the corresponding map node [61], but could be stored in HDFS if there is not enough storage space on the map node. On the other hand, the output of a reduce task is generally stored directly in HDFS [51]. In this work, datasets are considered as HDFS data blocks distributed on a cluster.

Composition Operator Each dataset *ds* must be processed by a composition operator $\oplus$, which could be in different forms, for example, a machine learning program based on a stochastic gradient descent (SGD) procedure to train the model or a statistical function to calculate a single value such as the sum or average of some measurements. Different composition operators are typically of different time complexity. Also, each operator $\oplus$ takes two operands $opr1$ and $opr2$ and produces output *comp_ds* of different size that resides on a node of the cluster. Once some datasets are available and ready to be composed, the location where a composition process takes place has to be specified. Determining such location depends on the resource availability of the cluster as well as the computational and storage requirements of the composition operator. Furthermore, data locality should be always considered to minimize the communication overhead.

Time Cost

Transfer Time In a cluster environment, datasets are often distributed on different nodes and have to be transferred over the network for composition. In general, *transfer_time = data_size / network_bandwidth*. On each PM, the uplink bandwidth may be equally shared (if using TCP-friendly protocols) if the PM sends data concurrently to other PMs; similarly, the downlink bandwidth may be equally shared if the PM receives data concurrently from other PMs. The time cost T_{dt} of data transfer is determined by both the data size DS and the sharing dynamics of bandwidth BW [62]:

$$T_{dt} = \frac{DS}{min(\frac{BW_{up}}{n_s}, \frac{BW_{down}}{n_r})}, \tag{2.1}$$

where n_s and n_r are the number of concurrent data transfers from a sender PM_s and to a receiver PM_r, respectively.

I/O Cost Generally, in such distributed environment, data is stored using a distributed file system e.g., HDFS where data is stored as blocks. A block can exist locally on the computing node node, or remotely over the network, and the cost of accessing the data is is denoted as $T_{I/O}$ However, computing the actual $T_{I/O}$ time requires the knowledge about the size of the data being accessed and the speed of the I/O operation which could be either a *read* or *write* operation.

Composition Time Figure 2.2 illustrates a single-composition process, where two datasets are aggregated by a composition operator $\oplus$. The time of such single-composition process is calculated as:

$$T_C = T_{I/O} + T_{dt} + f_{CT}(O(\oplus), DS), \tag{2.2}$$

where $T_{I/O}$ is the time consumed to read the input data and write the output data, T_{dt} is the total time for transferring the data from their source to the destination, and $f_{CT}(O(\oplus), DS)$ is a function that computes the time for the composition operation given the complexity $O(\oplus)$ of the composition operator and the data size DS to be composed. For a multi-composition process taking place concurrently in parallel, Equation 2.2 is insufficient to model the composition time. Considering a dynamic case where the composition process starts at different times, and since multiple compositions take place concurrently, the longest one is considered as the total time

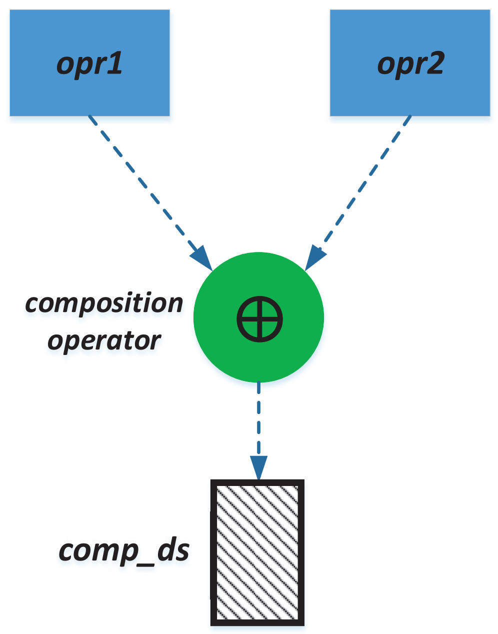

Figure 2.2 A single-composition operation.

needed for composition. In other words, the critical path (CP) is considered in this distributed scheme to define the total composition time (TCT), i.e.,

$$TCT = \sum (T_{I/O} + T_{dt} + f_{CT}(O(\oplus), DS))_{CP}, \tag{2.3}$$

which is the sum of I/O time, transfer time, and composition time along the CP.

2.3.2 Problem Definition

We formally define a Distributed Information Composition problem in Big Data Systems, referred to as DIC-BDS:

Definition 1. *Given n datasets (ds_1, ds_2, ... , ds_n) that: (i) are different in sizes (s_{ds_1}, s_{ds_2}, . . . , s_{ds_n}), (ii) become available at different time points (t_1, t_2, . . . ,t_n), and (iii) are distributed across m virtual machines (VM_1, VM_2, $\cdots$, VM_m) provisioned on a number of PMs, the aim is to compose these datasets using a composition operator $\oplus$ that takes two operands $opr1$ and $opr2$ at a time and follow a composition scheme to produce one final output F to minimize the TCT.*

2.3.3 Complexity Analysis

The NP-completeness of DIC-BDS is proved by reducing an existing NP-complete problem, Single Execution Time Scheduling (SETS) [58], to it in polynomial time.

First, a decision version of the problem is stated as follows:

Definition 2. *Given the input of DIC-BDS as defined in Definition 1 and a bound B, does there exist a composition scheme that yields the TCT such that $TCT \leq B$?*

The SETS problem [58, 42] is defined as follows: Given a set S of jobs that take unit time, a partial order $\prec$ on S, $k^{'}$ processors, and a time limit t_{max}, is there a scheduling function $g : S \rightarrow \{0, ..., t_{max} - 1\}$ such that the following three properties hold? (i) The scheduling function respects the ordering relation, i.e., $v \in S \prec v' \in S \rightarrow g(v) < g(v')$. (ii) The time limit is not exceeded, i.e., $\forall v \in S : g(v) < t_{max}$, and (iii) There are at most $k^{'}$ active jobs at each point of time, i.e., $\forall i \in \{0, \ldots , t_{max}\}$: $|\{v \in S | g(v) = i\}| \leq k'$.

Theorem 1: DIC-BDS $\in$ NP-complete.

Proof. Obviously, DIC-BDS is in the class of NP. Its NP-hardness is proved by reducing SETS to it as follows:

Table 2.1 Notations Used in the Cost Models and Problem Definition (Continued)

Parameters	**Definitions**
S_{root}	the root switch
$C_{S_{root}}$	the capacity of the root switch
$S_{in-rack}$	an in-rack switch
R	a rack of PMs
T_{dt}	time cost of data transfer
DS	data size
BW_{up}	the uplink bandwidth
BW_{down}	the downlink bandwidth
n_s, n_r	the number of concurrent data transfers from a sender PM_s, to a receiver PM_r
BWs_{up}	the uplink bandwidth within a rack
BWs_{down}	the downlink bandwidth within a rack
ns_s, ns_r	the number of concurrent data transfers from a sender $S_{in-rack}$, to a receiver $S_{in-rack}$
PM_i	the i-th PM
PM_s	the sender PM
PM_r	the receiver PM
$f_{CPU(i)}$	the CPU frequency of PM_i
$s_{RAM(i)}$	the memory size of PM_i
$r_{I/O(i)}$	the I/O speed of PM_i
$c_{disk(i)}$	the disk capacity of PM_i

Table 2.2 (Continued) Notations Used in the Cost Models and Problem Definition

Parameters	**Definitions**
VM_i	the i-th VM
m	the number of VMs
f'_{CPU}	the CPU frequency of a VM
s'_{RAM}	the memory size of a VM
$r'_{I/O}$	the I/O speed of a VM
c'_{disk}	the disk capacity of a VM
$T_{I/O}$	the cost of I/O
ds	dataset for composition
n	the number of datasets for composition
s_{ds}	the size of dataset ds in bytes
t_{ds}	the available time of dataset ds
$\oplus$	composition operator
$opr1$	first operand
$opr2$	second operand
$comp\_ds$	dataset resulting from a composition process
F	final composition result
TCT	Total Composition Time
T_C	Composition Time of a composition process
CP	Critical Path
$f_{CT}(O(\oplus), DS)$	function to compute T_C given $O(\oplus)$ and DS

Let I_{SETS} be an arbitrary instance of SETS, which has a set of jobs S and a partial order $\prec$ on S. Accordingly, an instance of DIC-BDS is constructed and denoted as $I_{DIC-BDS}$. For each partial order $\prec_i$ of I_{SETS}, a corresponding bucket b_i of $I_{DIC-BDS}$ is constructed such that the number of datasets in each bucket is the same as the number of jobs in the corresponding partial order of I_{SETS}. Also, the size of each dataset is equivalent in value to the number of instructions in the corresponding job. Moreover, the complexity of the composition operator $\oplus$ is determined by the size of the second operand $opr2$ as illustrated in the following example:

Suppose that I_{SETS} has a partial order $\prec_1$ that has three jobs: J_0, J_2, and J_1. Accordingly, $I_{DIC-BDS}$ has a bucket b_1 that has three datasets: ds_0, ds_2, and ds_1. Following the partial order $\prec_1$, starting by composing the first dataset with a dummy data set ds_{dummy} as $\oplus(ds_{dummy}, ds_0)$, which produces the first result denoted as r_0 of size s_{ds_0}. The second composition is $\oplus(r_0, ds_2)$ that produces a result of r_1 with an output of size s_{ds_2}. The final composition is $\oplus(r_1, ds_1)$ that produces a result of r_2 with an output of size s_{ds_1}.

Furthermore, in $I_{DIC-BDS}$, the number of VMs is set to be k', which is the same as the number of processors in I_{SETS}. Also, since the processors in I_{SETS} are homogeneous, only homogenous VMs in $I_{DIC-BDS}$ are considered. Furthermore, the focus is only on the execution time of I_{SETS}, which is equivalent to the composition time of $I_{DIC-BDS}$. Therefore, the constructed $I_{DIC-BDS}$ is a special case of DIC-BDS where both the transfer time and I/O time are set to be zero. It is obvious that this instance construction process can be done in polynomial time.

Next, we show that if there is a solution to I_{SETS}, that solution solves $I_{DIC-BDS}$ as well. Assuming that the answer of I_{SETS} is *true*, this means that there exists a scheduling scheme such that the three properties of SETS are satisfied. If that scheduling scheme is used as an order to perform the composition process on the corresponding datasets of $I_{DIC-BDS}$, the total composition time is minimized. On the other hand, if there is a solution to $I_{DIC-BDS}$, it implies that there exists an order that guarantees to minimize the total composition time, and this order can be used to schedule the execution of the corresponding jobs of I_{SETS}.

Hence, if the answer to the given instance of SETS is YES or NO, the answer to the constructed instance of DIC-BDS is also YES or NO, and vice versa. This completes the NP-hardness proof of DIC-BDS. $\square$

2.4 Algorithm Design

We design a Distributed Composition Scheme (DCS) as a heuristic approach to solve DIC-BDS defined in Section 2.3.2.

The main goal of DCS is to minimize TCT. In this scheme, there are two main types of datasets: (1) the original given datasets (ds_1, ds_2, $\cdots$, ds_n) that become available for composition at different time points (t_{ds_1}, t_{ds_2} ,$\cdots$, t_{ds_n}) , and (2) the intermediate results that become available during the entire composition process ($comp\_ds_1$, $comp\_ds_2$, $\cdots$, $comp\_ds_l$). According to Eq. 2.3, the composition time is defined as the sum of three time cost components: I/O time, data transfer time, and time consumed by the composition operator $\oplus$ for data composition. Typically, the composition time with a given composition operator is considered to be fixed, and the I/O time for reading/writing a given amount of datasets does not vary significantly. However, a network-based data transfer is dynamic in nature, largely depending on the location of the datasets. Thus, we focus on minimizing the time cost of data transfer by considering data locality. There are two main phases in the proposed DCS approach. The first phase is to partition the datasets into groups, and for that, we design two partitioning algorithms:

1. FDCS: Fixed-window Distributed Composition Scheme

2. DDCS-D: Dynamic-window Distributed Composition Scheme with Delay

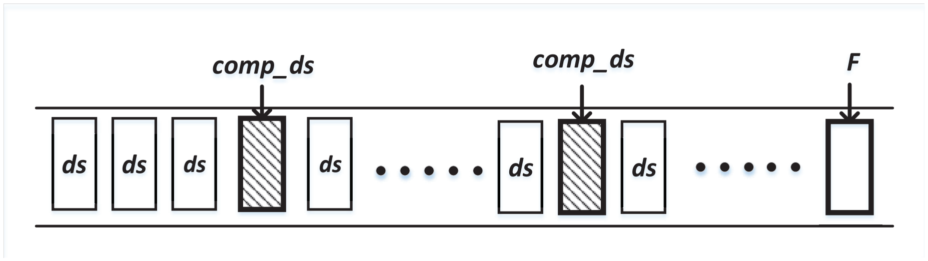

Figure 2.3 The Global List (GL).

The second phase of DCS is to schedule the formed groups for composition.

Moreover, two existing algorithms for performance comparison are adopted, i.e., greedy composition, and periodic time interval-based grouping, which are briefly introduced as well.

2.4.1 The Global List (GL)

Prior to providing the details of algorithm design, we design a data structure, referred to as Global List (GL), which is an important component in our solution.

As illustrated in Figure 2.3, the Global List (GL) is a list-based data structure, which is used to hold the datasets and maintain their order based on the time of their availability. GL starts with a pre-defined number of original datasets, and may change dynamically over time, as composed datasets are removed from GL and new datasets, i.e., either intermediate results produced by the composition process or original datasets arriving late, are inserted into GL. However, towards the end of the composition process, the number of datasets that need to be composed would decline until producing the last dataset, i.e., the final output F.

Based on this data structure, we design two partitioning algorithms to partition the datasets into groups, each of which is assigned to a computer node for composition. These partitioning algorithms build a Composition Tree $CTree$ (Subsection 2.4.4) to calculate the TCT as shown in Algorithm 3 and are followed by the group scheduling algorithm (Subsection 2.4.5) to determine the composition order.

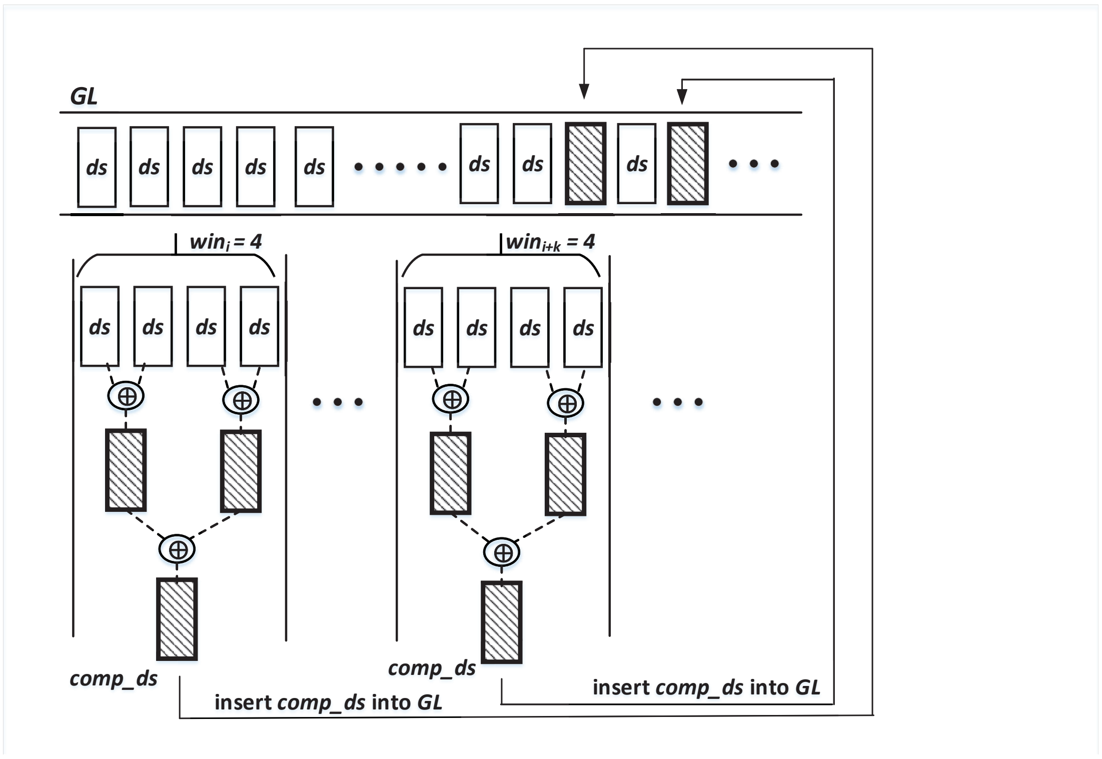

Figure 2.4 FDCS.

2.4.2 Fixed-window Distributed Composition Scheme (FDCS)

We design a Fixed-window Distributed Composition Scheme (FDCS), whose pseudocode is provided in Algorithm 1 with an illustration of its process in Figure 2.4.

In this algorithm, we prefix a window size x, which defines the number of datasets in a group used for composition. We first check if there are x or more datasets available on the GL. If yes, we create a group of x datasets from the GL and call a scheduling function for composition; otherwise, we wait until enough datasets have arrived to form a group for composition.

2.4.3 Dynamic-window Distributed Composition Scheme with Delay (DDCS-D)

We follow the concept of delay scheduling from [70], which aims to improve the performance of Hadoop system using a default fair scheduler, and design a Dynamic-

Algorithm 1: Fixed-window Distributed Composition Scheme (FDCS)

Input: a number of datasets (ds_1, ds_2, ..., ds_n) that become available at different time points (t_{ds_1}, t_{ds_2}, ..., t_{ds_n}), which are stored on the global list GL, and distributed among different virtual machines (vm_1, vm_2, $\cdots$, vm_m)

Output: a group gr_i of datasets that are ready for scheduling

```
 1: Initialize $x$ to be the pre-fixed number of datasets in a group;
 2: if ($GL.size() >= x$) then
 3:    create a group $gr_i$;
 4:    while ($gr_i.size() <= x$) do
 5:       $gr_i.add(dataset)$;
 6:    $schedule(gr_i)$;
 7: else
 8:    wait till there are $x$ or more datasets on $GL$;
 9:    if (not the last dataset) then
10:       create a group $gr_i$;
11:       while ($gr_i.size() <= x$) do
12:          $gr_i.add(dataset)$;
13:       $schedule(gr_i)$;
14:    else
15:       return
```

window Distributed Composition Scheme with Delay (DDCS-D). Delay scheduling is originally introduced for cluster scheduling where fairness is relaxed in order to explore data locality.

DDCS-D adopts a dynamically changing window size. It starts with a window size set to be the smallest group size, i.e., 2 datasets for composition. Every time when a group of datasets are formed, it checks the size of the GL. If the GL size is larger than the current window size, it increases the window size by adding one additional dataset; otherwise, the new window size is the same as the number of available datasets on the GL.

Moreover, to make the composition process more adaptive, we introduce a *delay*, which defines an amount of time DDCS-D has to wait before checking the size of the GL. In this case, we allow more datasets to arrive and be added to the GL, which may yield a larger group with more datasets for composition. However, an excessively long waiting time would delay the entire composition process. We will conduct experiments to provide insights into choosing an appropriate value for the delay. Compared with FDCS, DDCS-D is more adaptive to the arriving pace of the datasets. The pseudocode of DDCS-D is provided in Algorithm 2 with an illustration of its process in Figure 2.5.

Algorithm 2: Dynamic-window Distributed Composition Scheme with Delay (DDCS-D)

Input: a number of datasets (ds_1, ds_2, ..., ds_n) that become available at different time points (t_{ds_1}, t_{ds_2}, ..., t_{ds_n}), which are stored on the global list GL and distributed among different virtual machines (vm_1, vm_2, $\cdots$, vm_m)

Output: a group gr_i of datasets that are ready for scheduling

```
1: Initialize $win\_size = 2$;
2: while ($true$) do
3:     if ($GL.size() > win\_size$) then
4:         $win\_size = win\_size + 1$;
5:     else
6:         $win\_size = GL.size()$;
7:     create a group $gr_i$;
8:     while ($gr_i.size() < win\_size$) do
9:         $gr_i$.add($ds$);
10:    $schedule(gr_i)$;
11:    wait for a $delay$ amount of time;
12: return
```

2.4.4 Composition Tree $CTree$

As shown in Figure 2.6, a $CTree$, which is a binary tree, is constructed from multiple leaves to the root as the composition process proceeds. Each node with two incoming edges and one outgoing edge represents a dataset, and each edge represents a composition operation associated with a weight reflecting its cost. The root of the composition tree $CTree$ is the final output F that is generated from the last composition operation, and the TCT is calculated as the sum of the time cost components along the critical (longest) path (CP) of the tree, which may or may not be balanced.

Once a composition operation takes place, the tree is constructed or updated. In Algorithm 4, after each composition operation, the function $update\_CTree()$ creates a branch in the tree that contains: two (parents) nodes, two edges and one child node. Once the entire composition process is completed, the tree is fully constructed. Hence, the TCT can be computed by traversing the CP in the $CTree$, as shown in Algorithm 3.

```
Algorithm 3: Calculate the TCT using $CTree$
Input: Composition Tree $CTree$
Output: the total composition time $TCT$
1: initialize $TCT = 0$;
2: find the critical path $CP$ in $CTree$;
3: for all (edge $e$ along the $CP$) do
4:     $current\_cost = cost\_e$;
5:     $TCT = TCT + current\_cost$;
6: return $TCT$;
```

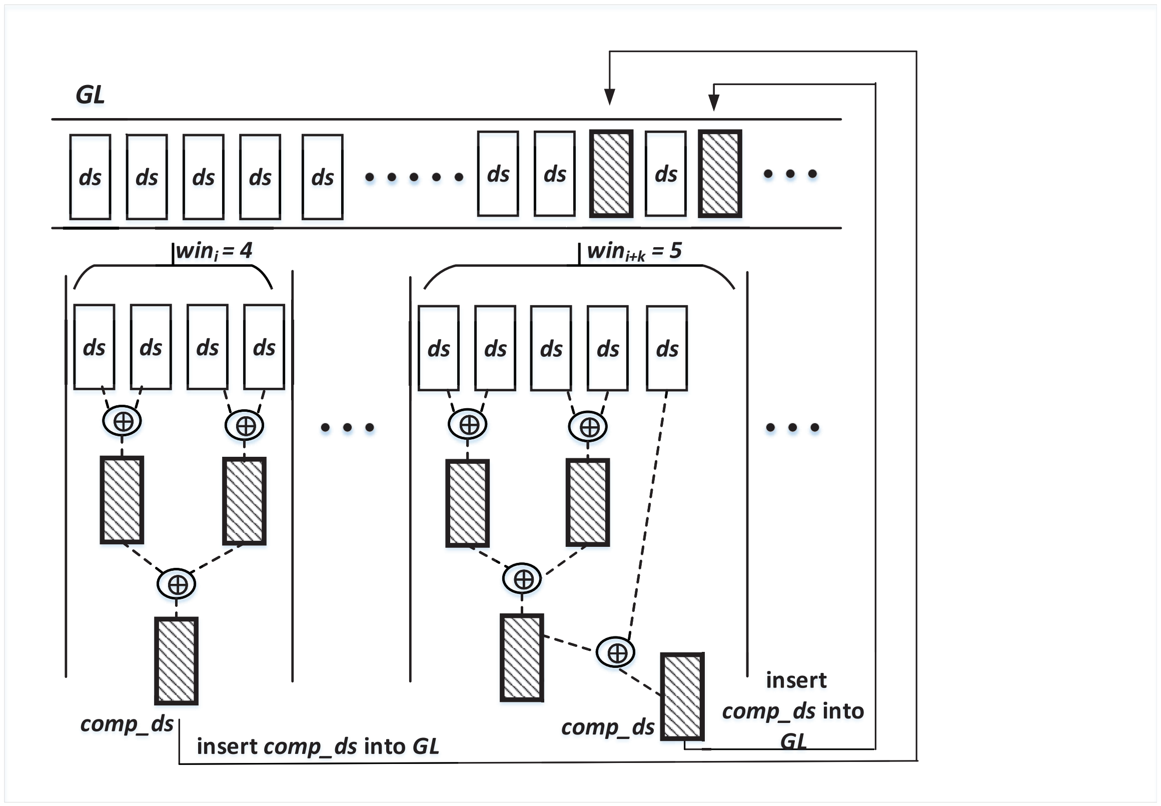

Figure 2.5 DDCS-D.

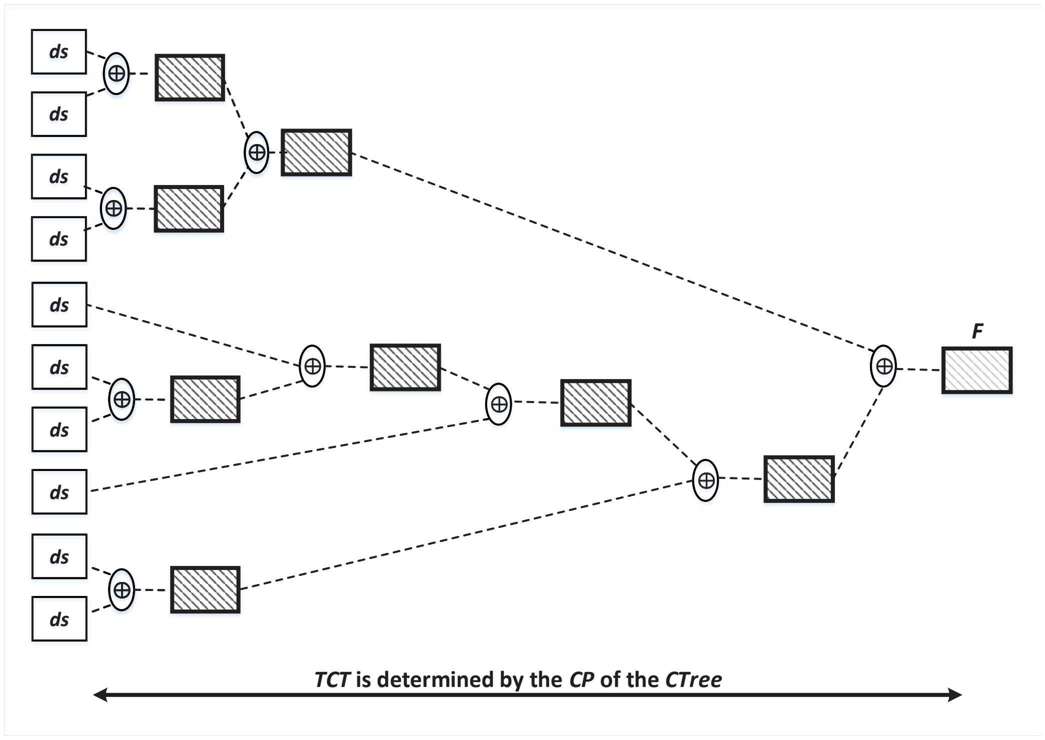

Figure 2.6 Illustration of the composition tree ($CTree$).

2.4.5 Group Scheduling for Composition

Once a group gr_i of datasets become available for composition, the scheduling algorithm begins by deciding the target node among the ones with available datasets, which performs all composition operations. We employ a *locality-based scheduling* approach to minimize transfer time cost by minimizing transfer overhead. Data locality is the placement of computation on the same node as its input data [15]. Given a group gr of datasets, we follow two rules to choose the target node as follows:

1. *Primary rule of majority vote*: We choose the target node to be the one that holds the majority of the datasets for composition.

2. *Secondary rule of minimum transfer cost*: We choose the target node to be the one with the minimum cost of transfer time based on the available network resources on the cluster, and transfer any needed non-local datasets to it.

Moreover, after performing the composition, we update $CTree$ and GL accordingly, as detailed in Algorithm 4.

Algorithm 4: Group Scheduling for Composition

Input: a group gr_i of datasets

Output: *true* if the composition is completed successfully; $false$, otherwise

```
1: while ($true$) do
2:     $gr_i.decide()$;
3:     $pairs[] = gr_i.pair\_up\_ds()$; group;
4:     for all ($pair\ p_i \in pairs[]$) do
5:         $comp\_ds = compose(p_i)$;
6:         $update\_CTree(p_i, comp\_ds)$;
7:         $GL\_insert(comp\_ds)$;
```

2.4.6 Algorithms for Comparison

Greedy Composition A greedy approach has been frequently used for dynamic information composition. It is a simple heuristic based on a greedy strategy. At any time, if there are two or more datasets on GL, it selects two datasets and performs composition regardless of data locality, as detailed in Algorithm 5 and illustrated in Figure 2.7.

Algorithm 5: Greedy Composition

Input: a number of datasets (ds_1, ds_2, ..., ds_n) that become available at different time points (t_{ds_1}, t_{ds_2}, ..., t_{ds_n}), which are stored on the global list GL and distributed among different virtual machines (vm_1, vm_2, $\cdots$, vm_m)

Output: GL after performing all the composition

```
1: set $start\_time$ ;
2: while ($true$) do
3:     if ($GL.size()$ $>= 2$) then
4:         opr1 = GL.getOpr1();
5:         opr2 = GL.getOpr2();
6:         $comp\_ds$ = compose(opr1, opr2);
7:         GL.insert($comp\_ds$);
8:     else
9:         wait till there are 2 or more datasets on GL;
10: return $GL$;
```

For the greedy composition process in Algorithm 5, we calculate the TCT as the available time of the final output or last dataset F, as shown in Algorithm 6.

Periodic Time Interval-based Grouping This is another simple heuristic, which repeatedly collects datasets to form a group in every time period of a certain length,

Algorithm 6: Determine the TCT for greedy composition

Input: the Global List GL

Output: the total composition time TCT

```
1: $F$ = GL.retrieveLast();
2: $end\_time$ = $F$.getAvalTime();
3: $TCT$ = $end\_time$ - $start\_time$
4: **return** $TCT$;
```

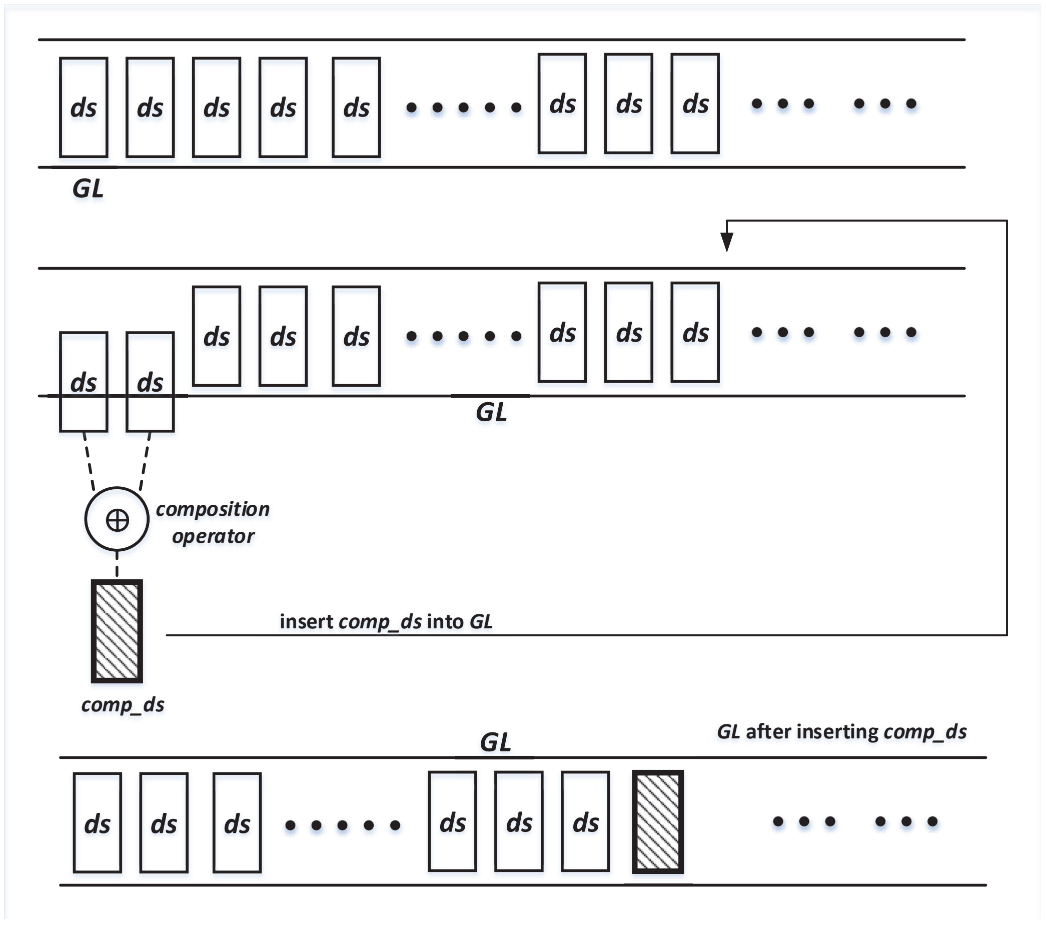

Figure 2.7 Greedy composition.

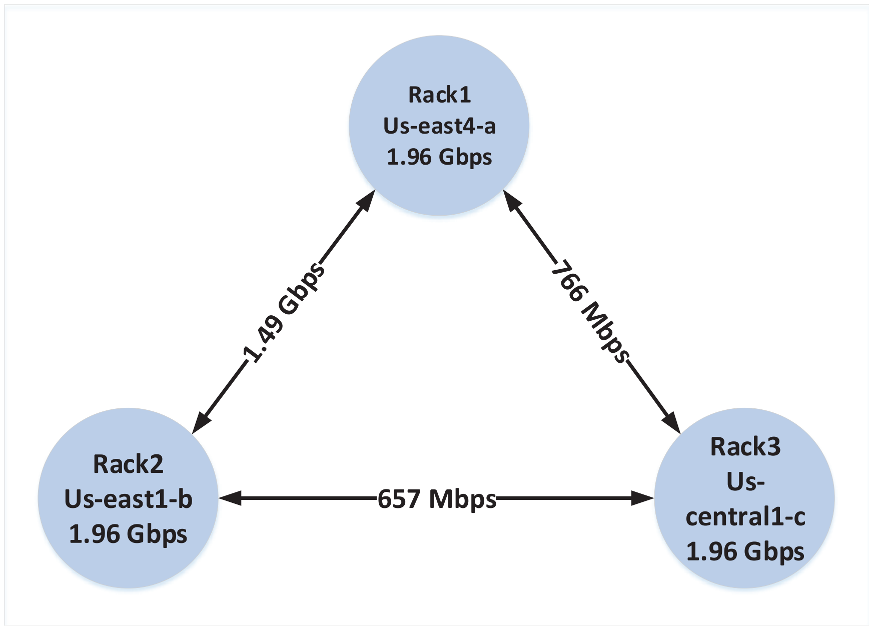

Figure 2.8 Intra-rack and inter-rack bandwidths on the cluster testbed.

e.g., 10 seconds. Both Greedy and Periodic algorithms follow the scheduling procedure as described in Subsection 2.4.5.

2.5 Performance Evaluation

2.5.1 Experimental Settings

We implement our algorithms in Python and use Google cloud to build a Hadoop cluster of 3 racks, each of which has 3 computer nodes. These racks are located in different geographical zones. As shown in Figure 2.8, the bandwidth on the same rack (intra-rack) is 1.96 Gbps, while the bandwidth between different racks (inter-rack) may differ.

We consider three degrees of time complexity for the composition operator: $O(n)$, $O(n \log n)$ and $O(n^2)$. To evaluate the performance, we consider different problem sizes in terms of the number of datasets from small to large scales in a range of [100,1000]. We set the size of each original dataset to be 64MB, the same as the default data block size in Hadoop 1.

We implement two proposed algorithms, i.e., FDCS and DDCS-D, and two algorithms in comparison, i.e., Greedy and Periodic with a fixed period of 10 seconds. Each composition experiment is repeated three times and the average performance is calculated and plotted for comparison. In each performance figure, the x-axis represents the number of datasets in the range of [100, 1000] and the y-axis represents the corresponding average TCT.

The source code of the algorithm implementation is made publicly available at https://github.com/Big-Data-World/Composition-in-Hadoop.git.

2.5.2 Experimental Results

For FDCS, we test different window sizes and select the one that yields the best performance for each composition operator of a different complexity.

Composition Operator of Complexity $O(n)$ The composition time T_C for an operator of complexity $O(n)$ is relatively small. We use a composition operator of this complexity to run four different algorithms, i.e., 1) FDCS, 2) DDCS-D, 3) Greedy, and 4) Periodic. We observe that FDCS performs better than DDCS-D, which is explained as follows: The window size of DDCS-D increases as the datasets arrive at a fast pace at the GL, and the time for composing the datasets in a given group increases accordingly, which yields a latency in the arrival time of the dataset at the GL. Therefore, the window size shrinks and the time for composing the datasets in a given group decreases, which makes the newly composed datasets be inserted into the GL faster. FDCS performs better because it provides more stable processing, while there is an overhead for DDCS-D due to the variation of the window size and the delay. Table 2.3 and Figure 2.9 show the performance measurements of different algorithms processing various numbers of datasets.

Composition Operator of Complexity $O(n \log n)$ The performance measurements of the algorithms using a composition operator of complexity $O(n \log n)$ are qualitatively similar to those produced by the algorithms using the composition operator of complexity $O(n)$. Table 2.4 and Figure 2.10 show the results of different algorithms processing various numbers of datasets.

Composition Operator of Complexity $O(n^2)$ In this case, DDCS-D starts outperforming FDCS, which is explained as follows: A composition operator of complexity $O(n^2)$ incurs a large composition time T_C, which implies that the composed datasets are inserted into the GL at a slower pace. FDCS has to wait until there is a sufficient number of datasets to form a group (as defined in the algorithm), thus causing a latency. On the other hand, DDCS-D dynamically updates the window size to accommodate the arrival pace of the datasets.

With a composition operator of complexity $O(n^2)$, it still causes some fluctuation in the window size but is not as frequent as in the cases of $O(n)$ and $O(n \log n)$. Therefore, DDCS-D cuts down the TCT more than FDCS. Table 2.5 and Figure 2.11 show the performance measurements of different algorithms in comparison for processing various numbers of datasets.

In all these experiments with different complexities, we observe that the Greedy algorithm performs the worst.

Algorithm Execution Dynamics

The Fluctuation of Window Size To explain the behavior of DDCS-D, we conduct an experiment to show the fluctuation of the window size over a period of time. Figure 2.14 plots the change of the window size with three degrees of complexity for a problem size of 300 datasets. As shown in Figure 2.14, DDCS-D with a composition operator of complexity $O(n)$ fluctuates the most, but exhibits a stable behavior with $O(n \log n)$. This is because the composition time T_C is relatively small and the arrival pace of the datasets to be inserted into the GL is high. Accordingly, the window size increases to accommodate more datasets, and the time to process a group increases, which slows down the arrival of more datasets. Hence, DDCS-D is adaptive by decreasing the window size. In the case of $O(n \log n)$, there is almost no fluctuation, and the windows size always tends to be the minimum, since datasets are inserted into the GL at a very slow pace as the composition time contributes more.

Optimization of the Window Size in FDCS To find the most efficient window size for FDCS, we conduct an experiment with 300 datasets. As illustrated in Table 2.6 and Figure 2.12, FDCS with a composition operator of complexity $O(n)$, $O(n \log n)$, and $O(n^2)$ yields the minimum TCT when the window size is 4, 2, and 3, respectively.

Optimization of the Delay in DDCS-D To find the most efficient delay for DDCS-D, we conduct an experiment with 300 datasets. Figure 2.13 and Table 2.7 show that DDCS-D with $O(n)$ achieves the minimum TCT when the delay is 1 second, DDCS-D with $O(n \log n)$ achieves the minimum TCT when the delay is 0 seconds (no delay at all), and DDCS-D with $O(n^2)$ achieves the minimum TCT when the delay is 0.4 seconds.

Analysis of the Results from DDCS-D With a composition operator of complexity $O(n)$, T_C is relatively small and hence the arrival pace of the newly composed datasets is high. Hence, the grouping in DDCS-D progresses quickly and it will eventually reach the minimum window size, which is 2. Therefore, we introduce a delay of 1 second to achieve the most efficient window size of 4, which yields the minimum TCT. For a composition operator of complexity $O(n \log n)$, the most efficient window size is 2, which is the minimum window size, and there is no delay introduced. In the case of an operator of complexity $O(n^2)$, T_C is higher than the other operators. Empirically, we observe that the most efficient window size is 3 with a delay of 0.4 seconds. The arrival pace of the newly composed datasets in the case of $O(n^2)$ is slower than its counterpart $O(n)$. Hence, we have a smaller window size and less delay with an operator of complexity $O(n^2)$.

2.6 Conclusion

We formulate a generic problem of Distributed Information Composition in Big Data Systems, referred to as DIC-BDS, which is proven to be NP-complete. Heuristic algorithms for DIC-BDS are designed that take into consideration the arrival dynamics of datasets, and their performance superiority over other existing methods for composition operators of various time complexities are demonstrated through extensive experiments on a real cloud-based cluster. The proposed composition algorithms add another level of intelligence for big data analytics in existing big data computing systems.

It would be of future interest to investigate the problem with other distributed frameworks such as Spark and evaluate our algorithms with real-life big data workflows.

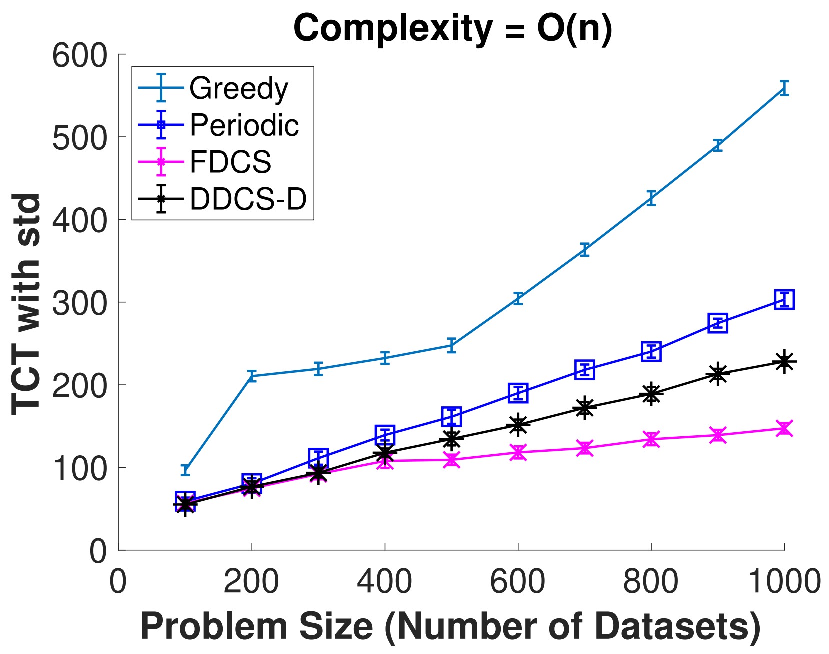

Figure 2.9 The average TCT (seconds) of different algorithms under different problem sizes with an operator of complexity $O(n)$.

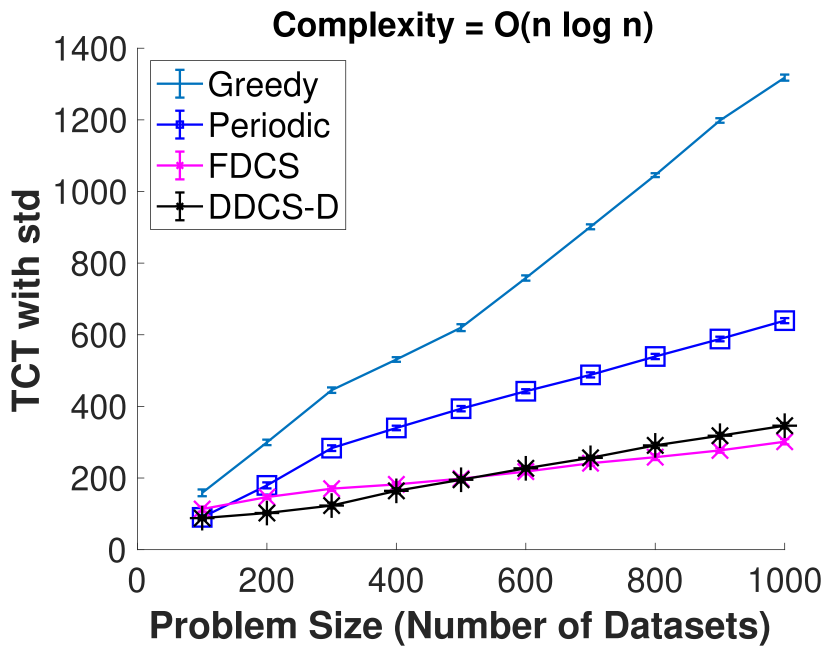

Figure 2.10 The average TCT (seconds) of different algorithms under different problem sizes with an operator of complexity $O(n \log n)$.

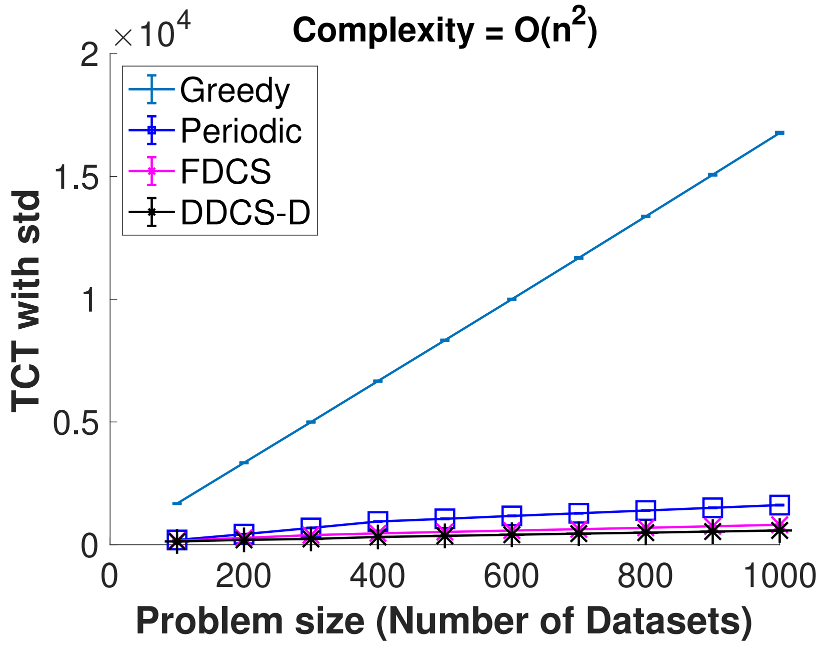

Figure 2.11 The average TCT (seconds) of different algorithms under different problem sizes with an operator of complexity $O(n^2)$.

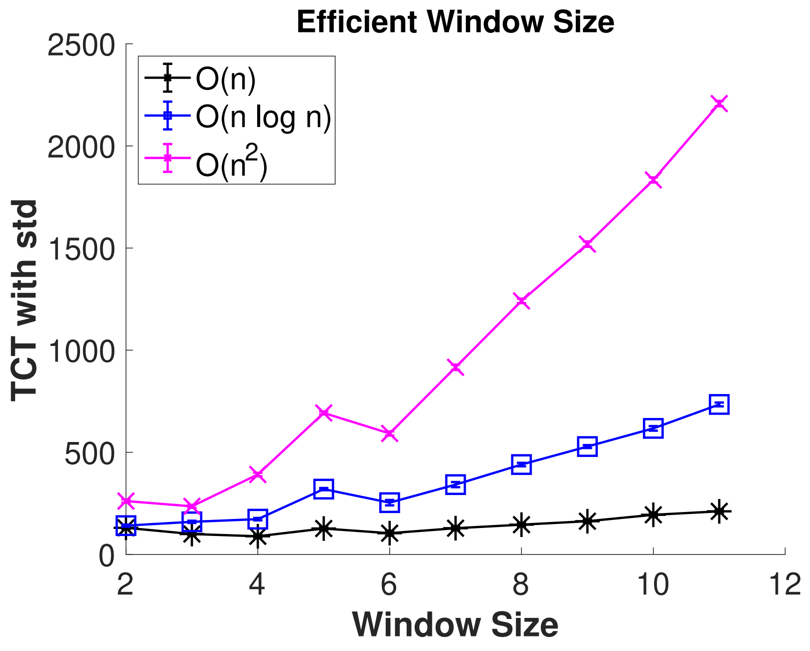

Figure 2.12 The average TCT of FDCS with composition operators of different complexities for processing 300 datasets with different window sizes.

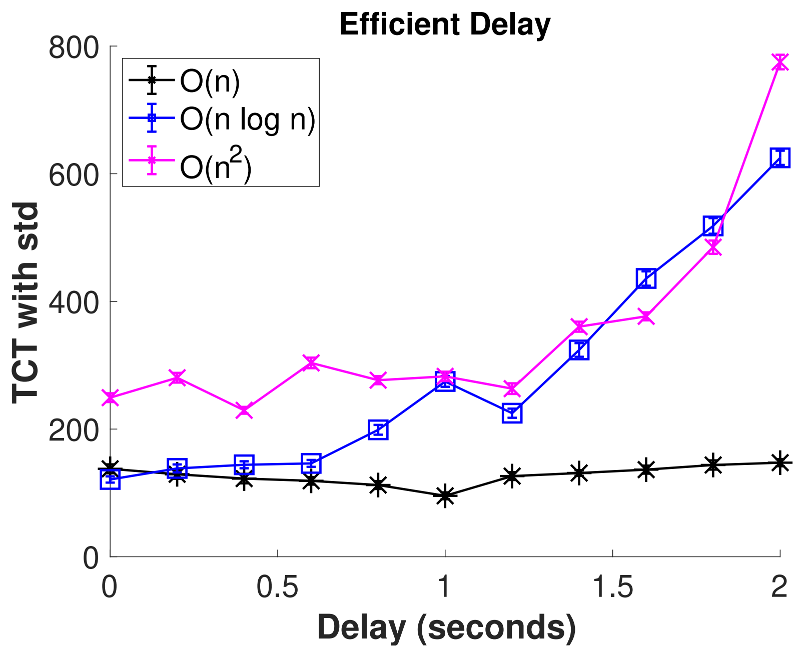

Figure 2.13 The average TCT of DDCS-D with composition operators of different complexities for processing 300 datasets with different delay time.

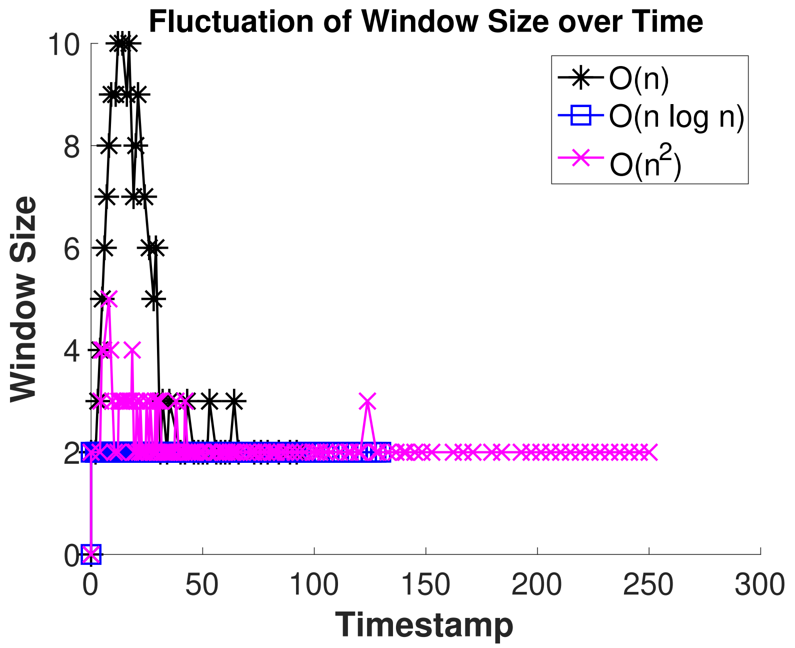

Figure 2.14 The average TCT of DDCS-D with composition operators of different complexities for processing 300 datasets with different delay time.

Table 2.3 The Average TCT (seconds) of Different Algorithms under Different Problem Sizes with an Operator of Complexity $O(n)$

# of ds	Greedy	StdDv	Periodic	StdDv	FDCS	StdDv	DDCS-D	StdDv
100	96.72	5.87	59.10	4.48	56.59	3.05	55.20	4.87
200	210.47	6.29	80.49	6.49	74.76	4.68	76.73	6.34
300	219.28	7.43	111.23	7.95	92.08	6.42	93.46	5.59
400	232.36	7.04	139.19	6.52	107.91	8.49	117.82	5.12
500	247.61	8.27	161.36	8.61	109.26	6.27	134.44	7.76
600	304.38	6.72	189.96	7.49	118.19	7.29	151.51	6.42
700	363.42	7.33	218.11	6.42	123.41	6.44	172.17	7.03
800	425.77	8.29	240.19	7.33	134.13	7.22	189.02	8.18
900	489.64	6.43	274.59	5.34	139.12	6.37	213.14	6.19
1000	558.92	8.33	303.10	8.19	147.41	6.31	228.15	5.04

Table 2.4 The Average TCT (seconds) of Different Algorithms under Different Problem Sizes with an Operator of Complexity $O(n \log n)$

# of ds	Greedy	StdDv	Periodic	StdDv	FDCS	StdDv	DDCS-D	StdDv
100	158.62	9.45	90.13	6.78	113.32	7.21	88.01	5.65
200	299.48	7.64	179.45	8.43	146.80	5.44	102.31	4.36
300	445.30	7.63	283.41	7.92	170.23	6.55	123.12	8.32
400	530.94	6.44	339.76	6.32	182.18	6.21	164.17	7.31
500	619.84	9.62	393.60	7.43	198.38	8.33	195.12	7.39
600	758.40	7.38	442.27	5.66	217.90	7.48	227.13	6.46
700	901.42	6.87	487.95	7.12	241.30	6.77	256.36	5.66
800	1045.62	5.42	539.30	7.33	258.19	8.01	291.10	7.21
900	1198.28	6.33	588.13	6.45	277.10	6.42	318.09	8.22
1000	1317.91	8.52	639.39	7.11	301.32	7.41	346.10	7.81

Table 2.5 The Average *TCT* (seconds) of Different Algorithms under Different Problem Sizes with an Operator of Complexity $O(n^2)$

# of *ds*	Greedy	StdDv	Periodic	StdDv	FDCS	StdDv	DDCS-D	StdDv
100	1676.83	15.36	193.24	4.38	159.57	3.64	133.78	5.01
200	3337.49	18.28	430.74	5.99	279.74	6.49	197.80	6.44
300	4996.32	18.97	683.61	8.43	391.51	7.84	236.06	8.54
400	6665.37	19.43	945.84	9.84	462.65	8.86	314.91	7.93
500	8329.39	21.87	1055.10	14.37	519.26	10.48	360.83	11.29
600	9999.06	20.56	1173.19	12.44	577.90	8.32	406.98	10.48
700	11682.34	26.44	1281.73	10.82	632.42	9.31	452.20	8.29
800	13373.28	25.64	1393.10	9.41	689.51	8.51	491.11	6.75
900	15071.32	29.39	1503.91	11.19	751.14	9.16	536.51	8.42
1000	16774.08	39.85	1614.38	12.85	811.18	8.17	581.13	7.71

Table 2.6 The Average TCT of FDCS with Composition Operators of Different Complexities for Processing 300 Datasets with Different Window Sizes

window_size	$O(n)$	StdDv	$O(n \log n)$	StdDv	$O(n^2)$	StdDv
2	130.42	6.29	141.17	5.26	261.41	7.88
3	100.05	6.89	159.77	5.49	235.73	7.31
4	89.09	7.76	172.81	5.93	391.52	8.42
5	127.01	8.38	319.92	5.44	692.83	7.19
6	104.09	12.31	253.61	13.28	592.40	10.29
7	128.52	9.07	341.94	12.62	916.40	13.25
8	145.83	7.67	440.14	8.54	1241.30	11.18
9	162.76	6.32	528.26	7.16	1519.19	13.28
10	194.19	11.31	617.22	11.28	1833.46	12.43
11	211.17	9.11	734.40	9.32	2207.11	13.18

Table 2.7 The Average *TCT* of DDCS-D with Composition Operators of Different Complexities for Processing 300 Datasets with Different Delay Time

delay (seconds)	$O(n)$	StdDv	$O(n \log n)$	StdDv	$O(n^2)$	StdDv
0	137.75	6.95	121.02	4.87	249.19	6.83
0.2	129.19	7.13	138.33	6.26	280.42	7.61
0.4	122.43	7.42	144.05	5.47	229.40	5.32
0.6	118.92	5.88	146.20	5.39	303.64	8.29
0.8	112.33	6.18	198.83	7.65	276.57	6.24
1	95.44	5.38	274.52	8.27	282.57	7.31
1.2	126.34	6.11	224.85	7.16	263.33	8.33
1.4	131.14	4.47	323.95	10.72	360.42	7.87
1.6	136.47	6.32	435.89	11.26	376.62	6.21
1.8	143.73	7.22	518.23	12.21	484.93	10.44
2	147.37	5.57	624.87	11.17	774.96	11.38

CHAPTER 3

PERFORMANCE PREDICTION FOR SPARK-HBASE APPLICATIONS IN BIG DATA SYSTEMS

3.1 Introduction

Many large-scale applications in various business and scientific domains require a combination of parallel computing with distributed data storage and management for big data processing. In fact, it has become a widely adopted practice in industry to deploy big data systems such as Hadoop on top of NoSQL databases (such as HBase [7] and MongoDB [8]), and employ parallel computing frameworks (such as MapReduce [18] and Spark [72, 71]) to ensure timely data processing and efficient delivery of analysis results in support of decision making and business intelligence. Such systems typically consist of a stack of technology layers, which provide a large number of configurable parameters so that end users can request system resources as needed through parameter settings in advance.

In this work, we tackle a general problem of optimizing the execution performance of big data applications deployed on high-performance computing platforms. In specific, we focus our research on Spark applications that run over Hadoop/HDFS [61] as a data storage system and HBase as a data management system, which are referred to as Spark-HBase applications in this context. The execution of such applications in big data systems typically has a life circle of computing that spans through several stages across multiple technology layers: submitting the application to YARN as a resource manager, assigning executors in Spark for data processing, coordinating with *RegionServers* in HBase to determine a logical data block, and accessing the actual data block stored in Hadoop Distributed File System (HDFS) [4]. Each of these layers has a large set of parameters for configuration, and it is crucial for any given Spark-HBase application to decide a subset of configurable parameters according to its computing needs and performance requirements, e.g., the number of executors and the number of cores for each executor in Spark as well as database operation API-related parameters in HBase. Deciding an appropriate setting of effective parameters is critical to understanding and optimizing application performance for end users, and to maximizing the utilization of system resources for infrastructure providers.

However, it is challenging for end users, who are primarily domain experts, to decide a satisfactory configuration for executing Spark-HBase applications in such complex computing systems. The execution complexity of Spark-HBase workflows compounded by system dynamics makes it a daunting task to select and configure a right set of parameters across different layers of the technology stack. In fact, in most of the existing big data systems, the parameters are typically set with default values, which, unfortunately, do not always lead to the best performance. These default values, once set, are oftentimes used for all applications of disparate types as end users generally do not have enough knowledge in computing to modify the system configuration. Even with the aid of certain knowledge, this problem is still largely unexplored and unresolved.

In this work, we investigate the problem of modeling and predicting the performance of Spark-HBase applications in big data systems by strategically selecting a subset of hyper parameters and setting their values using machine learning. Our goal is to achieve an accurate prediction of execution time using a performance-influence model that takes influential parameters as input features. Towards this goal, we start by conducting a large number of experiments to run various Spark-HBase applications with different parameter settings and collect their corresponding performance measurements. Such measurement data conveys informative knowledge and provides insights into the performance pattern and execution behavior of these applications under different configurations in big data systems, which facilitate performance optimization and configuration recommendation for such applications. We then conduct an in-depth comparison-based analytical study to investigate the effects of these parameters on application performance. Based on the data collected from the exploratory analysis and aided by domain knowledge, we design a class of regression-based prediction models to estimate the execution time of Spark-HBase applications, and illustrate the accuracy of such models using different performance metrics.

As a summary, we make the following contributions in this work:

- **Modeling of interactions between various components in Spark-HBase applications**. We construct rigorous cost models for various components involved in the execution of Spark-HBase applications to quantify the execution time of such applications, and explain the interactions between them.

- **Exploratory analysis.** We run a large number of experiments and conduct an in-depth analytical study to explore and investigate the effects of parameter selection and setting on the performance of Spark-HBase applications in terms of execution time.

- **Performance prediction using machine learning**. We utilize the data produced by the exploratory analysis to train a class of regression-based models to predict the execution time of Spark-HBase applications.

The rest of the work is organized as follows: Section 3.2 conducts a survey of related work in performance modeling and parameter setting. Section 3.3 constructs the cost models and defines the problem under study. We conduct an exploratory analysis of Spark-HBase application performance in Section 3.4, and design a class of regression-based models for performance prediction in Section 3.5. Section 3.6 presents the experimental results for performance evaluation. Section 3.7 concludes our work and sketches a plan of future research.

3.2 Related Work

In this section, we conduct a survey of related work on performance modeling and prediction in the context of: various computational jobs and applications, and scientific workflows. Specifically, the survey discusses different aspects of prediction models, characterization and profiling techniques, and performance parameters and metrics. Also, we conducted a brief survey on the literature of configuring parameters for both Spark and HBase frameworks.

3.2.1 Performance Modeling, Estimation and Prediction for Serial and Parallel Application

Modeling and prediction of applications performance has been extensively investigated in the literature where many studies presented models to estimate the different metrics of performance such as execution time, CPU utilization, memory usage and I/O cost [60, 34]. The motivation behind these studies is to: improve the performance, optimize resources allocation and detect performance issues or flows. In [63] Wu *et al.* presented a statistical model based on logistic regression to estimate the execution time of a serial job at a given point of time in scientific workflows; however, they did not consider parallel jobs in their proposed model. Others presented models for parallel applications [28, 35]. In [28], the authors constructed a prediction model based on neural networks to estimate the execution time for a parallel application called SMG2000, and they used the hardware and software attributes as model parameters. In [35] Lee *et al.* addressed the challenges of analytical performance modelling for parallel applications by applying statistical techniques to examine the parameters space. Additionally, they constructed predictive models based on piecewise polynomial regression and artificial neural networks. For big data applications, the authors of [36, 54, 32, 60] proposed different models to estimate and predict performance for different big data parallel computing applications. Song *et al.* [54] proposed a simplified prediction framework for Hadoop jobs. It was based on dynamically analyzing Hadoop jobs accompanied with locally weighted regression methods for performance prediction. In [60], Wang *et al.* applied analytical and simulation-driven models to leverage the multi-stage execution structure of Spark jobs and predict the performance of these jobs in terms of: execution time, memory usage and I/O cost. The authors of [52] propose different models to predict the execution time for several Spark applications deployed on hadoop cluster where their models are based on analytical and off-the-shelf machines learning techniques. Moreover, in [33], the authors present an approach to estimate the response of Spark streaming applications. However, their model is based on Palladio Component Model (PCM), and it focuses on the modeling and prediction the Spark performance for streaming; whereas our work focuses on modeling Spark applications for batch processing. Moreover, in their paper [34] they extend PCM to predict the response time and CPU utilization for Spark batch applications . Several works have been conducted to investigate and improve the performance of Spark applications that are coupled with HBase [50, 26, 49]. In [26], the authors conduct comparison experiments to show the difference in performance between MapReduce and Spark when executed on HBase and HDFS frameworks. Qin and Niu *et al.* [49] design AISHS that manages AIS data by storing it uniformly among the cluster nodes, and optimize the parallel query method to avoid data shuffling between RDD partitions and HBase regions. Furthermore, machine learning algorithms and techniques have been widely adopted in different disciplines in order to improve performance. In [40] Mao *et al.* presented Decima that is based on reinforcement learning and neural network to learn workload-specific scheduling algorithms for Spark clusters.

3.2.2 Scientific Workflows Modeling and Performance Estimation

The authors of [31] examine six different type of scientific workflows from different fields such as astronomy and bioinformatics. They categorized these workflows based on the consumption of I/O, memory usage and CPU utilization. The authors assume that designing general use workflow systems should not be determined based on the attributes of one single workflow. They develop two profiling tools: ioprof to collect data about process I/O, and pprof to collect data about process runtimes and consumption of memory and CPU. Profiling workflows indicate that some parts of the workflow can be optimized in order to improve the performance. Moreover, scientific workflows consist of a number of computational components of similar or different complexities that require splitting the data sets into smaller ones for the purpose of supporting concurrent processing [19]. The efficiency and quality of scheduling algorithm is determined by the task execution time, data access and information about the resources; Tasks may access data that are stored in memory, locally on a disk, or on some external storage [17]. Also, execution time depends mainly on the complexity of the program [19] [63], and processing a task of workflow in parallel can speed up the execution significantly [17]. Many papers [44, 39, 48] have presented studies to improve the performance of workflows. Mainly, we find that there are two approaches to estimate and predict the execution time for workflows; the first one is based on scheduling models as in [44], and the other approach is based on machine learning algorithms and techniques as in [39, 48]. In [44] Nadeem *et al.* and others proposed to estimate the execution time of a workflow using radial basis function neural network. Maheshwari *et al.* presented a multi-site workflow scheduling technique to model the performance and predict the execution time of workflows across geographically dispersed resources [39].

3.2.3 Spark parameters configuration

Spark has over 150 configurable parameters [47, 6] Several research efforts [47, 57] have studies the impact of configuring and tuning Spark parameters on the applications performance. The authors of [47] study the impact of configuring and tuning Spark parameters on the applications performance. They propose a novel tuning methodology following an efficient trial-and-error approach. They target parameters belonging to the categories of: Shuffle Behavior, Compression and Serialization, and Memory Management. However, they do not consider tuning the parallelism degree nor parameters related to YARN or MESOS. Furthermore, the authors in [37] conducted an exploratory analysis study to examine the effect of different parameters on the performance of Spark workflows, and propose a feature selection method based on information theory to identify the most important parameters, and conduct experiments to evaluate the performance of the method in identifying the best parameter setting for workflow execution.

3.2.4 HBase parameters configuration

HBase allows the end users to set and configure up to 197 parameters [3, 12]. Several papers studied the importance of parameters tuning and configuration for HBase applications and the impact of that on the performance [12, 13, 65]. In [12], the authors show that the HBase default configuration can lead to poor performance and hence propose and develop HConfig as a semi-automated configuration manager to optimize HBase performance. Moreover, Bao and others *et al.* [13] present PCM as a policy-driven configuration management system to identify workload sensitive configuration parameters, and shows that PCM outperforms the default configuration and provide higher throughput. At last, Xiong *et al.* [65] propose ATH as a novel approach to auto-tune the configuration parameters for HBase applications that is based on an ensemble learning performance model. We surveyed the previous papers to assist in selecting the parameters for HBase configuration in our model and experiments.

3.3 Problem Formulation

Spark [72, 71] provides a distributed system for processing data in memory, while HBase [7, 59] is a data management system that provides efficient access to data stored in HDFS. In this research, we propose a predictive model for Spark-HBase applications executed over a cluster of HDFS nodes in Hadoop.

3.3.1 Cost Models

Figure 3.1 shows an illustration of the architecture of a big data analytics system that consists of several layers. we provide a description and construct a cost model for each layer as follows:

Model of Applications Spark is a fast and efficient computing framework for processing large amounts of data [72, 71, 60] based on a core concept of Resilient Distributed Datasets (RDDs) [71] by leveraging distributed memory and providing parallel data processing [60]. It is an open-source Apache project, which is mainly designed to provide distributed, in-memory computation in a fault-tolerant manner. Additionally, due to the nature of its design, Spark is more suitable to run iterative machine learning and graph processing algorithms. A Spark job is executed in multiple stages, and each stage has a number of parallel tasks [11] that perform distinct operations following a Master/Slave approach, more specifically, a Driver/Worker structure. In this framework, the input data is partitioned into multiple sets and processed in parallel, and each worker performs tasks on a corresponding set of data. A Spark job is executed through a Directed Acyclic Graph (DAG) of stages. The Spark scheduler typically adopts delay scheduling [70] to assign tasks to workers based on data locality. For data access, if a task has to process a partition that is in memory on a node, Spark directs it to that node. Moreover, a task processing a particular partition is sent to a specific location (e.g., an HDFS file) if the involved RDD requires that.

Model of Spark job execution time Upon submission to a cluster, a Spark application is divided into one or more jobs and each job is divided into a number

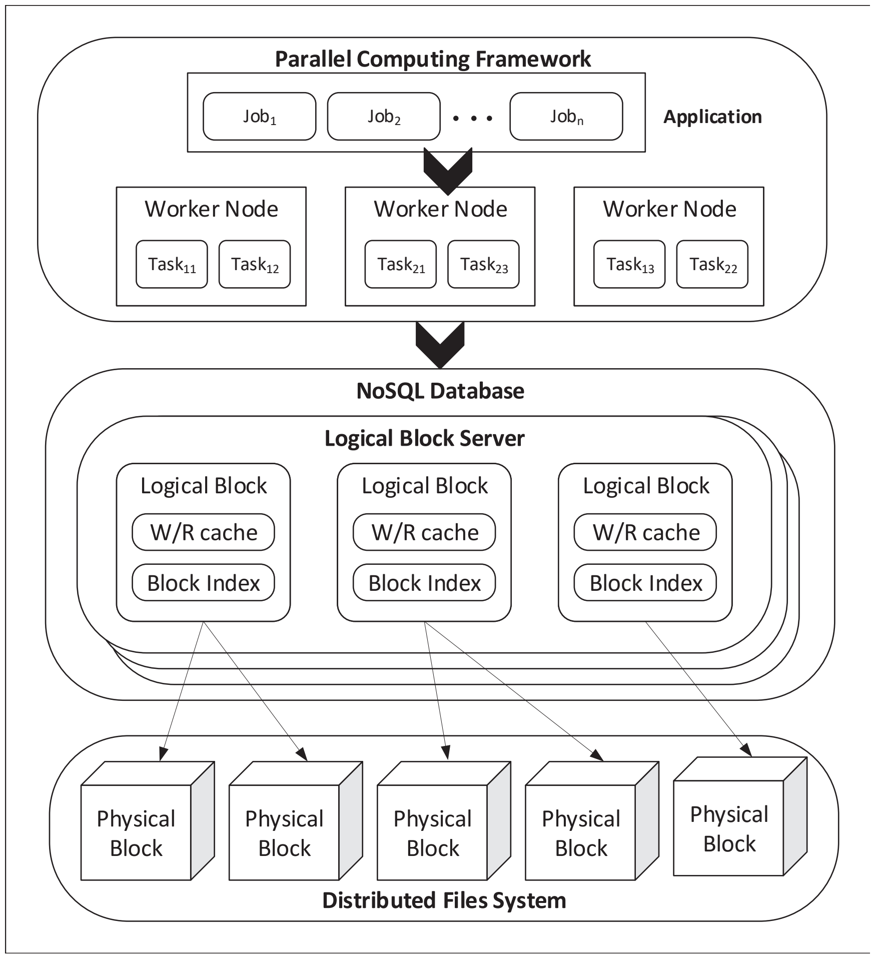

Figure 3.1 The architecture of a big data analytics system.

of stages based on the dependency relationship between RDDs, and each stage has several tasks. The stages can be executed sequentially or in parallel depending on the dependency between them. However, the tasks within the same stage are processed in parallel by Spark executors since there is no data dependency between them. The shuffle time is included within the stage time since the shuffle operations required for wide dependencies are the boundaries of the stages. A job and its corresponding stages are denoted as:

$$Job = \{Stage_i \mid 0 \leq i \leq M\}, \tag{3.1}$$

$$Stage = \{Tasks_{i,j} \mid 0 \leq j \leq N\}, \tag{3.2}$$

where M is the number of stages in a job and N is the number of tasks in a stage.

Furthermore, the Job Execution Time (JET) is the sum of the execution time of all stages that are executed sequentially in addition to the time spent on job startup and cleanup, modeled as:

$$T_{job} = \sum_{k=1}^{M} T_{stage_k}, \tag{3.3}$$

where the stage execution time is denoted as (assuming all tasks start at the same time):

$$T_{stage} = longest(T_{task_C}), \tag{3.4}$$

where T_{task_C} is the execution time of the longest task among all parallel tasks in the same stage. The task execution time is calculated as the sum of deserialization time, running time, and serialization time, modeled as [60]:

$$T_{task} = T_{deserialize} + T_{run} + T_{serialize}. \tag{3.5}$$

Generally, there are three main factors that contribute to JET: i) the execution time ET_{Spark} of the Spark application determined by the processing power and memory capacity, ii) the time $T_{I/O}$ consumed to access the data managed by HBase and stored in HDFS, and iii) the time T_{dt} needed to transfer the data over the network. As such, JET is a function of all these factors, denoted as:

$$JET = f(ET_{Spark}, T_{I/O}, T_{dt}). \tag{3.6}$$

Model of data management In most big data analytics systems, the underlying data management is provided by NoSQL databases such as HBase. HBase is an open-source Apache project that is inspired by Google's BigTable, and is a column-oriented, fault-tolerant NoSQL database used for real-time big data applications. HBase could be deployed on top of a distributed storage system, e.g., HDFS. Such deployment is widely adopted in industry including Facebook Messages, which is a classical application at Facebook handling a large number of messages regularly through HBase [7, 16, 12, 65, 59, 23].

As in traditional RDBMS, the data in HBase is managed in tables that contain multiple rows, each of which is referenced by a unique key. A row is made of columns, which are grouped into families. Data items are stored in cells, and each cell is identified by (row $\times$ column-family:column). HBase has a distributed architecture that consists of four main components: HMaster, ZooKeeper cluster, RegionServers (RSs), and HBaseClient (HTable). HBase stores data as indexed files in HDFS, and can host large tables with billions of rows and millions of columns. An HBase table is logically divided into regions, each of which contains a range of adjacent rows that are grouped together. A RegionServer may serve one or more regions, whereas a region is served by only one RegionServer. Each RegionServer has a daemon process, HRegionServer, which handles HRegions with a number of Stores, each of which keeps a column family. Furthermore, each Store contains a MemStore that holds the in-memory modification of the Store and StoreFiles that correspond to an HFile, which is the file format for HBase. Also, there is one HLog per RegionServer that logs the changes made to Stores. A region is split if the size of the StoreFile exceeds a threshold [49].

In big data analytics systems, the parallel processing framework with distributed data storage and management deals with logical blocks *(LBs)* instead of physical blocks *(PBs)* as shown in Figure 3.1. The size of an LB is much smaller than that of a PB since an LB only stores the indices and the read/write caches of the associated PBs. Parallel tasks directly process table-based data schemes. However, physical blocks are read and written indirectly through their indices stored in related logical blocks that are dispersed across the cluster. A logical block collects logically adjacent data sets in distributed storage and can be represented as a region in HBase [11]. We model HBase as a collection of k tables $T = T_1$, T_2, $\ldots$, T_k, and use $T_i = LB_{i,1}$, $LB_{i,2}$, $\ldots$, LB_{i,p_i} to denote p_i logical blocks for any T_i that belongs to T.

Model of data storage Hadoop Distributed File System (HDFS) is a framework to store very large files in a reliable and fault-tolerant manner, and is mainly designed to stream data at high bandwidth to user applications [61, 51]. Data is stored as blocks in HDFS, whose size typically ranges from 64 to 256 MB, and HDFS provides write-once-read-many semantics on data. Accessing data concurrently in HDFS is expensive; hence, integrating a management framework such as HBase provides more efficient data access in general.

Interaction between Spark and HBase over HDFS Figure 3.2 illustrates the dynamic interactions between different components of the framework in executing Spark-HBase applications. A Spark job is executed through a Directed Acyclic Graph (DAG) of stages, and there are several interaction points between Spark and HBase [5]. In this work, we focus on some basic Spark interactions where HBase can connect at any point to Spark DAG. HBaseContext is the core of all Spark and HBase integration since it pushes the HBase configurations into the Spark executors.

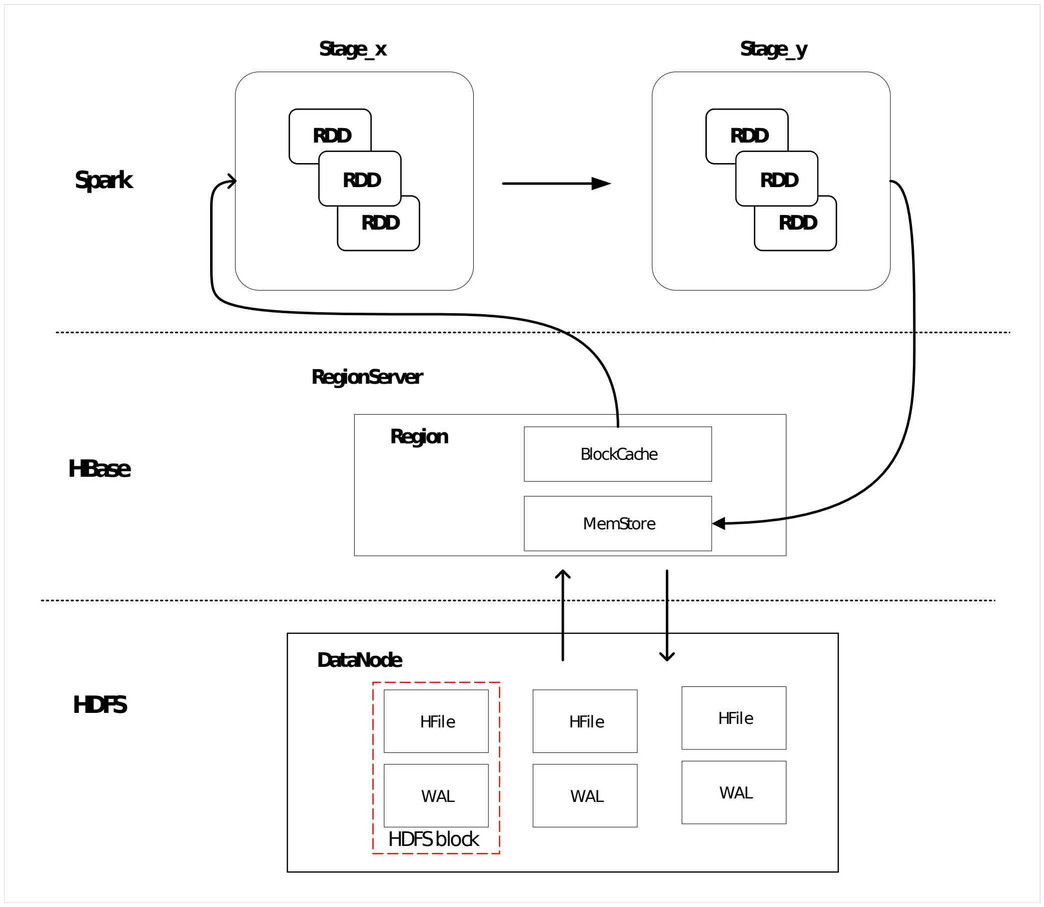

Figure 3.2 Illustration of the dynamic interactions between different components in big data systems.

Additionally, as an HDFS client, HBase stores its HFiles and Write Ahead Logs (AWL) in HDFS.

3.3.2 Problem Statement

Based on the above descriptions, the performance (specifically, execution time) y of an Spark-HBase application can be modeled as a function f of a vector $\mathbf{x}$ with features x representing various parameters of the computing and management framework in big data systems, i.e., $y = f(\mathbf{x})$. However, it is arduous to find an analytical form of f that is generally intractable due to the large number of parameters involved and the complexity of the execution process across multiple layers of big data systems.

We provide a formal definition of our problem as follows: Given a historical performance measurement dataset

$$\mathcal{D} = \{(\mathbf{x_1}, \mathbf{y_1}), (\mathbf{x_2}, \mathbf{y_2}), \ldots, (\mathbf{x_n}, \mathbf{y_n})\},$$

where $\mathbf{x}_i$ $(i = 1, 2, \ldots, n)$ is a set of specific values for the feature vector $\mathbf{x}$ that result in the corresponding performance y_i, our goal is to use a regression-based model to estimate the execution time based on the features of $\mathbf{x}$ such that $f(x_i)$ is close enough to the ground truth y_i for all training instances in $\mathcal{D}$ and could be used to predict y_i with high accuracy for any given $\mathbf{x}_i$ in the future.

In this context, we use a set of parameters to assemble the feature vector $\mathbf{x}$. These parameters are gathered across various stages during the life circle of an application execution process, including: i) application submission such as input data size, etc.; ii) Spark scheduling such as the number of executors, executor cores, executor memory, etc.; and iii) HBase API-related parameter settings such as scanner cache size, etc. However, it is not straightforward to determine which parameters to consider without further analysis.

Hence, we conduct a comprehensive exploratory analysis to understand and explore the effect of selected parameters on the application's execution time, and develop a regression-based predictor using these parameters as features.

3.4 Exploratory Analysis

We first conduct an empirical study of the effect of different parameters on Spark-HBase applications in big data systems by repeatedly running Spark-HBase applications. We run an application to count the number of records in HBase based on a specific criterion. The application is written in Scala and executed as Spark jobs on a cluster of seven virtual machine (VM) instances (one master node and six slave nodes) provisioned on two physical servers, each of which is equipped with eight virtual cores and 24GB of memory. By default, each slave node provisions one executor with one virtual core and 1GB of virtual memory. We run our experiments using four tables of different sizes as detailed in Table 3.1. We use the data of (NYC yellow taxi trips)[1] to build our tables, each consisting of four column families with twenty-one columns.

3.4.1 Spark Parameters:

We run a set of experiments to explore the effect of the following parameters in Spark:

Number of executors Figure 3.4 shows the average execution time of running experiments with different numbers of Spark executors per worker in a range of (2-5) executors on all tables. The results show that the execution time decreases with more executors processing the tasks especially for table-3, which is the largest among all tables in our experiments. This observation is justified as follows: having more rows retrieved from the application indicates that more data is being processed by RDDs, and hence more executors process these RDDs faster in parallel.

Executor Core Count Generally, the computing power of an executor is determined by the number of cores. More cores would be able to run more tasks in parallel, which speeds up the execution of an application with heavy read/scan. Figure 3.3 shows that as the core count increases, the execution time decreases. Although the exact execution dynamics and time for reading data from tables of different sizes vary, the performance pattern is consistent qualitatively.

[1]https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page. Retrieved on September 2020

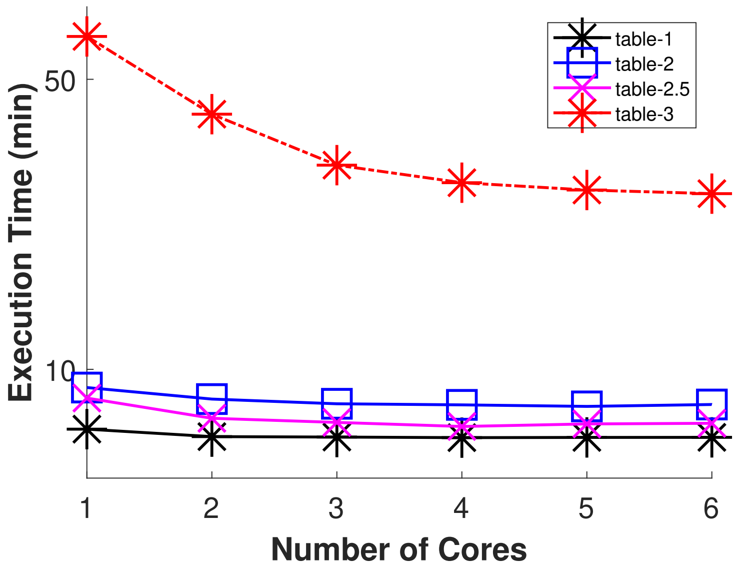

Figure 3.3 Illustration of the effect of number of cores on the execution time for application (count) on the tables.

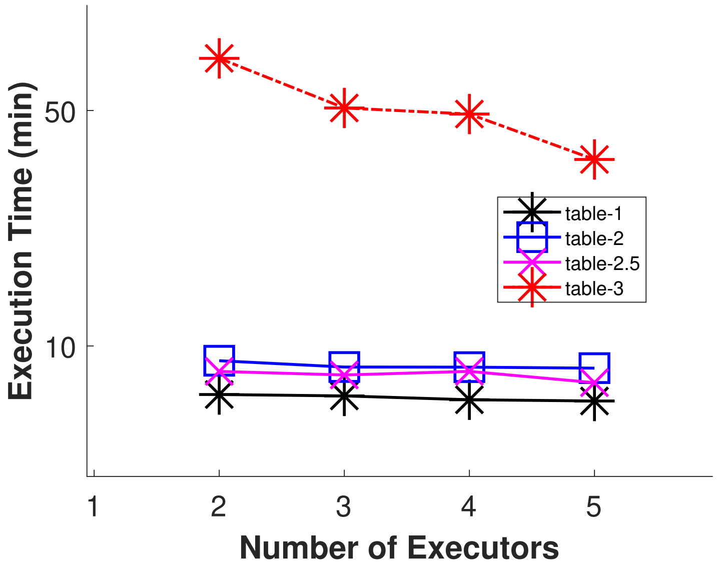

Figure 3.4 Illustration of the effect of number of executors on the execution time for all tables.

3.4.2 HBase Parameters:

The goal of the applications we use in our experiments is to read data from HFiles, which are stored in HDFS, using *scan* API. Therefore, in our experimental setting, we focus on examining the effect of tuning parameters that are related to scan and read operations, and we explore the impact of one parameter related to *BlockCache*:

BlockCache tuning BlockCache is designated to hold the indices to the data in HFiles. Disabling BlockCache may lead to a longer execution time since the data will be read directly from HDFS without caching it in-memory. The parameter *hfile.block.cache.size* specifies the percentage of heap to allocate to StoreFile block cache with a default value of 40%. Also, default configurations require that both *BlockCache* (used for reading) and *MemStore* (used for writing) do not exceed the threshold of 80% for successful cluster operation. Hence, in our experiments, we focus on *hfile.block.cache.size* with percentages of (40%, 50%, 60% and 70%). The experimental results show various performances with different tables used in terms of size and region count. More memory does not always yield better performance and we witness this clearly with tables (2, 3, and 2.5), where assigning 70% to *BlockCache* and 2G to heapsize exhibits the longest execution time due to garbage collection. However, both table-2 and table-2.5 provide better performance with 40% and 60% of *BlockCache*, respectively, and the gap between the first instance and the rest of instances in a set of experiments is the highest.

We would like to emphasize that the effects of these parameters are complex, which calls for the use of machine learning algorithms for performance modeling and prediction.

Table 3.1 Properties of Tables Used in the Experiments

Name	**Data Size (GB)**	**rows**	**retrieved rows**	**regions**
table-1	2.5	13,990,176	331,160	10
table-2	5	27,814,016	665,350	12
table-2.5	10	46,134,745	1,108,999	12
table-3	27	158,403,144	3,797,922	22

Table 3.2 List of Parameters Across Different Layers for Application Execution

Layers	**Parameters**	**Remarks**
Storage-HDFS	input data size	integer, MB
Management-HBase	BlockCache memory size	float
	scanner's number of rows	integer
	regions count	integer
	locality percentage	float
Computing-Spark	executor memory	integer, MB
	executor CPU	integer
	application type	integer
	number of executor	integer

3.5 Performance Prediction of Spark-HBase Applications in Big Data Systems

We propose to use machine learning-based regression algorithms to predict the execution time of Spark-HBase applications. We first describe the method used to optimize the model performance by tuning parameters and then discuss the prediction performance of the proposed model.

3.5.1 Domain Knowledge-based Feature Selection

Existing big data systems provide a number of interfaces for end users to set parameter values to meet the computing needs and performance requirements of their applications. Particularly, Spark and HBase provide around 150 and 197 parameters [47, 6, 3, 12], respectively, to set and tune in XML configuration files. It is computationally infeasible to use a black-box optimization approach to configure the parameters based on an exhaustive profiling strategy since the number of profiling experiments needed grows exponentially with the number of parameters. Therefore, we follow the human-in-the-loop (HITL) strategy to consider a subset of parameters related to Spark executors and HBase RegionServers as shown in Table 3.2 based on our domain knowledge and empirical study.

3.5.2 Model Choices

To accurately predict the performance of Spark-HBase applications, we consider and compare the performance of a set $\{\mathcal{M}\}$ of machine learning algorithms that have been widely used in practice: i) Linear Regression (LR) as a linear model, and ii) Support Vector Regression (SVR) as a kernel-based model.

We utilize the experiment-based cross validation method [37] to solve the following optimization problem:

$$argmin_{\mathcal{M}^*,\theta_m^*}\mathcal{L}(\mathcal{M}(X,\theta_m),y), \tag{3.7}$$

where $\mathcal{M}$ denotes a machine learning model with hyper parameter θ_m, and $\mathcal{L}$ is the loss function. We can use the best model $\mathcal{M}^*$ obtained by optimizing Equation 3.7 to predict application performance with new parameter settings.

3.6 Performance Evaluation

In this section, we set up an experimental environment for executing Spark-HBase applications, and present the prediction results of the machine learning models used for performance prediction.

3.6.1 Experimental Settings

We run our experiments on a cluster of seven HDFS nodes, where one node serves as a the namenode and the other nodes serve as datanodes, and use Apache YARN as a cluster manager. In the experiments, we run Apache Spark-2.4.7 with Scala-2.11.12 on top of Hadoop-2.7.1 with HBase-1.4.13. There are six Spark workers (with two executors by default) and five RegionServers, and one of the slave nodes is configured to be an HBase backup master. Each table has a number of regions managed by RegionServers, which is the same as the number of tasks processed by Spark executors as shown in Table 3.1, and each RegionServer manages twelve regions on average.

3.6.2 Data Source for Performance Prediction

We use the data collected from the applications detailed in Section 3.4 as the source to train and test our proposed models. Spark and HBase has a large parameter space for configuration, but we focus on those tunable parameters related to executors and HBase *scan* API instead of investigating the whole configuration space. For each numerical parameter, we take sample values within a valid range incrementally.

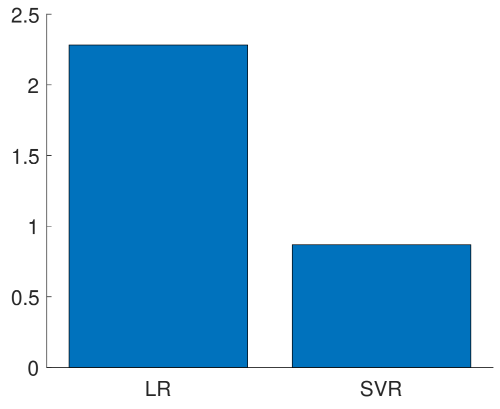

Figure 3.5 Normalized Root Mean Square Error (NRMSE).

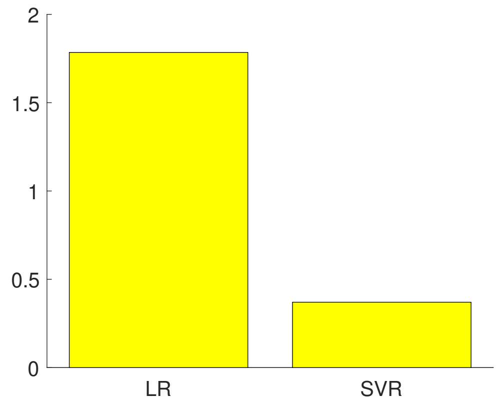

Figure 3.6 Normalized Mean Absolute Error (NMAE).

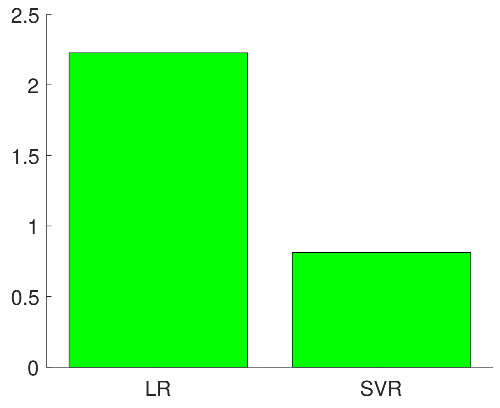

Figure 3.7 Normalized Mean Absolute Percentage Error (NMAPE).

3.6.3 Performance Prediction Results

We use `scikit-learn` library [22] to implement an optimized model in Python for Spark-HBase application execution prediction using two regression algorithms, and use the collected data of performance measurements to feed the model. We start by splitting the data into two parts for training and testing, respectively, and then use the training data to perform a 10-fold cross validation [14] to fine tune two regression models , i.e., LR and SVR. To measure the prediction accuracy of these models, we use the following performance metrics: Normalized Root Mean Square Error (NRMSE), Normalized Mean Absolute Error (NMAE), and Normalized Mean Absolute Percentage Error (NMAPE), as shown in Figure 3.6, Figure 3.6 and Figure 3.6, respectively. Generally, both LR and SVR perform fairly well with SVR achieving higher accuracy. However, we conclude that the data collected from the exploratory experiments are linearly separable, which explains the high accuracy obtained from the prediction models. Furthermore, we believe that running a more complex Spark application would produce performance measurements that could be non-linear, since a complex application might require setting more parameters than what we consider in our experiments. Also, we believe that considering other HBase APIs such as *put* would expose another scope of HBase parameters to be considered.

3.7 Conclusion

In this work, we investigate the problem of performance modeling and prediction for Spark-HBase applications in big data systems. We conduct an exploratory analysis of the effects of different parameters across different layers on Spark-HBase application performance. We propose to use machine learning-based regression algorithms to predict the performance of such applications. Experimental results show that the proposed tuned regression algorithms yield high accuracy compared with the actual execution time. We plan to explore the effects of more parameters for Spark applications using other HBase APIs, and expand the scope of our investigation to Spark-HBase scientific workflows.

REFERENCES

[1] Apache storm. https://storm.apache.org/index.html, Retrieved on 2019.

[2] Hadoop: Fair scheduler. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html, Retrieved on 2019.

[3] Apache hbase reference guide. http://hbase.apache.org/book.html, Retrieved on 2020.

[4] Hbase and hdfs. http://hbase.apache.org/book.html_hbase_and_hdfs, Retrieved on 2020.

[5] Hbase and spark. https://hbase.apache.org/book.htmlspark, Retrieved on 2020.

[6] Spark configuration. https://spark.apache.org/docs/latest/configuration.html, Retrieved on 2020.

[7] Apache hbase. https://hbase.apache.org/, Retrieved on 2021.

[8] Mongodb. https://www.mongodb.com/, Retrieved on 2021.

[9] Montage: Image mosaic service. http://montage.ipac.caltech.edu/, Retrieved on 2021.

[10] G. Ananthanarayanan, S. Kandula, A. Greenberg G, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *Osdi*, volume 10, page 24, 2010.

[11] L. Bao, C.Q. Wu, H. Qi, W. Chen, X. Zhang, W. Han, W. Wei, E. Tai, H. Wang, J. Zhai, and X. Chen. Las: Logical-block affinity scheduling in big data analytics systems. In *INFOCOM Conference on Computer Communications*, pages 522–530. IEEE, 2018.

[12] X. Bao, L. Liu, N. Xiao, F. Liu, Q. Zhange, and T. Zhu. Hconfig: Resource adaptive fast bulk loading in hbase. In *10th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*, pages 215–224. IEEE, 2014.

[13] X. Bao, L. Liu, N. Xiao, Y. Zhou, and Q. Zhange. Policy-driven configuration management for nosql. In *8th International Conference on Cloud Computing*, pages 245–252. IEEE, 2015.

[14] C. Bishop. *Pattern Recognition and Machine Learning.* Springer, 2006.

[15] X. Bu, J. Rao, and Cheng-zhong. Interference and locality-aware task scheduling for mapreduce applications in virtual clusters. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 227–238. ACM, 2013.

[16] F. Change, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[17] R. da Silv, R. Filgueira, I. Pietri, J. Ming, R. Sakellariou, and E. Deelman. A characterization of workflow management systems for extreme-scale applications. *Future Generation Computer Systems*, 75:228–238, 2017.

[18] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[19] E. Deelman, D. Gannon, M. Shields, and I. Taylor. Workflows and e-science: An overview of workflow system features and capabilities. *Future generation computer systems*, 25(5):528–540, 2009.

[20] Y. Demchenko, C. De Laat, and P. Membrey. Defining architecture components of the big data ecosystem. In *International Conference on Collaboration Technologies and Systems (CTS)*, pages 104–112. IEEE, 2014.

[21] M. Elteir, H. Lin, and W. Feng. Enhancing mapreduce via asynchronous data processing. In *Parallel and Distributed Systems (ICPADS), the 16th International Conference on*, pages 397–405. IEEE, 2010.

[22] F. Pedregosa. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[23] L. George. *HBase: the definitive guide: random access to your planet-size data.* O'Reilly Media, Inc., 2011.

[24] Z. Guo, G. Fox, and M. Zhou. Investigation of data locality in mapreduce. In *Proceedings of the International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 419–426. IEEE Computer Society, 2012.

[25] H. Harb, A. Makhoul, and S. Tawbiand R. Couturier. Comparison of different data aggregation techniques in distributed sensor networks. *IEEE Access*, 5:4250–4263, 2017.

[26] M. Hedjazi, I. Kourbane, Y. Gene, and B. Ali. A comparison of hadoop, spark and storm for the task of large scale image classification. In *26th Signal Processing and Communications Applications Conference (SIU)*, pages 1–4. IEEE, 2018.

[27] C. Intanagonwiwat, D. Estrin, R. Govindan, and J. Heidemann. Impact of network density on data aggregation in wireless sensor networks. In *C*, page 457. IEEE, 2002.

[28] E. Ipek, B. De Supinski, M. Schulz, and S. McKee. An approach to performance prediction for parallel applications. In *European Conference on Parallel Processing*, pages 196–205. Springer, 2005.

[29] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *SIGOPS operating systems review*, volume 41, pages 59–72. ACM, 2007.

[30] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the SIGOPS 22nd symposium on Operating systems principles*, pages 261–276. ACM, 2009.

[31] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi. Characterizing and profiling scientific workflows. *Future generation computer systems*, 29(3):682–692, 2013.

[32] M. Khan, Y. Jin, M. Li, Y. Xiang, and C. Jiang. Hadoop performance modeling for job estimation and resource provisioning. *IEEE Transactions on Parallel and Distributed Systems*, 27(2):441–454, 2015.

[33] J. Kroß and H. Krcmar. Modeling and simulating apache spark streaming applications. *Softwaretechnik-Trends*, 36(4):1–3, 2016.

[34] J. Kroß and H. Krcmar. Model-based performance evaluation of batch and stream applications for big data. In *25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 80–86. IEEE, 2017.

[35] B. Lee, D. Brooks, B. De Supinski, M. Schulz, k. Singh, and S. McKee. Methods of inference and learning for performance modeling of parallel applications. In *Proceedings of the 12th SIGPLAN symposium on Principles and practice of parallel programming*, pages 249–258. ACM, 2007.

[36] X. Lin, Z. Meng, C. Xu, and M. Wang. A practical performance model for hadoop mapreduce. In *International Conference on Cluster Computing Workshops*, pages 231–239. IEEE, 2012.

[37] W. Liu, C.Q. Wu, Q. Ye, A. Hou, and W. Shen. Performance modeling and prediction of big data workflows: An exploratory analysis. In *29th International Conference on Computer Communications and Networks (ICCCN)*, pages 1–10. IEEE, 2020.

[38] Y. Lu, I. Comsa, P. Kuonen, and B. Hirsbrunner. Dynamic data aggregation protocol based on multiple objective tree in wireless sensor networks. In *the tenth international conference on intelligent sensors, sensor networks and information processing (ISSNIP)*, pages 1–7. IEEE, 2015.

[39] K. Maheshwari, E. Jung, J. Meng, V. Morozov, V. Vishwanath, and R. Kettimuthu. Workflow performance improvement using model-based scheduling over multiple clusters and clouds. *Future generation computer systems*, 54:206–218, 2016.

[40] H. Mao, M. Schwarzkopf, S. Venkatakrishnan, Z. Meng, and M. Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 270–288, 2019.

[41] K. Maraiya, K. Kant, and N. Gupta. Wireless sensor network: a review on data aggregation. *International Journal of Scientific and Engineering Research*, 2(4):1–6, 2011.

[42] R. Mayer, C. Mayer, and L. Laich. The tensorflow partitioning and scheduling problem: it's the critical path! In *Proceedings of the 1st Workshop on Distributed Infrastructures for Deep Learning*, pages 1–6. ACM, 2017.

[43] S. Mistry, A. Bouguettaya, H. Dong, and A. Erradi. Qualitative economic model for long-term iaas composition. In *International Conference on Service-Oriented Computing*, pages 317–332. Springer, 2016.

[44] F. Nadeem, D. Alghazzawi, A. Mashat, K. Fakeeh, A. Almalaise, and H. Hagras. Modeling and predicting execution time of scientific workflows in the grid using radial basis function neural network. *Cluster Computing*, 20:2805–2819, 2017.

[45] B. Palanisamy, A. Singh, L. Liu, and B. Jain. Purlieus: locality-aware resource allocation for mapreduce in a cloud. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 58. ACM, 2011.

[46] T. Peterka, D. Goodell, R. Ross, H. Shen, and R. Thakur. A configurable algorithm for parallel image-compositing applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 4. ACM, 2009.

[47] P. Petridis, A. Gounaris, and J. Torres. Spark parameter tuning via trial-and-error. In *INNS Conference on Big Data*, pages 226–237. Springer, 2016.

[48] I. Pietri, G. Juve, E. Deelman, and R. Sakellariou. A performance model to estimate execution time of scientific workflows on the cloud. In *the 9th Workshop on Workflows in Support of Large-Scale Science*, pages 11–19. IEEE, 2014.

[49] J. Qin, L. MS, and J. Niu. Massive ais data management based on hbase and spark. In *the 3rd Asia-Pacific Conference on Intelligent Robot Systems (ACIRS)*, pages 112–117. IEEE, 2018.

[50] B. Shangguan, P. Yue, Z. Wu, and L. Jiang. Big spatial data processing with apache spark. In *the 6th International Conference on Agro-Geoinformatics*, pages 1–4. IEEE, 2017.

[51] 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*, pages 1–10.

[52] R. Singhal and P. Singh. Performance assurance model for applications on spark platform. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 131–146. Springer, 2017.

[53] I. Solis and K. Obraczka. In-network aggregation trade-offs for data collection in wireless sensor networks. *International Journal of Sensor Networks*, 1(3-4):200–212, 2006.

[54] G. Song, Z. Meng, F. Huet, F. Magoules, L. Yu, and X. Lin. A hadoop mapreduce performance prediction method. In *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, pages 820–825. IEEE, 2013.

[55] A. Stompel, K. Ma, E. Lum, J. Ahrens, and J. Patchett. Slic: scheduled linear image compositing for parallel volume rendering. In *Proceedings of the Symposium on Parallel and Large-Data Visualization and Graphics*, page 6. IEEE Computer Society, 2003.

[56] J. Tan, S. Meng, X. Meng, and L. Zhang. Improving reducetask data locality for sequential mapreduce jobs. In *INFOCOM*, pages 1627–1635. IEEE, 2013.

[57] R. Tous, A. Gounaris, C. Tripiana, J. Torres, S. Girona, E. Ayguadé, J. Labarta, Y. Becerra, D. Carrera, and M. Valero. Spark deployment and performance evaluation on the marenostrum supercomputer. In *the International Conference on Big Data (Big Data)*, pages 299–306. IEEE, 2015.

[58] J. D. Ullman. Np-complete scheduling problems. *Journal of Computer and System sciences*, 10(3):384–393, 1975.

[59] M. Vora. Hadoop-hbase for large-scale data. In *Proceedings of the International Conference on Computer Science and Network Technology*, pages 601–605. IEEE, 2011.

[60] K. Wang and M. Khan. Performance prediction for apache spark platform. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 166–173. IEEE, 2015.

[61] T. White. *Hadoop: The definitive guide*. O'Reilly Media, Inc., May 19 2012.

[62] C.Q. Wu and H. Cao. Optimizing the performance of big data workflows in multi-cloud environments under budget constraint. In *Proceedings of the 13th IEEE International Conference on Services Computing*, pages 138–145, San Francisco, USA, June 27 - July 2 2016.

[63] Q. Wu and V. Datla. On performance modeling and prediction in support of scientific workflow optimization. In *the World Congress on Services*, pages 161–168. IEEE, 2011.

[64] Q. Wu, J. Gao, Z. Chen, and M. Zhu. Pipelining parallel image compositing and delivery for efficient remote visualization. *Journal of Parallel and Distributed Computing*, 69(3):230–238, 2009.

[65] W. Xiong, Z. Bei, C. Xu, and Z. Yu. Ath: Auto-tuning hbase's configuration via ensemble learning. *IEEE Access*, 5:13157–13170, 2017.

[66] Y. Yao and J. Gehrke. Query processing in sensor networks. In *Cidr*, pages 233–244, 2003.

[67] H. Yu, C. Wang, and K. Ma. Massively parallel volume rendering using 2-3 swap image compositing. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 48.

[68] D. Yun, C.Q. Wu, and Y. Gu. An integrated approach to workflow mapping and task scheduling for delay minimization in distributed environments. *Journal of Parallel and Distributed Computing*, 84:51–64, 2015.

[69] M. Zaharia, D. Borthakur, J Sen. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Job scheduling for multi-user mapreduce clusters. Technical report, Technical Report UCB/EECS-2009-55, EECS Department, University of California, Berkeley, 2009.

[70] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278. ACM, 2010.

[71] M. Zaharia, M. Chowdhury, T. Das, A. Dave, M. McCauley J. Ma, M. 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*, pages 2–2. USENIX Association, April 2012.

[72] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.