Dominoes: Speculative Repair in Erasure-coded Hadoop System

Xi Yang*, Chen Feng*†‡, Zhiwei Xu† and Xian-He Sun*
*Department of Computer Science, Illinois Institute of Technology, Chicago, USA
Email: xyang34@hawk.iit.edu, sun@iit.edu
† SKL Computer Architecture, ICT, CAS, China
Email: {fengchen, zxu}@ict.ac.cn
‡University of Chinese Academy of Sciences, Beijing, China

***Abstract*—Data volume grows dramatically in the era of big data. To save capital cost on storage hardware, datacenters currently prefer using erasure coding rather than simply replication to resist data loss. Erasure coding can provide equivalent three-way fault tolerance to HDFS's default three replication mechanism but degrades data availability for task scheduling. In an erasure-coded system, data reconstruction time will be paid while tasks access the missing blocks during MapReduce job processing. Tasks' accessing corrupt data introduces task stragglers and degrades resource utilization. To overcome these challenges, we propose a novel mechanism, Dominoes, that coordinates lightweight data states checking and job scheduling to hide such recovery penalty during job processing and enhances job throughputs. The experimental results confirm Dominoes' effectiveness and efficiency that improves job throughput by 9% to 9.7% under failure at an overhead of 2.6% for failure-free jobs.**

***Keywords*-Hadoop; HDFS; erasure-coded system**

I. Introduction

In today's datacenters, data volume grows rapidly. Typically, in Facebook production cluster, 500TB to 900TB new data is being generated every day [1]. MapReduce programming paradigm is increasingly popular due to its transparent parallelism and efficient processing of huge datasets with high scalability [2]. MapReduce file systems, such as Google File System (GFS) [3] and Hadoop Distributed File System (HDFS) [4], have been proposed to use replications for failure tolerance in commodity clusters a decade ago. Such default three replication consumes redundant two-time storage space. Therefore, its resulting capital cost becomes considerable.

To this end, the erasure coding algorithms are widely used in latest production datacenters, such as Google ColossusFS (the successor to GFS) [5], Facebook HDFS [6] [7], and Microsoft Azure Storage system [8]. Erasure coding techniques significantly reduce the storage cost from 3X down to as low as 1.4X. Compare to the conventional triplication fault tolerance strategy, the erasure coding techniques can at least provide equivalent three-way reliability but at a much lower storage cost. Its reliability feature is an important supplement to hardware RAID techniques that erasure coding technique is adopted by General Parallel File System (GPFS) [9] [10] in High-Performance Computing centers. In erasure-coded systems, several blocks are assigned to a failure group and parity blocks are generated by computing raw data blocks. A "corrupt" block in this paper indicates a block is corrupt or missing, which is consistent with the description for unhealthy block states in HDFS. A job or a task is "flawed" if it processes any corrupt block. The system recovers the corrupt block by computing the corresponding raw blocks and parity blocks.

However, erasure-coded systems trade data availability for low storage cost, thus introducing drawbacks. Block recovery mechanisms can be categorized into two classes based on the granularity of data recovery workload. One is to repair block while accessing failure occurs; the other one is similar to conventional HDFS re-replication job that to recover all corrupt blocks system-wide. A routine system-wide data recovery job is necessary to repair corrupt data in time to avoid permanent data loss. The conventional HDFS re-replication mechanism only needs to replicate the missing block from its replications whereas an erasure-coded HDFS system requires multiple raw blocks and parity blocks to recover the unhealthy block. It costs considerable network traffic and I/O bandwidths for job reconstruction. Furthermore, to achieve a lower cost of data durability, the erasure coding strategy inevitably hurts data availability during task scheduling. This is because MapReduce prefers to schedule tasks onto the datanodes where the target data reside and such single raw data narrows the options of the data source for tasks. Hence, to avoid entire job failure due to the inaccessible target data blocks, a data recovery job must be conducted prior to the corresponding map task processing the data blocks. Furthermore, task stragglers due to inaccessible data blocks hurt performance and system utilization.

Lots of efforts have been contributed to reducing read latency while target blocks are inaccessible [8], improving performance of data reconstruction job [7] [11], reducing data reconstruction traffic [8] [12], and further reducing the storage cost of erasure coding [1]. However, few efforts consider scheduling MapReduce jobs, tasks, target data states and block recovery job coordinately.

This paper revisits job and task processing in erasure-coded Hadoop system and reveals the potentials to hide the data block reconstruction latency. We propose Dominoes to improve both job throughput and system efficiency. The contributions of this paper are threefold:

- We propose Dominoes that detects corrupt blocks on demand, and schedules jobs and map tasks in a speculative manner. In this way, Dominoes automatically detects and recovers corrupt blocks, and hides the penalty of data recovery.
- Dominoes simultaneously considers both job fairness and system job throughput. Besides, it adopts a loosely-coupled software design that introduces low overhead to the existing system.
- A prototype system has been built based on Facebook erasure-coded Hadoop system. To investigate the design trade-off, two scheduling strategies, FixBeforeJob and FixInMap, are implemented. The evaluation results show that Dominoes has an 11.5% to 48.5% and 9.1% to 9.7% performance improvement, respectively, compared with the other two strategies. Dominoes introduces a 2.6% overhead to failure-free workloads.

The rest of this paper is organized as follow: firstly, we present the background and motivation in section II. Next, we illustrate the design in Section III and implementation in Section IV respectively. We then evaluate the performance of Dominoes in Section V, discuss related work in Section VI and conclude in Section VII.

II. Background and Motivation

A. Routine Data Integrity Checking in Hadoop Framework

Apache Hadoop is the most popular open-source MapReduce implementation, which consists of Hadoop MapReduce and HDFS. MapReduce system adopts master/slave architecture, in which one node acts as the master and rest of nodes serve as slaves. Recently introduced the next generation Hadoop framework, Apache YARN [13], separates the role of centralized job manager into resource management and application master per job, but still inherently master/slaves architecture. It has the limitations on jobs scheduling [14] and data management. Particularly, the master/slaves architecture in HDFS is the same for both Hadoop and YARN systems: one master node, the namenode, manages all namespace and metadata information in a centralized manner whereas the rest of nodes, the datanodes, monitor their hosting blocks and provide I/O services to clients.

The namenode manages namespace information and handles all data retrieval operations. It mainly maintains three types of metadata information: file to blocks, block to locations, and machine to its hosting blocks. All metadata is kept in memory, but the first one will be checkpointed to disk. The latter two kinds of information can be dynamically rebuilt from reports of datanodes. Datanode maintains a block list that contains the blocks stored in its local disks, which is kept in memory.

The namenode detects datanodes health status via heartbeats but repair corrupt blocks in a lazy manner. Datanodes send routine heartbeats to the namenode every three seconds. The namenode acknowledges the status of individual datanode. By default, if namenode has not received a heartbeat from a datanode for more than 300 seconds, it will determine the datanode as a dead node. However, the blocks will not be recovered immediately, the system will lazily launch a system-wide data reconstruction job instead. Each datanode will scan data directories and reconcile the differences between blocks in memory and on the disk every 21,600 seconds (six hours). Also, datanode will report to namenode its block list every 21,600,000 milliseconds (six hours) by default [15]. Datanode can periodically scan its hosting blocks, but this service is considered costly and thus off by default. Data integrity is ensured during data transmission. Datanodes provide I/O services transparently to clients. A file will be chunked up into fixed-size blocks; blocks in a file are distributed across datanodes in the cluster. Data transmissions between datanodes are conducted in pipeline via packages, acknowledgment and checksum techniques are adopted to guarantee data integrity during the procedure.

Unfortunately, long detection interval, centralized namespace management and the lazy recovery mechanism result in data state inconsistency issues between namenode and actual data block status on datanode. Corrupt data block detection and block recovery are not the main concern, and the interval is long in conventional HDFS system. It is because replication provides multiple data sources for task processing. Once one block is inaccessible, the failure-redo mechanism will launch another attempt task to access the data from other equivalent replicas. Moreover, in an erasure-coded system, an encoded data block becomes a single copy, and data repair is needed once target block is inaccessible.

B. Corrupt Data Detection and Recovery in Erasure-coded HDFS System

This paper is based on Facebook erasure-coded HDFS system (HDFS-RAID) [16]. We review the capability of corrupt data detection and data recovery in the system.

Once the data have been written to the HDFS, they are stored in three-way replications as the conventional HDFS does. RaidNode is a new add-on master process that in charge of encoding data blocks and launching data recovery tasks. The system encodes the data blocks and generates parity blocks after a configured interval (usually one hour). The replications then will be erased after encoding, and only one copy set of raw data blocks will be kept in the system.

HDFS-RAID provides XOR and Reed-Solomon (RS) erasure coding techniques. The system can be configured with either technique. Besides the conventional data scanning and checking policy, this system provides a mas-

Table I
TIME GAP BETWEEN BLOCK CORRUPTION AND RECOVERY UNDER ROUTINE DETECTION

Number of Total File	Interval (Seconds)	
	Block size = 1MB	Block size = 64MB
1	7.19	42.577
2	18.308	388.237
4	26.335	277.686
8	53.585	70.631
16	58.584	1373.223
32	178.162	147.946
64	591.248	2360.541

Table II
BLOCK RECONSTRUCTION TIME

Block Size (MB)	Reconstruction Time(Seconds)
1	0.866
32	5.072
64	9.056
128	16.995
256	30.034

ter/slaves architecture to detect corrupt data blocks and launch recovery tasks. Both namenode and datanodes have responsibilities for checking data states. Namenode is the master that collects the latest reports from datanodes whereas each datanode checks its hosting blocks and report to the namenode. Contrary to the conventional HDFS, once the data blocks are identified as corrupt, the RaidNode first polls the corrupt data block information from the namenode, then launches recovery tasks instead of reconstructing all unavailable dataset in a batch way. However, the detection interval and recovery interval are proportional to data set scale. Table I presents the time gap between block corruption and recovered. Each file contains 32 blocks and one of them is corrupt. The values vary and depend on the arrival time of routine data detection but increase along with data sizes. It indicates the block under failure will last for a considerable long time in large scale systems. Facebook datacenter reports data lost at 100k blocks per day and data reconstruction traffic is 180TB per day in production erasure-coded Hadoop system [1]. Once the dataset is huge, the intervals are considered long that makes data states inconsistent between that in the namenode and that of actual data hosting in datanodes.

C. Impact of Unavailable Block to MapReduce Applications

1) Flawed Job: It will be a flawed job if the target input contains corrupt data that cannot be recovered in time. MapReduce scheduler pursues data locality that is acquired from centralized metadata management, but it lacks awareness of latest data states of target data blocks. The data states kept in centralized master may not reflect the actual states due to delay verify and delay report. There is a lack of communication between namenode and job scheduler (Job scheduler here is a component in both in the first generation MapReduce and Hadoop YARN), as well as in erasure-coded Hadoop system. The job scheduler assigns tasks based on the metadata that is kept in the namenode but regardless of the actual states of blocks stored in datanodes. Failure-redo mechanism will conduct another task attempts onto replications in conventional MapReduce framework but this mechanism is doomed to failure due to missing of a health data source and the lack of replication for other attempt tasks.

2) Data Reconstruction Penalty in Job Processing Runtime: To avoid entail job failure, it is reasonable to reconstruct the missing target data block in runtime. It may avoid rerunning the failure job, however, the penalty of data reconstruction will be added to the elapsed time of the task processing, which results in straggler tasks prolonging the completion time of map phase and eventually hurt the overall performance of MapReduce application. The corresponding overheads of pure data reconstruction time are demonstrated in Table II. Moreover, the data recovery will be addressed one by one, and the corresponding in-queue waiting time of data recovery tasks is not included in this table. Hence, the read delay, in practical terms, can be considerable, which significantly degrades MapReduce application performance.

3) Multiple Stragglers: An unavailable block will not only introduce a straggler task that attempts to access it. If there exists more than one map task, they will all become stragglers when one block is unavailable. This is because there is hidden data dependence between map tasks and input data block.

By intuition, MapReduce is a full parallelism in the map phase and map tasks are independent of each other. By default, splits that assigned to a map task are calculated and created by physical blocks. It is true that each map task processes its assigned split. However, the contents of a file are stored into contiguous blocks, and blocks are chunked up by configured fixed size, hence, a logical unit, such as a record may cross the boundary of physical blocks. Hadoop handles this issue transparently. A split can be considered as a reference about assigned data range, described by file path in HDFS, start offset, and length. If the start offset is non-zero, RecordReader of map task always trackbacks bytes of one synchronized marker length and skip the contents before it catches the first synchronized marker to ensure the data integrity. The RecordReader will stop reading the contents if the length of what it reads is not less than the split length. Hence, the RecordReader may need to read the last record in the split from anther following block. Thus, once a block is unavailable, more than one map task, probably three tasks, will fail consequently.

III. DESIGN

A. Design Principles

Figure 1 illustrates the idea of Dominoes. Dominoes detects the data states before task scheduling, schedules jobs

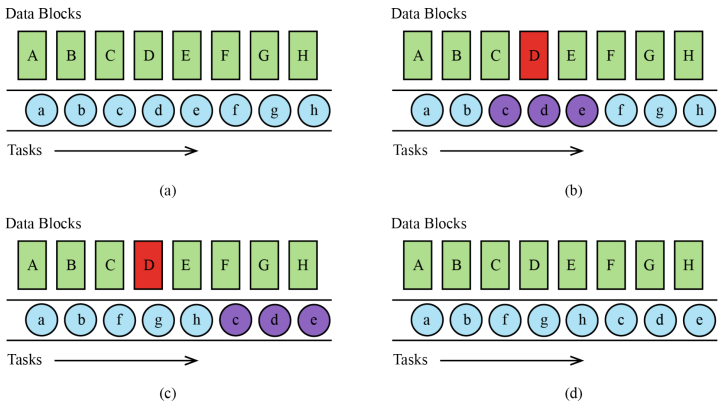


Figure 1. Example of Dominoes: (a) task a processes block A, task b process B, and so on; tasks will be scheduled according to data locality but from head to end of task queue in greedily manner; (b) red block D is detected as corrupt, the infected tasks c, d, and f are the related tasks; (c) move the infected tasks c, d, and f to the rear of task queue; (d) the tasks c, d, f process their target data block healthy while block D is recovered. Simplifying the case but without loss of generality, we use data block instead of split and the order of splits is the same after sorting according to split sizes. Job scheduling strategy is in the same manner.

and tasks accordingly. We name a task in a state of infected due to its dependent block is unhealthy. The goal is to hide the data recovery time during job processing. Dominoes consists of three software modules to meet the challenges from processing corrupt blocks in erasure-coded Hadoop system. Firstly, the centralized master should acquire information of target blocks in a reasonable very short time to recover the blocks in time for further job scheduling. Secondly, the system should handle stragglers that raise due to corrupt blocks and, should speculatively continue the attempt tasks for job completion. Thirdly, the job scheduler should be aware of the latest block states and schedules jobs and tasks accordingly, to improve overall system performance.

B. Detect Corrupt Blocks in Time

First of all, at the time before checking, if the target data set is triplication and has not been encoded yet, Dominoes skips checking and informs the scheduler the input dataset healthy. As the interval of data checking is long and the scheduler may not be aware of the latest states of target data in time, we develop a lightweight data checking mechanism to detect the corrupt data before jobs get scheduled. This strategy brings twofold benefits: 1) it identifies the problematic target data and trigger data repair; and 2) it informs the scheduler to avoid running flawed job and to schedule tasks in coordination with data recovery.

Corrupt block detection should be lightweight and respond in a short time; otherwise, such procedure may introduce higher overhead than it potentially benefits runtime systems. Distinguished from the original erasure-coded Hadoop, the scope of a block checking procedure in Dominoes is narrowed down to certain blocks in specified files rather than all block sets across the entire cluster; therefore, it significantly reduces the response time.

However, the lightweight data checking strategy has its limitations. It cannot detect data corruption due to bit flips before runtime, which might be covered by other routine blooming-filter and transmission check-sum techniques. Fortunately, if the applications adopt checksum detection during map tasks processing, corrupt data can be detected in the same way as conventional Hadoop detects it. Such exceptions due to corrupt data blocks will be captured and addressed in following subsection.

C. Tolerate Flawed Jobs or Tasks

Corrupt data blocks need to be recovered urgently since the corresponding infected tasks will become a penalty to the overall performance of the job. Hence, the results of lightweight checking should be immediately pushed to trigger data recovery procedure instead of periodically detecting and reporting to the centralized manager to shorten the interval between corrupt data block detection and data repair.

To transparently perform map task failure-redo processing, an automatically speculative runtime data recovery mechanism is proposed. Once the task cannot access its target blocks, the inaccessible blocks will be reported to the recovery manager immediately. The map tasks will be identified as failure map tasks but not in the same way as the conventional Hadoop. Such failure map tasks will not be in the first-class priority in task scheduling and be rescheduled onto another slaves. This is because there are no replications available in the system and simply re-running the task probably encounters failure due to missing of health target data blocks. Therefore, the map task will be held only the data repair procedure is completed.

D. Job Schedule According to Latest Data States

To avoid launching flawed jobs or tasks that result in degrading parallelism and wasting resource, the idea of this

module is to schedule jobs or tasks whose target data have passed through the lightweight data checking. Since the scheduler is a centralized manager and jobs are in queue while there are multiple jobs submitted. The job scheduler acquires the latest states of data blocks, and then schedules jobs and tasks considering target data states.

To improve job throughput and system utilization, when there are multiple jobs in queue, a job with corrupt data blocks will be held, and yield to the following ready jobs. Tasks are also in a queue, and the scheduler assigns tasks to slaves from begin to end of the queue but is aware of the data locality information. Hence, the infected tasks will be marked and moved to the end of the task queue. Meanwhile, the data recovery procedure is conducting and overlapping other map tasks that are processing. Moreover, it will not consume resources that are acquired by related map tasks, therefore, explores parallelism in the map phase.

The scheduler runs jobs and tasks speculatively even if it is aware of corrupt data if there is only one job in the queue. It is possible that corrupt data can be recovered in time; otherwise Hadoop will inform the corresponding user of a timeout due to data corruption.

IV. Implementation

The prototype of Dominoes is built on erasure-coded Facebook Hadoop-0.20 [16]. To coordinate the data checking, data recovery and job scheduling in Dominoes, we implement three software modules, JobDecider, FileFixer, and JobFileChecker to intercept the original job procedure. JobDecider is a new thread in NameNode, which contains two lists of jobs and a list of files, RunnableJobList stores jobs that are ready to run and WaitingJobList holds jobs with corrupt blocks, and EmergentFileQueue saves the information of files, which must be fixed as soon as possible. FileFixer is also a new thread in NameNode, which updates corrupt block information (further denoted as CBI). JobFileChecker is implemented in JobTracker to submit jobs with CBI at the beginning. The architecture of Dominoes is showed in Figure 2.

A. Job Submission

The CBI consists of four members, filename, block ID, start offset and its length. This is because RaidNode fixes corrupt blocks in the granularity of files, and block level information will be used for the task scheduling of Dominoes. The procedure of job submission in Dominoes is showed in Algorithm 1. Once a job is submitted in JobTracker, JobFileChecker will collect the CBI of a job by traversing all related data blocks from datanodes. JobFileChecker submits the job to the JobDecider according to the CBI, if all blocks are healthy, the job will be added into RunnableJobList, otherwise in WaitingJobList.

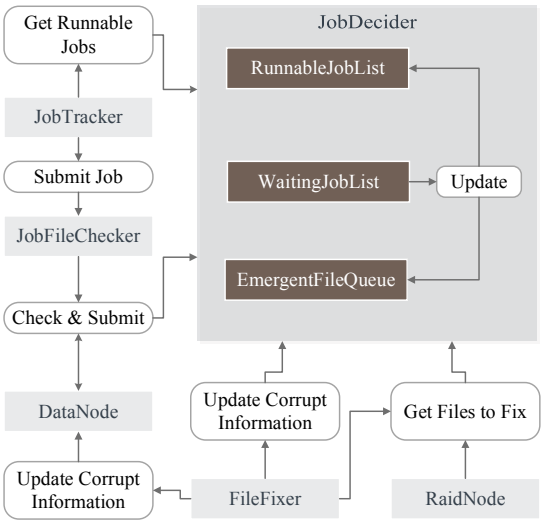


Figure 2. Architecture of Dominoes

Algorithm 1 Check&Submit

Description: $JobFileChecker$ is called by $JobTracker$ after a job is submitted to $JobTracker$
Input: $job \leftarrow$ the submitted job
Input: $jobFiles \leftarrow$ files required by the job

```
1: corruptionInfo ← newList
2: for file in jobFiles do
3:     for block in file.blocks do
4:         if block.healthyOnDataNodes() = false then
5:             corruptionInfo.add(file, block)
6:         end if
7:     end for
8: end for
9: if corruptionInfo is empty then
10:    JobDecider.addToRunnable(job.id)
11: else
12:    JobDecider.addToWaiting(job.id, corruptionInfo)
13: end if
```

B. Job Execution based on Data States

In Dominoes, a job will put on hold if the files are not ready. When there are jobs submitted to JobTracker but are not set to 'runnable' by JobDecider, JobTracker will ask JobDecider for runnable jobs periodically. If the RunnableJobList is not empty, the corresponding jobs will be returned. Otherwise, if a certain slots are underutilized, JobDecider gets the job at the head of WaitingJobList, puts all its files into EmergentFileQueue and returns the job, as shown in Algorithm 2.

C. Get Files to Fix

RaidNode asks the next file to fix periodically. JobDecider checks EmergentFileQueue firstly because the corresponding jobs have already set as runnable but with corrupt blocks.

Algorithm 2 GetRunnableJobs

Description: function on $JobDecider$ side, called by $JobTracker$ periodically

Input: $mro \leftarrow$ whether a job must be set runnable even its corrupt files are not ready

Output: a list of runnable jobs

```
1: returnList ← newList
2: if runnableJobList not empty then
3:     returnList.addAll(runnableJobList)
4:     runnableJobList.clear()
5: else if mro = true then
6:     job ← waitingJobList.poll()
7:     emergentFileQueue.addAll(job.files)
8:     returnList.add(job)
9: end if
10: return returnList
```

Algorithm 3 UpdateList

Description: update the states of jobs in $waitingJobList$, called by $JobDecider$ periodically

Input: $timeout \leftarrow$ the max time a job could be held before set runnable

```
1: updatedList ← newList
2: while job ← waitingJobList.poll() do
3:     if job.corruptFiles is empty then
4:         runnableJobList.add(job)
5:     else if job.durationTime > timeout then
6:         runnableJobList.add(job)
7:         emergentFileQueue.addAll(job.corruptFiles)
8:     else
9:         job.weightedValue ← updateValue(job)
10:        updatedList.add(job)
11:    end if
12: end while
13: waitingJobList.addAll(updatedList)
```

If the EmergentFileQueue is empty, JobDecider gets the first file of the first job in WaitingJobList, adds this file to FileFixer and then return it to RaidNode.

D. CBI Update

FileFixer stores the file-to-job mapping and traverses all blocks of files by communication with DataNodes. If a block is recovered, it will be removed from the CBI of the job. A file will be removed if all its blocks are recovered. Since a file put into EmergentFileQueue indicates that the job will try to run even it is not ready, checking the states of files in EmergentFileQueue is not necessary.

E. Job State Update

The job state update algorithm is presented in Algorithm 3. The update function is processed periodically, which traverse all jobs in WaitingJobList. A job will be set runnable if all its files are confirmed healthy. Since WaitingJobList determines the order of recovery order of files, it must consider the overall system efficiency and the job fairness. A threshold time is given for fairness, if the duration of a job held in WaitingJobList exceeds the threshold, the job will be set as runnable immediately, and all its files will be put into EmergentFileQueue for emergent repair. To improve the job throughput and resource utilization, WaitingJobList orders job by the number of un-fixed blocks. The duration time is also considered for fairness. Therefore, we propose a simple formula to calculate the weighted value:

$$\frac{Num_{unfixed\_block}}{2^{(duration/threshold)*ratio}} \quad (1)$$

Since threshold also determines the maximum job waiting time, a ratio can help adjust the inclination of the slope. The higher the ratio is, the more important the threshold is to the weighted value.

F. Data-State-Aware-Scheduler

1) Data State Aware Job Scheduling: As explained above, if job J1 is submitted before job J2, but J1 has corrupt data, J2 may run prior to J1 if J2 is ready or has smaller weighted value.

2) Data State Aware Task Scheduling: The CBI will also be returned to the JobTracker, which can help scheduling tasks. The block information in CBI indicates the range of the corrupt data, which may not have been repaired yet.

If a job is set as runnable but with corrupt data, a map task whose split overlaps a corrupt block cannot run successfully due to unhealthy data. Dominoes parses split target data range for each map task. Recall that CBI contains a file name, start offset and length, which indicates a data range that is unhealthy. By comparing such ranges between splits and the corrupt blocks, the infected map tasks are identified and marked. Those infected map tasks will be held at the end until the rest map tasks are scheduled.

V. Evaluation

This section evaluates the performance and the introduced overhead of Dominoes.

A. Scheduling Strategies for Flawed Jobs

As mentioned in section II, the original HDFS-RAID system [16] detects and fixes flawed data, the time gap is long and unacceptable. Thus, in our evaluation, we enhance the original HDFS-RAID system to handle corrupt data in two straightforward strategies, FixBeforeJob and FixInMap. FixBeforeRun checks the states of target data blocks and reports corrupt blocks to RaidNode. Jobs in FixBeforeRun strategy will be held until all corrupt blocks are fixed, even though the resource in the cluster is free. FixInMap behaves on the contrary, it processes jobs without checking target data blocks. It handles corrupt block, automatically recovers the corrupt data and then continues task processing.

B. Test Setup

We conduct our experiments on 33 nodes interconnected by a Gigabit Ethernet switch. One node is used as the master node and the rest are the slaves. Each node is equipped with a Quad-Core AMD Opteron Processor 2376, 8GB RAM, and one 150 GB SATA disk.

We choose WordCount for the evaluation jobs and submit four jobs with 30 seconds intervals. Each job processes four files, each file has 4GB data; that is 16GB data in total for one job. The first three jobs contain 10%, 8%, 5% corrupt data and the fourth has all health data. We inject data failure by deleting data blocks in local file system and bypass HDFS. We record the procedure of jobs in each strategy to capture the effectiveness and performance of Dominoes in detail. Four metrics are presented to measure the performance in system and user aspects. The system metrics are 1) completion time of all submitted jobs and 2) total time for map tasks; the user metrics are 3) average round-time for jobs and 4) average waiting time for jobs.

C. Performance Analysis on Scheduling Strategies

To illustrate the progresses of job execution under failure, we divide the procedure into four continuous stages: checking stage, hold on stage, map phase and to-the-end stage. The checking stage begins when the job is submitted into JobTracker and ends when the job is runnable, the hold on stage ends when the first map task runs, the map phase ends when all map tasks complete, the last stage ends when the job is finished.

From Figure 3(a) to Figure 3(c), the total execution time of all jobs are 1500s, 857s, and 774s under FixBeforeJob, FixInMap and Dominoes, respectively. In Figure 3(a), we investigate the order of files being fixed among jobs. Suppose $i-j$ indicates the jth file in $job-i$, the order is $1-1$, $1-2$, $2-1$, $2-2$, $3-1$, $3-2$, $1-3$, $1-4$, $2-3$, $2-4$, $3-3$ and $3-4$. The fixing order of the first six files is because $job-1$ fixes its first two file before $job-2$ is submitted, and yields to $job-2$ because the latter has less corrupt blocks. The number of corrupt blocks of $job-2$ being cacluated is less than that of $job-1$ due to the asynchronous CBI update. $Job-2$ yields to $job-3$ for the same reason. After file $3-2$ is fixed, $job-1$ reaches the timeout threshold and set as runnable immediately. Thus, file $1-3$ and $1-4$ are put into the EmergentFileQueue. $Job-2$ reaches its threshold and its unfixed files are added into EmergentFileQueue as well. During the checking stage of $job-1$ to $job-3$, $job-4$ is submitted and set as runnable immediately because all its files are ready. In Figure 3(b), all jobs are executing without checking stage. As all slots are occupied by previous jobs, later jobs have to wait. $Job-4$ waits about 450s before map stage even though all its files are ready. In Figure 3(c), $job-1$ is set as runnable immediately because all slots are free at the beginning. Later, $job-2$ and $job-3$ are submitted but have to wait because slots are busy. $Job-3$ starts before $job-2$ because it has less corrupt blocks and has higher priority in WaitingJobList. By comparing the procedure of $job-1$ in Figure 3(b) and Figure 3(c), we can see $job-1$ in Dominoes spends less time in execution. This is because the flawed tasks are scheduled to the tail of the task queue whereas the corrupt files have been fixed via EmergentFileQueue.

Figure 4 illustrates the performance of Dominoes with different parameters. Compared with $job-2$ in Figure 3(c), $job-2$ in Figure 4(a) starts prior to $job-3$ because it has a higher ratio, in which the threshold determines the order of jobs in WaitingJobList. Similarly, $job-2$ in Figure 4(b) also starts prior to $job-3$ because the timeout threshold is smaller.

D. Performance Metrics from System and User Aspects

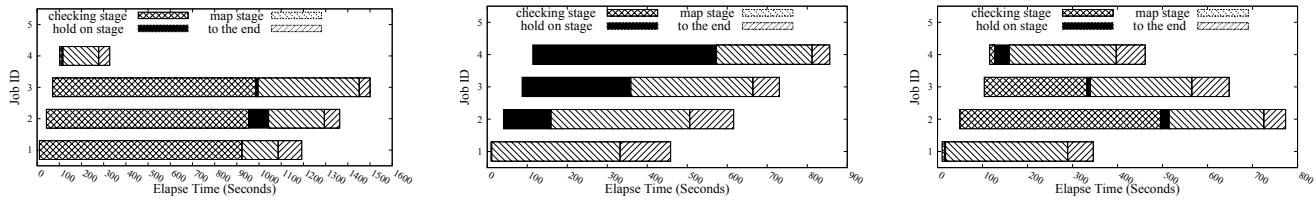
We measure the performance from system and user aspects; the corresponding results are presented in Figure 5 and Figure 6, respectively. We configure different system parameters for the runs. Figure 5(a) presents the overall job completion time that indicating the duration for all the jobs get done in system. Dominoes always finishes all jobs before the other modes, the accelerated rate compared to FixBeforeJob and FixInMap are 11.5% to 48.5% and 9.1% to 9.7%, respectively.

The less total time for map tasks of a job take, the more resource become available for other jobs. Therefore, Dominoes enhances job throughput when the cluster is heavily loaded. In Figure 5(b), Dominoes spends less time in map than FixInMap because the infected map tasks are scheduled to the end of map queue. Figure 6(a) presents the average round-time for jobs that Dominoes has the best performance. Average waiting time in job queue in Figure 6(b) indicates the fairness among users, the job executions will not be blocked in FixInMap that making the waiting duration in job queue minimum among three strategies.

In summary, FixBeforeJob has the longest job completion time. FixInMap has the shortest job waiting time but is inefficient for both system resource usage and average job round-time. Compared with FixBeforeJob and FixInMap, Dominoes considers both performance and fairness, thus it gains the highest job throughput and the minimal average job round time.

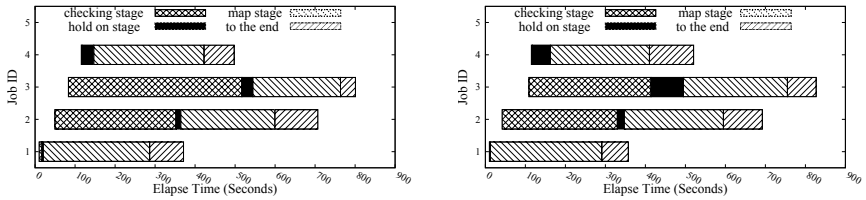
E. Dominoes Overhead Evaluation

We evaluate the potential overhead introduced by Dominoes by comparing the performance of the original HDFS-RAID and that of Dominoes for processing the same set of job in failure-free conditions. The results are showed in Figure 7. The processing progresses and elapsed time of jobs are almost the same. The completion time of HDFS-RAID and Dominoes are 762.8 and 782.6 seconds, respectively. Dominoes introduces an overhead of 2.6%, this is because the job is submitted to JobDecider before it actually runs,



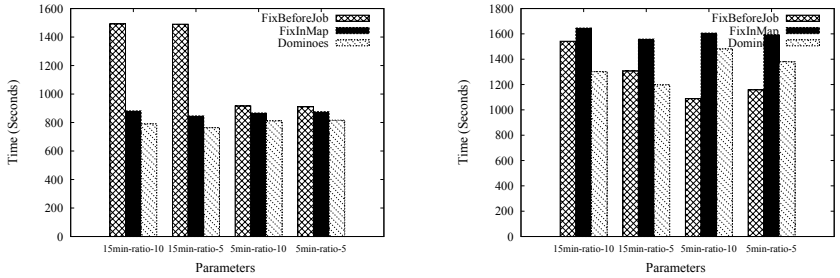
(a) FixBeforeJob (b) FixInMap (c) Dominoes

Figure 3. Performance of FixBeforeRun (a), FixInMap (b), and Dominoes (c) with parameter threshold 15 mins and ratio 5



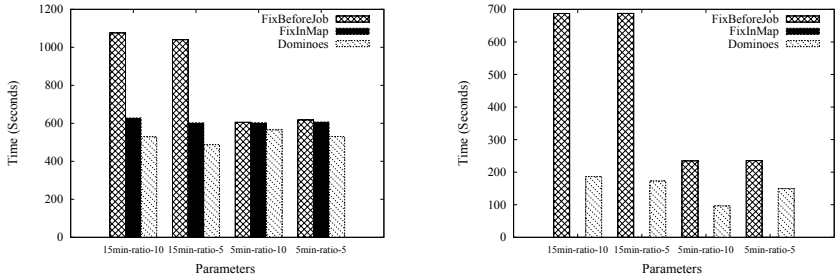
(a) 15 min and ratio 10 (b) 5 min and ratio 5

Figure 4. Performance of Dominoes under different parameters



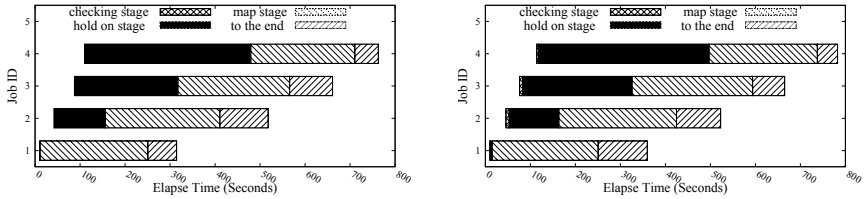
(a) Overall job completion time (b) Total time for map tasks

Figure 5. Performance metrics from system aspect



(a) Average job round-time (b) Average job waiting time

Figure 6. Performance metrics from user aspect



(a) HDFS-RAID (b) Dominoes

Figure 7. Performance with failure-free jobs

which involves cross-node communication in checking states of blocks. From Figure 7, it is clear that the overhead happens in the checking stage. When the cluster is heavily loaded, the overhead can be hidden by overlapped job waiting time.

VI. Related Work

Our work builds on HDFS-RAID system and existing erasure coding algorithms. We first discuss some of the important efforts on erasure-coded systems, especially for HDFS; we then present the related work in scheduling strategies for Hadoop system.

A. Erasure-coded HDFS

1) Corrupt Data Detection: The conventional HDFS features checksum and HDFS-RAID employs parity codes for data integrity. HARDFS [17] proposed a lightweight versioning mechanism on namespace management, replication management, and read/write protocols, to detect silent failures in HDFS. It enhances the data integrity of HDFS during amending data operations. On the contrary, our work relies on existing detect techniques but it conducts an on-demand check on the target data while scheduling jobs.

2) Reduce Storage Overhead: Gibson et al. proposed DiskReduce that applies RAID-5 and RAID-6 redundancy coding over HDFS, which efficiently introduce erasure coding to HDFS [18] [19]. Facebook datacenter uses Reed-Solomon codes to efficiently reduce storage cost from default 3X to 1.4X. The recently-introduced HACFS [1], according to the data access frequencies divide data into hot and cold dataset, and adopts two erasure coding techniques that achieve optimal reconstruction performance and storage overhead, respectively; it eventually obtains both optimal tradeoffs in system-wide. These work focus on fault tolerance performance and optimizing read degrade for MapReduce applications. Most efforts have been made from storage system aspect. Few of them has touched job or task processing, particularly failing to address the stragglers or flawed job that raise due to flawed data, in an erasure-coded Hadoop system.

3) Data Recovery Optimization: Microsoft proposed Local Reconstruction Code (LRC) [8] that encodes data into local parity codes and global parity codes for efficiently repairing single block failure and multiple failures respectively, which reduces involved data blocks for data recovery, therefore, greatly reducing network traffic and I/O bandwidth needed for reconstruction. Sathiamoorthy et al. applied the XOR algorithm [7] that can be calculated very efficient in modern computers, onto RAID-HDFS system, also grouping raw data blocks into two-dimensional layout, and computing parity blocks in vertical and horizontal way; Rashmi et al. introduced a novel erasure coding Hitchhiker [12] that built over RS codes. Both of them reduce the network and I/O bandwidth for recovering failure data blocks. Reconstruction for on-demand read has been adopted in Windows Azure Storage [8], in which reading for an unavailable data block will trigger reconstruction immediately. Though this system experiences reading delay, it avoids failure. Dominoes implements such delay read feature; moreover, earlier detect and data-state-aware scheduling can further avoid the read latency while processing tasks speculatively.

B. Hadoop Scheduling Strategies

1) Alleviating Stragglers in Hadoop System: On the other hand, numerous work has been dedicated to alleviating skews and stragglers for Hadoop system. As the latest completed map task determines the completion time of intermediate data shuffling phase, potentially prolongs the overall job completion time [20]. Conventional Hadoop adopts speculative tasks that launching multiple speculative tasks while a few percentile of map tasks remains [21]. Besides, lots of work is proposed to alleviate the stragglers and skews due to input data size skew [22] [23], computation skew [22], heterogeneous hardware [24], or even non-assumption skews [25]. Different from existing work, our paper focuses particularly to alleviate the stragglers that are caused by accessing unavailable data blocks in erasure-coded Hadoop system.

2) Delay Scheduling: Delay scheduling improves data locality [26] [27]. MapReduce prefers assigning tasks to the nodes where the target data reside in order to avoid network traffic and latency. On the other hand, MapReduce pursues parallelism greedily by assigning tasks while computing resources available. Meanwhile, the slaves communicate with job scheduler every three seconds, which consequently hurts data locality. Zaharia et al. proposed to delay a few seconds (e.g.,5 seconds) when assigning a task to optimize the possibility of data locality and improve overall job throughput system-wide, which especially benefits small jobs that are dominating workload in datacenters [26]. Guo et al. adopted the delay scheduling and further proposed to consider data locality for multiple potentials due according to replications coordinately [27]. Our work is similar to delay scheduling, aiming for high job throughput system-wide. However, the delay latency introduced in our work is paid per job for earlier failure detection and recovery, whereas delay overhead in delay scheduling is paid per task for data locality concern.

3) Exploiting Parallelism: On the one hand, when the reduce tasks cannot get the computing resource will significantly prolong the completion time [28] [29]. On the other hand, if reduce tasks occupy resource too early will degrade the parallelism of map tasks, and data shuffle can overlap computation but without occupying reduce tasks slots [30]. Those issues alleviated in YARN because the task type-free resource allocation policy but still hold. Similarly, Dominoes greedily exploits parallelism, but by 1) scheduling map tasks to avoid failures due to data unavailability and 2) data

recovery overlaps task processing whereas data detection overlaps job processing.

VII. Conclusion

Due to huge data volume consuming considerable storage space, many distributed systems have adopted erasure coding instead of replicas to reduce long-term storage overhead thus saving capital cost. Erasure-coded systems provide at least three-way equivalent fault tolerance for data loss so that data durability is guaranteed, however, data availability is significantly degraded. This introduces job failure or task stragglers due to unavailable data blocks that hurt job performance and job throughput. We propose a novel approach, namely Dominoes, combining lightweight data checking mechanism and job scheduling techniques to hide the latency and improve job throughput. The experimental results confirm Dominoes improves average performance by 9% to 9.7% under failure and at a 2.6% overhead for a failure-free condition.

Acknowledgment

The authors would like to thank the partial support from NASA AIST 2015 funding. This research is also supported in part by NSF under NSF grants CNS-0751200, CNS-1162540, and CNS-1526887.

References

[1] M. Xia, M. Saxena, M. Blaum, and D. A. Pease, "A tale of two erasure codes in hdfs," in *FAST*, 2015.

[2] J. Dean and S. Ghemawat, "MapReduce: Simplied Data Processing on Large Clusters," in *OSDI*. USENIX, 2004.

[3] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *Proceedings of ACM Symposium on Operating Systems Principles*, 2003.

[4] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *MSST*. IEEE, 2010.

[5] "Colossus: Successor to the google file system (gfs)," http://www.highlyscalablesystems.com/3202/colossus-successor-to-google-file-system-gfs/.

[6] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu, "Data warehousing and analytics infrastructure at facebook," in *ACM SIGMOD*. ACM, 2010.

[7] 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 *Proceedings of the VLDB Endowment*, 2013.

[8] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure coding in windows azure storage," in *USENIX ATC*. USENIX, 2012.

[9] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," in *FAST*, 2002.

[10] "Gpfs file placement optimizer," http://www-01.ibm.com/support/knowledgecenter/SSFKCN_3.5.0/com.ibm.cluster.gpfs.v3r5.gpfs200.doc/bl1adv_fposettings.htm.

[11] D. Zhao, K. Burlingame, C. Debains, P. Alvarez-Tabio, and I. Raicu, "Towards high-performance and cost-effective distributed storage systems with information dispersal algorithms," in *CLUSTER*. IEEE, 2013.

[12] S. Salinas, G. Schiavo, and E. J. Kremer, "A hitchhiker's guide to the nervous system: the complex journey of viruses and toxins," *Nature Reviews Microbiology*, 2010.

[13] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache Hadoop Yarn: Yet Another Resource Negotiator," in *SoCC*, 2013.

[14] K. Wang, N. Liu, I. Sadooghi, X. Yang, X. Zhou, T. Li, M. Lang, X.-H. Sun, and I. Raicu, "Overcoming hadoop scaling limitations through distributed task execution," in *Cluster*, 2015.

[15] Apache Software Fundation. (2015) HDFS Default Configuration 2.7.0. [Online]. Available: http://hadoop.apache.org/docs/r2.7.0/hadoop-project-dist/hadoop-hdfs/hdfs-default.xml

[16] "Hdfs-raid: Facebook's realtime distributed fs based on apache hadoop 0.20-append," https://github.com/facebookarchive/hadoop-20.

[17] T. Do, T. Harter, Y. Liu, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Hardfs: Hardening hdfs with selective and lightweight versioning," in *FAST*, 2013.

[18] G. Gibson, P. Nowoczynski, M. Polte, L. Xiao, and P. S. Center, "Failure recovery issues in large scale, heavily utilized disk storage systems."

[19] B. Fan, W. Tantisiriroj, L. Xiao, and G. Gibson, "Diskreduce: Raid for data-intensive scalable computing," in *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, ser. PDSW '09. ACM, 2009.

[20] G. Ananthanarayanan, A. Ghodsi, A. Warfield, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "PACMan: Coordinated Memory Caching for Parallel Jobs," in *NSDI*, 2012.

[21] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving MapReduce Performance in Heterogeneous Environments," in *OSDI*, 2008.

[22] E. Pettijohn, Y. Guo, P. Lama, and X. Zhou, "User-centric heterogeneity-aware mapreduce job provisioning in the public cloud," in *ICAC 14*. USENIX, 2014.

[23] S. Ibrahim, H. Jin, L. Lu, S. Wu, B. He, and L. Qi, "Leen: Locality/Fairness-Aware Key Partitioning for MapReduce in the Cloud," in *Cloud Computing Technology and Science (CloudCom)*. IEEE, 2010.

[24] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. Vijaykumar, "Tarazu: Optimizing MapReduce on Heterogeneous Clusters," in *ACM SIGARCH Computer Architecture News*, 2012.

[25] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skewtune: Mitigating Skew in MapReduce Applications," in *ACM SIGMOD*. ACM, 2012.

[26] 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 *EuroSys*. ACM, 2010.

[27] Z. Guo, G. Fox, and M. Zhou, "Investigation of Data Locality in MapReduce," in *CCGrid*. IEEE, 2012.

[28] Y. Wang, J. Tan, W. Yu, X. Meng, and L. Zhang, "Preemptive ReduceTask Scheduling for Fair and Fast Job Completion," in *ICAC*. USENIX, 2013.

[29] J. Tan, X. Meng, and L. Zhang, "Coupling Task Progress for Mapreduce Resource-Aware Scheduling," in *INFOCOM*. IEEE, 2013.

[30] Y. Guo, J. Rao, and X. Zhou, "iShuffle: Improving Hadoop Performance with Shuffle-on-Write," in *ICAC*, 2013.