On Data Parallelism of Erasure Coding in Distributed Storage Systems

Jun Li, Baochun Li
Department of Electrical and Computer Engineering, University of Toronto, Canada
{junli, bli}@ece.toronto.edu

***Abstract*—Deployed in various distributed storage systems, erasure coding has demonstrated its advantages of low storage overhead and high failure tolerance. Typically in an erasure-coded distributed storage system, systematic maximum distance separable (MDS) codes are chosen since the optimal storage overhead can be achieved and meanwhile data can be read directly without decoding operations. However, data parallelism of existing MDS codes is limited, because we can only read data from some specific servers in parallel without decoding operations. In this paper, we propose Carousel codes, designed to allow data to be read from an arbitrary number of servers in parallel without decoding, while preserving the optimal storage overhead of MDS codes. Furthermore, Carousel codes can achieve the optimal network traffic to reconstruct an unavailable block. We have implemented a prototype of Carousel codes on Apache Hadoop. Our experimental results have demonstrated that Carousel codes can make MapReduce jobs finish with almost 50% less time and reduce data access latency significantly, with a comparable throughput in the encoding and decoding operations and no additional sacrifice of failure tolerance or the network overhead to reconstruct unavailable data.**

***Keywords*-data parallelism, distributed storage, Reed-Solomon codes, minimum-storage regenerating codes, MapReduce**

I. Introduction

Distributed storage systems, such as the Hadoop Distributed File System (HDFS) [1] and Windows Azure Storage (WAS) [2], store tremendous volumes of data on a large number of commodity servers. As server failures are frequent [3], distributed storage systems must store redundancy to protect data against these failures. For example, in HDFS, data are replicated 3 times by default, such that any two failures can be tolerated.

Erasure coding, such as Reed-Solomon (RS) codes [4], has been increasingly replacing replications in distributed storage systems due to its high failure tolerance and low storage overhead [5], [6]. An (n, k) RS code that encodes k blocks of data into n blocks, can decode the original data from any k blocks among such n blocks. With the same number of failures to tolerate, MDS codes achieve the optimal storage overhead. For example, using an $(n = 6, k = 4)$ RS code, we can always decode data with no more than 2 unavailable blocks by paying a 1.5x storage overhead, while a 3x storage overhead must be paid for the 3-way replication to tolerate the same number of failures. In coding theory, this property is known as maximum distance separable (MDS) property.

As decoding operations can increase the latency of data access, MDS codes deployed in distributed storage systems are typically *systematic*, in that k out of total n blocks are exactly the same as the original data. These k blocks are hence called *data blocks*, and the other blocks are known as *parity blocks*. Therefore, reading data without decoding operations is possible as we just need to read the k data blocks, without requiring any of the parity blocks unless some data block is not available.

Systematic codes can reduce the latency and improve the throughput of reading data. However, data parallelism, which refers to the number of blocks that can be read by different processes simultaneously, is also limited by existing systematic erasure codes, since we cannot read original data directly from parity blocks. For example, when we run a MapReduce job on data stored in HDFS, with systematic erasure codes the number of map tasks will be limited by the number of data blocks as we cannot run any map tasks by reading only parity data from local servers. Therefore, unlike replication where we can easily extend data parallelism by increasing the number of copies, data parallelism of existing systematic erasure codes is forever limited by the number of data blocks unless additional network transfer is allowed, no matter how many blocks we have in total.

In this paper, we present *Carousel codes*, designed to extend data parallelism from reading k data blocks in parallel to reading all n blocks. Different from existing erasure codes, Carousel codes can sequentially embed the original data into all the blocks, instead of just k particular blocks, and therefore data can be read in parallel from all the blocks with a higher overall throughput. However, with all the blocks containing original data, we will have to perform decoding operations to read data even though only one block is unavailable. Hence, the construction of Carousel codes can allow the number of blocks that contain original data to be arbitrarily specified between k and n, in order to achieve a flexible tradeoff between data parallelism and data availability.

Moreover, Carousel codes can further achieve the optimal network traffic to reconstruct an unavailable block. With RS codes, for example, k blocks must be downloaded from remote servers to reconstruct one unavailable block, leading to a k-time increase of network traffic. As shown by Dimakis *et al.* [7], the optimal volume of network transfer to

reconstruct a block with MDS codes from d existing blocks is only $\frac{d}{d-k+1}$ times of a block, $k \leq d < n$. In this paper, we show that besides the MDS property and the configurable data parallelism, Carousel codes can achieve such optimal network transfer during reconstruction as well.

We have implemented Carousel codes in C++, and developed its prototype in Apache Hadoop. Our experimental results have shown that, compared to the original HDFS with RS codes, the completion time of MapReduce jobs running on our Hadoop cluster can be saved by up to 46.6%. Besides, the data access time to obtain data from HDFS can be significantly reduced as well.

II. Motivating Examples

We first provide an example to motivate the introduction of Carousel codes. In this example, as shown in Fig. 1a, we assume that in a distributed storage system, a $(5, 3)$ systematic RS code is deployed. Further, as a toy example, we assume that a file contains 3 blocks, which can be encoded into 5 blocks, and that the original data are embedded in the first 3 data blocks. To maximize failure tolerance, the 5 blocks are stored on different servers. Take running a MapReduce job on this file as an example, the number of map tasks depends on the number of data blocks [8]. In Hadoop, each map task will be preferably located on the local server that hosts the corresponding data block. In this example, there will be 3 map tasks running on 3 servers, each storing one of the 3 data blocks. The number of map tasks can not be extended to servers that store parity blocks, since no original data can be read solely from parity blocks without decoding.

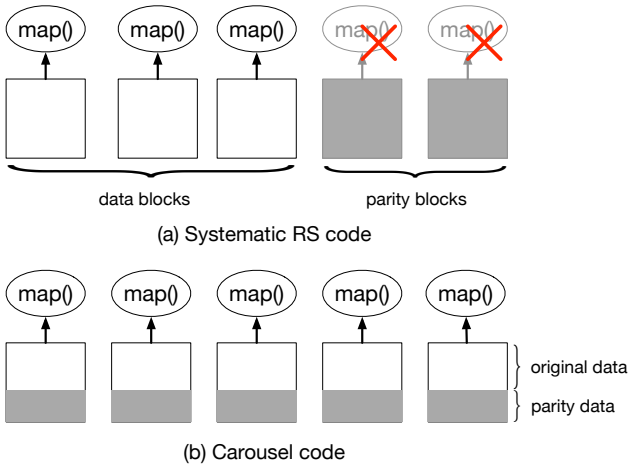

Figure 1. A comparison of data parallelism: systematic RS codes vs. Carousel codes.

Carousel codes, on the other hand, can maintain the same property as RS codes that any 3 blocks among 5 blocks are sufficient to decode the original data. Different from traditional systematic RS codes, we embed the original data into all the 5 blocks such that we can run 5 map tasks on servers with local data. As shown in Fig. 1b, $\frac{3}{5}$ of each block store original data and the rest store parity data. Compared with systematic RS codes, each map task will process 40% fewer data and we can expect less time spent on running map tasks and a shorter job completion time overall.

Similarly, when a remote client downloads the entire file from the distributed storage system, it can download data from all the servers with original data. When the read throughput is bottlenecked by the servers that store data, *e.g.*, on the hard disks or the outgoing links, reading from more such servers in parallel will increase the overall throughput of obtaining this file. In fact, it has been shown that $(n = 9, k = 6)$ RS codes can perform better on the overall read throughput than 3-way replication, by reading k blocks in parallel [9], [10]. With Carousel codes, we can achieve further improvements by extending the parallelism from k to n.

III. Related Work

Erasure coding in distributed storage systems. Achieving the MDS property, RS codes offer the optimal storage overhead to tolerate the same number of failures. Hence, RS codes have been widely deployed in various distributed storage systems [5], [10]–[15]. However, it has been shown that RS codes can incur a huge amount of extra traffic to reconstruct unavailable blocks [16], as an (n, k) RS code needs k blocks to reconstruct only one unavailable block. Hence, many commercial storage systems or research projects deploy other novel erasure codes that are designed specifically for distributed storage systems. For example, locally repairable codes or its variants have been deployed in [3], [6], [17], [18], such that only a small number of blocks needs to be downloaded during reconstruction.

The optimal traffic of reconstruction among all MDS codes, however, is achieved by minimum-storage regenerating (MSR) codes [7]. Rashmi *et al.* [19] proposed a general construction of (n, k, d) MSR codes where $d \geq 2k - 2$, which reconstruct one missing block from d existing ones. With some specific values of (n, k, d), some constructions of MSR codes (*e.g.,* [20]–[22]) have been proposed on a small Galois field. In this paper, we focus on improving data parallelism of MDS codes, and hence propose Carousel codes that achieve the optimal amount of network traffic as MSR codes and further extend data parallelism.

Performance of data access with erasure coding. As erasure coding has been widely deployed in distributed storage systems, it becomes more important to optimize the performance of data access related to erasure coding, and many efforts have been made on degraded read. Degraded read is performed when a data block is requested but unavailable, by decoding existing k blocks including parity blocks. For MapReduce jobs running over data in an erasure-coded distributed storage system, Li *et al.* [23] proposed a scheduling algorithm for MapReduce that mitigates the

latency when data must be obtained with degraded reads. On the other hand, Xia *et al.* [18] and Li *et al.* [24] both considered the skewness of data demand and store data into multiple tiers with different erasure codes, in order to reduce the cost of degraded reads by reducing its complexity on data with high demand. Hu *et al.* [25] considered the problem of load balancing in an erasure-coded distributed storage system, such that the tail access latency of degraded read can be reduced by placing anti-correlated blocks in terms of demand on the same server.

However, degraded read requests typically occupy a small portion among all read requests, and with systematic erasure codes, most read operations will visit data blocks only. In contrast, Carousel codes can improve the performance of data access, such as running MapReduce jobs or retrieving data from the storage system, by extending the degree of data parallelism. Although there have been previous works [26]–[30] that can also distribute original data into all blocks, they are not designed particularly considering the performance of parallell data processing. For example, some of existing works rely on data striping [27], [29], that divides data into very small pieces. Data striping makes original data in each block out of order, and thus can compromise the performance of MapReduce jobs running on it. On the other hand, none of these works can consume a high volume of network traffic during recontruction. In summary, Carousel codes are, to out best knowledge, the first erasure codes that can achieve both high data parallelism running MapReduce jobs and the optimal network traffic during reconstruction at the same time. Besides, we also offer the flexiblity to control the degree of data paralllism in Carousel codes.

IV. Preliminaries

In a distributed storage system, it is a common practice that data are stored into blocks of the same size [1], [2]. In order to tolerate potential failures that make blocks unavailable, (n, k) RS codes can encode k blocks into n blocks such that any k among n blocks are sufficient to decode the original data. In order to reduce the latency of data access, typically RS codes deployed in a distributed storage system are systematic. In other words, the n blocks contain k data blocks and $n - k$ parity blocks.

The encoding operations of RS codes are performed on the unit of *symbols*, and each block contains a certain number of such symbols. Typically, a symbol is simply a byte and the arithmetic operation is performed on the *Galois field* of size $GF(2^8)$. Though the symbol and its corresponding Galois field may have different sizes in practice, in this paper we assume that a symbol is a byte.

The encoding and decoding operations of RS codes can be interpreted and implemented as matrix multiplications on the Galois field. Given k original blocks of w bytes, we can represent them as $f_1, \ldots, f_k$, where f_i is a row vector of w symbols, $i = 1, \ldots, k$. An (n, k) RS code encodes the k blocks by multiplying them with an $n \times k$ generating matrix G, *i.e.*, $G \cdot \begin{bmatrix} f_1^T & \cdots & f_k^T \end{bmatrix}^T$, where f_i^T denotes the transpose of f_i, and each row in the result denotes one block after encoding. If the RS code is systematic, G must contain a $k \times k$ identity matrix. Dividing G into n submatrices of size $1 \times k$ such that $G = \begin{bmatrix} g_1^T & \cdots & g_n^T \end{bmatrix}^T$, the n blocks after encoding can be represented as $g_i F$, $i = 1, \ldots, n$, where $F = \begin{bmatrix} f_1^T & \cdots & f_k^T \end{bmatrix}^T$. Without loss of generality, we assume that the top k rows always constitute an identity matrix for systematic RS codes. In this way, $g_1 F, \ldots, g_k F$ are data blocks and the rest are parity blocks. If there are more than k blocks to store, they will be stored in multiple *stripes* where each stripe encodes k original blocks.

To achieve the MDS property, the generating matrix of an (n, k) RS code must guarantee that any k rows can constitute a non-singular submatrix. As such, we can decode the original data from any k blocks by calculating the inverse of this submatrix. For example, given k blocks $g_{i_1} F, \ldots, g_{i_k} F$, where $\{i_j | j = 1, \ldots, k\}$ is a k-subset of $\{1, \ldots, n\}$, the original k blocks in F can be calculated by

$$F = \left(\begin{bmatrix} g_{i_1}^T & \cdots & g_{i_k}^T \end{bmatrix}^T \right)^{-1} \cdot \begin{bmatrix} (g_{i_1} F)^T & \cdots & (g_{i_k} F)^T \end{bmatrix}^T. \quad (1)$$

From any k blocks, we can always obtain the inverse of the matrix in the equation above as any k rows in G constitute a non-singular matrix.

To reconstruct a block, at least k blocks are required. Without loss of generality, we reconstruct $g_1 F$ as an example, from k blocks $g_{i_1} F, \ldots, g_{i_k} F$ where $\{i_j | j = 1, \ldots, k\}$ is a k-subset of $\{2, \ldots, n\}$. As we can decode F from these k blocks by (1), we can simply obtain $g_1 F$ by calculating

$$g_1 \left(\begin{bmatrix} g_{i_1}^T & \cdots & g_{i_k}^T \end{bmatrix}^T \right)^{-1} \cdot \begin{bmatrix} (g_{i_1} F)^T & \cdots & (g_{i_k} F)^T \end{bmatrix}^T. \quad (2)$$

During such reconstruction, k blocks are downloaded to reconstruct only one block, leading to a high consumption of network bandwidth. In fact, it has been proved that if there are d blocks available, the optimal network transfer needed to reconstruct a block with an (n, k) MDS code equals the size of $\frac{d}{d-k+1}$ blocks, $k \leq d < n$, achieved by MSR codes [7]. Different from RS codes, all blocks encoded by (n, k, d) MSR codes are further divided into α segments where $\alpha = d - k + 1$. Thus, one segment will be obtained from each of the d existing blocks during reconstruction. In fact, an (n, k) RS code can be considered as a special case of MSR codes with $d = k$.

For simplicity, we assume that each block contains w bytes, and w is divisible by α. Hence, a block f_i can be written as an $\alpha \times \frac{w}{\alpha}$ matrix, $i = 1, \ldots, k$, where each row contains one segment. The generating matrix G will then be of size $n\alpha \times k\alpha$. Similarly, we can divide G into n submatrices of size $\alpha \times k\alpha$ and then all blocks after encoding can still be written as $g_i^T F$, $i = 1, \ldots, n$. Moreover, as MSR codes are MDS, given $\{i_j | j = 1, \ldots, k\}$ that is a k-subset

of $\{1,\ldots,n\}$, $\left[g_{i_1}^T \ \cdots \ g_{i_k}^T\right]^T$ must be non-singular, and we can still apply (1) to decode the original data. Systematic MSR codes must further have an identity submatrix of size $k\alpha \times k\alpha$ in G. To be consistent with RS codes, we still assume that the first k blocks are data blocks and the rest are parity blocks.

To reconstruct a block g_iF from d existing blocks, each existing block will be encoded into just one segment on its local server at first. For example, an existing block g_jF, $j \neq i$, will be multiplied by a row vector $v_{i,j}$ of size α on its left, *i.e.,* $v_{i,j}g_jF$. A server that downloads d such segments from d existing servers can encode them into g_iF. For clarity and simplicity, we omit the details of such reconstruction in this paper, which can be found in [19], [31] for interested readers.

V. Extending Data Parallelism

A. An illustrative example

We demonstrate the construction of Carousel codes from a toy example. In Fig. 2 we present an example of Carousel codes with $n = 3$ and $k = 2$. As $k = 2$, we assume that the original data contains 2 blocks. In this example, each block will be divided into 3 units of data, which are labeled as $1-3$ and $4-6$ in Fig. 2, respectively. The Carousel code computes 3 blocks from these 2 blocks. Still, each block contains 3 units of data, 2 of which correspond to those in the original data. The third unit in each block, on the other hand, contains parity data. Hence, all original data are evenly distributed into all the blocks. Compared with $(n = 3, k = 2)$ RS codes which contain 2 data blocks, this Carousel code achieves the same storage overhead while making it possible to extend data parallelism into all 3 blocks, by reading the top $\frac{2}{3}$ of each block. Moreover, it is easy to observe that any 2 blocks can decode the original data while the size of each block equals a half of the original data. Hence, the same as RS codes, this Carousel code is MDS as well.

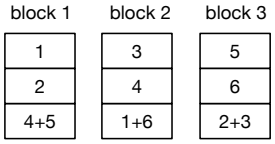

Figure 2. An example of Carousel codes with $n = 3$ and $k = 2$.

B. General construction

We now present the general construction of Carousel codes. Given n and k, the corresponding Carousel codes can be constructed from existing systematic RS codes.

Step 1: Expansion. Given an (n, k) systematic RS code, we expand each block into N units where $\frac{K}{N}$ is the irreducible fraction of $\frac{k}{n}$. Without loss of generality, we assume that the first k blocks are data blocks, and label the units in block i as $1+(i-1)N, 2+(i-1)N, \ldots, iN$, $i = 1,\ldots,k$.

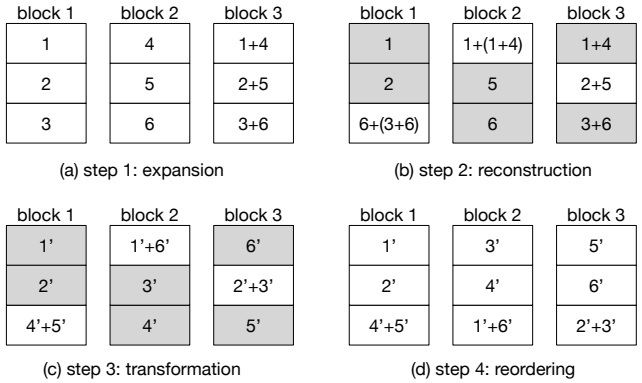

Figure 3. An illustration of the construction of Carousel codes with $n = 3$ and $k = 2$.

These units are known as *data units* as they come from data blocks. The units in the other $n-k$ parity blocks, referred to as *parity units*, can be represented as linear combinations of the units of the k data blocks in the same row. As an example, Fig. 3(a) shows a $(n = 3, k = 2)$ systematic RS code where the only parity block is a linear combination of both data blocks[1]. Here $\frac{2}{3}$ is already an irreducible fraction, and thus we expand each block into 3 units.

Step 2: Reconstruction. In this step, we first choose K units from each block in a round-robin manner. Specifically, we choose K units starting from the first one in block 1 down towards the K-th one where all the remaining $N-K$ units remain unchosen, from the second one in block 2 down towards the $(K+1)$-th one, and so on. If there are no K units below a starting unit, we will go back to the first one and keep choosing the units downwards until K units have been chosen. For example, in block N, we start choosing from the last unit and then choose the top $k-1$ units. In Fig. 3b, we highlight the chosen unit in each block. If $N \neq n$, we choose the first K units again from block $N+1$, and so on. In other words, for any positive integer j, the units chosen in block $i + jN$ are the same as those chosen in block i, $i = 1,\ldots,N$.

In general, in block i, the j-th unit is chosen if and only if $(j-i) \bmod N \in [0, K-1]$.

Hence, it can be proved that in each row of all the blocks, there will be k units chosen. Without loss of generality, we only consider the last unit. The last unit in a block will be chosen if and only if the starting unit in the block is the $(N-K+1)$-th unit or below. From the rule above we know that there are $K \cdot \frac{n}{N} = K \cdot \frac{k}{K} = k$ such blocks, from block $N-K+1$ to block N, block $2N-K+1$ to block $2N$, and so on. In other words, the unit in the last row will be chosen in k blocks. Similarly, we can prove that every row will contain k chosen units.

As all parity units are linear combinations of data units

[1]For simplicity, we may omit coefficients of linear combinations in this paper when there is no ambiguity.

in the same row, we can reconstruct any unchosen unit from the k chosen units in the same row. For example, in Fig. 3b we can reconstruct the third unit in block 1 from the two units in block 2 and block 3. In other words, unit 3 can be written as a linear combination of 6 and $3+6$. In this way, we can rewrite all chosen units by reconstructing them from the k unchosen units.

Step 3: Transformation. In this step, we transform all chosen units into a data unit. To achieve this, all chosen blocks from block 1 to block n should be mapped to new labels between 1 and Kn. For example in Fig. 3c, we map the first two chosen units 1 and 2 into unit $1'$ and $2'$. Similarly, we map the second and the third unit in block 2 into unit $3'$ and $4'$, and the third and the first units in block 3 as unit $5'$ and $6'$. In block i where $i \leq N$, the sequence of labels starts from the i-th unit, and goes downwards (and possibly back to the top) until the unit is not unchosen. Once again for any positive integer j, the sequence in block $i+jN$ should be the same as block i.

As all unchosen units can be written as linear combinations of k chosen units in the same row, we can map all unchosen units by substituting all the components in such linear combinations with the new labels of chosen units in the same row. Hence, we map the three unchosen units in the three blocks in Fig. 3c from $3 = 6+(3+6)$, $4 = 1+(1+4)$ and $2+5$, into $4'+5'$, $1'+6'$ and $2'+3'$. We will show in Sec. VI that the new code is equivalent to the original one after such a transformation by formalizing the construction. We can notice that the new code now distributes all data units evenly into all the blocks now.

Step 4: Reordering. In this step, we reorder the units in all the blocks such that all data units are at the top of their blocks. In all the blocks, we rotate units upwards until the unit with the smallest label reaches the top. In particular, for any integer $j \geq 0$, we should move units upwards by $i-1$ units in block $i+jN$. After this step, the construction is finished. We can see from Fig. 3d that the Carousel code with $n = 3$ and $k = 2$ in Fig. 2 is now constructed.

C. Properties

Before Step 4, we can decode/reconstruct any unit from any k units in the same unit. As Step 4 only changes the sequence of units in each block, we can still decode/reconstruct the original data from any k blocks in Carousel codes. Moreover, since the size of each block and the amount of original data in total remain unchanged, we can prove that Carousel codes are still MDS codes.

To reconstruct any block, we need to reconstruct all units in this block. Since before Step 4 any unit can be reconstructed from any k units in the same row, after the reordering in Step 4 the j-th unit in block i can be reconstructed from k of any j'-th units in block i' where $(i+j) \equiv (i'+j') \bmod n$. For example, we can reconstruct unit $4+5$ in block 1 from unit 4 in block 2 and unit 5 in block 3.

VI. Achieving Optimal Reconstruction Traffic

In this section, we extend the construction of Carousel codes by adding a new parameter d, which specifies the number of existing blocks required during reconstruction, $k \leq d < n$. With a given value of d, Carousel codes incur the same network traffic as MSR codes to reconstruct an unavailable block. In other words, Carousel codes can achieve the optimal network traffic during reconstruction. In fact, we will show that the construction above is a special case with $d = k$.

A. Expanding MSR codes

Given values of (n, k, d), Carousel codes can be constructed from an (n, k, d) systematic MSR code. In this paper, we use Rashmi *et al*'s construction of MSR codes [19]. Following the background knowledge of MSR codes in Sec. IV, the generating matrix G can be split into n submatrices $(g_1, \ldots, g_n)$ and encode the original data F into n blocks $(g_1F, \ldots, g_nF)$. Notice that when $d = k$, the corresponding MSR code is also an RS code.

As we know, with MSR codes each block contains $\alpha = d-k+1$ segments. To construct Carousel codes, we further split each segment into N units, where $\frac{K}{N}$ is the irreducible fraction of $\frac{\alpha k}{n}$. In other words, we expand g_i by multiplying each element with an identity matrix of size $N \times N$, $i = 1, \ldots, n$. For example, if $g_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$ and $N = 4$, we will expand g_1 into $\hat{g}_1 = \begin{bmatrix} \mathbf{I}_4 & \mathbf{0}_4 & \mathbf{0}_4 & \mathbf{0}_4 \\ \mathbf{0}_4 & \mathbf{I}_4 & \mathbf{0}_4 & \mathbf{0}_4 \end{bmatrix}$ where $\mathbf{I}_4$ and $\mathbf{0}_4$ are a 4×4 identity matrix and a 4×4 zero matrix, respectively. In this way, each segment now contains N units in $\hat{g}_i$, $i = 1, \ldots, n$, and then the block after expansion can be represented as $\hat{g}_1^T \hat{F}$, where $\hat{F}$ is obtained from $F = \begin{bmatrix} f_1^T & \cdots & f_k^T \end{bmatrix}^T$ by transforming the original data in f_i into an $N\alpha \times \frac{w}{N\alpha}$ matrix $\hat{f}_i$, $i = 1, \ldots, n$, such that $\hat{F} = \begin{bmatrix} \hat{f}_1^T & \cdots & \hat{f}_k^T \end{bmatrix}^T$. Following this expansion, g_iF and $\hat{g}_i\hat{F}$ are equivalent, containing the same data, where the only difference is that all bytes in the block are stored in matrices of different sizes, and hence the original MSR code and the new code are equivalent. In other words, the MDS property still holds so far, and the original data can be decoded as in (1) as well. In particular, when $d = k$, *i.e.*, $\alpha = 1$, there is only one segment in each block, and hence the expansion will work in the same way as Step 1 in Sec. V.

The reconstruction of the original MSR code can also be applied to the new code, with a proper adjustment. For example, with the original MSR code, we can reconstruct g_1F by multiplying a matrix G_1 with data encoded from d

existing blocks, *i.e.*,

$$G_1 \cdot \begin{bmatrix} v_{2,1}g_2F \\ \vdots \\ v_{d+1,1}g_{d+1}F \end{bmatrix}. \quad (3)$$

Similarly, we can expand G_1 and all vectors $v_{j,1}$, $j = 2, \ldots, d+1$ by multiplying each symbol with an identity matrix of size $N \times N$, and we use $\hat{G}_1$ and $\hat{v}_{j,1}$ to denote the expanded matrix and vectors, respectively. Hence the reconstruction can be represented as

$$\hat{G}_1 \cdot \begin{bmatrix} \hat{v}_{2,1}\hat{g}_2\hat{F} \\ \vdots \\ \hat{v}_{d+1,1}\hat{g}_{d+1}\hat{F} \end{bmatrix}. \quad (4)$$

It is easy to prove that (3) and (4) are equivalent as we replace every symbol inside (3) with a diagonal matrix, and hence the result in (4) will naturally be $\hat{g}_1\hat{F}$ if the result in (3) is g_1F. Therefore, we can still incur the optimal network traffic during reconstruction.

B. Symbol remapping

Similar to MSR code, we build a matrix $\hat{G}$ by concatenating $g_1, \ldots, g_n$ together, *i.e.*, $\hat{G} = \left[\hat{g}_1^T \ldots \hat{g}_n^T\right]^T$. For simplicity, we use $\hat{g}_{ij}$ to denote the j-th row in $\hat{g}_i$, $j = 1, \ldots, N\alpha$, $i = 1, \ldots, n$. Similar to Step 2 in Sec. V, we construct $\hat{G}_0$, a submatrix of $\hat{G}$, that is composed of nK rows in $\hat{G}$. In other words, in each block each segment offers $N\alpha \cdot \frac{k}{n} = K$ units. Assume that $\frac{N_0}{K_0}$ is the irreducible fraction of $\frac{K}{\alpha N} = \frac{k}{n}$. From every N_0 units we choose K_0 units in the way as how units are chosen in Step 2 in Sec.V. When $\alpha = 1$, there will be only $N_0 = N$ units in one segment, and hence the same $K_0 = K$ units will be chosen as in Step 2.

It can be proved that $\hat{G}_0$ is non-singular. We prove it by showing that all units corresponding to the rows chosen in $\hat{G}_0$ can decode the original data. Before expansion, we know that all segments in any k blocks can decode the original data, as MSR codes are MDS. After expansion, all units at the i-th row in any k blocks can decode the original data in the i-th row, $i = 1, \ldots, N$. As described above, from any N_0 units we choose K_0 segments in the same way as Step 2 in Sec. V. Hence, we can prove that in any row, there will be k out of the total n blocks that choose the units in the corresponding row, and hence the original data in this row can be decoded. Therefore, $\hat{G}_0$ will also be non-singular.

Therefore, by applying the technique of "symbol remapping" [19, Theorem 1] we can remap all rows in $\hat{G}$ that appear in $\hat{G}_0$ into unit vectors, which can compose an identity matrix, by multiplying the inverse of $\hat{G}_0$ on the right, *i.e.* $\hat{G}\hat{G}_0^{-1}$. The new code with $\hat{G}\hat{G}_0^{-1}$ as its generating matrix is equivalent to the original code with $\hat{G}$ as the generating matrix, since we can always linearly transform the original data by a non-singular matrix $\hat{G}_0$:

$$\hat{G}\hat{F} = \hat{G}\hat{G}_0^{-1} \cdot \hat{G}_0\hat{F}.$$

In other words, encoding $\hat{G}_0\hat{F}$ is equivalent to encoding $\hat{F}$ with the original code. Hence, by remapping $\hat{G}_0\hat{F}$ as the original data we can get a new code that is equivalent to the original one. As $\hat{G}_0\hat{F}$ can always be reverted back to $\hat{F}$, we can achieve the same performance of MDS property and network traffic during reconstruction as the original code, while the operation of decoding or reconstruction does not change. However, all rows in $\hat{G}_0$ becomes original data now in the output of the new code. Comparing with Step 2 and Step 3 in Sec. V, we can notice that they are simply the special case of this remapping when $d = k$, as after symbol remapping the unchosen rows in $\hat{G}$ actually become linear combinations of rows in $\hat{G}_0$.

C. Reordering

To make sure that the original data always appear at the beginning of each block, we can rotate all the rows in each segment in $\hat{G}\hat{G}_0^{-1}$ up in a way similar to Step 4 in Sec. V. In other words, for any integer $j \geq 0$, in block $i + jN_0$, all the rows in every N_0 units will be rotated up by i rows, $i = 1, \ldots, N_0$. In Fig. 4a we show an example of a block with $\alpha = 2$ and $N = 3$. In this block, the second and the third unit in each segment are chosen, and hence they are rotated up by one row and the first unit is put to the last row in the segment.

If $\alpha > 1$, there will be one more step that moves all data units in each block to the top of the block, such that data in all data units in a block are in the same sequence as the original data. In Fig. 4a, for example, we move all highlighted data units in the two segments upwards to the top of the block. This will not change the MDS property as only the sequence of data inside each block is changed.

During reconstruction, the coefficients in the matrix $\hat{v}_{i,j}$, which is used to encode the existing block for reconstruction, will also need to be reordered corresponding to the positions of units in the block. As shown in Fig. 4b, we divide $\hat{v}_{i,j}$ into 6 blocks and reorder these 6 blocks in the same order as in Fig. 4a. As reordering does not change the result of the linear transformation, we will get the same data as that before we perform symbol remapping. Hence, the rest of the reconstruction does not change, except that the rows in $\hat{G}_i$ should be reordered correspondingly, $i = 1, \ldots, n$. Hence, the amount of data transferred during reconstruction is the same as MSR codes, and the optimal network transfer is achieved.

VII. Achieving flexible parallelism

So far we have constructed Carousel codes that span original data across all the blocks. However, once any one block becomes unavailable, the request to read this

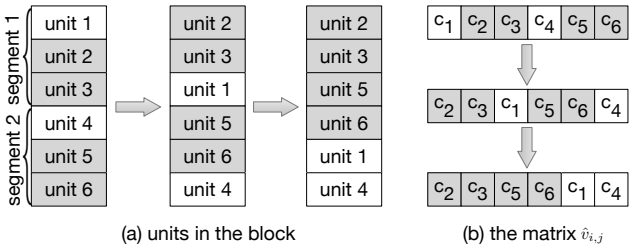

Figure 4. An example of reordering of units in a block and coefficients in $\hat{v}_{i,j}$, with 2 segments ($\alpha = 2$) which contains 3 units.

block will have to be served after reconstructing it. In this paper, we allow users to flexibly specify the degree of data parallelism by controlling the number of blocks that contain the original data. We use p, $k \leq p \leq n$, as a parameter to specify the degree of data parallelism in Carousel codes. In other words, we can construct (n, k, d, p) Carousel codes that encode data into n blocks in which p blocks contains the original data, while still achieving the MDS property that any k blocks can decode the original data and incur the optimal network traffic from d existing blocks during reconstruction. The construction we described previously is a special case of $n = p$.

To construct an (n, k, d, p) Carousel code, we first need to have an (n, k, d) MSR code. Then we expand the MSR code by splitting each segment into P units where $\frac{K}{P}$ is the irreducible fraction of $\frac{\alpha k}{p}$, *i.e.*, multiplying each element in the generating matrix with a $P \times P$ identity matrix. Still, we use $\hat{G}$ to denote the generating matrix of the expanded code. Then we make a submatrix of $\hat{G}$ by selecting K rows from the first p blocks, as in the construction of $(n = p, k, d, p)$ Carousel codes before. Hence, applying symbol remapping by multiplying the submatrix on the right of $\hat{G}$, we can get a new equivalent code that maintains all the properties of the MSR code while embedding the original data in the first p blocks.

As the original data will only stay in the first p blocks, it is only necessary to perform reordering of data in the first p blocks. Hence, during reconstruction, the coefficients of the vector that encode an existing block will only need to be reordered as well if this block is among the first p ones.

When we need to obtain all the original data, we can directly obtain them from the first p blocks if all of them are available. Moreover, when some blocks are not available, we show that we can always decode the original data by downloading $\frac{k}{p}$ blocks from each of any p blocks.

Without loss of generality, we assume that among the first p blocks, there are q available blocks, $q \leq p$. If $q = p$, all original data can be directly obtained without decoding. Hence, we only consider the case of $q < p$. From the q available blocks, we choose all data units. Then we use any $p - q$ other blocks that contain no original data to "replace" the unavailable blocks that contain original data. Specifically, if block i is not available ($i \leq p$), we will choose the same units from a replacing block as in Sec. VI-B. Hence, each one of the $q-p$ blocks replaces one unavailable block among the first p ones and offer the same amount of data to decode the original data.

We now prove that the data offered from the p blocks can decode the original data. As we know, the new code obtained after symbol remapping is equivalent to the original code. Hence, the chosen data can decode the original data if they can decode the original data before symbol remapping. As in the p blocks, the units are chosen in the same way as what we do in Sec. VI-B, they can decode the original data.

Therefore, we can see that an (n, k, d, p) Carousel code can be constructed such that p among all the n blocks contain the original data, and the original data can be decoded from any p blocks. Compared to systematic MDS codes that store all original data in k blocks, Carousel codes achieve a flexible trade-off between data parallelism and data availability.

VIII. Performance Evaluation

A. Real-World Implementation

To evaluate its performance, we have implemented Carousel codes in C++, where all operations, including encoding, decoding, and reconstruction, are performed by vector/matrix multiplications on a finite field of size 2^8. We have employed the Intel storage acceleration library (ISA-L) [32] when implementing finite field arithmetic operations. Since the construction of Carousel codes is developed from RS codes and MSR codes, and for comparison purposes, we have also implemented RS and MSR codes[2] using ISA-L.

In our implementation, we optimize the complexity of encoding operations of Carousel codes, by considering the sparsity of its generating matrix. This observation is based on the fact that in the construction of Carousel codes, each unit can be reconstructed by k units in other k blocks before reordering. In other words, each parity unit can be written as a linear combination of k data units. That means that the generating matrix of Carousel codes is sparse.

In Fig. 5, we demonstrate the generating matrices of $(3, 2)$ RS codes and $(3, 2, 2, 3)$ Carousel codes. The corresponding codes have been illustrated in Fig. 3, where the Carousel codes are constructed from the RS codes. We can see that with RS codes, each bit in parity blocks is a linear combination of two bits in original data. On the other hand, the size of the generating matrix of Carousel codes is expanded by three times from that of RS codes, because of the expansion we use in the construction. However, we can observe that the generating matrix of Carousel codes is a sparse matrix, *i.e.,* there are only two non-zero coefficients in rows that correspond to parity units. By considering this

[2]The construction of MSR codes in this paper is based on the product-matrix construction proposed by Rashmi *et al.* [19].

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}$$

(a) RS

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \end{bmatrix}$$

(b) Carousel

Figure 5. Comparison of the generating matrix of $(3, 2)$ RS codes and $(3, 2, 2, 3)$ Carousel codes.

sparsity, we can reduce the encoding complexity of parity data from a linear combination of six bits to two bits only, the same as the original RS codes. As for MSR codes and the corresponding Carousel codes, we can observe the same sparsity in the generating matrix of Carousel codes, offering the same opportunities to achieve a low encoding complexity.

Therefore, in our implementation we can firstly construct the generating matrix of Carousel codes as described above. When encoding data with this generating matrix, we can skip all multiplications with zero coefficients. The same method can be applied to optimize the complexity of decoding and reconstruction operations. We will see in the evaluation results how the size of the generating matrix affects the complexity and further the throughput of different operations, and how our implementation helps to save the complexity.

Based on our implementation of Carousel codes, we have developed a prototype on Hadoop 2.7.2 to store data with Carousel codes. We have first developed a tool that converts the original data into blocks encoded with Carousel codes. As such blocks now contain both original and parity data, we have developed a new `FileInputFormat` class that can know the boundary between the original data and parity data in each block by the parameters of Carousel codes, and read-only original data from each block when running Hadoop jobs.

B. Performance of encoding, decoding, and reconstruction

We first measured the performance of Carousel codes and compared with RS and MSR codes, by running encoding and decoding operations on Amazon EC2 instances of type `c4.4xlarge` with 16 CPU cores and 30 GB memory. All operations are run repetitively for 20 times and we show the mean as the results. In the evaluation, we run two Carousel codes constructed from RS codes ($d = k$) and MSR codes with $d = 2k-1$. The remaining parameters used in the plots are set as $n = 2k$ for all codes, $d = 2k-1$ for MSR codes, and $p = 2k$ for both Carousel codes.

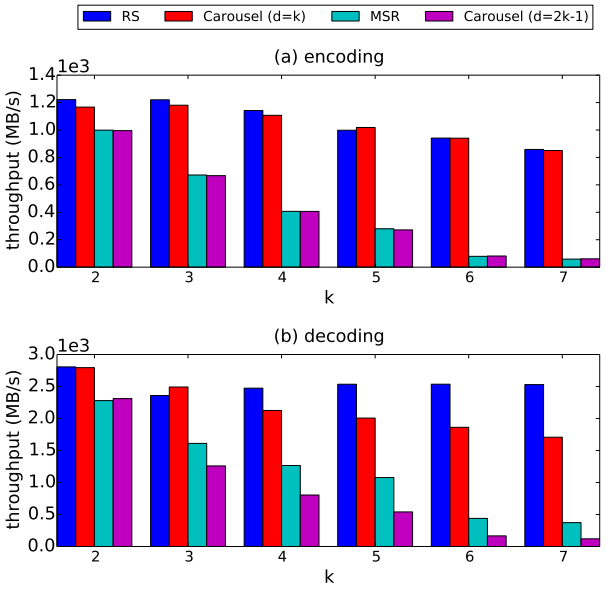

Figure 6. Comparison of the encoding and decoding throughput for various values of k with $n = 2k$ for RS codes, MSR codes ($d = 2k - 1$), and Carousel codes ($d = k$ and $d = 2k - 1$).

As shown in Fig. 6, we first observe that the size of the generating matrix can affect both the encoding and decoding throughput. Note that in the decoding operation with Carousel codes, we use only k blocks like RS and MSR codes even though we could have taken all data units from p blocks, for the purpose of fair comparison. We notice that with all values of k, the encoding and decoding throughput of MSR codes is significantly lower than RS codes. The gap keeps increasing with the value of k. This is because with $d = 2k - 1$, MSR codes split each block into k segments, such that the encoding complexity of each output bit will be increased by k times. From this observation, we can reasonably believe that Carousel codes will further limit the throughput as blocks are further divided into more units.

However, in Fig. 6a we can see that Carousel codes actually do not sacrifice encoding throughput compared to corresponding RS/MSR codes with the same value of k. We believe that this is because we take advantages of the sparsity of the generating matrix in Carousel codes. In other words, even though the size of the generating matrix is expanded, the complexity of encoding one output bit does not change.

When decoding data, we decode all original data from block 2 to block $k + 1$, including $k - 1$ data blocks and 1 parity block with RS/MSR codes. In other words, we test the scenario where one data block is not available, because decoding is not necessary with all data blocks available. From Fig. 6b, we find that Carousel codes becomes slower than the corresponding RS/MSR codes. The reason is that in this experiment, we have $\frac{k}{p} = \frac{1}{2}$, and hence half data in each block are parity data. When decoding data with RS/MSR

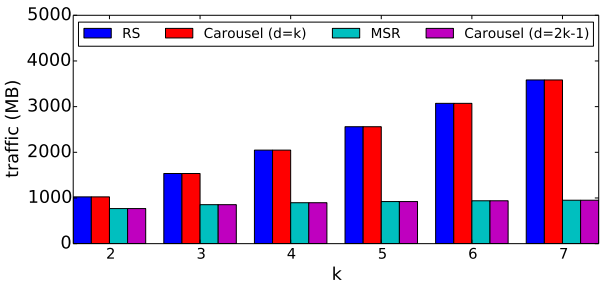

Figure 7. Comparison of network traffic incurred during reconstruction for various values of k with $n = 2k$ for RS codes, MSR codes ($d = 2k - 1$), and Carousel codes ($d = k$ or $2k - 1$, and $p = n$) .

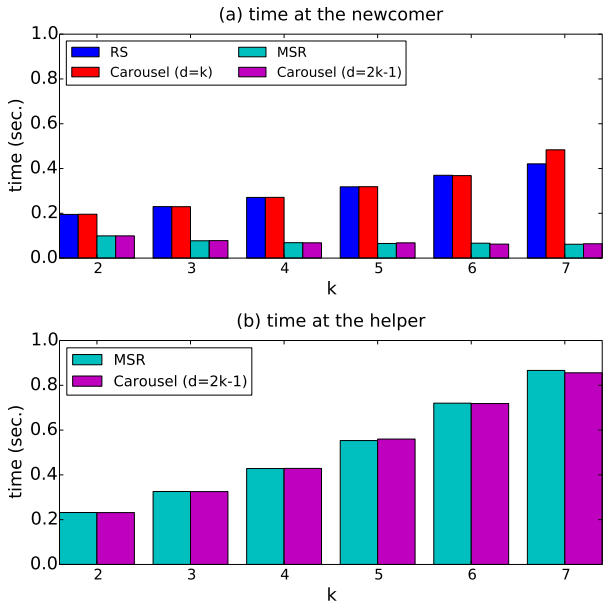

Figure 8. Completion time of reconstruction operations for various values of k with $n = 2k$ for RS codes, MSR codes ($d = 2k - 1$), and Carousel codes ($d = k$ or $2k - 1$, and $p = n$).

codes, we have $k-1$ data blocks already. In other words, we only need to compute data in the first block. With Carousel codes we can only get no more than half original data from any k blocks and have to compute the rest original data. A higher throughput can be achieved with Carousel codes if more than k blocks can be visited, which we leave as our future work.

As for the reconstruction, we first compare the traffic incurred by the corresponding erasure code during reconstruction. We measure the size of data obtained from existing blocks during reconstruction and calculate the total size from d newcomers. As expected, we can observe that the traffic incurred by Carousel codes during reconstruction is the same as RS or MSR codes with the same values of parameters.

Moreover, we measure the time spent at the side of existing blocks (called *helpers*) and the side of the replacement blocks (called *newcomers*) of various erasure codes. In this experiment, we encode original data into n blocks of size 512 MB and reconstruct the first block from d existing blocks (from block 1 to block $d + 1$). In Fig. 8, we demonstrate the time to finish the operation at helpers and newcomers. As RS codes do not require any operation at the side of helpers except sending data out, we only show the time of MSR codes and the Carousel codes with the same value of d, *i.e.*, $d = 2k - 1$. We can see that once again the Carousel codes can achieve comparable throughput at both helpers and newcomers with the corresponding RS/MSR codes.

C. Performance of running Hadoop jobs

In order to evaluate the performance of Hadoop jobs running on data encoded with Carousel codes, we run two representative performance benchmarks, `terasort` and `wordcount`, in a Hadoop cluster with 30 slave servers running on Amazon instances of type `r3.large` with 2 CPU cores, 15 GB memory, and 32 GB local SSD storage space. For each of these benchmarks, we generate a file of size 3 GB. With 512 MB as the block size in HDFS, such a file will be stored in 6 blocks. Hence, we use an $(n = 12, k = 6, d = 10, p)$ Carousel codes to encode this file where $p = 6, 8, 10$, or 12, and we end up having 12 blocks in HDFS. As a comparison, we also run the same jobs with data encoded with a $(n = 12, k = 6)$ systematic RS code. All benchmarks are run repetitively for 20 times and we show the mean with the 10th and 90th percentiles.

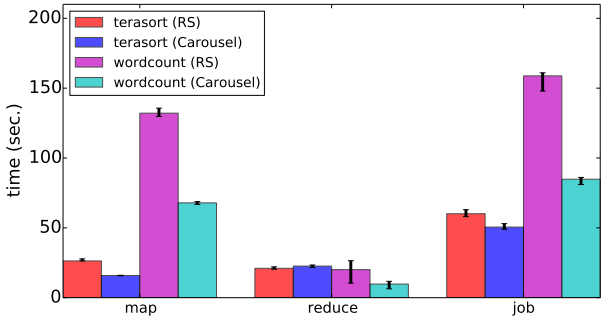

Figure 9. Comparison of Hadoop jobs running on data encoded with systematic RS codes and Carousel codes, with $n = 12, k = 6, d = 10$, and $p = 12$.

In Fig. 9 we show the average time spent in map and reduce tasks as well as the average job completion time to run the two benchmarks with $(12, 6, 10, 6)$ Carousel codes and $(12, 6)$ systematic RS codes. We can observe that the two benchmarks have very different characteristics. The major bottleneck of `wordcount` is at map tasks, while `terasort` has a significant computational overhead at the reduce tasks as well. As shown in Fig. 9, the average time of map tasks can be significantly saved with Carousel codes for both `wordcount` (by 46.8%) and `terasort` (by 39.7%). As $p = 12$ and $k = 6$, each job with systematic RS codes can

run 6 map tasks in parallel, and each job with Carousel codes can launch 12 simultaneous map tasks. Hence, in theory the saving of time in the map task is 50%, and more saving can be expected with higher storage overhead permitted. We can observe that in `wordcount` the actual saving is very close to the theoretical optimum. Because of this, the job completion time of `wordcount` can be significantly reduced by Carousel codes as most time are spent by map tasks. On the other hand, though the reduce task takes a similar amount of time as the map task in `terasort`, the job completion time can still be saved by 15.9%.

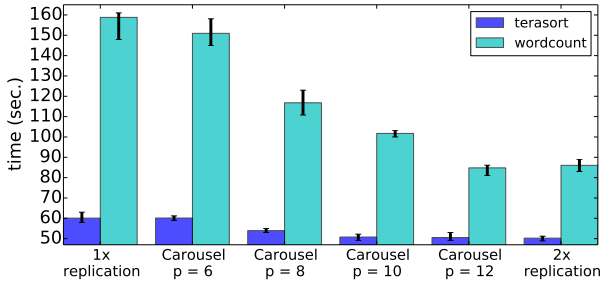

Figure 10. Comparison of Hadoop jobs running on data encoded with $(12, 6, 10, p)$ Carousel codes with various values of p, as well as on data with 1-way and 2-way replication.

In Fig. 10, we further compare the performance of job completion times using Carousel codes with various values of p. We can see that with p increasing from 6 to 12, the job completion time of both `terasort` and `wordcount` gradually decreases. When $p = 6$, the performance of Carousel codes is similar to the 1-way replication and $(12, 6)$ systematic RS codes in Fig. 9, since they all have 6 blocks with the original data in HDFS. When the value of p reaches 12, the performance of Carousel codes becomes comparable with the 2-way replication, which cannot be achieved with systematic MDS codes. Meanwhile, in this case Carousel codes cost much less storage space and are able to tolerate more failures than the 2-way replication.

D. Performance of data access in HDFS

Now we evaluate the decoding performance of Carousel codes with more than k blocks. We measure this performance by downloading the 3 GB file used to run the Hadoop benchmarks above from HDFS in the same Hadoop cluster. In order to emulate the read throughput of the enterprise hard disk [33], we limit the read throughput of datanodes in Hadoop by 300 Mbps. In this experiment, we encode data with a $(12, 6)$ systematic RS code or a $(12, 6, 10, 10)$ Carousel codes, or store data in 3-way replication.

For replicated data, we use the built-in `hadoop fs -get <src> <dst>` command to retrieve the file, which downloads each block from every datanode sequentially. For the systematic RS codes and Carousel codes, we write a program to download the original data from different blocks

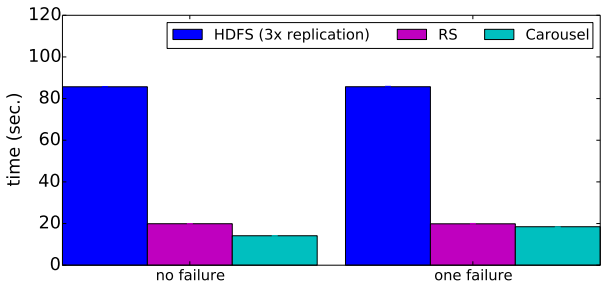

Figure 11. Comparison of the time of retrieving a 3 GB file from HDFS, with data encoded with $(12, 6)$ systematic RS codes and $(12, 6, 10, 10)$ Carousel codes, and replicated three times.

in parallel. If some block that contains original data is not available, it will obtain parity data from another block and decode the original data simultaneously.

In the results shown in Fig. 11, we can find that by obtaining data from different blocks in parallel, a significantly amount of time can be saved. In this case, by extending the degree of data parallelism from 6 to 10, Carousel codes can further save the time by 29.0%. In addition, we have introduced failures into HDFS by randomly removing one block that contains original data. At this time, the saving of time with the Carousel code is compromised, due to its higher decoding complexity than RS codes. However, it still spends less time than the RS code and 75.4% less time than the built-in Hadoop command.

IX. Conclusion

Systematic MDS erasure codes have significantly saved storage overhead without sacrificing failure tolerance. However, the degree of data parallelism of data encoded with such codes is limited by the number of data blocks. In this paper, we present *Carousel codes* that can flexibly configure the degree of data parallelism to all the blocks at most. We have shown that Carousel codes can achieve the optimal storage overhead by satisfying the MDS property, and the optimal network transfer to reconstruct the unavailable block as well. Through extensive experiments running in a Hadoop cluster on Amazon EC2, we demonstrate that most operations of Carousel codes, such as encoding and reconstruction, are comparable with existing systematic MDS codes, while the time to run Hadoop jobs on the encoded data and to access the original data can both be significantly reduced.

Acknowledgment

This research is partially supported by the SAVI NSERC Strategic Networks project, two NSERC Strategic Partnership Projects, and an NSERC Discovery Research Program.

References

[1] D. Borthakur, "HDFS Architecture Guide," http://hadoop.apache.org/common/docs/current/hdfs_design.html.

[2] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. ul Haq, M. I. ul Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas, "Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency," in *Proc. of ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[3] 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. VLDB Endowment*, 2013.

[4] I. Reed and G. Solomon, "Polynomial Codes Over Certain Finite Fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[5] Z. Zhang, A. Wang, K. Zheng, U. M. G., and V. B, "Introduction to HDFS Erasure Coding in Apache Hadoop," *Cloudera Engineering Blog*, Sept. 2015, https://blog.cloudera.com/blog/2015/09/introduction-to-hdfs-erasure-coding-in-apache-hadoop/.

[6] 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. USENIX Annual Technical Conference (USENIX ATC)*, 2012.

[7] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran, "Network Coding for Distributed Storage Systems," *IEEE Tran. Inform. Theory*, vol. 56, no. 9, Sept. 2010.

[8] "Partitioning Your Job into Maps and Reduces," https://wiki.apache.org/hadoop/HowManyMapsAndReduces.

[9] R. Li, Z. Zhang, K. Zheng, and A. Wang, "Progress Report: Bringing Erasure Coding to Apache Hadoop," *Cloudera Engineering Blog*, Feb. 2016, http://blog.cloudera.com/blog/2016/02/progress-report-bringing-erasure-coding-to-apache-hadoop/.

[10] "Erasure Code Support," *Sheepdog Wiki*, https://github.com/sheepdog/sheepdog/wiki/Erasure-Code-Support.

[11] B. Fan, W. Tantisiriroj, L. Xiao, and G. Gibson, "Diskreduce: Replication as a Prelude to Erasure Coding in Data-intensive Scalable Computing," *PDL Technical Report (CMU-PDL-11-112)*, 2011, http://www.pdl.cmu.edu/PDL-FTP/HECStorage/CMU-PDL-11-112.pdf.

[12] "HDFS-RAID," http://wiki.apache.org/hadoop/HDFS-RAID.

[13] J. Arnold, "Erasure Codes with OpenStack Swift –Digging Deeper," https://swiftstack.com/blog/2013/07/17/erasure-codes-with-openstack-swift-digging-deeper/, July 2013.

[14] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly, "The Quantcast File System," *Proc. the VLDB Endowment*, vol. 6, no. 11, pp. 1092–1101, 2013.

[15] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan, "Availability in Globally Distributed File System," in *Proc. USENIX Operating Systems Design and Implementation*, 2010.

[16] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A Solution to the Network Challenges of Data Recovery in Erasure-Coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster," in *Proc. USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2013.

[17] ——, "A "HitchHiker's" Guide to Fast and Efficient Data Reconstruction In Erasure-Coded Data Centers," in *Proc ACM SIGCOMM*, 2014.

[18] M. Xia, M. Saxena, M. Blaum, and D. A. Pease, "A Tale of Two Erasure Codes in HDFS," in *Proc. USENIX Conference on File and Storage Technologies (FAST)*, 2015, pp. 213–226.

[19] K. V. Rashmi, N. B. Shah, and P. V. Kumar, "Optimal Exact-Regenerating Codes for Distributed Storage at the MSR and MBR Points via a Product-Matrix Construction," *IEEE Trans. Inform. Theory*, vol. 57, no. 8, pp. 5227–5239, 2011.

[20] I. Tamo, Z. Wang, and J. Bruck, "Zigzag Codes: MDS Array Codes With Optimal Rebuilding," *IEEE Transactions on Information Theory*, vol. 59, no. 3, pp. 1597–1616, 2013.

[21] E. En Gad, R. Mateescu, F. Blagojevic, C. Guyot, and Z. Bandic, "Repair-Optimal MDS Array Codes Over GF(2)," in *Proc. IEEE International Symposium on Information Theory Proceedings (ISIT)*, 2013, pp. 887–891.

[22] Y. Wang, X. Yin, and X. Wang, "MDR Codes: A New Class of RAID-6 Codes With Optimal Rebuilding and Encoding," *IEEE Journal on Selected Areas in Communications*, vol. 32, no. 5, pp. 1008–1018, May 2014.

[23] R. Li, P. P. C. Lee, and Y. Hu, "Degraded-First Scheduling for MapReduce in Erasure-Coded Storage Clusters," in *Proc. 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2014, pp. 419–430.

[24] J. Li and B. Li, "Zebra: Demand-Aware Erasure Coding for Distributed Storage Systems," in *Proc. IEEE/ACM International Symposium on Quality of Service (IWQoS)*, 2016.

[25] Y. Hu and D. Niu, "Reducing Access Latency in Erasure Coded Cloud Storage With Local Block Migration," in *Proc. IEEE INFOCOM*, 2016.

[26] L. Xu and J. Bruck, "X-code: MDS Array Codes with Optimal Encoding," *IEEE Transactions on Information Theory*, vol. 45, no. 1, pp. 272–276, 1999.

[27] J. S. Plank, "The Raid-6 Liberation Code," *International Journal of High Performance Computing Applications*, 2009.

[28] Y. Cassuto and J. Bruck, "Cyclic Lowest Density MDS Array Codes," *IEEE Transactions on Information Theory*, vol. 55, no. 4, pp. 1721–1729, 2009.

[29] Y. Fu, J. Shu, and X. Luo, "A Stack-Based Single Disk Failure Recovery Scheme for Erasure Coded Storage Systems," in *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*. IEEE, 2014, pp. 136–145.

[30] Y. Fu and J. Shu, "D-Code: An Efficient RAID-6 Code to Optimize I/O Loads and Read Performance," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2015, pp. 603–612.

[31] N. B. Shah, K. V. Rashmi, P. V. Kumar, and K. Ramchandran, "Interference Alignment in Regenerating Codes for Distributed Storage: Necessity and Code Constructions," *IEEE Trans. on Inform. Theory*, vol. 58, no. 4, pp. 2134–2158, 2012.

[32] "Intel Storage Acceleration Library," https://01.org/intel%C2%AE-storage-acceleration-library-open-source-version.

[33] "Seagate Enterprise Performance 15K HDD," http://www.seagate.com/ca/en/internal-hard-drives/enterprise-hard-drives/hdd/enterprise-performance-15k-hdd/#specs.