Demand-aware Erasure Coding for Distributed Storage Systems

Jun Li, *Member, IEEE,* and Baochun Li, *Fellow, IEEE*

Abstract—Distributed storage systems provide cloud storage services by storing data on commodity storage servers. Conventionally, data are protected against failures of such commodity servers by replication. Erasure coding consumes less storage overhead than replication to tolerate the same number of failures and thus has been replacing replication in many distributed storage systems. However, with erasure coding, the overhead of reconstructing data from failures also increases significantly. Under the ever-changing workload where data accesses can be highly skewed, it is challenging to deploy erasure coding with appropriate values of parameters to achieve a well trade-off between storage overhead and reconstruction overhead.
In this paper, we propose *Zebra*, a framework that encodes data by their demand into multiple tiers that deploy erasure codes with different values of parameters. Zebra automatically determines the number of such tiers and dynamically assigns erasure codes with optimal values of parameters into corresponding tiers. With Zebra, a flexible trade-off between storage overhead and reconstruction overhead is achieved with multiple tiers. When demand changes, Zebra adjusts itself with a marginal amount of network transfer. We demonstrate that Zebra can work with two representative families of erasure codes in distributed storage systems, Reed-Solomon codes and local reconstruction codes.

Index Terms—distributed storage system, demand skewness, erasure coding, reconstruction, storage overhead, Reed-Solomon code, local reconstruction code

1 Introduction

Distributed storage systems [1], [2], [3] provide fundamental storage services in the cloud, such as Amazon's EBS (Elastic Block Store) and S3, and Windows Azure Storage from Microsoft. In addition, they support various cloud services by hosting data required by such services. For example, Amazon's EC2 service uses EBS to host data in virtual machines. Similarly, Windows Azure storage offers data storage to other cloud services in Windows Azure.

Typically, a distributed storage system stores a massive amount of data over a large number of commodity servers. Due to the nature of the commodity hardware, as well as other reasons such as software glitches, power failures, upgrade and maintenance operations, and even overload, servers in distributed storage systems are subject to frequent failures on a daily basis. For example, in a Facebook cluster with 3000 servers, 50 failures that lead to data unavailability can be expected every day [4].

As failures are norm rather than exceptions in distributed storage systems, redundant data must be stored such that a specific number of failures can be tolerated without loss of data. The naive and the conventional way to store redundant data in a distributed storage system is replication, *i.e.*, saving multiple copies of the same data on different servers. Saving N copies on N servers (N-way replication) can tolerate up to $N-1$ server failures without loss of data. However, replication is very expensive in terms of its storage overhead, especially for data at a petabyte scale. For example, with 3-way replication, to store 10 PB of data, we need to spend additional 20 PB to store the other two copies.

Because of the high storage overhead of replication, distributed storage systems are migrating from replication to erasure coding [5], [6], [7], [8], [9], as erasure coding can tolerate the same or an even higher number of failures with much less storage overhead [10]. Using Reed-Solomon (RS) codes [11], the most common choice of erasure codes in distributed storage systems, as an example, we can encode r parity blocks from k data blocks of the same size, such that any k among these $k+r$ blocks can recover the original data. In other words, at most r server failures can be tolerated if all $k+r$ blocks are stored on different servers. When $k=10$ and $r=4$, we can tolerate at most 4 failures with only 1.4x storage overhead. With the increasing of k, we can achieve even lower storage overhead while tolerating the same number of failures.

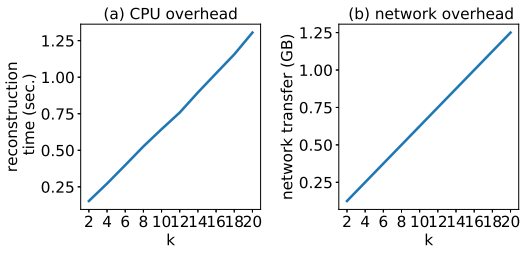


Fig. 1. CPU and network overhead to reconstruct one block with RS code $(r=2)$, where each block contains 64 MB.

However, RS codes can incur significantly higher overhead when we need to reconstruct an unavailable block. When a block becomes unavailable after a server failure, we need to reconstruct it on another existing server to maintain

- *Jun Li is with the School of Computing and Information Sciences, Florida International University. Email: junli@cs.fiu.edu*
- *Baochun Li is with the Department of Electrical and Computer Engineering, University of Toronto. Email: bli@ece.toronto.edu*

the level of failure tolerance. With a $(k = 10, r = 4)$ RS code, for example, if one block is not available, we need to obtain 10 blocks to reconstruct it. Therefore, when data are not available due to a server failure, a read request of such data needs to be performed by *degraded read* [1] that reconstructs them from blocks on other available servers. Higher overhead of reconstruction can lead to a higher access latency during the degraded read. With typical values of k, the network transfer incurred by reconstruction can be huge: in a Facebook's cluster the daily median of top-of-rack network transfer incurred by reconstruction can be as much as 180 TB [12]. Fig. 1 illustrates the overhead of time and network transfer to reconstruct one block of 64 MB with RS codes[2]. We can see that both the time and network transfer incurred by reconstruction increases linearly with k. In other words, a smaller value of k means less reconstruction overhead, leading to less network transfer for data reconstruction as well as lower latency for degraded read.

Therefore, in a distributed storage system, while we desire for a small value of k to achieve low reconstruction overhead, it takes a large value of k to save storage overhead. Currently, most distributed storage systems deploy only one erasure code to encode data, optimized either for storage overhead or reconstruction overhead. However, in practical distributed storage systems, the demand of data can be highly skewed. In Fig. 2a, we show the demand of data in a workload trace measured from a Facebook cluster [14]. This workload contains more than 10^4 files while we only show the 100 most demanded files in the figure, with all the rest having no more than 11 visits. We can see that a very small portion of data are highly demanded while the rest are barely touched. Even among blocks belonging to the same file, it has also been observed that the demand to such blocks can be skewed when running data analytical jobs [15], [16]. With only one erasure code we cannot accommodate cold data with low storage overhead while achieving low reconstruction overhead for hot data. Moreover, the demand for the data can change over time dynamically. We sort files in the trace by their demand, such that the file with higher overall demand has the lower index. As shown in Fig. 2b, while file No. 1 and 3 have consistent demand over time, the other three files only have transient high demand at some time and have no visit any other time. Therefore, it is also challenging to find an erasure code that can work well adaptively with the ever-changing demand.

In this paper, we propose *Zebra*, a novel framework for distributed storage systems deploying erasure codes. According to the demand of data, Zebra can split data into multiple *tiers* such that data in different tiers are encoded with erasure codes with different values of parameters. Although existing works [17], [18], [19] also features such tiered architectures, they require static configurations of parameters in each tier. Nevertheless, Zebra offers the flexibility to dynamically configure parameters of erasure codes deployed in each tier, and even the number of tiers.

1. As the result of a degraded read can also be used to recover an unavailable block on a replacement server, we assume that there is no or little need to recover a block individually and only consider the reconstruction overhead in the degraded read in this paper.

2. The time of the reconstruction is measured using the zfec library [13] running on an Intel Core i7 processor.

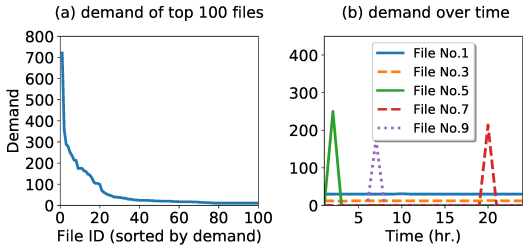


Fig. 2. The demand skewness of files in a Facebook workload trace.

By solving geometric programming problems, Zebra determines parameter values of erasure codes in each tier, such that hot data can be reconstructed with low overhead and cold data can enjoy low storage overhead at the same time. Meanwhile, Zebra can also assign data to the best fitting tier by their demand, so as to minimize reconstruction overhead. When demand changes, Zebra can dynamically migrate data accordingly into different tiers or even change parameter values of tiers, while carefully controlling the overhead in the migration. For the hot data in the tier with the highest demand, Zebra can be further extended to achieve better load balance under a high volume of demand. Besides RS codes, we show that Zebra can also be applied with local reconstruction codes [8], another kind of erasure codes for distributed storage systems with lower reconstruction overhead.

We run simulations under various workload traces to evaluate the performance of the Zebra framework. We demonstrate the performance of Zebra working with RS codes and local reconstruction codes. The evaluation results show that Zebra can reduce reconstruction overhead, especially by 89.5% for the hot data. Moreover, the hot data also enjoy less demand by 63.2% due to the better load balance in Zebra. The cold data, on the other hand, will incur less storage overhead to maintain their tolerance against failures. With the ever-changing workload, we demonstrate that the network transfer of migration can be well controlled, such that it occupies no more than 12.6% of the network transfer of demand in the worst case.

2 Motivation and Examples

In this section, we present the general idea that motivates the design of the Zebra framework. We assume that in a distributed storage system, data are stored in blocks of 64 MB, where we compute 1 parity block from every 3 data blocks. In other words, a $(k = 3, r = 1)$ RS code is deployed in this distributed storage system.

As a toy example in Fig. 3a, assume that we have 6 data blocks in total, *i.e.*, $A_1 - A_3$ and $B_1 - B_3$. We can encode A_1, A_2, and A_3 into one parity block P_1, and B_1, B_2, and B_3 into the other parity block P_2. We then call each three data blocks and the corresponding parity block as one *stripe*, such as A_1, A_2, A_3, and P_1. With six data blocks in Fig. 3a, we thus have two stripes. More stripes can be added this way if there are more data blocks. Such two stripes fall into one single tier because they are encoded from the same $(3, 1)$ RS code. In this way, we can tolerate a failure of

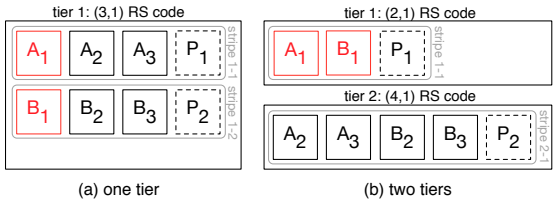


Fig. 3. Comparison of data encoded in one and two tiers of RS codes. A_i and B_i are data blocks, $i = 1, 2, 3$. P_1 and P_2 are parity blocks encoded by corresponding RS codes. Colored red, A_1 and B_1 are hot blocks with 100 visits per unit of time, while other blocks are cold with only 10 visits.

any single block, with 1.33x storage overhead. When one of them becomes unavailable, we need to obtain the other three blocks in the same stripe to reconstruct it until it is available on a replacement server. Therefore, the amount of data to be read from disks and transferred through network during degraded read is 3×64 MB $= 192$ MB.

However, as we know that the demand to different blocks can be significantly skewed, we assume that the demand of the 6 data blocks is not equal to each other, where A_1 and B_1 are highly demanded with 100 visits per unit of time, and all other four blocks are visited by 10 times per unit of time. Under the Zebra framework, we can then encode these 6 data blocks into two tiers, *i.e.,* we encode the two hot blocks with a $(2, 1)$ RS code and the other four blocks with a $(4, 1)$ RS code. Therefore, there is only one parity block in each stripe, and we still have 2 parity blocks in total, maintaining the same storage overhead as above. Meanwhile, we can still tolerate the failure of any single block. As server failures are independent to the demand of data (especially when the demand of data are balanced on servers), each request will have an equal chance to meet a failed server and such a request will have to be served by degraded read. Though the cold blocks need to obtain 4 blocks to reconstruct, the hot blocks have dominant demand with much less blocks to obtain. Hence, this time we can significantly reduce the average number of blocks to visit per unit of time. On average per degraded read, we need to have $\frac{(2\times2\times100+4\times4\times10)\times64\text{ MB}}{2\times100+4\times10} = 149.3$ MB to read, saving corresponding disk I/O and network transfer by 22.2%. For the two hot blocks in particular, their reconstruction overhead can simply be reduced from 3 to 2 blocks, *i.e.*, 33.3% reduction in time, disk I/O, and network transfer.

In this example, we deploy two tiers of RS codes. However, Zebra is not limited to only two tiers. In fact, we can flexibly deploy any number of tiers inside the Zebra framework with different parameter values of RS codes, where the number of tiers and the parameter values can be efficiently calculated according to the demand of data and the requirements of storage overhead and failure tolerance. Zebra can automatically assign data into corresponding tiers by their demand. Besides RS codes, we further show that Zebra can work with local reconstruction codes.

3 Related Work

At the scale of petabyte storage, erasure coding has become more and more attractive to distributed storage systems because of its low storage overhead and high failure tolerance. Hence, many distributed storage systems, such as HDFS [9], [18], Openstack Swift [5], Google file system [6], and Windows Azure storage [8], are moving towards or have deployed erasure coding as an alternative to replications, where in most cases RS codes are chosen by these distributed storage systems. However, they all choose only one kind of erasure code with fixed parameters. It is then hard to trade well between storage overhead and reconstruction overhead of erasure codes, under the dynamical workload with highly skewed data demand [15], [16], [20].

Traditional RS codes can incur high overhead of reconstruction when some of the data are not available due to failures inside distributed storage systems [12], [21]. There has been a growing attention of improving the overhead of reconstruction of erasure codes. For example, local repairable codes [22] can achieve low reconstruction overhead by allowing unavailable data to be reconstructed from a small number of other servers. Similar ideas have been applied in the design of other erasure codes [8], [21], [23], [24], [25]. On the other hand, another family of erasure codes, called regenerating codes, are designed to achieve the optimal network transfer in the reconstruction [26]. All these erasure codes, however, are optimized for their own objectives over all encoded data, unaware of that the demand of data can be high skewed. As data with different demand may have different performance objectives, applying one single erasure code over all data may not achieve all their objectives. Different from these erasure codes, in Zebra we propose to dynamically assign data into multiple tiers by their demand so as to achieve a flexible trade-off between storage and reconstruction overhead.

Some distributed storage systems, such as HDFS [27], allow data to be stored under a tiered architecture, where data in different tiers are stored by different erasure codes or replication with preconfigured parameters and can be automatically migrated between different tiers [17], [18], [19]. However, all these systems require users to configure the parameter of erasure code in each tier statically, and they cannot well adapt themselves to the ever-changing workload. Besides, as in different tiers the storage overhead incurred by the corresponding erasure codes will also be different, it is hard to control the overall storage overhead. The Zebra framework, however, does not need to specify parameters of each tier or even the number of tiers. According to the demand of data, Zebra can configure itself flexibly, where only the overall storage overhead and failure tolerance need to be manually specified.

Different from the above works where the reconstruction overhead is evaluated in terms of network traffic or disk I/O, a tree-structured topology can be created, which routes the traffic through the edges of the tree and alleviates the bottleneck of sending data from existing servers to the replacement server [28], [29], [30]. The purpose of such works, instead, is to save the time of reconstruction. Such works can be applied into our framework without affecting the network overhead during reconstruction, and thus we focus on network overhead only in this paper.

4 System Model

In this paper, we assume that in a distributed storage system, data are stored in blocks with the same size. This

is a common practice in distributed storage systems [3].

Assume that we have N blocks in total, and each block B_i is associated with demand of d_i visits per unit of time, $i = 1, \ldots, N$. We also assume that any r block failures should be tolerated without data loss. Hence, if RS codes are deployed, in the Zebra framework each block will be encoded with a (k_i, r) RS code, where we call k_i as the *rank* of block B_i. For convenience, we let $D = (d_1, \ldots, d_N)$, and $K = (k_1, \ldots, k_N)$. We also want to control the overall storage overhead such that the overall storage space consumed is no more than C times of the original data.

In this model, we assume that the erasure codes deployed in the distributed storage system should be systematic. In other words, a (k, r) systematic erasure code computes $k + r$ blocks from original data, in which k blocks are the same as the original data, *i.e.*, data blocks. The other r blocks are known as parity blocks. Such $k + r$ blocks belong to the same stripe and are stored on $k + r$ different servers. From the k data blocks in each stripe, we can always directly obtain any data block *without* decoding as long as it is available. Hence, we can assume that all demand will go directly to the corresponding data blocks rather than parity blocks, unless the demanded data blocks are unavailable. Moreover, when some data block is not available, only reconstruction, instead of decoding, needs to be performed.

We now use this model to represent the way to encode data in one or multiple tiers with erasure codes. Fig. 3 (without loss of generality, we can rewrite A_i as B_{i+3}, $i = 1, 2, 3$) illustrates two examples of this model with systematic RS codes, where $N = 6$, $C = \frac{4}{3}$, and $D = \{100, 10, 10, 100, 10, 10\}$. In Fig. 3a, $k_i = 3$ for all i, *i.e.*, all blocks are encoded in one tier with a $(3, 1)$ RS code. On the other hand, in Fig. 3b, we have $K = \{2, 4, 4, 2, 4, 4\}$ such that the six blocks are encoded into two tiers with a $(2, 1)$ and a $(4, 1)$ RS code.

In this way, we can see that the number of tiers in the model does not need to be explicitly defined as blocks with the same rank can be categorized into the same tier. Once the rank of each block is equal to each other, for example, it becomes the conventional case of only one tier. Hence, we are not limited by a given number of tiers, and we can easily change the number of tiers when demand changes.

In this paper, our objective is to minimize the average reconstruction overhead of degraded read, with respect to the constraint of the overall storage overhead C. As shown in Fig. 1, the reconstruction overhead of a block increases linearly with its rank. We assume that each server has the same chance to be unavailable. Thus, the chance of degraded read should also increase linearly with the demand of corresponding block. Combining all demand together, we can define the average reconstruction overhead as $\frac{\sum_{i=1}^{N} d_i k_i}{\sum_{i=1}^{N} di}$. As the demand D is already given, it is equivalent to minimize $\sum_{i=1}^{N} d_i k_i = D \cdot K$, which we call as the overall reconstruction overhead.

Besides the overall reconstruction overhead, we need to control the overall storage overhead, which can be computed with D and K in the model. Since each block has the same size, we assume that the size of each block is 1 for convenience. Thus, the storage space consumed to store block B_i and its parity is $1 + \frac{r}{k_i}$. The sum of storage space of all blocks is $N + \sum_{i=1}^{N} \frac{r}{k_i}$. Since the total storage space we can use under the constraint of the overall storage overhead C is CN, we can write this constraint as $\sum_{i=1}^{N} \frac{1}{k_i} \leq \frac{(C-1)N}{r}$.

Therefore, we can solve K to minimize the overall reconstruction overhead with respect to the storage overhead by the following integer geometric programming problem.

$$\min \quad D \cdot K \tag{1}$$

$$\text{s.t.} \quad \sum_{i=1}^{N} \frac{1}{k_i} \leq \frac{(C-1)N}{r}, \tag{2}$$

$$k_i \in \mathbb{Z}^+. \tag{3}$$

Typically, a geometric programming problem can be easily converted to a convex optimization problem and solved efficiently [31]. However, there are some other issues that makes it challenging to achieve a practical solution by directly solving this problem.

First, in a distributed storage system there can be an extremely large number of data blocks. For example, in HDFS the default block size is 64 MB. If there are 1 PB of data stored in HDFS, there will be over 10^7 blocks in total. No solver of convex optimization problems can solve our model in a reasonable amount of time. Besides, the geometric programming problem in (1)-(3) is an integer programming problem. This also significantly increase the complexity to solve it.

Second, the solution solved from (1)-(3) is an offline solution. In other words, we need to know the demand in advance before we can get the optimal solution, which makes it impractical. We need to find an online algorithm that can solve K in advance of the demand.

Third, given a solution of this problem, we can't even guarantee that it is feasible. For example, if the solution is $K = \{8, 8, 8, 8, 8, 8\}$, we will need to encode 6 blocks with an $(8, r)$ erasure code. This is impossible without rearranging the data blocks into a smaller size. Variable-size blocks, however, will incur significantly more complexity to manage data inside the distributed storage system. In this paper, we retain the assumption of fixed-size blocks and manage to encode data with such solutions without incurring much additional overhead.

In the rest of the paper, we introduce the Zebra framework that solves these practical issues efficiently and then show the encoding scheme under the Zebra framework. We start from introducing Zebra with RS codes, and then extend Zebra to make it work with local reconstruction codes.

5 Zebra Framework

5.1 Limiting the complexity

In the Zebra framework, we propose a few heuristics to compute ranks of blocks efficiently. We start from temporarily removing the integer constraint (3) and resolving the complexity issues of the non-integer geometric programming problem in (1)-(2) by studying its property.

Without loss of generality, we assume that in D, $d_i \geq d_j$ if $i > j$. In other words, we sort the element in D in a non-ascending order. In this way, an optimal solution of K in (1)-(2) should also be in non-descending order.

We prove this property by contradiction. Assume that there exist i and j in an optimal solution of (1)-(2) such

that $k_i > k_j$ where $i < j$, and then the reconstruction overhead of block i and j is $d_i k_i + d_j k_j$. If $d_i > d_j$, it is straightforward that $d_i k_i + d_j k_j > d_i k_j + d_j k_i$. Therefore, we can get a solution with even lower overall reconstruction overhead by exchange the rank of block B_i and B_j, which is contradictory to the assumption that the original solution is optimal. On the other hand, if $d_i = d_j$, we can assign a new rank, $\frac{2k_i k_j}{k_i + k_j}$, to both block B_i and B_j to get a lower overall reconstruction overhead, while still satisfying the condition in (2). This is also contradictory to the assumption that the original solution is optimal.

From this property, we can directly get a corollary that $k_i = k_j$ if $d_i = d_j$. In other words, if two block have the same demand, they will also have the same rank in the optimal solution. Inspired by this property, we can refine the original model to significantly decrease the complexity to solve it, by reducing the number of variables to solve.

First, in this paper, we assume that all blocks of the same file should have the same or similar demand in a distributed storage system. Notice that if all blocks of the same file have the same demand, this step will not hurt the optimality of the solution. In practice, if the demand of blocks in a single file is not the same, we can use the demand of the hottest block of the file as the demand of the file. The intuition of this assumption is that typically a distributed storage system that stores large-size files will have distributed data processing system running upon it, such as Hadoop and Spark, which will visit each block of the file distributively. Therefore, once a file is visited, all of its blocks will be visited first. Although in some cases [15], [16] that some data in a block will be selected for future processing, leading to different workload in different blocks eventually, we focus on heterogeneous demand on the file level instead of the block level in this paper.

Second, we extend this property by assuming that blocks with similar demand will also have similar ranks. Thus, we can classify the demand of all blocks into discrete categories. The simplest way is to set a parameter t where any demand that falls into the interval $(tx - t, tx]$ will be approximated as tx, $\forall x \in \mathbb{Z}^*$. Hence, files with similar demand can be temporarily merged into the same one when calculating their ranks, and thus we can further reduce the complexity to solve the model. For the cold data, this is especially useful, as cold data typically occupy a very large portion of all the data, yet with similar demand of very little values. Thus, we can quickly categorize cold data into few intervals, and then thousands of files can be grouped into few ones.

We run the simulation on the workload from Facebook that we show in Fig. 2. In this workload, there are 15565 files in total. If we store data into blocks of 64 MB, there will be 1.8×10^7 blocks. The result in Fig. 4 shows that we can reduce the number of files into 55 even when $t = 1$. Notice that when $t = 1$, we actually don't merge files unless their demand is exactly the same. When we increase the value of t, we can even further reduce the complexity of solving k.

5.2 Solving the ranks

After the two steps described above, we can refine the model in (1)-(2) such that there are n files, where each file F_i is associated with size w_i and demand d_i. Each file will be encoded with a (k_i, r) RS code, and the overall storage overhead should be no more than C. Hence, the problem to solve the optimal K can be redefined as

$$\min \quad \sum_{i=1}^{n} w_i d_i k_i \tag{4}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} \frac{w_i}{k_i} \leq \frac{(C-1)N}{r} \tag{5}$$

$$k_i > 0, \forall i. \tag{6}$$

This is still a geometric programming problem. Notice that here we tentatively remove the constraint (3) that each k_i must be an integer. In the Zebra framework, we will solve this problem first, and then round k_i to a nearby integer. For example, we can always round k_i to its ceiling $\lceil k \rceil$. In this way, we won't break the requirement of storage overhead in (4), while the worst case of additional reconstruction overhead is $\sum_{i=1}^{N} w_i d_i$. Thus the approximation ratio of this ceiling rounding algorithm is $1 + \frac{1}{\min_i k_i}$.

Comparing to this naive rounding algorithm, in Zebra we use an iterative rounding algorithm that achieve even lower reconstruction overhead. The idea of this iterative algorithm is in each round solving the geometric programming problem in (4)-(6), and rounding one k_i to its nearest positive integer which leads to the minimum change to the overall reconstruction overhead. In the next round, the file corresponding to the k_i in the previous round is removed from the problem and the storage overhead it has incurred is also deducted in (5). We keep running this iteration such that in every round one k_i will be picked up and removed from the model until there is only one file left and the last k_i will be rounded to its ceiling. The details of this algorithm is shown in Fig. 5.

We now run Zebra on the same Facebook workload used in Fig. 6a, and illustrate the overall reconstruction overhead which is calculated by (4). Note that in practice not all requests will be served by degraded read. However, as explained in Sec. 4, the overall reconstruction overhead is linearly proportional to the average reconstruction overhead per degraded read in a given workload, regardless of the actual probability of degraded read. We assume that any two block failures should be tolerated, *i.e.*, $r = 2$, and the overall storage overhead should be no more than 1.2x. Compared to a single erasure code, which we use a $(10, 2)$ RS code in

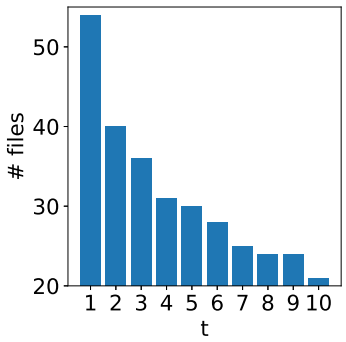


Fig. 4. The number of files with various numbers of t running in a Facebook workload.

this case to meet the requirement of storage overhead, Zebra with the iterative rounding algorithm can save up to 39.2% reconstruction overhead. We can also observe that with a smaller value of t, the overall reconstruction overhead can be further saved, as with a smaller t, there will be more files and potentially more tiers.

In practice, our observation shows that under a real workload, the approximation ratio of the reconstruction overhead with ranks of blocks solved by Zebra can be closer to 1. We can see that the performance of the approximated solutions of K is close to the non-integer solution of K (ideal K), where the ceiling rounding algorithm can incur at most 10.0% more reconstruction overhead in the worst case. On the other hand, the iterative rounding algorithm can achieve even lower approximation ratio, by incurring at most 6.0% more reconstruction overhead. The reason is that instead of rounding k_i at the same time, we round different k_i iteratively every round in Fig. 5. Therefore, the difference of storage overhead caused by the rounding in one round will be reflected in the next round, reducing the differences of overall reconstruction overhead of remaining data. A more detailed comparison of these two rounding algorithm can be found in Fig. 6b, where the iterative rounding algorithm on average incurs 49.1% less additional reconstruction overhead than the ceiling rounding algorithm. The saving tends to be more significant when we have a smaller value of t.

Due to the low number of files in the refined model, the completion time to solve K is swift. In fact, we can always solve K with both of the two rounding algorithms within 2.5 second, with any values of t.

5.3 Encoding data in multiple tiers

From the ranks solved above, we can now encode data into multiple tiers. The number of tiers is determined by merging files with the same rank into one tier. We actually encode

Input: w_i and d_i where $i = 1, \ldots, N$, $C > 1$, $r \in \mathbb{Z}^+$, $N \in \mathbb{Z}^+$
Output: k_i where $i = 1, \ldots, N$

$S = \emptyset, T = \{1, \ldots, N\}$
$R = 0$
while $T \neq \emptyset$ **do**
  solve

$$\min \quad \sum_{i \in T} w_i d_i k_i \tag{7}$$

$$\text{s.t.} \quad \sum_{i \in T} \frac{w_i}{k_i} \leq \frac{(C-1)N}{r} - R \tag{8}$$

$$k_i > 0, \forall i \in T. \tag{9}$$

  $\forall i \in T$, let k'_i be the positive integer nearest to k_i
  $\forall i \in T$, calculate $\delta_i = |w_i d_i (k_i - k'_i)|$
  $\hat{i} = \arg\min_{i \in T} \delta_i$
  $R = R + \frac{w_{\hat{i}}}{k'_{\hat{i}}}$
  $k_{\hat{i}} = k'_{\hat{i}}$ (when $|T| > 1$) or $\lceil k_{\hat{i}} \rceil$ (when $|T| = 1$)
  $S = S + \{\hat{i}\}, T = T - \{\hat{i}\}$
end while

Fig. 5. The iterative rounding algorithm used in Zebra

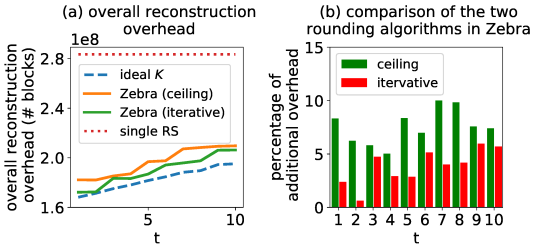


Fig. 6. Comparisons of the overall reconstruction overhead and the approximation ratio of the two rounding algorithms.

blocks in each tier with a (k_i, r) RS code, where k_i is the rank of the file and r is the number of failures to tolerate. In particular, for the blocks of rank 1, *i.e.*, $k_i = 1$, they will be replicated with $1 + r$ copies. Since data in each tier are migrated from multiple files, the chance of having a tier with no more than k_i blocks is negligible. Inside each tier, we encode every k_i blocks into r parity blocks with the (k_i, r) RS code where such $k_i + r$ blocks belong to a stripe. All blocks in each stripe will be stored into different servers. If we have remaining blocks or the total number of blocks is less than k_i, we will temporarily encode them with an (l, r) RS code if the number of remaining blocks is l. Since $l < k_i$, this will incur additional storage overhead. However, given the large number of blocks stored in a distributed storage system, this additional storage overhead is marginal.

In terms of storage overhead of additional metadata introduced by Zebra, we can store the rank along with the existing metadata of each file. Other metadata, such as r, C, t, and other additional parameters introduced in the rest of this paper, are global information and require only few bytes to store.

5.4 Balancing load of hot data

So far, when the optimal solution of K solved from (7)-(9) has k_i no more than 1, we will use a $(1, r)$ RS code, equivalent to $(1 + r)$-way replication, to store the corresponding block. As blocks with higher demand will have lower ranks, we can believe that only the hottest block will be replicated. This makes perfect sense because the demand is so high that we probably cannot afford the cost to reconstruct them with RS code if one data block is not available. However, as shown in Fig. 2, the demand of the hot blocks can be much more variable than cold blocks. Therefore, even if all of the hottest blocks are stored with $(1 + r)$-way replication, their demand will still be highly variable.

We notice that in the iterative rounding algorithm, the storage overhead of each file, *i.e.*, $1 + \frac{r}{k_i}$, is determined by $\frac{1}{k_i}$. When we choose any integer that is more than k_i, we actually use less storage space than the space assigned to this file in the non-integer solution of (7)-(9). Therefore, if $k_i \leq 1$, we can store the hot blocks of this file with more copies such that the average load of these hot blocks can be reduced and better balanced, while not affecting the ranks of other blocks. Specifically, the number of copies will be $1 + r \left\lfloor \frac{1}{k_i} \right\rfloor$, and thus a block with lower k_i will have more copies than $r + 1$. Meanwhile, the storage overhead of this

block will not exceed $1+\frac{r}{k_i}$, *i.e.*, the overall storage overhead will not exceed the given limit of C. The reconstruction overhead will not be affected as well, since we only change the number of copies for the block with $k_i \leq 1$.

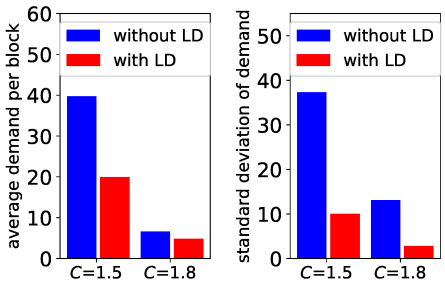


Fig. 7. Comparison of demand of hot blocks (of rank 1) with and without load balance (LD), when $r = 2$ and $t = 1$.

To implement this load balancing, we only need to change the three lines in Fig. 5. If $k_i < 1$, we calculate $\delta_i = w_i d_i (1 - k_i)$ in line 6, and $R = R + w_{\hat{i}} \left\lfloor \frac{1}{k_{\hat{i}}} \right\rfloor$ in line 8. In line 9, we let $k_{\hat{i}} = 1$ and meanwhile replicate this file with $1 + r \left\lfloor \frac{1}{k_{\hat{i}}} \right\rfloor$ copies. If $k_i \geq 1$, these three lines remain unchanged. In this way, the storage space assigned to the chosen file by the non-integer solution of (7)-(9) in this round is preserved.

We show in Fig. 7 the mean and standard deviation of the demand of blocks of rank 1 under the Facebook workload, with C, the overall storage overhead, no more than 1.5 and 1.8. With the load-balanced replication, we can observe in Fig. 7a that the demand of hot blocks on average can be saved by 49.8% when the limit of the storage overhead is 1.5, and by 26.9% when the storage overhead increases to $C = 1.8$. The load of hot blocks can also be significantly more balanced. From Fig. 7b, we can observe that the standard deviation of all these hot blocks can be reduced by 57.3% ($C = 1.5\%$) and 76.4% ($C = 1.8\%$).

6 Deploying Zebra with Online Demand

6.1 Online demand

Before we deploy the Zebra framework in any practical scenarios, we should be aware that ranks inside the Zebra framework can only be constructed with the given demand, however, they must work well with the demand in future.

To meet this requirement with online demand, the demand D we use in the model will not be purely determined by the most recent demand, but a linear combination of demand over a longer period of time. Specifically, we split the time into intervals. For example, if the interval is one hour, we measure the demand of each file every hour, and the demand measured in this hour is the most recent demand in the next interval.

If we use the most recent demand to solve K, we may imprudently increase k_i of some block if this block gets transiently high demand in the latest interval and in the next interval there will be no such high demand. Therefore, we also need to consider the consistency of the demand besides the latest demand. In this paper, we use α to achieve a flexible trade-off between the consistency and the transiency of the demand. Assume that D_0 is the most recent demand and D is the demand we use to calculate K in the last interval. After this interval, we are going to update the demand D as

$$(1 - \alpha)D + \alpha D_0, \alpha \in [0, 1]. \tag{10}$$

It is straightforward that when $\alpha = 1$, we will always use the latest demand to calculate K used in the next interval. On the other hand, as α goes to 0, the latest demand will be less and less taken into account. When $\alpha = 0$, D won't be updated at any time. In this way, the transient demand will get smoother over time. Therefore, we won't be easily tricked by the transient demand. In other words, a smaller α can help to make it more consistent in the updated demand D over time.

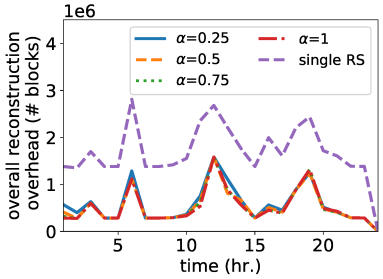


Fig. 8. Overall reconstruction overhead with demand updated online.

Once again, we run the simulation with the hourly updated demand on the workload from Facebook, with $r = 2$, $C = 1.2$, and $t = 1$. Fig. 8 illustrates the results. Compared to the single RS code, Zebra can work well with online demand, and we can on average save 68.3% of reconstruction overhead in general. We can observe that in this workload, the transiency is quite significant (from the overhead of the single RS code), and thus with a larger α we can slightly better adapt to the general workload change. In fact, the best choice of α depends on the characteristics of the workload, and we will show the results with more workload in Sec. 8.

6.2 Data migration

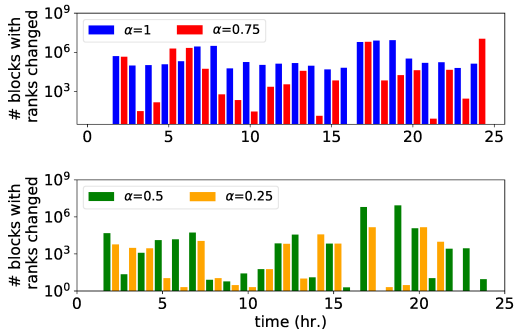


Fig. 9. The number of blocks to migrate into different ranks after each hour.

Once the demand changes after each time interval, we will need to update the files if their ranks are changed. This may involve quite a lot of migration overhead, especially the network transfer. An naive way to update files with new ranks is to compute new parity blocks with the new RS code and then remove the old parity blocks. To encode r parity blocks from k data blocks with a (k, r) RS code, we need to transfer at least $k + r - 1$ blocks, by computing all parity blocks on one server and sending $r-1$ ones to other servers. Thus, if the rank of a file is updated, the traffic to generate new parity blocks will be even more than the amount of this file. Fig. 9 shows the number of blocks that have different ranks after each hour. We can see that there can be 525.9 TB of data to migrate to new RS codes at some hour when $\alpha = 1$. The migration overhead varies with different values of α, and it is hard to predict (in some hours we have more blocks to migrate when $\alpha = 0.75$ than $\alpha = 1$ and $\alpha = 0.5$. In other words, we need to generate new parity blocks for almost half of all data. Moreover, when $\alpha = 1$, the amount of data to migrate can even exceed the amount of data requested by users. This will not only hinder the migration process from finishing quickly, but hurt the performance of the data access as well. Hence, though migration is unavoidable to meet the ever-changing demand, our objective is to reduce the network transfer for migration to a marginal level, for any values of α.

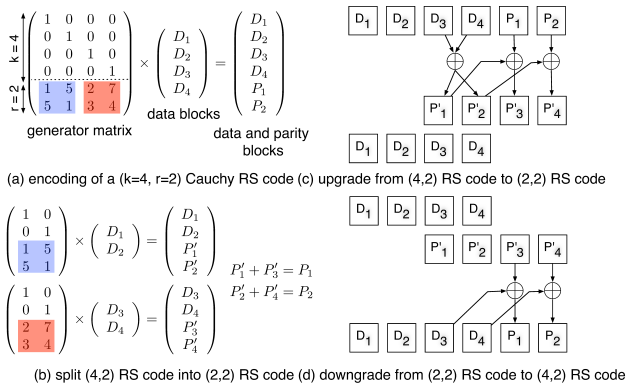


Fig. 10. Construction of the Cauchy RS code and its migration.

In this paper, we propose a different way to encode data such that we can control the migration overhead without increasing the overall reconstruction overhead significantly. We use Cauchy RS codes [32], [33] in Zebra, which contains a Cauchy matrix in its generator matrix. With a (k, r) RS code, we have a $(k + r) \times k$ matrix G as its *generator matrix*, such that the encoding operation can be formalized as the multiplication of the generator matrix and the k data blocks, as illustrated in Fig. 10a. In particular, if the first k rows of G are an identity matrix, the corresponding RS code will be systematic. In this way, we can write G as $G = \begin{bmatrix} I \\ \hat{G} \end{bmatrix}$. In a Cauchy RS code, the matrix $\hat{G}$ is a Cauchy matrix. A benefit of Cauchy RS code is that all encoding operations can be converted into XOR operations. More importantly, Cauchy matrix makes it easy to migrate from one RS code into another RS code with significantly less overhead, since any submatrix of a Cauchy matrix is still a Cauchy matrix (Fig. 10b).

Because of this property, we can easily downgrade or upgrade data between an (mk, r) and a (k, r) RS code, $m \in \mathbb{Z}^+$. We show in Fig. 10c and Fig. 10d how we can migrate between these two RS codes. To downgrade from a (k, r) RS code to an (mk, r) RS code, by applying the property of the Cauchy matrix, we only need to XOR the r parity blocks in the m stripes together. To upgrade from an (mk, r) RS code to a (k, r) RS code, we need to generate the parity blocks in the $m - 1$ stripes under the (k, r) RS code and XOR the new parity blocks and the existing parity blocks into the parity blocks of the last stripe. We show in Table 1 the network transfer of both upgrade and downgrade, as well as the network transfer of the naive migration scheme. We can see that the Cauchy matrix can help to save network transfer in both cases, when $m < 2 + \frac{k-1}{r}$. Apparently when $m = 2$ this condition can always be satisfied, and we will use this property to save the migration overhead.

TABLE 1
Network transfer between (mk, r) and (k, r) RS codes.

	without Cauchy matrix	with Cauchy matrix
downgrade	$mk + r - 1$ blocks	$(m - 1)r$ blocks
upgrade	$m(k + r - 1)$ blocks	$(m - 1)(k + 2r - 1)$ blocks

However, we can rely on the Cauchy matrix only when the rank of the new code is multiple times or can be divided into the rank of the old code. To maximize this effect, we can also change the way to solve ranks. We set $k_{\max}$, an upper bound of ranks of all blocks, such that all k_i should be no more than $k_{\max}$. Moreover, the rank of all blocks should be a divisor of $k_{\max}$. Given k_i solved from (7)-(9) with an additional constraint $k_i \leq k_{\max}, \forall i \in T$, we need to additionally change line 5 in Fig. 5 by letting k'_i be one of the divisors of k_{max} that is nearest to k_i, $\forall i \in T$, and then encode the corresponding file with a (k'_i, r) RS code.

When we migrate m stripes of blocks encoded with a (k, r) RS code into one stripe with an (mk, r) RS code, we may also need to move data blocks because two blocks in different stripes may be stored in the same server. In Zebra, if $k_{\max}$ is set, we store every $k_{\max}$ data blocks into different servers, and compute parity blocks with the corresponding (k, r) RS codes which will be stored into other servers. Notice that such (k, r) RS codes are constructed by splitting the generator matrix of the (k_{max}, r) Cauchy RS codes, as if there were upgraded from the (k_{max}, r) RS codes before. Therefore, when we need to migrate into an (mk, r) RS code, we will not need to move any data blocks, but only migrate parity blocks as described above. This method can also be applied to the upgrade case as well. Apparently, we can maximize the effect of the Cauchy matrix by wisely selecting the value of $k_{\max}$. For example, when $k_{\max} = 16$, we have 5 available ranks $(1, 2, 4, 8, 16)$. Thus, when downgrading or upgrading to any neighbor ranks we can always exploit the Cauchy matrix to save network transfer (with $m = 2$). For some other values of $k_{\max}$, some rank may not be integer multiples of its neighboring rank (*e.g.*, 3 and 4 when $k_{\max} = 12$), we will have to remove all existing parity blocks and generate new parity blocks with the new ranks.

There is also a side effect of $k_{\max}$ which set a lower bound of the data reliability. This is because with a higher

value of k, the encoded data will have lower data reliability. For example, data encoded with a $(10, 2)$ RS code will have a lower reliability than data encoded with a $(4, 2)$ RS code, even though they can both tolerate two failures. This can happen in all designs with tiered architectures [17], [18], [19]. By limiting the highest value of k, we can also allow the user to control the data reliability in the worst case.

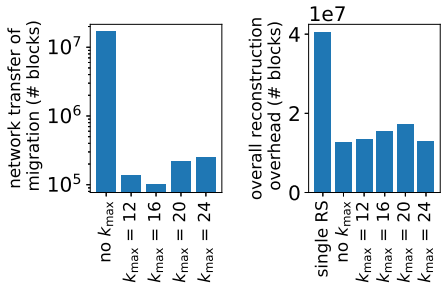


Fig. 11. Network transfer incurred by the migration and the overall reconstruction overhead with various values of $k_{\max}$, when $\alpha = 0.5$.

In Fig. 11, we compare the performance with different values of $k_{\max}$. We run the Facebook workload by updating ranks of data every hour and calculate all the network transfer incurred by the migration, with $C = 1.2$, $r = 2$, $t = 1$, and $\alpha = 0.5$. We can see that all values of $k_{\max}$ in Fig. 11 can significantly reduce the network transfer such that only 0.6% of the original network transfer will be incurred ($k_{\max} = 16$). The reasons are twofold. First, limiting the number of possible ranks naturally reduces the chance of having a different rank when the demand of a file changes. Second, for files with very low demand or no demand, *i.e.*, when d_i is close to 0, their ranks are both high and subject to frequent changes even if their own demand does not change over time. Since the change of ranks will also change the storage overhead of files, the storage overhead given to cold data can also change when the demand of hot data changes. Therefore, the cold data can also be subject to frequent migration even though their demand does not change so frequently. Limiting the rank to the divisors of k_{max} can make cold data less sensitive to the change of demand and more likely to keep their own ranks (k_{max} in most cases). As most data are cold data in a typical distributed storage systems, most (unnecessary) migration overhead can be avoided.

The overall reconstruction overhead, on the other hand, will increase by between 3.2% ($k_{\max} = 24$) and 36.5% ($k_{\max} = 20$), still much less than one single RS code.

We show the results of other values of α in Fig. 12. We can see that with any values of α, the total network transfer for migration will be reduced by more than 93.1%. In the worst case ($\alpha = 1$), the network transfer in migration is no more than 63.9 TB, which occupies less than 4.1% of all traffic incurred by demand. Meanwhile, at most 7.4% more overhead for reconstruction will be incurred. Hence, we reduce the network transfer for migration to a marginal level without incurring much additional reconstruction overhead.

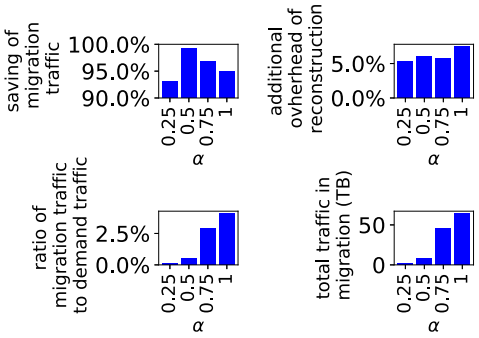


Fig. 12. Comparison of network transfer incurred by migration with various values of α, when $k_{\max} = 12$.

7 Zebra with Local Reconstruction Codes

Besides RS codes, the principles of the Zebra framework can also be used for other erasure codes. In this section, we apply Zebra with another kind of erasure codes that are specifically proposed for distributed storage systems, *i.e.*, local reconstruction codes, currently deployed in Windows Azure Storage [8].

7.1 Local Reconstruction Codes

Local reconstruction codes are proposed in order to achieve a lower number of blocks visited than RS codes when an unavailable block is reconstructed. Local reconstruction codes are associated with three integer parameters, k, l, g, where k denotes the number of data blocks. There are also two types of parity blocks in local reconstruction codes, *i.e.*, local and global parity blocks, denoted by l and g respectively where $l|k$. We show in Fig. 13 an example of a (k, l, g) local reconstruction codes where $k = 6$, $l = 2$, and $g = 1$.

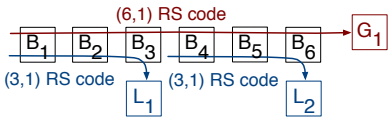


Fig. 13. An illustration of a $(6, 2, 1)$ local reconstruction code, with 6 data blocks ($B_1 - B_6$), 2 local parity blocks (L_1 and L_2) and 1 global parity blocks (G_1).

A local reconstruction code is built upon RS codes. A (k, g) RS code is used to compute g global parity blocks from k data blocks. For every k/l data blocks, we use a $(k/l, 1)$ RS code to compute one local parity block. Hence, we have l local parity blocks, and then we can reconstruct any data block or local parity block from k/l blocks, which is naturally less than k blocks with RS codes. For example, in Fig. 13, if B_1 is not available, we can reconstruct it with 3 blocks, *i.e.*, B_2, B_3, and L_1. On the other hand, it is proved [8] that given a (k, l, g) local reconstruction code, any $g + 1$ failures can be tolerated.

7.2 Configuring local reconstruction codes with Zebra

The same as RS codes in Sec. 5.1, we can temporarily group blocks into n files by their demand when calculating their ranks, which is updated per time interval by (10), and we

use w_i and d_i to denote the size and the demand of file i, $i = 1, \ldots, n$. For each file we encode all blocks in this file with a (k_i, l_i, g_i) local reconstruction code. In other words, in this file we encode every k_i data blocks with the (k_i, l_i, g_i) local reconstruction codes in one stripe, and generate l_i local parity blocks and g_i global parity blocks. Assuming that any r failures should be tolerated, for all i we can set $g_i = r - 1$.

As the demand to data all goes to data blocks, we only consider the reconstruction overhead of data blocks. To apply Zebra to local reconstruction codes, we need to dynamically determine the "rank" of data blocks by their demand. As the rank is defined as the number of blocks to visit in the reconstruction in RS codes, in local reconstruction codes the rank of a data block can be defined as k_i/l_i. Hence we use the technique in Sec. 6.2 by letting $k_i = k_{\max}$, which is naturally an upper bound of ranks of all data blocks. The overall reconstruction overhead can be written as $\sum_{i=1}^{N} w_i d_i \frac{k_{\max}}{l_i}$.

On the other hand, the storage overhead of a local reconstruction code is $1 + \frac{l_i + g_i}{k_i}$. As the value of g_i only depends on the number of failures to tolerate and k_i is a constant, the overall storage overhead of all files,

$$\sum_{i=1}^{n} w_i \left(1 + \frac{l_i + g_i}{k_{\max}}\right) = N\left(1 + \frac{r-1}{k_{\max}}\right) + \sum_{i=1}^{n} w_i \cdot \frac{l_i}{k_{\max}}.$$

linearly depends on the inverse of the rank. By letting $\kappa_i = k_{\max}/l_i$, we can solve the rank of data blocks in each file by a geometric programming problem that is similar to (4)-(6).

$$\min \quad \sum_{i=1}^{n} w_i d_i \kappa_i \tag{11}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} \frac{w_i}{\kappa_i} \leq \frac{(C - 1 - \frac{r-1}{k_{\max}})N}{r} \tag{12}$$

$$\kappa_i > 0, \forall i. \tag{13}$$

Based on this problem, we can get the iterative rounding algorithm to solve l_i in each file, as shown in Fig. 14. As local reconstruction codes can be modeled as geometric programming problems similar to RS codes, this algorithm is largely based on the iterative rounding algorithm in Fig. 5, except for the following differences:

First, we always choose l_i as a divisor of $k_{\max}$, such that κ_i is always a divisor of $k_{\max}$ as well. Once a non-integer solution κ_i is obtained after line 4, we will round it to the nearest integer that is a divisor of $k_{\max}$. In this way, we can also apply the technique used in Fig. 10 to construct the generator matrix of the RS codes to compute the local parity nodes. In other words, in one stripe with $k_{\max}$ data blocks, the l_i generator matrixes that correspond to the l_i local parity blocks can be split from the Cauchy generator matrix of a $(k_{\max}, 1)$ RS code, helping us to save traffic in the migration when the value of l_i is updated.

In Fig. 15, we compare the overall reconstruction overhead of various values of $k_{\max}$ with one single local reconstruction codes, running under the Facebook workload, with $C = 1.2$, $r = 2$, $t = 1$, $\alpha = 0.5$, and one hour per time interval. To meet the requirements of storage overhead and failure tolerance, we use a $(20, 4, 1)$ local reconstruction codes as the single LRC in Fig. 15. It turns out that with

Input: w_i and d_i where $i = 1, \ldots, n$, $C > 1$, $r \in \mathbb{Z}^+$, and $N \in \mathbb{Z}^+$
Output: l_i where $i = 1, \ldots, n$
1: $S = \emptyset$, $T = \{1, \ldots, n\}$
2: $R = 0$
3: **while** $T \neq \emptyset$ **do**
4: solve the problem

$$\min \quad \sum_{i \in T} w_i d_i \kappa_i \tag{14}$$

$$\text{s.t.} \quad \sum_{i \in T} \frac{w_i}{\kappa_i} \leq \frac{(C - 1 - \frac{r-1}{k_{\max}})N}{r} - R \tag{15}$$

$$\kappa_i > 0, \forall i \in T \tag{16}$$

5: **for each** $i \in T$ **do**
6: **if** $|T| > 1$ **then**
7: let κ_i' be the integer that is both nearest to κ_i and a divisor of $k_{\max}$
8: **else**
9: let κ_i' be the integer that is no less than κ_i and also a divisor of $k_{\max}$
10: **end if**
11: calculate $\delta_i = |w_i d_i (\kappa_i - \max(1, \kappa_i'))|$
12: **end for**
13: $\hat{i} = \arg\min_{i \in T} \delta_i$
14: **if** $\kappa_{\hat{i}} \geq 1$ **then**
15: $R = R + \frac{w_{\hat{i}}}{k_{\hat{i}}'}$
16: $\kappa_{\hat{i}} = \kappa_{\hat{i}}'$, $l_i = \frac{k_{\max}}{\kappa_{\hat{i}}}$
17: **else**
18: $R = R + w_{\hat{i}} \cdot \left\lfloor \frac{1}{\kappa_{\hat{i}}} \right\rfloor$
19: $l_i = k_{\max} \cdot \left\lfloor \frac{1}{\kappa_i} \right\rfloor$, $\kappa_{\hat{i}} = 1$
20: **end if**
21: $S = S + \{\hat{i}\}$, $T = T - \{\hat{i}\}$
22: **end while**

Fig. 14. The iterative rounding algorithm to solve l_i of local reconstruction codes.

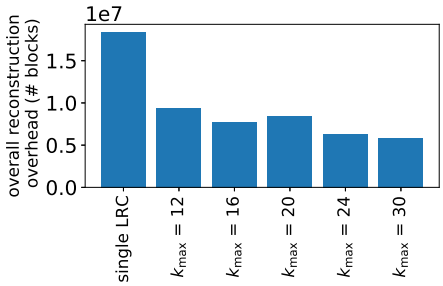


Fig. 15. Comparison of overall reconstruction overhead of various values of $k_{\max}$ with one single local reconstruction code (LRC), when $\alpha = 0.5$.

Zebra we can save reconstruction overhead by between 49.1% ($k_{\max} = 12$) and 68.2% ($k_{\max} = 30$). On the other hand, we can observe in Fig. 16 that with various values of $k_{\max}$, the traffic incurred in the migration between time intervals can be well controlled, where the total amount of traffic occupies no more than 1.2% of the data demand.

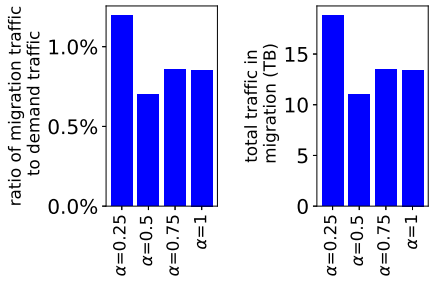


Fig. 16. Comparison of network transfer incurred by migration with various values of α, when $k_{\max} = 16$.

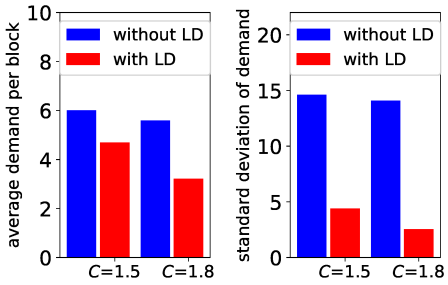


Fig. 17. Comparison of demand of hot blocks (of rank 1) with and without load balance (LD), when $r = 2$ and $t = 1$.

Second, we apply the technique used in Sec. 5.4 to balance the load of hot blocks. When κ_i is solved to be less than 1, we change the way to generate the local parity blocks in the original local reconstruction codes, by replicating each data block with $\left\lfloor \frac{1}{\kappa_i} \right\rfloor$ more copies in line 19. In Fig. 17, we calculate the mean and the standard deviation of the demand of such hot blocks with $k_{\max} = 24$. Similar results as RS codes in Fig. 7 can be observed where both the mean and the standard deviation of the demand can be significantly reduced. We also notice that the average demand of hot blocks with local reconstruction codes is much less that that with RS codes in Fig. 7, implying that local reconstruction codes allow more blocks to have low ranks than RS codes with the same storage overhead. The reason can be inferred from the fact that with the same storage overhead, local reconstruction codes are designed to have lower reconstruction overhead than RS codes.

8 Simulation

8.1 Methodology

We evaluate the performance of the Zebra framework with more workloads in this section. The two workloads [14] used in this paper are obtained from a 600-machine cluster at Facebook running MapReduce jobs, each of which spans 24 hours. We show characteristics of the two workloads in Table 2, and we have been using the FB2 workload to evaluate the design of Zebra throughout this paper. In Table 2, both workloads have a significant skewness of data demand, where more than half of data have demand less than the 1.6 and only a small portion has very high demand.

TABLE 2
Characteristics of workloads used in the simulation.

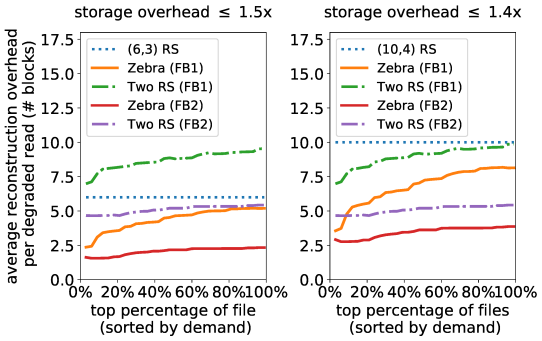
workload	size	# files	max demand	mean demand
FB1	0.96 PB	15223	688	1.6
FB2	1.07 PB	16256	721	1.5

Like previous evaluations, we divide the time in each workload into intervals, where each time interval spans one hour. In each interval, we calculate the rank of files with the demand updated by (10). The new file that appears for the first time in an interval will be stored with with a default rank. The default rank is calculated as the minimum divisor of $k_{\max}$ that is no less than $\frac{r}{C-1}$, and can hence achieve the required storage overhead C with r failures to tolerate. In the simulation, the size of each block is 64 MB. After each interval, the ranks of all existing files will be updated. Given the rank of each file, we can further calculate the average overhead of reconstruction per degraded read, the storage overhead of blocks, and the network transfer of migration.

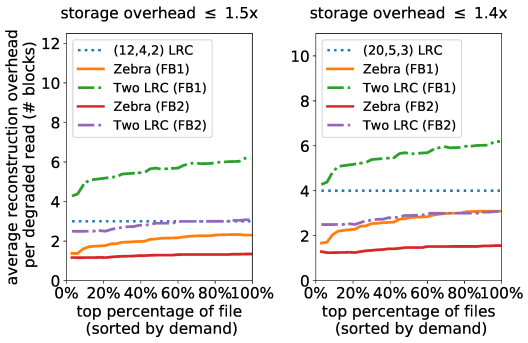
In the simulation, given the number of failures to tolerate and the storage overhead, we compare the performance of one static erasure code (RS code or local reconstruction code) and Zebra with the corresponding erasure code. Besides, we add another scheme that encodes data into two tiers of erasure codes. This scheme is similar to the method proposed in [19] which implements two tiers with two other preconfigured erasure codes. In our simulation, the two tiers both deploy RS codes or local reconstruction codes for the purpose of fair comparison. For convenience, we name the two tiers as hot tier and cold tier, as the hot tier will store hot data with low reconstruction overhead while the cold tier can provide low storage overhead for the cold data. In this scheme, under the constraint of the overall storage overhead, we try to assign as much hot data as possible to the hot tier and store the rest in the cold tier. In the simulation with RS codes, we deploy a $(4, r)$ RS code in the hot tier and a $(12, r)$ RS code in the cold tier. With local reconstruction code, a $(16, 8, r-1)$ local reconstruction code is deployed in the hot tier, and a $(16, 2, r-1)$ local reconstruction code in the cold tier.

8.2 Results

We first evaluate the reconstruction overhead. At each time interval, we calculate the average of the reconstruction overhead per degraded read, where we define the reconstruction overhead as the number of blocks to read in the reconstruction. In other words, if the rank of a block is k, its reconstruction overhead at that time is k as well. This average reconstruction overhead per degraded read can be considered as the expected reconstruction overhead of visiting an unavailable block once we have a failure in the distributed storage system. In all the simulations below, we set $t = 1$, $\alpha = 0.5$, and $k_{\max} = 24$, unless mentioned otherwise. With such settings, we evaluate the reconstruction overhead with RS codes and local reconstruction codes, respectively. In the simulation with RS codes, we use two RS codes, *i.e.*, $(6, 3)$ RS code ($C = 1.5, r = 3$) and $(10, 4)$ RS code ($C = 1.4, r = 4$) as single erasure codes, respectively. In the other simulation with local reconstruction codes, we



(a) RS codes.



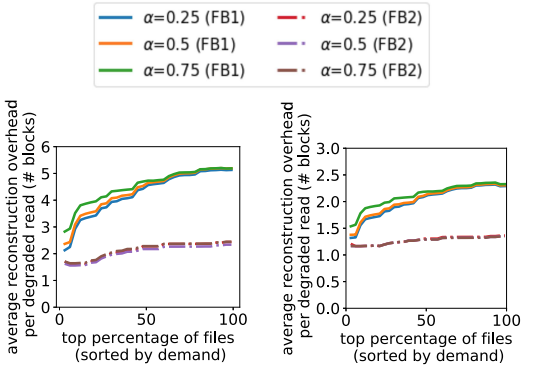
(b) local reconstruction codes (LRC).

Fig. 18. Average reconstruction overhead per degraded read with storage overhead 1.5 and 1.4.

use $(12, 4, 2)$ and $(20, 5, 3)$ local reconstruction codes, which achieve the same storage overhead and failure tolerance as corresponding RS codes.

We show the average reconstruction overhead in Fig. 18, where we sort files by their demand, and then calculate the average reconstruction overhead per degraded read of files in a given percentage of top demanded files with $C = 1.5$ and 1.4, respectively. From Fig. 18a and Fig. 18b, we can see that all data can expect lower reconstruction overhead per degraded read than the single erasure code with the same storage overhead, no matter with RS codes or with local reconstruction codes. The reason is that in the Zebra framework, even though cold data may have higher reconstruction overhead than the single erasure code, their demand can actually occupy a very small portion of all demand. Hence, on average, the reconstruction overhead per degraded read of all data can be lower than the single erasure code, especially for the hotter data. Two erasure codes can also achieve lower average reconstruction overhead than the single erasure codes. However, Zebra can achieve better results as it tries to minimize the overall reconstruction overhead. Similar results can also be obtained when we require for a higher value of storage overhead. In many cases, with two erasure codes we cannot even compete with one single erasure code as the static configuration of the two erasure codes cannot match the unknown demand in advance.

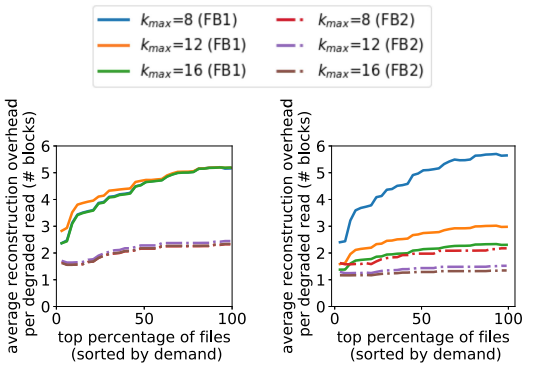
On average of the two workloads, we can save the average reconstruction overhead by up to 73.6% ($C = 1.5$) for the top 15% demanded files in Fig. 18a, and by up to 72.1% when $C = 1.4$. Comparing RS codes with local reconstruction codes, with the same storage overhead, local reconstruction codes can save the average reconstruction overhead by 80.5% ($C = 1.5$) and 87.5% ($C = 1.4$), respectively. This is credited to the low reconstruction overhead of local reconstruction codes.



(a) RS codes. (b) local reconstruction codes.

Fig. 19. Average reconstruction overhead with various values of α.

We compare the choices of α in Fig. 19, in which we use 1.5 as the storage overhead. We find that with both RS codes and local reconstruction codes, the two workloads demonstrate different reconstruction overhead with different values of α, where FB1 favors lower α while FB2 is not as sensitive to α as FB1. Hence, we believe that the best choice of α depends on the characteristics of the workload.



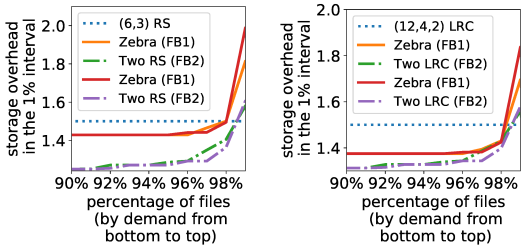
(a) RS codes. (b) local reconstruction codes.

Fig. 20. Average reconstruction overhead with various values of k_{max}.

We also compare the reconstruction overhead with different values of k_{max} in Fig. 20. With $C = 1.5$ and $r = 3$, we find that RS codes are not sensitive to different values of k_{max}, as the demand for data with high ranks is so low that little reconstruction overhead is contributed by such data. However, local reconstruction codes are more easily affected by the change of k_{max} than RS codes. This is because with a higher value of k_{max}, a block with the same reconstruction overhead is associated with lower storage overhead. For example, data encoded with a $(8, 4, 2)$ local reconstruction code have lower storage overhead than that with a $(16, 8, 2)$ local reconstruction code, while they (data blocks) both require two blocks to reconstruct. Therefore, with a higher

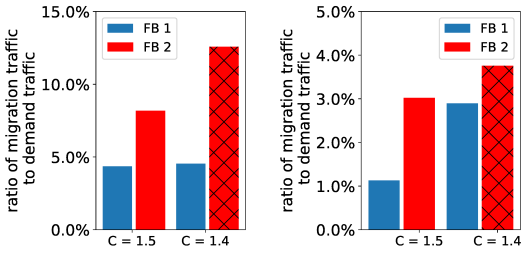
k_{max}, more data can have lower reconstruction overhead. With RS codes, on the other hand, different values of k_{max} do not change the reconstruction overhead of data, but only introduce one more tiers with lower storage overhead. This tier, as mentioned above, has little demand that cannot significantly change the overall reconstruction overhead.

In Fig. 21, we measure the average storage overhead of files in all time intervals, to compare the storage overhead of data with different demand. This time we sort the files by their demand (from bottom to top) and calculate the average storage overhead of data in a 1% interval of files. For example, data points with 99% on the x-axis indicate the average storage overhead of files with demand in the bottom 99%-100% interval, *i.e.*, the top 1% most demanded files. In fact, 90% files have very low demand in both FB1 and FB2, and hence they all have very similar storage overhead. Hence, in Fig. 21 we focus on the top 10% files. We can see that at most 2% files have storage overhead higher than the given constraint in Zebra, indicating their extremely high demand. On the other hand, due to the static configuration of the two erasure codes, more data with high demand have to be stored in the cold tier with unnecessarily low storage overhead. Hence, with the two static erasure codes, we cannot fully utilize the storage space, and this also explains why two erasure codes have higher reconstruction overhead in Fig. 18.



(a) RS codes. (b) local reconstruction codes (LRC).

Fig. 21. Storage overhead of data with different demand.

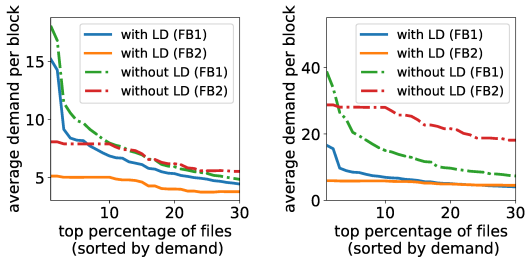


(a) RS codes. (b) local reconstruction codes.

Fig. 22. The ratio of the total network transfer incurred by the migration to the network transfer incurred to serve demand.

We illustrate the network transfer incurred by the migration in Fig. 22, by comparing it with the total network transfer to serve the demand. With various constraints of the overall storage overhead, we can see that the total migration traffic never exceeds 13% of the total demand traffic, thanks to the Cauchy RS codes used in Zebra. Though more migration traffic can be observed with higher storage overhead, we can see that this increased amount of traffic is marginal.

Finally, we show the distribution of demand with the Zebra framework with Fig. 23. We can observe that in both FB1 and FB2, the load balance in Zebra can help to significantly save the peak demand of blocks. With the load balance in Zebra, we can achieve 12.1% less demand on average for the top 15% demanded files with RS codes in FB1 and 34.6% in FB2. With local reconstruction codes, we can save the load on hottest files by 50.9% (FB1). Moreover, the load balance can help to make the demand of blocks more stable among files with different demand. It can also be observed that hotter files can enjoy more savings of demand. From the top 3% demanded files to the top 30% demand files in Fig. 23b, the saving of demand can change from 63.3% to 20.9%. The reason is easy to infer, as the load balance in Zebra can only work for the replicated blocks, *i.e.*, the hottest block of rank 1. Thus, with a higher percentage, we will have more cold data, which not only reduce the average demand, but dilute the load balance achieved by the hot data as well.



(a) RS codes. (b) local reconstruction codes.

Fig. 23. Comparison of average demand of blocks, with/without the load balance (LD) in Zebra.

With the same workload, we notice that RS codes can offer better load balance than local reconstruction codes. Though initially it looks conflicting to the fact that with local reconstruction codes more data can be replicated with the same storage overhead, we notice that to offer the same level of failure tolerance, local reconstruction codes cannot replicate the same number of copies as RS codes. For example, with $r = 3$ RS codes can replicate data blocks at least 4 times while local reconstruction codes can replicate only twice. On the other hand, as local reconstruction codes tend to have lower ranks than RS codes with the same storage overhead, the average demand can be saved more significantly with the load balance in Zebra.

9 Conclusions

In this paper, we exploit the skewness of demand in distributed storage systems and propose the Zebra framework that can efficiently and dynamically encode data into multiple tiers with Reed-Solomon codes or local reconstruction codes, according to their demand. The Zebra framework can achieve a much lower reconstruction overhead for the hot

data, while spending less storage space to store the cold data. Zebra can also help to balance the demand, especially for the hot data. With the ever-changing demand, Zebra can update itself accordingly with a low network transfer.

REFERENCES

[1] 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. u. Haq, M. I. u. 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 *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2011, pp. 143–157.

[2] S. Ghemawat, H. Gobioff, and S. Leung, "The Google File System," in *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[3] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Proc. 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[4] 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.

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

[6] 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.

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

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

[9] "[HDFS-7295] Erasure Coding Support inside HDFS," https://issues.apache.org/jira/browse/HDFS-7285.

[10] H. Weatherspoon and J. D. Kubiatowicz, "Erasure Coding vs. Replication: A Quantitative Comparison," in *Proc. International Workshop on Peer-To-Peer Systems (IPTPS)*, 2002.

[11] 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.

[12] 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. 5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2013.

[13] Z. Wilcox-O'Hearn, "zfec," http://pypi.python.org/pypi/zfec.

[14] "SWIM Workload Repository," https://github.com/SWIMProjectUCB/SWIM/wiki/Workloads-repository.

[15] X. Zhang, J. Wang, and J. Yin, "Sapprox : Enabling Efficient and Accurate Approximations on Sub-datasets with Distribution-aware Online Sampling," *Proceedings of the VLDB Endowment*, vol. 10, no. 3, pp. 109–120, 2016.

[16] J. Wang, J. Yin, J. Zhou, X. Zhang, and R. Wang, "DataNet: A Data Distribution-Aware Method for Sub-Dataset Analysis on Distributed File Systems," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 504–513.

[17] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan, "The HP AutoRAID Hierarchical Storage System," *ACM Transactions on Computer Systems (TOCS)*, vol. 14, no. 1, pp. 108–136, 1996.

[18] W. Wang and H. Kuang, "Saving capacity with HDFS RAID," 2014, https://code.facebook.com/posts/536638663113101/saving-capacity-with-hdfs-raid/.

[19] 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.

[20] Y. Chen, S. Alspaugh, and R. Katz, "Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of MapReduce Workloads," *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 1802–1813, August 2012.

[21] D. S. Papailiopoulos, J. Luo, A. G. Dimakis, C. Huang, and J. Li, "Simple Regenerating Codes: Network Coding for Cloud Storage," in *Proc. IEEE INFOCOM*, 2012.

[22] D. Papailiopoulos and A. Dimakis, "Locally Repairable Codes," in *Proc. IEEE International Symposium on Information Theory Proceedings (ISIT)*, July 2012, pp. 2771–2775.

[23] F. Oggier and A. Datta, "Self-repairing Homomorphic Codes for Distributed Storage Systems," in *Proc. IEEE INFOCOM*, 2011.

[24] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A "HitchHiker's" Guide to Fast and Efficient Data Reconstruction in Erasure-coded Data Centers," in *Proc. ACM SIGCOMM*, 2014.

[25] C. Huang, M. Chen, and J. Li, "Pyramid Codes: Flexible Schemes to Trade Space for Access Efficiency in Reliable Data Storage Systems," *ACM Trans. Storage*, vol. 9, no. 1, pp. 3:1–3:28, March 2013.

[26] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, "Network Coding for Distributed Storage Systems," *IEEE Trans. Inform. Theory*, vol. 56, no. 9, pp. 4539–4551, 2010.

[27] D. Borthakur, "HDFS Architecture Guide," *Hadoop Apache Project*, http://hadoop.apache.org/common/docs/current/hdfs_design.pdf.

[28] J. Li, S. Yang, X. Wang, and B. Li, "Tree-Structured Data Regeneration in Distributed Storage Systems With Regenerating Codes," in *Proc. IEEE INFOCOM*, 2010.

[29] S. Mitra, R. Panta, M.-R. Ra, and S. Bagchi, "Partial-Parallel-Repair (PPR): A Distributed Technique for Repairing Erasure Coded Storage," in *Proc. European Conference on Computer Systems (Eurosys)*. ACM, 2016.

[30] R. Li, X. Li, P. P. C. Lee, and Q. Huang, "Repair Pipelining for Erasure-Coded Storage," in *Proc. USENIX Annual Technical Conference (USENIX ATC)*, Santa Clara, CA, 2017, pp. 567–579.

[31] S. Boyd and L. Vandenberghe, *Convex Optimization*. New York: Cambridge University Press, 2004.

[32] J. Blomer, M. Kalfane, M. Karpinski, R. Karp, M. Luby, and D. Zuckerman, "An XOR-based Erasure-resilient Coding Scheme," International Computer Science Institute, Tech. Rep. Technical Report TR-95-048, August 1995.

[33] J. Plank and L. Xu, "Optimizing Cauchy Reed-Solomon Codes for Fault-Tolerant Network Storage Applications," in *Proc. IEEE International Symposium on Network Computing and Applications*, July 2006, pp. 173–180.

Jun Li received his Ph.D. degree from the Department of Electrical and Computer Engineering, University of Toronto, in 2017, and his B.S. and M.S. degrees from the School of Computer Science, Fudan University, China, in 2009 and 2012. He is currently an assistant professor in the School of Computing and Information Sciences, Florida International University. His research interests include erasure codes and distributed storage systems.

Baochun Li received his Ph.D. degree from the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, in 2000. Since then, he has been with the Department of Electrical and Computer Engineering at the University of Toronto, where he is currently a Professor. He holds the Bell Canada Endowed Chair in Computer Engineering since August 2005. His research interests include large-scale distributed systems, cloud computing, peer-to-peer networks, applications of network coding, and wireless networks. He is a member of the ACM and a fellow of the IEEE.