FRESH: Fair and Efficient Slot Configuration and Scheduling for Hadoop Clusters

Jiayin Wang* jane@cs.umb.edu
Yi Yao† yyao@ece.neu.edu
Ying Mao* yingmao@cs.umb.edu
Bo Sheng* shengbo@cs.umb.edu
Ningfang Mi† ningfang@ece.neu.edu

*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*—Hadoop is an emerging framework for parallel big data processing. While becoming popular, Hadoop is too complex for regular users to fully understand all the system parameters and tune them appropriately. Especially when processing a batch of jobs, default Hadoop setting may cause inefficient resource utilization and unnecessarily prolong the execution time. This paper considers an extremely important setting of slot configuration which by default is fixed and static. We proposed an enhanced Hadoop system called FRESH which can derive the best slot setting, dynamically configure slots, and appropriately assign tasks to the available slots. The experimental results show that when serving a batch of MapReduce jobs, FRESH significantly improves the makespan as well as the fairness among jobs.**

I. Introduction

MapReduce has become a popular paradigm for parallel large data processing with no database support. Hadoop, as an open source implementation of MapReduce, has been widely adopted in both academia and industry. We envision that Hadoop-like MapReduce systems will become ubiquitous in the future to serve a variety of applications and customers. It is motivated by two factors. First, big data processing has shown the potential of benefiting many applications and services ranging from financial service, health applications, customer analysis, to social network applications. With more and more daily generated data, powerful processing ability will become the focus for research and development. Second, with the rise of cloud computing, it is now inexpensive and convenient for regular customers to rent a large cluster for data processing. The availability of such elastic environments will certainly expand the potential user group of Hadoop systems. While Hadoop is welcomed in the community, there are still some challenges which, if not solved, may hinder the further development. How to improve the performance in terms of execution times is the top issue on the list, especially when we imagine that a Hadoop cluster will often serve a large volume of jobs in a batch in the future [1]. Researchers have put tremendous efforts on job scheduling, resource management, program design, and Hadoop applications [2]–[10] aiming to improve the system performance. In this paper, we target on an extremely important Hadoop parameter, slot configuration, and develop a suite of solutions to improve the performance, with respect to the makespan of a batch of jobs and the fairness among them.

In a classic Hadoop cluster, each job consists of multiple map and reduce tasks. The concept of "slot" is used to indicate the capacity of accommodating tasks on each node in the cluster. Each node usually has a predefined number of slots and a slot could be configured as either a map slot or a reduce slot. The slot type indicates which type of tasks (map or reduce) it can serve. At any given time, only one task can be running per slot. While the slot configuration is critical for the performance, Hadoop by default uses fixed numbers of map slots and reduce slots at each node throughout the lifetime of a cluster. The values are usually set with heuristic numbers without considering job characteristics. Such static settings certainly cannot yield the optimal performance for varying workloads. Therefore, our main target is to address this issue and improve the makespan performance. Besides the makespan, fairness is another performance metric we consider. Fairness is critical when multiple jobs are allowed to be concurrently executed in a cluster. With different characteristics, each job may consume different amount of system resources. Without a careful plan and management, some jobs may starve while other take advantages and finish the execution much faster. Prior work has studied this issue and proposed some solutions. But we found that the previous work did not accurately define the fairness for this two-phase MapReduce process. In this paper, we present a novel fairness definition that captures the overall resource consumption. Our solution also aims to achieve a good fairness among all the jobs.

Specifically, we propose a new approach, "FRESH", to achieving fair and efficient slot configuration and scheduling for Hadoop clusters. Our solution attempts to accomplish two major tasks: (1) decide the slot configuration, i.e., how many map/reduce slots are appropriate; and (2) assign map/reduce tasks to available slots. The targets of our approach include minimizing the makespan as the major objective and meanwhile improving the fairness without degrading the makespan. FRESH includes two models, static slot configuration and dynamic slot configuration. In the first model, FRESH derives the slot configuration before launching the Hadoop cluster and uses the same setting during the execution just like the conventional Hadoop. In the second model, FRESH allows a slot to change its type after the cluster has been started. When a slot finishes its task, our solution dynamically configures the slot and assigns it the next task. Our experimental results show

that FRESH significantly improves the performance in terms of makespan and fairness in the system .

II. Related Work

Job scheduling is an important direction for improving the performance of a Hadoop system. The default FIFO scheduler cannot work fairly in a shared cluster with multiple users and a variety of jobs. FAIR SCHEDULER [11] and Capacity Scheduler [12] are widely used to ensure each job can get a proper share of the available resources. Both of them consider fairness separately in map phase and reduce phase, but not the overall executions of jobs.

To improve the performance, Quincy [4] and Delay Scheduling [2] optimize data locality in the case of FAIR SCHEDULER. But these techniques trade fairness off against data locality. Coupling Scheduler in [13]–[15] aims to mitigate the starvation of reduce slots in FAIR SCHEDULER and analyze the performance by modeling the fundamental scheduling characteristics for MapReduce. W. Wang [16] presents a new queueing architecture and proposes a map task scheduling to strike the right balance between data-locality and load-balancing. Another category of schedulers consider user-level goals while improving the performance. ARIA [5] allocates the appropriate amounts of resources to the jobs to meet the predefined deadline. iShuffle [17] separates shuffle phase from reduce tasks and provides a platform service to manage and schedule data output from map phase. However, all these techniques are still based on static slot configurations.

Another important direction to improve performance in Hadoop is the resource aware scheduling. RAS [9] aims at improving resource utilization across machines and meeting jobs completion deadline. MROrchestrator [10] introduces an approach to detect task resource utilization at each TaskTracker as a local resource manager and allocate resources to tasks at the JobTracker as a global resource manager. Furthermore, some other work is focused on heterogeneous environments. M. Zaharia et al. proposes a LATE scheduler [18] to stop unnecessary speculative executions in a heterogeneous Hadoop cluster. LsPS [19] uses the present heterogeneous job size patterns to tune the scheduling schemes.

Finally, we have proposed TuMM [20] in our prior work which dynamically adjusts slots configurations in Hadoop based on FIFO. This paper, however, considers concurrent execution of multiple jobs, which is a completely different and more complicated problem setting. We also include a new objective of fairness in this paper. The Hadoop community recently released Next Generation MapReduce (NGM) [7] which offers a resource container instead of fixed-size slots. Our work can certainly be integrated into the Hadoop framework (NGM) for makespan and fairness improvement.

III. System Model and Overall Fairness

We consider that a user submits a batch of n jobs, $J = \{J_1, J_2, \ldots, J_n\}$, to a Hadoop cluster with S slots in total. Each job J_i contains $n_m(i)$ map tasks and $n_r(i)$ reduce tasks. Let s_m and s_r be the total numbers of map slots and reduce slots in the cluster, i.e., $S = s_m + s_r$. We assume that an admission control mechanism is in effect in the cluster such that there is an upper bound limit on the number of jobs that can run concurrently. Specifically, we assume in this paper that at any time, there are at most k jobs running in map phase and at most k jobs running in reduce phase. Thus, the maximum number of active jobs in the cluster is $2k$. Here, k is a user-specified parameter for balancing the trade-off between fairness and makespan. Our objective is to minimize the makespan (i.e., the total completion length) of the job set J while achieving the fairness among these jobs as well.

To solve the problem, we develop a new scheduling solution FRESH for allocating slots to Hadoop tasks. Essentially, we need to address two issues. First, given the total number of slots, how to allocate them for map and reduce, i.e, how many map slots and reduce slots are appropriate. Second, when a slot is available, which task should be assigned to it. FRESH considers two models, i.e., static slot configuration (see Fig. 1) and dynamic slot configuration (see Fig. 2). In the first model, the numbers of map and reduce slots are decided before launching the cluster, similar to the conventional Hadoop system. In this model, we assume that job profiles are available as prior knowledge. Our goal is to derive the best slot setting, thus addressing the first issue. During the execution, FAIR SCHEDULER is used to assign tasks to available slots. In the second model of dynamic slot configuration, FRESH allows a slot to change its type in an online manner and thus dynamically controls the allocation of map and reduce slots. In addition, FRESH includes an algorithm that assigns tasks to available slots.

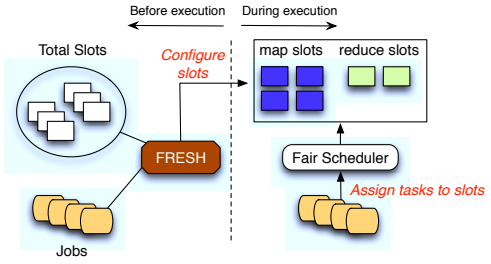

Fig. 1: Static Slot Configuration.

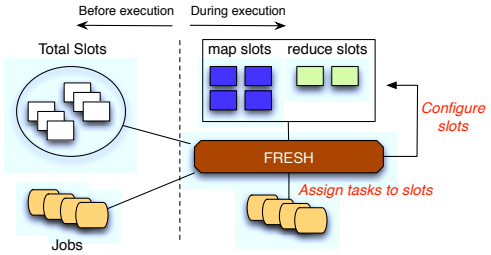

Fig. 2: Dynamic Slot Configuration.

IV. Static Slot Configuration

In this section, we present our algorithm in FRESH for static slot configuration, where the assignments of map and reduce slots are preset in configuration files and loaded when the Hadoop cluster is launched. Our objective is to derive the

optimal slot configuration given the workload profiles of a set of Hadoop jobs.

We assume that the workload of each job is available as prior knowledge. This information can be obtained from historical execution records or empirical estimation. Let $\bar{t}_m(i)$ and $\bar{t}_r(i)$ be the average execution time of a map task and a reduce task of job J_i. We define $w_m(i)$ and $w_r(i)$ as the workloads of map tasks and reduce tasks of J_i, which represent the summation of the execution time of all map tasks and reduce tasks of J_i. Therefore, $w_m(i)$ and $w_r(i)$ can be defined as: $w_m(i) = n_m(i) \cdot \bar{t}_m(i), \quad w_r(i) = n_r(i) \cdot \bar{t}_r(i)$.

Let c_m and c_r represent the number of slots that a job can occupy to run its map and reduce tasks. Recall that FAIR SCHEDULER is used to assign tasks and each active job is evenly allocated slots for its tasks. In addition, under our admission control policy, a busy cluster has k jobs concurrently running in map phase and k jobs concurrently running in reduce phase. Therefore, we have c_m and c_r defined as follows: $c_m = \frac{s_m}{k}, \quad c_r = \frac{s_r}{k}$.

We develop a new algorithm (see Algorithm 1) to derive the optimal static slot configuration. Our basic idea is to enumerate all possible settings of s_m and s_r, and calculate the makespan for any given pair (s_m, s_r). We use $\mathcal{M}$ and $\mathcal{R}$ to represent the sets of jobs that are currently running in their map phase and reduce phase, respectively. Each element in $\mathcal{M}$ and $\mathcal{R}$ is the job index of the corresponding running job. Initially, $\mathcal{M} = \{1, 2, \ldots, k\}$ and $\mathcal{R} = \{\}$. According to their definitions, when job J_i finishes its map phase, the index i will be moved from $\mathcal{M}$ to $\mathcal{R}$. The sizes of $\mathcal{M}$ and $\mathcal{R}$ are upper-bounded by the parameter k. Additionally, we use W_i to represent the *remaining workload* of J_i in the current phase (either map or reduce phase). Before J_i enters the map phase, W_i is set to its workload of the map phase $w_m(i)$, i.e., $W_i = w_m(i)$. During the execution in the map phase, W_i will be updated according to the progress. When job J_i finishes its map phase, W_i will be set to its workload of the reduce phase, i.e., $W_i = w_r(i)$.

Algorithm 1 presents the details of our solution. The outer loop enumerates all possible slot configurations (i.e., s_m and s_r). For each particular configuration, we first calculate the workloads of each job's map and reduce phases, i.e., $w_m(i)$ and $w_r(i)$, and set the initial value of W_i (see lines 3–5). In line 6, we initialize some important variables, where $\mathcal{M}$ and $\mathcal{R}$ are as defined above, $\mathcal{R}'$ represents an ordered list of pending jobs that have finished their map phase, but have not entered their reduce phase yet, and T, initialized as zero, records the makespan of this set of jobs. The core component of the algorithm is the while loop (see lines 7–22) that calculates the makespan and terminates when both $\mathcal{M}$ and $\mathcal{R}$ are empty. In this loop, our algorithm mimics the execution order of all the jobs. Both $\mathcal{M}$ and $\mathcal{R}$ keep intact until one of the running jobs finishes its current map (or reduce) phase. In each round of execution in the while loop, our algorithm finds the first job that changes the status and then updates the job sets accordingly. This target job could be in either map or reduce phase depending on its *remaining workload* and the number of slots assigned to each job. In Algorithm 1, lines 8–9 find two jobs (J_u and J_v) which have the minimum remaining workloads in $\mathcal{M}$ and $\mathcal{R}$, respectively. These two jobs are the candidates to first finish their current phases. Variables c_m and c_r represent the number of slots assigned to each of them under FAIR SCHEDULER scheduling policy. Thus, the remaining execution times for J_u and J_v to complete their phases are $\frac{W_u}{c_m}$ and $\frac{W_v}{c_r}$, respectively.

Algorithm 1 Static Slot Configuration

```
1: for s_m = 1 to S do
2:     s_r = S − s_m
3:     for i = 1 to n do
4:         w_m(i) = n_m(i)·t̄_m(i),   w_r(i) = n_r(i)·t̄_r(i)
5:         W_i = w_m(i)
6:     M = {1, 2, ..., k}, R = {}, R' = {}, T = 0
7:     while M ⋃ R ≠ φ do
8:         u = argmin_{i∈M} W_i,  c_m = s_m / |M|
9:         v = argmin_{i∈R} W_i,  c_r = s_r / |R|
10:        if W_u/c_m < W_v/c_r then
11:            M ← M − u,   T = T + W_u/c_m,   W_u = w_r(u)
12:            DeductWorkload(M, w_m(u))
13:            DeductWorkload(R, (w_m(u)/c_m)·c_r)
14:            pick a new job from J and add its index to M
15:            if |R| < k then R ← R + u
16:            else add u to the tail of R'
17:        else
18:            R ← R − v,   T = T + W_v/c_r
19:            DeductWorkload(R, W_v)
20:            DeductWorkload(M, (W_v/c_r)·c_m)
21:            if |R'| > 0 then move R'[0] to R
22:    if T < Opt_MS then Opt_MS = T, Opt_SM = s_m
23: return Opt_SM and Opt_MS
```

If J_u finishes its map phase first (the case in lines 10–16), then we remove u from $\mathcal{M}$, update the current makespan, and set the remaining workload of J_u to the workload of its reduce phase (line 11). We also update the remaining workloads of all other active jobs in $\mathcal{M}$ and $\mathcal{R}$ (lines 12–13). In addition, the algorithm picks a new job to enter its map phase in line 14. Finally, we add u to $\mathcal{R}$ to start its reduce phase if the capacity limit of $\mathcal{R}$ is not reached. Otherwise, u is added to the tail of the pending list $\mathcal{R}'$ (lines 15–16).

The function `DeductWorkload` is called to update the remaining workloads for active jobs in $\mathcal{M}$ or $\mathcal{R}$. As shown below, the inputs of this function include a job set $\mathcal{A}$ (e.g., $\mathcal{M}$, $\mathcal{R}$) and the value of the completed workload w. The remaing workload of each job i in $\mathcal{A}$ is then updated by deducting w.

```
function DeductWorkload(A, w){
    /*A: a set of job IDs, w: a workload value */
    for i ∈ A do W_i ← W_i − w end }
```

Once job J_v finishes its reduce phase (see the other case in lines 17–21), we update the current makespan as well as the remaining workloads of all other active jobs in $\mathcal{M}$ and $\mathcal{R}$. Similarly, index v is removed from $\mathcal{R}$. If $\mathcal{R}'$ is now not empty,

then the first job in $\mathcal{R}'$ will be moved to $\mathcal{R}$. At the end, in lines 22–23, the algorithm compares the present makespan T to the variable Opt_MS which keeps track of the minimum makespan, and updates Opt_MS if needed. Another auxiliary variable Opt_SM is used to record the corresponding slot configuration. The time complexity of Algorithm 1 is $O(S \cdot N_T^2)$, where $N_T = \sum_i (n_m(i) + n_r(i))$ is the total number of tasks of all the jobs. In practice, the computation overhead of Algorithm 1 is quite small. For example, with 500 slots and 100 jobs each having 400 tasks, the computation overhead is 0.578 seconds on a desktop server with 2.4GHz CPU.

V. Dynamic Slot Configuration

In this section, we turn to discuss the model in FRESH for dynamic slot configuration. The critical target of this model is to enable a slot to change its type (i.e., map or reduce) after the cluster is launched. To accomplish it, we develop solutions for both configuring slots and assigning tasks to slots. In addition, we redefine the fairness of resource consumption among jobs. Therefore, our goal is to minimize the makespan of jobs while achieving the best fairness without degrading the makespan performance. The rest of this section is organized as follows. We first introduce the new *overall fairness* metric. Then we present the algorithm for dynamically configuring map and reduce slots. Finally, we describe how FRESH assigns tasks to the slots in the cluster.

A. Overall Fairness Measurement

Fairness is an important performance metric for our algorithm design. However, the traditional fairness definition does not accurately reflect the total resource consumptions of jobs. In this subsection, we present a new approach to quantify fairness measurement, where we define the resource usage in MapReduce process and use Jain's index [21] to represent the level of fairness.

In a conventional Hadoop system, FAIR SCHEDULER evenly allocates the map (resp. reduce) slots to active jobs in their map (resp. reduce) phases. Although the fairness is achieved in map and reduce phases separately, it does not guarantee the fairness among all jobs when we combine the resources (slots) consumed in both map and reduce phases. For example, assume a cluster has 4 map slots and 4 reduce slots running the following 3 jobs: J_1 (2 map tasks and 9 reduce tasks), J_2 (3 map tasks and 4 reduce tasks), and J_3 (7 map tasks and 3 reduce tasks). Assume every task can be finished in a unit time. The following table shows the slot assignment with FAIR SCHEDULER at the beginning of each time point ('M' and 'R' indicate the type of slots allocated for the jobs). Eventually, all three jobs are finished in 5 time units. However, they occupy 11, 7, and 10 slots respectively.

	0	1	2	3	4
J_1	2(M)	4(R)	2(R)	1(R)	2(R)
J_2	1(M)	2(M)	2(R)	1(R)	1(R)
J_3	1(M)	2(M)	4(M)	2(R)	1(R)

In this paper, we define a new fairness metric, named *overall fairness*, as follows. At any time point T, let J' represents the set of currently active jobs in the system, and T_i represents the starting time of job J_i in J'. We use two matrices $t_m[i,j]$ and $t_r[i,j]$ to represent the execution times of job J_i's j-th map task and j-th reduce task, respectively. Note that these two matrices include the unfinished tasks. Therefore, the resources consumed by job J_i by time T can be expressed as

$$r_i(T) = \frac{\sum_j t_m[i,j] + \sum_j t_r[i,j]}{T - T_i}. \quad (1)$$

where the above formula represents the effective resources J_i has consumed during the period of $T - T_i$. The bigger $r_i(T)$ is, the more resources J_i has been assigned to. In addition, we use Jain's index on r_i to indicate the overall fairness ($F(T)$) at the time point T, i.e., $F(T) = \frac{(\sum_i r_i(T))^2}{|J'| \sum_i r_i^2(T)}$, where $F(T) \in [\frac{1}{|J'|}, 1]$ and a larger value indicates better fairness.

B. Configure Slots

The function of configuring slots is to decide how many slots should serve map/reduce tasks based on the current situation. Specifically, when a task is finished and a slot is freed, our system needs to determine the type of this available slot in order to serve other tasks. In this subsection, we present the algorithm in FRESH that appropriately configures map and reduce slots.

First of all, our solution makes use of the statistical information of the finished tasks from each job. This information is available in Hadoop system variables and log files. Let $\bar{t}_m(i)$ and $\bar{t}_r(i)$ be the average execution times of job J_i's map task and reduce task, respectively. Once a task completes, we can access its execution time and then update $\bar{t}_m(i)$ or $\bar{t}_r(i)$ for job J_i which that particular task belongs to. In addition, we use $n'_m(i)$ and $n'_r(i)$ to indicate the number of remaining map tasks and reduce tasks, respectively, in job J_i. The *remaining workload* of a job J_i is then defined as follows: $w'_m(i) = n'_m(i) \cdot \bar{t}_m(i), w'_r(i) = n'_r(i) \cdot \bar{t}_r(i)$, where $w'_m(i)$ is J_i's remaining map workload and $w'_r(i)$ is the remaining reduce workload of J_i. Finally, we estimate the total remaining workloads of all the pending map and reduce tasks. Let RW_m represents the summation of all the remaining map workloads of jobs in $\mathcal{M}$ while RW_r represents the summation of all the remaining reduce workloads for jobs in $\mathcal{R}$ and $\mathcal{R}'$. RW_m and RW_r can be calculated as:

$$RW_m = \sum_{i \in \mathcal{M}} w'_m(i), \quad RW_r = \sum_{i \in \mathcal{R} \bigcup \mathcal{R}'} w'_r(i).$$

Note that RW_m includes the jobs running in their map phases (in $\mathcal{M}$) while RW_r includes the jobs running in their reduce phases (in $\mathcal{R}$) as well as the jobs that have finished their map phases but wait for running their reduce phase (in $\mathcal{R}'$).

The intuition of the algorithm in FRESH is to keep jobs in their map and reduce phases in a consistent progress so that all jobs can be properly pipelined to avoid waiting for slots or having idle slots. Therefore, the numbers of map and reduce slots should be proportional to the total remaining workloads RW_m and RW_r, i.e., the heavier loaded phase gets more slots to serve its tasks. However, this idea may not work well in the

map-reduce process because there could be a sudden change on the remaining workloads. The problem arises when a job finishes its map phase and enters the reduce phase. Based on the definition of RW_m and RW_r, this job will bring its reduce workload to RW_r and a new job which starts its map phase then add its new map workload to RW_m. Such workload updates, however, could greatly change the weights of RW_m and RW_r. For example, if $RW_m >> RW_r$, most of slots are map slots. Assume a reduce-intensive job just finishes its map phase and incurs a lot of reduce workload, the system has no sufficient reduce slots to serve the new reduce tasks. It takes some time for the cluster to adjust to this sudden workload change as it has to wait for the completions of many map tasks before configuring those released slots to be reduce slots.

We develop Algorithm 2 to derive the optimal slot configuration in an online mode. We follow the basic design principal with a threshold-based control to mitigate the negative effects from those sudden changes in map/reduce workloads. When a map/reduce task is finished, the algorithm collects the task execution time and updates a set of statistical information including the average task execution time, the number of remaining tasks, the remaining workload of job J_i, and the total remaining workloads (see lines 1–4). Following that, the algorithm calls a function, called `CalExpSm`, to calculate the expected number of map slots ($expSm$) based on the current statistical information. If the expectation is more than the current number of map slots (s_m), this free slot will become a map slot. Otherwise, we set it to be a reduce slot.

Algorithm 2 Configure a Free Slot

1: **if** a map task of job J_i is finished **then**
2: update $\bar{t}_m(i)$, $n'_m(i)$, $w'_m(i)$ and RW_m
3: **if** a reduce task of job J_i is finished **then**
4: update $\bar{t}_r(i)$, $n'_r(i)$, $w'_r(i)$ and RW_r
5: $expSm =$`CalExpSm()`
6: **if** $expSm > s_m$ **then** set the slot to be a map slot
7: **else** set the slot to be a reduce slot

The details of the function `CalExpSm` are presented in Algorithm 3. We use θ_{cur} to represent the expected ratio of map slots based on the current remaining workload. In line 2, we choose an active job J_a which has the minimum map workload, i.e., job J_a is supposed to first finish its map phase. If J_a is still far from the completion of its map phase, then the risk of having a sudden workload change is low and the function just returns $\theta_{cur} \cdot S$ as the expected number of map slots. We set a parameter τ_1 as the progress threshold. When job J_a is close to the end of its map phase, i.e., the progress exceeds τ_1, the function will consider the potential issue with the sudden change of the workload (lines 6–13). Essentially, the function tries to estimate the map and reduce workloads when J_a enters its reduce phase, and calculate the expected ratio of map slots θ_{exp} at that point. A sudden change of the workload happens when θ_{exp} is quite different from the ratio of map slots θ_{cur} we get based on the current configuration. In the case that a sudden change is predicted, we will use $\theta_{exp} \cdot S$ instead as the guideline for new slot configuration. Otherwise, the function still returns $\theta_{cur} \cdot S$.

Specifically in Algorithm 3, when J_a finishes its remaining map workload $w'_m(a)$, we assume the other jobs in $\mathcal{M}$ have made roughly the same progress. Thus the total map workload will be reduced by $w'_m(a) \cdot k$. Then a new job will join the set $\mathcal{M}$, let it be job J_b, and $w_m(b)$ will be added to the total remaining workload RW_m (line 7). Meanwhile, J_a will belong to either set $\mathcal{R}$ or $\mathcal{R}'$ and its reduce workload $w_r(i)$ will be added to the total remaining reduce workload RW_r (line 8). Variable θ_{exp} in line 9 denotes the expected ratio of map slots at that point. Next, the function estimates the number of map slots following the configuration ratio θ_{cur} when J_a finishes its map phase. It involves the number of slots freed from the current point. It is apparent that there is no other map slot released based on the definition of J_a, thus we just need to estimate the number of available reduce slots during this period. In Algorithm 3, we use η to represent this number and estimate its value based on the following Theorem 1. In line 11, we predict the number of map slots s'_m using the current configuration ratio, i.e., $\eta \cdot \theta_{cur}$ slots will become map slots. Eventually, the function compares the estimated ratio to the expected ratio in line 12. If the difference is over a threshold τ_2, we will consider the future expected ratio θ_{exp}. Otherwise, we will continue to use the current configuration ratio θ_{cur}.

Algorithm 3 Function $CalExpSm()$

1: $\theta_{cur} = \frac{RW_m}{RW_m + RW_r}$
2: $a = \text{argmin}_{i \in \mathcal{M}} w'_m(i)$
3: **if** the progress of job $J_a < \tau_1$ **then**
4: return $\theta_{cur} \cdot S$
5: **else**
6: Let job J_b be the next that will start its map phase
7: $RW'_m = RW_m - w'_m(a) \cdot k + w_m(b)$
8: $RW'_r = RW_r + w_r(a)$
9: $\theta_{exp} = RW'_m/(RW'_r + RW'_m)$
10: calculate η using Theorem 1
11: $s'_m = s_m + \theta_{cur} \cdot \eta$
12: **if** $\frac{|\frac{s'_m}{S} - \theta_{exp}|}{\theta_{exp}} > \tau_2$ **then** return $\theta_{exp} \cdot S$
13: **else** return $\theta_{cur} \cdot S$

Theorem 1: Assume reduce tasks are finished at a rate of one per r time units, the number (η) of available reduce slots when J_a finishes its remaining map workload is equal to:

$$\eta = \frac{\sqrt{m_a^2 + 4 \cdot c \cdot w'_m(a)} - m_a}{2 \cdot c \cdot r}.$$

where $c = \frac{\theta_{cur}}{2 \cdot r \cdot k}$, $w'_m(a)$ indicates the remaining map workload of J_a, and m_a is the number of map slots assigned to J_a.

Proof: Assume job J_a will finish its remaining map workload $w'_m(a)$ in x time units. According to our assumption, a reduce task will be finished every r time units. Therefore, when J_a finishes its map phase, there will be $\eta = \frac{x}{r}$ reduce slots released as well. Assume that we use θ_{cur} to allocate slots and k jobs in $\mathcal{M}$ are evenly assigned newly released slots. Job J_a will continuously obtain $\frac{x \cdot \theta_{cur}}{r \cdot k}$ new map slots. It is equivalent to having half of them $\frac{x \cdot \theta_{cur}}{2 \cdot r \cdot k}$ from the beginning.

Therefore, the remaining time for J_a to finish the map phase can be estimated as $x = w'_m(a)/(\frac{x \cdot \theta_{cur}}{2 \cdot r \cdot k} + m_a)$, where m_a is the number of map slots currently assigned to J_a. By solving the above equation, we have $x = \frac{\sqrt{m_a^2 + 4 \cdot c \cdot w'_m(a)} - m_a}{2 \cdot c}$. Thus $\eta = \frac{\sqrt{m_a^2 + 4 \cdot c \cdot w'_m(a)} - m_a}{2 \cdot c \cdot r}$. ■

C. Assign tasks to slots

Once the type of the released slot is determined, FRESH will assign a task to that slot. Basically, we need to select an active job and let the available map/reduce slot serve a map/reduce task from that job. In FRESH, we follow the basic idea in FAIR SCHEDULER but use the new *overall fairness* metric instead: calculate the resource consumption for each job based on Eq.(1) and choose the job with the most deficiency of *overall fairness*.

Algorithm 4 Assign a Task to a Slot

```
1:  Initial: C = {}, now ← current time in system
2:  if the slot is configured for map tasks then C ← M
3:  else C ← R
4:  for each job J_i ∈ C do
5:      total_i = 0
6:      for each task j in job J_i do
7:          if task j is finished then e_j ← f_j − s_j
8:          else if task j is running then e_j ← now − s_j
9:          else e_j ← 0
10:         total_i = total_i + e_j
11:     r_i = total_i / (now − T_i)
12: s = argmin_{i∈A} r_i
13: assign a task of job J_s to the slot
```

Algorithm 4 illustrates our solution in FRESH for assigning a task to an available slot. We use $\mathcal{C}$ to indicate a set of candidate jobs. Initially, $\mathcal{C}$ is empty and we use variable now to indicate the current time. When a slot is configured to serve map tasks (using Algorithm 2), $\mathcal{C}$ is a copy of $\mathcal{M}$. Otherwise, $\mathcal{C}$ is a copy of $\mathcal{R}$ (lines 2–3). The outer loop (lines 4–11) then calculates the resource consumption for each job in $\mathcal{C}$ at the current time. Variable $total_i$, initialized as 0 in line 5, is used to record the total execution time for all finished and running tasks in job J_i (lines 6–10). We use e_j to denote the execution time of task j. If task j has been finished, its execution time e_j is the difference between its finish time f_j and its start time s_j (line 7). If task j is still running, then e_j is equal to the current time now deducted by its start time s_j (line 8). Once the total execution time for job J_i is obtained, we get the resources consumption r_i by normalizing the total execution time $total_i$ by the duration between the current time and the start time of job J_i (T_i), as shown in line 11. Finally, the job with the minimum resource consumption is chosen to be served by the available slot.

VI. PERFORMANCE EVALUATION

In this section, we evaluate the performance of FRESH and compare it with other alternative schemes. We use FRESH-Static and FRESH-Dynamic to represent our static slot configuration and dynamic slot configuration respectively.

A. Experimental Setup and Workloads

1) Implementation: We have implemented FRESH on Hadoop version 0.20.2. For FRESH-static, we develop an external program to derive the best slot setting and apply it to the Hadoop configuration file. The Hadoop system itself is not modified for FRESH-static. To implement FRESH-dynamic, we have added a few new components to Hadoop. First, we implement the admission control policy with the parameter k, i.e., at most k jobs are allowed to be concurrently running in map phase or in reduce phase. Second, we create two new modules in $JobTracker$. One module updates the statistical information such as the average execution time of a task, and estimates the remaining workload of each active job. The other module is designed to configure a free slot to be a map slot or a reduce slot and assign a task to it according to the algorithms in section V. The two threshold parameters in Algorithm 3 are set as $\tau_1 = 0.8$ and $\tau_2 = 0.6$. We have tested with different values and found that the performance is close when $\tau_1 \in [0.7, 0.9]$ and $\tau_2 \in [0.5, 0.7]$. Due to the page limit, we omit the discussion about these two heuristic values. In addition, job profiles (execution time of tasks) are generated based on the experimental results when a job is individually executed. However, we randomly introduce $\pm 30\%$ bias to the measured execution time and use them as the job profiles representing rough estimates.

2) Hadoop Cluster and Workloads: All the experiments are conducted on Amazon AWS EC2 platform. We use two clusters: one with 10 slave nodes and the other with 20 slave nodes (m1.xlarge instances [22]). Each node is configured with 4 slots since an m1.xlarge instance has 4 virtual cores. Totally, there are either $S = 40$ slots or $S = 80$ slots in a cluster in our experiments. Our workloads for evaluation consider general Hadoop benchmarks with large datasets as the input. In particular, four datasets are used in our experiments: 4GB/8GB wiki category links data, and 4GB/8GB movie rating data. The wiki data include the information about wiki page categories and the movie rating data are the user rating information. We choose the following six Hadoop benchmarks from Purdue MapReduce Benchmarks Suite [23] to evaluate the performance: (1) *Classification*: Take the movie rating data as input and classify the movies based on their ratings; (2) *Invertedindex*: Take a list of Wikipedia documents as input and generate word to document indexing; (3) *Wordcount*: Take a list of Wikipedia documents as input and count the occurrences of each word; (4) *Grep*: Take a list of Wikipedia documents as input and search for a pattern in the files; (5) *Histogram Rating*: Generate a histogram of the movie rating data (with 5 bins); (6) *Histogram Movies*: Generate a histogram of the movie rating data (with 8 bins).

B. Performance Evaluation

Given a batch of MapReduce jobs, our performance metrics are makespan and fairness among all jobs. We mainly compare to the conventional Hadoop system with FAIR SCHEDULER. In our setting, each slave has 4 slots, thus there are three possible static settings in conventional Hadoop in terms of

the ratio of map/reduce slots. We use *Fair-1:3*, *Fair-2:2*, and *Fair-3:1* to represent these three settings respectively. We have conducted two categories of tests with different workloads: *simple workloads* consist of the same type of jobs (selected from the six MapReduce benchmarks), and *mixed workloads* represent a set of hybrid jobs. In the reset of this subsection, we separately present the evaluation results with these two categories of workloads.

1) Performance with Simple Workloads: For testing *simple workloads*, we generate 10 Hadoop jobs for each of the above 6 benchmarks. Every set of 10 jobs share the same input data set and they are consecutively submitted to the Hadoop cluster with an interval of 2 seconds. In addition, we have tested different values of k to show the effect of the admission control policy, particularly $k = 1, 3, 5, 10$.

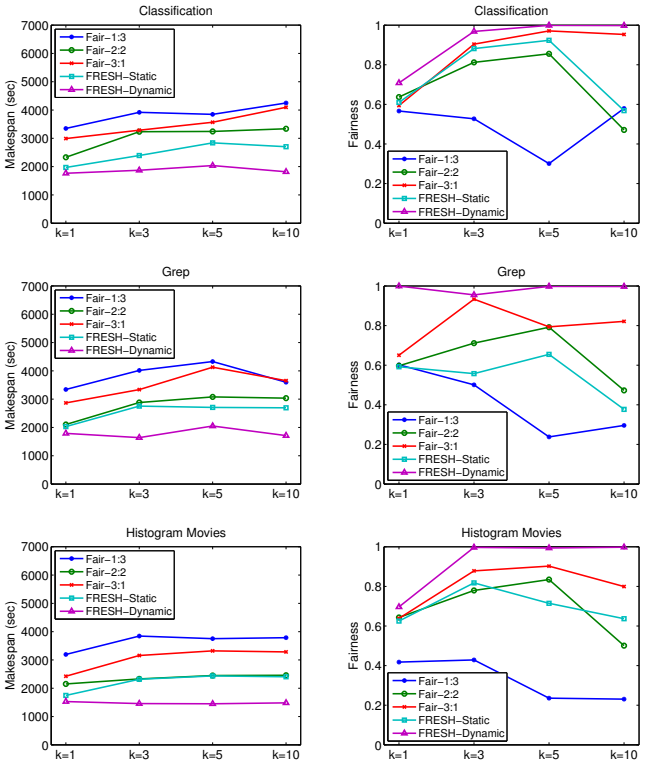

Fig. 3: Makespan and fairness of simple workloads under FRESH and FAIR SCHEDULER (with 10 slave nodes)

Due to the page limit, we show the evaluation results with three benchmarks in Fig. 3. For the performance of makespan, we observe that in conventional Hadoop, the best makespan is achieved mostly when $k = 1$, i.e., only one job in map and reduce phase which is equivalent to FIFO(First-In-First-Out) scheduler. It indicates that while improving the fairness, FAIR SCHEDULER sacrifices the makespan of a set of jobs. FRESH-static performs no worse than the best setting with FAIR SCHEDULER. In some workload such as "Classification", the improvement is adequate. In addition, FRESH-dynamic always yields a significant improvement in all the tested settings. For example, FRESH-Dynamic improves about 32.75% in makespan compared to FAIR SCHEDULER with slot ratio 2:2. The performance of fairness, in most of the cases, is an increasing function on k. Especially when $k = 10$, where all jobs are allowed to be concurrently executed (no admission control), almost all schemes obtain a good fairness value. Since all jobs are from the same benchmark in *simple workloads*, they finish their map phases in wave and enter the reduce phase roughly in the same time. Thus FAIR SCHEDULER performs well in this case (k=10). Overall, FRESH-Dynamic outperforms all other schemes and achieves very-close-to-1 fairness value even when $k = 3$ or $k = 5$.

2) Performance with Mixed Workloads: Additionally, we evaluate the performance with mixed workloads consisting of different benchmarks. We specifically form five sets (Set A to Set E) of mixed jobs whose details will be introduced when we present the evaluation results. For mixed workloads, the order of the jobs has a big impact on the performance. In our experiments, after generating all the jobs, we conduct the Johnsons's algorithm [24], which can build an optimal two-stage job schedule, to determine the order of the jobs.

TABLE I: Set A (12 Mixed Jobs)

	Benchmark	Dataset	Map #	Reduce #
01	Classification	8GB Movie Rating Data	270	250
02	Classification	4GB Movie Rating Data	129	120
03	Invertedindex	8GB Wikipedia	256	250
04	Invertedindex	4GB Wikipedia	128	120
05	Wordcount	8GB Wikipedia	256	200
06	Wordcount	4GB Wikipedia	128	100
07	Grep[a-g][a-z]*	8GB Wikipedia	270	250
08	Grep[a-g][a-z]*	4GB Wikipedia	128	100
09	Histogram_ratings	8GB Movie Rating Data	270	250
10	Histogram_ratings	4GB Movie Rating Data	129	120
11	Histogram_movies	8GB Movie Rating Data	270	200
12	Histogram_movies	4GB Movie Rating Data	129	100

Our mixed workloads are specified as follows. Set A∼C contain equal number of jobs from every benchmark. The details of Set A are listed in Table I. It has two jobs from each benchmark, one job uses 4G data set and the other uses 8G dataset. Set B is a smaller workload, with one job from each benchmark, and all jobs use 8G dataset. In addition, Set C represents a larger workload which doubles the workload of Set A, i.e., 4 jobs from each benchmark. Furthermore, we create Set D and Set E to represent map-intensive and reduce-intensive workloads. Based on our experiments, benchmark "Inverted Index" and "Grep" are reduce-intensive, and "Classification" and "Histogram Rating" are map-intensive. Thus, both Set D and Set E contain 12 jobs, and 8 jobs in Set D are map-intensive and 8 jobs in Set E are reduce-intensive.

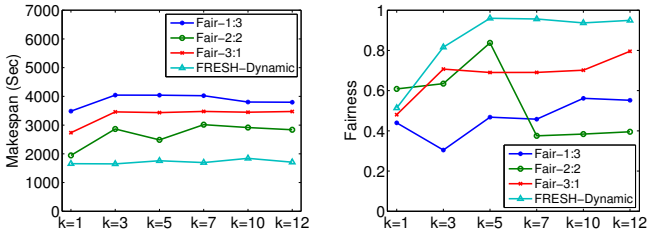

Fig. 4: Makespan and fairness of Set A (with 10 slave nodes)

Fig. 4 shows the makespan and the overall fairness with Set A and different values of k. FRESH-Dynamic's makespan is always superior to FAIR SCHEDULER. We observe that the best makespan of FAIR SCHEDULER is achieved with *Fair-2:2*.

Compare to this best setting, FRESH-Dynamic still improves the makespan by 27.62% when $k = 5$. For fairness, FAIR SCHEDULER performs much worse with the mixed workloads (the best value is around 0.8) than with *simple workloads*, because diverse jobs finish their map phases at different time points. On the other hand, FRESH-Dynamic significantly improves the overall fairness, especially when $k \geq 5$ (with fairness values around 0.95).

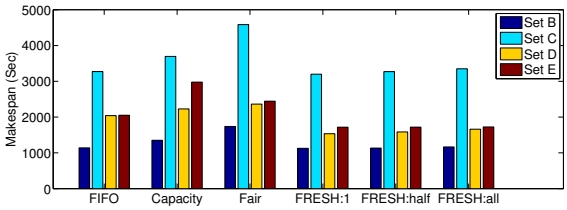

Fig. 5: Makespan of Set B∼E (with 10 slave nodes)

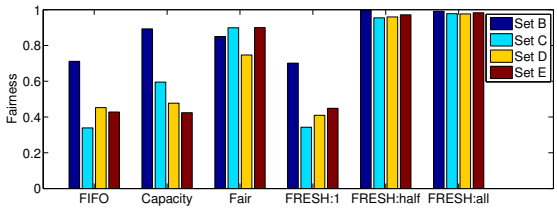

Fig. 6: Fairness of Set B∼E (with 10 slave nodes)

In addition, we present the results with Set B∼E. We test FRESH-Dynamic with three different settings of k: only one job, half of the jobs, and all the jobs, represented by *FRESH:1*, *FRESH:half*, and *FRESH:all* respectively. We compare to the conventional Hadoop with FIFO scheduler, Capacity scheduler [12] and FAIR SCHEDULER. For Capacity scheduler, we create two queues and each queue has the same number of task slots. Each queue can obtain at most 90% slots in the cluster. Also, we separate jobs equally to these queues. The experimental results are illustrated in Fig. 5 and Fig. 6. In Fig. 5, in most of the cases, FAIR SCHEDULER yields the worst performance of makespan with different sets of jobs. On average, FRESH improves 31.32% of makespan compared to the FAIR SCHEDULER and 25.1% to Capacity scheduler. When a set of jobs is neither map-intensive or reduce-intensive (Set B and Set C), FIFO performs as well as FRESH. However, when workloads in map and reduce phases are unbalanced (Set D and Set E), FRESH improves 24.47% of makespan compared to FIFO. Overall, FRESH achieves an excellent and stable makespan performance with different sets of jobs. In Fig. 6, FIFO apparently yields the worst performance of fairness. Both FRESH:half and FRESH:all outperform FAIR SCHEDULER with > 0.95 fairness values in all the tested cases. Finally, we test Set B on a large cluster with 20 slave nodes and the results are shown in Table II. We can observe a consistent performance gain from FRESH as in the smaller cluster of 10 slave nodes. Compared to FAIR SCHEDULER, FRESH reduces the makespan by 31.1%.

TABLE II: Makespan and fairness of Set B with 20 slave nodes

	FIFO	Fair	FRESH:1	FRESH:half	FRESH:all
Makespan	1936s	2263s	1560s	1571s	1590s
Fairness	0.598	0.961	0.575	0.959	0.979

VII. CONCLUSION

This paper studies a Hadoop cluster serving a batch of MapReduce jobs. We target on the slot configuration and task scheduling problems. We develop FRESH, an enhanced version of Hadoop, which supports both static and dynamic slot configurations. In addition, we present a novel definition of the overall fairness. Our solution can yield good makespans while the fairness is also achieved. We have conducted extensive experiments with various workloads and settings. FRESH shows a significant improvement on both makespan and fairness compared to a conventional Hadoop system.

REFERENCES

[1] J. Dean, S. Ghemawat, and G. Inc, "Mapreduce: simplified data processing on large clusters," in *OSDI'04*, 2004.

[2] M. Zaharia, D. Borthakur, J. S. Sarma *et al.*, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *EuroSys'10*, 2010.

[3] A. Verma, L. Cherkasova, and R. H. Campbell, "Two sides of a coin: Optimizing the schedule of mapreduce jobs to minimize their makespan and improve cluster performance," in *MASCOTS'12*, Aug 2012.

[4] M. Isard, Vijayan Prabhakaran, J. Currey *et al.*, "Quincy: fair scheduling for distributed computing clusters," in *SOSP'09*, 2009, pp. 261–276.

[5] A. Verma, L. Cherkasova, and R. H. Campbell, "Aria: Automatic resource inference and allocation for mapreduce environments," in *ICAC'11*, 2011, pp. 235–244.

[6] J. Polo, D. Carrera, Y. Becerra *et al.*, "Performance-driven task co-scheduling for mapreduce environments," in *NOMS'10*, 2010.

[7] Next generation mapreduce scheduler. [Online]. Available: http://goo.gl/GACMM

[8] X. W. Wang, J. Zhang, H. M. Liao, and L. Zha, "Dynamic split model of resource utilization in mapreduce," ser. DataCloud-SC '11, 2011.

[9] J. Polo, C. Castillo, D. Carrera *et al.*, "Resource-aware adaptive scheduling for mapreduce clusters," in *Proceedings of the 12th ACM/IFIP/USENIX international conference on Middleware*, 2011.

[10] B. Sharma, R. Prabhakar, S.-H. Lim *et al.*, "Mrorchestrator: A fine-grained resource orchestration framework for mapreduce clusters," in *IEEE 5th International Conference on Cloud Computing*, 2012.

[11] Fair scheduler. [Online]. Available: http://hadoop.apache.org/common/docs/r1.0.0/fair_scheduler.html

[12] Capacity scheduler. [Online]. Available: http://hadoop.apache.org/common/docs/r1.0.0/capacity_scheduler.html

[13] J. Tan, X. Meng, and L. Zhang, "Performance analysis of coupling scheduler for mapreduce/hadoop," IBM T. J. Watson Research Center, Tech. Rep., 2012.

[14] ——, "Delay tails in mapreduce scheduling," in *SIGMETRICS'12*, 2012.

[15] ——, "Coupling task progress for mapreduce resource-aware scheduling," in *INFOCOM'13*, 2013, pp. 1618–1626.

[16] W. Wang, K. Zhu, L. Ying *et al.*, "Map task scheduling in mapreduce with data locality: Throughput and heavy-traffic optimality," in *INFOCOM'13*, 2013, pp. 1609–1617.

[17] Y. Guo, J. Rao, and X. Zhou, "ishuffle: Improving hadoop performance with shuffle-on-write," *ICAC'13*, 2013.

[18] M. Zaharia, A. Konwinski, A. D. Joseph *et al.*, "Improving mapreduce performance in heterogeneous environments," in *OSDI'08*, 2008.

[19] Y. Yao, J. Tai, B. Sheng, and N. Mi, "Scheduling heterogeneous mapreduce jobs for efficiency improvement in enterprise clusters," in *IM'13*, 2013.

[20] Y. Yao, J. Wang, B. Sheng, and N. Mi, "Using a tunable knob for reducing makespan of mapreduce jobs in a hadoop cluster," in *CLOUD'13*, 2013.

[21] R. K. Jain, D.-M. W. Chiu, and W. R. Hawe, "A quantitative measure of fairness and discrimination for resource allocation in shared systems," Digital Equipment Corporation, Tech. Rep., Dec. 1984.

[22] "Amazon EC2 Instances," http://aws.amazon.com/ec2/instance-types/.

[23] Purdue mapreduce benchmarks suite. [Online]. Available: http://web.ics.purdue.edu/~fahmad/benchmarks.htm

[24] S. M. Johnson, "Optimal two- and three-stage production schedules with setup times included," *Naval Research Logistics Quarterly*, vol. 1, no. 1, pp. 61–68, 1954.