2017

EA2S2: An Efficient Application-Aware Storage System for Big Data Processing in Heterogeneous Clusters

Teng Wang
Jiayin Wang
Son Nam Nguyen, *University of Massachusetts Boston*
ZHENGYU YANG, *Northeastern University*
Ningfang Mi, et al.

EA^{2}S^2 : An Efficient Application-Aware Storage System for Big Data Processing in Heterogeneous Clusters

Teng Wang[*], Jiayin Wang[*], Son Nam Nguyen[*], Zhengyu Yang[†], Ningfang Mi[†], and Bo Sheng[*]
[*]Department of Computer Science, University of Massachusetts Boston, 100 Morrissey Boulevard, Boston, MA 02125
[†]Department of Electrical and Computer Engineering, Northeastern University, 360 Huntington Ave., Boston, MA 02115

Abstract—**Big data processing frameworks such as Hadoop have been widely adopted to process a large volume of data. A lot of prior work has focused on the allocation of resources and the execution order of jobs/tasks to improve the performance in a homogeneous cluster. In this paper, we investigate storage layer design in a heterogeneous system considering a new type of bundled jobs where the input data and associated application jobs are submitted in a bundle. Our goal is to break the barrier between resource management and the underlying storage layer, and improve data locality, an important performance factor for resource management, from the aspect of storage system. We develop a sampling-based randomized algorithm for the network file system to determine the placement of input data blocks. The main idea is to query a selected set of candidate nodes, and estimate their workload at run time combining centralized and per-node information. The node with the smallest workload is selected to host the data block. Our evaluation is based with system implementation and comprehensive experiments on NSF CloudLab platforms. We have also conducted simulation for large-scale clusters. The results show significant performance improvements in terms of execution time and data locality.**

I. Introduction

The current big data processing platforms adopt the 'scale-out' solution, where a cluster of servers collaborate to accomplish big data applications. The application job and the corresponding input data are split into small tasks and associated data blocks respectively. Each task can be executed on any of the cluster nodes that can allocate the task's demanded computing resources. In such platforms, the storage layer is usually managed by a network file system where data blocks are distributed across the cluster with replicas. Data locality is extremely important to the system performance when executing the big data application because the current platforms only prefer 'move-compute-to-data' paradigm, but do not enforce it. It is possible that a task is executed on a server that does not host the task's input data block. In this case, the input data block will have to be transferred from another cluster node incurring a transmission overhead into the task execution.

The data locality issue has been explored in a lot of prior work, and new enhancement schemes have been proposed to increase the chance of executing a task on the server hosting its input data. However, the performance improvement from these existing solutions is limited by dynamic run-time factors in practice, especially in a heterogeneous cluster where each server has different hardware and software configurations. The fundamental cause that hinders the optimization on data locality is the separation of job execution and input data uploading processes. In all the prior work, data locality is only considered during the job execution by the job/task scheduler assuming that the input data have been uploaded into the network file system in a separate process. Without considering the initial data block placement during the uploading process, the existing solutions cannot optimize the data locality with various job workloads and cluster configurations. In addition, the prior work considers that the system performance of execution time or makespan refers to the job execution process excluding the time spent in uploading the input data. In practice, however, there are many *bundled jobs* submitted to a processing cluster that include both application jobs and the reference of the input data that needs to be transferred to the network file system. For this type of bundled jobs, the performance should include the time spent in both uploading the input file and executing the associated jobs.

In this paper, we target on the execution of bundled jobs in a heterogeneous cluster, and present a new storage layer that efficiently places data blocks across the cluster to improve the data locality. When dispatching data blocks of the input data, our solution considers the associated application jobs and estimates the future executions. We apply randomized algorithms to simplify and speed up the solution. Essentially, our algorithm queries a small set of randomly selected candidate nodes for hosting a data block to estimate their run-time workload. The data block will be uploaded to the node with the smallest workload aiming to balance the workload of all the nodes and thus improve the data locality. We implement our solution in Hadoop YARN platform and evaluate it with experiments and simulation. The results are highly supportive with significant performance improvements.

The rest of this paper is organized as follows: Section II reviews the prior work, and Section III introduces the background information and main motivations. The details of our solution are presented in Section IV. We examine the performance in Section V, and finally conclude in Section VI.

II. Related Work

Scheduling is a significant direction in Big Data processing systems. During various computing systems, Hadoop YARN [1] is well used in both academia and industry. In

This work was partially supported by National Science Foundation grant CNS-1552525, National Science Foundation Career Award CNS-1452751, and AFOSR grant FA9550-14-1-0160.

Hadoop YARN, Fair Scheduler [2], Capacity Scheduler [3] , and the latest DRF Scheduler [4] are embedded in the native Hadoop YARN system to ensure each job can obtain a proper share of the available resources. To improve the performance of the computing systems, many scheduling works focus on different directions. Some major directions and related works are introduced as follows.

Job Aware Scheduling: Some scheduling algorithms take the job characteristics into consideration. ARIA [5] and Deadline-constraint Scheduler [6] allocate appropriate resources to jobs to meet the predefined deadline. Sparrow [7] focuses on the scheduling problems with large amount of small jobs.

Resource Aware Scheduling: Improving resource utilization of the cluster is an important direction in the scheduling. In this area, RAS [8] aims to improve resource utilization across machines and meet jobs completion deadline. A fine-grained resource scheduling, Haste [9], focuses on improving resource utilization by leveraging the information of requested resources, resource capacities, and dependency between tasks. In addition, FRESH [10] and OMO [11] have developed dynamic resource management schemes depending on the various workloads of different jobs. However, these works cannot perform well in the heterogeneous environment.

Heterogeneity Aware Scheduling: Heterogeneous environment is normality in practice because of different hardware and software settings in each node of the cluster. In this direction, Tetris [12] is a cluster scheduler which packs tasks to machines based on their multiple resource requirements. In addition, LATE [13], Hopper [14], Grass [15] and eSplash [16] are proposed to stop unnecessary speculative executions in order to improve the performance in heterogeneous clusters.

Data Locality Aware Scheduling: Some task scheduling mechanisms focus on optimizing the locality of jobs' input data in the distributed file system. NKS [17] and [18] propose data placement algorithms in the homogeneous environment. Some recent works [19]–[21] distribute input data in heterogeneous clusters according to the disk capacity of each node. However, the various resource capacities of each node, and the resource demands of each task are not considered comprehensively in these approaches.

Inspired by the preceding works, we develop EA^{2}S^2, an application-aware storage system for the heterogeneous clusters, to improve the system performance by optimizing the input data placement. Based on the resource demands of each task, capacities and processing capabilities of multiple resources in each node, EA^{2}S^{2}can efficiently distribute data blocks and reduce the makespan of a batch of jobs. EA^{2}S^{2}is implemented in Hadoop YARN and also can be integrated into other cluster computing systems.

III. Background and Motivation

In this section, we present our target environment settings and the background information about the representative Hadoop YARN platforms. We also introduce the policies and issues in the existing storage layer of Hadoop, and bring up our motivation on the efficient storage design.

A. Cluster Setting and Job Size

In this paper, our target computing environment is a large scale heterogeneous cluster consisting of hundreds or thousands servers. A typical Hadoop cluster consists of a master node and multiple slave nodes. In Hadoop framework, each slave node declares it resource capacity in terms of the number of CPU cores and the size of its memory. When submitting a MapReduce job, the user specifies the resource demands of each type of tasks (map and reduce tasks). The input file of a job is split into fixed-sized blocks, and each map task processes one block of data. The output data generated by map tasks serve as the input data of reduce tasks.

The cluster Resource Manager (RM) running on the master node, therefore, dispatches each task to one of the servers for execution by allocating a resource container there with the demanded CPU and memory resources. For any server, the total amount of resources occupied by all the containers running on it cannot exceed its declared resource capacity. In addition, we consider that all the servers in the cluster may show heterogeneous run-time performance due to the following factors. First, the hardware and software configurations on all the servers may not be identical. Second, there might be other processing frameworks or services running in the same cluster interfering the performance of Hadoop at the runtime.

For the job workload, this paper is focused on relatively small jobs with input data size in the scale of GBs. It matches the realistic workload mentioned in the prior work [7] and observed in our experimental results. We have conducted the TPC-DS benchmark [22] on a Hadoop cluster built with Apache Hive [23]. A 100GB database is created, and 65 SQL queries in TPC-DS are tested yielding 151 MapReduce jobs. The average value of the input file size is 5.5GB, and 80% jobs' input files are smaller than 8GB.

B. Data Locality

The focus of this paper is to improve the data locality which is an important performance factor in a Hadoop system. This subsection briefly reviews the background and presents the issues of the traditional system.

A Hadoop cluster is built upon a network file system (HDFS) that manages the placement of all the data blocks (with replicas). Hadoop defines three types of data locality for map tasks: node-local, rack-local, and off-switch, referring to different cases of the location of the resource container and input data block of a task. 'Node-local' indicates that the container for executing the task is allocated on a node that hosts the corresponding input data block. In the case of 'rack-local', the node that is about to execute the map task does not host the input data block, but there is a copy of the data in the same rack, and can be transferred to the executing node. In 'off-switch', the data block has to be transferred from another rack for the execution. In either 'rack-local' or 'off-switch', a network overhead is incurred for transferring the data block to the node that allocates a container to execute the task. It is not negligible considering the size of a data block, short execution time of a map task, and heavy traffic loads in realistic clusters. Table I shows a comparison of the average execution times of node-local and rack-local map tasks in four different MapReduce benchmark jobs.

	word count	terasort	word mean	grep
Node local (NL)	35.27s	9.67s	19.53s	9.44s
Rack local (RL)	41.08s	25.56s	31.68s	21.82s

TABLE I: Comparison of execution times of NL and RL tasks

In this paper, we use 'non-node-local' and 'rack-local' interchangeably to represent the locality of rack-local and off-switch in which case the input data has to be migrated to the computing node. The overall impact of rack-local tasks also depends on the frequency of their occurrences. The native Hadoop YARN uses a random data block placement strategy where each data block is hosted by a randomly selected node. Unfortunately, it does yield a considerable portion of rack-local tasks. Consider a cluster of n servers and a batch of jobs each with m blocks ($m \ll n$). Assume the HDFS keeps k replicas of each data block, and all jobs map task resource demands are the same. Once a node finishes a task and releases the resources, the cluster RM will choose to serve a job J and assign one of its map task to the node. Assume there are $j \in [1, m]$ pending tasks, we use $P(j)$ to represent the probability that a server hosts at least one input data block of these j pending tasks. $P(j)$ can be derived as $P(j) = 1 - (1 - \frac{k}{n})^j$, where $\frac{k}{n}$ is the probability that a node is chosen to host one replica of one of these j input data blocks. Thus, the expected number of node-local tasks is $\sum_{j \in [1,m]} P(j)$ and the complementary subset will be rack-local tasks. Fig. 1 shows the analytical results of the rack-local tasks in a cluster of 100 servers ($n = 100$) with varying numbers of input data blocks and replicas. Apparently, rack-local tasks take a large potion of all the tasks when $m \ll n$. The same observation is also confirmed in our experiments where a 10-node cluster processes 40 MapReduce jobs each with 4 input data blocks, and the HDFS keeps only one replica of every block. Out of 160 map tasks, 82% are rack-local tasks.

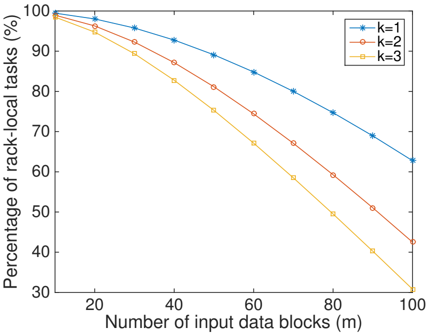

Fig. 1: Expected percentage of rack-local tasks with varying input data size and three replica settings ($n = 100$)

C. Bundled Jobs

In a traditional Hadoop system, executing a job and uploading its input data are separately handled by Resource Manager (RM) and the network file system (HDFS) respectively. Data locality is preferred by RM, but not enforced. As we have shown with analysis and experiments, the ratio of node-local tasks is quite low under our target workload setting. In this paper, we aim to improve the data locality in the process of uploading the input data by appropriately arranging the initial placement of the data blocks.

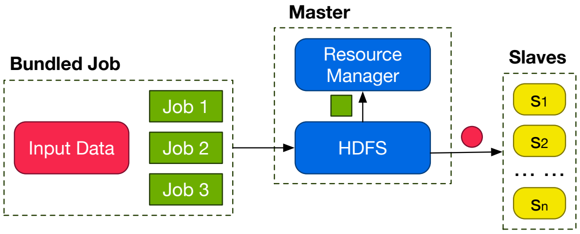

Fig. 2: Illustration of processing a bundled job

We consider the Hadoop cluster is running a type of bundled jobs, where the input data and its processing jobs are submitted together as a bundle. The input data can be uploaded by the user, or from external storage sites such as Amazon S3. In this setting, the user considers the cluster as a computing site, but not for long-term or permanent data storage. After the jobs are finished, the input data will be deleted from the HDFS. Fig. 2 illustrates the basic process. The main difference compared to the traditional Hadoop is that the HDFS is aware of the jobs associated with the input data, and thus could use this information to better dispatcth the data blocks in the cluster.

Therefore, our problem is formulated as follows. The objective is develop an efficient data block placement scheme in HDFS to help minimize the makespan of a batch of bundled jobs executed by a heterogenous cluster.

Cluster setting: We consider a heterogeneous cluster of n servers, $S = \{S_1, S_2, \ldots, S_n\}$, and each server S_i is configured with resource capacity R_i.

Job setting: We assume that there are a limited set of types of jobs, each identified by a unique job ID in the system. A batch of bundled jobs are submitted to the cluster by the users. Each bundled job BJ_i includes one input data file, and a set of application jobs to process the data. We use J_{ij} to represent the j-th job in BJ_i, and we assume BJ_i's input data consists of m_i blocks (referring to m_i map tasks). We further use D_{ij} to represent the resource demands of a map task of J_{ij}.

HDFS setting: Assume the HDFS configures the data block size as B bytes, and keeps k replicas of each data block.

The following Table II is a summary of the notations that will be used in the rest of this paper.

n/S_i	number of servers in the cluster/i-th server
R_i	resource capacity of S_i
BJ_i / J_{ij}	i-th bundled job / j-th application job of BJ_i
m_i	number of data blocks of job BJ_i input data
D_{ij}	resource demands of a map task of J_{ij}
E_{ij}	avg execution time of a map task of J_{ij}
b_i / BU_i	data block i / block usage info of b_i
B/k	data block size/number of replicas in HDFS

TABLE II: Notations

IV. System Design of EA^{2}S^2

In this section, we present the details of our new storage system design. We will first introduce the main sketch of the algorithm, and then discuss about the most important component of estimating the workload. Finally, we incorporate dynamic and empirical factors into our design to further improve the performance.

A. *Main Algorithm*

Generally speaking, the fundamental cause of non-local tasks is the mismatch of each server's workload and its hosted data blocks. When a sever exhausts its hosted data blocks sooner, i.e., finishes its assigned workload faster, than other servers, it will migrate some pending tasks from other severs, and execute them as non-local tasks. In traditional Hadoop systems, while uploading input files and submitting jobs are separate, it is impossible for HDFS to consider the future workload when distributing the data blocks.

In this paper, when handling the bundled jobs, HDFS is able to estimate the workload information to place the data blocks more appropriately. However, Hadoop is a quite complex system and the run-time execution environment is highly dynamic. It is difficult to quantitatively derive the impact of non-local tasks on the overall performance of execution time. Therefore, we consider an alternative objective of balancing the *workload*, which is defined as the execution time of local tasks, among all the servers.

Intuition. In particular, we develop an **E**fficient **A**pplication-**A**ware **S**torage **S**ystem (EA^{2}S^2) for cluster-based big data platforms such as Hadoop. The main idea of our solution EA^{2}S^2 is to utilize the application information in the bundled job, and estimate the future workload of each server to determine the placement of the input data blocks. We consider that each server maintains the historical execution time of the different types of tasks assigned to the server. In order to estimate the workload of a server, the master node needs to query the server to fetch the historical records. This design of distributed structure rather than gathering all cluster information into the master node is based on the following two facts and principles. First, it has been a common practice in large scale systems that the centralized management at the master node should be as simplified as possible to reduce its workload. It is inappropriate to keep a large amount of per-node information on the master node. The Hadoop system also follows the principal having NodeManager as per-node agent and ApplicationMaster as per-job agent to mitigate the management tasks on the master node. Second, some run-time dynamic factors of each slave node may not be available at the master node. For example, the resource utilization of a salve node affects the estimation of its workload, but cannot be obtained from the master node.

Sampling-based Randomized Algorithm. In a large scale cluster, it is inefficient to query all the nodes to estimate their workload and decide where to upload an input data block to. Especially in our target workload with small job sizes, uploading input files and executing the bundled jobs do not take a long time. Thus the overhead incurred by querying a lot of slave nodes could be considerable in the whole process.

Our design is based on sampling algorithms adopting the power of two/multiple choices. For each data block, the algorithm randomly selects some nodes as candidates, and then compares the estimated workload of each candidate. The node with the smallest workload will be selected to host the data block. Algorithm 1 illustrates the detailed steps. The main structure is a loop (lines 2–11) enumerating all the input data blocks. For each of them, we use HN to represent the set of hosting nodes initialized as an empty set (line 3). The inner while loop (lines 4–9) is to select k distinct hosting nodes to place the k replicas of the data block b_j. The sampling algorithm randomly select d slave nodes from the candidate set CS and query them to estimate their workload. CS is derived by a function of HN. For the first replica, all slave nodes are candidates. But for the rest of the replicas, i.e., when HN is not empty, the candidate set may be a subset of the nodes in the cluster according to replica management policies. For example, Hadoop system prefers to distribute the replicas into different racks. In lines 7–8, the algorithm compares the estimated workload of the candidate nodes, and adds the one with the smallest workload into HN. Finally, in line 10, the data block b_j is uploaded to all the slave nodes in HN.

Algorithm 1: Uploading Input Data

```
    input : $m_i$ : number of input data blocks of $J_i$,
            $k$ : number of replicas
 1  Split input file into $m_i$ blocks, $\{b_1, b_2, \ldots, b_{m_i}\}$;
 2  for $j = 1$ to $m_i$ do
 3      $HN = \{\}$ ;                       // set of hosting nodes
 4      while $|HN| < k$ do
 5          $CS =$ SelCand ($HN$) ;         // candidate set
 6          $RS$ : Randomly select $d$ nodes from $CS$;
 7          $a = \texttt{argmin}_{x \in RS}$ EstLoad ($x$);
 8          add $S_a$ to $HN$;
 9      end
10      Upload data block $b_j$ to every node in $HN$;
11  end
```

Batch Sampling. The above Algorithm 1 illustrates the basic sketch of EA^{2}S^2. In our implementation, we enhance the design by adopting a batch sampling scheme similar to [7] to improve the efficiency. Instead of querying d candidate servers for each replica of the m_i data blocks, our algorithm queries a batch of M servers, and place all m_i blocks (each with k replicas) to a subset of these servers.

The value of M is determined by the expected number of distinct servers selected by Algorithm 1. For one round of selection, a slave node is chosen with a probability of $\frac{d}{n}$. After all $m_i \cdot k$ rounds, the probability that a server is selected in at least one rounds $1 - (1 - \frac{d}{n})^{m_i \cdot k}$. Therefore, we set

$$M = n \cdot (1 - (1 - \frac{d}{n})^{m_i \cdot k}).$$

Comparing to the basic sampling algorithm, batch sampling yields less overhead and apparently the performance is no worse than basic sampling in terms of balancing the workload.

B. Estimation of Workload

In this subsection, we present the details of the estimation of the function $EstLoad(x)$ in Algorithm 1. The workload of each node is affected by various factors, and we classify the relevant information into the following two categories for estimating the workload.

Centralized information. This category of information is static and available at the master node. It includes cluster configuration, such as resource capacity of each slave node and HDFS block size, and the run-time information that can be obtained from the management modules running on the master node, such as the currently active jobs (from ResourceManager) and data block distribution (from NameNode). We consider that retrieving this type of information yields no overhead.

Per-node information. This category of information is specific to each node and kept on each node. The storage system has to contact the node to fetch the information. It includes run-time system status such as resource utilization, and performance statistics such as the historical execution time of a type of tasks.

Particularly, our algorithm considers the following information for estimating the workload on each node:

- Resource capacity of S_i (R_i): This is centralized and static information of each node including a tuple value, where each element indicates the capacity of a type of resource. For example, ⟨8 cores, 64G⟩ represents 8 CPU cores and 64G memory.
- Bundled job information: This is centralized information recorded on the master node throughout the execution of the bundled job. It includes the input data blocks and the associated application jobs. Given a data block ID, the master node is able to find which bundled job it belongs to, and retrieve the set of application jobs that are planned to process the data block.
- Execution time of a job J_{ij}'s map task (E_{ij}): This is per-node statistical information maintained by each node. This information is updated every time a task is finished at the slave node. This is the most important information for estimating the workload, and each node uses a 3-dimensional matrix ET to keep the record. For a given job J_{ij} (j-th job in bundled job BJ_i), its map tasks's execution time is recorded based on the job ID/type, the resource demands, and the resource utilization of the node. All three parameters are relevant to the execution time of the map task. When the master node queries a slave node for the average execution time E_{ij}. The slave node will check the matrix and return $E_{ij} = ET[JobID, ResourceDemand, ResourceUtilization]$. When the requested cell in ET has no value, the slave node returns the value in the closest cell. We also use EWMA (exponentially weighted moving average) to maintain the average execution time for each element in ET.
- Data block usage information (BU_i): This is a per-node information that records the usage information of the data blocks. For any data block b_i hosted at the node, BU_i is a set of job IDs indicating the jobs whose map tasks have processed the block b_i. This information helps estimate the pending jobs that may execute node-local tasks on the node.

Algorithm 2 presents the details of our workload estimation function. First, the master prepares a query by forming a requested job set (RJ) of all application jobs in active bundled jobs (lines 2–4). Then set RJ is sent to node x. For each job $J_{ij} \in RJ$, its map task's resource demands D_{ij} is sent with the query as well. After receiving the query, the slave node x retrieves the job ID and resource demands of each $J_ij \in RJ$, and checks its current resource utilization. Then it searches the performance matrix ET and return a list of average execution times (E_{ij}), one for each job J_{ij}. In addition, the slave node returns the block usage information BU_y for each block b_j it hosts (line 6). Once E_{ij} and BU_y are received, the master node estimates the workload of node x as follows. First, the algorithm identifies the pending application jobs that have not been finished by checking the bundled job information and BU_i (lines 7–13). For each pending job, we also count how many input data blocks are hosted at node x represented by c_{ij}. In lines 14–17, our algorithm estimate the execution time of J_{ij}'s map tasks by multiple the obtained average execution time E_{ij} with the estimated number of rounds of execution considering concurrent execution of multiple map tasks. Given demand D_{ij} and resource capacity R_x, the number of concurrent tasks that can be execute at node x can be estimated as

$$\frac{R_x}{D_{ij}} = \min_u\{\frac{R_x(u)}{D_{ij}(u)}\}, \quad (1)$$

where u is the index number of different types of resources. Then the number of rounds is estimated as $c_{ij}/\frac{R_x}{D_{ij}}$ (line 15).

C. Dynamic Adjustment

Finally, our algorithm also adjusts the workload estimation based on dynamic factors. At run-time, the workload estimation may be inaccurate and lead to a wrong decision of the hosting slave node. We include the following two empirical information to help the algorithm adjust the estimation.

First, when querying the candidate slave nodes, our algorithm also request the *node-local task ratio*, which is defined as the ratio between the number of node-local tasks and total number of tasks that have been executed at a slave node in a pre-configured past time window. For example, if a slave node has executed 100 map tasks in the past 10 minutes, and 80 of them are node-local tasks (i.e., the other 20 are rack-local tasks), the node's node-local task ratio is 0.8. A low node-local task ratio indicates that our algorithm overestimates the node's workload, and thus allocates too few data blocks on the node. Assume LTR_i indicate the node-local task ratio of node i. When comparing the workload of d nodes, our

Algorithm 2: Estimate Workload

```
Function EstLoad(node $x$):
    for any active bundle job $BJ_i$ do
        $RJ = RJ \cup J_{ij}$ ;    // requested jobs
    end
    send a query including $RJ$ and resource
      demands ($D_{ij}$) to node $x$;
    receive $E_{ij}$ and $BU_y$;    /* avg exe time of
      $J_{ij}$, and block usage info of $b_y$ at $x$ */
    for each $BU_y$ do
        find the associated $BJ_i$ of block $b_y$;
        for each $J_{ij} \notin BU_j$ do
            $PJ = PJ \cup \{J_{ij}\}$;    // pending jobs
            $c_{ij} \leftarrow c_{ij} + 1$;
        end
    end
    for each $J_{ij} \in PJ$ do
        $r_{ij} = c_{ij}/\frac{R_x}{D_{ij}}$;    // rounds of execution
        $w = w + E_{ij} \cdot r_{ij}$;
    end
    return w;
```

algorithm considers the value of $LTR_i \cdot EstLoad(i)$ (line 7 in Algorithm 1).

Second, we consider the resource capacity of each candidate node and the currently occupied resources in the cluster. With multiple resources (CPU and memory in Hadoop), one of them could become the bottleneck while other resources are still available in a busy cluster. The bottleneck resource that has been fully occupied is more important to the cluster at the moment. Therefore, when comparing the workload of the candidate slave nodes, our algorithm checks the current bottleneck resource, and compares its capacity on each candidate. The slave node with more bottleneck resources is more likely to be assigned with more tasks by the RM. And our estimation of the workload might be lower than its actual value. Therefore, we adjust our workload estimation for node i as $(1 + \epsilon_i) \cdot EstLoad(i)$, where ϵ_i is proportional to node i's capacity of the bottleneck resource, and $\sum \epsilon_i = \tau$. The total adjustment over all candidate nodes is limited to a small portion defined by τ (in our algorithm, $\tau = 0.1$).

V. Performance Evaluation

In this section, we present the evaluation results of our solution EA^{2}S^2. We conduct both experiments and simulation with comprehensive settings. The major performance metrics we examine are the execution time (makespan of multiple bundled jobs) and the data locality (the ratio of node-local tasks).

A. System Implementation and Alternatives for Comparison

EA^{2}S^{2}is implemented on Hadoop YARN 2.7.1. We have also implemented the following two alternatives based on only centralized information for performance comparison.

- Capacity Block Placement: This alternative approach estimates the workload of a node based on its resource capacity and the resource demands of the bundled job. When uploading data blocks for a bundled job BJ_i, the workload of a candidate node x is estimated as $w_x = \sum_j \frac{R_x}{D_{ij}}$ where each term is the same as in Eq. 1. The basic intuition here is that the node which can allocate resources for more tasks in the bundled job is preferred to host more data blocks.
- Total Block Placement: This approach simply estimate the workload of a slave node as its hosted input data blocks of all active bundled jobs. This information can be retrieved by querying the NameNode service running on the master node.

We implement two new modules on the master node, Bundledjob-Agent (BA) and HDFS-Agent (HA), and one new module on the slave nodes called Node-Agent(NA). BA handles the submission of bundled jobs, and passes the request of uploading the input files to HDFS. Once the input data is ready, BA submits the associated application jobs to Hadoop scheduler. Our main sampling algorithm for uploading data blocks is implemented in HA. It also interacts with BA to fetch the information of bundled jobs, and queries Hadoop RESTful APIs to obtain cluster information. In addition, NA running on each slave node maintains the per-node information and responds to the queries from HA.

B. Evaluation

We evaluate EA^{2}S^2 on small clusters of 4 and 10 slave nodes with experiments, and also conduct simulation for large-scale tests. By comparing makespan and local task percentage with native Hadoop, Capacity Block Placement and Total Block Placement, we show that EA^{2}S^2 improves clusters' performance with shorter makespan and higher data locality comparing with other policies. In all our experiments, the block size of HDFS is 256MB and the number of replicas is set as $k = 1$. In EA^{2}S^2, the number of candidate slave nodes in Algorithm 1 is set as $d = 2$.

1) 4 Nodes Cluster: We first test the performance on a 5-node cluster (1 master and 4 slaves), where each node is equipped with 8 CPU cores and 16G memory.

Workload: We run a batch of commonly used benchmarks,

- WordCount (WC): count the occurrences of each word;
- TeraSort (TS): sort (key,value) tuples on the key with the synthetic data as input;
- WordMean (WM): count the average length of the words;
- Grep: grep words that match specified lambda expression.

Table III shows resource demands of each benchmark.

Job	map_core	map_mem	red_num	red_core	red_mem
WC	1	4	1	1	4
TS	2	3	1	2	3
WM	2	4	1	2	4
Grep	1	3	1	1	3

TABLE III: Resource Demands of Application Jobs

We consider that a bundled job consists of a 5G input file, and four application jobs (one from each benchmarks above).

Four bundled jobs are submitted continuously to eh cluster yielding totally 20 Hadoop jobs (a Grep job includes two individual Hadoop jobs, 'search' and 'sort'). For each setting, we repeat the test for five times and present the average values here.

Environment Setting: We configure a heterogeneous environment by pushing CPU stress with linux stress tool and setting every slave in various resource capacities. The following three settings are considered in our evaluation.

Setting 1, 2 (CPU Stress): We configure the cluster under CPU stress following Setting 1 and Setting 2 (Table IV). The unit for CPU is number of vcores and for memory it is GB. Stress values indicate the CPU interference from other processes. A higher value represents more stress, thus less CPU capability for the Hadoop framework. In Setting 1, large capacity slaves are under more pressure comparing with Setting 2 in which slaves with less capacity are suffering more CPU stress.

	CPU	mem	stress
n1	6	7	0
n2	8	10	8
n3	12	12	12
n4	10	8	4

	CPU	mem	stress
n1	12	12	0
n2	10	8	8
n3	6	7	12
n4	8	10	4

TABLE IV: 4-Node Cluster Setting 1, 2 (Capacity and Stress)

Setting 3 (Homogeneous): Master and slaves are configured with 8 vcores, 12G memory and no stress.

Performance Evaluation: We first present the average execution time of map tasks on each slave node in Fig. 3 and 4. The results include the performance of both node-local tasks and non-local tasks. The execution times of local tasks in Fig. 3 validate our heterogeneous setting with CPU stress. The node with a larger stress value spends longer time to finish the map task. But in Setting 3 of a homogeneous cluster, the execution times on all the nodes are very close to each other as shown in Fig. 4. In addition, we observe that the execution times of non-local tasks are significantly prolonged compared to the corresponding local tasks. The network overhead, especially in a network with heavy traffic, yields a considerable negative impact on the task/job execution.

Next, we examine the overhead incurred by our solution n4 in the process of uploading input files. In our solution, HDFS has to query candidate slave nodes and decide the placement of data blocks. Fig. 5 shows the increased time of uploading a 5G input files comparing to the baseline of native Hadoop. Apparently, the overhead of n4 is negligible (as low as 0.23%). Capacity Block Placement also incurs a similar overhead because of the access to system RESTful APIs.

Finally, we illustrate the performance of makespan in Fig. 6. Each value represents the time elapsed from uploading the first file to the finish of the last job. In homogeneous system, Total Block Placementwhich evenly places blocks among slaves achieves the best performance by shortening the overall time by 4.38% comparing to native Hadoop. However, in both heterogeneous cases (Setting 1 and 2), n4 is superior to all other schemes. Combining per-node information to estimate the workload mitigates the dynamic uncertainness and mismatch between computing ability and hosted data blocks on each slave node. Our solution improves the makespan by 8.92% and 7.18% respectively in Setting 1 and 2.

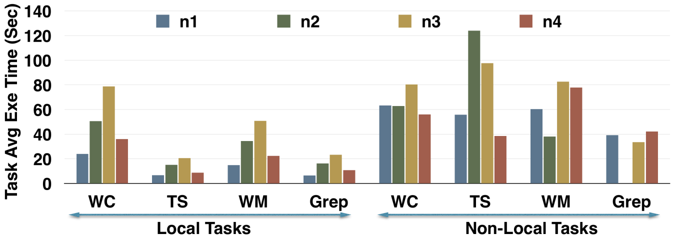

Fig. 3: Task average execution time in Setting 1 and Setting 2

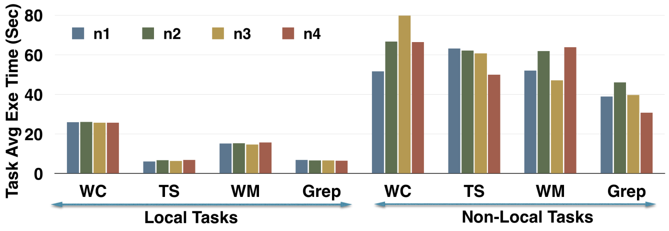

Fig. 4: Task average execution time in Setting 3

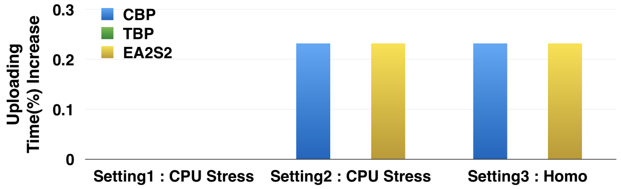

Fig. 5: Increased time of uploading a 5G input file

2) 10 Slave Cluster Tests: We also conduct experiments on a 11-node cluster with smaller input data files to evaluate the scenario where the data blocks are fewer than the cluster size.

Environment Setting: We launch a cluster with one master node and 10 slave nodes on NSF CloudLab platform. Physically there are 8 ARMv8 cores at 2.4GHz, 64 GB memory and 120 GB storage in each server. Network bandwidth is limited under 300Mbps. In our tests, we generate a heterogeneous capacity configuration as follows:

- 2 slave nodes are configured with ⟨4 vcores, 16G⟩;
- 4 slave nodes are configured with ⟨8 vcores, 32G⟩;
- 2 slave nodes are configured with ⟨4 vcores, 32G⟩;
- 2 slave nodes are configured with ⟨2 vcores, 8G⟩.

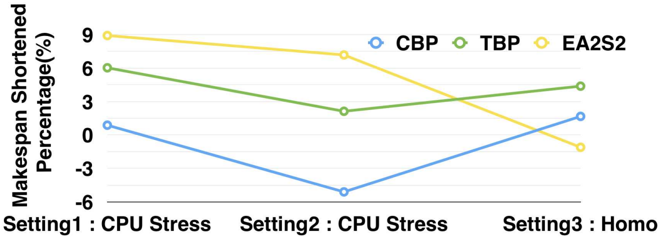

Fig. 6: Makespan of 4 slaves cluster including uploading time

We disturb slave's performance by introducing diverse CPU stress as shown in Table V. In addition, homogeneous environment is tested as Setting 3.

	Setting 1, CPU	Setting 2, CPU
slave 1,2,7,8	0	32
slave 3,4	24	16
slave 5,6	16	24
slave 9,10	32	0

TABLE V: Stresses of 10 slaves cluster in Setting 1 and 2

Workload: Similar to the previous experiments, we consider that a bundled job includes an input file and four benchmark application jobs. In this test, we submit 20 bundled jobs one after another with an interval of 20 seconds. Each setting is repeated for 5 times and the average value is presented below.

Performance Evaluation: The results are shown in Fig. 7 including the overall makespan and the number of local tasks. In Setting 3, the numbers of local tasks in native Hadoop and n4 are similar, because per-node information may not yield a significant impact. As a result the makespan of n4 is slightly worse than that of native Hadoop considering the overhead our solution incurs in HDFS. In the two heterogeneous settings, however, n4 performs much better than native Hadoop. In Setting 1, with a certain amount of CPU stress applied to the cluster, the performance of native Hadoop is actually close to that in Setting 3 with no stress. It indicates that this particular setting and the default scheduler in native Hadoop somehow mitigates the computation interferences very well. In such a case, our solution n4 still reduces the makespan by 4.9%. In Setting 2, more nodes are under high CPU stress. Both native Hadoop and n4 yield longer makespans. Comparing to Setting 3 with no stress, native Hadoop's makespan is increased by 12.54% while n4's makespan is increased by 6.69%. n4 is 3.56% more efficient than native Hadoop in Setting 2. In terms of data locality, n4 significantly increases the number of local tasks by 41.46%.

3) Simulation: Furthermore, we implement a simulator to mimic the execution of Hadoop jobs in order to evaluate our solution in a large scale cluster. All the parameters related to the job execution are imported from our experiments.

Environment Setting: We configure a heterogeneous cluster by randomly selecting values for resource capacity of each slave nodes. The number of CPU vcores is randomly selected in $[2-8]$ while the memory capacity is chosen in $[4G, 16G]$. We also use a random integer L chosen from $[1, \sqrt{clustersize}]$ to indicate the computation ability of a slave node. We assume that HDFS configures the block size to be 256M, and the number of replicas is $1/10$ of cluster size.

In addition, we set the task execution time according to our experiments, i.e., the average map task execution time of WC, TS, WM and Grep is 28s, 9s, 15s and 8s respectively in a homogeneous system without extra pressure. For every task in this simulation, we assign it a baseline time BT after it is accepted by a node S_i.

$$BT = \begin{cases} avg & \text{if this task is a node-local task;} \\ avg + \S & \text{otherwise,} \end{cases}$$

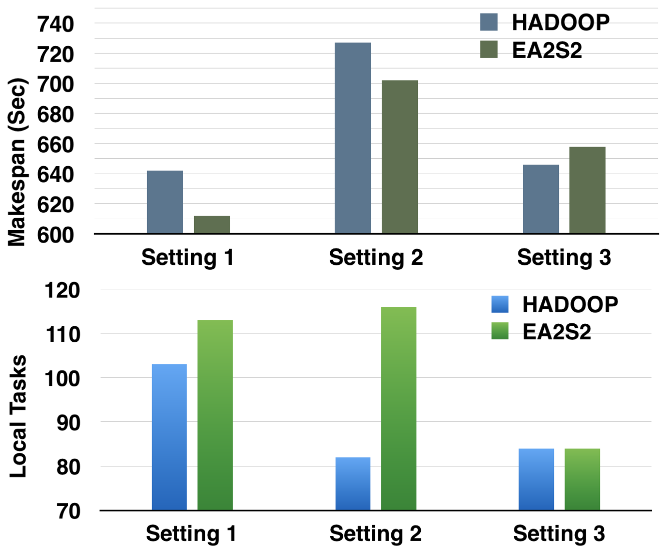

Fig. 7: Makespan and local task amount of 10 nodes cluster

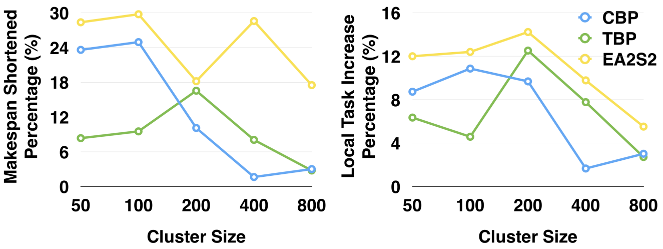

Fig. 8: Makespan of Simulated Clusters

where $\S$ is a random value indicating the extra overhead for non-local tasks. Its value selection range is also set based on our experiments. Eventually, the actual execution time of this task is set as $BT \cdot L$.

Worload: In our simulation, each input file size is an integer randomly picked from $[clustersize/100, clustersize/5]$, it does not change between policies. The application jobs we consider are same benchmarks mentioned in Fig. III. Each input file is bundled with four benchmarks, and 5 bundled jobs are submitted in one session. We repeat five sessions for each setting and show the average value below.

Performance Evaluation: Fig. 8 shows the performance of makespan and data locality with the cluster size ranging from 50 to 800. The performance of makespan is represented by the shortened percentage compare to the baseline of native Hadoop. We observe that in large scale clusters, the performance improvement of all three schemes are more significant in most cases. But both Capacity Block Placement and Total Block Placement show a trend of decreased improvement when the cluster size reaches 400. Our solution EA^{2}S^2, however, keeps a high improvement ranging from 17.51% to 29.75% in all the tested cases. One of the main causes is the improvement of data locality. EA^{2}S^2 increases the number of local tasks by $5.52\% - 14.23\%$ in the simulation.

VI. Conclusion

In this paper, we develop a new storage system to efficiently serve bundled jobs in large scale big data processing platforms. Our system improve the data block placement when uploading them by considering the associated application jobs. Sampling-based randomized algorithm is adopted to select the candidate node with the minimum workload to host new data blocks. In addition, we develop algorithm to efficiently and accurately estimate the workload of each candidate node. Our solution is evaluated with experiments and simulation. The results show that our solution is significantly superior to the native Hadoop storage system.

References

[1] Apache Hadoop NextGen MapReduce (YARN). http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[2] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, et al. Job scheduling for multi-user mapreduce clusters. Technical report, EECS Department, University of California, Berkeley, Apr 2009.

[3] Capacity Scheduler. http://hadoop.apache.org/common/docs/r1.0.0/capacity_scheduler.html.

[4] Ali Ghodsi, Zaharia, Benjamin Hindman, and etc. Dominant resource fairness: Fair allocation of multiple resource types. In *8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 323–336, Berkeley, CA, USA, 2011. USENIX Association.

[5] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. Aria: Automatic resource inference and allocation for mapreduce environments. In *ICAC*, pages 235–244, 2011.

[6] Kamal Kc and Kemafor Anyanwu. Scheduling hadoop jobs to meet deadlines. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 388–392. IEEE, 2010.

[7] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low ncy scheduling. In *24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 69–84, New York, NY, USA, 2013. ACM.

[8] Jordà Polo, Claris Castillo, David Carrera, et al. Resource-aware adaptive scheduling for mapreduce clusters. In *Middleware*, 2011.

[9] Yi Yao, Jiayin Wang, Bo Sheng, Jason Lin, and Ningfang Mi. Haste: Hadoop yarn scheduling based on task-dependency and resource-demand. In *2014 IEEE International Conference on Cloud Computing*, CLOUD '14, pages 184–191, Washington, DC, USA, 2014. IEEE Computer Society.

[10] Jiayin Wang, Yi Yao, Ying Mao, Bo Sheng, and Ningfang Mi. Fresh: Fair and efficient slot configuration and scheduling for hadoop clusters. In *CLOUD*, 2014.

[11] Jiayin Wang, Yi Yao, Ying Mao, Bo Sheng, and Ningfang Mi. Optimize mapreduce overlap with a good start(reduce) and a good finish(map). In *IPCCC*, Dec 2015.

[12] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. *SIGCOMM Comput. Commun. Rev.*, (4), August 2014.

[13] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[14] Xiaoqi Ren, Ganesh Ananthanarayanan, Adam Wierman, and Minlan Yu. Hopper: Decentralized speculation-aware cluster scheduling at scale. *SIGCOMM Comput. Commun. Rev.*, 45(4):379–392, August 2015.

[15] Ganesh Ananthanarayanan, Michael Chien-Chun Hung, Xiaoqi Ren, Ion Stoica, Adam Wierman, and Minlan Yu. Grass: Trimming stragglers in approximation analytics. In *11th USENIX Symposium (NSDI 14)*.

[16] Jiayin Wang, Teng Wang, Zhengyu Yang, Ningfang Mi, and Sheng Bo. eSplash: Efficient Speculation in Large Scale Heterogeneous Computing Systems. In *35th IEEE International Performance Computing and Communications Conference (IPCCC)*. IEEE, 2016.

[17] Xiaohong Zhang, Zhiyong Zhong, Shengzhong Feng, Bibo Tu, and Jianping Fan. Improving data locality of mapreduce by scheduling in homogeneous computing environments. In *9th International Symposium on Parallel and Distributed Processing with Applications*, pages 120–126. IEEE, 2011.

[18] Zhenhua Guo, Geoffrey Fox, and Mo Zhou. Investigation of data locality in mapreduce. In *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 419–426. IEEE Computer Society, 2012.

[19] Jiahui Jin, Junzhou Luo, Aibo Song, Fang Dong, and Runqun Xiong. Bar: an efficient data locality driven task scheduling algorithm for cloud computing. In *11th IEEE/ACM International Symposium*, pages 295–304. IEEE Computer Society, 2011.

[20] Xiaohong Zhang, Yuhong Feng, Shengzhong Feng, Jianping Fan, and Zhong Ming. An effective data locality aware task scheduling method for mapreduce framework in heterogeneous environments. In *Cloud and Service Computing (CSC)*, pages 235–242. IEEE, 2011.

[21] Jiong Xie, Shu Yin, Xiaojun Ruan, Zhiyang Ding, Yun Tian, James Majors, Adam Manzanares, and Xiao Qin. Improving mapreduce performance through data placement in heterogeneous hadoop clusters. In *Parallel & Distributed Processing*, pages 1–9. IEEE, 2010.

[22] TPC Benchmark DS (TPC-DS): The Benchmark Standard for decision support solutions including Big Data. http://www.tpc.org/tpcds/.

[23] Hive TPC Benchmarks. https://github.com/hortonworks/hive-testbench.