On Distributed Information Composition in Big Data Systems

Haifa AlQuwaiee†, Songlin He†, Chase Q. Wu†, Qiang Tang†, Xuewen Shen†‡
† Department of Computer Science, New Jersey Institute of Technology, Newark, NJ 07102, USA
‡ School of Electronics and Information, Communication University of Zhejiang, Hangzhou, Zhejiang 310018, China
Email: {ha226, sh533, chase.wu, qiang, xws}@njit.edu

***Abstract*—Modern big data computing systems exemplified by Hadoop employ parallel processing based on distributed storage. The results produced by parallel tasks such as computing modules in scientific workflows or reducers in the MapReduce framework are typically stored in a distributed file system across multiple data nodes. However, most existing systems do not provide a mechanism to compose such distributed information, as required by many big data applications. We construct analytical cost models and 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. We rigorously prove that DIC-BDS is NP-complete, and propose two heuristic algorithms: Fixed-window Distributed Composition Scheme (FDCS) and Dynamic-window Distributed Composition Scheme with Delay (DDCS-D). We conduct extensive experiments in Google clouds with various composition operators of commonly considered degrees of complexity including $O(n)$, $O(n\log n)$, and $O(n^2)$. 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.**

***Index Terms*—Big data; distributed algorithms; information composition; task scheduling**

I. 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 [1]. 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] [2]. 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 [3], [4]. As we enter the era of big data, workflow applications have been increasingly deployed in big data systems as exemplified by Hadoop [5], [6] using different computing frameworks such as MapReduce for batch parallel data processing [7], Spark for in-memory data processing [8] and Storm for streaming data processing [9].

[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.

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 [10], [11] and CyberShake [11] 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 Hadoop Distributed File System (HDFS) [6]. 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, we construct analytical cost models and 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. We rigorously prove that DIC-BDS is NP-complete, and propose two heuristic algorithms: Fixed-window Distributed Composition Scheme (FDCS) and Dynamic-window Distributed Composition Scheme with Delay (DDCS-D). We conduct extensive experiments 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. Our 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.

978-1-7281-2451-3/19/$31.00 ©2019 IEEE
DOI 10.1109/eScience.2019.00-67

The rest of the paper is organized as follows: Section II conducts a survey of related work. Section III presents the cost models and problem formulation. Section IV details the design of the proposed algorithms. Section V presents experimental results and Section VI concludes our work.

II. Related work

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

In [12], Mayer *et al.* formulated a set of partitioning and scheduling problems in TensorFlow and proved them to be NP-complete. In [13], 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 our study. However, the solutions proposed in their work cannot be directly applied to our problem since they require prior knowledge about the execution of a workflow, which is not always available in practice.

Our 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 [14], 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 our work and has been extensively explored in many other methods [15], [16], [17], [18], [19], [20], [21], [22], [23].

Information integration or aggregation has also been studied in other contexts such as service-oriented computing [24] 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 [25], [26], [27], [28]. These approaches consider minimizing the number of communication messages. Since we focus 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 [25]. 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 adapt two algorithms in 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 [29], [30], [31], [32]. More specifically, in Periodic Sensor Networks (PSN) [33], [34], this method guides sensors to send data collected over a period of time to a Cluster Head (CH) through an aggregation tree.

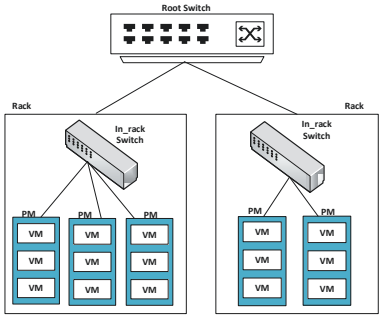

Fig. 1. A general cluster structure.

III. Problem formulation

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

A. Cost Models

1) Cluster Model

As illustrated in Fig. 1, we model a cluster as a tree of Physical Machines (PMs) connected via high-speed switches. Without loss of generality, we consider two-level switches. 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 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} [2]. However, provisioning VMs on PMs is beyond the scope of this paper.

2) 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, our composition model has two components: 1) datasets to be composed, and 2) a composition operator.

a) Datasets

Suppose that we have 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 we consider Hadoop system, the dataset or data block size ranges from 64 to 128 MB as widely implemented in HDFS [6].

In the MapReduce framework [7], 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 [5], 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 [6]. In this work, we consider datasets as HDFS data blocks distributed on a cluster.

b) 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, we need to specify the location where a composition process takes place. 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.

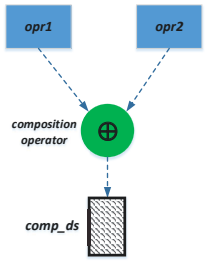

Fig. 2. A single-composition operation.

3) Time Cost

a) 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 [2]:

$$T_{dt} = \frac{DS}{min(\frac{BW_{up}}{n_s}, \frac{BW_{down}}{n_r})}, \quad (1)$$

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.

Also, given the network topology in our cluster model, we need to consider the uplink and downlink bandwidths for the root switch S_{root} and in-rack switches $S_{in-rack}$. We write Eq. 1 as follows:

$$T_{dt} = \frac{DS}{min(\frac{BW_{up}}{n_s}, \frac{BW_{down}}{n_r})} + \frac{DS}{min(\frac{BWs_{up}}{ns_s}, \frac{BWs_{down}}{ns_r})}, \quad (2)$$

where ns_s and ns_r are the number of concurrent data transfers from sender $S_{in-rack}$ and to receiver $S_{in-rack}$, respectively.

b) I/O Cost

Data could be stored locally on the map or reduce node, or distributed in HDFS. Accessing data on disk is expensive especially in the case of big data. We denote the cost of reading and writing data on disk as:

$$T_{I/O} = MAX(I/O_{local}, I/O_{HDFS}). \quad (3)$$

Computing the actual I/O time requires the knowledge about the size of the data being accessed and the speed of the I/O operation.

c) Composition Time

Fig. 2 illustrates a single-composition process, where two datasets are aggregated by a composition operator $\oplus$. We calculate the time of such a single-composition process as:

$$T_C = T_{I/O} + T_{dt} + f_{CT}(O(\oplus), DS), \quad (4)$$

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, Eq. 4 is insufficient to model the composition time. Considering a dynamic case where the composition process starts at different times, since multiple compositions take place concurrently, we consider the longest one as the total time needed for composition. In other words, we consider the critical path (CP) in this distributed scheme to define the total composition time (TCT), i.e.,

$$TCT = \sum (T_{I/O} + T_{dt} + f_{CT}(O(\oplus), DS))_{CP}, \quad (5)$$

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

B. 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, ⋯, VM_m) provisioned on a number of PMs, we aim 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.*

C. Complexity Analysis

We prove the NP-completeness of DIC-BDS by reducing an existing NP-complete problem, Single Execution Time Scheduling (SETS) [35], to it in polynomial time.

We first consider a decision version of our problem 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 [35], [12] 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

TABLE I
NOTATIONS USED IN THE COST MODELS AND PROBLEM DEFINITION.

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
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
I/O_{local}	the cost of I/O on a local node
I/O_{HDFS}	the cost of I/O in HDFS
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

function $g : S \to \{0,...,t_{max}-1\}$ such that the following three properties hold? (i) The scheduling function respects the ordering relation, i.e., $v \in S \prec v' \in S \to g(v) < g(v')$. (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. We prove its NP-hardness by reducing SETS to it as follows.

Let I_{SETS} be an arbitrary instance of SETS, which has a set of jobs S and a partial order $\prec$ on S. Accordingly, we construct an instance of DIC-BDS denoted as $I_{DIC-BDS}$. For each partial order $\prec_i$ of I_{SETS}, we construct a corresponding bucket b_i of $I_{DIC-BDS}$ 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, we show that 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 $\prec_1$, we start 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 $I_{DIC-BDS}$, we set the number of VMs to be k', which is the same as the number of processors in I_{SETS}. Also, since the processors in I_{SETS} are homogeneous, we only consider homogenous VMs in $I_{DIC-BDS}$. Furthermore, we focus 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 we use that scheduling scheme 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 we have 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. □

IV. ALGORITHM DESIGN

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

The main goal of DCS is to minimize TCT. In this scheme, we have 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. 5, 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

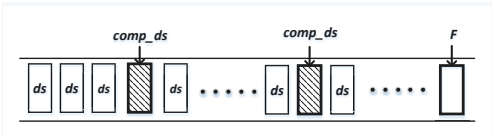

Fig. 3. The global list (GL).

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

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

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

A. 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 Fig. 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* (Section IV-D) to calculate the *TCT* as shown in Alg. 3 and are followed by the group scheduling algorithm (Section IV-E) to determine the composition order.

B. Fixed-window Distributed Composition Scheme (FDCS)

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

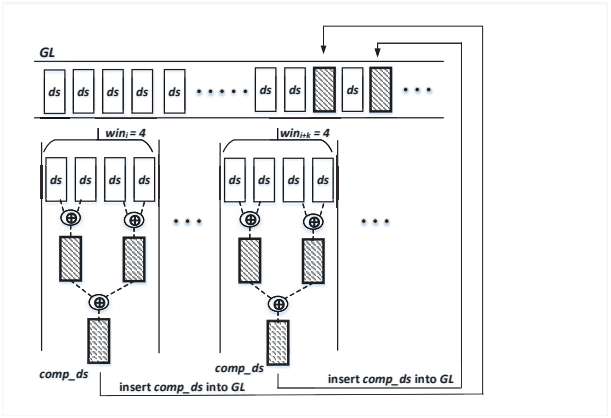

Fig. 4. FDCS.

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
```

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

We adapt the concept of delay scheduling from [36], which aims to improve the performance of Hadoop system using a default fair scheduler, and design a Dynamic-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 Alg. 2 with an illustration of its process in Fig. 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
```

D. Composition Tree CTree

As shown in Fig. 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 Alg. 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 Alg. 3.

E. 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 [17]. 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

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;
```

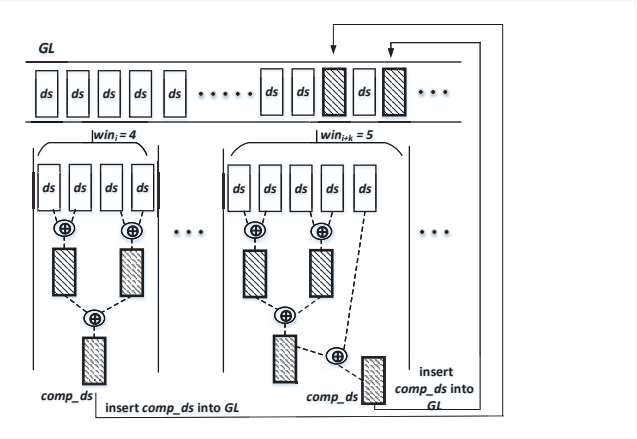

Fig. 5. DDCS-D.

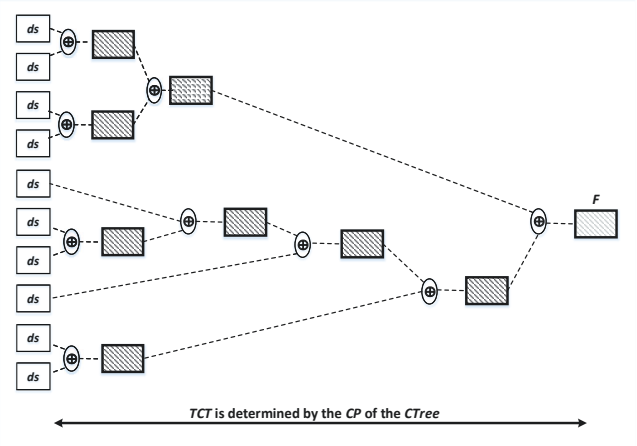

Fig. 6. Illustration of the composition tree (*CTree*).

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 Alg. 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();
4:    for all (pair p_i ∈ pairs[]) do
5:       comp_ds = compose(p_i);
6:       update_CTree(p_i, comp_ds);
7:       GL_insert(comp_ds);
```

F. Algorithms for Comparison

1) 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 Alg 5 and illustrated in Fig 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 Alg. 5, we calculate the *TCT* as the available time of the final output or last dataset *F*, as shown in Alg 6.

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;
```

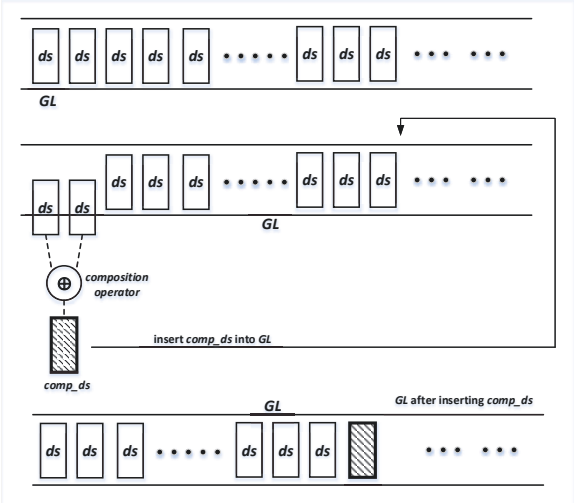

Fig. 7. Greedy composition.

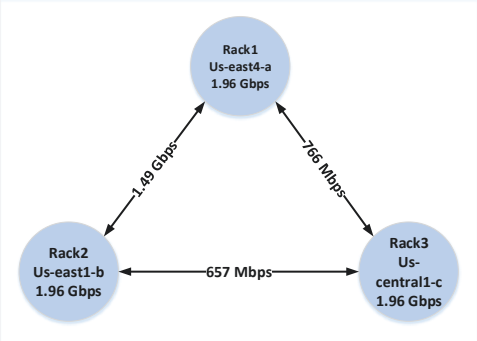

Fig. 8. Intra-rack and inter-rack bandwidths on the cluster testbed.

2) 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, e.g., 10 seconds. Both Greedy and Periodic algorithms follow the scheduling procedure as described in Section IV-E.

V. Algorithm Implementation and Performance Evaluation

A. 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 Fig. 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.

B. 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.

1) 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 II and Fig. 9 show the performance measurements of different algorithms processing various numbers of datasets.

2) 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 III and Fig. 10 show the results of different algorithms processing various numbers of datasets.

3) 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 IV and Fig. 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.

4) Algorithm Execution Dynamics

a) 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. Fig 14 plots the change of the window size with three degrees of complexity for a problem size of 300 datasets. As shown in Fig 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.

b) 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 V and Fig 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.

c) Optimization of the Delay in DDCS-D

To find the most efficient delay for DDCS-D, we conduct an experiment with 300 datasets. Fig 13 and Table VI 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.

d) 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 $O(n^2)$.

VI. Conclusion

We formulated a generic problem of Distributed Information Composition in Big Data Systems, referred to as DIC-BDS,

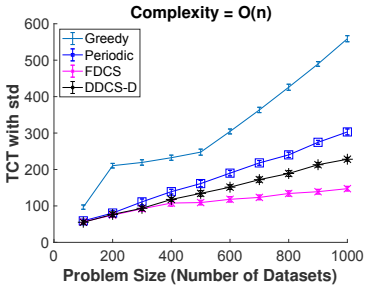

Fig. 9. The average TCT (seconds) of different algorithms under different problem sizes with an operator of complexity $O(n)$ and the corresponding standard deviations.

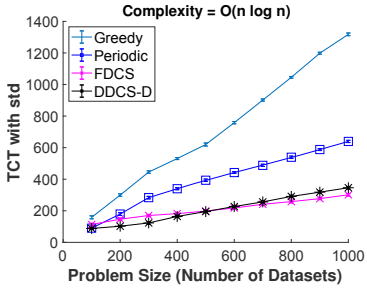

Fig. 10. The average TCT (seconds) of different algorithms under different problem sizes with an operator of complexity $O(n\log n)$ and the corresponding standard deviations.

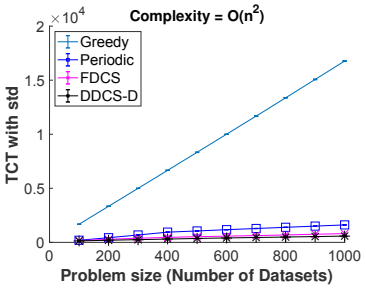

Fig. 11. The average TCT (seconds) of different algorithms under different problem sizes with an operator of complexity $O(n^2)$ and the corresponding standard deviations.

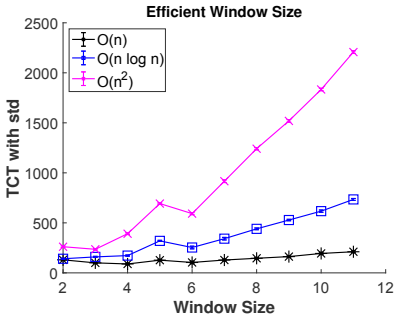

Fig. 12. The average TCT of FDCS with composition operators of different complexities for processing 300 datasets with different window sizes.

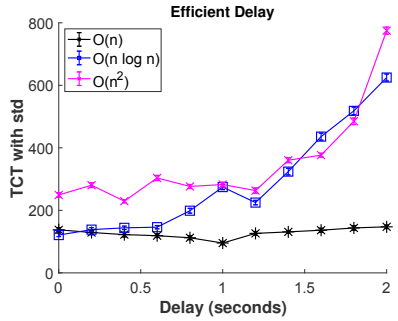

Fig. 13. The average TCT of DDCS-D with composition operators of different complexities for processing 300 datasets with different delay time.

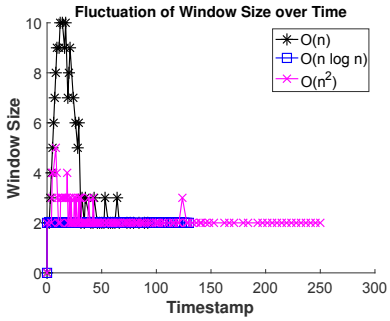

Fig. 14. The fluctuation of the window size over time in DDCS-D for three degrees of complexity.

TABLE II
THE AVERAGE TCT (SECONDS) OF DIFFERENT ALGORITHMS UNDER DIFFERENT PROBLEM SIZES WITH AN OPERATOR OF COMPLEXITY $O(n)$ AND THE CORRESPONDING STANDARD DEVIATIONS.

# 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 III
THE AVERAGE TCT (SECONDS) OF DIFFERENT ALGORITHMS UNDER DIFFERENT PROBLEM SIZES WITH AN OPERATOR OF COMPLEXITY $O(n\log n)$ AND THE CORRESPONDING STANDARD DEVIATIONS.

# 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 IV
THE AVERAGE TCT (SECONDS) OF DIFFERENT ALGORITHMS UNDER DIFFERENT PROBLEM SIZES WITH AN OPERATOR OF COMPLEXITY $O(n^2)$ AND THE CORRESPONDING STANDARD DEVIATIONS.

# 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 V
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

which was proven to be NP-complete. We designed heuristic algorithms for DIC-BDS that take into consideration the arrival dynamics of datasets, and demonstrated their performance superiority over other existing methods for composition operators of various time complexities 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 our future interest to investigate the problem with other distributed frameworks such as Spark and evaluate our algorithms with real-life big data workflows. Also, we plan to implement and integrate these algorithms into Hadoop.

TABLE VI
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

ACKNOWLEDGMENTS

This research is sponsored by the U.S. National Science Foundation under Grant No. CNS-1828123 with New Jersey Institute of Technology. The work is also partially supported by the Key Research and Development Program of Zhejiang Province, China under Grant No. 2019C03138 with Communication University of Zhejiang, China.

REFERENCES

[1] Y. Demchenko, C. D. Laat, and P. Membrey, "Defining architecture components of the big data ecosystem," in *2014 International Conference on Collaboration Technologies and Systems (CTS)*. IEEE, 2014, pp. 104–112.

[2] C. 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*, San Francisco, USA, June 27 - July 2 2016, pp. 138–145.

[3] 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*, vol. 75, pp. 228–238, 2017.

[4] 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*, vol. 25, no. 5, pp. 528–540, 2009.

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

[6] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. IEEE, 2010, pp. 1–10.

[7] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[8] M. Zaharia, M. Chowdhury, T. Das, A. Dave, M. M. 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*. USENIX Association, April 2012, pp. 2–2.

[9] "Apache storm," https://storm.incubator.apache.org, 2015.

[10] "Montage: Image mosaic service," http://montage.ipac.caltech.edu/docs/grid.html,2005.

[11] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi, "Characterizing and profiling scientific workflows," *Future generation computer systems*, vol. 29, no. 3, pp. 682–692, 2013.

[12] 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*. ACM, 2017, pp. 1–6.

[13] D. Yun, C. 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*, vol. 84, pp. 51–64, 2015.

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

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

[16] G. Ananthanarayanan, S. Kandula, A. G. G, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri." in *Osdi*, vol. 10, no. 1, 2010, p. 24.

[17] 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*. ACM, 2013, pp. 227–238.

[18] "Hadoop: Fair scheduler," https://hadoop.apache.org/docs/r2.7.4/hadoop-yarn/hadoop-yarn-site/FairScheduler.html, 2017.

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

[20] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *ACM SIGOPS operating systems review*, vol. 41, no. 3. ACM, 2007, pp. 59–72.

[21] Z. Guo, G. Fox, and M. Zhou, "Investigation of data locality in mapreduce," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. IEEE Computer Society, 2012, pp. 419–426.

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

[23] J. Tan, S. Meng, X. Meng, and L. Zhang, "Improving reducetask data locality for sequential mapreduce jobs," in *INFOCOM, 2013 Proceedings IEEE*. IEEE, 2013, pp. 1627–1635.

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

[25] 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*, vol. 69, no. 3, pp. 230–238, 2009.

[26] 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*, 2008, p. 48.

[27] 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*. ACM, 2009, p. 4.

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

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

[30] Y. Yao and J. Gehrke, "Query processing in sensor networks," in *Cidr*, 2003, pp. 233–244.

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

[32] K. Maraiya, K. Kant, and N. Gupta, "Wireless sensor network: a review on data aggregation," *International Journal of Scientific and Engineering Research*, vol. 2, no. 4, pp. 1–6, 2011.

[33] H. Harb, A. Makhoul, and S. T. R. Couturier, "Comparison of different data aggregation techniques in distributed sensor networks," *IEEE Access*, vol. 5, pp. 4250–4263, 2017.

[34] I. Solis and K. Obraczka, "In-network aggregation trade-offs for data collection in wireless sensor networks," *International Journal of Sensor Networks*, vol. 1, no. 3-4, pp. 200–212, 2006.

[35] J. D. Ullman, "Np-complete scheduling problems," *Journal of Computer and System sciences*, vol. 10, no. 3, pp. 384–393, 1975.

[36] M. Zaharia, D. Borthakur, J. S. 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*. ACM, 2010, pp. 265–278.