# Two Sides of a Coin: Optimizing the Schedule of MapReduce Jobs to Minimize Their Makespan and Improve Cluster Performance

Abhishek Verma
*University of Illinois at Urbana-Champaign, IL, US.*
verma7@illinois.edu

Ludmila Cherkasova
*Hewlett-Packard Labs Palo Alto, CA, US.*
lucy.cherkasova@hp.com

Roy H. Campbell
*University of Illinois at Urbana-Champaign, IL, US.*
rhc@illinois.edu

***Abstract*—Large-scale MapReduce clusters that routinely process petabytes of unstructured and semi-structured data represent a new entity in the changing landscape of clouds. A key challenge is to increase the utilization of these MapReduce clusters. In this work, we consider a subset of the production workload that consists of MapReduce jobs with no dependencies. We observe that the order in which these jobs are executed can have a significant impact on their overall completion time and the cluster resource utilization. Our goal is to automate the design of a job schedule that minimizes the completion time (makespan) of such a set of MapReduce jobs. We offer a novel abstraction framework and a *heuristic*, called *BalancedPools*, that efficiently utilizes performance properties of MapReduce jobs in a given workload for constructing an optimized job schedule. Simulations performed over a realistic workload demonstrate that 15%-38% makespan improvements are achievable by simply processing the jobs in the *right* order.[1]**

***Keywords*-MapReduce, Hadoop, batch workloads, optimized schedule, minimized makespan.**

## I. Introduction

As cloud computing continues to evolve, an increasing number of companies are exploiting the MapReduce paradigm and its open-source implementation Hadoop as a scalable platform for *Big Data* processing. The data analysis applications range in functionality, complexity, resource needs, and data delivery deadlines. This diversity creates competing requirements for program design, job scheduling, and workload management policies in MapReduce environments. However, in spite of different user objectives one goal is common: to improve the usability and performance of the MapReduce framework.

The job execution efficiency is particularly important for processing production workloads when a given set of MapReduce jobs and workflows needs to be executed periodically on new data. Typically, the default FIFO scheduler is used for processing production jobs since the primary performance objective is to *minimize the overall execution time (makespan)* of a given set. Such production workloads are analyzed off-line for optimizing their execution. To ease the task of writing complex analytics programs, high-level SQL-like abstractions such as Pig and Hive have been proposed. There is a slew of optimization methods introduced for improving data read/write efficiency in a set of production jobs. For different MapReduce jobs operating over the same dataset, a more efficient job scheduling [1] proposes merge their executions so that the input data is only scanned once.

In this work, we consider a subset of a production workload formed by the jobs with no dependencies. Such independent jobs arise, for example, while processing different datasets, or optimized Pig/Hive queries resulting in a single MapReduce job[2]. We discuss a different cause for a job execution inefficiency inherent to the MapReduce computation that processes map and reduce tasks separated by a synchronization barrier. The order in which jobs are executed can have a significant impact on the overall processing time, and therefore, on the achieved cluster utilization. For data-dependent jobs, the successive job can only start after the current one is entirely finished. However, for data-independent jobs, once the previous job completes its map stage and begins the reduce stage, the next job can start executing its map stage with the released map resources in a pipelined fashion. Thus, there is an overlap in job executions when different jobs use complementary cluster resources: map and reduce slots. Note that a larger overlap in job executions leads to better job pipelining, increased cluster utilization, and an improved execution time, while using the same number of machines (and thus for free).

We introduce a simple abstraction of a MapReduce job as a pair of its map and reduce stage durations. This representation enables us to apply the classic Johnson algorithm [2] that was designed for building an optimal two-stage job schedule. Since the set of production jobs is executed periodically, it permits their automated profiling from *past executions*. When jobs in a batch need to process new datasets, we use the knowledge of extracted job profiles to pre-compute new estimates of jobs' map and reduce stage durations, and then construct an optimized schedule for *future executions*. We evaluate performance benefits of the constructed schedule through extensive simulations over a variety of realistic workloads. The performance results are workload and cluster-size dependent, but we typically achieve up to 10%-25% makespan improvements.

However, the proposed abstraction obscures the amount of resources each job may be able to utilize, and in some cases, Johnson's schedule may lead to a suboptimal makespan. We design *BalancedPools*, a *novel heuristic* that efficiently uti-

[1]This work was completed during A. Verma's internship at HP Labs. R. Campbell and A. Verma are supported in part by NSF CCF grants #0964471, IIS #0841765 and Air Force Research grant FA8750-11-2-0084.

[2]11 out of 17 PigMix queries (http://wiki.apache.org/pig/PigMix) translate to a single independent MapReduce job.

lizes characteristics and properties of MapReduce jobs in a given workload for constructing the optimized job schedule. We evaluate the performance benefits of the constructed schedule through simulations over a variety of realistic workloads. The detailed evaluation of the proposed heuristic demonstrates makespan improvements of up to 15%-38% for situations where Johnson's schedule is suboptimal. The results of our simulation study are validated through experiments on a 66-node Hadoop cluster. The remainder of the paper presents our results in more detail.

## II. Background: Job Profiles and MapReduce Performance Model

This work continues a direction initiated in ARIA [3]. This model can be used for predicting the completion time of the map and reduce stages as a function of the input dataset size and allocated resources.

ARIA introduces a MapReduce performance model that is based on useful *theoretical performance bounds* described as follows. Let us consider a job that is represented as a set of $n$ tasks processed by $k$ servers (or by $k$ slots in MapReduce environments). The assignment of tasks to slots is done using an online, *greedy* algorithm: assign each task to the slot which finished its running task the earliest. Let $avg$ and $max$ be the *average* and *maximum* duration of the $n$ tasks respectively. Then the job makespan (the job completion time) under the greedy task assignment is rticle traffic logs that were collected (and compressed) every hour. *WikiTrends* counts the number of times each article has been visited in the given input dataset, i.e., the article access frequency (popularity count) over time. The example's input dataset has 71 files that correspond to 71 map tasks, and the application is defined with 64 reduce tasks. Figure 1 shows the WikiTrends execution with 16 map and 16 reduce slots. Therefore, the job execution has 5 map waves (that comprise the map stage) and 4 reduce waves (that constitute the reduce stage). The first shuffle may overlap with a significant portion of the map stage. At the same time, there is a strict barrier between map and reduce task processing: a reduce task execution may only start when all map tasks are completed and the intermediate data has been shuffled to the reducer. We aim to define the job execution time as **the sum of the complementary, non-overlapping map and reduce stage execution times**.

Let us consider job $J$ that is partitioned into $N_M^J$ map tasks and $N_R^J$ reduce tasks. Let $J$ be already executed in a given Hadoop cluster. Below, we explain both our profiling approach and the proposed MapReduce performance model for estimating the job completion time as a function of allocated resources. Let $S_M^J$ and $S_R^J$ be the number of map and reduce slots allocated to the **future** execution of job $J$.

The **map stage** consists of a number of map tasks. If the number of tasks is greater than the number of slots, the task assignment proceeds in multiple rounds, which we call as *waves*. From the distribution of the map task durations of the past run, we compute the average duration $M_{avg}$ and the maximum duration $M_{max}$. Then the lower and upper bounds on the duration of the entire map stage in the future execution with $S_M^J$ map slots (denoted as $T_M^{low}$ and $T_M^{up}$ respectively) are estimated as follows:

$$T_M^{low} = N_M^J / S_M^J \cdot M_{avg}$$

$$T_M^{up} = (N_M^J - 1)/S_M^J \cdot M_{avg}/S_M^J + M_{max}$$

The **reduce stage** consists of the *shuffle* and *reduce* phases, and their execution time bounds can be computed similarly.

The *shuffle phase* begins only after the first map task has completed. The shuffle phase completes when the entire map stage is complete and all the intermediate data generated by the map tasks has been shuffled to the reduce tasks and has been sorted. The shuffle phase of the *first* reduce wave may be significantly different from the shuffle phase that belongs to the next reduce waves. This happens because the shuffle phase of the first reduce wave overlaps with the entire map stage, and hence its depends on the number of map waves and their durations. Therefore, from the past execution, we extract two sets of measurements: $(Sh_{avg}^1, Sh_{max}^1)$ for shuffle phase of the first reduce wave (called, *first shuffle*) and $(Sh_{avg}^{typ}, Sh_{max}^{typ})$ for shuffle phase of the other waves (called, *typical shuffle*). Moreover, we characterize a first shuffle in a special way and include only the non-overlapping portion (with map stage) in our metrics: $Sh_{avg}^1$ and $Sh_{max}^1$. This way, we carefully estimate the latency portion that contributes explicitly to the job completion time. The *typical shuffle* phase is computed as follows:

$$T_{Sh}^{low} = \left(N_R^J / S_R^J - 1\right) \cdot Sh_{avg}^{typ}$$

$$T_{Sh}^{up} = \left((N_R^J - 1)/S_R^J - 1\right) \cdot Sh_{avg}^{typ} + Sh_{max}^{typ}$$

The *reduce phase* begins only after the shuffle phase is complete. From the distribution of the reduce task durations of the past run, we compute the *average* and *maximum* metrics: $R_{avg}$ and $R_{max}$ that are used to compute the lower and upper bounds of completion times of the reduce phase.

Finally, we can put together the lower and upper bounds of the entire reduce stage ($T_R^{low}$, $T_R^{up}$) by summing up durations of shuffle and reduce phases.

$$T_R^{low} = Sh_{avg}^1 + T_{Sh}^{low} + (N_R^J \cdot R_{avg}/S_R^J)$$

$$T_R^{up} = Sh_{max}^1 + T_{Sh}^{up} + ((N_R^J - 1) \cdot R_{avg}/S_M^J + R_{max})$$

Typically, the *average of lower and upper bounds* is a good approximation of the stage completion time:

$$T_M^{avg} = (T_M^{low} + T_M^{up})/2 \quad \text{and} \quad T_R^{avg} = (T_R^{low} + T_R^{up})/2.$$

For the rest of the paper, we use $T_M^{avg}$ and $T_R^{avg}$ as the estimates of the map and reduce stage execution time.

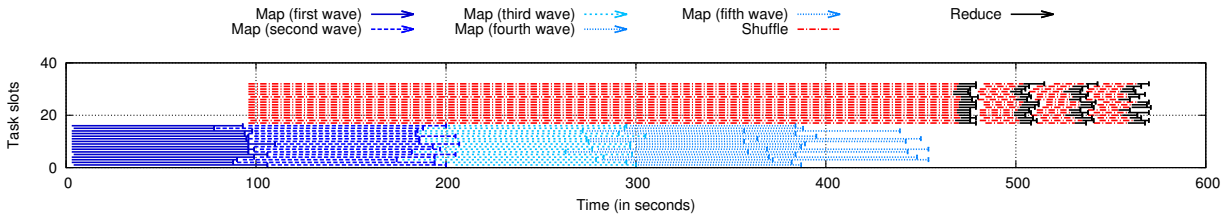



Figure 1. WikiTrends application executed in a Hadoop cluster with 16 map and 16 reduce slots.

## III. Optimized Batch Scheduling

In this section, we discuss the problem of minimizing the overall completion time for a given set of MapReduce jobs. We present a simple but effective *abstraction* of the MapReduce job execution that enables us to apply the classic Johnson algorithm for building an optimized job schedule. Then we discuss possible inefficiencies of this abstraction and a novel heuristic as an alternative solution for an optimized schedule of a given set of MapReduce jobs.

### A. Problem Definition

Each MapReduce job consists of a specified number of map and reduce tasks. The job execution time and specifics of the execution depend on the amount of resources (map and reduce slots) allocated to the job. Section II presents an example of Wikitrends application processing. Figure 1 shows a detailed visualization of how 71 map and 64 reduce tasks of this application are processed in the Hadoop cluster with 16 map and 16 reduce slots. Instead of the detailed job execution at the task level, we introduce a simple **abstraction**, where each MapReduce job $J_i$ is defined by durations of its map and reduce stages $m_i$ and $r_i$, i.e., $J_i = (m_i, r_i)$. Section II presents our profiling approach and performance model for computing the estimates of average map and reduce stage durations when the job is executed on a new dataset. This model is applied to derive the proposed new abstraction $J_i = (m_i, r_i)$.

Let us consider the execution of two (independent) MapReduce jobs $J_1$ and $J_2$ in a Hadoop cluster with a FIFO scheduler. There are no data dependencies between these jobs. Therefore, once the first job completes its map stage and begins reduce stage processing, the next job can start its map stage execution with the released map resources in a pipelined fashion (see Figure 2). There is an "overlap" in executions of map stage of the next job and the reduce stage of the previous one.

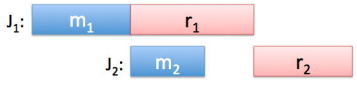
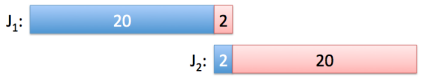
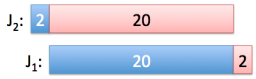



Figure 2. Pipelined execution of two MapReduce jobs $J_1$ and $J_2$.

We note an interesting observation about the execution of such jobs. Some of the execution orders may lead to a significantly less efficient resource usage and an increased processing time. As a motivating example, let us consider two independent MapReduce jobs that utilize all the given Hadoop cluster's resources and that result in the following map and reduce stage durations: $J_1 = (20s, 2s)$ and $J_2 = (2s, 20s)$. In the Hadoop cluster with the FIFO scheduler, they can be processed in two possible ways:




(a) $J_1$ is followed by $J_2$.




(b) $J_2$ is followed by $J_1$.

Figure 3. Impact of different job schedules on overall completion time.

- $J_1$ is followed by $J_2$ (as shown in Figure 3 (a)). The reduce stage of $J_1$ overlaps with the map stage of $J_2$ leading to overlap of only $2s$. Thus, the total completion time of processing two jobs is $20s + 2s + 20s = 42s$.
- $J_2$ is followed by $J_1$ (as shown in Figure 3 (b)). The reduce stage of $J_2$ overlaps with the map stage of $J_1$ leading to a much better pipelined execution and a larger overlap of 20s. Thus, the total makespan is $2s + 20s + 2s = 24s$.

Thus, there can be a significant difference in the overall job completion time (75% in the example above) depending on the execution order of the jobs.

We consider the *following problem*. Let $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$ be a set of $n$ MapReduce jobs with no data dependencies between them. We aim to determine an order (a schedule) of execution of jobs $J_i \in \mathcal{J}$ such that the makespan of the entire set is minimized.

### B. Johnson's Algorithm

In 1953, Johnson [2] proposed an optimal algorithm for two stage production schedule. In the original problem formulation, a set of production items and two machines ($S_1$ and $S_2$) are given. Each item must pass through stage one that is served by machine $S_1$, and then stage two that is served by machine $S_2$. Each machine can handle only one item at a time. The production item $i$ in the set is represented by two positive[3] numbers $(s_i^1, s_i^2)$ that define service times for the item to pass through stages one and two respectively.

There is a striking similarity between the problem formulation described above and the problem that we would like to solve: building a schedule that minimizes the makespan of a given set of MapReduce jobs. We can represent each MapReduce job $J_i$ in our batch set $\mathcal{J}$ by a pair of computed durations $(m_i, r_i)$ of its map and reduce stages, and these stage durations fairly define the "busy" processing times

[3]In fact, Johnson's schedule is also optimal for the case when $s_i^2 = 0$.

by the map and reduce slots respectively. This abstraction enable us to apply Johnson's algorithm (offered for building the optimal two-stage jobs' schedule) to our scheduling problem for a set of MapReduce jobs. Now, we explain the essence of Johnson's algorithm in terms of MapReduce jobs.

Let us consider a collection $\mathcal{J}$ of $n$ jobs, where each job $J_i$ is represented by the pair $(m_i, r_i)$ of map and reduce stage durations respectively. Let us augment each job $J_i = (m_i, r_i)$ with an attribute $D_i$ that is defined as follows:

$$D_i = \begin{cases} (m_i, \texttt{m}) & \text{if } \min(m_i, r_i) = m_i, \\ (r_i, \texttt{r}) & \text{otherwise.} \end{cases}$$

The first argument in $D_i$ is called the *stage duration* and denoted as $D_i^1$. The second argument is called the *stage type* (map or reduce) and denoted as $D_i^2$.

Algorithm 1 shows how an optimal schedule can be constructed using Johnson's algorithm. First, we sort all the $n$ jobs from the original set $\mathcal{J}$ in the ordered list $L$ in such a way that job $J_i$ precedes job $J_{i+1}$ if and only if $min(m_i, r_i) \leq min(m_{i+1}, r_{i+1})$. In other words, we sort the jobs using the stage duration attribute $D_i^1$ in $D_i$ (it represents the smallest duration of the two stages). Then the algorithm works by taking jobs from list $L$ and placing them into the schedule $\sigma$ from the both ends (head and tail) and proceeding towards the middle. If the stage type in $D_i$ is m, i.e., represents the map stage, then the job $J_i$ is placed from the head of the schedule, otherwise from the tail. The complexity of Johnson's Algorithm is dominated by the sorting operation and thus is $\mathcal{O}(n \log n)$.

**Algorithm 1** Johnson's Algorithm

**Input:** A set $\mathcal{J}$ of $n$ MapReduce jobs. $D_i$ is the attribute of job $J_i$ as defined above.
**Output:** Schedule $\sigma$ (order of jobs execution.)

1: Sort the original set $\mathcal{J}$ of jobs into the ordered list $L$ using their stage duration attribute $D_i^1$
2: $head \leftarrow 1, tail \leftarrow n$
3: **for each** job $J_i$ in $L$ **do**
4: **if** $D_i^2$ = m **then**
5: *// Put job $J_i$ from the front*
6: $\sigma_{head} \leftarrow J_i$, head $\leftarrow$ head + 1
7: **else**
8: *// Put job $J_i$ from the end*
9: $\sigma_{tail} \leftarrow J_i$, tail $\leftarrow$ tail - 1
10: **end if**
11: **end for**

Let us illustrate the job schedule construction with Johnson's algorithm for a simple example with five MapReduce jobs shown in Figure 4. These jobs are augmented with additional computed attribute $D_i$ shown in the last column.

At first, this collection of jobs is sorted into a list $L$ according to the attribute $D_i^1$ (i.e., first argument of $D_i$). The sorted list of jobs is shown in Figure 5. Then we follow Johnson's algorithm and start placing the jobs in the schedule $\sigma$ from both ends toward the middle, and construct the following schedule:

| $J_i$ | $m_i$ | $r_i$ | $D_i$ |
|---|---|---|---|
| $J_1$ | 4 | 5 | (4, m) |
| $J_2$ | 1 | 4 | (1, m) |
| $J_3$ | 30 | 4 | (4, r) |
| $J_4$ | 6 | 30 | (6, m) |
| $J_5$ | 2 | 3 | (2, m) |

Figure 4. Example of five MapReduce jobs.

| $J_i$ | $m_i$ | $r_i$ | $D_i$ |
|---|---|---|---|
| $J_2$ | 1 | 4 | (1, m) |
| $J_5$ | 2 | 3 | (2, m) |
| $J_1$ | 4 | 5 | (4, m) |
| $J_3$ | 30 | 4 | (4, r) |
| $J_4$ | 6 | 30 | (6, m) |

Figure 5. The ordered list $L$ of five MapReduce jobs.

- $J_2$ is represented by $D_2$=(1, m). Since $D_2^2$ = m then $J_2$ goes to the head of $\sigma$, and $\sigma = (J_2, ...)$.
- $J_5$ is represented by $D_5$=(1, m). Again, $J_5$ goes to the head of $\sigma$, and $\sigma = (J_2, J_5, ...)$.
- $J_1$ is represented by $D_1$=(4, m), and it goes to the head of $\sigma$, and $\sigma = (J_2, J_5, J_1, ...)$.
- $J_3$ is represented by $D_3$=(4, r). Since $D_3^2$ = r then $J_3$ goes to the tail of $\sigma$, and $\sigma = (J_2, J_5, J_1, ..., J_3)$.
- $J_4$ is represented by $D_4$=(1, m) and it goes to the head of $\sigma$, and $\sigma = (J_2, J_5, J_1, J_4, J_3)$.

Job ordering $\sigma = (J_2, J_5, J_1, J_4, J_3)$ defines Johnson's schedule for the job execution with the minimum overall makespan. For our example, the makespan of the optimal schedule is 47. The worst schedule is defined by the reverse order of the optimal one, i.e., $(J_3, J_4, J_1, J_5, J_2)$. The worst job schedule has a makespan of 78 (this is 66% increase in the makespan compared to the optimal time). Indeed, the optimal schedule may provide significant savings.

### C. BalancedPools Heuristic Algorithm

While the simple abstraction for MapReduce jobs proposed in Section III-B enables us to apply the elegant Johnson algorithm for constructing the optimized job schedule, it raises the following questions about its abstraction:

- How well does this abstraction correspond to the reality of complex execution of MapReduce jobs?
- How accurate is the *computed* makespan of Johnson's schedule for estimating the *measured* makespan of a given set of MapReduce jobs?
- What are the situations where the generated Johnson schedule might lead to suboptimal results?

When a MapReduce job is represented as a pair of map and reduce stage durations, it obscures the number of tasks that comprise the job's map and reduce stages and the number of slots that process these tasks. For example, Figure 1 shows how 71 map and 64 reduce tasks of Wikitrends application are processed in the Hadoop cluster with 16 map and 16 reduce slots. Note, that the last, fifth wave of the map stage has for processing only 7 tasks (71-16x4). Thus, out of 16 available map slots only 7 slots are used by the current application and the remaining 9 map slots can be immediately used for processing of the next job. Therefore, processing of the next job's map stage may start before the previous job completes its map stage. As a result, the makespan computed by the Johnson algorithm might be pessimistic compared to the real execution of the job schedule on the Hadoop cluster. To provide better estimates for the makespan of a given set of MapReduce jobs under

different job schedules, we use the MapReduce simulator SimMR [4] that can faithfully replay MapReduce job traces at the tasks/slots level: the completion times of simulated jobs are within 5% of the original ones.

Let us revisit MapReduce job processing and discuss situations where Johnson's schedule might provide a suboptimal solution. Consider the set of five jobs shown in Figure 4 (see Section III-B). Below we describe *two different scenarios* that, in spite of their differences, lead to the same job profiles and stage durations as shown in Figure 4. Therefore, if we apply Johnson's algorithm, it will produce the same schedule $\sigma = (J_2, J_5, J_1, J_4, J_3)$ for minimizing the makespan of this set. In both scenarios, we consider a Hadoop cluster with 30 worker nodes, each configured with a single map and single reduce slot, i.e., with 30 map and 30 reduce slots overall.

*Scenario1:* Let each job in the set be comprised of 30 map and 30 reduce tasks. Thus, each job utilizes either all map or all reduce slots during its processing. In this scenario, there is a perfect match between the assumptions of the classic Johnson algorithm for two-stage production system and MapReduce job processing.

*Scenario2:* Let jobs $J_1, J_2$, and $J_5$ be comprised of 30 map and 30 reduce tasks, and jobs $J_3$ and $J_4$ consist of 20 map and 20 reduce tasks. Figure 6(a) visualizes the execution of these five MapReduce jobs according to the generated Johnson schedule $\sigma = (J_2, J_5, J_1, J_4, J_3)$.

We use a different color scheme for map (blue/dark) and reduce (red/light) stages, the height of the stages reflects the amount of resources used by the jobs, the width represents the stage duration, the jobs appear at the time line as they are processed by the schedule.

While the first three jobs $J_2, J_5$, and $J_1$ utilize all map and all reduce slots during their processing, the last two jobs $J_4$ and $J_3$ only use 20 map and 20 reduce slots, and hence map stage processing of $J_3$ starts earlier than the map stage of $J_4$ is completed because there are 10 map slots available in the system. The first 10 tasks of $J_3$ are processed concurrently with 20 map tasks of $J_4$. When $J_4$ completes its map stage and releases 20 map slots, then the next 10 map tasks of $J_3$ get processed. However, this slightly modified execution leads to the same makespan of 47 time units as under *Scenario1* because processing of $J_3$'s reduce stage can only start when the entire map stage of $J_3$ is finished.

We claim that Johnson's schedule for *Scenario2* described above is suboptimal, by outlining a better solution. Let us partition these five jobs into two pools with a tailored amount of cluster resources allocated to each pool:

1) *Pool1* with $J_1, J_2$, and $J_5$ (10x10 map/reduce slots);
2) *Pool2* with $J_3$ and $J_4$ ( 20x20 map/reduce slots).

First of all, a different amount of resources allocated to jobs in *Pool1* changes these jobs' map and reduce stage durations. Each of these jobs has 30 map and 30 reduce tasks for processing. When these 30 tasks are processed with 10 slots, the execution takes three times longer: both map and reduce stages are processed in three waves, compared with

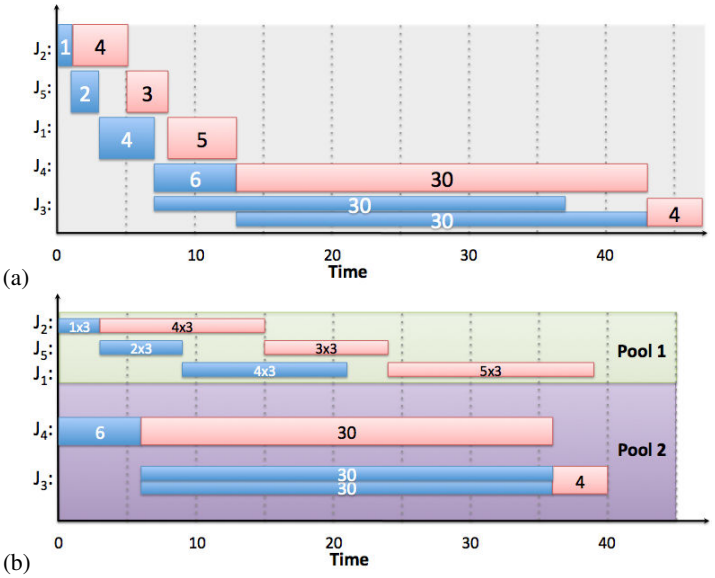



Figure 6. Example with five MapReduce jobs: (a) job processing with Johnson's schedule; (b) an alternative solution with BalancedPools.

a single wave for the stage execution with 30 slots. For jobs in each pool, we apply Johnson's algorithm to generate the optimized schedules:

1) *Pool1* is processed according to $\sigma_1 = (J_2, J_5, J_1)$. This schedule results in the makespan of 39 time units;
2) *Pool2* is executed according to $\sigma_2 = (J_4, J_3)$. This schedule results in the makespan of 40 time units.

Figure 6(b) visualizes the job execution of these two pools. Jobs in *Pool1* and *Pool2* are processed concurrently (each set follows its own schedule). The cluster resources are partitioned between the two pools in a tailored manner. Using this approach, the overall makespan for processing these five jobs is 40 time units, that is almost 20% improvements compared to 47 time units using Johnson's schedule. This example exploits additional properties specific to MapReduce environments and the execution of MapReduce jobs. In particular, the job stage durations closely depend on the amount of allocated resources (map and reduce slots). In this way, we can change the jobs' *appearance*. The main objective function of such an algorithm is to partition the jobs into two pools with specially tailored resource allocations such that the makespan of jobs in these pools are balanced, and the overall completion time of jobs in both pools is minimized. In general, the problem of balancing the map and reduce tasks in slots to achieve the minimum makespan for a set of MapReduce jobs is NP-hard. This can be easily proved by a simple polynomial reduction from the 3PARTITION problem [5]. We design a heuristic called the *BalancedPools* algorithm. As shown in Algorithm 2, we iteratively partition the jobs into two pools and then try to identify the adequate resource allocations for each pool such that the makespans of these pools are balanced. Within each pool we apply Johnson's algorithm for job scheduling, where map and reduce stage durations are computed with the performance model described in

**Algorithm 2** BalancedPools Algorithm

**Input:** 1) List $J$ of $n$ MapReduce jobs.
2) $M$: Number of machines in the cluster.
**Output:** Optimized Makespan

```
1: Sort J based on increasing number of map tasks
2: BestMakespan ←SIMULATE(J, JOHNSONORDER(J), M)
3: for split ← 1 to n − 1 do
4:    // Partition J into list of small Jobs_α and big Jobs_β
5:    Jobs_α ← (J_1, ··· , J_split)
6:    Jobs_β ← (J_split+1, ··· , J_n)
7:    SizeBegin ← 1, SizeEnd ← M
8:    // Binary search for the pool size that balances completion
      times of both pools
9:    repeat
10:      SizeMid ← (SizeBegin + SizeEnd)/2
11:      Makespan_α ← SIMULATE(Jobs_α,
         JOHNSONORDER(Jobs_α), SizeMid)
12:      Makespan_β ← SIMULATE(Jobs_β,
         JOHNSONORDER(Jobs_β), M - SizeMid)
13:      if Makespan_α < Makespan_β then
14:         SizeEnd ← SizeMid
15:      else
16:         SizeBegin ← SizeMid
17:      end if
18:   until SizeBegin ≠ SizeEnd
19:   Makespan ← MAX(Makespan_α, Makespan_β)
20:   if Makespan < BestMakespan then
21:      BestMakespan ← Makespan
22:   end if
23: end for
```

Section II. The pool makespan is estimated (accurately within 5%) with MapReduce simulator SimMR [4] as a part of the algorithm. The use of the simulator in the solution is absolutely necessary and justified. As we demonstrated, the makespan computation that follows Johnson's schedule and its simple abstraction may result in a significant inaccuracy, and more accurate estimates might be obtained only via MapReduce simulations at the task/slot level. The complexity of the algorithm is $\mathcal{O}(n^2 \log n \log M)$. However, SimMR can simulate a 1000 job workload on a 100 node Hadoop cluster in less than 2 seconds. The designed algorithm can be extended to a larger number of pools at the price of a significantly higher complexity.

The job execution with two pools is implemented using Capacity scheduler [6] that allows resource partitioning into different pools with a separate job queue for each pool.

## IV. EVALUATION

This section evaluates the benefits of Johnson's schedule and the novel *Balanced Pools* algorithm for minimizing the makespan of a set of MapReduce jobs using a variety of synthetic and realistic workloads derived from the Yahoo! M45 cluster. First, we evaluate the benefits of different schedules with simulation environment SimMR. Then, we validate the simulation results by performing similar experiments in a 66-node Hadoop cluster.

### A. Workloads

We use the following workloads in our experiments:

**1) Yahoo! M45:** This workload represents a mix of 100 MapReduce jobs [4] that is based on the analysis performed on the Yahoo! M45 cluster [7], and is generated as follows:

- Each job consists of the number of map and reduce tasks drawn from the distribution $\mathcal{N}(154, 558)$ and $\mathcal{N}(19, 145)$ respectively, where $\mathcal{N}(\mu, \sigma)$ is the normal distribution with mean $\mu$ and standard deviation $\sigma$.
- Map and reduce task durations are defined by $\mathcal{N}(50, 200)$ and $\mathcal{N}(100, 300)$ respectively[5].
- To avoid that map and reduce stage durations of the jobs look similar to each other (since they are drawn from the same distribution), an additional *scale factor* is applied to map and reduce task durations of each job.

To perform a sensitivity analysis, we have created two job sets (100 jobs each) based on Yahoo! M45 workload:

1) *Unimodal* set that uses a single scale factor for the overall workload, i.e., the scale factor for each job is drawn uniformly from $[1, 10]$.
2) *Bimodal* set where a subset of jobs (80%) are scaled using a factor uniformly distributed between $[1, 2]$ and the remaining jobs (20%) are scaled using $[8, 10]$. This mimics workloads that have a large fraction of *short* jobs and a smaller subset of *long* jobs.

**2) Synthetic:** Additionally, we create a synthetic workload with 100 jobs having a number of map and reduce tasks drawn uniformly from $[1, 100]$ and $[1, 50]$ respectively. The map and reduce task durations are normally distributed using $\mathcal{N}(100, 1000)$ and $\mathcal{N}(200, 2000)$ respectively. We create two versions of synthetic workload: 1) *Unimodal*, where each job is scaled using a factor uniformly distributed between $[1, 10]$ and 2) *Bimodal*, where 80% of the jobs are scaled using a factor uniformly distributed between $[1, 2]$ and the remaining 20% of jobs are scaled using $[8, 10]$.

### B. Simulation Results

First, we analyze the proposed job schedule algorithms and their performance using the simulation environment SimMR [4] that was designed for evaluation and analysis of different workload management strategies in MapReduce environments. SimMR can replay execution traces of real workloads collected in Hadoop clusters as well as generate and execute synthetic traces based on statistical properties of workloads. Simulating synthetic workloads is especially attractive since it enables a sensitivity analysis of scheduling policies for a variety of different MapReduce workloads.

Figure 7 shows the results for the synthetic workloads with *Unimodal* and *Bimodal* distributions. These graphs reflect five lines: *Min* and *Max* show theoretical makespans under Johnson's (optimal) schedule and reverse Johnson's

[4]We also run a mix with 10-20 jobs and obtained similar performance results.

[5]The study [7] did not report statistics of individual task durations. We use a greater range for reduce tasks since they combine shuffle, sort, reduce phase processing, and time for writing three data copies back to HDFS.

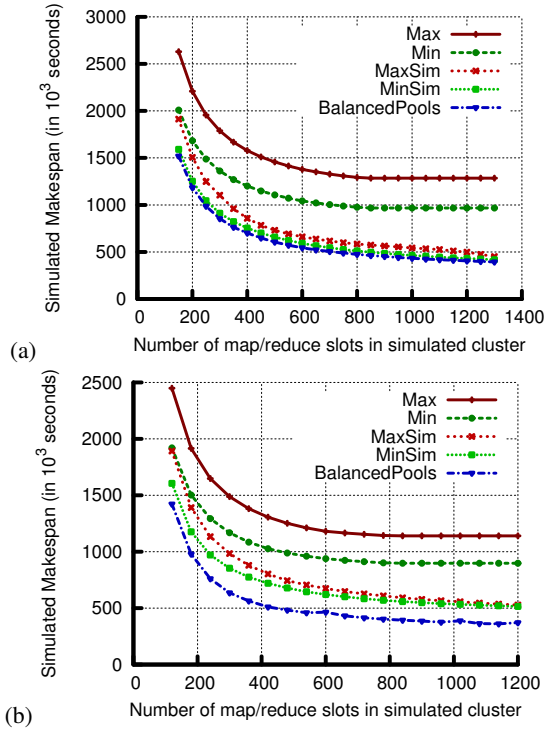



Figure 7. Simulating synthetic workload: (a) *Unimodal* and (b) *Bimodal*.

(worst) schedule respectively. That is, if MapReduce jobs would precisely satisfy the two-stage system assumptions then the overall makespan can be easily computed from the abstraction $J_i$=$(m_i, r_i)$. A difference between *Min* and *Max* reflects achievable performance benefits under the optimal schedule for this abstraction. *MinSim* and *MaxSim* show simulated makespans with SimMR for a set of given MapReduce jobs under Johnson's schedule and reverse Johnson's schedule respectively. We should stress that once we consider MapReduce jobs at the tasks/slots level Johnson's schedule and reverse Johnson's schedule do not guarantee the optimal and worst makespan for this set of jobs. The difference between *MinSim* and *MaxSim* reflects a lower bound of potential benefits (since the "worst" makespan might be much worse than under *MaxSim*). Finally, *BalancedPools* reflects the simulated (with SimMR) makespan of the job schedule constructed with the new *BalancedPools* heuristic.

The X axis reflects the Hadoop cluster size (without loss of generality, we assume 1 map and 1 reduce slot per node). The algorithm performance is a function of the cluster size: with its increase (i.e., when available resources in the cluster are plentiful), the performance benefits are diminishing as expected. However, for different workloads the points of diminishing return are different. This simulation exercise is useful for evaluating the required cluster size to support the specific (targeted) makespan for a set of given jobs.

Figure 7 shows that the simplified abstraction $J_i$=$(m_i, r_i)$ and makespan computations that use it (i.e., *Min* and *Max*) are inaccurate for estimating the real makespan of MapReduce jobs (due to lack of tasks/slots information), and in the rest of the graphs we omit these lines. This comparison strongly justifies the introduction of the simulator SimMR in the new heuristic for accurate makespan estimates.

Figure 7(a) shows up to 25% of makespan decrease with Jonhson's schedule (*MinSim*) compared to *MaxSim* for *Unimodal* case. The benefits diminishing for larger cluster sizes. The *BalancedPools* schedule behaves similar to Johnson's in this case. However, results are very different for the *Bimodal* workload shown in Figure 7(b). The *BalancedPools* heuristic provides up to 38% of makespan improvements, which are much better compared to Johnson's schedule (it is suboptimal for this workload). *BalancedPools* achieves signifficant additional makespan improvements compared to Johnson's algorithm for a variety of different cluster sizes.

Figure 8 shows results of simulating the Yahoo! M45 workload (*Unimodal* and *Bimodal* types). Interestingly, Johnson's schedule provides diminished returns in both cases for Yahoo!'s workload. We can see only up to 12% of

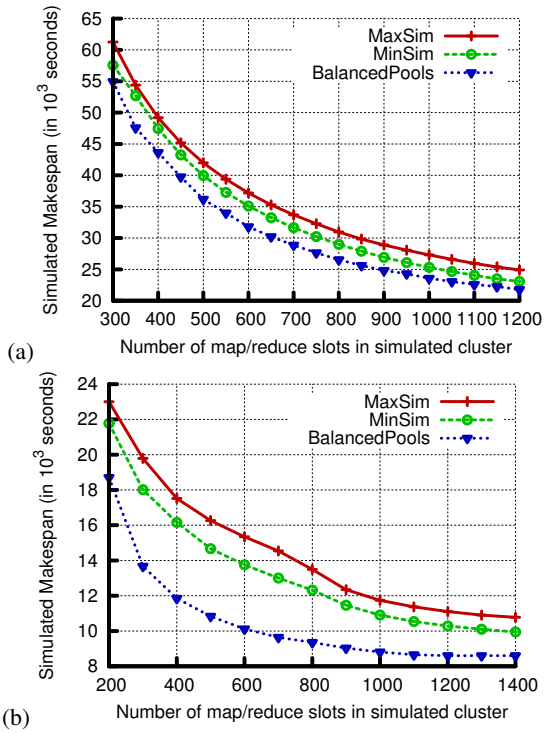



Figure 8. Simulating Yahoo!'s workload: (a) *Unimodal* and (b) *Bimodal*.

makespan improvements for most experiments. The *BalancedPools* heuristic significantly outperforms Johnson's algorithm: by 10%-30% in most cases. It shows overall makespan improvements up to 38% for the *Bimodal* Yahoo! M45 workload as shown in Figure 8(b).

Performance benefits under Johnson's algorithm and the *BalancedPools* heuristic are clearly workload and cluster size dependent. The proposed framework automatically constructs the optimized job schedule and provides the estimates of its makespan as a function of allocated resources. We validate the simulation results through experiments on a 66-node Hadoop cluster. These results closely follow the simulation results.

## V. Related Work

Scheduling of incoming jobs and the assignment of processors to the scheduled jobs has been an important factor for optimizing the performance of parallel and distributed systems. It has been studied extensively in scheduling theory (see a variety of papers and textbooks on the topic [8], [9], [10], [11], [12], [13]). Designing an efficient distributed server system often assumes choosing the "best" task assignment policy for a given model and user requirements. However, the question of "best" job scheduling or task assignment policy is still open for many models.

Job scheduling and workload management in MapReduce environments is a new topic, but it has already received much attention. Originally, Hadoop was designed for periodically running large batch workloads with a *FIFO* scheduler. As the number of users sharing the same MapReduce cluster increased, a new *Capacity* scheduler [6] and *Hadoop Fair Scheduler* (HFS) [14] were introduced to support more efficient cluster sharing. There are a few research prototypes of Hadoop schedulers that aim to optimize explicitly some given scheduling metric, e.g. FLEX [15], ARIA [3], etc. However, the existing schedulers do not provide a support for minimizing the completion time for a set of jobs.

*CoScan* [1] offers a special scheduling framework that merges the execution of Pig programs with common data inputs in such a way that this data is only scanned once. It augments Pig programs with a set of *(deadline, reward)* options to achieve. It then formulates the schedule as an optimization problem and offer a heuristic solution. While this approach aims to optimize the execution times of a set of Pig programs, their problem is quite different from the problem of minimizing the makespan of a set of independent jobs considered in our paper.

*Starfish* project [16] applies *dynamic instrumentation* to collect a detailed run-time monitoring information about job execution that enables the analysis and prediction of job execution under different configuration parameters. It offers a workflow-aware scheduler that correlate data (block) placement with task scheduling to optimize the workflow completion time. In our work, we propose complementary optimizations based on optimal scheduling of independent jobs to minimize the overall completion time.

Moseley et al. [17] is the closest work to ours. It formalizes MapReduce scheduling as a generalized version of the classical two-stage flexible flow-shop problem with identical machines. They provide a 12-approximate algorithm for the offline problem of minimizing the total *flowtime*, which is the sum of the time between the arrival and the completion of each job. In our work, we pursue a different performance objective and propose heuristics for minimizing the maximum completion time for a set of jobs.

## VI. Conclusion

In this work, we considered the problem of finding a schedule that minimizes the overall completion time of a given set of independent MapReduce jobs. We designed a novel framework and a new *heuristic*, called *BalancedPools*, that efficiently utilize characteristics and properties of MapReduce jobs in a given workload for constructing the optimized job schedule. Currently, we are evaluating this heuristic with a variety of different MapReduce workloads to measure achievable performance gains. Data analysis tasks are often specified with higher-level SQL-type abstractions like Pig and Hive, that may result in MapReduce jobs with dependencies. The next step is to address a more general problem of minimizing the makespan of batch workloads that additionally include workflows of MapReduce jobs.

## References


[1] X. Wang, C. Olston, A. Sarma, and R. Burns, "CoScan: Cooperative Scan Sharing in the Cloud," in *Proc. of SOCC*, 2011.

[2] S. Johnson, "Optimal Two- and Three-Stage Production Schedules with Setup Times Included," *Naval Res. Log. Quart.*, 1954.

[3] A. Verma, L. Cherkasova, and R. H. Campbell, "ARIA: Automatic Resource Inference and Allocation for MapReduce Environments," in *Proc. of ICAC,*, 2011.

[4] ——, "Play It Again, SimMR!" in *Proc. of Intl. IEEE Cluster'2011*.

[5] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*. WH Freeman & Co., 1979.

[6] "Capacity Scheduler Guide." [Online]. Available: http://hadoop.apache.org/common/docs/r0.20.1/capacity_scheduler.html

[7] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, "An Analysis of Traces from a Production MapReduce Cluster," in *Proc of CCGrid'2010*.

[8] J. Blazewicz, *Scheduling in computer and manufacturing systems*. Springer-Verlag, New York, USA, 1996.

[9] R. Blumofe and C. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 720–748, 1999.

[10] C. Chekuri and M. Bender, "An efficient approximation algorithm for minimizing makespan on uniformly related machines," *Integer Programming and Combinatorial Optimization*, pp. 383–393, 1998.

[11] B. Lampson, "A scheduling philosophy for multiprocessing systems," *Communications of the ACM*, vol. 11, no. 5, 1968.

[12] J. Leung, *Handbook of scheduling: algorithms, models, and performance analysis*. CRC Press, 2004.

[13] L. Tan and Z. Tari, "Dynamic task assignment in server farms: Better performance by task grouping," in *Proc. of the Intl. Symp. on Computers and Communications (ISCC)*, 2002.

[14] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. of EuroSys*. ACM, 2010, pp. 265–278.

[15] J. Wolf and et al, "FLEX: A Slot Allocation Scheduling Optimizer for MapReduce Workloads," *ACM/IFIP/USENIX Intl. Middleware Conference*, 2010.

[16] H. Herodotou and S. Babu, "Profiling, What-if Analysis, and Costbased Optimization of MapReduce Programs." in *Proc. of the VLDB Endowment, Vol. 4, No. 11*, 2011.

[17] B. Moseley, A. Dasgupta, R. Kumar, and T. Sarlós, "On scheduling in map-reduce and flow-shops," in *Proc. of SPAA*, 2011.