ActCap: Accelerating MapReduce on Heterogeneous Clusters with Capability-Aware Data Placement

Bo Wang*, Jinlei Jiang*†‡, Guangwen Yang*
*Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology
Ministry of Education Key Laboratory for Earth System Modeling, Center for Earth System Science
Tsinghua University, Beijing 100084, China
†Technology Innovation Center at Yinzhou
Yangtze Delta Region Institute of Tsinghua University, Ningbo 315000, China
‡Corresponding author: jjlei@tsinghua.edu.cn
bo-wang11@mails.tsinghua.edu.cn, ygw@tsinghua.edu.cn

***Abstract*—As a widely used programming model and implementation for processing large data sets, MapReduce performs poorly on heterogeneous clusters, which, unfortunately, are common in current computing environments. To deal with the problem, this paper: 1) analyzes the causes of performance degradation and identifies the key one as the large volume of inter-node data transfer resulted from even data distribution among nodes of different computing capabilities, and 2) proposes ActCap, a solution that uses a Markov chain based model to do node-capability-aware data placement for the continuously incoming data. ActCap has been incorporated into Hadoop and evaluated on a 24-node heterogeneous cluster by 13 benchmarks. The experimental results show that ActCap can reduce the percentage of inter-node data transfer from 32.9% to 7.7% and gain an average speedup of 49.8% when compared with Hadoop, and achieve an average speedup of 9.8% when compared with Tarazu, the latest related work.**

Keywords*—*MapReduce, Heterogeneous Clusters, Data Placement, Load Balancing, Big Data

I. Introduction

The human society has stepped into the big data era where applications that process terabytes (TB) or petabytes (PB) of data are common in science, industry and commerce. Usually, such applications are termed data-intensive applications in order to be distinguished from compute-intensive ones that are compute bound or in other words, spend most of their execution time on computation. The era of big data presents many challenges and requires new ways to store, manage, access and process the colossal amount of available data (both structured and unstructured). Since parallel processing is scalable and can gain performance improvement by several orders of magnitude, it is generally accepted as a must for data-intensive applications in the big data era.

MapReduce [9] is a programming model and an associated implementation for parallel large data sets processing on clusters with hundreds or thousands of nodes. Due to its scalability and ease of programming, MapReduce has been adopted by many companies, including Google, Yahoo, Microsoft, and Facebook. Nowadays we can see MapReduce applications in a wide range of areas such as distributed sort, web link-graph reversal, finding term-vector per host, web log analysis, inverted index construction, document clustering, collaborative filtering, machine learning, and statistical machine translation, to name but just a few.

In spite of the above facts, MapReduce is far from perfect. It is a lasting effort to improve the performance of MapReduce applications. From the perspective of data supply, the research work can be roughly divided into two categories. The first category tries to improve the performance by more efficient data placement. Typical examples are RCFile [17], Xie [31], MRA++ [7], and CoHadoop [12]. The philosophy behind this category is that loading data is the most time-consuming step of MapReduce especially for data-intensive applications and faster data loading usually means improved performance. The second category tries to improve the performance by modifying the task scheduling strategy. Typical examples are Delay Scheduling [33] and Tarazu [2]. The key idea behind this category is to reduce the volume of data transferred over the network—since bandwidth is a scarce resource in a MapReduce environment, reduced data transfer usually means improved performance.

While MapReduce applications perform well on homogeneous clusters as a result of the above work, they perform poorly on heterogeneous clusters. As pointed out in [2], the performance of Hadoop—an open-source implementation of MapReduce—could decrease 20∼75% on a heterogeneous cluster. Unfortunately, computing environments with heterogeneous clusters are common nowadays due to the tradeoff between hardware cost, power efficiency and so on. Moreover, this trend will continue [27]. Indeed, even if heterogeneity is not introduced at the design time, the daily maintenance or hardware upgrade would result in additional heterogeneity. In addition, due to resource sharing, heterogeneity could also arise from background load variation and other coexisting running jobs [35]. Therefore, it is of significance to do MapReduce optimization on heterogeneous clusters.

In this paper, we argue that the fundamental cause of the poor performance of MapReduce applications on heterogeneous clusters is the even distribution of data among nodes with different processing capabilities, which is the default policy of the distributed file systems (e.g., GFS and HDFS) that back MapReduce applications up. For those nodes of higher capability, they can finish local data processing soon and become idle. After that, the scheduler of the MapReduce

978-1-4799-8381-0/15/$31.00 ©2015 IEEE

framework will assign them new tasks whose data are located on other nodes. For these tasks, data transfer over the network, which is time-consuming as aforementioned, is inevitable. That is the problem.

To deal with the problem, we suggest distributing data among nodes according to their computing capabilities and propose ActCap, a solution for predicting the computing capabilities of nodes and doing data placement accordingly. The contributions of our paper are as follows.

- We identify the key cause of the poor performance of MapReduce applications on heterogeneous clusters as large volume of inter-node data transfer resulted from even data distribution among nodes of different computing capabilities and suggest the idea of distributing data according to the capabilities of computing nodes.
- We propose a Markov chain based approach to measure the computing capability of a node, which covers not only the effect of hardware configuration but also the background workloads.
- We devise ActCap, a solution that can determine the nodes computing capabilities of a cluster on the fly via a Markov chain based model and then do data placement accordingly.
- Extensive experiments have been conducted to evaluate ActCap. The experimental results show that ActCap can reduce inter-node data transfer from 32.9% to 7.7% on the given heterogeneous cluster.

The rest of this paper is organized as follows. Section II gives a brief introduction to MapReduce and analyzes the reasons why MapReduce applications perform poorly on heterogeneous clusters. Section III describes the key ideas and algorithms of ActCap. Section IV shows the experimental results. Section V reviews the related work. Section VI discusses our work and the paper ends in Section VII with some conclusions.

II. Problem Analysis

In this section, we first give a brief introduction to MapReduce and then set out to find out the reasons why MapReduce applications perform poorly on heterogeneous clusters.

A. Overview of MapReduce

The execution of a MapReduce program consists of three phases, that is, the Map phase, the Shuffle phase, and the Reduce phase. In the Map phase, the MapReduce framework reads the input data from the source specified by the program, partitions it into splits, and assigns them to Mappers, the designated nodes of a MapReduce cluster where the Map tasks will execute. After that, the user-providing Map function is executed simultaneously on multiple Mappers and generates a set of intermediate results. In the Shuffle phase, all the Map-generated results are combined and sorted by key first, and then transferred as needed to Reducers, the designated nodes where the Reduce tasks will execute. In the Reduce phase, multiple Reduce tasks are executed in parallel on the Reducers to produce a final result.

In a MapReduce environment, the underlying distributed file system shares the same cluster with the MapReduce framework. By default, the distributed file system places data evenly across all nodes of that cluster regardless of their capabilities. In recognition of network as a scarce resource, the MapReduce framework bears in mind the idea that "moving data is more expensive than moving computation" and exploits *data locality* (i.e., co-locating computation with data) as much as possible to guarantee the performance of applications. For example, in the Map phase, the framework (or more precisely, the Master program) will try to run map tasks on the same machine, or at least the same rack where input data locates.

There are many MapReduce frameworks available and in this paper, we base our prototype on YARN [32], the latest version of Apache Hadoop, which is probably the most popular open-source implementation of the MapReduce model.

B. Reasons for Poor Performance on Heterogeneous Cluster

There are many reasons that can lead to performance degradation of MapReduce applications. Data skew and skewed popularity are the inherent attributes of applications and have received extensive studies [21] [20] [15]. Since data skew usually appears in the Reduce phase and can be relatively well handled by YARN, we pay no attention to it in this paper. Similarly, we pay no attention to skewed content popularity since the solution to it can be easily duplicated. Anyway, these studies provide a good basis for us to improve MapReduce performance further even in a heterogenous environment.

Zaharia et al. [35] first examined the reasons why MapReduce suffers from performance degradation in heterogeneous environments. They owed the reasons to the breaking down of (implicit) assumptions made during MapReduce design and proposed LATE [35], a new scheduling algorithm for better identifying and managing straggler tasks. Ahmad et al. [2] thought the poor performance is due to two key factors, namely the heavy network communication caused by the built-in load balancing mechanism during the Map phase and the amplified load imbalance of Reduce computation. They proposed Tarazu [2] to solve the problem from the perspective of scheduling. While both LATE and Tarazu can improve MapReduce performance on heterogeneous clusters, the issues they address are fundamentally different. In this paper we try to address the same issue as that of Tarazu, but from a data placement perspective and with a focus on the Map phase.

To better illustrate the problem and make a good starting point for ActCap, we conducted a comparison experiment with the results shown in Table I, where the data is evenly distributed among all the nodes and the terms Big and Small are used to indicate the computing capability of the corresponding nodes. We can see from the table that almost all the blocks are processed locally when the job runs on homogeneous clusters, regardless of the computing capability of the nodes. However, things are different for job execution on the heterogeneous cluster, where about one third of the data stored on small nodes is sent to other nodes for processing and about half of the data processed by big nodes is fetched from other nodes. The reasons are as follows.

For those nodes of higher capability (i.e., the big nodes in Table I), they can finish local data processing soon and become

TABLE I. DATA TRANSFER CHARACTERISTICS OF THE WORD-COUNT PROGRAM ON HETEROGENEOUS AND HOMOGENEOUS CLUSTERS USING THE SETTINGS IN SECTION IV-A

	Heterogeneous		Homogeneous	
	Small	Big	Small-Only	Big-Only
Local	63.3%	50.5%	99.6%	97.3%
In	3.2%	46.7%	0.4%	2.7%
Out	33.5%	2.9%	0.4%	2.7%

idle. After that, in order to make full use of cluster resources, the built-in load balancer of the MapReduce framework will assign them new tasks with input data locating on other nodes. The same thing will happen for small nodes when they are idle. For these tasks, time-consuming data transfer is inevitable. Obviously, this problem seldom exists for homogeneous clusters whose nodes are of the same computing capability, for they can finish the given tasks almost at the same time.

From the above analysis, we can conclude that one key cause, if not the root one, of the poor performance of MapReduce applications on heterogeneous clusters is the large volume of inter-node data transfer resulted from the default policy of the underlying distributed file system—distributing data evenly among nodes regardless of their computing capabilities. To solve the problem, we suggest data be distributed among nodes according to their computing capabilities. As an endeavor in this direction, we propose ActCap, a solution that dynamically determines the node computing capability of a cluster and then does data placement accordingly.

III. ACTCAP DESIGN

ActCap is designed with an assumption that data comes into the system continuously. Such an assumption is made because applications in the real world usually undergo a process to load data from some external sources to the MapReduce cluster. To achieve the purpose of ActCap, three key problems involved are how to define the computing capability of a node, how to determine the computing capability of a node on the fly, and how to distribute the incoming data to nodes according to their capabilities.

A. Computing Capability Definition

Determining the computing capability of a node is a prerequisite step to do capability-aware data placement. At the core of this step is how to define the computing capability. Roughly speaking, there are three main kinds of methods. The first and simplest kind is to use hardware specifications (e.g., merely CPU frequency, number of CPUs/CPU cores, memory capacity or a more complex combination of them) in a static way. Methods of this kind suffer from the problem that the same configuration might mean different processing capability to different workloads. The second kind, with *horse power factor* [19] as a typical example, dynamically calculates the capability of a node using not only information about hardware specifications but also the load on that node. Methods of this kind have the same problem with that of the first kind. The third and last kind, as shown in [31], uses an indirect sampling-based way—the node capability is got via running a small portion of the data set first on that node. Due to the presence of data skew, the result got might not be accurate enough. In addition, it is costly to do so.

In our opinion, both hardware specifications and workloads can impact the "real" computing capability of a node. Moreover, the interference of jobs running concurrently on the same node makes things even more complex. So, a better way to measure the computing capability of a node is to use the number of data blocks processed by that node in a given period. Taking all these factors into account, we utilize a two-state Markov chain model, as illustrated in Fig. 1, to describe the behavior of a node. A Markov chain model is selected because it is a widely used and well proven method in performance prediction [23] and workloads in data centers usually show some similarity in a certain period [24]. In the model, the state ON indicates the computing capability of the node is insufficient, or in other words, too many blocks in the node are sent out and processed remotely (by other nodes). The state OFF indicates the node is of sufficient computing capability and almost all the blocks in the node are processed locally.

B. Computing Capability Determination

With the concept of computing capability defined, we can now calculate the computing capability of each node in a given cluster. Since the workloads in a cluster usually vary with time as aforementioned, computing capability determination is done periodically in order to get the most accurate results. In each period, we execute an algorithm to find out the number of incapable (or overloaded) nodes and to rank the capabilities of all nodes. With the number of incapable nodes found and the capabilities of all nodes ranked, we can then exclude those incapable nodes when doing data placement in that period.

To fulfill the task of node computing capability determination, we build the following Markov chain model on the basis of the computing capability definition in Section III-A to depict the dynamic behavior of a given cluster.

For a MapReduce cluster of m nodes, its behavior can be modeled by a Markov chain of $m+1$ states as illustrated in Fig. 2. We say the cluster is in state i if there are i overloaded nodes in it. Let $\alpha(t)$ be the number of incapable nodes at the time t (t=0,1,2,...), $\beta(t)$ be the number of nodes that switch state from ON to OFF, and $\gamma(t)$ be the number of nodes that switch state from OFF to ON, then the Markov chain given in Fig. 2 can be described by the following stochastic process:

$$\alpha(t+1) = \alpha(t) - \beta(t) + \gamma(t) \tag{1}$$

Since each node in the cluster changes independently, $\beta(t)$ and $\gamma(t)$ are mutually independent and both follow the binomial distribution below:

$$\begin{cases} \beta(t) \sim B(\alpha(t), p_{off}) \\ \gamma(t) \sim B(m-\alpha(t), p_{on}) \end{cases} \tag{2}$$

$\Leftrightarrow$

$$\begin{cases} Pr\{\beta(t)=x\} = \binom{x}{\alpha(t)} p_{off}^{x} (1-p_{off})^{\alpha(t)-x} \\ Pr\{\gamma(t)=x\} = \binom{x}{m-\alpha(t)} p_{on}^{x} (1-p_{on})^{m-\alpha(t)-x} \end{cases} \tag{3}$$

Let p_{ij} be the transition probability from state i to state j, $P = [p_{ij}]$ be the matrix denoting one-step transition probabilities of $\alpha(t)$, $\Pi = (\pi_0, \pi_1, \ldots, \pi_m)$ be the probability

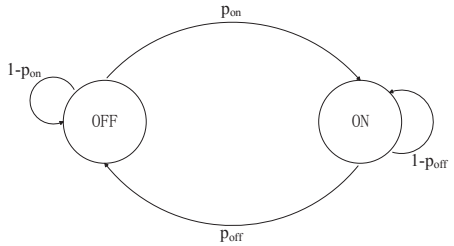


Fig. 1. A two-state Markov chain. The state ON indicates too many blocks are processed remotely and the state OFF indicates almost all the blocks are locally processed.

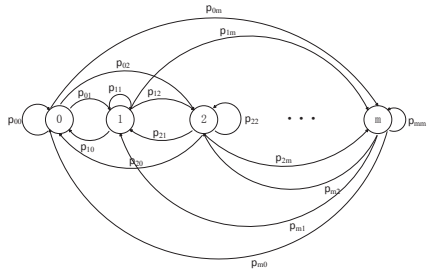


Fig. 2. The Markov chain used to describe the load of a cluster with *m* nodes, where the number associated with the state indicates how many nodes are in ON state.

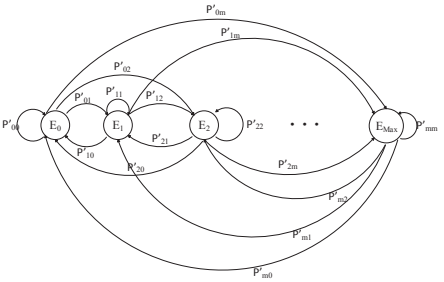


Fig. 3. The Markov chain used to describe the computing capability of one node, where E_i associated with the state denotes the rank of computing capability.

distribution of $\alpha(t)$ when t reaches infinity, and $\binom{i}{j} = 0$ for $i < 0$ or $i > j$, then we have:

$$p_{ij} = Pr\{\alpha(t+1) = j | \alpha(t) = i\} \tag{4}$$

$$= \sum_{r=0}^{i} Pr\{\beta(t) = r, \gamma(t) = j - i + r | \alpha(t) = i\} \tag{5}$$

$$= \sum_{r=0}^{i} Pr\{\beta(t) = r | \alpha(t) = i\} Pr\{\gamma(t) = j - i + r | \alpha(t) = i\} \tag{6}$$

$$= \sum_{r=0}^{i} \binom{r}{i} p_{off}^{r} (1 - p_{off})^{i-r} \binom{j-i+r}{m-i} p_{on}^{j-i+r} (1 - p_{on})^{m-j-r} \tag{7}$$

$$\Pi = \lim_{t \to \infty} \Pi_0 P^t \tag{8}$$

$$\alpha(t) = \sum_{i=0}^{m} \pi_i i \tag{9}$$

where $\Pi_0 = (1, 0, 0, \ldots, 0)$ is the initial value of Π, π_i is the limiting probability of $\alpha(t)$ in state i, and P^t is the matrix of t-step transition probabilities.

Let $\kappa(t), \lambda(t), \mu(t)$ denote the number of blocks *processed locally*, *sent out* (to other nodes for processing) and *read in* (from other nodes and processed locally) of each node. We say the node is in ON state if the inequality $\dfrac{\lambda(t) - \mu(t)}{\kappa(t)} > \xi$ is satisfied. Otherwise we say the node is in OFF state. To calculate p_{off} and p_{on}, we divide each phase into n intervals, in each of which we calculate whether the node is in ON or OFF state, count the number of changing from ON to OFF state and changing from OFF to ON state, and then get the probabilities respectively. Actually, p_{off} and p_{on} should be different for different nodes. However, doing so would make the model/algorithm unnecessarily complex without much benefit. In addition, an absolute (accurate) value makes no sense in our case. Therefore, we use the same value for all nodes.

To rank the computing capabilities of all nodes in the given cluster, we introduce $\tau(t)$ to denote the total number of data blocks consumed by all nodes of that cluster. Then we have the following equations:

$$\tau(t) = \sum_{i=1}^{m} (\kappa_i(t) + \lambda_i(t)) \tag{10}$$

$$\phi_i(t) = a \frac{\kappa_i(t)}{\tau_i(t)} + b \frac{\mu_i(t)}{\kappa_i(t)} - c \frac{\lambda_i(t)}{\kappa_i(t)} \tag{11}$$

$$\varepsilon_i(t) = map\_to\_rank(\phi_i(t)) \tag{12}$$

where the weights *a, b, c* in Equation 11 are initially set to *a=10, b=15, c=17* and then periodically optimized by Gradient Descent method [26] in our experiments. The computing capability value derived from Equation 11 is continuous, so we define $Max + 1$ ranks $\varepsilon_i(t) \in E = \{E_0, E_1, E_2, \ldots, E_{Max}\}$ and use Equation 12 to map the continuous value to discrete ranks. Since $\{\varepsilon(t), t = 0, 1, 2, \ldots\}$ is a stochastic process from which we can also construct a Markov chain with $Max + 1$ states (as illustrated in Fig. 3). The stochastic process is in state E_i when the computing capability $\phi(t)$ is mapped to rank E_i at any phase t.

Let p'_{ij} be the transition probability from state i to state j, in other words, if $\varepsilon(t) = E_i$ the probability that $\varepsilon(t+1) = E_j$ is p'_{ij}. We divide each phase into n intervals, in each of which we calculate computing capability of the node, count the number of changing from E_i to E_j, and then get the probability p'_{ij} using statistics method. $P' = [p'_{ij}]$ denotes the matrix of one-step transition probabilities. Let n-tuple $\Pi'_0 = (0, \ldots, 0, 1, 0, \ldots, 0)$ denote the initial state. Π'_0 indicates that at phase $t = 0$ $Pr\{\varepsilon(0) = E_{Max/2}\} = 100\%$, then at phase t the state variable is:

$$\Pi' = \lim_{t \to \infty} \Pi'_0 P'^t \tag{13}$$

$$\psi = \sum_{i=0}^{Max} \pi i \tag{14}$$

$$\varepsilon(t) = E_\psi \tag{15}$$

After we get the number of incapable nodes $\alpha(t)$ and the capabilities of all nodes, we sort the nodes in descending order of their capabilities and add the last $\alpha(t)$ nodes to $List_{incap}$ as the incapable ones.

C. Capability-Aware Data Placement

The complete node-capability-aware data placement solution is described in Algorithm 1, where data replication of HDFS is also taken into account. The algorithm has three inputs, where $E = [\varepsilon_i]_k$ is the node computing capabilities predicted with the way stated in Section III-B, $List_{incap}$ is the list of incapable nodes, and id is the data block identifier. The output of this algorithm is $Nodes\_List$, a list of nodes where the replicas of the incoming data block should be seated. For each replica, we do the node selection process once according to the latest computing capabilities of all nodes.

We use consistent hashing in the *ChooseFromCandidates* function to choose a right data node to store the given data block. As illustrated in Fig. 4, we place all candidate nodes (e.g., DN_1,..., DN_k) on a ring according to their computing capabilities. The region between a data node and

Algorithm 1 Capability-Aware Data Placement.

Input: $E = [\varepsilon_i]_k$, $List_{incap}$, id;
Output: **Nodes_List**;
1: **if** *IsDataNode(writer)* && writer $\notin List_{incap}$ **then** ▷ choose node for the 1^{st} replica
2: add $writer \rightarrow$ **Nodes_List**;
3: **else**
4: $Nodes_{cand}$={node|node$\in$*GetRack(writer)*}$-List_{incap}$;
5: ChooseFromCandidates($Nodes_{cand}$,id,$chosen\_node$);
6: add $chosen\_node \rightarrow$ **Nodes_List**;
7: **end if**
8: $Nodes_{cand}$={node|node$\notin$*GetRack*(1^{st} *replica*)}$-List_{incap}$; ▷ choose node for the 2^{nd} replica
9: ChooseFromCandidates($Nodes_{cand}$,id,$chosen\_node$);
10: add $chosen\_node \rightarrow$ **Nodes_List**;
11: $Nodes_{cand}$={node|node$\notin$*GetRack*(2^{nd} *replica*)}$-List_{incap}$; ▷ choose node for the 3^{rd} replica
12: ChooseFromCandidates($Nodes_{cand}$,id,$chosen\_node$);
13: add $chosen\_node \rightarrow$ **Nodes_List**;
14: **if** $replica\_number > 3$ **then** ▷ choose node(s) for the rest replica(s)
15: $Nodes_{cand}$={node|node$\notin List_{incap}$};
16: ChooseFromCandidates($Nodes_{cand}$,id,$chosen\_node$);
17: add $chosen\_node \rightarrow$ **Nodes_List**;
18: **end if**
19: **return** Nodes_List

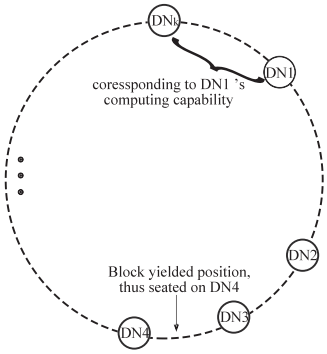


Fig. 4. The candidate nodes form an ActCap ring, where the region between a data node and its predecessor reflects the computing capability of the corresponding node.

its predecessor on the ring represents the computing capability of that data node. For each data block, a position on the ring is calculated firstly by hashing its id. The data node with its region containing that position is then selected to store the corresponding data block.

IV. Evaluation

We implement ActCap by modifying HDFS (version 2.2.0) [16]. In each data node we run a daemon to collect information about data blocks as described in III. In the name node, we calculate capabilities of data nodes with the collected data and run the capability-aware data placement algorithm. Capability calculation is done every 5 minutes. We use YARN (version 2.2.0) to execute the benchmarks on a heterogenous cluster.

A. Platform and Settings

The cluster used for experiments has 24 nodes, 4 big ones and 20 small ones, all running CentOS 6.4. The big node has a Quad-Core Xeon X3220 CPU (6MB L2 Cache, 2.50GHz), 8GB DDR3 RAM, and a 1TB SATA hard disk. The small node has a Xeon E5504 CPU (1MB L2 Cache, 2.00GHz, and deliberately configured to use only one core during the experiments), 4GB DDR3 RAM, and a 500GB SATA hard disk.

TABLE II. Benchmark characteristics

Benchmark	Input size(GB)	Data source	#Maps & #Reduces	Shuffle size(GB)	Execution time on Hadoop(s)
grep	32	wikipedia	514 & 40	$5.1*10^{-6}$	565
histogram-ratings	30	netflix data	480 & 40	$6.3*10^{-5}$	819
histogram-movies	30	netflix data	480 & 40	$6.8*10^{-5}$	621
classification	30	netflix data	480 & 40	$7.9*10^{-3}$	4060
word-count	32	wikipedia	514 & 40	0.24	1152
inverted-index	32	wikipedia	514 & 40	0.27	1664
term-vector	32	wikipedia	514 & 40	0.29	1850
sequence-count	32	wikipedia	480 & 40	0.55	1597
k-means	30	netflix data	480 & 4	26.48	5158
self-join	30	puma	480 & 40	26.89	1169
adjacency-list	30	puma	480 & 40	29.38	1727
ranked-inverted-count	40	puma	640 & 40	42.45	1801
tera-sort	20	puma	320 & 40	21.31	1134

Each node is equipped with a Gigabit Ethernet NIC (Network Interface Controller) that connects to a Gigabit Ethernet switch, thus resulting in a per-node bisection bandwidth of 1Gbps or so. Since this value is much higher than that available in typical large-scale clusters with thousands of nodes, which is about 50Mbps as pointed out in [30], we divide the cluster into six sub-clusters and configure the network environment to make the bandwidth between sub-clusters be of 50 Mbps as did in Tarazu [2]. For these six sub-clusters, two of them have two big nodes each and the other four have five small nodes each. We do nothing to the network within a sub-cluster.

B. Benchmarks

We use 13 benchmarks released in the PUMA suite [3], covering all cases of shuffle-light, shuffle-medium, and shuffle-heavy. Table II summarizes the characteristics of these benchmarks in terms of input data size, data source, the number of Map/Reduce tasks, shuffle size, and execution time on Hadoop.

Shuffle-light cases have very little data transfer in shuffle phase, including *grep*, *histogram-ratings*, *histogram-movies*, and *classification*. Shuffle-heavy cases, the shuffle data size of which is very large (as shown in Table II, almost the same volume as the input data size), include *k-means*, *self-join*, *adjacency-list*, *ranked-inverted-count*, and *tera-sort*. The shuffle data size of shuffle-medium cases is between shuffle-light and shuffle-heavy, including *word-count*, *inverted-index*, *term-vector*, and *sequence-count*.

C. Experimental Results

In our experiments, we deploy two clients to submit jobs to the cluster using the *FairScheduler* scheme, which fairly shares the cluster resources. Each client randomly submits jobs to the cluster. Multiple jobs run together on the cluster in order to better emulate the real-world situation (i.e., jobs are running with different background workloads) and to avoid the case that jobs are executed in special orders. Due to the interference of jobs concurrently running on the cluster, the execution time and the percentage of transferred blocks vary with time. For example, Fig. 5 shows the experimental result of a *Sequence-Count* job. We use the widely-used four quartile method [18] to get the average value, with the result shown in Fig. 6.

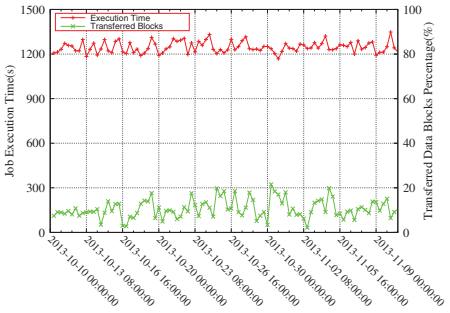


Fig. 5. Job execution time and the percentage of transferred blocks of a *Sequence-Count* job.

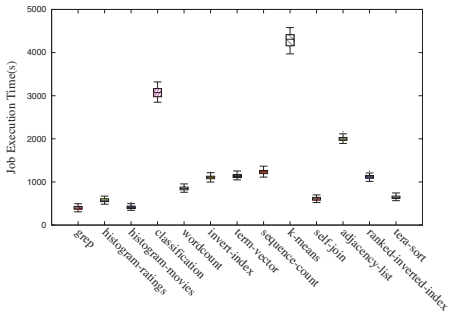


Fig. 6. Average job execution time got with the widely-used four quartile method.

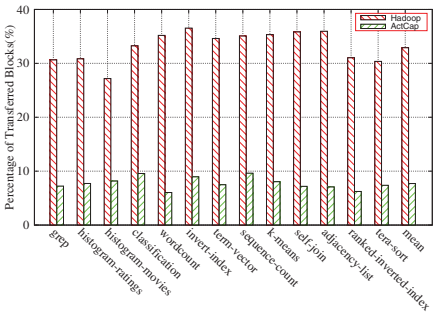


Fig. 7. Percentage of transferred blocks comparison between Hadoop and ActCap.

TABLE III. PARAMETER SETTINGS FOR YARN AND HDFS

Parameter	Description	Value
yarn.nodemanager.resource.memory-mb	The memory could be allocated for YARN	4608 for Big, 1536 for Small
yarn.scheduler.minimum-allocation-mb	The minimum memory allocated for every task	512
yarn.scheduler.maximum-allocation-mb	The maximum memory allocated for every task	1024
yarn.app.mapreduce.am.resource.mb	The memory allocated for the MR AppMaster	512
yarn.scheduler.minimum-allocation-vcores	The minimum vcore number for every task	1
yarn.scheduler.maximum-allocation-vcores	The maximum vcore number for every task	1
yarn.resourcemanager.scheduler.class	Task Scheduler schema	FairScheduler
dfs.blocksize	Block size of HDFS	67108864
dfs.replica	HDFS replica number	3

1) Comparison with Hadoop: For a fair comparison with Hadoop, we tune Hadoop for hardware heterogeneity by customizing the number of processors and memory capacity per node for each node type so as to account for the differences in core numbers and memory capacity. Since YARN can adapt the number of simultaneously running tasks according to the available CPU and memory resources as well as the configuration parameters and the big node in our cluster has four CPU cores whereas the small node has one CPU core, we configure the parameters to ensure a big node could run four tasks simultaneously, and a small node could run one task. In detail, we configure *1024*4+512=4608MB* memory for Hadoop on big node, and *1024*1+512=1536MB* memory for Hadoop on small node, where 1024MB memory is used for each task and 512MB memory for the *Application Master*. The key parameters are shown in Table III. For those parameters not listed in the table, we use the default values.

As shown in Fig. 7, the percentage of transferred blocks of Hadoop ranges from 27.18% to 35.96%, whereas that of ActCap ranges from 6.04% to 9.63%. In a word, ActCap could significantly reduce data transfer for all the jobs. The reduction of transferred data implies less network consumption of and competition (with *Shuffle*) for bisection network bandwidth and thus decreases the job execution time significantly.

Fig. 8 shows the normalized execution time of the Map and the Reduce phases. Since the execution time fluctuates with time, we show the result of one specially selected case, whose job execution time is closest to the average value. For the shuffle-light jobs *grep*, *histogram-movies*, *histogram-ratings*, and *classification*, although they have small shuffle volume and short execution time, the inter-node block transfer time occupies a very large part of the whole time. Thus, the reduction of transferred data significantly reduces the Map time as well as the whole execution time — the average execution time of the four jobs on ActCap is 65.6% of that on Hadoop. For the shuffle-medium jobs *word-count*, *inverted-index*, *term-vector*, and *sequence-count*, reducing data transfer also shortens job execution time. Their average execution time on ActCap is about 67.3% of that on Hadoop. For the shuffle-heavy jobs *k-means*, *self-join*, *adjacency-list*, *ranked-inverted-index*, and *tera-sort*, the network consumption by data transfer has a great influence on the Shuffle phase and thus, reducing data transfer can shorten job execution time drastically — the average time on ActCap is 68.6% of that on Hadoop. Among the shuffle-heavy jobs, *tera-sort* and *self-join* get the best optimization, and *k-means* gets the least performance improvement because its Reduce phase takes as long as 62.9% of the whole job execution time.

2) Comparison with Tarazu: We implement Tarazu and use the same parameters as recommended. To ensure fairness, we only run the 11 benchmarks mentioned in Tarazu, with the results shown in Fig. 8. Except for *k-means*, job execution time of ActCap is smaller than that of Tarazu for ten of the eleven benchmarks, with speedups of ActCap over Tarazu ranging from 3.8% to 33.7%, and an average speedup of 9.8%. For the job *k-means*, the normalized execution time of ActCap is 0.873, while that of Tarazu is 0.546. The reasons for this are: (i) the Reduce phase of *k-means* takes a large part (62.9% of the whole job time), the optimization is not as good as that of Reduce-light jobs; (ii) the *k-means* benchmark sets the number of Reduce tasks to 4 in order to cluster the input data into 4 groups. When running on Hadoop and ActCap, Reduce tasks are scheduled using a first-come-first-served scheme and have a high chance to run on small nodes because there are more small nodes. As a result, little performance gain is got. On the contrary, Tarazu can reduce the Reduce time ratio from 62.9% to 27.8% by deliberately scheduling all Reduce tasks to the big nodes. Therefore, it can achieve better results in the term of total job execution time.

3) Comparison with Other Data Placement Approaches: Xie et al.[31] proposed a similar data placement approach but using an indirect sampling-based way to measure node capability. Such an approach may work well on exclusively used clusters. However, for clusters shared by many applications, the node capability varies constantly due to background workloads and other concurrent jobs. In other words, Xie's approach is

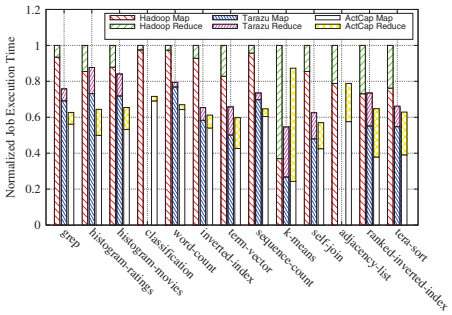


Fig. 8. Job execution time comparison between Hadoop, Tarazu and ActCap.

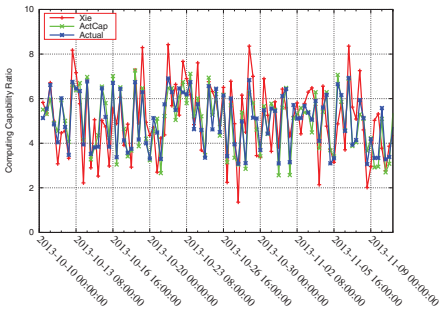


Fig. 9. Comparison of the predicting accuracy via ActCap and Xie's approach.

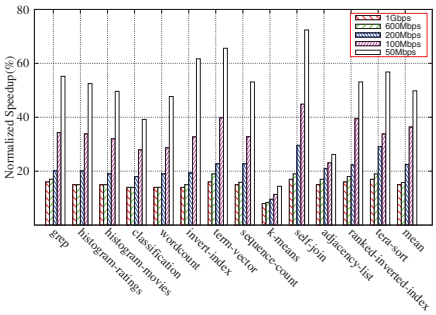


Fig. 10. The impact of network bandwidth on the usability of ActCap.

not *accurate* enough to reflect the change of node processing capability and therefore, it can only get limited performance gain. To illustrate this, we choose two nodes, one big and one small, and get their computing capabilities during the whole experiment process using ActCap and Xie's approach. Fig. 9 shows the result. It is easy to see that the computing capability ratio predicted by ActCap is much closer to the actual value than that predicted by Xie's approach.

We also compare ActCap with other approaches such as CPU Ratio and One-ActCap. CPU Ratio sets the skew factor on the basis of CPU power. For example, since a big node has four CPU cores running at 2.5GHz and a small node has one CPU core running at 2.0GHz, the skew factor is set to $4 * 2.5/2.0 = 5$. One-ActCap predicts computing capabilities using ActCap only once. Fig. 11 shows the comparison results in terms of the data transfer percentage and the speedup. Obviously, ActCap gets the smallest data transfer percentage and the highest speedup.

4) The impact of network bandwidth: We show the impact of network bandwidth on ActCap by specifying various bisection bandwidths between sub-clusters. As shown in Fig. 10, ActCap works well under all network settings and can get an average speedup of 15%, 15.8%, 22.5%, 36.4%, and 49.8% respectively.

5) The adaptability of ActCap to heterogeneity: We examine this by four cluster configurations, that is, 4 Big+20 Small, 2 Big+20 Small, 4 Big+10 Small, and 4 Big+2 Small. As shown in Fig. 12, ActCap works well on all these four clusters in the term of job speedup over standard Hadoop.

V. Related Work

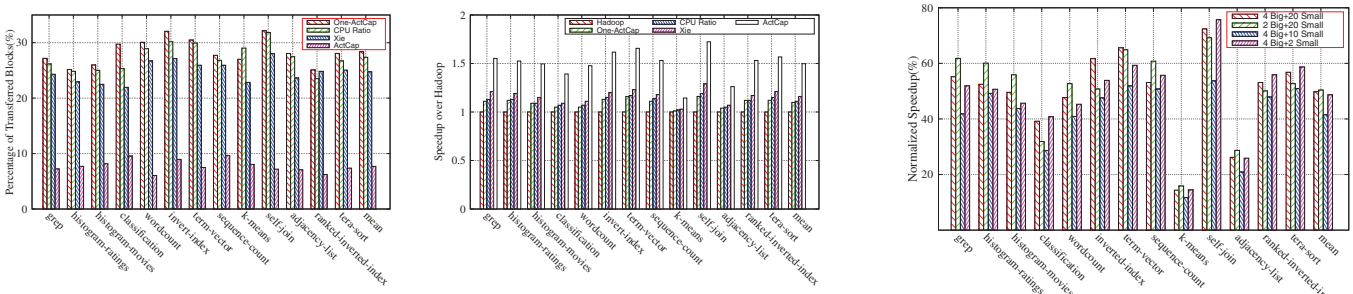
MapReduce Implementations. Due to its high impact, MapReduce, after the original implementation by Google, has been implemented by the Apache open-source community [16] and ported to computing environments other than traditional clusters, for example, graphics processors [13] and mobile systems [11]. Besides, other MapReduce-like systems [8] and high-level facilities [29] were proposed. Also, MapReduce has expanded its application from batch processing to iterative computation [28] [34] and stream processing [25].

MapReduce Optimization via Data Placement. Approaches in this category try to boost the speed of data loading. HadoopDB [1] and Hadoop++ [10], aiming at database query, achieve the purpose by co-grouping related input data. RCFile [17] provides a new data placement structure which ensures that data in the same row of conventional database systems is located in the same node (and thus reduces data transfer over the network). CoHadoop [12] deals with the problem from a different way—it provides a mechanism for applications to specify where to store the data. Xie et al. [31] suggested the same idea as ours, that is, distributing data to nodes according to their capabilities. However, they exploited a static method to determine the computing capability of each node. As shown in Section IV-C3, the performance gain of their approach is lower than that of ours, for their approach does not taken into account the change of node processing capability in a shared environment. Besides the above work, in-memory structures [28][34] and data caching method [6] were also suggested to eliminate the bottleneck encountered during loading data from hard disks. Due to space limitation and in consideration of the scope of this paper, we will stop here without detailing them.

MapReduce Optimization via Task Scheduling. Typical examples are Delay Scheduling [33], LATE [35], and Tarazu [2], where LATE and Tarazu were specially designed for heterogeneous environments. Delay Scheduling tries to maintain data locality by later decision—when a job to be scheduled next according to fairness cannot launch a local task, the scheduler will wait for a while and let other jobs launch tasks instead. In this way, nearly optimal data locality can be achieved and as a result, task throughput is increased. LATE found that the poor performance of MapReduce on heterogeneous clusters arose from the breaking down of underlying assumptions with the built-in scheduling mechanisms. To deal with the problem, it proposed better techniques for identifying, prioritizing, and scheduling backup copies of straggler tasks. Tarazu [2] identified the key reason for the poor performance of MapReduce on heterogeneous clusters as the competition for bisection network bandwidth between remote tasks and shuffle phase. To deal with the problem, it introduced such components as Communication-Aware Load Balancing of Map Computation, Communication-Aware Scheduling of Map computation, and Predictive Load Balancing of Reduce Computation to respectively prevent shuffle-critical tasks stealing, interleave remote tasks with local ones, and skew the intermediate key distribution among the Reduce tasks. While we believe these approaches can function, we also think the performance improvement would be limited, for they do not touch the essentials of the problem as we do.

VI. Discussion

This section will clarify some concerns about ActCap.



(a) Block transfer percentage (b) Speedup over Hadoop

Fig. 11. Block transfer percentage and speedup comparison between various data placement solutions.

Fig. 12. The adaptability of ActCap to heterogeneity.

A. Usability of ActCap

1) Heterogeneity is pervasive: Heterogeneity comes from many reasons. In a cluster, even for machines of the same hardware specifications, they may show different processing capabilities due to, for example, resources sharing and background workloads. As did in [2], we in this paper focus on heterogeneity with general-purpose CPU architectures, which is considered to be the key cause of performance degradation despite better straggler management policies like LATE [35]. Indeed, architectural heterogeneity has drawn much attention since the early 1990s [22] [4]. Today, it becomes even more popular for many reasons. However, heterogeneity comes with cost [14]. This lays a good basis for work like ActCap, Tarazu [2], and LATE [35].

2) Network over-subscription will last: Network over-subscription is an implicit assumption during MapReduce design. It is due to network over-subscription that the MapReduce frameworks adopt the philosophy of "moving computation to data" and highlight *data locality* as much as possible. Though data center networks change greatly, it is not an easy task to make the bisection bandwidth scale up with the number of nodes [30] and no evidence shows that network over-subscription will disappear. In our opinion, network bandwidth will remain a scarce global resource at least in the near future because data processing at the scale of TB or PB gets more common along with the exponential growth of data and resources today are shared by more users due to the emergence of new technology (e.g., virtualization) and application mode (e.g., cloud computing). The lasting network over-subscription makes our work about ActCap remain significant.

3) ActCap can also function to in-memory and iterative MapReduce: With the price of memory getting cheaper and cheaper comes into being the concept of in-memory computing. As a result, many information technology companies (e.g., GridGain[1], ScaleOut Software[2], and Hazelcast[3]) have released in-memory MapReduce solutions. Also new ways (e.g., Spark [34] and M3R [28]) are suggested to do iterative in-memory MapReduce computation. Due to the advantages of memory over hard disk as well as other optimizations, these systems can achieve better, sometimes surprising, performance. Although ActCap is designed for disk-based MapReduce computing, here we argue that it can also benefit these systems in a heterogeneous environment because data in these systems is also evenly distributed among many nodes (but in memory instead of on disk) at the beginning and inter-node data transfer is inevitable when both big and small nodes present.

[1]http://www.gridgain.com/products/in-memory-hadoop-accelerator/

[2]http://www.scaleoutsoftware.com/

[3]http://www.hazelcast.com/

B. Limitation of ActCap

ActCap in its current version can still be improved. First of all, we pay no attention to skewed data popularity. However, as we have pointed out in Section II, the solution in [5] can be easily incorporated into ActCap. Indeed, we have provided support for larger number of replicas in Algorithm 1. Next, as pointed out in Section II, ActCap does not consider data skew in the Reduce phase as YARN can handle it relatively well. In spite of the fact, we can get extra performance gain if we could incorporate better skew-mitigating method into YARN. Finally, ActCap assumes data comes into the system continuously. As time passes, the effective computing capabilities of nodes might change accordingly, making the once optimal placement become sub-optimal. For this issue we argue it is true only for some of existing data and we can eliminate the effect by re-distributing that portion of data at a much longer time scale.

In the end, ActCap will not work on those clusters whose disks always operate close to their capacities because there is little room left for ActCap to do biased data distribution. In our opinion, MapReduce clusters in the real world usually do not operate that way. In addition, after the benefit of uneven data distribution were generally accepted, faster servers with bigger disks will appear, making this limitation no longer exist.

VII. Conclusion

We have described ActCap, a solution that tries to improve MapReduce performance on ever-growing heterogeneous clusters by node-capability-aware data placement. At the core of ActCap are the Markov chain models for on-the-fly determination of the computing capabilities of nodes in a cluster and the algorithm to do capability-aware data placement. Unlike the default policy of Hadoop (or more precisely, HDFS) that data is distributed (almost) evenly among nodes regardless of their capabilities, node-capability-aware data placement essentially leads to non-uniform data distribution—those nodes of higher capabilities will have more data. ActCap has been implemented in Hadoop and evaluated on a 24-node heterogeneous cluster with 13 benchmarks. The experimental results show that, compared with Hadoop, it can reduce the percentage of inter-node data transfer from 32.9% to 7.7%, which means an average speedup of 49.8% over Hadoop. In addition, it can achieve an average speedup of 9.8% over Tarazu, the latest work about MapReduce optimization on heterogeneous clusters.

ACKNOWLEDGMENT

This work is co-supported by Natural Science Foundation of China (61170210, 61433008, U1435216), National High-Tech R&D (863) Program of China (2012AA012600), and National Science and Technology Major Project of China (2013zx01039-002-002). We would like to thank the anonymous reviewers for their comments and suggestions.

REFERENCES

[1] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin, "Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads," *Proceedings of the VLDB Endowment, VLDB'09*, vol. 2, no. 1, pp. 922–933, 2009.

[2] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. Vijaykumar, "Tarazu: optimizing mapreduce on heterogeneous clusters," in *In Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'12*. ACM, 2012, pp. 61–74.

[3] F. Ahmad, S. Lee, M. Thottethodi, and T. Vijaykumar, "Puma: Purdue mapreduce benchmarks suite," 2012, http://web.ics.purdue.edu/ fahmad/benchmarks.htm.

[4] V. Almeida, I. Vasconcelos, J. N. C. Árabe, and D. A. Menascé, "Using random task graphs to investigate the potential benefits of heterogeneity in parallel systems," in *Proceedings of the 1992 ACM/IEEE conference on Supercomputing, Supercomputing'92*. IEEE Computer Society Press, 1992, pp. 683–691.

[5] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris, "Scarlett: coping with skewed content popularity in mapreduce clusters," in *Proceedings of the sixth conference on Computer systems*. ACM, 2011, pp. 287–300.

[6] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "Pacman: Coordinated memory caching for parallel jobs," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, NSDI'12*. USENIX, 2012, pp. 20–20.

[7] J. C. Anjos, I. Carrera, W. Kolberg, A. L. Tibola, L. B. Arantes, and C. R. Geyer, "Mra++: Scheduling and data placement on mapreduce for heterogeneous environments," *Future Generation Computer Systems*, vol. 42, pp. 22–35, 2015.

[8] P. Costa, A. Donnelly, A. Rowstron, and G. OShea, "Camdoop: Exploiting in-network aggregation for big data applications," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, NSDI'12*. USENIX, 2012, pp. 3–3.

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

[10] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad, "Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing)," *Proceedings of the VLDB Endowment, VLDB'10*, vol. 3, no. 1-2, pp. 515–529, 2010.

[11] A. Dou, V. Kalogeraki, D. Gunopulos, T. Mielikainen, and V. H. Tuulos, "Misco: a mapreduce framework for mobile systems," in *Proceedings of the 3rd International Conference on PErvasive Technologies Related to Assistive Environments*. ACM, 2010, p. 32.

[12] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson, "Cohadoop: flexible data placement and its exploitation in hadoop," *Proceedings of the VLDB Endowment, VLDB'11*, vol. 4, no. 9, pp. 575–585, 2011.

[13] W. Fang, B. He, Q. Luo, and N. K. Govindaraju, "Mars: Accelerating mapreduce with graphics processors," *IEEE Transactions on Parallel and Distributed Systems, TPDS'11*, vol. 22, pp. 608–620, 2011.

[14] P. B. Godfrey and R. M. Karp, "On the price of heterogeneity in parallel systems," *Theory of Computing Systems*, vol. 45, no. 2, pp. 280–301, 2009.

[15] B. Gufler, N. Augsten, A. Reiser, and A. Kemper, "Load balancing in mapreduce based on scalable cardinality estimates," in *2012 IEEE 28th International Conference on Data Engineering, ICDE'12*. IEEE, 2012, pp. 522–533.

[16] A. Hadoop, "Hadoop," 2013, http://hadoop.apache.org.

[17] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu, "Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems," in *2011 IEEE 27th International Conference on Data Engineering, ICDE'11*. IEEE, 2011, pp. 1199–1208.

[18] R. J. Hyndman and Y. Fan, "Sample quantiles in statistical packages," *The American Statistician*, vol. 50, no. 4, pp. 361–365, 1996.

[19] R. K. Joshi and D. J. Ram, "Anonymous remote computing: a paradigm for parallel programming on interconnected workstations," *IEEE Transactions on Software Engineering, TSE'99*, vol. 25, no. 1, pp. 75–90, 1999.

[20] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skewtune: mitigating skew in mapreduce applications," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, SIGMOD'12*. ACM, 2012, pp. 25–36.

[21] Y. Le, J. Liu, E. Funda, and D. Wang, "Online load balancing for mapreduce with skewed data input," in *2014 IEEE International Conference on Computer Communications, INFOCOM'14*. IEEE, 2014, pp. 2004–2012.

[22] D. Menascé and V. Almeida, "Cost-performance analysis of heterogeneity in supercomputer architectures," in *Proceedings of the 1990 ACM/IEEE conference on Supercomputing, Supercomputing'90*. IEEE Computer Society Press, 1990, pp. 169–177.

[23] N. Mi, G. Casale, L. Cherkasova, and E. Smirni, "Injecting realistic burstiness to a traditional client-server benchmark," in *Proceedings of the 6th International Conference on Autonomic Computing, ICAC'09*. ACM, 2009, pp. 149–158.

[24] A. K. Mishra, J. L. Hellerstein, W. Cirne, and C. R. Das, "Towards characterizing cloud backend workloads: insights from google compute clusters," *ACM SIGMETRICS Performance Evaluation Review, SIGMETRICS'10*, vol. 37, no. 4, pp. 34–41, 2010.

[25] nathanmarz, "Storm," https://github.com/nathanmarz/storm.

[26] S. M. Ross, *Introduction to probability models, Tenth Edition*. Access Online via Elsevier, 2011.

[27] B. Sharma, V. Chudnovsky, J. L. Hellerstein, R. Rifaat, and C. R. Das, "Modeling and synthesizing task placement constraints in google compute clusters," in *Proceedings of the 2nd ACM Symposium on Cloud Computing, SoCC'11*. ACM, 2011, p. 3.

[28] A. Shinnar, D. Cunningham, V. Saraswat, and B. Herta, "M3r: increased performance for in-memory hadoop jobs," *Proceedings of the VLDB Endowment, VLDB'12*, vol. 5, no. 12, pp. 1736–1747, 2012.

[29] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009.

[30] A. Vahdat, M. Al-Fares, N. Farrington, R. N. Mysore, G. Porter, and S. Radhakrishnan, "Scale-out networking in the data center," *Micro, IEEE*, vol. 30, no. 4, pp. 29–41, 2010.

[31] J. Xie, S. Yin, X. Ruan, Z. Ding, Y. Tian, J. Majors, A. Manzanares, and X. Qin, "Improving mapreduce performance through data placement in heterogeneous hadoop clusters," in *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, IPDPSW'10*. IEEE, 2010, pp. 1–9.

[32] A. YARN, "Yarn," http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[33] 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, EuroSys'10*. ACM, 2010, pp. 265–278.

[34] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10*. USENIX, 2010, pp. 10–10.

[35] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments." in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI'08*. USENIX, 2008, p. 7.