H-WorD: Supporting Job Scheduling in Hadoop with Workload-driven Data Redistribution

Petar Jovanovic[1], Oscar Romero[1], Toon Calders[2,3], and Alberto Abelló[1]

[1] Universitat Politècnica de Catalunya, BarcelonaTech
Barcelona, Spain ({petar|oromero|aabello}@essi.upc.edu)
[2] Universite Libre de Bruxelles, Brussels, Belgium (toon.calders@ulb.ac.be)
[3] University of Antwerp, Antwerp, Belgium (toon.calders@uantwerpen.be)

Abstract. Today's distributed data processing systems typically follow a query shipping approach and exploit data locality for reducing network traffic. In such systems the distribution of data over the cluster resources plays a significant role, and when skewed, it can harm the performance of executing applications. In this paper, we address the challenges of automatically adapting the distribution of data in a cluster to the workload imposed by the input applications. We propose a generic algorithm, named *H-WorD*, which, based on the estimated workload over resources, suggests alternative execution scenarios of tasks, and hence identifies required transfers of input data a priori, for timely bringing data close to the execution. We exemplify our algorithm in the context of MapReduce jobs in a Hadoop ecosystem. Finally, we evaluate our approach and demonstrate the benefits of automatic data redistribution.

1 Introduction

For bringing real value to end-users, today's analytical tasks often require processing massive amounts of data. Modern distributed data processing systems have emerged as a necessity for processing, in a scalable manner, large-scale data volumes in clusters of commodity resources. Current solutions, including the popular Apache Hadoop [13], provide fault-tolerant, reliable, and scalable platforms for distributed data processing. However, network traffic is identified as a bottleneck for the performance of such systems [9]. Thus, current scheduling techniques typically follow a query shipping approach where the tasks are brought to their input data, hence data locality is exploited for reducing network traffic. However, such scheduling techniques make these systems sensitive to the specific distribution of data, and when skewed, it can drastically affect the performance of data processing applications.

At the same time, distributed data storage systems, typically independent of the application layer, do not consider the imposed workload when deciding data placements in the cluster. For instance, *Hadoop Distributed File System* (HDFS) places data block replicas randomly in the cluster following only the data availability policies, hence without a guarantee that data will be uniformly distributed among *DataNodes* [12]. To address this problem, some systems have provided rules (in terms of formulas) for balancing data among cluster nodes, e.g., HBase [1], while others like HDFS provided means for correcting the data balancing offline [12]. While such techniques may help balancing data, they either overlook the real workload over the cluster resources, i.e., the usage of data, or at best leave it to the expert users to take it into consideration. In complex multi-tenant environments, the problem becomes more severe as the skewness of data can easily become significant and hence more harmful to performance.

In this paper, we address these challenges and present our workload-driven approach for data redistribution, which leverages on having a complete overview of the cluster workload and automatically decides on a better redistribution of workload and data. We focus here on the *MapReduce* model [6] and *Apache Hadoop* [13] as its widely used open-source implementation. However, notice that the ideas and similar optimization techniques as the ones proposed in this paper could be applied to other frameworks as well.

In particular, we propose an algorithm, named *H-WorD*, for supporting task scheduling in **H**adoop with **Wor**kload-driven **D**ata Redistribution. *H-WorD* starts from a set of previously profiled MapReduce jobs that are planned for execution in the cluster; e.g., a set of jobs currently queued for execution in a batch-queuing grid manager system. It initializes the cluster workload, following commonly used scheduling techniques (i.e., exploiting data locality, hence performing *query shipping*). Then, *H-WorD* iteratively reconsiders the current workload distribution by proposing different execution scenarios for map tasks (e.g., executing map tasks on nodes without local data, hence performing also *data shipping*). In each step, it estimates the effect of a proposed change to the overall cluster workload, and only accepts those that potentially improve certain quality characteristics. We focus here on improving the overall makespan[4] of the jobs that are planned for execution. As a result, after selecting execution scenarios for all map tasks, *H-WorD* identifies the tasks that would require *data shipping* (i.e., transferring their input data from a remote node). Using such information, we can proactively perform data redistribution in advance for boosting tasks' data locality and parallelism of the MapReduce jobs.

On the one hand, the *H-WorD* algorithm can be used *offline*, complementary to existing MapReduce scheduling techniques, to automatically instruct redistribution of data beforehand, e.g., plugged as a guided *rebalancing scheme* for HDFS [2]. On the other hand, *H-WorD* can be used *on the fly*, with more sophisticated schedulers, which would be able to take advantage of a priori knowing potentially needed data transfers, and leveraging on idle network cycles to schedule such data transfers in advance, without deferring other tasks' executions.

Contributions. In particular, our main contributions are as follows.

- We present an automatic approach for workload-driven data redistribution in Apache Hadoop.

[4] We define *makespan* as the total time elapsed from the beginning of the execution of a set of jobs, until the end of the last executing job [5].

- We introduce a novel algorithm, *H-WorD*, which, led by the real workload in the cluster, decides on a data redistribution that would improve the performance of executing that workload.
- We evaluate our approach using well-known MapReduce benchmarks, for showing the effectiveness of the *H-WorD* algorithm.

Outline. The rest of the paper is structured as follows. Section 2 introduces a running example used throughout the paper. Section 3 discusses the motivation and presents the problem of data redistribution in Hadoop. Section 4 formalizes the notation and presents the *H-WorD* algorithm. In Section 5, we report on our experimental findings. Finally, sections 6 and 7 discuss related work and conclude the paper, respectively.

2 Running Example

To illustrate our approach and facilitate the explanations throughout the paper, we introduce a running example based on a set of three MapReduce *WordCount*[5] jobs, with different input data sets. A MapReduce job executes in two consecutive phases, namely **map** and **reduce** [6]. **Map phase** processes an input file from HDFS. The file is split in logical data blocks of the same size (e.g., 64MB or 128MB), physically replicated for fault tolerance, and distributed over the cluster nodes. Each data block is processed by a single *map task*.

Table 1. Example MapReduce jobs

job ID	*file ID*	*size (MB)*	*#tasks*	$dur^{mapTask}$ *(s)*	$dur^{mapInTransfer}$ *(s)*
1	f1	1920	15	40	6.34
2	f2	640	5	40	6.34
3	f3	1280	10	40	6.34

We profiled the example MapReduce jobs using an external tool, called *Starfish* [8]. *Starfish* can create job profiles on the fly, by applying sampling methods (e.g., while jobs are queued waiting for execution), or from previous jobs' executions. The portion of the profiles of the example jobs focusing on map tasks are presented in Table 1. We trace the number of map tasks, the average duration of each task ($dur^{mapTask}$), as well as the average duration of transferring its input data block over the network (i.e., $dur^{mapInTransfer}$).

Furthermore, we consider a computing cluster with three computing nodes, each with a capacity of 2CPUs and 2GB of memory, connected through the network with 100Mbps of bandwidth (see Figure 1). We deployed Hadoop 2.x on the given cluster, including HDFS and MapReduce. In addition, for simplifying the explanations, we configured HDFS for creating only one replica of each input data block. In Figure 1, we depict the initial distribution of the input data in the cluster. Note that each input data block is marked as $DB\mathbf{X}_{fid}$, where **X** is an identifier of a block inside a file, and *fid* is the id of the file it belongs to.

For reasons of simplicity, we configured all example jobs to require *containers* (i.e., bundles of node resources) with 1CPU and 1GB of memory for

[5] WordCount Example: `https://wiki.apache.org/hadoop/WordCount`

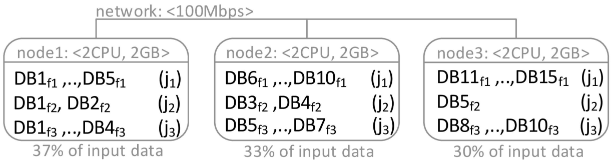
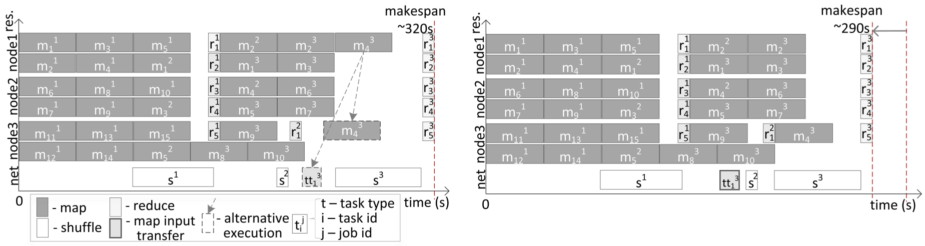

network: <100Mbps>
node1: <2CPU, 2GB>
DB1$_{f1}$,..,DB5$_{f1}$ (j$_1$)
DB1$_{f2}$, DB2$_{f2}$ (j$_2$)
DB1$_{f3}$,..,DB4$_{f3}$ (j$_3$)
37% of input data
node2: <2CPU, 2GB>
DB6$_{f1}$,..,DB10$_{f1}$ (j$_1$)
DB3$_{f2}$,DB4$_{f2}$ (j$_2$)
DB5$_{f3}$,..,DB7$_{f3}$ (j$_3$)
33% of input data
node3: <2CPU, 2GB>
DB11$_{f1}$,..,DB15$_{f1}$ (j$_1$)
DB5$_{f2}$ (j$_2$)
DB8$_{f3}$,..,DB10$_{f3}$ (j$_3$)
30% of input data

Fig. 1. Example cluster configuration and initial data distribution

accommodating each map and reduce task, i.e., *mapreduce.map.memory.mb = mapreduce.reduce.memory.mb = 1024*, and *mapreduce.map.cpu.vcores = mapreduce.reduce.cpu.vcores = 1.*

3 The Problem of Skewed Data Distribution

We further applied the default scheduling policy of Hadoop (i.e., exploiting data locality) to our running example. An execution timeline is showed in Figure 2:left, where the x-axis tracks the start and end times of tasks and the y-axis shows the resources the tasks occupy at each moment. For clarity, we further denote a task t_i^j both with the task id i, and the job id j. Notice in Figure 1 that the job ids refer to groups of input data blocks that their map tasks are processing, which determines the placement of the map tasks in the cluster for exploiting data locality. First, from the timeline in Figure 2:left, we can notice that although the distribution of input data is not drastically skewed, it affects the execution of job 3, since for executing map task m_4^3, we need to wait for available computing resources on *node*1.



(Baseline) **(With data redistribution)**

Fig. 2. Timeline of executing example MapReduce jobs

Furthermore, we can also observe some idle cycles on the computing resources (i.e., *node*3), that obviously could alternatively accommodate m_4^3, and finish the map phase of job 3 sooner. However, *node*3 does not contain the needed input data at the given moment, thus running m_4^3 on *node*3 would require transferring its input data (i.e., tt_1^3), which would also defer its execution (see alternative execution of m_4^3 in Figure 2:left).

Having such information beforehand, we could redistribute data in a way that would improve utilization of cluster resources, and improve the makespan. Such data redistribution could be done *offline* before starting the execution of MapReduce jobs. However, note that there are also idle cycles on the network resource (e.g., between s^1 and s^2, and between s^2 and s^3). This is exactly where having more information about the imposed workload makes the difference. In particular, knowing that the higher workload of *node*1 can potentially affect the makespan of the jobs' execution, we could take advantage of idle network resources and plan for timely *on the fly* transferring of m_4^3's input data to another node, in overlap with other tasks' execution, and hence improve the execution makespan. Such alternative execution scenario is depicted in Figure 2:right.

We showcased here in a simple running example that in advance data redistribution can moderately improve the makespan. However, typical scenarios in Hadoop are much more complex, with larger and more complex cluster configurations, greater number of jobs, more complex jobs, and larger input data sizes. Thus, it is obvious that estimating the imposed workload over cluster resources and deciding on data and workload redistribution is intractable for humans and requires efficient automatic means. At the same time, in such real world scenarios improving resource utilization and minimizing the execution makespan is essential for optimizing the system performance.

We further studied how to automatically, based on the *estimated workload*, find new *execution scenarios* that would improve *data distribution* in the cluster, and hence reduce the makespan. Specifically, we focused on the following challenges:

- **Resource requirements.** For obtaining the workload that a job imposes over the cluster, we need to model *cluster resources*, input *MapReduce jobs*, and the *resource requirements* of their tasks.
- **Alternative execution scenarios.** We need to model alternative execution scenarios of MapReduce jobs, based on the distribution of input data in a cluster and alternative destination resources for their tasks. Consequently, alternative execution scenarios may pose different resource requirements.
- **Workload estimation.** Next, we need an efficient model for estimating the workload over the cluster resources, for a set of jobs, running in certain execution scenarios.
- **Data redistribution.** Lastly, we need an efficient algorithm, that, using the estimated workload, selects the most favorable execution scenario, leading to a better distribution of data in a cluster, and to reducing the makespan.

4 Workload-driven Redistribution of Data

In this section, we tackle the previously discussed challenges, and present our algorithm for workload-driven redistribution of data, namely, *H-WorD.*

4.1 Resource requirement framework

In this paper, we assume a set of previously profiled MapReduce jobs as input (see the example set of jobs in Table 1). Notice that this is a realistic scenario for batched analytical processes that are run periodically, hence they can be planned together for better resource utilization and lower makespan. For instance, in a grid manager system, a set of jobs are queued, waiting for execution, during which time we can decide on a proper distribution of their input data.

A set of MapReduce jobs is submitted for execution in a cluster, and each job j_x consists of sets of *map* and *reduce* tasks.

$$J := \{j_1, ..., j_n\},\ j_x := MT_x \cup RT_x \tag{1}$$

The set of all tasks of J is defined as $T_J = \bigcup_{x=1}^{n} j_x = \bigcup_{x=1}^{n}(MT_x \cup RT_x)$.

These tasks can be scheduled for execution in the cluster that comprises two main *resource types*, namely: *computing resources* (i.e., nodes; R_{cmp}), and *communication resources* (i.e., network; R_{com}).

$$R := R_{cmp} \cup R_{com} = \{r_1, ..., r_n\} \cup \{r_{net}\} \tag{2}$$

Each resource r (*computing* or *communication*) has a certain capacity vector $\boldsymbol{C}(r)$, defining capacities of the physical resources that are used for accommodating MapReduce tasks (i.e., containers of certain CPU and memory capacities, or a network of certain bandwidth).

$$\forall r \in R_{cmp}, \boldsymbol{C}(r) := \langle c_{cpu}(r), c_{mem}(r)\rangle; \forall r \in R_{com}, \boldsymbol{C}(r) := \langle c_{net}(r)\rangle \tag{3}$$

Each task t_i^j requires resources of certain resource types (i.e., *computing* and *communication*) during their execution. We define a *resource type requirement* RTR_k of task t_i^j, as a pair $[S, d]$, such that t_i^j requires for its execution one resource from the set of resources S of type k ($S \subseteq R_k$), for a duration d.

$$RTR_k(t_i^j) := [S, d], st.:\ S \subseteq R_k \tag{4}$$

Furthermore, we define a *system requirement* of task t_i^j, as a set of *resource type requirements* over all resource types in the cluster, needed for the complete execution of t_i^j.

$$SR(t_i^j) := \{RTR_1(t_i^j), ..., RTR_l(t_i^j)\} \tag{5}$$

Lastly, depending on specific resources used for its execution, task t_i^j can be executed in several different ways. To elegantly model different execution scenarios, we further define the concept of *execution modes*. Each *execution mode* is defined in terms of a *system requirement* that a task poses for its execution in a given scenario (denoted $SR(t_i^j)$).

$$\mathcal{M}(t_i^j) := \{SR_1(t_i^j), ..., SR_m(t_i^j)\} \tag{6}$$

Example. The three example MapReduce jobs (job 1, job 2, and job 3; see Table 1), are submitted for execution in the Hadoop cluster shown in Figure 1. Cluster comprises three computing resources (i.e., $node1$, $node2$, and $node3$), each with a capacity of $\langle 2CPU, 2GB\rangle$, connected through a network of bandwidth capacity $\langle 100Mbps\rangle$. Map task m_1^1 of job 1 for its data local execution mode requires a container of computing resources, on a node where the replica of its input data is placed (i.e., $node1$), for the duration of 40s. This requirement is captured as $RTR_{cmp}(m_1^1) = [\{node1\}, 40s]$. □

4.2 Execution modes of map tasks

In the context of distributed data processing applications, especially MapReduce jobs, an important characteristic that defines the way the tasks are executed, is the distribution of data inside the cluster. This especially stands for executing map tasks which require a complete data block as input (e.g., by default 64MB or 128MB depending on the Hadoop version).

Data distribution. We first formalize the distribution of data in a cluster (i.e., data blocks stored in HDFS; see Figure 1), regardless of the tasks using these data. We thus define function f_{loc} that maps logical data blocks $DBX_{fid} \in \mathbb{DB}$ of input files to a set of resources where these blocks are (physically) replicated.

$$f_{loc} : \mathbb{DB} \rightarrow \mathcal{P}(R_{cmp}) \tag{7}$$

Furthermore, each map task m_i^j processes a block of an input file, denoted $db(m_i^j) = DBX_{fid}$. Therefore, given map task m_i^j, we define a subset of resources where the physical replicas of its input data block are placed, i.e., *local resource set* LR_i^j.

$$\forall m_i^j \in MT_J, LR_i^j := f_{loc}(db(m_i^j)) \tag{8}$$

Conversely, for map task m_i^j we can also define *remote resource sets*, where some resources may not have a physical replica of a required data block, thus executing m_i^j may require transferring input data from another node. Note that for keeping the replication factor fulfilled, a *remote resource set* must be of the same size as the *local resource set.*

$$\forall m_i^j \in MT_J, \mathbb{RR}_i^j := \{RR_i^j | RR_i^j \in (\mathcal{P}(R_{cmp}) \setminus LR_i^j) \wedge |RR_i^j| = |LR_i^j|\} \tag{9}$$

Following from the above formalization, map task m_i^j can be scheduled to run in several *execution modes.* The system requirement of each execution mode of m_i^j depends on the distribution of its input data. Formally:

$$\forall m_i^j \in MT_J, \mathcal{M}(m_i^j) = \{SR_{loc}(m_i^j)\} \cup \textstyle\bigcup_{k=1}^{|\mathbb{RR}_i^j|} \{SR_{rem,k}(m_i^j)\}, s.t.: \tag{10}$$
$$SR_{loc}(m_i^j) = \{[LR_i^j, d_i^{j,cmp}]\};\ SR_{rem,k}(m_i^j) = \{[RR_{i,k}^j, d_{i,k}^{j,cmp}], [\{r_{net}\}, d_{i,k}^{j,com}]\}$$

Intuitively, a map task can be executed in the *local execution mode* (i.e., $SR_{loc}(m_i^j)$), if it executes on a node where its input data block is already placed, i.e., without moving data over the network. In that case, a map task requires a computing resource from LR_i^j for the duration of executing map function over the complete input block (i.e., $d_i^{j,cmp} = dur^{mapTask}$). Otherwise, a map task can also execute in a *remote execution mode* (i.e., $SR_{rem}(m_i^j)$), in which case, a map task can alternatively execute on a node without its input data block. Thus, the map task, besides a node from a *remote resource set*, may also require transferring input data block over the network. Considering that a *remote resource set* may also contain nodes where input data block is placed, hence not requiring data transfers, we probabilistically model the duration of the network usage.

$$d_{i,k}^{j,com} = \begin{cases} \frac{|RR_{i,k}^j \setminus LR_i^j|^2}{|RR_{i,k}^j|} \cdot dur^{mapInTransfer}, \text{if } \textit{on the fly} \text{ redistirbution} \\ 0, \text{if } \textit{offline} \text{ redistirbution} \end{cases} \tag{11}$$

In addition, note that in the case that data redistribution is done *offline*, given data transfers will not be part of the jobs' execution makespan (i.e., $d_{i,k}^{j,com} = 0$).

Example. Notice that there are three execution modes in which map task m_4^3 can be executed. Namely, it can be executed in the *local execution mode* $SR_{loc}(m_4^3) = \{[\{node1\}, 40s]\}$, in which case, it requires a node from its *local resource set* (i.e., $LR_4^3 = \{node1\}$). Alternatively, it can also be executed in one of the two *remote execution modes.* For instance, if executed in the *remote execution mode* $SR_{rem,2}(m_4^3) = \{[\{node3\}, 40s], [\{net\}, 6.34s]\}$, it would require a node from its *remote resource set* $RR^3_{4,1} = \{node3\}$, and the network resource for transferring its input block to *node*3 (see dashed boxes in Figure 2:left). □

Consequently, selecting an *execution mode* in which a map task will execute, directly determines its system requirements, and the set of resources that it will potentially occupy. This further gives us information of cluster nodes that may require a replica of input data blocks for a given map task.

To this end, we base our *H-WorD* algorithm on selecting an *execution mode* for each map task, while at the same time collecting information about its resource and data needs. This enables us to plan data redistribution beforehand and benefit from idle cycles on the network (see Figure 2:right).

4.3 Workload estimation

For correctly redistributing data and workload in the cluster, the selection of *execution modes* of map tasks in the *H-WorD* algorithm is based on the estimation of the current workload over the cluster resources.

Algorithm 1 getWorkload

```
inputs: $SR(t_i^j)$; output: $W : R \to \mathbb{Q}$
1: for all $r \in R$ do
2:     $W(r) \leftarrow 0$;
3: end for
4: for all $[S, d] \in SR(t_i^j)$ do
5:     for all $r \in S$ do
6:         $W(r) \leftarrow W(r) + \frac{d}{|S|}$;
7:     end for
8: end for
```

In our context, we define a *workload* as a function $W : R \to \mathbb{Q}$, that maps the cluster resources to the time for which they need to be occupied. When selecting an *execution mode*, we estimate the current workload in the cluster in terms of tasks, and their current *execution modes* (i.e., *system requirements*). To this end, we define the procedure *getWorkload* (see Alg. 1), that for map task t_i^j, returns the imposed workload of the task over the cluster resources R, when executing in *execution mode* $SR(t_i^j)$.

Example. Map task m_4^3 (see Figure 2:left), if executed in *local execution mode* $SR_{loc}(m_4^3)$, imposes the following workload over the cluster: $W(node1) = 40, W(node2) = 0, W(node3) = 0, W(net) = 0$. But, if executed in *remote execution mode* $SR_{rem,2}(m_4^3)$, the workload is redistributed to *node3*, i.e., $W(node1) = 0, W(node2) = 0, W(node3) = 40$, and to the network for transferring input data block to *node3*, i.e., $W(net) = 6.34$. □

Following from the formalization in Section 4.1, a *resource type requirement* of a task defines a set of resources S, out of which the task occupies one for its execution. Assuming that there is an equal probability that the task will be scheduled on any of the resources in S, when estimating its workload imposed over the cluster we equally distribute its complete workload over all the resources in S (steps 4 - 8). In this way, our approach does not favor any specific cluster resource when redistributing data and workload, and is hence agnostic to the further choices of the chosen MapReduce schedulers.

4.4 The *H-WorD* algorithm

Given the workload estimation means, we present here *H-WorD*, the core algorithm of our workload-driven data redistribution approach (see Alg. 2).

Algorithm 2 *H-WorD*

```
inputs: $MT_J$
 1: $todo \leftarrow MT_J$;
 2: for all $r \in R$ do $W(r) \leftarrow 0$; end for
 3: for all $t \in MT_J$ do
 4:     $SR_{cur}(t) \leftarrow SR_{loc}(t)$;
 5:     $W_t \leftarrow \mathbf{getWorkload}(SR_{cur}(t))$;
 6:     for all $r \in R$ do
 7:         $W(r) \leftarrow W(r) + W_t(r)$;
 8:     end for
 9: end for
10: while $todo \neq \emptyset$ do
11:     $t \leftarrow \mathbf{nextFrom}(todo)$; $todo \leftarrow todo \setminus \{t\}$;
12:     $SR_{new}(t) \leftarrow SR_x(t) | \mathbf{q}(W + \Delta_{x,cur}) = \min_{SR_j(t) \in \mathcal{M}(t) \setminus \{SR_{cur}(t)\}} \left\{ \mathbf{q}(W + \Delta_{j,cur}) \right\}$
13:     if $\mathbf{q}(W) > \mathbf{q}(W + \Delta_{new,cur})$ then
14:         $SR_{cur}(t) \leftarrow SR_{new}(t)$;
15:         $W \leftarrow W + \Delta_{new}$;
16:     end if
17: end while
```

H-WorD initializes the total workload over the cluster resources following the policies of the Hadoop schedulers which mainly try to satisfy the data locality first. Thus, as the baseline, all map tasks are initially assumed to execute in a *local execution mode* (steps 2 - 9).

H-WorD further goes through all map tasks of input MapReduce jobs, and for each task selects an execution mode that potentially brings the most benefit to the jobs' execution. In particular, we are interested here in reducing the execution makespan, and hence we introduce a heuristic function $\mathbf{q}$(W), which combines the workloads over all resources, and estimates the maximal workload in the cluster, i.e., $\mathbf{q}(W) = max_{r \in R}(W(r))$. Intuitively, this way we obtain a rough estimate of the makespan of map tasks executing in the cluster.

Accordingly, for each map task, *H-WorD* selects an *execution mode* that imposes the minimal makespan to the execution of input MapReduce jobs (Step 12). The *delta workload* that a change in *execution modes* ($SR_{cur} \rightarrow SR_{new}$) imposes is obtained as: $\Delta_{new,cur} = \mathbf{getWorkload}(SR_{new}(t)) - \mathbf{getWorkload}(SR_{cur}(t))$.

Finally, for the selected (*new*) *execution mode* $SR_{new}(t)$, *H-WorD* analyzes if such a change in execution modes actually brings benefits to the execution of input jobs, and if the global makespan estimate is improved (Step 13), we assign the *new* execution mode to the task (Step 14). In addition, we update the current total workload over the cluster due to changed execution mode of the map task (Step 15).

Example. An example of the *H-WorD* execution is shown in Table 2. After *H-WorD* analyzes the *execution modes* of task m_4^3, it finds that the *remote execution mode* $SR_{rem,2}(m_4^3)$ improves the makespan (i.e., 440 → 400). Thus, it decides to select this *remote execution mode* for m_4^3. □

It should be noted that the order in which we iterate over the map tasks may affect the resulting workload distribution in the cluster. To this end, we apply

Table 2. *H-WorD* algorithm: example of the improved *makespan* for task m_4^3

Workload	*Initial*	...	*After task* m_4^3	...
$\mathbf{W}(node1)$	440	...	400	...
$\mathbf{W}(node2)$	400	...	400	...
$\mathbf{W}(node3)$	360	...	400	...
$\mathbf{W}(net)$	0	...	15	...
Makespan: $\mathbf{q}$(W)	440	...	400	...

here a recommended *longest task time* priority rule in job scheduling [5], and in each iteration (Step 11) we select the task with the largest duration, combined over all resources. *H-WorD* is extensible to other priority rules.

Computational complexity. When looking for the new *execution mode* to select, the *H-WorD* algorithm at first glance indicates combinatorial complexity in terms of the cluster size (i.e., number of nodes), and the number of replicas, i.e., $|RR_t| = \frac{|R_{cmp}|!}{(|R_{cmp}|-|LR_t|)!\cdot|LR_t|!}$. The search space for medium-sized clusters (e.g., 50-100 nodes), where our approach indeed brings the most benefits, is still tractable (19.6K-161.7K), while the constraints of the replication policies in Hadoop, which add to *fault tolerance*, additionally prune the search space.

In addition, notice also that for each change of *execution modes*, the corresponding data redistribution action may need to be taken to bring input data to the remote nodes. As explained in Section 3, this information can either be used to redistribute data *offline* before scheduling MapReduce jobs, or incorporated with scheduling mechanisms to schedule input data transfers *on the fly* during the idle network cycles (see Figure 2:right).

5 Evaluation

In this section we report on our experimental findings.

Experimental setup. For performing the experiments we have implemented a prototype of the *H-WorD* algorithm. Since the HDFS currently lacks the support to instruct the data redistribution, for this evaluation we rely on simulating the execution of MapReduce jobs. In order to facilitate the simulation of MapReduce jobs' executions we have implemented a basic scheduling algorithm, following the principles of the resource-constrained project scheduling [10].

Inputs. Besides *WordCount*, we also experimented with a reduce-heavy MapReduce benchmark job, namely *TeraSort*[6]. We started from a set of three profiled MapReduce jobs, two *WordCount* jobs resembling jobs 1 and 2 of our running example, and one *TeraSort* job, with 50 map and 10 reduce tasks. We used the *Starfish* tool for profiling MapReduce jobs [8]. When testing our algorithm for larger number of jobs, we replicate these three jobs.

Experimental methodology. We scrutinized the effectiveness of our algorithm in terms of the following parameters: *number of MapReduce jobs*, *initial skewness of data distribution inside the cluster*, and *different cluster sizes.* Notice that we define *skewness of data distribution* inside a cluster in terms of

[6] *TeraSort:* `https://hadoop.apache.org/docs/r2.7.1/api/org/apache/hadoop/examples/terasort/package-summary.html`

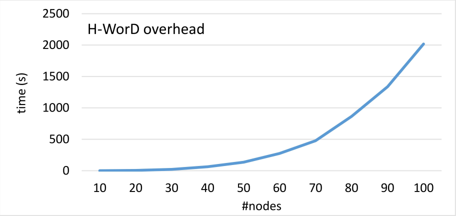


Fig. 3. *H-WorD* overhead (skew: 0.5, #jobs: 9)

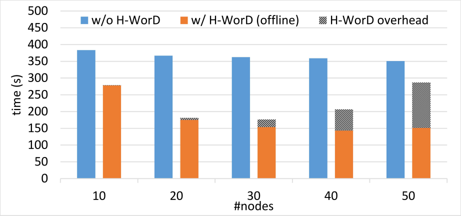


Fig. 4. Performance gains - #nodes (skew: 0.5, #jobs: 9)

the percentage of input data located on a set of X nodes, where X stands for the number of configured replicas. See for example 37% skewness of data in our running example (bottom of Figure 1). This is important in order to guarantee a realistic scenario where multiple replicas of an HDFS block are not placed on the same node. Moreover, we considered the default Hadoop configuration with 3 replicas of each block. In addition, we analyzed two use cases of our algorithm, namely *offline* and *on the fly redistribution* (see Section 4.4). Lastly, we analyzed the *overhead* that *H-WorD* potentially imposes, as well as the *performance improvements* (in terms of jobs' *makespan*) that *H-WorD* brings.

Scrutinizing *H-WorD*. Next, we report on our experimental findings.

Note that in each presented chart we analyzed the behavior of our algorithm for a single input parameter, while others are fixed and explicitly denoted.

Algorithm overhead. We first analyzed the overhead that the *H-WorD* algorithm imposes to scheduling of MapReduce jobs. Following the complexity discussion in Section 4.4, for small and medium-sized clusters (i.e., from 20 to 50 nodes), even though the overhead is growing exponentially (0.644s → 135.68s; see Figure 3), it still does not drastically delay the execution of input MapReduce jobs. This can also be observed in Figure 4, where we can still see the improvements brought by data redistribution, including the *H-WorD* overhead. However, if we keep increasing the cluster size, we can notice that the overhead, although tractable, soon becomes severely high to affect the performance of MapReduce jobs' execution (e.g., 2008s for the cluster of 100 nodes). While these results show the applicability of our approach for small and medium-sized clusters, they at the same time motivate our further research towards defining different heuristics for pruning the search space.

Performance improvements. We further report on the *performance improvements* that *H-WorD* brings to the execution of MapReduce jobs.

- *Cluster size.* We start by analyzing the effectiveness of our approach in terms of the number of computing resources. We can observe in Figure 4 that skewed data distribution (50%) can easily prevent significant scale-out improvements with increasing cluster size. This shows another advantage of *H-WorD* in improving execution makespan, by benefiting from balancing the workload over the cluster resources. Notice however that the makespan improvements are bounded here by the fixed parallelism of reduce tasks (i.e., no improvement is shown for clusters over 40 nodes).

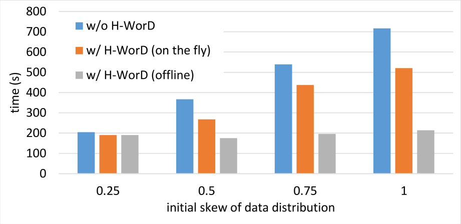


Fig. 5. Performance gains - data skewness (#nodes: 20, #jobs: 9)

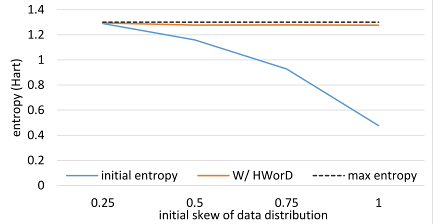


Fig. 6. "Correcting" skewness - entropy (#nodes: 20, #jobs: 9)

- *"Correcting" data skewness.* We further analyzed how *H-WorD* improves the execution of MapReduce jobs by "correcting" the skewness of data distribution in the cluster (see Figure 5). Notice that we used this test also to compare *offline* and *on the fly* use cases of our approach. With a small skewness (i.e., 25%), we observed only very slight improvement, which is expected as data are already balanced inside the cluster. In addition, notice that the makespan of *offline* and *on the fly* use cases for the 25% skewness are the same. This comes from the fact that "correcting" small skewness requires only few data transfers over the network, which do not additionally defer the execution of the tasks. However, observe that larger skewness (i.e., 50% - 100%) may impose higher workload over the network, which in the case of *on the fly* data redistribution may defer the execution of some tasks. Therefore, the performance gains in this case are generally lower (see Figure 5). In addition, we analyzed the effectiveness of our algorithm in "correcting" the data distribution by capturing the distribution of data in the cluster in terms of a Shannon entropy value, where the percentages of data at the cluster nodes represent the probability distribution. Figure 6 illustrates how *H-WorD* effectively corrects the data distribution and brings it very close ($\Delta \approx 0.02$) to the maximal entropy value (i.e., uniform data distribution). Notice that the initial entropy for 100% skew is in this case higher than 0, since replicas are equally distributed over 3 cluster nodes.
- *Input workload.* Lastly, we analyze the behavior of our algorithm in terms of the input workload, expressed in terms of the number of MapReduce jobs. We observed (see Figure 7) that the performance gains for various input load sizes are stable (∼48.4%), having a standard deviation of 0.025. Moreover, notice that data redistribution abates the growth of makespan caused by increasing input load. This demonstrates how our approach smooths the execution of MapReduce jobs by boosting data locality of map tasks.

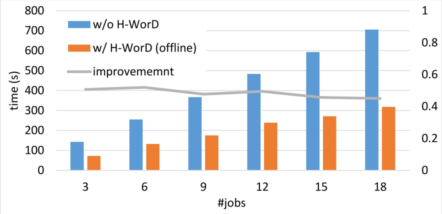


Fig. 7. Performance gains - workload (skew: 0.5, #nodes: 20)

6 Related Work

Data distribution. Currently, distributed file systems, like HDFS [12], do not consider the real cluster workload when deciding about the distribution of data over the cluster resources. By default, data are distributed randomly, without a guarantee that they will be balanced. Additional tools, namely *balancer*, are provided to balance data offline. However, such balancing is still done blindly, without considering the real usage of such data by the given workload.

Data locality. Hadoop's default scheduling techniques (i.e., Capacity [3] and Fair [4] schedulers), typically rely on exploiting data locality in the cluster, i.e., favoring *query shipping*. Moreover, other, more advanced scheduling proposals, e.g., [9,15], to mention a few, also favor query shipping and exploiting data locality in Hadoop, claiming that it is crutial for performance of MapReduce jobs. The approach in [15] in addition proposes techniques that address the conflict between data locality and *fairness* in scheduling MapReduce jobs. For achieving higher data locality, they delay jobs that cannot be accommodated locally to their data. While we agree that data locality is essential for boosting the execution of MapReduce jobs, these approaches overlook the fragileness of such techniques to skewed distribution of data in a cluster.

Combining data and query shipping. To address such problem, other approaches (e.g., [7,14]) propose combining *data* and *query shipping* in a Hadoop cluster. In [7], the authors investigate on data locality, and claim that having a global overview of the executing tasks, rather than one task at a time as in current techniques, gives better opportunities for optimally scheduling tasks and selecting local or remote execution. [14], on the other side, uses a stochastic approach, and builds a model for predicting a workload over the cluster resources, when deciding on data locality for map tasks. We find the ideas and techniques for estimating resource workload of these approaches motivational for our work. However, these techniques do not leverage on the estimated workload to perform in advance data transfers and additionally boost data locality for map tasks.

Finally, the first approach that tackles the problem of adapting data placement to the workload is presented in [11]. We find this work especially interesting for our research. The authors argue for the benefits of having a data placement aware of the cluster workload. However, while the authors propose an efficient framework for load-aware data placement, they consider data placements for single jobs, in isolation. In addition, they propose different placement techniques depending on the job types. We, on the other side, propose more generic approach relying only on an information gathered from job profiles, and consider a set of different input jobs at a time.

7 Conclusions and Future Work

In this paper, we have presented *H-WorD*, our approach for workload-driven redistribution of data in Hadoop. *H-WorD* starts from a set of MapReduce jobs and estimates the workload that such jobs impose over the cluster resources. *H-WorD* further iteratively looks for alternative execution scenarios and identifies more favorable distribution of data in the cluster beforehand. This way *H-WorD* improves resource utilization in a Hadoop cluster and reduces the makespan of MapReduce jobs. Our approach can be used for automatically instructing redistribution of data and as such is complementary to current scheduling solutions in Hadoop (i.e., those favoring data locality).

Our initial experiments showed the effectiveness of the approach and the benefits it brings to the performances of MapReduce jobs in a simulated Hadoop cluster execution. Our future plans focus on providing new scheduling techniques in Hadoop that take full advantage of a priori knowing more favorable data distribution, and hence use idle network cycles to transfer data to the tasks in advance, without additionally deferring their executions.

References

1. Apache HBase. `https://hbase.apache.org/`, [02-March-2016]
2. Cluster Rebalancing in HDFS. `http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html#Cluster+Rebalancing`, [02-March-2016]
3. Hadoop: Capacity Scheduler. `http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html`, [04-March-2016]
4. Hadoop: Fair Scheduler. `https://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-site/FairScheduler.htmll`, [04-March-2016]
5. Błażewicz, J., Ecker, K.H., Pesch, E., Schmidt, G., Weglarz, J.: Handbook on scheduling: from theory to applications. Springer Science & Business Media (2007)
6. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM 51(1), 107–113 (2008)
7. Guo, Z., Fox, G., Zhou, M.: Investigation of data locality in mapreduce. In: CCGrid. pp. 419–426 (2012)
8. Herodotou, H., Lim, H., Luo, G., Borisov, N., Dong, L., Cetin, F.B., Babu, S.: Starfish: A self-tuning system for big data analytics. In: CIDR. pp. 261–272 (2011)
9. Jin, J., Luo, J., Song, A., Dong, F., Xiong, R.: BAR: an efficient data locality driven task scheduling algorithm for cloud computing. In: CCGrid. pp. 295–304 (2011)
10. Kolisch, R., Hartmann, S.: Heuristic algorithms for the resource-constrained project scheduling problem: Classification and computational analysis. Springer (1999)
11. Palanisamy, B., Singh, A., Liu, L., Jain, B.: Purlieus: locality-aware resource allocation for mapreduce in a cloud. In: SC. pp. 58:1–58:11 (2011)
12. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: MSST. pp. 1–10 (2010)
13. Vavilapalli, V.K., Murthy, A.C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., Baldeschwieler, E.: Apache hadoop YARN: yet another resource negotiator. In: SOCC. p. 5 (2013)
14. Wang, W., Zhu, K., Ying, L., Tan, J., Zhang, L.: Map task scheduling in MapReduce with data locality: Throughput and heavy-traffic optimality. In: INFOCOM. pp. 1609–1617 (2013)
15. Zaharia, M., Borthakur, D., Sarma, J.S., Elmeleegy, K., Shenker, S., Stoica, I.: Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In: EuroSys. pp. 265–278 (2010)