Degraded-First Scheduling for MapReduce in Erasure-Coded Storage Clusters

Runhui Li, Patrick P. C. Lee, Yuchong Hu
Department of Computer Science and Engineering, The Chinese University of Hong Kong
{*rhli, pclee*}*@cse.cuhk.edu.hk, yuchonghu@gmail.com*

***Abstract*—We have witnessed an increasing adoption of erasure coding in modern clustered storage systems to reduce the storage overhead of traditional 3-way replication. However, it remains an open issue of how to customize the data analytics paradigm for erasure-coded storage, especially when the storage system operates in failure mode. We propose degraded-first scheduling, a new MapReduce scheduling scheme that improves MapReduce performance in erasure-coded clustered storage systems in failure mode. Its main idea is to launch degraded tasks earlier so as to leverage the unused network resources. We conduct mathematical analysis and discrete event simulation to show the performance gain of degraded-first scheduling over Hadoop's default locality-first scheduling. We further implement degraded-first scheduling on Hadoop and conduct testbed experiments in a 13-node cluster. We show that degraded-first scheduling reduces the MapReduce runtime of locality-first scheduling.**

I. Introduction

Clustered storage systems, such as GFS [16], HDFS [30], Azure [4], have been widely deployed in enterprises to provide a scalable and reliable storage platform for big data analytics based on MapReduce [9] or Dryad [21]. They stripe data across thousands of *nodes* (or servers) connected over a network, on which parallel data computations can be performed. As a storage system scales, node failures are commonplace [16], and temporary data unavailability becomes prevalent due to frequent transient failures [13] and system upgrades [24]. To ensure data availability at any time, traditional designs of GFS, HDFS, and Azure replicate each data block into three copies to provide double-fault tolerance [4, 16, 30]. However, as the volume of global data surges to the zettabyte scale [15], the 200% redundancy overhead of 3-way replication becomes a scalability bottleneck.

Erasure coding provides an alternative to ensuring data availability. It operates by encoding data blocks into parity blocks, such that a subset of data and parity blocks can sufficiently recover the original data blocks. It is known that erasure coding costs less storage overhead than replication under the same fault tolerance [32]. For example, it can reduce the redundancy overhead of 3-way replication from 200% to 33%, while still achieving higher availability [20].

Extensive efforts (e.g., [13, 20, 29]) have studied the use of erasure coding in clustered storage systems (e.g., GFS, HDFS, Azure) that provide data analytics services. Although erasure coding generally has higher performance overhead than replication, recent results show that the overhead of erasure coding can be mitigated through efficient coding constructions [27], read parallelization [11], and hardware-assisted computations [26]. In particular, when data is unavailable due to node failures, reads are *degraded* in erasure-coded storage as they need to download data from surviving nodes to reconstruct the missing data. In view of this, several studies [20, 22, 29] propose to optimize degraded reads in erasure-coded clustered storage systems, by reducing the amount of downloaded data for reconstruction.

Despite the extensive studies on erasure-coded clustered storage systems, it remains an open issue of how to customize the data analytics paradigm, such as MapReduce [9], for such systems, especially when they operate in failure mode and need to perform degraded reads. In this work, we explore Hadoop's version of MapReduce on HDFS-RAID [18], a middleware layer that extends HDFS to support erasure coding. Traditional MapReduce scheduling emphasizes locality, and implements *locality-first scheduling* by first scheduling local tasks that run on the nodes holding the input data for the tasks. MapReduce is designed with replication-based storage in mind. In the presence of node failures, it re-schedules tasks to run on other nodes that hold the replicas. However, the scenario becomes different for erasure-coded storage, where MapReduce tasks must issue degraded reads to download data from other surviving nodes. Such degraded tasks are typically scheduled to launch after all local tasks are completed, and when they launch, they compete for network resources to download data from surviving nodes. This can significantly increase the overall runtime of a MapReduce job. Thus, a key motivation of this work is to customize MapReduce scheduling for erasure-coded storage in failure mode.

Our observation is that while local tasks are running, the MapReduce job does not fully utilize the available network resources. Thus, this paper proposes *degraded-first scheduling*, whose main idea is to schedule some degraded tasks at earlier stages of a MapReduce job and allow them to download data first using the unused network resources. To this end, this paper makes three contributions:

- We propose *degraded-first scheduling*, a new MapReduce scheduling scheme that improves MapReduce performance in erasure-coded clustered storage systems operating in failure mode. We conduct simple mathematical analysis to demonstrate that degraded-first scheduling improves Hadoop's default locality-first scheduling. Our numerical results show that degraded-first scheduling reduces the MapReduce runtime by

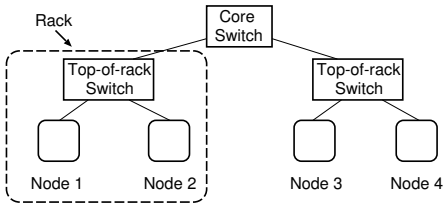


Figure 1. Example cluster with four nodes grouped into two racks.

15% to 43%. We also propose two heuristics that achieve locality preservation and rack awareness, so as to improve the robustness of degraded-first scheduling in general configurations.

- We implement a discrete event simulator for MapReduce to explore the performance gain of degraded-first scheduling in a large-scale cluster. We show that degraded-first scheduling reduces the runtime of locality-first scheduling by up to 39.6% when the cluster runs a single MapReduce job, and by up to 48.6% when multiple MapReduce jobs run simultaneously.
- We implement degraded-first scheduling on Hadoop, and compare the performance of locality-first scheduling and degraded-first scheduling in a 13-node Hadoop cluster. Degraded-first scheduling reduces the MapReduce runtime of locality-first scheduling by up to 27.0% and 28.4% for single-job and multi-job scenarios, respectively.

The rest of the paper proceeds as follows. Section II first presents background details of Hadoop and erasure codes. Section III motivates via an example the issue of Hadoop's default locality-first scheduling. Section IV presents the design of degraded-first scheduling. Section V describes our discrete event MapReduce simulator and presents simulation results. Section VI describes our implementation of degraded-first scheduling on Hadoop and present testbed experimental results. Section VII reviews related work, and Section VIII concludes the paper.

II. Background

A. Hadoop

We consider a Hadoop cluster composed of multiple nodes (or servers) that are grouped into different *racks*. Typical clusters connect all nodes via a hierarchy of switches. Without loss of generality, we consider a simplified two-level case where nodes within each rack are connected via a top-of-rack switch, and all the racks are connected via a core switch. Figure 1 illustrates an example.

Hadoop runs on a distributed file system HDFS [30] for reliable storage. HDFS divides a file into fixed-size *blocks*, which form the basic units for read and write operations. Since node failures are common [16], HDFS uses *replication* to maintain data availability, such that each block is replicated into multiple (by default, three) copies and distributed across different nodes.

Hadoop implements MapReduce [9] for data-intensive computations on HDFS data. We first define the terminologies as follows. A MapReduce program (called *job*) is split into multiple *tasks* of two types: a *map* task processes an input block and generates intermediate results, and a *reduce* task collects the intermediate results through a *shuffle* step, processes them, and outputs the final results to HDFS. MapReduce uses a single *master* node to coordinate multiple *slave* nodes to run the tasks. Each slave has a fixed number of map and reduce *slots*, and each map (reduce) slot is used for running one map (reduce) task. If a slave has free slots available, it requests the master for map or reduce tasks through periodic heartbeat messages. The master then performs task scheduling and decides which task to run first.

In typical deployment environments of MapReduce, network bandwidth is scarce [9]. Thus, MapReduce emphasizes data locality by trying to schedule a map task to run on a (slave) node that stores a replica of the data block, or a node that is located near the data block. This saves the time of downloading blocks from other nodes over the network. Note that reduce tasks cannot exploit locality because they need to download intermediate outputs from multiple slaves. Here, a map task can be classified into three types: (i) *node-local*, in which the task processes a block stored in the same node, (ii) *rack-local*, in which the task downloads and processes a block stored in another node of the same rack, and (iii) *remote*, in which the task downloads and processes a block stored in another node of a different rack. In this paper, we collectively call the first two types *local*, since rack-local tasks can run as fast as node-local tasks if the network speed within the same rack is sufficiently high. The default task scheduling scheme in Hadoop first assigns map slots to local tasks, followed by remote tasks. We call this approach *locality-first scheduling*.

B. Erasure Coding

To reduce the redundancy overhead due to replication, *erasure coding* can be used. An erasure code is defined by parameters (n, k), such that k original blocks (termed *native blocks*) are encoded to form $n - k$ *parity blocks*, and any k out of the n blocks can recover the original k native blocks. We call the collection of the n blocks a *stripe*. Examples of erasure codes include Reed-Solomon codes [28] and Cauchy Reed-Solomon codes [3]. Hadoop's authors propose a middleware layer called HDFS-RAID [18], which operates on HDFS and transforms block replicas into erasure-coded blocks. HDFS-RAID divides a stream of native blocks into groups of k blocks, and encodes each group independently into a stripe according to the parameters (n, k).

In the presence of node failures, native blocks stored in the failed nodes are unavailable (we call them the *lost blocks*). In replication, a read to a lost block can be re-directed to

Algorithm 1 Locality-First Scheduling on HDFS-RAID

```
while a heartbeat comes from slave s do
    for each running job j in the job list do
        for each free map slot on slave s do
            if j has an unassigned local task then
                assign the local task to s
            else if j has an unassigned remote task then
                assign the remote task to s
            else if j has an unassigned degraded task then
                assign the degraded task to s
            end if
        end for
    end for
end while
```

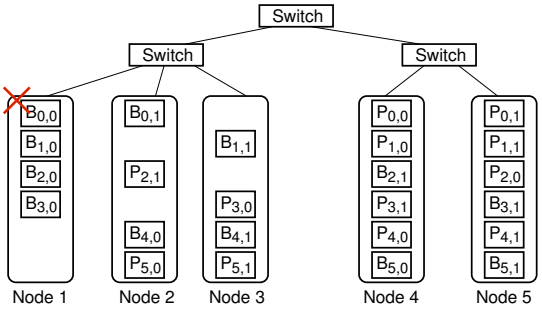


Figure 2. A five-node cluster with 12 native blocks and 12 parity blocks, assuming a (4,2) coding scheme is used for fault-tolerance. We assume that Node 1 fails while MapReduce is running.

another block replica. However, in erasure coding, reading a lost block requires a *degraded read*, which reads the blocks from any k surviving nodes of the same stripe and reconstructs the lost blocks[1]. Although erasure codes are designed to tolerate multiple node failures, it is known that single-node failures are the most common failure recovery scenario in practice [20, 22]. Thus, our discussion focuses on the failure mode where the cluster has only one failed node while a MapReduce job is running, and we address multi-node failures using simulations (see Section V).

MapReduce is compatible with HDFS-RAID and also follows locality-first scheduling. The main difference from traditional replication is that HDFS-RAID reconstructs the lost block via a degraded read. We define a new type of map tasks called *degraded tasks*, which first read data from other surviving nodes to reconstruct the lost block and then process the reconstructed block. Degraded tasks are given the lowest priority in the default locality-first scheduling, and they are scheduled after local and remote tasks. Algorithm 1 summarizes the pseudo-code of the default locality-first scheduling on HDFS-RAID.

III. Motivating Example

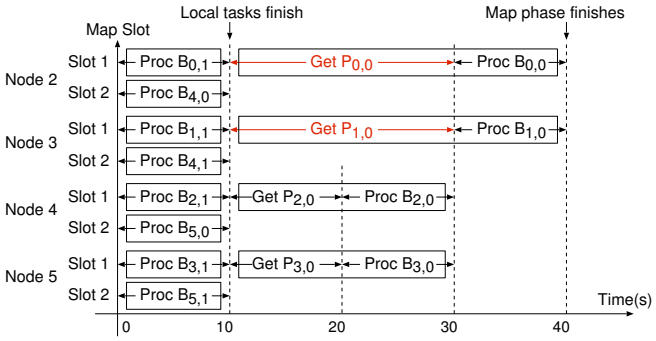
In this section, we elaborate via a motivating example why the default locality-first scheduling hurts MapReduce performance in failure mode. We then provide intuitions how we can improve MapReduce performance.

We first review the default block placement policy of HDFS, and later extend the policy for HDFS-RAID. By default, HDFS uses 3-way replication and places the three replicas using the following rule: the first replica is placed in a random node, and the second and third replicas are placed in two different random nodes that are located in a different rack from the first replica. This placement policy can tolerate (1) an arbitrary double-node failure; and (2) an arbitrary single-rack failure. Correspondingly, for HDFS-RAID, we consider an erasure code satisfying that (1) $n - k \geq 2$; and (2) at most $n - k$ out of n blocks of any stripe are placed on the same rack.

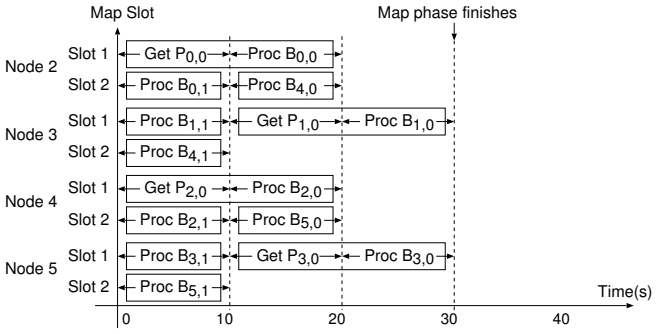
We design an example shown in Figure 2 that realizes the above two conditions. The figure has a two-rack cluster, in which the first rack has three nodes and the second one has two nodes. The racks are connected by 100Mbps Ethernet switches. Let the block size be 128MB. If we ignore transmission overhead, then transmitting a block from one node to another node takes around 10s. Suppose now that the cluster stores a 12-block file, and we use a $(4, 2)$ erasure code to encode a file into six stripes. In the i-th stripe, where $0 \leq i \leq 5$, let $B_{i,0}$ and $B_{i,1}$ be the two native blocks, and $P_{i,0}$ and $P_{i,1}$ be the two parity blocks. We assume that each node has two map slots, meaning that it can run at most two map tasks simultaneously. According to [35], half of map tasks take 1s to 19s to complete. Thus, we assume that the time for processing a map task is also 10s.

We now explain why locality-first scheduling hurts MapReduce performance in failure mode. We consider the duration of the map phase. Suppose that a MapReduce job is processing the stored data while Node 1 is failed, so degraded tasks are triggered to process the lost blocks $B_{0,0}$, $B_{1,0}$, $B_{2,0}$, and $B_{3,0}$. Figure 3(a) shows the map-slot activities with locality-first scheduling. According to Algorithm 1, each surviving node is first assigned local tasks to process its stored blocks. After all the local tasks are completed, the degraded tasks are launched. Suppose we assign the degraded tasks for the lost blocks $B_{0,0}$, $B_{1,0}$, $B_{2,0}$, and $B_{3,0}$ to Nodes 2, 3, 4, and 5, respectively, such that a node just needs to download the first parity block $P_{0,0}$, $P_{1,0}$, $P_{2,0}$, $P_{3,0}$, respectively, from another node to reconstruct the lost block for processing. ~~Node 4 downloads $P_{2,0}$ from Node 3 (in a different rack) and Node 5 downloads $P_{3,0}$ from Node 4 (in the same rack).~~ However, Nodes 2 and 3, located in the same rack, need to compete for the download link of the rack so as to download $P_{0,0}$ and $P_{1,0}$ from another rack, respectively. This doubles the download time, from 10s to 20s. The entire map phase lasts for 40s.

[1]We consider the conventional degraded read approach, which always reads from any k surviving nodes. Some special erasure code constructions have been proposed (e.g., [20, 22, 29]) to reduce the number of blocks read. Our work also applies to such erasure code constructions.



(a) locality-first scheduling



(b) degraded-first scheduling

Figure 3. Map-slot activities in the entire map phase.

The major issue of locality-first scheduling in erasure-coded storage in failure mode is that degraded tasks start degraded reads together and compete for the network resources to download blocks from other racks. Obviously, the competition significantly increases the overall duration of degraded tasks.

From the example, we observe that at the earlier stage of the map phase, while local tasks are being processed, network resources are not fully utilized. Thus, it is natural to *move the launch of some degraded tasks ahead to take advantage of the unused network resources*, so as to relieve the competition for network resources among degraded tasks later. Let us revisit the example, and suppose that we move ahead two degraded tasks for processing $B_{0,0}$ and $B_{2,0}$ to the beginning of the map phase. Figure 3(b) shows the new map-slot activities for the revised task scheduling scheme, which we call *degraded-first scheduling*. This eliminates the competition for network resources, and reduces the duration of the map phase from 40s to 30s, i.e., a 25% saving.

IV. Design of Degraded-First Scheduling

We present the design of degraded-first scheduling, whose main idea is to move part of degraded tasks to the earlier stage of the map phase. The advantages are two-fold. First, the degraded tasks can take advantage of the unused network resources while the local tasks are running. Second, we avoid the network resource competition among degraded tasks at the end of the map phase. In this section, we first present the basic version of degraded-first scheduling. We then conduct mathematical analysis to show the improvement of degraded-first scheduling over the default locality-first scheduling in Hadoop. Finally, we present the enhanced version of degraded-first scheduling that takes into account the topological configuration of the cluster.

A. Basic Design

Our primary design goal is to *evenly* spread the launch of degraded tasks among the whole map phase. This design goal follows two intuitions.

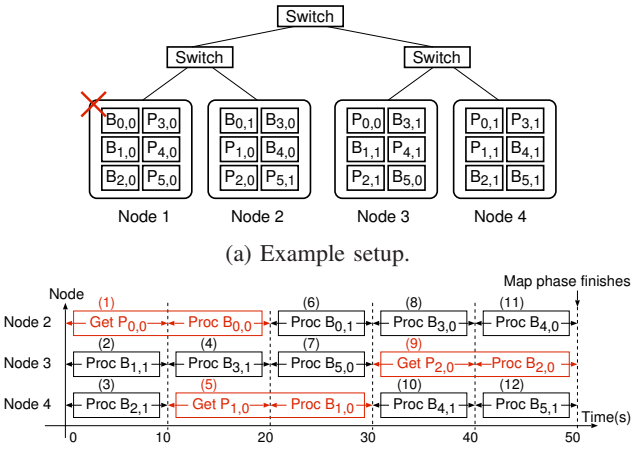
- *Finish running all degraded tasks before all local tasks.* If some degraded tasks are not yet finished after all local tasks are finished, they will be launched together and compete for network resources for degraded reads.
- *Keep degraded tasks separate.* If two or more degraded tasks run almost at the same time, they may compete for network resources for degraded reads.

The key challenge here is how to determine the right timing for launching degraded tasks, so that they are evenly spread among the whole map phase. One possible solution is to predict the overall running time of the whole map phase and launch degraded tasks evenly within the predicted time interval. However, this approach is difficult to realize for two reasons. First, different MapReduce jobs may have highly varying processing time of a map task. Thus, it is difficult to accurately predict how long the whole map phase would be. Second, even if we can make accurate predictions, it is possible that no free map slots are available when a degraded task is ready to launch. Thus, the launch of some degraded tasks may be delayed, defeating the original purpose of evenly spreading the degraded tasks.

Therefore, we propose a heuristic design that arranges the launch of degraded tasks with respect to the proportion of map tasks that have been launched. Algorithm 2 shows the pseudo-code of the basic version of degraded-first scheduling, which extends the default Algorithm 1. Given a MapReduce job, we first determine the total number of all map tasks to be launched (denoted by M) and the total number of degraded tasks to be launched (denoted by M_d). We also monitor the number of all map tasks that have been launched and that of degraded tasks that have been launched, denoted by m and m_d, respectively. Algorithm 2 launches a degraded task with a higher priority than a local task if the proportion of degraded tasks that have been launched is *no more than* the proportion of all map tasks that have been launched. In this way, we control the pace of launching degraded tasks and have them launched evenly in the whole map phase. It is worth noting that we assign at most one degraded task for every heartbeat (see the if-condition in Line 4), since launching more than one degraded task at the same node will lead to competition for network resources among these degraded tasks.

Algorithm 2 Basic Degraded-First Scheduling

```
1: while a heartbeat comes from slave s do
2:     isDegradedTaskAssigned = false
3:     for each running job j in the job list do
4:         if isDegradedTaskAssigned == false and
             s has a free map slot then
5:             if j has an unassigned degraded task then
6:                 if m/M ≥ m_d/M_d then
7:                     assign a degraded task to s
8:                     isDegradedTaskAssigned = true
9:                 end if
10:            end if
11:        end if
12:        for each free map slot on slave s do
13:            if j has an unassigned local task then
14:                assign the local task to s
15:            else if j has an unassigned remote task then
16:                assign the remote task to s
17:            end if
18:        end for
19:    end for
20: end while
```



(a) Example setup.

(b) Map slot activities of surviving nodes.

Figure 4. Example of the execution flow of the map phase based on the basic version of degraded-first scheduling (Algorithm 2). Each number in the brackets represents the order of the blocks being assigned a map slot.

We elaborate via an example how Algorithm 2 works. Figure 4(a) illustrates a cluster that contains four slaves and provides fault tolerance via a (4,2) erasure code. The processed file has a total of 12 native blocks, with three native blocks stored in each node. We fix both the downloading and processing times to be 10s as in Section III. To better describe the main idea, we configure each node with one map slot only. Suppose now Node 1 fails. Thus, there are a total of 12 map tasks, and three of them are degraded tasks for processing the lost blocks $B_{0,0}$, $B_{1,0}$ and $B_{2,0}$. We assume that the master assigns map tasks in the order Nodes 2, 3, and 4. Figure 4(b) shows the execution flow of the whole map phase. According to Algorithm 2, the three degraded tasks are assigned as the 1st, 5th, and 9th map tasks, and the three degraded tasks are launched at 0s, 10s, and 30s, respectively. We see that all the degraded tasks do not compete for network resources for downloading the parity blocks during degraded reads.

It is important to note that in normal mode (i.e., without any node failure), there is no degraded task, and hence degraded-first scheduling operates the same way as locality-first scheduling (see Lines 12-18 of Algorithm 2).

To summarize, Algorithm 2 launches a degraded task with a higher priority if appropriate. This is why we call the algorithm "degraded-first scheduling".

B. Analysis

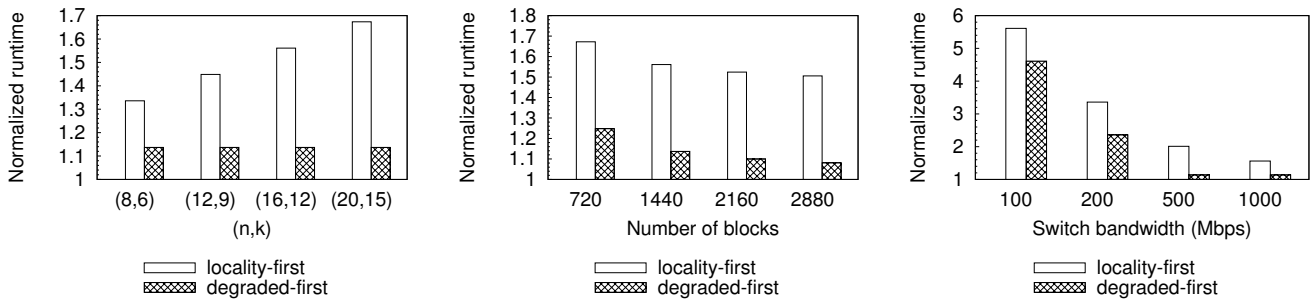
We conduct simple mathematical analysis to compare the default locality-first scheduling and our basic degraded-first scheduling in terms of the runtime of a MapReduce job. Our goal is to provide preliminary insights into the potential improvement of degraded-first scheduling in failure mode.

Analysis setting. We first describe the setting of our analysis. We consider a cluster with N homogeneous nodes that are evenly grouped into R racks (with $\frac{N}{R}$ nodes each). Let L be the number of map slots allocated for each node (i.e., it can run at most L map tasks simultaneously). Let T be the processing time of a map task. Let S be the input block size. Let W be the download bandwidth of each rack. For fault tolerance, we use an (n, k) erasure code to encode k native blocks to generate $n-k$ parity blocks. We distribute the stripes of n native and parity blocks evenly among the N nodes (as in parity declustering [19]). Let F be the total number of native blocks to be processed by MapReduce (i.e., the number of blocks in the each node is $\frac{F}{N}$).

Suppose that we only focus on the map-only MapReduce job, and neglect the shuffle overhead and the time for reduce tasks. Then in normal mode (without any node failure), the runtime of a MapReduce job is $\frac{FT}{NL}$.

We now consider the MapReduce performance in failure mode. We focus on the case where a single node fails (see our justification in Section II). This implies that there are a total of F map tasks, among which $\frac{F}{N}$ are degraded tasks. We assume that the degraded tasks are evenly distributed among all racks, such that each rack contains $\frac{F}{NR}$ degraded tasks (assuming $\frac{F}{NR}$ is an integer). When a degraded task issues a degraded read to a lost block, it downloads k out of the $n-1$ surviving blocks of the same stripe. We assume that the degraded read is bottlenecked by inter-rack traffic. If the stripes are evenly distributed across all racks and each degraded task randomly picks k out of $n-1$ blocks to download, then the expected time needed for downloading blocks from other racks for each lost block being reconstructed is $\frac{(R-1)kS}{RW}$.

Locality-first scheduling. We first consider the default locality-first scheduling. In failure mode, all degraded tasks are launched after the completion of processing all local



(a) Erasure coding scheme (b) Numbers of native blocks (c) Rack download bandwidth

Figure 5. Numerical analysis results of locality-first scheduling and degraded-first scheduling.

tasks (which takes time $\frac{FT}{NL}$). Each rack is assumed to have $\frac{F}{NR}$ degraded tasks, so the total time required for all degraded reads is $\frac{F}{NR} \cdot \frac{(R-1)kS}{RW}$. We further assume that the $\frac{F}{N}$ blocks of the degraded tasks can be processed in parallel by all available map slots in the cluster in a single map-slot duration T. Thus, by summing up the times for local tasks and degraded tasks, the MapReduce runtime under locality-first scheduling is:

$$\frac{FT}{NL} + \frac{F}{NR} \cdot \frac{(R-1)kS}{RW} + T.$$

Degraded-first scheduling. We next consider degraded-first scheduling, which spreads the launch of degraded tasks over the whole map phase. Here, we assume that the map tasks are launched in a *lock-step* manner, such that the map phase is divided into *rounds*, each of which launches the same number of map tasks given all available slots. Since there are a total of F map tasks and $(N-1)L$ available map slots in failure mode, there are approximately $\frac{F}{(N-1)L}$ rounds of launching the map tasks.

We need to address two cases in our analysis: (1) the degraded tasks can finish degraded reads in one round and (2) the degraded tasks need to finish degraded reads in more than one round. In the first case, it implies that all degraded reads can be finished within $\frac{F}{(N-1)L}$ rounds, each of which spans a slot duration T. Assuming that all degraded tasks are further processed in parallel within a map-slot duration T (as in above), the upper bound of the runtime is $\frac{FT}{(N-1)L} + T$. In the second case, the bottleneck is the inter-rack data transmission of degraded reads. Since each rack has $\frac{F}{NR}$ degraded tasks, it needs to spend the download time $\frac{F}{NR} \times \frac{(R-1)kS}{RW}$. An additional time T is needed to process the degraded tasks. While the racks are downloading data during degraded reads, the local map tasks can be processed in parallel. Thus, the runtime of the second case should be $\frac{F}{NR} \cdot \frac{(R-1)kS}{RW} + T$. Combining both cases, the MapReduce runtime under degraded-first scheduling is:

$$\max\left(\frac{FT}{(N-1)L} + T,\ \frac{F}{NR} \cdot \frac{(R-1)kS}{RW} + T\right).$$

Numerical results. Based on our analysis, we now present the MapReduce runtime results under different parameter settings. The default cluster configurations are fixed as: $N = 40$, $R = 4$, $L = 4$, $S = 128$MB, $W = 1$Gbps, $T = 20$s, $F = 1440$, and $(n, k) = (16, 12)$. We then vary one of the parameters and compare the MapReduce runtimes under locality-first scheduling and degraded-first scheduling. The runtimes are normalized over that in normal mode (without node failures).

Figure 5(a) shows the runtime results for different erasure coding schemes (8,6), (12,9), (16,12), and (20,15). Degraded-first scheduling always requires less runtime than locality-first scheduling, with the runtime reduction ranging from 15% to 32%. The runtime of degraded-first scheduling stays the same since all degraded tasks can finish their degraded reads in one round in our parameter settings. On the other hand, the runtime of locality-first scheduling increases with k, since more data needs to be transmitted for degraded reads after all local tasks are completed.

Figure 5(b) shows the runtime results versus the number of blocks F, varied from 720 to 2880. The normalized runtimes of both scheduling algorithms decrease with F, since the processing time of local tasks becomes more dominant. Nevertheless, the runtime of degraded-first scheduling is less than that of locality-first scheduling by 25% to 28%.

Finally, Figure 5(c) shows the runtime results versus the download bandwidth W, varied from 100Mbps to 1Gbps. As the download bandwidth increases, both scheduling algorithms see reduced runtime as degraded reads takes less time to finish. It is worth noting that degraded-first scheduling has the same runtime for $W = 500$Mbps and $W = 1$Gbps, since the degraded tasks now can finish degraded reads in one round. Overall, degraded-first scheduling reduces the runtime of locality-first scheduling by 18% to 43%.

To summarize, in all cases, degraded-first scheduling can reduce the MapReduce runtime of the default locality-first scheduling in failure mode. Note that our analysis builds on simplified settings. We resort to simulations for more general scenarios (see Section V).

C. Enhanced Design

We describe two heuristics that further enhance the performance of our basic degraded-first scheduling implementation in Algorithm 2. Both heuristics take into account the topology of the cluster when we schedule a set of map tasks across the slaves.

Locality preservation. The default locality-first scheduling achieves high locality by first launching local tasks whenever they are available. On the other hand, Algorithm 2 may break the locality. Specifically, if we first assign degraded tasks to a node, then the node may not have enough map slots to process its local tasks. The master may instead assign some of the local tasks of the node to other nodes of different racks, and these tasks become remote tasks. Having additional remote tasks is clearly undesirable as they compete for network resources as degraded tasks do.

We implement locality preservation by restricting the launch of degraded tasks, such that we prevent the local map tasks from being unexpectedly assigned to other nodes. We provide a function AssignToSlave to determine whether to launch a degraded task to a specific slave. Specifically, given a set of unassigned map tasks to be scheduled, we estimate the processing time for the local map tasks of each slave s (denoted by t_s), and the expected processing time $E[t_s]$ for the local map tasks across all slaves. If $t_s > E[t_s]$, it means that slave s does not have spare resources to process a degraded task, so we do not assign a degraded task to it.

We point out that our implementation also works for *heterogeneous* settings, where some slaves may have better processing power than others in the same cluster. If we estimate the processing time for the local map tasks based on not only the number of local map tasks, but also the computing power of the slave node, then we allow the more powerful slaves to process a degraded task while processing more local map tasks.

Rack awareness. In failure mode, launching multiple degraded tasks in the same rack may result in competition for network resources, since the degraded tasks download data through the same top-of-rack switch. However, Algorithm 2 is oblivious to where a degraded task is launched.

To realize rack awareness, we provide a function AssignToRack to ensure that multiple degraded tasks are not assigned to the same rack at nearly the same time. Specifically, we keep track of the duration since the last degraded task is assigned to each rack r (denoted by t_r), and the expected duration $E[t_r]$ across all racks. We avoid assigning a degraded task to a slave in rack r if r satisfies both of the following conditions: (1) if $t_r < E[t_r]$, and (2) if t_r is less than some threshold. For the latter condition, we choose the threshold as $\frac{(R-1)kS}{RW}$, which is the expected time for a degraded read (see Section IV-B). Satisfying both conditions imply that rack r has just recently launched a degraded task that is still performing a degraded read. If we launch another degraded task to rack r, it will lead to unnecessary competition for network resources.

Algorithm 3 Enhanced Degraded-First Scheduling

```
1: function AssignToSlave(slave s)
2:     if t_s < E[t_s] then
3:         return false
4:     end if
5:     return true
6: end function

7: function AssignToRack(rack r)
8:     if t_r < min(E[t_r], threshold) then
9:         return false
10:    end if
11:    return true
12: end function

13: procedure Main Algorithm
14:    while a heartbeat comes from slave s do
15:        isDegradedTaskAssigned = false
16:        Compute t_s, E[t_s], t_r, E[t_r]
17:        for each running job j in the job list do
18:            if isDegradedTaskAssigned = false and
                 s has a free map slot then
19:                if j has an unassigned degraded task then
20:                    if m/M ≥ m_d/M_d and
                         AssignToSlave(s) == true and
                         AssignToRack(rackID(s)) == true then
21:                        assign a degraded task to s
22:                        isDegradedTaskAssigned = true
23:                    end if
24:                end if
25:            end if
26:            for each free map slot on slave s do
27:                if j has an unassigned local task then
28:                    assign the local task to s
29:                else if j has an unassigned remote task then
30:                    assign the remote task to s
31:                end if
32:            end for
33:        end for
34:    end while
35: end procedure
```

Putting it all together. Algorithm 3 shows the enhanced version of degraded-first scheduling, which includes locality preservation and rack awareness through the functions AssignToSlave and AssignToRack, respectively.

V. Simulation

We present simulation results and examine the performance improvement of degraded-first scheduling in general scenarios. We implement a discrete event simulator for MapReduce (see Section V-A). We first compare enhanced degraded-first scheduling (`EDF`), which incorporates locality preservation and rack awareness, with the default locality-first scheduling (`LF`) (see Section V-B). We then compare the basic and enhanced versions of degraded-first scheduling (`BDF` and `EDF`, respectively), and show how locality preser-

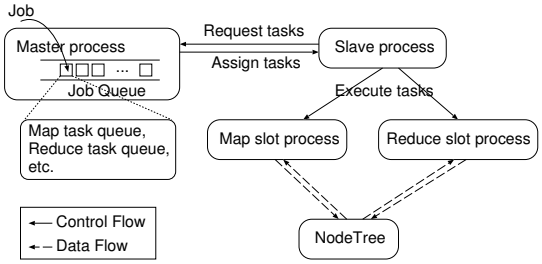


Figure 6. The simulator architecture.

vation and rack awareness improve MapReduce performance in both general and extreme cases (see Section V-C).

A. Simulator Overview

Our MapReduce simulator is a C++-based discrete event simulator built on CSIM20 [8]. Figure 6 illustrates the simulator architecture. We deploy *processes* to simulate different components of a MapReduce system. There are two types of processes: (1) node processes, which include both master and slave processes, and (2) slot processes, each of which manages either a map or reduce slot. We also implement a *NodeTree* structure, which simulates a storage cluster with two levels of switches (see Figure 2) and handles all intra-rack and inter-rack transmission requests.

The flow of the simulator is as follows. Each slave process periodically sends heartbeats to the master process (every 3s) and indicates in each heartbeat if it has any free slots. When a MapReduce job is submitted, the master process initializes the job by splitting the job into map and reduce tasks, and enqueues the initialized job into a job queue. The map and reduce tasks will later be assigned to slave processes via the responses to their periodical heartbeats. If a process needs to transmit blocks across racks (e.g., due to shuffle-and-sort and degraded reads), then it notifies the NodeTree structure to hold the communication link for a duration needed for the data transmission.

We can configure our simulator with different parameters, such as the number of nodes, number of racks, number of map/reduce slots per node, erasure coding scheme, scheduling scheme, etc. We can also set a heterogeneous cluster in which nodes have different processing capabilities. We can also simulate multiple MapReduce jobs with different numbers of map/reduce tasks and amounts of data being shuffled between map and reduce tasks. The master maintains a job queue for all jobs in first-in-first-out (FIFO) order, as the default MapReduce implementation in Hadoop (version 0.22.0).

B. Locality-First vs. Degraded-First Scheduling

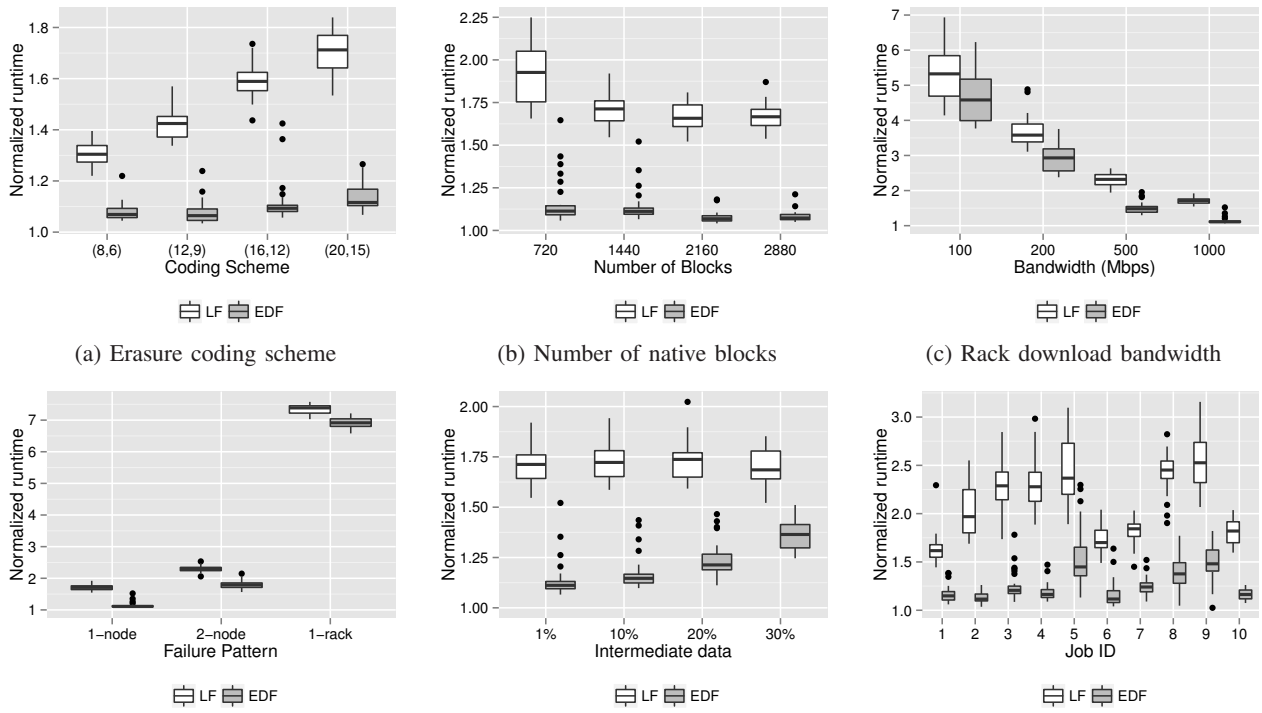
We start with considering a homogeneous cluster with the following default configurations. It contains 40 nodes evenly grouped into four racks (with 10 nodes each). The rack download bandwidth is 1Gbps. The block size is 128MB. We use (20,15) erasure codes. In contrast with the map-only MapReduce job considered in Section IV-B, we here consider a MapReduce job with both map and reduce tasks, whose processing times follow normal distributions with mean 20s and standard deviation 1s, and mean 30s and standard deviation 2s, respectively. We allocate each node with four map slots and one reduce slot. We create 1440 blocks in total, and randomly place them in the nodes based on the requirements in Section III. The total number of map tasks is equal to the total number of blocks (i.e., 1440), while the number of reduce tasks is fixed at 30. We assume that each map task shuffles intermediate data of size 1% of the block size to the reduce tasks. We simulate the single-node failure mode by randomly disabling one of the nodes.

In each simulation experiment, we vary one of the configuration parameters and evaluate the impact on MapReduce performance. For each set of parameters, we generate 30 cluster configurations with different random seeds. In each configuration, we measure the MapReduce runtime of each of `LF` and `EDF` in failure mode, and also measure the MapReduce runtime in normal mode without node failures for reference. We compute the *normalized runtime* of each of `LF` and `EDF` over that of normal mode. We plot a *boxplot* to show the minimum, lower quartile, median, upper quartile, and maximum of the 30 results and any outlier. Figure 7 shows the results, which we elaborate below.

Figures 7(a)-(c) show the normalized runtime results versus the erasure coding parameters (n, k), the number of native blocks to be processed, and the download bandwidth, respectively. The results conform to our analysis findings in Section IV-B. We briefly summarize the results here. First, from Figure 7(a), if we use an erasure coding scheme with larger (n, k), `EDF` reduces the normalized runtime of `LF` by a larger margin, ranging from 17.4% for (8,6) to 32.9% for (20,15). Second, from Figure 7(b), as the number of native blocks to be processed increases, the reduction of `EDF` over `LF` drops, but `EDF` still reduces the normalized runtime by 34.8% to 39.6%. Finally, from Figure 7(c), the normalized runtimes of both `EDF` and `LF` increase, while `EDF` reduces the normalized runtime of `LF` by up to 35.1% on average when the rack download bandwidth is 500Mbps.

We consider additional scenarios. Figure 7(d) shows the normalized runtime results under different failure patterns, including a single-node failure, a double-node failure, and a rack failure. As more nodes fail, the normalized runtimes of both scheduling schemes increase. `EDF` reduces the normalized runtime of `LF` by 33.2%, 22.3%, and 5.9% on average for a single-node failure, a double-node failure, and a rack failure, respectively.

Figure 7(e) shows the normalized runtime results versus the amount of intermediate data shuffled by map tasks. The amount of intermediate data is configured as 1% to 30% of that of the data processed by the map tasks. `LF` remains unaffected in general since the degraded tasks run at the end



(a) Erasure coding scheme (b) Number of native blocks (c) Rack download bandwidth

(d) Failure pattern (e) Amount of intermediate data (f) Multi-job scenario

Figure 7. Comparisons of normalized runtimes of LF and EDF versus different parameters.

of the map phase and they only compete with the shuffled data generated by themselves, while EDF has increasing normalized runtime as its degraded tasks compete with the shuffled data generated by the currently running local map tasks. Nevertheless, EDF still reduces the normalized runtime of LF by 20.0% to 33.2%.

We thus far focus on a single MapReduce jobs, and we now evaluate LF and EDF when multiple MapReduce jobs are simultaneously running. We generate 10 jobs, whose inter-arrival times follow an exponential distribution with mean 120s. The jobs are scheduled based on the default first-in-first-out (FIFO) scheduling to allocate map/reduce slots among different jobs. Figure 7(f) shows the normalized runtime of each of the 10 jobs. We see that EDF remains effective even in the multiple-job scenario, as it reduces the normalized runtime of LF by 28.6% to 48.6%.

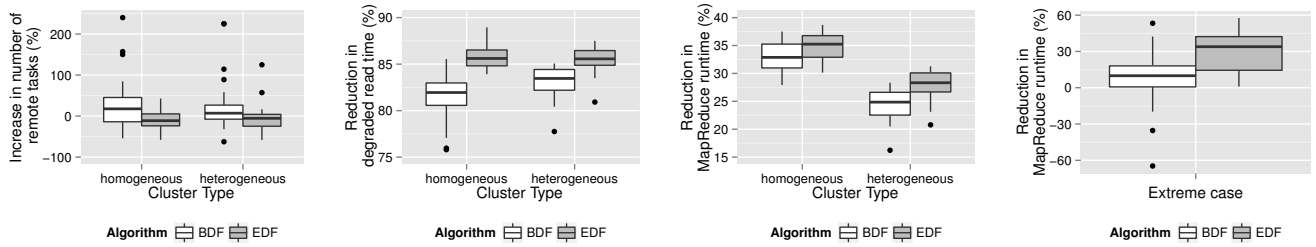
C. Basic vs. Enhanced Degraded-First Scheduling

We now compare BDF and EDF under various settings. We consider both homogeneous and heterogeneous clusters. In addition, we consider an extreme case where EDF significantly outperforms BDF.

We describe our simulation setup. The homogeneous cluster has the same default configuration as in Section V-B, while the heterogeneous cluster follows the same configuration as the homogeneous one except that half of the nodes have worse processing power with the mean processing times of the map and reduce tasks set to 40s and 60s, respectively. We compute different metrics of BDF and EDF, including the number of remote tasks launched, degraded read time (i.e., the time from issuing a degraded read request until k blocks are downloaded from surviving nodes), and MapReduce runtime, and compare the results with that of LF in failure mode (with a single node failure). We present boxplots of 30 samples for each setting.

Figure 8(a) first compares the percentage increase in the number of remote tasks of BDF and EDF compared to LF. BDF has more remote tasks than LF, by 35.4% and 25.4% on average for homogeneous and heterogeneous clusters, respectively. The reason is that BDF assigns degraded tasks earlier and has fewer slots for the originally local tasks in LF, some of which now become remote tasks. On the other hand, EDF has *fewer* remote tasks than LF by 10.7% and 6.7% on average for homogeneous and heterogeneous clusters, respectively. EDF assigns degraded tasks to the nodes that have low processing time for local tasks, and hence prevents these nodes from stealing local tasks from other nodes. Thus, the overall number of remote tasks drops.

Figure 8(b) shows the reduction of degraded read time of BDF and EDF compared to LF. BDF reduces the degraded read time by 80.5% and 83.1% on average for homogeneous and heterogeneous clusters, respectively, while EDF achieves the average reduction by 85.4% and 85.5%, respectively. EDF further reduces the degraded read time since it assigns



(a) Increase in number of remote tasks

(b) Reduction in degraded read time

(c) Reduction in MapReduce runtime

(d) Reduction in MapReduce runtime (extreme case)

Figure 8. Comparisons of BDF and EDF in homogeneous and heterogeneous clusters and in an extreme case.

degraded tasks to different racks to prevent them from competing for network resources.

Figure 8(c) shows the reduction of MapReduce runtime of BDF and EDF over LF. BDF achieves 32.3% and 24.4% of runtime saving in homogeneous and heterogeneous clusters, respectively, while the savings of EDF are 34.0% and 27.9%, respectively.

Although EDF does not significantly reduce MapReduce runtime compared to BDF, we find that the EDF remains robust even in an extreme case. We consider a cluster configuration that is the same as the homogeneous one except that five of the nodes are bad and have much worse processing power, such that the processing times of a local map task are 3s for regular nodes and 30s for the bad nodes. We run a map-only job over a file with 150 blocks stored in the cluster. We then compare BDF and EDF in failure mode, where one of the normal nodes fails. Figure 8(d) shows reduction of MapReduce runtime of BDF and EDF compared to LF over 30 runs in the extreme case. BDF only reduces the MapReduce runtime by 11.7% on average, while EDF can make an average reduction of 32.6%. We note that EDF has 36.1% fewer of remote tasks and 34.6% less of degraded read time on average than BDF (not shown in the figure). This shows the importance of locality preservation and rack awareness to make degraded-first scheduling robust in general and even extreme scenarios.

VI. Experiments

We implement degraded-first scheduling by modifying the source code of Hadoop 0.22.0. We run MapReduce on HDFS-RAID [18], which extends HDFS to support erasure-coded storage. We conduct testbed experiments and compare enhanced degraded-first scheduling (EDF) with Hadoop's default locality-first scheduling (LF).

We run experiments on a small-scale Hadoop cluster testbed composed of a single master node and 12 slave nodes. The 12 slaves are grouped into three racks with four slaves each. The slaves in the same rack are connected via a 1Gbps top-of-rack switch, and the top-of-rack switches are connected via a 1Gbps core switch. Each of the master and slave nodes runs Ubuntu 12.04 on an Intel Core i5-3570 3.40GHz quad-core CPU, 8GB RAM, and a Seagate ST1000DM003 7200RPM 1TB SATA disk.

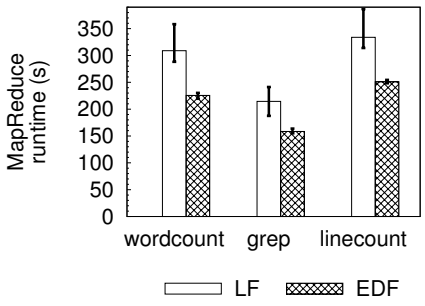
Our experiments consider three I/O-heavy MapReduce jobs, all of which run over a collection of text files.

- *WordCount*: It counts the occurrences of each word. The map tasks tokenize the words in text files and emit each word and its local count to the reduce tasks, which sum up the local counts for each word and write the results to HDFS.
- *Grep*: It searches for lines containing a given word. The map tasks scan through the text files and emit the lines containing the given word to the reduce tasks, which aggregate and write the lines to HDFS.
- *LineCount*: It counts the occurrences of each line. It works like WordCount, and shuffles more lines than Grep from the map tasks to the reduce tasks.

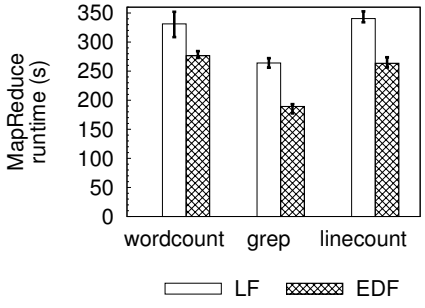
We configure the HDFS block size as 64MB and use a (12,10) erasure code to provide failure-tolerance. The blocks are placed in the slaves in a round-robin manner for load balancing. Each slave has four map slots and one reduce slot. We set the number of reduce tasks to eight for each MapReduce job. We then generate 15GB of plain text data from the Gutenberg website [17]. The data is divided into 240 blocks and written to HDFS. Then HDFS-RAID transforms the replicated data of HDFS into erasure-coded data. Then the 240 native blocks are evenly placed in the 12 slaves, each containing 20 blocks. We consider a single-node failure, which we simulate by erasing data in one randomly picked node and killing the slave daemon there.

We evaluate the MapReduce runtime, defined as the time interval between the launch of first map task and the completion of the last reduce task. The results are averaged over five runs.

We first consider a single-job scenario, in which we run each of the three jobs over the input data. Figure 9(a) shows the runtime of each job in a single job scenario. EDF reduces the MapReduce runtime of LF by 27.0%, 26.1% and 24.8% for WordCount, Grep, and LineCount, respectively. We observe that LF has a larger runtime variance than EDF, mainly because it does not consider rack-awareness. This causes the number of degraded tasks assigned to



(a) Single-job scenario



(b) Multi-job scenario

Figure 9. Comparisons of MapReduce runtime in single-job and multi-job scenarios. The minimum and maximum runtimes are also plotted as the endpoints of the vertical line in each bar.

different racks to become unbalanced, and hence increases the runtime variance.

We also consider a multi-job scenario, in which we submit the three MapReduce jobs in the order of WordCount, Grep, and LineCount in a short time so that the three jobs are scheduled by Hadoop in a first-in-first-out order. Figure 9(b) shows the runtime results. EDF can reduce the MapReduce runtime by 16.6%, 28.4%, and 22.6% for WordCount, Grep, and LineCount, respectively. We note that WordCount has the least runtime reduction. The reason is that EDF launches the degraded tasks of a job while the reduce tasks of the previous job are still downloading the intermediate data generated by the map tasks. This leads to competition for network resources and delays the completion of the previous job. Nevertheless, our results demonstrate the improvements of EDF over LF in both single-job and multi-job scenarios.

Table I provides a breakdown analysis of task runtime of different jobs in the single-job scenario. We compare the average runtime of normal map tasks (local and remote tasks), degraded tasks and reduce tasks. The runtime of a task is defined as the time interval between its launch and completion. It includes data transmission time (for remote and degraded tasks) as well as the data processing time. We see that EDF reduces the average runtime of degraded tasks compared to LF, by 43.0%, 34.6%, and 47.7% for WordCount, Grep, and LineCount, respectively. Since EDF reduces the average runtime of the overall map phase, the average runtime of reduce tasks is also reduced, by around 26%. The normal tasks have a similar average runtime in both LF and EDF, so EDF does not affect the processing of normal tasks.

VII. Related Work

There have been extensive empirical studies on examining the practical use of erasure coding in clustered storage systems (e.g., [1, 5, 10, 12, 13, 20, 22, 23, 25, 29, 33, 37]). DiskReduce [12] extends HDFS to encode replicated data with erasure coding offline. Zhang *et al.* [37] further implement an online encoding framework for HDFS and study various MapReduce workloads on erasure-coded HDFS. In addition, several studies focus on enhancing the degraded read performance in erasure-coded clustered storage systems under failure mode. Khan *et al.* [22] present an algorithm that minimizes disk I/Os for single failure recovery for arbitrary erasure codes. New erasure code constructions are proposed and evaluated on Azure [20] and HDFS [10, 23, 25, 29]. Our work complements them by designing a proper task scheduling algorithm to improve the MapReduce performance in failure mode.

Our work aims to enhance the baseline Hadoop MapReduce design, and this objective is also shared by previous work. For example, authors of [2, 14, 36] propose new task scheduling algorithms for heterogeneous clusters so as to prevent a MapReduce job from being delayed by stragglers. Authors of [31, 34, 35] propose fair task scheduling algorithms for MapReduce on multi-user clusters, and mitigate the resource starvation of small jobs in the presence of large jobs. Besides scheduling, some studies propose to modify the default block placement policy of HDFS, so as to improve data availability [7] and write performance [6]. Such HDFS/MapReduce enhancements focus on replication-based storage, while ours target erasure-coded storage.

VIII. Conclusions

This paper explores the feasibility of running data analytics in erasure-coded clustered storage systems. We present degraded-first scheduling, a new MapReduce scheduling scheme designed for improving MapReduce performance in erasure-coded clustered storage systems that run in failure mode. We show that the default locality-first scheduling launches degraded tasks at the end, thereby making them compete for network resources. Degraded-first scheduling launches degraded tasks earlier to take advantage of the unused network resources. We also propose heuristics that leverage topological information of the storage system to improve the robustness of degraded-first scheduling. We conduct simple mathematical analysis and discrete event simulation to show the performance gains of degraded-first scheduling. We further conduct testbed experiments in a Hadoop cluster, and show that degraded-first scheduling can reduce the MapReduce runtime of locality-first scheduling

Table I
AVERAGE RUNTIME (IN SECONDS) OF DIFFERENT TYPES OF TASKS IN THE SINGLE-JOB SCENARIO.

	Number of tasks	WordCount		Grep		LineCount	
		LF	EDF	LF	EDF	LF	EDF
Normal map	220	30.94	29.12	11.69	10.43	35.91	33.25
Degraded map	20	84.97	48.42	77.97	50.96	91.48	47.88
Reduce	8	247.90	182.05	161.08	122.60	273.70	199.35

by 27.0% for a single-job scenario and 28.4% for a multi-job scenario. The source code of degraded-first scheduling is available at **http://ansrlab.cse.cuhk.edu.hk/software/dfs**.

ACKNOWLEDGMENTS

This work was supported in part by grants AoE/E-02/08 and ECS CUHK419212 from the University Grants Committee of Hong Kong.

REFERENCES

[1] M. Abd-El-Malek, W. Courtright II, C. Cranor, G. Ganger, J. Hendricks, A. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. Sambasivan, et al. Ursa Minor: Versatile Cluster-based Storage. In *Proc. of USENIX FAST*, Dec 2005.

[2] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *Proc. of USENIX OSDI*, page 14, 2010.

[3] J. Bloemer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman. An XOR-Based Erasure-Resilient Coding Scheme. Technical Report TR-95-048, International Computer Science Institute, UC Berkeley, Aug 1995.

[4] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proc. of ACM SOSP*, Oct 2011.

[5] J. C. W. Chan, Q. Ding, P. P. C. Lee, and H. H. W. Chan. Parity Logging with Reserved Space: Towards Efficient Updates and Recovery in Erasure-coded Clustered Storage. In *Prof. of USENIX FAST*, 2014.

[6] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging Endpoint Flexibility in Data-Intensive Clusters. In *Proc. of ACM SIGCOMM*, 2013.

[7] A. Cidon, S. Rumble, R. Stutsman, S. Katti, J. Ousterhout, and M. Rosenblum. Copysets: Reducing the Frequency of Data Loss in Cloud Storage. In *Proc. of USENIX ATC*, 2013.

[8] CSIM. http://www.mesquite.com/products/csim20.htm.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. of USENIX OSDI*, Dec 2004.

[10] K. S. Esmaili, L. Pamies-Juarez, and A. Datta. CORE: Cross-Object Redundancy for Efcient Data Repair in Storage Systems. In *Proc. of IEEE BigData*, 2013.

[11] B. Fan, W. Tantisiriroj, and G. Gibson. Diskreduce: Replication as a prelude to erasure coding in data-intensive scalable computing. In *Carnegie Mellon Univsersity, Parallel Data Laboratory, Tech. Rep. Technical Report CMU-PDL-11-112*, 2011.

[12] B. Fan, W. Tantisiriroj, L. Xiao, and G. Gibson. DiskReduce: RAID for Data-Intensive Scalable Computing. In *Proc. of Annual Workshop on Petascale Data Storage (PDSW)*, Nov 2009.

[13] D. Ford, F. Labelle, F. I. Popovici, M. Stokel, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in Globally Distributed Storage Systems. In *Proc. of USENIX OSDI*, Oct 2010.

[14] R. Gandhi, D. Xie, and Y. C. Hu. PIKACHU: How to Rebalance Load in Optimizing MapReduce On Heterogeneous Clusters. In *Proc. of USENIX ATC*, 2013.

[15] J. Gantz and D. Reinsel. Extracting Value from Chaos. http://www.emc.com/digital_universe, Jun 2011.

[16] S. Ghemawat, H. Gobioff, and S. Leung. The Google File System. In *Proc. of ACM SOSP*, Dec 2003.

[17] Gutenberg. http://www.gutenberg.org.

[18] HDFS-RAID. http://wiki.apache.org/hadoop/HDFS-RAID.

[19] M. Holland, G. A. Gibson, and D. P. Siewiorek. Architectures and algorithms for on-line failure recovery in redundant disk arrays. *Distrib. Parallel Databases*, 2(3):295–335, July 1994.

[20] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure Coding in Windows Azure Storage. In *Proc. of USENIX ATC*, Jun 2012.

[21] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Proc. of ACM EuroSys*, Jun 2007.

[22] O. Khan, R. Burns, J. Plank, W. Pierce, and C. Huang. Rethinking Erasure Codes for Cloud File Systems: Minimizing I/O for Recovery and Degraded Reads. In *Proc. of USENIX FAST*, Feb 2012.

[23] R. Li, J. Lin, and P. P. C. Lee. CORE: Augmenting Regenerating-Coding-Based Recovery for Single and Concurrent Failures in Distributed Storage Systems. In *Proc. of IEEE MSST*, May 2013.

[24] D. T. Meyer, M. Shamma, J. Wires, Q. Zhang, N. C. Hutchinson, and A. Warfield. Fast and Cautious Evolution of Cloud Storage. In *Proc. of USENIX HotStorage*, Jun 2010.

[25] D. Papailiopoulos, J. Luo, A. Dimakis, C. Huang, and J. Li. Simple Regenerating Codes: Network Coding for Cloud Storage. In *Proc. of IEEE INFOCOM*, Mar 2012.

[26] J. S. Plank, K. M. Greenan, and E. L. Miller. Screaming fast Galois Field arithmetic using Intel SIMD instructions. In *Proc. of USENIX FAST*, Feb 2013.

[27] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. OHearn. A performance evaluation and examination of open-source erasure coding libraries for storage. In *Proc. of USENIX FAST*, 2009.

[28] I. Reed and G. Solomon. Polynomial Codes over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[29] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. Xoring Elephants: Novel Erasure Codes for Big Data. In *Proc. of VLDB Endowment*, pages 325–336, 2013.

[30] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proc. of IEEE MSST*, May 2010.

[31] J. Tan, X. Meng, and L. Zhang. Delay Tails in MapReduce Scheduling. In *Proc. of ACM SIGMETRICS*, Jun 2012.

[32] H. Weatherspoon and J. D. Kubiatowicz. Erasure Coding Vs. Replication: A Quantitative Comparison. In *Proc. of IPTPS*, Mar 2002.

[33] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable Performance of the Panasas Parallel File System. In *Proc. of USENIX FAST*, Feb 2008.

[34] J. Wolf, D. Rajan, K. Hildrum, R. Khandekar, V. Kumar, S. Parekh, K.-L. Wu, and A. Balmin. FLEX: A Slot Allocation Scheduling Optimizer for MapReduce Workloads. In *Middleware 2010*, pages 1–20. Springer, 2010.

[35] 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 ACM EuroSys*, pages 265–278, 2010.

[36] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proc. of USENIX OSDI*, 2008.

[37] Z. Zhang, A. Deshpande, X. Ma, E. Thereska, and D. Narayanan. Does Erasure Coding Have a Role to Play in my Data Center? Technical Report MSR-TR-2010-52, Microsoft Research, May 2010.