Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

Meeting deadlines for approximation processing in MapReduce environments*

Ming-hao HU†‡, Chang-jian WANG, Yu-xing PENG

(*National Laboratory for Parallel and Distributed Processing, School of Computer, National University of Defense Technology, Changsha 410073, China*)

†E-mail: minghao_hu@yeah.net

Received Mar. 13, 2016; Revision accepted June 24, 2016; Crosschecked Nov. 22, 2017

Abstract: To provide timely results for big data analytics, it is crucial to satisfy deadline requirements for MapReduce jobs in today's production environments. Much effort has been devoted to the problem of meeting deadlines, and typically there exist two kinds of solutions. The first is to allocate appropriate resources to complete the entire job before the specified time limit, where missed deadlines result because of tight deadline constraints or lack of resources; the second is to run a pre-constructed sample based on deadline constraints, which can satisfy the time requirement but fail to maximize the volumes of processed data. In this paper, we propose a deadline-oriented task scheduling approach, named 'Dart', to address the above problem. Given a specified deadline and restricted resources, Dart uses an iterative estimation method, which is based on both historical data and job running status to precisely estimate the real-time job completion time. Based on the estimated time, Dart uses an approach–revise algorithm to make dynamic scheduling decisions for meeting deadlines while maximizing the amount of processed data and mitigating stragglers. Dart also efficiently handles task failures and data skew, protecting its performance from being harmed. We have validated our approach using workloads from OpenCloud and Facebook on a cluster of 64 virtual machines. The results show that Dart can not only effectively meet the deadline but also process near-maximum volumes of data even with tight deadlines and limited resources.

Key words: MapReduce; Approximation jobs; Deadline; Task scheduling; Straggler mitigation
https://doi.org/10.1631/FITEE.1601056 **CLC number:** TP311

1 Introduction

MapReduce (Dean and Ghemawat, 2008), one of the most popular big data processing frameworks, along with its open-source implementation Hadoop (Apache, 2016), has been widely used to process big data analytics (Herodotou *et al.*, 2011). In the production environment, MapReduce jobs with deadline constraints, often seen in real-time SQL query, personalized advertisement recommendation, word frequency statistics, and so on, are crucial for business applications. However, the rapid growth of data volumes, along with the limitation of cluster capacities and the concurrency of multiple running jobs, has made it inevitable that deadline-bound jobs tend to operate on only a subset of their data in order to meet strict deadline constraints (Liu *et al.*, 1994; Venkataraman *et al.*, 2007; Agarwal *et al.*, 2013; OREILLY, 2013). Since these jobs are spawned on large datasets and the accuracy is proportional to the fraction of data processed, the natural goal of scheduling such jobs is not only to satisfy deadline requirements but also to process as much data as possible to improve the accuracy (Bell Laboratories, 2001; Lohr, 2009; Ananthanarayanan *et al.*, 2014).

Much research effort has been devoted to the

‡ Corresponding author

* Project supported by the National Key Research and Development Program of China (No. 2016YFB1000101)

ORCID: Ming-hao HU, http://orcid.org/0000-0003-2986-4139 problem of meeting deadlines (Kc and Anyanwu, 2010; Polo *et al.*, 2010; Verma *et al.*, 2011, 2012; Li *et al.*, 2014; Zacheilas and Kalogeraki, 2014; Wang *et al.*, 2015). Traditional resource schedulers (Polo *et al.*, 2010; Verma *et al.*, 2011) dynamically allocate an appropriate amount of resources to complete deadline-bound jobs before their deadlines. However, although these methods are capable of processing the entire dataset, missed deadlines could occur because of tight deadline constraints or lack of resources. Other methods such as sampling-based approximate query processing (AQP) systems (Acharya *et al.*, 1999; Agarwal *et al.*, 2013) execute an appropriately sized sample selected from pre-constructed sample sets to meet deadline constraints. However, these methods may process a submaximal-sized sample because errors are brought in when constructing the sample set.

We have proposed a task scheduling approach in our previous work (Hu *et al.*, 2015), named 'Dart', to overcome the shortcomings of the current solutions. Given a specified deadline and restricted resources, Dart uses an iterative estimation method based on both historical data and job running status to precisely calculate the real-time job completion time. Then Dart continues to launch map tasks so that the estimated job completion time would approach but does not exceed the deadline. Once the estimated time exceeds the deadline, Dart terminates the map phase and finishes the remaining part of the job. We call the above procedure the approach stage.

Although Dart can efficiently schedule tasks to meet deadlines and maximize the input size, there are still several factors that can affect the performance. First, since a MapReduce job is completed only when its last task finishes, stragglers can significantly prolong the job completion time, thus leading to missed deadlines. One of the most classical techniques to mitigate stragglers is speculation (Zaharia *et al.*, 2008). Speculative execution means launching duplicates of stragglers and picking the earliest finished one. However, simply launching new duplicates may still delay job completion. Therefore, accurate estimation is required before implementing speculation. Second, task failures in MapReduce can lead to a decrease in the processed data volume, while simply restarting the failed task can delay job completion. Third, uneven distribution of intermediate data to partitions can cause data skew, which would increase the duration of the reduce phase, thus prolonging job completion as well.

To mitigate the effect of stragglers, we add a new stage into Dart, the revise stage. This stage begins right after the approach stage ends. In this stage, Dart makes speculative decisions based on a speculative window when the map task finishes. Dart launches duplicates of stragglers only when this window is large enough for the execution of speculative copies; otherwise, it kills the slowest straggler. Besides, Dart optimizes the input size of reduce tasks to mitigate the reduce straggler. As for task failures, Dart restarts the failed task only when the estimated job completion time based on the restarted task is less than the deadline. Finally, to efficiently tackle data skew, Dart approximately calculates the job completion time as the worst-case value while considering data skew by doing statistical analysis on historical data.

We evaluate the performance of Dart in terms of its ability to meet deadlines and maximize volumes of processed data. We use state-of-the-art deadline guarantee approaches as baselines and conduct comprehensive experiments. The results demonstrate that compared to traditional methods, Dart can not only effectively meet the deadline but also process near-maximum volumes of data even when the deadline is set to be extremely small and limited resources are allocated, and it can efficiently handle a variety of real system challenges such as stragglers, task failures, and data skew.

2 Dilemma in meeting deadlines

To motivate our work for guaranteeing deadlines in MapReduce environments, we first describe the main challenges that current methods are facing when dealing with the deadline constraints. We then show that missed deadlines can easily occur due to the presence of stragglers and quantify its potential benefits if stragglers can be effectively mitigated.

2.1 Inability to meet deadlines

Given a deadline-bound job, traditional resource management techniques dynamically allocate appropriate resources to complete the job while achieving the deadline goal (Polo *et al.*, 2010; Verma *et al.*, 2011). However, deadlines can be missed in two normal situations. First, for each deadline-bound job, there exists a minimum deadline threshold (denoted as MDT), which is equal to the job completion time with maximum resource allocations (denoted as MRA). Obviously, the user-defined deadline given in current solutions should be larger than MDT; otherwise, the job would miss its deadline even with maximum resources. Second, since cluster capacities are limited and multiple submitted jobs may run concurrently, the resources allocated to the new deadline-bound job may be insufficient for finishing all its tasks before the time limit, thus failing to meet the deadline requirement.

To evaluate the problem in meeting deadlines, we build a simulator based on Verma *et al.* (2011)'s work. Given a MapReduce job with detailed execution information, a specified deadline, and fixed amount of resources, the simulator estimates the number of slots required for the job completion within the deadline and tries to allocate the appropriate number of slots within the resource budget. The jobs used are extracted from a workload trace at Carnegie Mellon University (CMU)'s research cluster (Ren *et al.*, 2013), and each job is characterized by the start and finish times of its tasks.

Fig. 1a shows the impact of tight deadlines on the percentage of deadlines missed with different amount of resources. We see that the probability of missed deadlines significantly increases as the given deadline decreases. Even with maximum resource allocations, the probability sharply increases to nearly 100% as the deadline drops below the minimum deadline threshold. Fig. 1b illustrates the problem of the lack of resources by plotting the percentage of deadlines missed for varying values of resources. We can see that even with loose deadlines (1.2MDT), the probability increases to nearly 95% when the amount of available resources decreases to approximately 70% of the maximum resource allocation. Besides, we can see that no matter how much resource is allocated, almost all jobs have missed their deadlines when the deadline is set to be less than the minimum deadline threshold (0.9MDT).

In summary, the above results show that tight deadlines and lack of resources are two main reasons for missing the deadlines when using current approaches to tackle the problem.

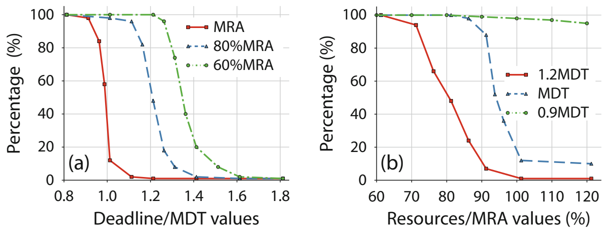


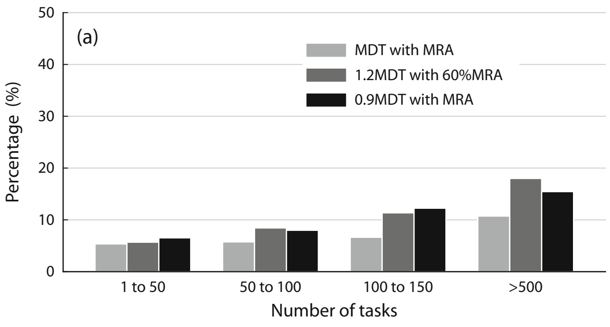
Fig. 1 Percentage of deadlines missed with changing deadlines and resource allocations: (a) percentage of deadlines missed under tight deadlines; (b) percentage of deadlines missed under limited resources

2.2 Inability to process maximum volumes of data

To meet the given deadline despite tight deadlines and lack of resources, there is a trend that MapReduce jobs operate only on a subset of their data by processing a random sample from shuffled input (Liu *et al.*, 1994; Venkataraman *et al.*, 2007; OREILLY, 2013), instead of using all the datasets. For these deadline-bound approximation jobs, the goal is to maximize the volumes of data processed (or tasks completed) so as to improve the accuracy of approximate results within the specified time limit.

Modern AQP systems such as BlinkDB (Agarwal *et al.*, 2013) and AQUA (Acharya *et al.*, 1999) choose an appropriately sized sample from pre-constructed sample sets to meet the performance goals. These methods simply assume that the job completion time scales linearly with the input size and then run a few subsamples to obtain an approximately linear relation between them so as to estimate the maximum size of the sample based on the deadline constraints.

We evaluate the performance of such pre-sampling methods in the CMU trace under three configurations (normal, lack of resources, and tight deadlines). We first quantify the ability of meeting deadlines. Fig. 2a shows that nearly 90% of jobs with less than 500 tasks can successfully meet their deadlines, but the performance degrades when the job consists of a large number of tasks. Next, we measure how many data volumes the pre-sampling method can process. From Fig. 2b, we can see that the variation between the real amount of data and the maximum amount of data increases from 10% for small jobs (1–50 tasks) to nearly 40% for large jobs (>500 tasks). This means that although the pre-sampling method can provide soft deadline guarantees for approximation jobs, it fails in maximizing the amount of processed data.



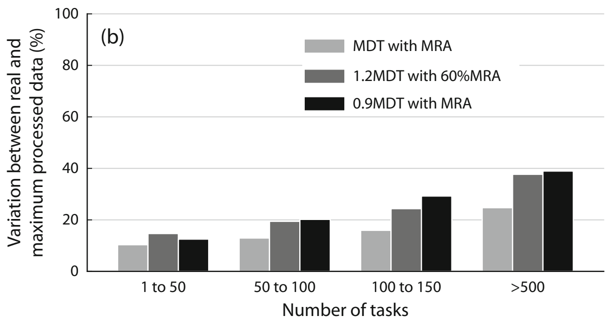


Fig. 2 Performance evaluation of the pre-sampling method: (a) percentage of deadlines missed under different configurations; (b) performance degradation in processing data volumes

There are two main causes leading to a submaximum-sized sample. First, assume a linear relation between the input size and job completion time is too simple to support precise estimations because the job completion time depends not only on the input size but also on the computation complexity of the job, the physical distribution of its dataset, and available resources for execution. Second, since many variables such as available resources and network bandwidth keep changing at runtime, it is rather inaccurate to use the statistics that are generated by the subsamples statically run before the job's execution to estimate the maximum size of the sample.

2.3 Straggler problem and its impact

Stragglers can significantly delay job completion, thereby causing missed deadlines. In the production clusters at Facebook and Microsoft Bing, even after applying the state-of-the-art straggler mitigation techniques such as LATE (Zaharia *et al.*, 2008) or Mantri (Ananthanarayanan *et al.*, 2010), latency-sensitive jobs have stragglers that are on average eight times slower than the median task in that job. These stragglers can significantly prolong the job completion time by 47% (Ananthanarayanan *et al.*, 2013).

Many studies have been devoted to the straggler problem, and one of the most classical techniques is speculation (Zaharia *et al.*, 2008). For straggler tasks, this technique is designed to spawn multiple copies on fast nodes and pick the earliest finished one while killing the rest. A task is classified as a straggler if its progress rate, which is calculated as its progress score divided by its running time, is less than half of the median progress rate among the tasks in its phase.

Although speculative execution can successfully handle stragglers, this method needs to wait for gathering enough execution information of tasks to detect stragglers and then launch speculative copies. Besides, these speculative copies would compete for resources with the unscheduled tasks, which may lead to processing a submaximum amount of completed tasks. In fact, there exists a mathematical analysis demonstrating that it is optimal to speculate conservatively to maximize the amount of processed tasks during the early waves of a job and speculate aggressively to fully use the allocated resources during the final wave of a job (Ananthanarayanan *et al.*, 2014).

For deadline-bound jobs, effectively mitigating stragglers means that more tasks can be completed within the time limit, thus improving the accuracy of results. To highlight this impact, we measure the potential improvement in the CMU trace by replacing the progress rate of straggler tasks with the median progress rate of tasks in the same phase. The result shows that the amount of completed tasks of deadline-bound jobs can increase by nearly 36%.

3 Iterative estimation of job completion time

To tackle the dilemma described above, the amount of completed tasks should be maximized within the time limit and redundant tasks should be terminated. Such a termination mechanism means that the time at which extra tasks should be terminated must be carefully confirmed to meet the approximation goal. Thus, we first present a novel method to iteratively estimate the real-time job completion time, which can later support our core design of Dart. Table 1 contains the notations used in the remainder of this section.

Table 1 Notations used in Section 3

Notation	Description
T_J	Real-time job completion time
M	Number of map tasks
R	Number of reduce tasks
T_M	Map phase duration
T_NS	Nonoverlapping shuffle phase duration
T_R	Reduced phase duration
k	Number of scheduled map tasks
T_p	Past time
T_m^a	Average duration of map tasks
T_S	Overall shuffle phase duration
T_OS	Overlapped shuffle phase duration
$B_\mathrm{J}^\mathrm{avg}$	Average transferring speed at the shuffle phase
p	Average partition size of map tasks
T_q	Start time of the shuffle phase
V_p	Current transferred data volume
d	Average input size of reduce tasks
$f(d)$	Execution time of reduce tasks with d

Given a MapReduce job J, which consists of M map tasks and R reduce tasks, we divide the job into three phases: map, shuffle, and reduce. Obviously, to estimate the real-time job completion time, we should first know the duration of each phase. Fig. 3 shows a typical MapReduce job which has six map tasks and three reduce tasks. For clarity, we use a single bar to represent the shuffle phase and abstract the transmission of intermediate data. Since the shuffle phase begins right after the first map task is completed, it partially overlaps with the map phase. Besides, each reduce task begins only after it has received all partitions shuffled from all map tasks.

We define T_J as the real-time job completion time, T_M as the map phase duration, T_NS as the nonoverlapping shuffle phase duration, and T_R as the reduce phase duration. Obviously, T_J can be calculated by the following equation:

$$T_\mathrm{J} = T_\mathrm{M} + T_\mathrm{NS} + T_\mathrm{R}. \tag{1}$$

As long as we figure out T_M, T_NS, and T_R, we can calculate T_J. For simplicity, we first assume that the job has a uniform data distribution.

3.1 Map phase duration

To dynamically estimate the real-time duration of the map phase, we define k as the number of scheduled map tasks of J, which is proportional to the volumes of processed data, and assume that J has already been executed for a period T_p and k map tasks have been scheduled. At this moment, the scheduler is ready to launch the next map task on workers.

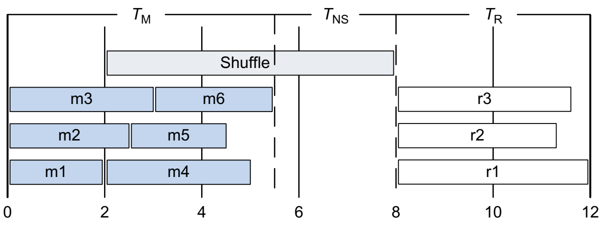


Fig. 3 A typical MapReduce job consisting of map, shuffle, and reduce phases, running 12 time units

It has been observed, across a variety of jobs, that the map task durations are generally stable (Chen *et al.*, 2011). Therefore, we define T_m^a as the average duration of map tasks. Due to data locality (Zaharia *et al.*, 2010), we further divide T_m^a into an average duration of local map tasks (denoted as T_l^a) and an average duration of remote map tasks (denoted as T_r^a).

We iteratively update a new T_M (denoted as T_M^n) every time before scheduling the next map task to represent the estimated map phase duration while considering the next map task launched. We also maintain an old T_M (denoted as T_M^o) to represent the last updated map phase duration. T_M^n is used to estimate the newest T_J. One potential T_M^n is $T_p + T_\mathrm{m}^\mathrm{a}$, which means that the next map task is estimated to be the last completed one among current scheduled map tasks. The other situation is that T_M^n is equal to T_M^o, which may occur when an already running task is a remote map task while the next map task is node-local. Therefore, T_M^n is calculated as follows:

$$T_\mathrm{M}^\mathrm{n} = \max(T_\mathrm{M}^\mathrm{o}, T_p + T_\mathrm{m}^\mathrm{a}). \tag{2}$$

For example, Fig. 4 shows that, when T_p is three time units, $T_\mathrm{m}^\mathrm{a} = 2.5$ and $k = 5$. At this moment, m3 is completed and the scheduler is ready to launch the remote map task m6. Therefore, T_M^n is calculated as $\max(5, 3 + 2.5) = 5.5$.

3.2 Nonoverlapping shuffle phase duration

For a given job J with known k and T_p, the nonoverlapping shuffle phase duration T_NS can be calculated as

$$T_\mathrm{NS} = T_\mathrm{S} - T_\mathrm{OS}. \tag{3}$$

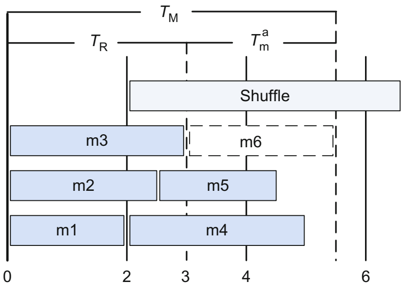


Fig. 4 Estimating the map phase duration at runtime

To calculate the overall shuffle phase duration T_S, we use the division of total transferred data volume and the average transferring speed B_J^{avg} as an approximate value. The reason for abstracting B_J^{avg} is that, in the shuffle phase, there usually exist three kinds of bottlenecks that limit the shuffle performance (Chowdhury *et al.*, 2011, 2014), namely, the bottleneck in the sender, the bottleneck in the receiver, and the bottleneck in the network. These bottlenecks lead to one simple characteristic: at least one link—whether it is the sender's link to the network, the contended link inside the network, or the network's link to the receiver—is fully used during the shuffle phase. This characteristic is important because the transferring speed of the shuffle phase would reach its upper bound and remain stable if any link is fully used. Therefore, for a given J, we can define its abstract average transferring speed of the shuffle phase B_J^{avg}.

To calculate the total transferred data volume, let p denote the average partition size of map task outputs. Since the number of reduce tasks, R, is fixed, the total transferred data volume while considering the next map task launched is calculated as $(k+1)\times p\times R$. Therefore, T_S is calculated as

$$T_S=\frac{(k+1)\cdot p\cdot R}{B_J^{avg}}. \tag{4}$$

To obtain B_J^{avg}, we define T_q as the start time of the shuffle phase. During the shuffle phase, the scheduler keeps recording the current volume that has been transferred at T_p (denoted as V_p) and then calculates B_J^{avg} as the division of the current transferred volume and the current transferred time:

$$B_J^{avg}=\frac{V_p}{T_p-T_q}. \tag{5}$$

Fig. 5 shows that when T_p is three time units, the total transferred data volume is the output of all six map tasks (considering the unscheduled m6). From the figure, we can see that T_{OS} can be easily calculated as

$$T_{OS}=T_M^n-T_q. \tag{6}$$

Finally, T_{NS} can be calculated as follows:

$$\begin{aligned}T_{NS}&=\frac{(k+1)\cdot p\cdot R}{B_J^{avg}}-(T_M^n-T_q)\\&=\frac{(k+1)\cdot p\cdot R\cdot(T_p-T_q)}{V_p}-(T_M^n-T_q).\end{aligned} \tag{7}$$

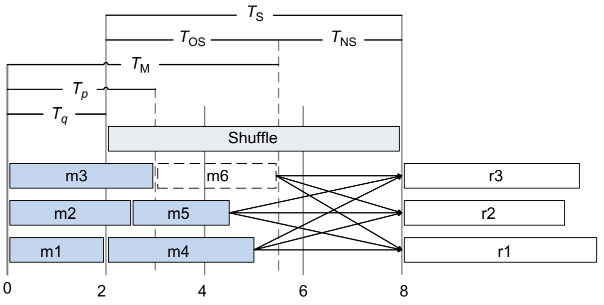


Fig. 5 Estimating the nonoverlapping shuffle phase duration at runtime

3.3 Reduce phase duration

In a MapReduce job, the reduce task starts only after it has received the volumes shuffled from all map tasks. Therefore, for the last reduce task, it starts only after the whole shuffle phase ends. We define that, for a given job J with k, T_R is the execution time of the last reduce task, which is estimated to be the average execution time of reduce tasks.

Based on the analysis of previous work, a task's execution time depends on the input size, its computation complexity, and the available resources (Ananthanarayanan *et al.*, 2010). The input size is decided by the changing k at runtime. The computation complexity is unknown to the scheduler, but it is the same for all reduce tasks. Besides, we assume that the resources allocated to each reduce task are the same, which is reasonable in a production environment.

To obtain the estimated T_R, we want to first figure out the relationship between k and the input size. Here, we first define d as the average input size of reduce tasks. Because each map task transfers one partition to one reduce task, it is easy to calculate d

as follows:

$$d = (k+1) \cdot p. \tag{8}$$

We then estimate $f(d)$, a variable denoting the average execution time of reduce tasks with given d, by running a series of sample reduce tasks with discrete $d \in [0, Mp]$. However, multiple sample tasks may take a long time and add a significant delay if they are running during the job's execution. Therefore, we estimate $f(d)$ only offline, precomputing the distribution of $f(d)$ to accurately and quickly estimate T_{R} (Ferguson *et al.*, 2012). Because the computation complexity and the allocated resources of sample tasks are the same as their counterparts of real reduce tasks, these sample results are close to real execution times.

Since $f(d)$ changes with d nonlinearly, giving a d to estimate $f(d)$ can be modeled as a nonlinear regression problem (Bates and Watts, 1988). We use a power function $f(d) = a \times d^b \times e$ to represent the nonlinear equation, where a and b are the unknown variables waiting to be estimated, and e is the error which has a mean of 1 and is always greater than 0. Taking the logarithm of both sides of the equation, we can obtain $\ln[f(d)] = \ln a + b \times \ln d + \ln e$. Note that our model is then transformed into a linear regression problem where $Y = \ln[f(d)]$ and $X = \ln d$. Therefore, we apply the least-squares method to figure out $\ln a$, b, and $\ln e$ (Marquardt, 1963). Since we assume that the mean of e is 1, the expected value of $\ln e$ is 0. After obtaining the above variables, we can do the curve fitting of $f(d)$ (Motulsky and Ransnas, 1987) so that given the specified k, T_{R} is calculated as follows:

$$T_{\mathrm{R}} = a \cdot ((k+1) \cdot p)^b \cdot e. \tag{9}$$

4 Approach–revise scheduling algorithm

By using the above iterative method, we can obtain an accurate estimated job completion time. Based on this method, we present our task scheduling algorithm, called the approach–revise algorithm, to solve the dilemma described in Section 2. For deadline-bound jobs, this algorithm is designed to maximize the number of scheduled map tasks within the specified deadline.

The algorithm consists of two stages: approach and revise. In the approach stage, the scheduler continuously launches map tasks to let the real-time job completion time approach the specified deadline. The scheduler does not terminate redundant map tasks until the job completion time exceeds the deadline.

After terminating redundant map tasks, the job completion time would be theoretically less than the deadline. However, stragglers can significantly prolong the job completion time, thereby leading to a missed deadline. So, a revise stage is designed to tackle the straggler problem and make sure that the real job completion time is less than the deadline. Since both map and reduce tasks could become stragglers, these two cases must be dealt with in the revise stage.

4.1 Approaching optimal k

To solve the dilemma described in Section 2, first, assume that the deadline D is set to be small enough or limited resources are allocated to job J so that only part of all map tasks can be processed. We continue to use the notation defined in Section 3: k represents the number of scheduled map tasks during runtime, and $T_{\mathrm{J}}(k)$ represents the real-time job completion time, which is a function of k. With the above statement, our objective is then to select k to maximize $T_{\mathrm{J}}(k)$ while $T_{\mathrm{J}}(k)$ must be less than D:

$$\begin{aligned} &\arg\max_k T_{\mathrm{J}}(k) \\ &\text{s.t. } T_{\mathrm{J}}(k) < D. \end{aligned} \tag{10}$$

To achieve the objective, Dart iteratively approaches the optimal k by using a greedy scheduling strategy. During runtime, the next map task is launched only if the estimated job completion time is less than the deadline; otherwise, the task is not launched. We use C to represent the set of assigned containers, c to represent the specified container, and m to represent the specified map task. The container is the abstract notion of resources such as memory and CPU in Hadoop Yarn (Vavilapalli *et al.*, 2013), which is similar to the notion of slot in Hadoop 1.0 but more fine-grained.

Algorithm 1 shows the details of the iterative approach scheduling procedure. Once the scheduler receives available containers, it tries to allocate specified map task m on every available container c. Before launching m on c, the scheduler calculates the real-time job completion time based on

m, which is denoted as $T_{\mathrm{J}}(k_m)$ by using the iterative estimation method. m can be launched on c and $k+=1$ only when $T_{\mathrm{J}}(k_m)$ is less than D. Otherwise, different strategies should be implemented based on the data locality property of m.

If m is a local map task, then for every map task n that is scheduled after m, $T_{\mathrm{J}}(k_n)$ is definitely larger than D. So, to avoid missing the deadline, the rest of all unscheduled map tasks must be terminated (including m), and the container c of m is released. The terminating mechanism, which can terminate the map phase when some user-defined conditions are satisfied, is part of our previous work (Wang *et al.*, 2014), and we integrate it into Dart.

If m is a remote map task, then for a local map task n that is scheduled after m, $T_{\mathrm{J}}(k_n)$ may be less than D because $T_{\mathrm{l}}^{\mathrm{a}}$ is less than $T_{\mathrm{r}}^{\mathrm{a}}$. Therefore, although m cannot be launched, n can be launched. So, the map phase cannot be simply terminated. Instead, the scheduler releases the assigned container and sets the task m as a new status: local-only. When m is local-only, the above scheduling procedure can proceed only when the assigned container is node-local.

Algorithm 2 shows the logic of workers. Once the event of launching a task is received, the worker launches the task with the specified container. During the execution of tasks, the worker reports the status of running tasks to the scheduler by sending heartbeats periodically. When a map task is completed, the worker releases the container and sends the task-completed event to the scheduler.

Algorithm 1 Iterative approaching scheduling

```
Require: $J$, $C$, $D$, $k$
Ensure: $k$, MapPhaseTerminate
 1: if receiveResourceEvent($C$) then
 2:   for each available container $c \in C$ do
 3:     if $J$ has specified map task $m$ on $c$ then
 4:       $T_{\mathrm{J}}$ = getEstimatedJobCompletionTime($k$, $m$, $c$)
 5:       if $T_{\mathrm{J}} < D$ then
 6:         sendLaunchTaskEvent($m$, $c$)
 7:         $k+=1$
 8:       else if isLocalMapTask($m$) then
 9:         sendReleaseContainerEvent($c$)
10:         MapPhaseTerminate = True
11:       else if isRemoteMapTask($m$) then
12:         sendReleaseContainerEvent($c$)
13:         Status$_m$ = Local-only
14:       end if
15:     end if
16:   end for
17: end if
18: return $k$, MapPhaseTerminate
```

Algorithm 2 Scheduling logic of the worker

```
Require: $m$, $c$
 1: if receiveLaunchTaskEvent($m$, $c$) then
 2:   launchTask($m$, $c$)
 3: end if
 4: if isCompleted($m$, $c$) then
 5:   sendTaskCompletedEvent($m$)
 6:   realeaseContainer($c$)
 7: end if
 8: if receiveReleaseContainerEvent($c$) then
 9:   releaseContainer($c$)
10: end if
```

4.2 Revising k and straggler mitigation

After the map phase is terminated, the scheduler no longer launches new map tasks. However, since there may exist stragglers among both map and reduce tasks, the job completion time can be significantly prolonged, leading to missed deadlines. Therefore, the revise stage is designed to both mitigate map and reduce stragglers.

4.2.1 Mapping straggler mitigation

First, we use a simple example to illustrate the map straggler scenario. Fig. 6 shows a MapReduce job J whose D is 10 time units. When T_p is 3, map task m6 has been terminated because $T_{\mathrm{J}}(k_6)$ is larger than D and m6 is a local task. However, m4 happens to be a straggler, and it can delay the completion of the entire job, causing a missed deadline.

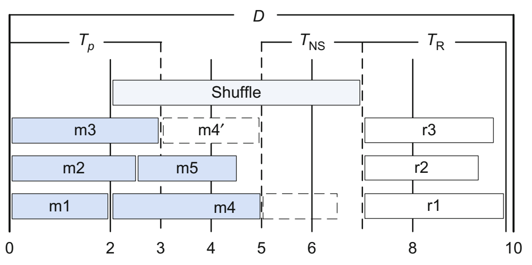


Fig. 6 Example illustrating our main ideas of straggler mitigation

Dart combines two methods to deal with map stragglers. One is to simply kill the slow task, which can mitigate the effect of stragglers, but the value of k will decrease. Another method is to speculatively schedule a duplicate copy on other nodes, which does not increase T_{NS} or T_{R}, and the value of k can remain unchanged. Taking Fig. 6 as an example, if m4 were to be killed, then k would decrease to 4, but T_{J} could, again, be less than D; if a copy of m4, saying m4', were to be launched on another container, then k would be 5 and T_{J} could still be less than D.

Dart uses a speculative window (denoted as W) to decide whether to kill map stragglers or launch speculative copies, which is calculated as follows:

$$W = D - T_p - T_{\mathrm{NS}} - T_{\mathrm{R}}. \tag{11}$$

This window illustrates the available time left for executing a new speculative map task. Speculative copies of stragglers can be launched only if the window time is larger than the average duration of map tasks; otherwise, stragglers are directly killed to avoid missed deadlines. We use S to represent the set of map stragglers s that is detected periodically during runtime to represent the specified straggler, and T_s^{rem} to represent the remaining execution time of s. Algorithm 3 shows the details of the revising algorithm.

Algorithm 3 The revising algorithm

```
Require: $k$, $C$, $S$, MapPhaseTerminate
Ensure: $k$, $S$
 1: if MapPhaseTerminate = True then
 2:   if receiveResourceEvent($C$) then
 3:     for each available container $c \in C$ do
 4:       if $S \neq$ null then
 5:         $W$ = getSpeculativeWindow()
 6:         $s$ = findMaxRemainingTime($S$)
 7:         if $W \geq T_{\mathrm{m}}^{\mathrm{a}}$ then
 8:           sendSpeculationEvent($s$, $c$)
 9:           $S = S - s$
10:         else if $0 < W < T_{\mathrm{m}}^{\mathrm{a}}$ then
11:           sendKillTaskEvent($s$)
12:           $k -= 1$
13:           $S = S - s$
14:         else
15:           $k$, $S$ = killExtraMapStragglers($k$, $S$)
16:         end if
17:       end if
18:     end for
19:   end if
20: end if
21: return $k$, $S$
```

Note that the window time could be less than 0, which means that currently the estimated T_{J} already exceeds D. We use a heuristic way to deal with this situation, that is, to kill the straggler whose remaining execution time T_s^{rem} is larger than a StragglingThreshold (Algorithm 4).

Algorithm 4 Killing extra map stragglers

```
Require: $k$, $S$
Ensure: $k$, $S$
1: for each straggler $s \in S$ do
2:   $T_s^{\mathrm{rem}}$ = getRemainingTime($s$)
3:   if $T_s^{\mathrm{rem}}$ > StragglingThreshold then
4:     sendKillTaskEvent($s$)
5:     $k -= 1$
6:     $S = S - s$
7:   end if
8: end for
9: return $k$, $S$
```

When the threshold is set to be 0, all stragglers will be killed, which can completely satisfy meeting deadlines but hurt the accuracy of results. On the contrary, the threshold can be set to be extremely large so that the volume of processed data will be maximized, but the job may fail in meeting the deadline. In practice, we have found that a good choice of the StragglingThreshold is the 80th percentile of the average duration of map tasks. The reason is that some stragglers may be launched at an early time and nearly be finished in the revise stage. We wish to kill as few stragglers as possible without exceeding the deadline. Our experiment shows that this setting is a great tradeoff when considering the abilities to both meet deadlines and maximize data volumes, which is shown in Section 6.5.

4.2.2 Reducing straggler mitigation

Not only map tasks but also reduce tasks can be stragglers for two reasons. First, although we assume that all reduce tasks start at the same time, in practice, we find that this assumption fails because of the dynamically changing network bandwidth of nodes. Second, skewed data can result in the variation of the execution time of reduce tasks.

To mitigate reduce stragglers, Dart iteratively optimizes the input size of every reduce task before launching it. For the reduce task r_i that is ready to be launched, the maximum remaining time of its execution is equal to $D - T_p$. Therefore, its maximum available input size (denoted as $d_i^{\max}$) can be calculated by the following equation:

$$f(d_i^{\max}) = D - T_p. \tag{12}$$

Recall that $f(d)$ is a nonlinear function denoting the execution time of reduce tasks with d. Once $d_i^{\max}$ is known, the input size d_i can be compared with it. If $d_i > d_i^{\max}$, then we apply a simple, uniform sampling on d_i to reduce the size to $d_i^{\max}$. Algorithm 5 shows the details of this scheme.

Algorithm 5 Reducing straggler mitigation

```
Require: $r_i$, $d_i$, $D$, $T_p$
Ensure: $d_i$
1: if isReadyForLaunch($r_i$) then
2:     $d_i^{\max}$ = getMaximalAvailableSize($D$, $T_p$)
3:     if $d_i > d_i^{\max}$ then
4:         $d_i$ = uniformSampling($d_i$, $d_i^{\max}$)
5:     end if
6: end if
7: return $d_i$
```

5 Dart implementation

In this section, we first present the architecture design of Dart. Then we present the mechanisms Dart used to deal with task failures and data skew. Finally, we illustrate how Dart estimates the duration of map tasks and the remaining execution time of stragglers.

5.1 Architecture design

We implemented Dart prototype on top of Hadoop 2.2.0 (Apache, 2016). We have added two extra components in the framework and modified the ApplicationMaster of Hadoop MapReduce. The implementation consists of the following four main components (Fig. 7):

1. Task sampler: Before scheduling the job, the task sampler runs multiple sample reduce tasks to obtain the distribution of the task's execution time with different input sizes. Then the results are sent to the job profiler to construct the whole profile information of the job.

2. Job profiler: The job profiler collects the statistic job profile information from both the previous execution of the recurring jobs and the task sampler. After the job is initialized, it sends this information to ApplicationMaster for the initial scheduling.

3. Job completion time estimator: Given the job profile information, the completion time estimator calculates the job completion time based on the iterative estimation method before scheduling map tasks and sends the estimated values to the scheduler.

4. Task scheduler: When the job is submitted, the task scheduler communicates with the Resource Manager (RM) by sending heartbeats periodically to request appropriate resources for executing tasks. When the scheduler receives available resources, it runs the iterative approaching algorithm to dynamically launch map tasks. Once the termination condition is satisfied, the scheduler terminates redundant map tasks and turns to the revise stage. In this stage, whenever a map task is completed, the scheduler runs the revising algorithm to mitigate stragglers. Besides, the scheduler monitors individual tasks and collects the real-time job profile information for more precise scheduling.

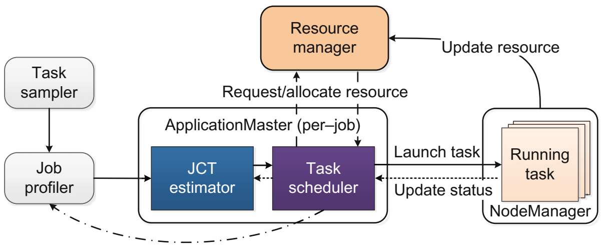


Fig. 7 Architecture design of Dart

5.2 Performance optimization

Our previous approach is designed based on the assumptions that no task failure occurs and that the job has a uniform data distribution. However, during the real execution of MapReduce jobs, task failures and data skew significantly influence the performance of scheduling progress and may cause missed deadlines. Therefore, we make two optimization designs to deal with these adverse scenarios in Dart.

5.2.1 Task failures

In MapReduce, when a map task fails, the system simply restarts the failed task in default. Since a restarted task can result in the increase of job completion time and therefore cause missed deadlines, the problem of task failures must be handled. Dart uses a simple method to handle this problem: first, $k- = 1$ when a task fails; second, this failed task can be restarted only if the estimated job completion time based on the restarted task is less than the deadline, and $k+ = 1$ after the failed task restarts.

More fine-grained methods can be used to tackle task failures, such as running a smaller input if the original task cannot be restarted, but it can potentially complicate our implementation. In practice, we found that task failures have little impact on the scheduling performance even with our coarse-grained method.

5.2.2 Data skew

Skewed data is caused by the uneven distribution of data to partitions. Each reduce task receives different amounts of shuffled volumes, making reduce task durations vary over a large range, which brings the estimated error of reduce phase duration. The estimation error then leads to either suboptimal k or missed deadlines.

Handling data skew in Dart can be divided into two parts: before and after the reduce phase begins. First, we represent the maximum execution time of reduce tasks as the duration of the reduce phase before this phase begins. Note that this is an approximate value because we assume that all reduce tasks start at the same time. We estimate this value by calculating $f(d_{k_n})$, where d_{k_n} is the estimated maximum reduce input based on the current k_n. k_i $(0 < i < n)$ represents the maximum reduce input in the ith previous execution. d_{k_n} is calculated based on the fact that the reduce task's input is proportional to the map task's output. Therefore, since k_i $(0 < i < n)$ and d_{k_i} can be accessed from the job profile of J, we can calculate d_{k_n} as

$$d_{k_n} = \frac{1}{n-1}\sum_{i=1}^{n-1}\frac{k_n}{k_i}d_{k_i}. \tag{13}$$

After the reduce phase begins, the individual reduce input is known and the assumption that all reduce tasks start concurrently is broken. Therefore, we use the method described in Section 4.2.2 to deal with the data skew problem.

5.3 Time estimation

Dart combines previous execution information with real-time task status to estimate important parameters such as the average duration of map tasks T_{m}^{a} and the remaining execution time of stragglers T_s^{rem}. T_{m}^{a} is initialized from the job profile information of the previous execution. During runtime, once any map task m_i finishes, T_{m}^{a} is updated as $\alpha\times T_{\text{m}}^{\text{a}}+(1-\alpha)\times T_i^{\text{m}}$, where α is an updating factor and T_i^{m} is the duration of m_i. In practice, we have found that setting α as 0.8 can result in good performance. Besides, the average partition size of map tasks p is updated when any map task is completed.

Different methods for estimating T_s^{rem} can be plugged into Dart. For example, we can simply use the score of unfinished progress divided by its progress rate to illustrate the remaining duration, or we can use task execution information such as the fraction of input data read and the past time to linearly estimate the time to completion, which performs well when analytic jobs are IO-intensive.

6 Evaluation

We deployed Dart on a cluster that consists of 64 virtual machines (VMs). Each VM contained two virtual cores and 4 GB RAM. Based on two real-world workloads, we measured the performance improvements of Dart by comparing it with current approaches and then evaluated the estimation accuracy of Dart. We next analyzed Dart's ability to mitigate stragglers and performed a sensitivity analysis to confirm the range of the threshold used in Dart. All experiments were repeated at least five times, and we plotted median values across runs and used error bars to show the minimum and maximum values.

6.1 Experimental setup

6.1.1 Workload

We used workload traces from CMU's research cluster (Ren *et al.*, 2013) and Facebook's production cluster (Chen *et al.*, 2012) to evaluate our work. The trace of CMU included more than 20-month logs and captured over 100 000 MapReduce jobs on a 64-node cluster, while the trace of Facebook was from a mix of batch jobs and captured over half a million jobs on a 3500-node cluster. We used a scaled down version of CMU's trace that comprises 4208 jobs scheduled by one single jobtracker and replay of a 24-hour sample of Facebook's trace by using the SWIM tool (Cloudera, 2013), both of which consist of common MapReduce applications such as 'word count', 'sort', 'join', and 'group by'. To fit within our cluster, we used the same inter-arrival times, number of tasks, and task-to-rack mappings as in the traces. Finally, we obtained all jobs binned by the number of tasks, using three distinctions Small (<100 tasks), Medium (100–1000 tasks), and Large (>1000 tasks).

Since the above jobs are common MapReduce jobs that have no deadline constraints, we convert these jobs to deadline-bound jobs as follows: For a MapReduce job J, we define δ as the ratio of actual resource allocations to the MRA of J and μ as the ratio of the given deadline to the MDT described in Section 2. So, $\delta = 1$ means that J obtains the maximum resource allocations, and each task of J can obtain a free container to run without waiting, and $\mu = 1$ means that the deadline of J is set to be the minimum deadline threshold.

We further marked three main environmental configurations for implementing our experiments (Table 2). Note that C1 is the situation where the deadline is set to be too small to complete all tasks of the job even with maximum resources, C2 is another situation where the assigned resources are limited, and C3 is the ideal situation where the jobs obtain the maximum resources while all deadlines are set to be the minimum deadline threshold.

Table 2 Experimental configurations

Configuration	δ	μ
C1	1.00	0.95
C2	0.5	1.6
C3	1	1

6.1.2 Baseline

We compared Dart with two existing methods. One is the state-of-the-art scheduling framework ARIA (Verma *et al.*, 2011), which can dynamically estimate and allocate appropriate resources required for completing all tasks of the job within the deadline. The other is preselected sampling (denoted as PS) (Agarwal *et al.*, 2013), which produces several samples before executing the job and the scheduler picks the most appropriate sample to operate on for meeting the deadline.

6.1.3 Metric

Since our goal is to meet the deadline while maximizing k, we measure the performance improvements of Dart in three ways. First, we measure the ability of meeting deadlines by analyzing the percentage of deadlines missed. Second, we measure the ability of maximizing volumes of processed data as follows: Given the specified deadline and resources, we simulate the jobs by replaying the trace at the task level to figure out the maximum number of scheduled map tasks (denoted as $k_{\max}$) and then define the deviation coefficient as $\frac{k}{k_{\max}}$, where k is the real number of scheduled map tasks. Finally, to measure the estimation accuracy of our method, we define the estimation error as $|\frac{T-\hat{T}}{T}|$, where T is the actual time and $\hat{T}$ is the estimated value.

6.2 Improvements in meeting deadlines

We first compared Dart with our baselines in meeting deadlines by running 140 jobs, which are extracted from two workload traces, in the three environmental configurations described in Table 2.

In Fig. 8a, the x axis shows the different environmental configurations and the y axis shows the percentage of deadlines missed. The results show that only about 3.5% of all jobs miss their deadlines by using Dart, while the performance of PS is a little worse, achieving 5.8%. ARIA tries to allocate enough resources to complete all tasks of the job, which can achieve good performances in the ideal C3 but significantly result in missed deadlines in C1 and C2, thereby showing the worst performance.

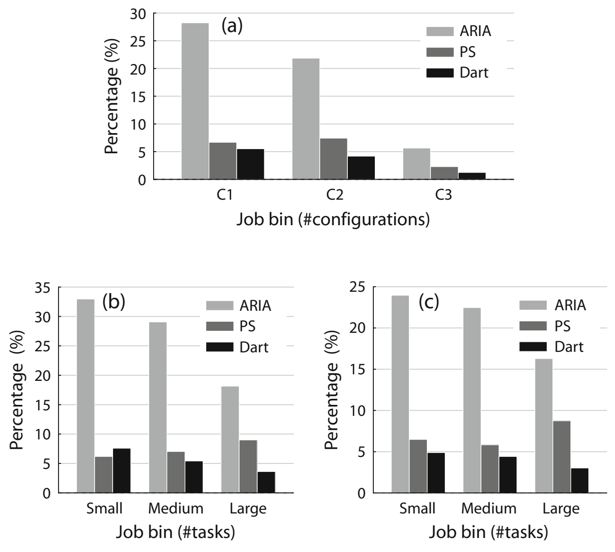


Fig. 8 Overall comparison of the percentage of deadlines missed: (a) comparison of the percentage of missed deadlines in three configurations; (b) comparison binned by the job size in C1; (c) comparison binned by the job size in C2

We then illustrate the results divided by the job size in C1 and C2. Figs. 8b and 8c show the percentage of missed deadlines obtained using Dart, ARIA, and PS to deal with small, medium, and large jobs in C1 and C2, respectively. It can be seen that our method performs better for large jobs compared to small and medium jobs. The reason is that, since large jobs tend to have longer job completion time and multiple waves of map tasks, Dart can provide a more accurate estimation by using the iterative estimation method as the execution proceeds. ARIA also benefits from large jobs because its dynamic estimation method is based on the coarse-grained multi-wave model. On the contrary, the performance of PS deteriorates as the job size increases, because it is more difficult to provide an appropriately sized sample that is statically built before the execution.

Further, we can analyze how the benefits are sensitive to changing deadlines and the amount of resources. We plot the percentage of deadlines missed over different deadlines and amount of resources. Fig. 9a shows that for jobs that have maximum resource allocations, Dart keeps the percentage of missed deadlines under 10% when the given deadline changes from MDT to 0.8MDT, reaching up to 49.8% improvement against PS and 90.5% improvement against ARIA when the deadline drops to 0.8MDT. Besides, Fig. 9b shows that for jobs whose deadlines are set to be 1.6MDT, the percentage of deadlines missed changes from 2.4% to 8.2% by using Dart when the allocated resources change from 0.7MRA to 0.3MRA. We can obtain up to 38.5% improvement aginst PS and 86.3% improvement against ARIA over 0.3MRA.

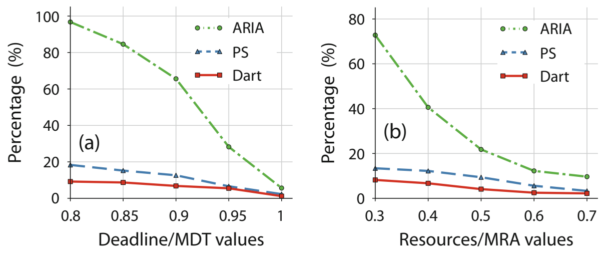


Fig. 9 Performance comparison with changing deadlines and resource allocations: (a) comparison of the percentage of missed deadlines under tight deadlines; (b) comparison of the percentage of missed deadlines under limited resources

6.3 Improvements in maximizing the processed data

We then illustrate the ability of Dart in maximizing the processed data within the given deadlines compared to PS. We used only the results in C1 and C2, and Fig. 10 summarizes our results. The x axis shows the deviation coefficient $\frac{k}{k_{\max}}$. Values below 100% represent experiments that have not processed the maximum data volumes, and values above 100% represent experiments that process more than the maximum volumes, which means that they would have missed the deadlines. The y axis shows the CDF of Dart and PS.

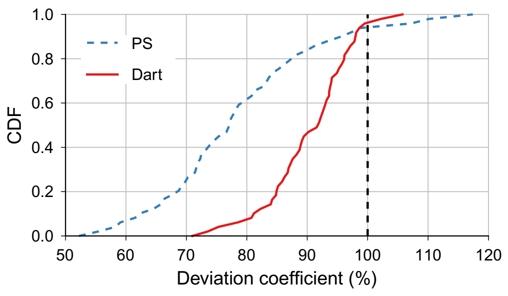


Fig. 10 Cumulative distribution function (CDF) of the deviation coefficient with PS as the baseline

Totally, Dart improved in processing more data than PS by about 13.8%, and achieved an average deviation coefficient of 88.4%, which means that Dart is capable of processing near-maximal data volumes. Moreover, note that although Dart and PS had nearly the same fraction of values exceeding 100%, Dart processed less exceeded data than PS. We further calculated the coefficient of variation (CoV) of the deviation coefficient, which is the ratio of the standard deviation to the mean, to measure the dispersion of the results' distribution. Dart reduced the CoV by 51.3% compared to PS, meaning that Dart conspicuously reduces the variance of processed data volumes.

Fig. 11 shows the results of the average deviation coefficient divided by the job size. The value of Dart remained stable between 85.2% and 95.1%, and the value increased as the job size became larger. However, the value of PS decreased from 89.2% to 73.7% as the job size increased. Besides, this shows that Dart can process near-maximum data volumes despite the variation in job sizes, while PS is suitable only for small-sized jobs.

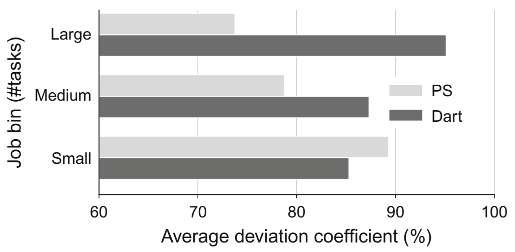


Fig. 11 Comparison of the average deviation coefficient with PS as the baseline

6.4 Estimation accuracy

6.4.1 Overall comparison of estimation errors

To understand why Dart can achieve such good performance, we evaluated the estimation accuracy of the job completion time. Accurate estimation can reduce errors in task scheduling so that the deadline can be met while more data can be processed.

We first evaluated the overall estimation error by plotting the CDF of Dart and ARIA (Fig. 12a). Dart totally reduced the estimation error by 55.8% compared to ARIA. Moreover, 80% of jobs run by Dart had an estimation error of less than 14.8%, while the estimation errors of 80% of ARIA's jobs were less than 33.2%. These results showed that Dart gives a more accurate estimation of job completion time and efficiently reduces the variance in the estimation error compared to ARIA.

To figure out why Dart can obtain such accurate estimations, we further illustrate the average estimation error of detailed durations including map, shuffle, and reduce phases. Fig. 12b shows the results of the map phase. From the figure, we can see that Dart performs much better than ARIA in C1 and C3, reducing the average estimation error by 67.7% and 65.4%, respectively. The reason is that the map phases of jobs in C1 or C3 have only one wave of tasks since the maximum resources are allocated, and ARIA's model would consider the duration of the map phase as the average duration of map tasks in this situation. However, although the map phase processes in one wave, each map task has a different start time, causing the map phase's duration to be usually larger than the average duration of map tasks. Dart estimates the duration of the map phase based on the iterative estimation method rather than ARIA's coarse-grained method, and therefore can obtain a more accurately estimated duration.

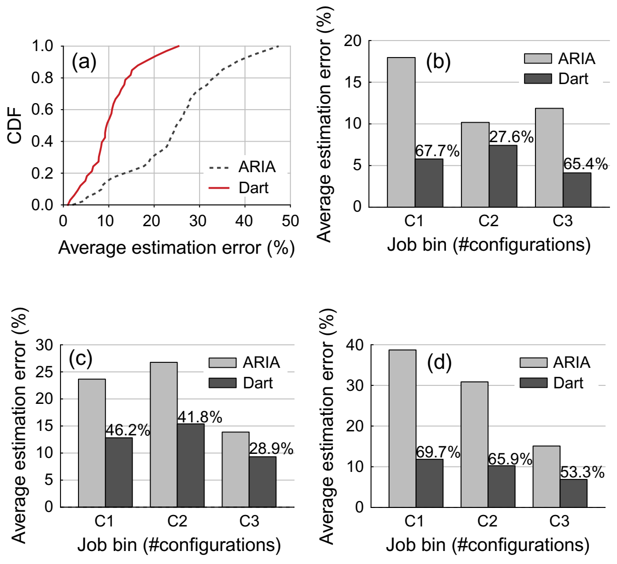


Fig. 12 Overall comparison of the estimation error: (a) CDF of the estimation error with ARIA as the baseline; (b) map phase; (c) shuffle phase; (d) reduce phase. Numbers on the bar represent percentage reduction in the estimation error by using Dart

Fig. 12c shows the estimation error in the shuffle phase. We observed that Dart also performs better in all three configurations, reducing the error by 28.9%–46.2%, but its performance is not as good as the performances in the map phase. This is probably because Dart uses only a heuristic to do the estimation and there are many uncertain factors influencing the shuffle phase, such as the changing bandwidth and data skew. The model of ARIA estimates the shuffle phase's duration based on the information of previous execution, which is capable of handling the jobs in the ideal C3. However, ARIA performs much worse in C1 and C2 because the shuffle phase is shortened as only a part of the map tasks is processed.

We further evaluated our estimation method for the reduce phase. Fig. 12d shows that Dart reduces the average estimation error by 53.3%–69.7%, which is much better than those for the shuffle phase. The main reason is that, since the execution time of reduce tasks is dynamically changing with the input size, Dart does use previous information, but runs samples in advance to do nonlinear regression and then estimates the duration of the reduce phase based on the regression model and real-time input of reduce tasks. On the contrary, ARIA uses statically historical information to estimate the dynamically changing duration, thus causing inaccurate estimations, especially in C1 and C2.

6.4.2 Fine-grained analysis of estimation errors

Besides the general results presented above, we provided the average estimation error of different phases during the job execution (Table 3). We divided the job execution into four parts: MS–SS (map phase starts–shuffle phase starts), SS–ME (shuffle phase starts–map phase ends), ME–SE (map phase ends–shuffle phase ends), and RS–RE (reduce phase starts–reduce phase ends). This is due to the fact that the shuffle phase partially overlaps with the map phase and the reduce phase starts after the shuffle phase ends. For the estimation of the map phase, we can see that the error drops sharply to 4.53% after the shuffle phase starts because of our fine-grained method. As for the estimation of the shuffle phase, the error is large at first, nearly 20% when the shuffle phase starts, but decreases to 10.62% at the end of the shuffle phase. The error of the reduce phase estimation drops sharply after the map phase ends because the number of map tasks becomes fixed and finally drops to 5.93% during the reduce phase.

Table 3 Average estimation error of different phases during the job execution

Phase	Average estimation error (%)			
	MS–SS	SS–ME	ME–SE(RS)	RS–RE
Map	11.71	4.53	–	–
Shuffle	–	19.84	10.62	–
Reduce	15.48	12.76	7.11	5.93

To measure the accuracy of our method when estimating the duration of reduce tasks, we have run four classical and representative applications with different configurations to evaluate the estimation error (Table 4). Table 4 shows that although the linear assumption results in good estimation accuracy for applications which are IO-intensive such as sort and word count, the performance of the linear method degrades largely for CPU-intensive applications such as WikiTrends and Naive Bayes. On the contrary, Dart can obtain better estimation accuracy for both IO- and CPU-intensive applications. Further, the more sample tasks Dart uses, the more accurate the estimation is. When the number of samples increases to 10, Dart totally reduces the average estimation error by 71.4% compared to the linear method.

Table 4 Average estimation error of reduce tasks

App.	Average estimation error (%)			
	Linearity	Dart (3)	Dart (5)	Dart (10)
Sort	10.12	15.02	9.84	4.41
Word count	19.23	18.28	11.41	8.71
WikiTrends	37.80	27.73	18.03	12.16
Naive Bayes	31.68	25.92	14.34	6.45

Linearity means that we assume the duration of reduce tasks scales linearly with the input size; the number in brackets indicates how many sample tasks have been used to perform the estimation

6.5 Effects of straggler mitigation

To evaluate the performance of Dart on mitigating stragglers, we manually slowed down eight VMs to simulate stragglers in our cluster of 64 VMs by running four CPU-intensive processes and four IO-intensive processes on each VM. Twenty jobs with different sizes, such as sort, grep, and word count, were selected from the workloads for running to evaluate the performance. We used three methods as Dart's baselines: (1) LATE (Zaharia *et al.*, 2008), which speculatively launches copies of stragglers on fast nodes; (2) Dart without revising (D-WOR), which does not use the revising mechanism to mitigate stragglers; (3) Dart with map revising (D-WMR), which applies straggler mitigation on map tasks but not on reduce tasks.

Table 5 shows the percentage of deadlines missed in both C1 and C2. We can see that in C1, Dart significantly reduces the percentage compared to LATE. The performance of Dart is nearly twice that of D-WOR. Small jobs benefit the most, with the performance improved by 67.7%, 47.5%, and 2.6% compared to those of LATE, D-WOR, and D-WMR, respectively. This is because small jobs tend to have shorter completion time, resulting in little time for speculation. However, Dart can still efficiently kill stragglers even when the speculative window is short. As for the results in C2, Dart also gains considerable performance improvements over LATE. One difference is that the performance of D-WOR is even worse than that of LATE. The reason may be that since the deadline in C2 is set to be longer than that in C1, there is more time left for speculation, thereby benefiting LATE. Besides, more time means that Dart can use the revising mechanism to carefully handle both stragglers while D-WOR does nothing. Finally, note that Dart is a little better than D-WMR, indicating that mitigating both map and reduce stragglers can achieve the best performance, but the main improvement is due to map straggler mitigation.

Table 5 Percentage of deadlines missed with different sizes of jobs in C1 and C2

	Job bin	Percentage of deadlines missed (%)			
		LATE	D-WOR	D-WMR	Dart
C1	Small	22.9	14.1	7.6	7.4
	Medium	13.0	9.5	5.9	5.6
	Large	10.3	8.7	4.2	3.8
C2	Small	10.8	12.5	4.7	3.8
	Medium	7.4	14.8	4.3	3.5
	Large	5.2	11.7	3.1	2.5

The improvements in the previous sections are based on a StragglingThreshold of 80% $T_{\mathrm{m}}^{\mathrm{a}}$. Since StragglingThreshold is the metric to decide whether to kill a task when the speculative window is negative, we analyze the sensitivity of Dart's performance to the StragglingThreshold (Fig. 13). Low values of StragglingThreshold result in good performance of meeting deadlines but low values of the deviation coefficient, while high values incur performance loss in meeting deadlines but increase the value of the deviation coefficient. This is because the low value of StragglingThreshold allows tasks to be killed more easily, which can significantly reduce the completion time but fail in maximizing the processed data volumes, while the high value shows the opposite. Our results show that this exploration–exploitation tradeoff is optimized at a StragglingThreshold of 80% and the performance drops off sharply around this point.

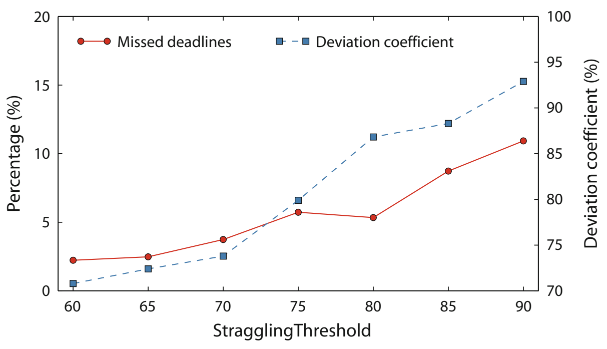


Fig. 13 Performance versus StragglingThreshold

6.6 Effects of optimization designs

Finally, we measured the effects of optimization designs described in Section 5. We first evaluated how Dart handles task failures by running the same 140 jobs in C1 and C2 but restarting some VMs during runtime to force some tasks to fail. We measured only overall performance improvements such as the percentage of deadlines missed and the average deviation coefficient. Fig. 14 shows the results. From the figure, we can see that the percentage of deadlines missed and the deviation coefficient remain stable despite the change in the number of failed tasks, indicating that task failures have little impact on the performance of Dart.

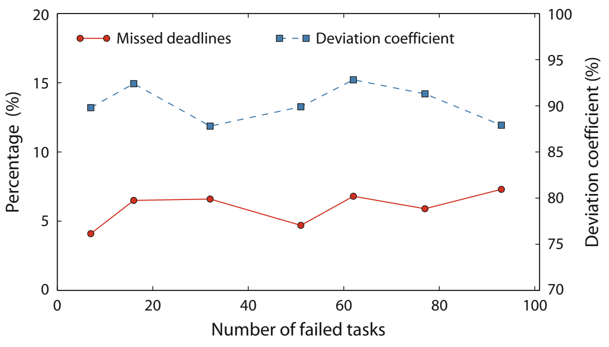


Fig. 14 Performance with task failures

We next evaluated the performance improvement due to the tackling of data skew by using two applications. One is the inverted index (II), which builds an II from a 13 GB dataset of Wikipedia, and the other is the page rank (PR), which ranks websites by counting the number and quality of links to a page. We applied the algorithm to a dataset whose size is 2.1 GB (Freebase Data Dumps, https://developers.google.com/freebase/). Our baseline is Dart without skew tuning (D-WOST), which does not apply any mechanism to tackle data skew. Fig. 15a shows the improvement in missed deadlines, and we can see that Dart reduces the percentage of deadlines missed by 58.3% in II and 65.1% in PR, compared to D-WOST. On the other hand, Fig. 15b shows that Dart increases the deviation coefficient by 19.5% in II and 10.6% in PR. In summary, Dart performs better in all aspects of performance compared to D-WOST by implementing the optimization designs.

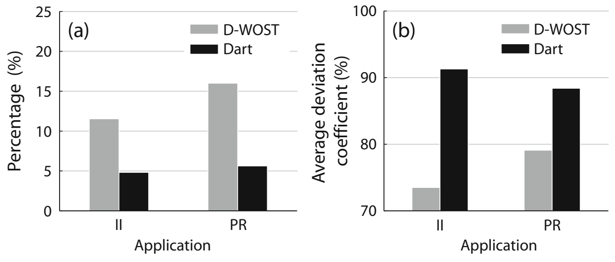


Fig. 15 Improvement in meeting deadlines due to tackling data skew: (a) improvement in the percentage of missed deadlines; (b) improvement in the deviation coefficient

7 Related work

7.1 Deadlines in data-parallel clusters

Supporting deadline requirements for MapReduce jobs has been an area of active research, and several recent schemes have been proposed (Kc and Anyanwu, 2010; Polo *et al.*, 2010; Verma *et al.*, 2011; Ferguson *et al.*, 2012; Verma *et al.*, 2012; Li *et al.*, 2014; Zacheilas and Kalogeraki, 2014; Wang *et al.*, 2015). Some researchers estimated and allocated the minimum amount of resources to complete a job before its deadline (Kc and Anyanwu, 2010; Polo *et al.*, 2010; Verma *et al.*, 2011), but missing deadlines can still occur when the deadline is small or resources are lacking. Wang *et al.* (2015) proposed a deadline-constraint scheduling algorithm in MapReduce, which is based on a novel sequence-based execution strategy, but they did not consider the approximation processing cases in MapReduce.

There are also some studies focusing on the deadline-aware scheduling mechanism for data-parallel jobs which are modeled as the directed acyclic graph (DAG) (Ferguson *et al.*, 2012) or the workflow that contains multiple interdependent MapReduce jobs (Li *et al.*, 2014). We believe that these approaches can be combined with Dart to support a more generalized deadline-aware approximation processing.

As for multiple concurrent job scheduling scenarios, Verma *et al.* (2012) proposed a resource management mechanism that is based on earliest deadline first (EDF). Zacheilas and Kalogeraki (2014) presented the least laxity first (LLF) policy for dynamically determining job ordering and resource allocation. Although Dart currently focuses on a single-job scenario and makes only task-level scheduling decisions, it can be further integrated into the resource management frameworks mentioned above to fit in a multiple-job scenario.

7.2 Approximate scheduling

Approximate query techniques (Acharya *et al.*, 1999; Bell Laboratories, 2001; Agarwal *et al.*, 2013) execute jobs on an appropriately sized sample of data based on the query's accuracy or deadline requirements. However, since they have not considered the impact of data locality when choosing samples, these sample-based approaches can achieve low locality and process only a suboptimal amount of data.

Other studies on approximation techniques focus on optimizing the scheduling performance of approximation jobs. KMN (Venkataraman *et al.*, 2007) focuses on improving data locality by choosing any K local blocks out of a total of N blocks and balancing network transmission by launching a small number of additional tasks. However, it does not pay attention to the problem of meeting deadlines for MapReduce jobs.

MapCheckReduce (MCR) (Wang *et al.*, 2014) uses the check–kill mechanism to speed up approximation jobs. It can terminate the input stage once some conditions are satisfied. However, MCR has only a simple termination condition to support error constraints, but no support for the deadline constraint.

7.3 Estimating job completion time

To estimate the job completion time accurately, current studies such as ParaTimer (Morton *et al.*, 2010a) and Parallax (Morton *et al.*, 2010b) first estimate the job's remaining work and then use prior runs of the same job on a data sample to estimate the relative processing speed. Finally, they estimate the job completion time as the sum of past time and the time remaining for unfinished work, which is calculated as the division of the remaining work and the measured speed. However, this coarse-grained method is inaccurate because the speed measured in samples would change with the input size and it may be overly optimistic about failures and data skew.

8 Conclusions

In this paper, we have presented a deadline-oriented task scheduling approach, called Dart, for satisfying the deadline requirements of MapReduce approximation jobs. Dart consists of two scheduling stages: approach and revise. In the approach stage, a greedy approaching scheduling method is performed to efficiently launch map tasks. In the revise stage, both map and reduce straggler mitigation techniques are proposed to prevent missed deadlines. Besides, an iterative estimation method is used in Dart to estimate the real-time job completion time when scheduling map tasks. Dart also offers solutions for task failures and data skew problems to further improve the performance. We have implemented Dart in Hadoop 2.2.0 and conducted comprehensive experiments on workloads from OpenCloud and Facebook. The results show that Dart successfully meets 96.5% of the given deadlines and processes near-maximum data volumes even with tight deadlines and limited resources. The state-of-the-art performance is largely due to the fact that Dart reduces the estimation error by 55.8% compared to the baseline. Further experiments show that Dart can efficiently handle a variety of real system challenges such as stragglers, task failures, and data skew.

In conclusion, Dart is an advanced approach to provide deadline guarantees for the MapReduce approximation job due to its multiple advantages, such as highly accurate estimation, dynamic task scheduling, and straggler mitigation. We believe that Dart will be a valuable component for large-scale MapReduce applications when meeting deadlines is the first priority.

References

Acharya, S., Gibbons, P., Poosala, V., 1999. Aqua: a fast decision support system using approximate query answers. Proc. 25th Int. Conf. on Very Large Data Bases, p.754-757.

Agarwal, S., Mozafari, B., Panda, A., *et al.*, 2013. Blinkdb: queries with bounded errors and bounded response times on very large data. Proc. 8th ACM European Conf. on Computer Systems, p.29-42. https://doi.org/10.1145/2465351.2465355

Ananthanarayanan, G., Kandula, S., Greenberg, A.G., *et al.*, 2010. Reining in the outliers in Map-Reduce clusters using Mantri. Proc. 10th USENIX Symp. on Operating Systems Design and Implementation, p.24-38.

Ananthanarayanan, G., Ghodsi, A., Shenker, S., *et al.*, 2013. Effective straggler mitigation: attack of the clones. Proc. 10th USENIX Symp. on Networked Systems Design and Implementation, p.185-198.

Ananthanarayanan, G., Hung, M.C.C., Ren, X., *et al.*, 2014. Grass: trimming stragglers in approximation analytics. Proc. 11th USENIX Symp. on Networked Systems Design and Implementation, p.289-302.

Apache, 2016. The Apache Hadoop Project. http://hadoop.apache.org/

Bates, D.M., Watts, D.G., 1988. Nonlinear regression inference using the linear approximation. *In*: Jantsch, E., Waddington, C. (Eds.), Nonlinear Regression: Iterative Estimation and Linear Approximations. Wiley Online Library, p.142-167. https://doi.org/10.1002/9780470316757.ch2

Bell Laboratories, 2001. Approximate Query Processing: Taming the Terabytes. http://www.vldb.org/conf/2001/tut4.pdf

Chen, Y., Ganapathi, A., Griggith, R., *et al.*, 2011. The case for evaluating MapReduce performance using workload suites. Proc. IEEE 19th Int. Symp. on Modeling, Analysis & Simulation of Computer and Telecommunication Systems. https://doi.org/10.1109/MASCOTS.2011.12

Chen, Y., Alspaugh, S., Katz, R., 2012. Interactive analytical processing in big data systems: a cross-industry study of MapReduce workloads. *Proc. VLDB Endow.*, **5**(12):1802-1813. https://doi.org/10.14778/2367502.2367519

Chowdhury, M., Zaharia, M., Ma, J., *et al.*, 2011. Managing data transfers in computer clusters with orchestra. *SIGCOMM Comput. Commun. Rev.*, **41**(4):98-109. https://doi.org/10.1145/2043164.2018448

Chowdhury, M., Zhong, Y., Stoica, I., 2014. Efficient coflow scheduling with varys. *SIGCOMM Comput. Commun. Rev.*, **44**(4):443-454. https://doi.org/10.1145/2740070.2626315

Cloudera, 2013. Statistical Workload Injector for MapReduce. https://github.com/SWIMProjectUCB/SWIM

Dean, J., Ghemawat, S., 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM*, **51**(1):107-113. https://doi.org/10.1145/1327452.1327492

Ferguson, A.D., Bodik, P., Kandula, S., 2012. Jockey: guaranteed job latency in data parallel clusters. Proc. 7th ACM European Conf. on Computer Systems, p.99-112. https://doi.org/10.1145/2168836.2168847

Herodotou, H., Lim, H., Luo, G., 2011. Starfish: a self-tuning system for big data analytics. Proc. 7th Biennial Conf. on Innovative Data Systems Research, p.261-272.

Hu, M., Wang, C., You, P., *et al.*, 2015. Deadline-oriented task scheduling for mapreduce environments. *LNCS*, **9529**:359-372. https://doi.org/10.1007/978-3-319-27122-4_25

Kc, K., Anyanwu, K., 2010. Scheduling Hadoop jobs to meet deadlines. IEEE 2nd Int. Conf. on Cloud Computing Technology and Science, p.388-392. https://doi.org/10.1109/CloudCom.2010.97

Li, S., Hu, S., Wang, S., *et al.*, 2014. Woha: deadline-aware Map-Reduce workflow scheduling framework over Hadoop clusters. IEEE 34th Int. Conf. on Distributed Computing Systems, p.93-103. https://doi.org/10.1109/ICDCS.2014.18

Liu, J., Shih, K., Lin, W., *et al.*, 1994. Imprecise computations. *Proc. IEEE*, **82**:83-94. https://doi.org/10.1109/5.259428

Lohr, S., 2009. Simple probability samples. *In*: Sampling: Design and Analysis. Addison-Wesley, London, p.35-67.

Marquardt, D.W., 1963. An algorithm for least-squares estimation of nonlinear parameters. *J. Soc. Ind. Appl. Math.*, **11**(2):431-441.

Morton, K., Balazinska, M., Grossman, D., 2010a. ParaTimer: a progress indicator for MapReduce dags. Proc. ACM SIGMOD Int. Conf. on Management of Data, p.507-518. https://doi.org/10.1145/1807167.1807223

Morton, K., Friesen, A., Balazinska, M., *et al.*, 2010b. Estimating the progress of MapReduce pipelines. Proc. IEEE 26th Int. Conf. on Data Engineering, p.681-684. https://doi.org/10.1109/ICDE.2010.5447919

Motulsky, H.J., Ransnas, L.A., 1987. Fitting curves to data using nonlinear regression: a practical and nonmathematical review. *FASEB J.*, **1**(5):365-374.

OREILLY, 2013. Interactive Big Data Analysis Using Approximate Answers. https://tinyurl.com/k5favda/

Polo, J., Carrera, D., Becerra, Y., *et al.*, 2010. Performance-driven task co-scheduling for MapReduce environments. Proc. IEEE Int. Congress on Network Operations and Management Symp., p.373-380. https://doi.org/10.1109/NOMS.2010.5488494

Ren, K., Kwon, Y., Balazinska, M., *et al.*, 2013. Hadoop's adolescence: an analysis of Hadoop usage in scientific workloads. *Proc. VLDB Endow.*, **6**(10):853-864. https://doi.org/10.14778/2536206.2536213

Vavilapalli, V.K., Murthy, A.C., Douglas, C., *et al.*, 2013. Apache Hadoop Yarn: yet another resource negotiator. Proc. 4th Annual Symp. on Cloud Computing, p.5:1-5:16. https://doi.org/10.1145/2523616.2523633

Venkataraman, S., Panda, A., Ananthanarayanan, G., *et al.*, 2007. The power of choice in data-aware cluster scheduling. Proc. 11th USENIX Symp. on Operating Systems Design and Implementation, p.301-316.

Verma, A., Cherkasova, L., Campbell, R.H., 2011. Aria: automatic resource inference and allocation for MapReduce environments. Proc. 8th ACM Int. Conf. on Autonomic Computing, p.235-244. https://doi.org/10.1145/1998582.1998637

Verma, A., Cherkasova, L., Kumar, V.S., *et al.*, 2012. Deadline-based workload management for MapReduce environments: pieces of the performance puzzle. Proc. IEEE Int. Congress on Network Operations and Management Symp., p.900-905. https://doi.org/10.1109/NOMS.2012.6212006

Wang, C., Peng, Y., Tang, M., *et al.*, 2014. MapCheckReduce: an improved MapReduce computing model for imprecise applications. Proc. IEEE Int. Congress on Big Data, p.366-373. https://doi.org/10.1109/BigData.Congress.2014.61

Wang, X., Shen, D., Bai, M., *et al.*, 2015. SAMES: deadline-constraint scheduling in MapReduce. *Front. Comput. Sci.*, **9**(1):128-141. https://doi.org/10.1007/s11704-014-4138-y

Zacheilas, N., Kalogeraki, V., 2014. Real-time scheduling of skewed MapReduce jobs in heterogeneous environments. Proc. 11th Int. Conf. on Autonomic Computing, p.189-200.

Zaharia, M., Konwinski, A., Joseph, A.D., *et al.*, 2008. Improving MapReduce performance in heterogeneous environments. Proc. 8th USENIX Symp. on Operating Systems Design and Implementation, p.7-21.

Zaharia, M., Borthakur, D., Sen, S., *et al.*, 2010. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. Proc. 5th European Conf. on Computer Systems, p.265-278. https://doi.org/10.1145/1755913.1755940