Network Traffic Driven Storage Repair

Danilo Gligoroski

Dep. of Information Security and Communication Technology,
NTNU, Norwegian University of Science and Technology,
Trondheim, Norway
E-mail: danilog@ntnu.no

Katina Kralevska

Dep. of Information Security and Communication Technology,
NTNU, Norwegian University of Science and Technology,
Trondheim, Norway
E-mail: katinak@ntnu.no

Rune E. Jensen

Department of Computer Science,
NTNU, Norwegian University of Science and Technology,
Trondheim, Norway
E-mail: runeerle@idi.ntnu.no

Per Simonsen

MemoScale AS,
Norway
E-mail: per.simonsen@memoscale.com

Abstract: Recently we constructed an explicit family of locally repairable and locally regenerating codes. Their existence was proven by Kamath et al. but no explicit construction was given. Our design is based on HashTag codes that can have different sub-packetization levels. In this work we emphasize the importance of having two ways to repair a node: repair only with local parity nodes or repair with both local and global parity nodes. We say that the repair strategy is network traffic driven since it is in connection with the concrete system and code parameters: the repair bandwidth of the code, the number of I/O operations, the access time for the contacted parts and the size of the stored file. We show the benefits of having repair duality in one practical example implemented in Hadoop. We also give algorithms for efficient repair of the global parity nodes.

Keywords: Vector codes, Repair bandwidth, Repair locality, Exact repair, Parity-splitting, Global parities, Hadoop.

Reference to this paper should be made as follows: Gligoroski, D., Kralevska, K., Jensen R.E. and Simonsen, P. (xxxx) 'Network Traffic Driven Storage Repair', *International Journal of Big Data Intelligence*, Vol. x, No. x, pp.xxx–xxx.

Biographical notes: Danilo Gligoroski received his PhD in Computer Science at the Ss Cyril ad Methodius University in Skopje, Republic of Macedonia. He is a professor at the Dep. of Information Security and Communication Technology, Norwegian University of Science and Technology (NTNU). His research interests include .

Katina Kralevska received her PhD in Telematics at the Norwegian University of Science and Technology (NTNU). She is an associate professor at the same university, at the Dep. of Information Security and Communication Technology. Her research interest include

Rune E. Jensen is a PhD student at the Norwegian University of Science and Technology (NTNU). His research interest is an optimization ... .

Per Simonsen has MS degree in ... . He is currently CEO of MemoScale AS company that implements various erasure codes for Hadoop, CEPH, ...

1 Introduction

The omnipresence of digitalization in the modern human civilization resulted in exponential growth of the digital universe. According to Cisco Global Cloud Index (CGI) [1] the amount of all data stored by 2021 will be 7.2 ZB, with a proportion: 1.3 ZB stored in data centers, and 5.9 ZB stored in local devices. As a result of this, the importance of distributed storage systems rose to the level of being a backbone and a critical component for all existing infrastructures. Distributed storage systems became the crucial component for delivering IT services, providing storage services, enabling communications and networking to the users, devices and business processes [1].

In order to provide a reliable service, distributed storage systems use a simple data replication (usually triplication). Replication becomes very expensive in terms of storage overhead due to the enormous amount of data stored in these data centers (measured with hundreds of petabytes). One alternative for providing reliability in data storage systems with significantly less overhead is to use erasure codes. Several big providers of distributed storage such as Windows Azure [2] and Facebook Analytics Hadoop cluster [3] have implemented different erasure codes. Recently, the official Apache distribution of Hadoop 3.0.0 [4] has started to give an option to use several classical Reed-Solomon erasure codes such as $(5, 3)$, $(9, 6)$ and $(14, 10)$ codes in its file system HDFS. Reed-Solomon codes [5] are Maximum Distance Separable (MDS) codes, and thus they are optimal from the storage overhead point of view, but in practice they are expensive in the number of computations and in the amount of network traffic used in the recovery process. In 2010, Dimakis et al. [6] showed the existence of another family of MDS codes - Minimum Storage Regenerating codes that minimize the amount of data transmitted over the network for repairing one failed node with MDS codes.

While the benefits of using erasure codes instead of simple replication are obvious in terms of the storage overhead, there are other aspects that are not that favorable for erasure codes. One of them is the so called *repair efficiency*. In the case of a replication, the repair process is just a simple read (or copy) from the redundant data. On the other hand, the repair process with erasure coding involves data access from the non-failed nodes, transfer of the accessed data and decode computations at the node being repaired. It is essential to consider the concrete system and code parameters such as the repair bandwidth of the code, the number of I/O operations, the access time for the contacted parts and the size of the stored file when choosing the repair strategy with erasure codes. Arguably, we say the repair strategy is network traffic driven. In particular, there are two main metrics of the repair efficiency with erasure codes in distributed storage systems: the amount of transferred data during a repair process (*repair bandwidth*) and the number of accessed nodes in a repair process (*repair locality*). *Regenerating Codes* (RCs) [6] and *Locally Repairable Codes* (LRCs) [7],[8],[9] are optimized erasure codes for each of these two metrics, respectively.

In our recent work [10], we combine the benefits of RCs and LRCs together in one code construction. We construct an explicit family of locally repairable and locally regenerating codes whose existence was proven in a recent work by Kamath et al. [11] about codes with local regeneration. In that work, an existential proof was given, but no explicit construction was given. Our explicit family of codes is based on HashTag codes [12, 13]. HashTag codes are MDS vector codes with different vector length α (also called a sub-packetization level) that achieve the optimal repair bandwidth of MSR codes or near-optimal repair bandwidth depending on the sub-packetization level. We apply the technique of parity-splitting of HashTag codes in order to construct codes with locality in which the local codes are regenerating codes and which hence, enjoy both advantages of locally repairable codes as well as regenerating codes. It is observed in [14] that 98.08% of the failures in Facebook's data-warehouse cluster that consists of thousands of nodes are single failures. Thus, we optimize the repair for single failures although HashTag codes provide repair bandwidth savings for multiple failures as it is reported in [13].

We also show (although just with a concrete example) that the bound on the size of the finite field where these codes are constructed, given in the work by Kamath et al. [11] can be lower. The presented explicit code construction has a practical significance in distributed storage systems as it provides system designers with greater flexibility in terms of selecting various system and code parameters due to the flexibility of HashTag code constructions.

We discuss the repair duality and its importance. Repair duality is a situation of having two ways to repair a node: to repair it only with local parity nodes or repair it with both local and global parity nodes. To the best of the authors' knowledge, this is the first work that discusses the repair duality and how it can be applied based on concrete system and code parameters. Results from a Hadoop implementation illustrate the benefits of repair duality.

We further optimize the repair efficiency by giving an algorithm for shuffling the data in the parity nodes in order to reduce the repair bandwidth for the global parities.

The paper is organized as follows. Section 2 presents related work, and Section 3 presents mathematical preliminaries. In Section 4, we describe a framework for explicit constructions of locally repairable and locally regenerating codes. The repair process is analyzed in Section 5 where we explain the repair duality. In Section 6, we give an algorithm for efficient repair of the global parity nodes. Experimental results of measurements in Hadoop are given in Section 7. Conclusions are summarized in Section 8.

2 Related Work

Minimum Storage Regenerating (MSR) codes are an important class of RCs that minimize the amount of data stored per node, due to the MDS property, and the repair bandwidth. The reduction in the repair bandwidth is achieved on the cost of contacting $n-1$ nodes, i.e., increasing the locality. There have been several research directions for MDS storage codes. The tradeoff between the sub-packetization level and the repair bandwidth of MDS codes have been investigated in [12], [13], [15], [16],[17]. In [18], Goparaju et al. presented a construction of MSR codes for optimal repair of the systematic nodes for any number of accessed nodes $d \in \{k+1, \ldots, n-1\}$. The reduced locality comes at the cost of increased sub-packetization level. Another approach for constructing MSR codes with low sub-packetization level for different number of helper nodes is presented in [19]. Explicit constructions of MDS codes, including the MSR point, for optimal repair of the systematic nodes can be found in [12, 13]. Itani et al. used fractional repetition codes to minimize the total system recovery cost for single and multiple failures under various dynamic scenarios [20, 21].

On the other hand, LRCs relax the MDS requirement in order to minimize the number of nodes accessed during a repair. Studies on implementation and performance evaluation of LRCs can be found in [22, 2, 3, 23]. Locally repairable codes with multiple repair alternatives have been proposed in [24, 23], while codes with unequal locality have been presented in [25]. Zhang et al. presented local erasure recovery scheme by parity splitting of Vandermonde matrices [26] where the regular entries of a Vandermonde matrix enable low-complexity software implementations. Another way of constructing LRCs for distributed storage is based on low-density parity-check (LDPC) codes [27]. LDPC codes have an inherent local repair property as LRCs but their reliability increases with the code length that on the other hand has a negative impact on the computation and the buffer requirements.

Combining the benefits of RCs and LRCs in one storage system can bring huge savings in practical implementations. For instance, repair bandwidth savings by RCs are important when repairing huge amounts of data, while a fast recovery and an access to small number of nodes enabled by LRCs are desirable for repair of frequently accessed data. Several works present code constructions that combine the benefits of RCs and LRCs [28, 11, 29]. Rawat et al. in [28] and Kamath et al. in [11] have independently investigated codes with locality in the context of vector codes, and they call them locally repairable codes with local minimum storage regeneration (MSR-LRCs) and codes with local regeneration, respectively. Rawat et al. [28] provided an explicit construction, based on Gabidulin maximum rank-distance codes, of vector linear codes with all-symbol locality for the case when the local codes are MSR codes. However, the complexity of these codes increases exponentially with the number of nodes due to the two-stage encoding. In [11], Kamath et al. gave an existential proof without presenting an explicit construction. Another direction of combining RCs and LRCs is to use repair locality for selecting the accessed nodes in a RC [29], while an interpretation of LRCs as exact RCs was presented in [30]. Two different erasure codes, product and LRC codes, are used to dynamically adapt to the workload changes in Hadoop Adaptively-Coded Distributed File System (HACFS) [31]. Though there are several proposals for combining two different types of storage codes, they are lacking some of the desired attributes (see Table 1). Readers interested in an extended overview of erasure codes for distributed storage are referred to [32].

3 Mathematical Preliminaries

Inspired by the work of Gopalan et al. about locally repairable codes [7], Kamath et al. extended and generalized the concept of locality in [11]. In this paper, we use notation that is mostly influenced (and adapted) from those two papers. ***Notations***. For two integers $0 < i < j$, we denote the set $\{i, i+1, \ldots, j\}$ by $[i:j]$, while the set $\{1, 2, \ldots, j\}$ is denoted by $[j]$. Vectors and matrices are denoted with a bold font.

[11]: A $\mathbb{F}_q$-linear vector code of block length n is a code $\mathcal{C} \in \left(\mathbb{F}_q^{\alpha}\right)^n$ having a symbol alphabet $\mathbb{F}_q^{\alpha}$ for some $\alpha \geq 1$, i.e.,

$$\mathcal{C} = \{\mathbf{c} = (\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_n), \mathbf{c}_i \in \mathbb{F}_q^{\alpha} \text{ for all } i \in [n]\},$$

and satisfies the additional property that for given $\mathbf{c}, \mathbf{c}' \in \mathcal{C}$ and $a, b \in \mathbb{F}_q$,

$$a\mathbf{c} + b\mathbf{c}' = (a\mathbf{c}_1 + b\mathbf{c}'_1, a\mathbf{c}_2 + b\mathbf{c}'_2, \ldots, a\mathbf{c}_n + b\mathbf{c}'_n)$$

also belongs to $\mathcal{C}$ where $a\mathbf{c}_i$ is a scalar multiplication of the vector $\mathbf{c}_i$. ■

Throughout the paper, we refer to the vectors $\mathbf{c}_i$ as vector symbols or nodes. Working with systematic codes, it holds that for the systematic nodes $\mathbf{c}_i = \mathbf{d}_i$ for $1 \leq i \leq k$ and for the parity nodes $\mathbf{c}_{k+i} = \mathbf{p}_i$ for $1 \leq i \leq r$. For every vector code $\mathcal{C} \in \left(\mathbb{F}_q^{\alpha}\right)^n$ there is an associated scalar linear code $\mathcal{C}^{(s)}$ over $\mathbb{F}_q$ of length $N = \alpha n$. Accordingly, the dimension of the associated scalar code $\mathcal{C}^{(s)}$ is $K = \alpha k$. For a convenient notation, the generator matrix $\mathbf{G}$ of size $K \times N$ of the scalar code $\mathcal{C}^{(s)}$ is such that each of the α consecutive columns corresponds to one code symbol $\mathbf{c}_i, i \in [n]$, and they are considered as n thick columns $\mathbf{W}_i, i \in [n]$. For a subset $\mathcal{I} \subset [n]$ we say that it is an information set for $\mathcal{C}$ if the restriction $\mathbf{G}|_{\mathcal{I}}$ of $\mathbf{G}$ to the set of thick columns with indexes lying in $\mathcal{I}$ has a full rank, i.e., $\text{rank}(\mathbf{G}|_{\mathcal{I}}) = K$.

The minimum cardinality of an information set is referred as quasi-dimension κ of the vector code $\mathcal{C}$. As the vector code $\mathcal{C}$ is $\mathbb{F}_q$-linear, the minimum distance

Table 1 Comparison of locally repairable and locally regenerating codes presented in this paper with other erasure codes for storage.

Code	Systematic	Construction type	Order of GF	Any sub-packetization	Optimal repair of global parities
MSR-LRCs [28]	Yes	Explicit	High	No (only MSR point)	No
Codes with local regeneration [11]	Yes	Existence	High	No (only MSR point)	No
HACFS [31]	Yes	Explicit	Low	No	Product codes: Yes, LRC: No
This paper	Yes	Explicit	Low	Yes	Yes

$d_{\min}$ of $\mathcal{C}$ is equal to the minimum Hamming weight of a non-zero codeword in $\mathcal{C}$. Finally, a vector code of block length n, scalar dimension K, minimum distance $d_{\min}$, vector-length parameter α and quasi-dimension κ is shortly denoted with $[n, K, d_{\min}, \alpha, \kappa]$. While in the general definition of vector codes in [11] the quasi-dimension κ does not necessarily divide the dimension K of the associated scalar, for much simpler and convenient description of the codes in this paper we take that $k = \kappa$, i.e., $K = \alpha\kappa$. In that case the erasure and the Singleton bounds are given by:

$$d_{\min} \leq n - \kappa + 1. \tag{1}$$

In [6], Dimakis et al. studied the repair problem in a distributed storage system where a file of M symbols from a finite field $\mathbb{F}_q$ is stored across n nodes, each node stores $\frac{M}{k}$ symbols. They introduced the metric repair bandwidth γ, and proved that the repair bandwidth of a MDS code is lower bounded by

$$\gamma \geq \frac{M}{k}\frac{d}{d-k+1}, \tag{2}$$

where d is the number of accessed available nodes (helpers).

[6]: *The repair bandwidth of a (n, k) MDS code is minimized for $d = n - 1$. MSR codes achieve the lower bound of the repair bandwidth equal to*

$$\gamma_{MSR}^{min} = \frac{M}{k}\frac{n-1}{n-k}. \tag{3}$$

A (n, k) MSR code has the maximum possible distance $d_{min} = n - k + 1$ in addition to minimizing the repair bandwidth, but it has the worst possible locality.

Corollary 1: *The locality of a (n, k) MSR code is equal to $n - 1$.*

Any $[n, K, d_{\min}, \alpha, \kappa]$ vector code $\mathcal{C}$ is MDS if and only if its generator matrix can be represented in the form $\mathbf{G} = [\mathbf{I}|\mathbf{P}]$, where the $K \times (N - K)$ parity matrix

$$\mathbf{P} = \begin{bmatrix} \mathbf{G}_{1,1} & \mathbf{G}_{1,2} & \dots & \mathbf{G}_{1,\kappa} \\ \mathbf{G}_{2,1} & \mathbf{G}_{2,2} & \dots & \mathbf{G}_{2,\kappa} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{G}_{\kappa,1} & \mathbf{G}_{\kappa,2} & \dots & \mathbf{G}_{\kappa,n-\kappa} \end{bmatrix}, \tag{4}$$

possesses the property that every square block submatrix of $\mathbf{P}$ is invertible. The $\mathbf{G}_{i,j}$ entries are square submatrices of size $\alpha \times \alpha$, and a block submatrix is composed by different entries of $\mathbf{G}_{i,j}$.

In order to analyze codes with local regeneration, Kamath et al. introduced a new family of vector codes called uniform rank-accumulation (URA) codes in [11]. They showed that exact-repair MSR codes belong to the class of URA codes.

Definition 3.2: [11, Def. 2] Let $\mathcal{C}$ be a $[n, K, d_{\min}, \alpha, \kappa]$ vector code with a generator matrix $\mathbf{G}$. The code $\mathcal{C}$ is said to have (l, δ) information locality if there exists a set of punctured codes $\{\mathcal{C}_i\}_{i\in\mathcal{L}}$ of $\mathcal{C}$ with respective supports $\{S_i\}_{i\in\mathcal{L}}$ such that

- $|S_i| \leq l + \delta - 1$,
- $d_{\min}(\mathcal{C}_i) \geq \delta$, and
- $\text{rank}(\mathbf{G}|_{\bigcup_{i\in\mathcal{L}}}) = K$.

If we put $\delta = 2$ in Def.3.2, then we get the definition of information locality introduced by Gopalan et al. [7]. They derived the upper bound for the minimum distance of a $(n, k, d)_q$ code with information locality l for $\delta = 2$ as

$$d_{\min} \leq n - k - \left\lceil \frac{k}{l} \right\rceil + 2. \tag{5}$$

A general upper bound was derived in [11] as

$$d_{\min} \leq n - k + 1 - \left(\left\lceil \frac{k}{l} \right\rceil - 1 \right)(\delta - 1). \tag{6}$$

Huang et al. showed the existence of Pyramid codes that achieve the minimum distance given in (5) when the field size is big enough [22]. Finally, based on the work by Gopalan et al. [7] and Pyramid codes by Huang et al. [22], Kamath et al. proposed a construction of codes with local regeneration based on a parity-splitting strategy in [11].

4 Codes with Local Regeneration from HashTag Codes by Parity-Splitting

In [12, 13], a new class of vector MDS codes called HashTag codes is defined. HashTag codes achieve the lower bound of the repair bandwidth given in (3) for $\alpha = r^{\lceil \frac{k}{r} \rceil}$, while they have near-optimal repair bandwidth for small sub-packetization levels. HashTag codes are of a great practical importance due to their properties: flexible sub-packetization level, small repair bandwidth, and optimized number of I/O operations. We briefly give the basic definition of HashTag codes before we construct codes with local regeneration from them by using the framework of parity-splitting discussed in [11].

Definition 4.1: A $(n, k, d)_q$ HashTag linear code is a vector systematic code defined over an alphabet $\mathbb{F}_q^\alpha$ for some $\alpha \geq 1$. It encodes a vector $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_k)$, where $\mathbf{x}_i = (x_{1,i}, x_{2,i}, \ldots, x_{\alpha,i})^T \in \mathbb{F}_q^\alpha$ for $i \in [k]$, to a codeword $\mathcal{C}(\mathbf{x}) = \mathbf{c} = (\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_n)$ where the systematic parts $\mathbf{c}_i = \mathbf{x}_i$ for $i \in [k]$ and the parity parts $\mathbf{c}_i = (c_{1,i}, c_{2,i}, \ldots, c_{\alpha,i})^T$ for $i \in [k:n]$ are computed by the linear expressions that have a general form as follows:

$$c_{j,i} = \sum f_{\nu,j,i} x_{j_1,j_2}, \tag{7}$$

where $f_{\nu,j,i} \in \mathbb{F}_q$ and the index pair (j_1, j_2) is defined in the j-th row of the index array $\mathbf{P}_{i-r}$ where $\nu \in [r]$. The r index arrays $\mathbf{P}_1, \ldots, \mathbf{P}_r$ are defined as follows:

$$\mathbf{P}_1 = \begin{bmatrix} (1,1) & (1,2) & \ldots & (1,k) \\ (2,1) & (2,2) & \ldots & (2,k) \\ \vdots & \vdots & \ddots & \vdots \\ (\alpha,1) & (\alpha,2) & \ldots & (\alpha,k) \end{bmatrix},$$

$$\mathbf{P}_i = \begin{bmatrix} (1,1) & (1,2) & \ldots & (1,k) & \overbrace{(?,?) \ldots (?,?)}^{\lceil \frac{k}{r} \rceil} \\ (2,1) & (2,2) & \ldots & (2,k) & (?,?) \ldots (?,?) \\ \vdots & \vdots & \ddots & \vdots & \\ (\alpha,1) & (\alpha,2) & \ldots & (\alpha,k) & (?,?) \ldots (?,?) \end{bmatrix}.$$

where the values of the indexes $(?,?)$ are determined by a scheduling algorithm that guarantees the code is MDS, i.e., the entire information $\mathbf{x}$ can be recovered from any k out of the n vectors $\mathbf{c}_i$. ■

Algorithm 1 gives a high level description of one scheduling algorithm for Def. 4.1. An interested reader is referred to [12, 13] for more details.

Algorithm 1 High level description of an algorithm for generating HTEC for an arbitrary sub-packetization level

Input: n, k, α;
Output: Index arrays $\mathbf{P}_1, \ldots, \mathbf{P}_r$.

1: **Initialization:** $\mathbf{P}_1, \ldots, \mathbf{P}_r$ are initialized as index arrays $\mathbf{P} = ((i,j))_{\alpha \times k}$;
2: Append $\left\lceil \frac{k}{r} \right\rceil$ columns to $\mathbf{P}_2, \ldots, \mathbf{P}_r$ all initialized to $(0,0)$;
3: # Phase 1
4: Set the granulation level $run \leftarrow \left\lceil \frac{\alpha}{r} \right\rceil$ and $step \leftarrow 0$;
5: **repeat**
6: Replace $(0,0)$ pairs with indexes (i,j) such that both Condition 1 and Condition 2 are satisfied;
7: Decrease the granulation level run by a factor r and $step \leftarrow \left\lceil \frac{\alpha}{r} \right\rceil - run$;
8: **until** The granulation level $run > 1$
9: # Phase 2
10: If there are still $(0,0)$ and unscheduled elements from the systematic nodes, choose (i,j) such that only Condition 2 is satisfied;
11: Return the index arrays $\mathbf{P}_1, \ldots, \mathbf{P}_r$.

Example 1: *The linear expressions for the parity parts for a* $(9, 6)$ *HashTag code with* $\alpha = 9$ *are given here. The way how we obtain them is explained in Section 4.1 in [13]. We give one set of coefficients* $f_{\nu,j,i}$ *for equation (7) from the finite field* $\mathbb{F}_{32}$ *with irreducible polynomial* $x^5 + x^3 + 1$. *This code achieves the lower bound of repair bandwidth in (3), i.e., the repair bandwidth is* $\gamma = \frac{8}{3} = 2.67$ *for repair of any systematic node. Due to the big size* 54×81, *the systematic generator matrix of the associated scalar code is presented graphically in Fig. 1 instead of presenting it numerically.*

$$\begin{aligned}
c_{1,7} &= 7x_{1,1} + 10x_{1,2} + 18x_{1,3} + 11x_{1,4} + 17x_{1,5} + 6x_{1,6} \\
c_{2,7} &= 26x_{2,1} + 17x_{2,2} + 25x_{2,3} + 27x_{2,4} + 31x_{2,5} + 4x_{2,6} \\
c_{3,7} &= 22x_{3,1} + 12x_{3,2} + 27x_{3,3} + 31x_{3,4} + 31x_{3,5} + 23x_{3,6} \\
c_{4,7} &= 17x_{4,1} + 9x_{4,2} + 14x_{4,3} + 4x_{4,4} + 21x_{4,5} + 25x_{4,6} \\
c_{5,7} &= 20x_{5,1} + 5x_{5,2} + 5x_{5,3} + 13x_{5,4} + 11x_{5,5} + 16x_{5,6} \\
c_{6,7} &= 25x_{6,1} + 16x_{6,2} + 30x_{6,3} + 28x_{6,4} + 10x_{6,5} + 24x_{6,6} \\
c_{7,7} &= 20x_{7,1} + 8x_{7,2} + 21x_{7,3} + 9x_{7,4} + 3x_{7,5} + 25x_{7,6} \\
c_{8,7} &= 23x_{8,1} + 4x_{8,2} + 12x_{8,3} + 16x_{8,4} + 8x_{8,5} + 17x_{8,6} \\
c_{9,7} &= 2x_{9,1} + 21x_{9,2} + 8x_{9,3} + 16x_{9,4} + 7x_{9,5} + 25x_{9,6}
\end{aligned}$$

$$\begin{aligned}
c_{1,8} &= 8x_{1,1} + 24x_{1,2} + 21x_{1,3} + 19x_{1,4} + 6x_{1,5} + 20x_{1,6} + 8x_{4,1} + 6x_{2,4} \\
c_{2,8} &= 3x_{2,1} + 12x_{2,2} + 6x_{2,3} + 3x_{2,4} + 16x_{2,5} + 10x_{2,6} + 30x_{5,1} + 24x_{1,5} \\
c_{3,8} &= 23x_{3,1} + 20x_{3,2} + 30x_{3,3} + 7x_{3,4} + 16x_{3,5} + 10x_{3,6} + 21x_{6,1} + 27x_{1,6} \\
c_{4,8} &= 14x_{4,1} + 7x_{4,2} + 10x_{4,3} + 14x_{4,4} + 24x_{4,5} + 20x_{4,6} + 16x_{1,2} + 31x_{5,4} \\
c_{5,8} &= 25x_{5,1} + 11x_{5,2} + 29x_{5,3} + 12x_{5,4} + 20x_{5,5} + 24x_{5,6} + 15x_{2,2} + 6x_{4,5} \\
c_{6,8} &= 17x_{6,1} + 27x_{6,2} + 4x_{6,3} + 21x_{6,4} + 15x_{6,5} + 11x_{6,6} + 19x_{3,2} + 21x_{4,6} \\
c_{7,8} &= 19x_{7,1} + 23x_{7,2} + 16x_{7,3} + 4x_{7,4} + 14x_{7,5} + 16x_{7,6} + 9x_{1,3} + 8x_{8,4} \\
c_{8,8} &= 5x_{8,1} + 26x_{8,2} + 22x_{8,3} + 30x_{8,4} + 22x_{8,5} + 21x_{8,6} + 24x_{2,3} + 26x_{7,5} \\
c_{9,8} &= 10x_{9,1} + 8x_{9,2} + 10x_{9,3} + 27x_{9,4} + 28x_{9,5} + 20x_{9,6} + 16x_{3,3} + 4x_{7,6}
\end{aligned}$$

$$\begin{aligned}
c_{1,9} &= 20x_{1,1} + 20x_{1,2} + 30x_{1,3} + 17x_{1,4} + 12x_{1,5} + 27x_{1,6} + 28x_{7,1} + 9x_{3,4} \\
c_{2,9} &= 18x_{2,1} + 10x_{2,2} + 20x_{2,3} + 21x_{2,4} + 13x_{2,5} + 7x_{2,6} + 2x_{8,1} + 6x_{3,5} \\
c_{3,9} &= 31x_{3,1} + 25x_{3,2} + 12x_{3,3} + 18x_{3,4} + 15x_{3,5} + 24x_{3,6} + 31x_{9,1} + 28x_{2,6} \\
c_{4,9} &= 6x_{4,1} + 16x_{4,2} + 26x_{4,3} + 4x_{4,4} + 21x_{4,5} + 27x_{4,6} + 26x_{7,2} + 8x_{6,4} \\
c_{5,9} &= 7x_{5,1} + 6x_{5,2} + 26x_{5,3} + 6x_{5,4} + 15x_{5,5} + 16x_{5,6} + 28x_{8,2} + 4x_{6,5} \\
c_{6,9} &= 20x_{6,1} + 20x_{6,2} + 12x_{6,3} + 20x_{6,4} + 18x_{6,5} + 26x_{6,6} + 19x_{9,2} + 30x_{5,6} \\
c_{7,9} &= 26x_{7,1} + 2x_{7,2} + 6x_{7,3} + 20x_{7,4} + 17x_{7,5} + 23x_{7,6} + 8x_{4,3} + 31x_{9,4} \\
c_{8,9} &= 20x_{8,1} + 15x_{8,2} + 13x_{8,3} + 20x_{8,4} + 10x_{8,5} + 24x_{8,6} + 31x_{5,3} + 9x_{9,5} \\
c_{9,9} &= 6x_{9,1} + 2x_{9,2} + 31x_{9,3} + 12x_{9,4} + 16x_{9,5} + 30x_{9,6} + 20x_{6,3} + 13x_{8,6}
\end{aligned}$$

We adapt the parity-splitting code construction for designing codes with local regeneration described in [11] for the specifics of HashTag codes. The construction is described in Algorithm 2. For simplifying the description, we take some of the parameters to have

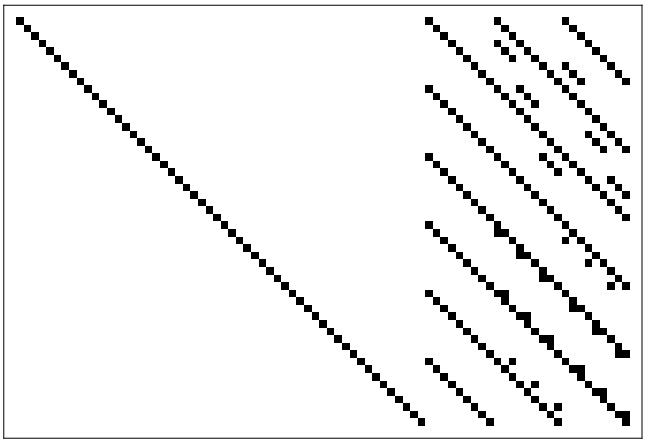

Figure 1: A systematic generator matrix of the associated scalar code. Here the black squares on the main diagonal represent the value 1, but the black squares in the parity parts represent the non-zero values in $\mathbb{F}_{32}$. Note that if the parity matrix is partitioned in 9×9 square submatrices, it has the same form as in equation (4).

specific relations, although it is possible to define a similar construction with general values of the parameters. Namely, we take that $r|k$ and $r|\alpha$. We also take that the parameters for the information locality (l, δ) are such that $l|k$ and $\delta \leq r$.

Algorithm 2 Locally Repairable HashTag Codes
Input: A (n, k) HashTag MDS code with a sub-packetization level α with the associated linear parity equations (7), i.e. with the associated systematic generator matrix $\mathbf{G}$. The MDS code can be, but it does not necessarily have to be a MSR code.
Input: The information locality (l, δ)
Output: A generator matrix $\mathbf{G}'$ with information locality (l, δ)

1: Split k systematic nodes into l disjunctive subsets $S_i, i \in [l]$, where every set has $\frac{k}{l}$ nodes. While this splitting can be arbitrary, take the canonical splitting where $S_1 = \{1, \ldots, \frac{k}{l}\}$, $S_2 = \{\frac{k}{l}+1, \ldots, \frac{2k}{l}\}$, ..., $S_l = \{\frac{(l-1)k}{l}+1, \ldots, k\}$.
2: Split each of the α linear equations for the first $\delta - 1$ parity expressions (7) into l sub-summands where the variables in each equation correspond to the elements from the disjunctive subsets.
3: Associate the obtained $\alpha \times l \times (\delta - 1)$ sub-summands to $l \times (\delta - 1)$ new local parity nodes.
4: Rename the remaining $r - \delta + 1$ parity nodes that were not split in Step 1 - Step 3 as new global parity nodes.
5: Obtain a new systematic generator matrix $\mathbf{G}'$ from the local and global parity nodes.
6: Return $\mathbf{G}'$ as a generator matrix of a $[n, K = k\alpha, d_{\min}, \alpha, k]$ vector code with information locality (l, δ).

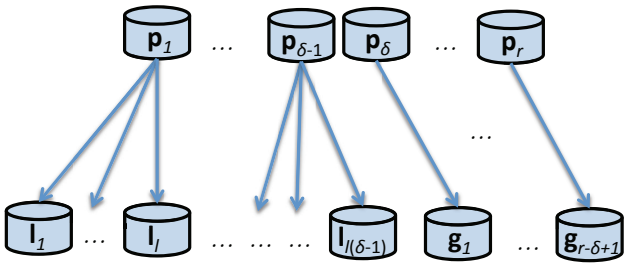

Figure 2: There are r parity nodes from a systematic (n, k) MDS code with a sub-packetization level α. The parity splitting technique generates l local parity nodes from every parity node $\mathbf{p}_1, \ldots, \mathbf{p}_{\delta-1}$ and renames the parity nodes $\mathbf{p}_\delta, \ldots, \mathbf{p}_r$ as global parity nodes $\mathbf{g}_1, \ldots, \mathbf{g}_{r-\delta+1}$.

A graphical presentation of the parity-splitting procedure is given in Fig. 2.

Theorem 2: *If the used (n, k) MDS HashTag code in Algorithm 2 is MSR, then the obtained $[n, K = k\alpha, d_{\min}, \alpha, k]$ code with information locality (l, δ) is a MSR-Local code, where*

$$d_{\min} = n - k + 1 - \left(\frac{k}{l} - 1\right)(\delta - 1). \tag{8}$$

Proof: Since in Algorithm 2 we take that $r|k$ and $r|\alpha$, it means that the scalar dimension of the code is $K = ml\alpha$ for some integer m. Then the proof continues basically as a technical adaptation of the proof of Theorem 5.5 that Kamath et al. gave for the pyramid-like MSR-Local codes constructed with the parity-splitting strategy in [11]. ■

Note that if $\alpha < r^{\frac{k}{r}}$, then HashTag codes are sub-optimal in terms of the repair bandwidth. Consequently, the produced codes with Algorithm 2 are locally repairable, but they are not MSR-Local codes.

Example 2: *Let us split the MSR code given in Example 1 into a code with local regeneration and with information locality $(l = 2, \delta = 2)$. In Step 1 we split 6 systematic nodes $\{\mathbf{c}_1, \ldots, \mathbf{c}_6\}$ into $l = 2$ disjunctive subsets $S_1 = \{\mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3\}$ and $S_2 = \{\mathbf{c}_4, \mathbf{c}_5, \mathbf{c}_6\}$. According to Step 2 of Algorithm 2, the first global parity $\mathbf{c}_7$ in Example 1 is split into two local parities $\mathbf{l}_1 = (l_{1,1}, \ldots, l_{9,1})^T$ and $\mathbf{l}_2 = (l_{1,2}, \ldots, l_{9,2})^T$ as follows:*

$$\begin{array}{ll}
l_{1,1} = \mathbf{7}x_{1,1} + \mathbf{10}x_{1,2} + \mathbf{18}x_{1,3} & l_{1,2} = \mathbf{11}x_{1,4} + \mathbf{17}x_{1,5} + \mathbf{6}x_{1,6} \\
l_{2,1} = \mathbf{26}x_{2,1} + \mathbf{17}x_{2,2} + \mathbf{25}x_{2,3} & l_{2,2} = \mathbf{27}x_{2,4} + \mathbf{31}x_{2,5} + \mathbf{4}x_{2,6} \\
l_{3,1} = \mathbf{22}x_{3,1} + \mathbf{12}x_{3,2} + \mathbf{27}x_{3,3} & l_{3,2} = \mathbf{31}x_{3,4} + \mathbf{31}x_{3,5} + \mathbf{23}x_{3,6} \\
l_{4,1} = \mathbf{17}x_{4,1} + \mathbf{9}x_{4,2} + \mathbf{14}x_{4,3} & l_{4,2} = \mathbf{4}x_{4,4} + \mathbf{21}x_{4,5} + \mathbf{25}x_{4,6} \\
l_{5,1} = \mathbf{20}x_{5,1} + \mathbf{5}x_{5,2} + \mathbf{5}x_{5,3} & l_{5,2} = \mathbf{13}x_{5,4} + \mathbf{11}x_{5,5} + \mathbf{16}x_{5,6} \\
l_{6,1} = \mathbf{25}x_{6,1} + \mathbf{16}x_{6,2} + \mathbf{30}x_{6,3} & l_{6,2} = \mathbf{28}x_{6,4} + \mathbf{10}x_{6,5} + \mathbf{24}x_{6,6} \\
l_{7,1} = \mathbf{20}x_{7,1} + \mathbf{8}x_{7,2} + \mathbf{21}x_{7,3} & l_{7,2} = \mathbf{9}x_{7,4} + \mathbf{3}x_{7,5} + \mathbf{25}x_{7,6} \\
l_{8,1} = \mathbf{23}x_{8,1} + \mathbf{4}x_{8,2} + \mathbf{12}x_{8,3} & l_{8,2} = \mathbf{16}x_{8,4} + \mathbf{8}x_{8,5} + \mathbf{17}x_{8,6} \\
l_{9,1} = \mathbf{2}x_{9,1} + \mathbf{21}x_{9,2} + \mathbf{8}x_{9,3} & l_{9,2} = \mathbf{16}x_{9,4} + \mathbf{7}x_{9,5} + \mathbf{25}x_{9,6}
\end{array}$$

The remaining two global parities are kept as they are given in Example 1, they are only renamed as $\mathbf{g}_1 = (c_{1,8}, c_{2,8}, \ldots, c_{9,8})^T$ and $\mathbf{g}_2 = (c_{1,9}, c_{2,9}, \ldots, c_{9,9})^T$. The

overall code is a $(10, 6)$ *code or with the terminology from [2] it is a* $(6, 2, 2)$ *code.* ■

Example 3: *Let us split the same MSR code now with parameters* $(l = 3, \delta = 2)$*. In Step 1 we split 6 systematic nodes* $\{\mathbf{c}_1, \ldots, \mathbf{c}_6\}$ *into* $l = 3$ *disjunctive subsets* $S_1 = \{\mathbf{c}_1, \mathbf{c}_2\}$*,* $S_2 = \{\mathbf{c}_3, \mathbf{c}_4\}$ *and* $S_3 = \{\mathbf{c}_5, \mathbf{c}_6\}$*. In Step 2 of Algorithm 2, the first global parity* $\mathbf{c}_7$ *is split into three local parities:* $\mathbf{l}_1 = (l_{1,1}, \ldots, l_{9,1})^T$*,* $\mathbf{l}_2 = (l_{1,2}, \ldots, l_{9,2})^T$ *and* $\mathbf{l}_3 = (l_{1,3}, \ldots, l_{9,3})^T$ *as follows:*

$$
\begin{array}{lll}
l_{1,1} = 7x_{1,1} + 10x_{1,2} & l_{1,2} = 18x_{1,3} + 11x_{1,4} & l_{1,3} = 17x_{1,5} + 6x_{1,6} \\
l_{2,1} = 26x_{2,1} + 17x_{2,2} & l_{2,2} = 25x_{2,3} + 27x_{2,4} & l_{2,3} = 31x_{2,5} + 4x_{2,6} \\
l_{3,1} = 22x_{3,1} + 12x_{3,2} & l_{3,2} = 27x_{3,3} + 31x_{3,4} & l_{3,3} = 31x_{3,5} + 23x_{3,6} \\
l_{4,1} = 17x_{4,1} + 9x_{4,2} & l_{4,2} = 14x_{4,3} + 4x_{4,4} & l_{4,3} = 21x_{4,5} + 25x_{4,6} \\
l_{5,1} = 20x_{5,1} + 5x_{5,2} & l_{5,2} = 5x_{5,3} + 13x_{5,4} & l_{5,3} = 11x_{5,5} + 16x_{5,6} \\
l_{6,1} = 25x_{6,1} + 16x_{6,2} & l_{6,2} = 30x_{6,3} + 28x_{6,4} & l_{6,3} = 10x_{6,5} + 24x_{6,6} \\
l_{7,1} = 20x_{7,1} + 8x_{7,2} & l_{7,2} = 21x_{7,3} + 9x_{7,4} & l_{7,3} = 3x_{7,5} + 25x_{7,6} \\
l_{8,1} = 23x_{8,1} + 4x_{8,2} & l_{8,2} = 12x_{8,3} + 16x_{8,4} & l_{8,3} = 8x_{8,5} + 17x_{8,6} \\
l_{9,1} = 2x_{9,1} + 21x_{9,2} & l_{9,2} = 8x_{9,3} + 16x_{9,4} & l_{9,3} = 7x_{9,5} + 25x_{9,6}
\end{array}
$$

The remaining two global parities are kept as they are given in Example 1, but they are just renamed as $\mathbf{g}_1 = (c_{1,8}, c_{2,8}, \ldots, c_{9,8})^T$ *and* $\mathbf{g}_2 = (c_{1,9}, c_{2,9}, \ldots, c_{9,9})^T$*. The overall code is a* $(11, 6)$ *code or with the terminology from [2] it is a* $(6, 3, 2)$ *code.* ■

There are two interesting aspects of Theorem 2 that should be emphasized: **1.** We give an explicit construction of an MSR-Local code (note that in [11] the construction is existential), and **2.** Examples 2 and 3 show that the size of the finite field can be slightly lower than the size proposed in [11]. Namely, the MSR HashTag code used in our example is defined over $\mathbb{F}_{32}$, while the lower bound in [11] suggests the field size to be bigger than $\binom{9}{6} = 84$. We consider this as a minor contribution and an indication that a deeper theoretical analysis can further lower the field size bound given in [11].

5 Repair Duality

Theorem 3: *Let* $\mathcal{C}$ *be a* (n, k) *MSR HashTag code with* $\gamma_{MSR}^{min} = \frac{M}{k}\frac{n-1}{n-k}$*. Further, let* $\mathcal{C}'$ *be a* $[n, K = k\alpha, d_{\min}, \alpha, k]$ *code with local regeneration and with information locality* (l, δ) *obtained by Algorithm 2. If we denote with* γ_{Local}^{min} *the minimum repair bandwidth for single systematic node repair with* $\mathcal{C}'$*, then*

$$\gamma_{Local}^{min} = \min\left(\frac{M}{k}\frac{\frac{k}{l} + \delta - 2}{\delta - 1}, \frac{M}{k}\frac{n-1}{n-k}\right). \tag{9}$$

Proof: When repairing one systematic node, we can always treat local nodes as virtual global nodes from which they have been constructed by splitting. Then with the use of other global nodes we have a situation of repairing one systematic node in the original MSR code for which the repair bandwidth is $\frac{M}{k}\frac{n-1}{n-k}$. On the other hand, if we use the MSR-Local code, then we have the following situation. There are $\frac{k}{l}$ systematic nodes in the MSR-Local code, and the total length of the MSR-Local code is $\frac{k}{l} + \delta - 1$. The file size for the MSR-Local code is decreased by a factor l, i.e., it is $\frac{M}{l}$. If we apply the MSR repair bandwidth for these values we get:

$$\frac{\frac{M}{l}}{\frac{k}{l}} \cdot \frac{\frac{k}{l} + (\delta - 1) - 1}{\delta - 1} = \frac{M}{k}\frac{\frac{k}{l} + \delta - 2}{\delta - 1}.$$

■

Theorem 3 is one of the main contributions of this work: It emphasizes the *repair duality* for repairing one systematic node: by the local and global parity nodes or only by the local parity nodes. We want to emphasize the practical importance of Theorem 3. Namely, in practical implementations regardless of the theoretical value of γ_{Local}^{min}, the number of I/O operations and the access time for the contacted parts can be either crucial or insignificant. In those cases an intelligent repair strategy implemented in the distributed storage system can decide which repair procedure should be used: the one with global parity nodes or the one with the local parity nodes.

While the repair bandwidth in equation (9) decreases by increasing the values of δ and l, it comes at the cost of decreasing the rate of the code for introducing the repair locality. We formalize this by the following Proposition.

Proposition 4: *If the input code in Algorithm 2 is* (n, k)*, then the output locally repairable code is*

$$(n + l \times (\delta - 1) - \delta + 1, k). \tag{10}$$

Proof: For a given initial code (n, k) the total number of parity nodes in the produced locally repairable codes is a sum of the parity nodes counted in Step 3 and Step 4 in Algorithm 2. The sum is

$$l \times (\delta - 1) + r - \delta + 1.$$

Thus the total number of nodes in the output locally repairable codes is

$$k + l \times (\delta - 1) + r - \delta + 1 = n + l \times (\delta - 1) - \delta + 1.$$

■

The penalty paid by the decremented rate of the final locally repairable code as described in the expression (10) stays linear in l if $\delta = 2$, but increases by a multiplicative factor $l \times (\delta - 1)$ if $\delta \geq 3$. That explains the reasons why in practical implementations of locally repairable codes such as those in Windows Azure [2] and the initial definition of locally repairable codes introduced in [7], the value of δ is kept low, i.e., $\delta = 2$.

To further illustrate different choices for introducing locality and the existence of repair duality for those codes, in Fig. 3 and Fig. 4 we plot the values for γ_{Local}^{min} from the expression (9) for two families of codes with $k = 8$ and $k = 12$. We took a normalized value $M = 1$. For comparative reasons, on the left sub-figure we plot the values for $\delta = 2$ and on the right sub-figure we plot the values for $\delta = 3$. While in principle, the values of l do not necessarily need to divide the value of k, in Algorithm 2

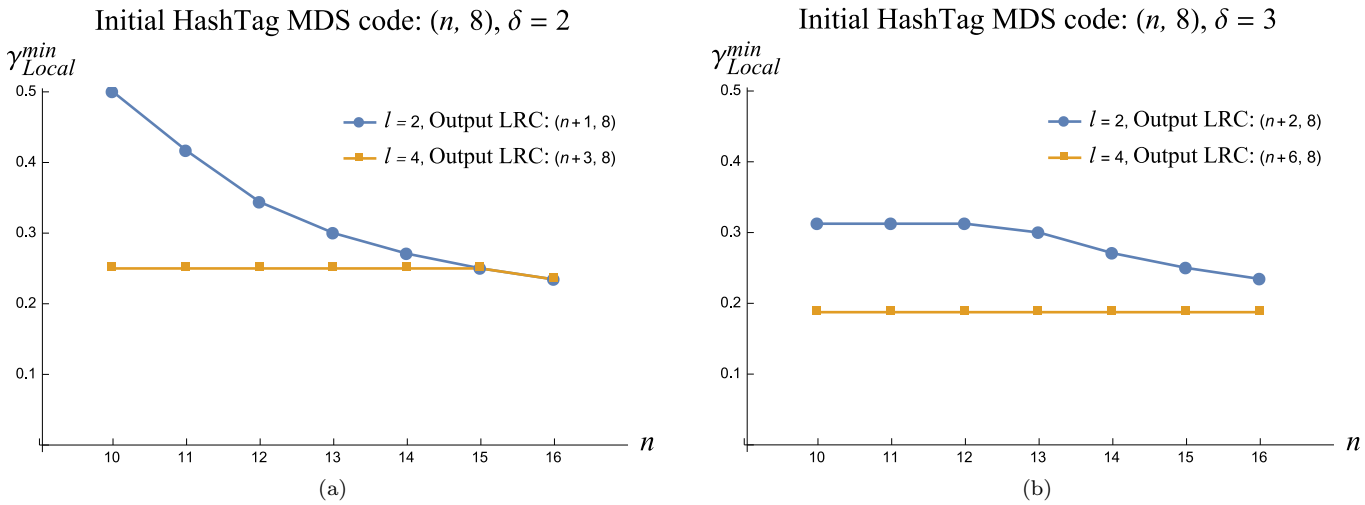

Figure 3: Repair bandwidth for one node for different codes where $k = 8$. The initial HashTag codes that are input in Algorithm 2 are $(n, 8)$ for different values of n. The left sub-figure (a) is for $\delta = 2$ and the right sub-figure (b) is for $\delta = 3$.

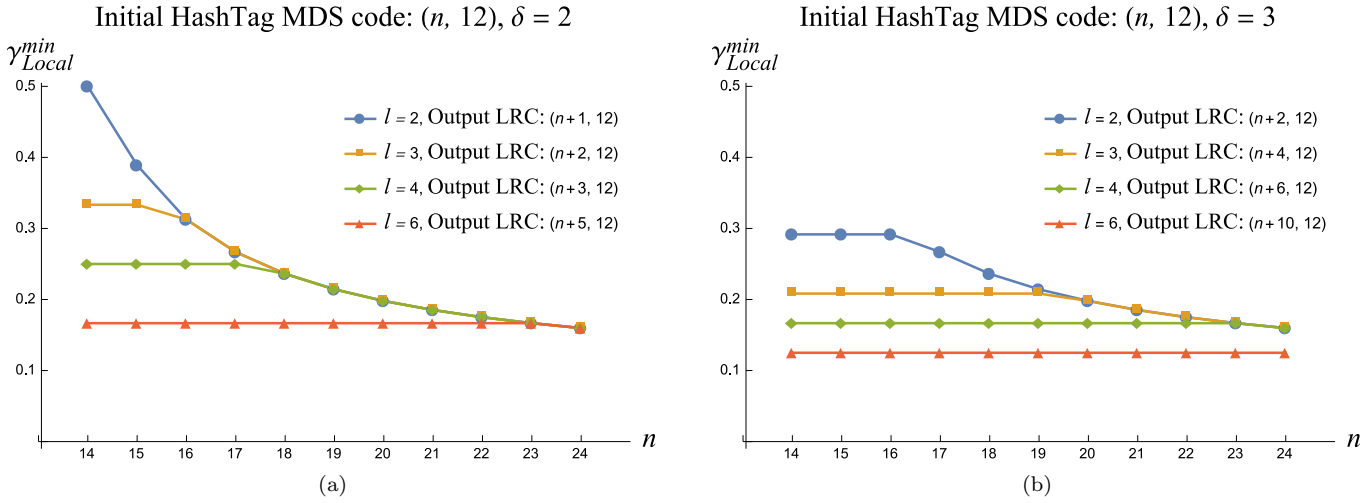

Figure 4: Repair bandwidth for one node for different codes where $k = 12$. The initial HashTag codes that are input in Algorithm 2 are $(n, 12)$ for different values of n. The left sub-figure (a) is for $\delta = 2$ and the right sub-figure (b) is for $\delta = 3$.

it is very convenient if actually l divides k. That is the reason why in Fig. 3 that is produced for (n, k) codes where $k = 8$ we plot the values for codes obtained for $l = 2$ and $l = 4$. Similarly, the values for l are $l = 2, 3, 4$ and 6 for the codes $(n, 12)$ in Fig. 4.

The flat lines in Fig. 3 and Fig. 4 mean that the repair bandwidth in expression (9) is achieved by using the local parity nodes, i.e., the value is $\frac{M}{k}\frac{\frac{k}{l}+\delta-2}{\delta-1}$. Since this expression does not depend on n, its value is a constant for different values of n. However, in cases when the minimum in expression (9) is achieved by using the global parity nodes, the repair bandwidth is $\frac{M}{k}\frac{n-1}{n-k}$ and the plot of the values γ^{min}_{Local} has a decreasing shape for increasing values of n.

Another important aspect that is illustrated by Fig. 3 and Fig. 4 is the price that is paid for achieving a low repair bandwidth with locally repairable codes. For example let us take an initial HashTag code $(n, k) = (10, 8)$. A LRC code produced with information locality $(l = 2, \delta = 2)$ is a $(n + 1, 8) = (11, 8)$ code and has a repair bandwidth of 0.5. By using information locality $(l = 4, \delta = 2)$, the LRC code is a $(n + 3, 8) = (13, 8)$ code, but the repair bandwidth drops down to 0.25. The situation with $\delta = 3$ decreases the repair bandwidth further, but worsens the code rate as well. In Fig. 3(b) an initial HashTag code $(n, k) = (10, 8)$ produces a LRC code $(n + 2, 8) = (12, 8)$ with a repair bandwidth of 0.3125 for $l = 2$, and produces a LRC code $(n + 6, 8) = (16, 8)$ with a repair bandwidth of just 0.1875 for $l = 4$.

We illustrate the benefits of having a choice how the repair is done (either by the local nodes or by the global nodes) by the following practical example.

Example 4: *Let us consider the* $(9, 6)$ *MSR HashTag code given in Example 1 and its corresponding local variant from Algorithm 2 with information locality* $(l = 2, \delta = 2)$ *given in Example 2. That means that the code with local regeneration has 6 systematic nodes, 2 local and 2 global parity nodes.*

Let us analyze the number of reads when we recover one unavailable systematic node. If we recover with the local nodes, then we have to perform 3 sequential reads, reading the whole data in a contiguous manner from 3 nodes. If we repair the unavailable data with the help of both local and global parity nodes, it reduces to the case of recovery with a MSR code, where the number of sequential reads is between 8 and 24 (average 16 reads) but the amount of transferred data is equivalent to 2.67 nodes.

More concretely, let us assume that we want to recover the node $\mathbf{x}_1 = (x_{1,1}, x_{2,1}, \ldots, x_{9,1})^T$.

1. *For a recovery only with the local parity* $\mathbf{l}_1$*, 3 sequential reads of* $\mathbf{l}_1$*,* $\mathbf{x}_2$ *and* $\mathbf{x}_3$ *are performed.*
2. *For a recovery with the local and global parities:*
   (a) *First, read* $l_{1,1}$*,* $l_{2,1}$ *and* $l_{3,1}$ *from* $\mathbf{l}_1$*, and* $x_{1,2}$*,* $x_{2,2}$ *and* $x_{3,2}$ *from* $\mathbf{x}_2$ *and* $x_{1,3}$*,* $x_{2,3}$ *and* $x_{3,3}$ *from* $\mathbf{x}_3$ *to recover* $x_{1,1}$*,* $x_{2,1}$ *and* $x_{3,1}$*.*
   (b) *Additionally, read* $x_{1,4}$*,* $x_{2,4}$ *and* $x_{3,4}$ *from* $\mathbf{x}_4$ *and* $x_{1,5}$*,* $x_{2,5}$ *and* $x_{3,5}$ *from* $\mathbf{x}_5$ *and* $x_{1,6}$*,* $x_{2,6}$ *and* $x_{3,6}$ *from* $\mathbf{x}_6$*.*
   (c) *Then, read* $c_{1,8}$*,* $c_{2,8}$ *and* $c_{3,8}$ *from the global parity* $\mathbf{g}_1$ *to recover* $x_{4,1}$*,* $x_{5,1}$ *and* $x_{6,1}$*.*
   (d) *Finally, read* $c_{1,9}$*,* $c_{2,9}$ *and* $c_{3,9}$ *from the global parity* $\mathbf{g}_2$ *to recover* $x_{7,1}$*,* $x_{8,1}$ *and* $x_{9,1}$*.*

Now, let a small file of 54 KB be stored across 6 systematic, 2 local and 2 global parity nodes. The sub-packetization level is $\alpha = 9$*, thus every node stores 9 KB, sub-packetized in 9 parts, each of size 1 KB. If the access time for starting a read operation is approximately the same as transferring 9 KB, then repairing with local and global parity nodes is more expensive since we have to perform in average 12 reads, although the amount of transferred data is equivalent to 2.67 nodes.*

On the other hand, let us have a big file of 540 MB stored across 6 systematic nodes and 2 local and 2 global parity nodes. The sub-packetization level is again $\alpha = 9$*, thus every node stores 90 MB, sub-packetized in 9 parts, each of size 10 MB. The access time for starting a read operation is again approximately the same as transferring 9 KB, which is insignificant in comparison with the total amount of transferred data in the process of repairing of a node. In this case, it is better to repair a failed node with local and global parity nodes since it requires a transfer of 240 MB versus the repair just with local nodes that requires a transfer of 270 MB.*

6 HashTag LRC Codes With Efficient Recovery of a Global Parity Node

HashTag codes as well as different MSR codes [18] and LRC codes [7, 8, 9] have the property that the bandwidth for a recovery of a failed parity node (global parity nodes in the case of LRC) is equal as in Reed-Solomon codes. That means the recovery of a failed parity node is not optimal and requires a bandwidth of k nodes.

Algorithm 3 Locally Reparable HashTag Codes with Efficient Recovery of a Global Parity Node
Input: Number of data nodes k, information locality $(l, 2)$ and number of global nodes g.
Output: A generator matrix $\mathbf{G}'$ with information locality $(l, 2)$, and sub-packetization level $\alpha = g$.

1: With Algorithm 1 produce a MDS HashTag code with k data nodes, $r = g + l - 1$ parity nodes and $\alpha = g$ substripes.
2: Get the corresponding index arrays $P_1, \ldots, P_r$.
3: Set global parity nodes $\{\mathbf{g}_1, \ldots, \mathbf{g}_\alpha\} = \{\mathbf{p}_2, \ldots, \mathbf{p}_r\}$.
4: Set the substripes of the node $\mathbf{g}_i$ as $\mathbf{g}_i = (g_{1,i}, \ldots, g_{\alpha,i})^T$.
5: Set the matrix of all global substripes

$$\mathbf{G}_{\alpha\times\alpha}[g_{i,j}] = \begin{bmatrix} g_{1,1} & g_{1,2} & \cdots & g_{1,\alpha} \\ g_{2,1} & g_{2,2} & \cdots & g_{2,\alpha} \\ \vdots & \vdots & \ddots & \vdots \\ g_{\alpha,1} & g_{\alpha,2} & \cdots & g_{\alpha,\alpha} \end{bmatrix}$$

6: Split k systematic nodes into l disjunctive subsets $S_i, i \in [l]$, where every set has $\frac{k}{l}$ nodes. While this splitting can be arbitrary, take the canonical splitting where $S_1 = \{1, \ldots, \frac{k}{l}\}$, $S_2 = \{\frac{k}{l} + 1, \ldots, \frac{2k}{l}\}$, ..., $S_l = \{\frac{(l-1)k}{l} + 1, \ldots, k\}$.
7: Split each of the α linear equations for the first parity expression in (7) into l sub-summands where the variables in each equation correspond to the elements from the disjunctive subsets.
8: Associate the obtained $\alpha \times l$ sub-summands to l new local parity nodes.
9: From $\mathbf{G}_{\alpha\times\alpha}[g_{i,j}]$ obtain a new systematic generator matrix $\mathbf{G'}_{\alpha\times\alpha}[g'_{i,j}]$ as follows:

$$g'_{i,j} = \begin{cases} g_{i,j} & \text{if } i = j, \\ f_{i,j,1}g_{i,j} + f_{i,j,2}g_{j,i} & \text{if } i < j, \\ f_{i,j,3}g_{i,j} + f_{i,j,4}g_{j,i} & \text{if } i > j, \end{cases} \tag{11}$$

where the coefficients $f_{i,j,1}, \ldots, f_{i,j,4} \in \mathbb{F}_q$ form 2×2 non-singular matrices $\begin{bmatrix} f_{i,j,1} & f_{i,j,2} \\ f_{i,j,3} & f_{i,j,4} \end{bmatrix}$.
10: Return $\mathbf{G}'$ as a generator matrix of a $[k + l + g, K = kg, d_{\min}, g, k]$ vector code with information locality $(l, 2)$.

The problem of recovery of a parity node that is optimal and achieves the MSR bound was recently solved by Tian et al. in [33]. Their approach is to take any (n, k) MDS code, where $r = n - k$, and then to increase the

sub-packetization level by a factor r. Thus, by producing a HashTag code with Algorithm 1, and applying first the technique from [33], and then the parity-splitting technique defined in Algorithm 2 we can construct HashTag LRC codes that can efficiently recover a global parity node. However, it comes at the cost of an increased sub-packetization level from α to $r\alpha$.

Here, for $\delta = 2$, i.e., for codes with information locality $(l, 2)$ we present another approach in Algorithm 3 that does not increase the sub-packetization level α of the initial HashTag code.

Theorem 5: *The bandwidth for repair of one global node produced by Algorithm 3 is equal to the bandwidth for repair of one data node of a MDS HashTag produced by Algorithm 1.*

Proof: Without a loss of generality let us assume that the lost global parity node is $\mathbf{g}_1 = (g'_{1,1}, \ldots, g'_{\alpha,1})^T$. From the relations (11) it follows that we can reconstruct $g'_{1,1}$ by reading the first k substripes from all k data nodes $x_{1,1}, \ldots, x_{\alpha,1}$ plus one element $x_{\mu,\nu}$ where the concrete values for μ and ν are obtained from the output of Algorithm 1. That is the same amount of bandwidth as with the recovery of the first substripe of a data node in the original HashTag.

For recovery of the substripe $g'_{2,1}$ we use equation (11) and we read the substripe $g'_{1,2}$ from the second global node. Since we already have read the values $x_{1,1}, \ldots, x_{\alpha,1}$, by using the coefficients $f_{1,2,1}$, $f_{1,2,2}$, $f_{1,2,3}$ and $f_{1,2,4}$ and possibly reading one extra stripe element $x_{\mu,\nu}$ where the concrete values for μ and ν are obtained from the output of Algorithm 1 we can compute the values $g_{i,j}$ and $g_{j,i}$, i.e., we compute the value $g'_{2,1} = f_{1,2,3}g_{1,2} + f_{1,2,4}g_{2,1}$. Total number of substripes read in this step is 2 (the same as in the original HashTag algorithm when recovering the second substripe of a data node).

The procedure then continues until the last substripe $g'_{\alpha,1}$. In every step the amount of substripes read is the same as in the original HashTag code when recovering a data node. ■

7 Experiments in Hadoop

The repair duality discussed in Example 4 of previous section was mainly influenced by one system characteristic: the access time for starting a read operation. In different environments of distributed storage systems there are several similar system characteristics that can affect the repair duality and its final optimal procedure. We next discuss this matter for Hadoop.

Hadoop is an open-source software framework used for distributed storage and processing of big data sets [34]. From release 3.0.0-alpha2 Hadoop offers several erasure codes such as $(5, 3)$, $(9, 6)$ and $(14, 10)$ Reed-Solomon (RS) codes. Hadoop Distributed File System (HDFS) has the concepts of *Splits* and *Blocks*. A Split is a logical representation of the data while a Block describes the physical alignment of data. Splits and Blocks in Hadoop are user defined: a logical split can be composed of multiple blocks and one block can have multiple splits. All these choices determine in a more complex way the access time for I/O operations.

To verify the performance of HashTag codes and their locally repairable and locally regenerating variants we implemented them in C/C++ and used them in HDFS.

For the code $(9, 6)$ we used one NameNode, nine DataNodes, and one client node. All nodes had a size of 50 GB and were connected with a local network of 10 Gbps. The nodes were running on Linux machines equipped with Intel Xeon E5-2676 v3 running on 2.4 GHz. We have experimented with different block sizes (90 MB and 360 MB), different split sizes (512 KB, 1 MB and 4 MB) and different sub-packetization levels ($\alpha = 1, 3, 6$, and 9) in order to check how they affect the repair time of one lost node. The measured times to recover one node are presented in Fig. 5. Note that the sub-packetization level $\alpha = 1$ represents the RS code that is available in HDFS, while for every other $\alpha = 3, 6, 9$ the codes are HashTag codes. In all measurements HashTag codes outperform RS. The cost of having significant number of I/O operations for the sub-packetization level $\alpha = 9$ is the highest for the smallest block and split size (Block size of 90 MB and Split size of 512 KB). This is shown by yellow bars. As the split sizes increase, the disadvantage of bigger number of I/Os due to the increased sub-packetization diminishes, and the repair time decreases further (red and blue bars).

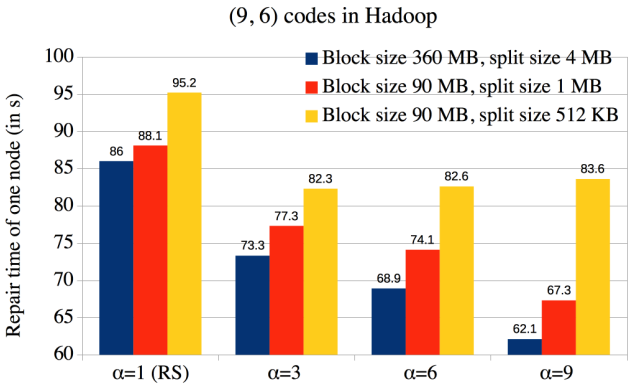

Figure 5: Experiments in HDFS. Time to repair one lost node of 50 GB with a $(9, 6)$ code for different sub-packetization levels α. Note that the RS code for $\alpha = 1$ is available in the latest release 3.0.0-alpha2 of Apache Hadoop.

In Fig. 6, we compare the repair times for one lost node of 50 GB with the codes from Examples 2 and 3. The cost of bigger redundancy with the locally repairable code $(10, 6)$ (which is also a locally regenerative code since the sub-packetization level is $\alpha = 9$) with $(l = 2, \delta = 2)$ (the red bar) is still not enough to outperform

the ordinary $(9, 6)$ HashTag MSR code (the blue bar). However, paying even higher cost by increasing the redundancy for the other locally regenerative code $(11, 6)$ with $(l = 3, \delta = 2)$ (the yellow bar) finally manages to outperform the repairing time for the ordinary $(9, 6)$ HashTag MSR code.

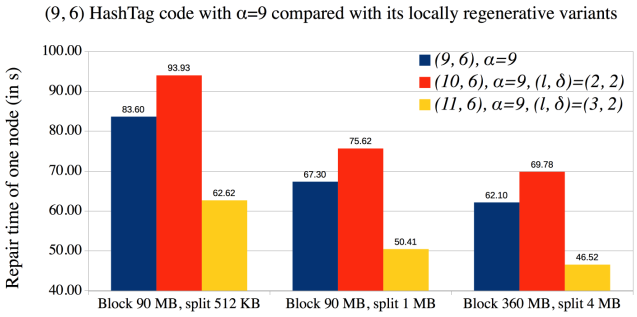

Figure 6: Comparison of repair times for one lost node of 50 GB for an ordinary $(9, 6)$ HashTag MSR code ($\alpha = 9$), and its locally regenerative variants with $(l = 2, \delta = 2)$ and $(l = 3, \delta = 2)$.

The situation of comparing the slightly less optimal $(9, 6)$ HashTag code with $\alpha = 6$ with its locally repairable variants with $(l = 2, \delta = 2)$ and $(l = 3, \delta = 2)$ is different, and this is presented in Fig. 7. In this case, all variants of locally repairable codes outperform the original HashTag code, i.e., they repair a failed node in shorter time than the original HashTag code from which they were constructed.

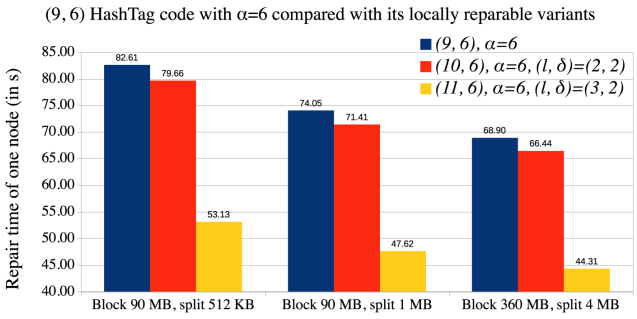

Figure 7: Comparison of repair times for one lost node of 50 GB for a $(9, 6)$ HashTag code with $\alpha = 6$, and its locally repairable variants with $(l = 2, \delta = 2)$ and $(l = 3, \delta = 2)$.

8 Conclusions

We constructed an explicit family of locally repairable and locally regenerating codes. We applied the technique of parity-splitting on HashTag codes and constructed codes with locality. For these codes we showed that there are two ways to repair a node (repair duality), and in practice which way is applied depends on optimization metrics such as the repair bandwidth, the number of I/O operations, the access time for the contacted parts and the size of the stored file. Additionally, we showed that the size of the finite field can be slightly lower than the theoretically obtained lower bound on the size in the literature. We solved the problem of efficient repair of the global parities when the number of global parities is equal to the sub-packetization level.

References

[1] Cisco. Cisco global cloud index: Forecast and methodology, 2016-2021. white paper, updated february 1, 2018. *Cisco Global Cloud Index (CGI)*, 2018.

[2] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure Coding in Windows Azure Storage. In *USENIX Annual Technical Conference*, pages 15–26, 2012.

[3] M. Sathiamoorthy, M. Asteris, D. S. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. XORing elephants: Novel erasure codes for big data. *Proc. VLDB Endow.*, 6(5):325–336, 2013.

[4] The Apache Software Foundation. *Welcome to Apache Hadoop!* The Apache Software Foundation, 2018.

[5] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, June 1960.

[6] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran. Network coding for distributed storage systems. *IEEE Trans. Inf. Theory*, 56(9):4539–4551, Sept. 2010.

[7] P. Gopalan, C. Huang, H. Simitci, and S. Yekhanin. On the locality of codeword symbols. *IEEE Trans. on Inf. Theory*, 58(11):6925–6934, 2012.

[8] F. E. Oggier and A. Datta. Self-repairing homomorphic codes for distributed storage systems. In *INFOCOM*, pages 1215–1223, 2011.

[9] D.S. Papailiopoulos, J. Luo, A.G. Dimakis, C. Huang, and J. Li. Simple regenerating codes: Network coding for cloud storage. In *IEEE INFOCOM*, pages 2801–2805, 2012.

[10] D. Gligoroski, K. Kralevska, R. E. Jensen, and P. Simonsen. Repair duality with locally repairable and locally regenerating codes. In *3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress*, pages 979–984, Nov 2017.

[11] G. M. Kamath, N. Prakash, V. Lalitha, and P. V. Kumar. Codes with local regeneration and erasure correction. *IEEE Trans. on Inf. Theory*, 60(8):4637–4660, Aug 2014.

[12] K. Kralevska, D. Gligoroski, and H. Øverby. General sub-packetized access-optimal regenerating codes. *IEEE Communications Letters*, 20(7):1281–1284, July 2016.

[13] K. Kralevska, D. Gligoroski, R. E. Jensen, and H. Overby. Hashtag erasure codes: From theory to practice. *IEEE Transactions on Big Data*, PP(99):1–1, 2017.

[14] K.V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A "hitchhiker's" guide to fast and efficient data reconstruction in erasure-coded data centers. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 331–342, 2014.

[15] A. Singh Rawat, I. Tamo, V. Guruswami, and K. Efremenko. MDS Code Constructions with Small Sub-packetization and Near-optimal Repair Bandwidth. *ArXiv e-prints*, September 2017.

[16] A. Chowdhury and A. Vardy. Improved schemes for asymptotically optimal repair of mds codes. In *55th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 950–957, 2017.

[17] K. Kralevska and D. Gligoroski. An Explicit Construction of Systematic MDS Codes with Small Sub-packetization for All-Node Repair. *ArXiv e-prints*, June 2018.

[18] S. Goparaju, A. Fazeli, and A. Vardy. Minimum storage regenerating codes for all parameters. *IEEE Transactions on Information Theory*, 63(10):6318–6328, Oct 2017.

[19] K. Mahdaviani, S. Mohajer, and A. Khisti. Product matrix msr codes with bandwidth adaptive exact repair. *IEEE Transactions on Information Theory*, PP(99):1–1, 2018.

[20] M. Itani, S. Sharafeddine, and I. Elkabani. Dynamic single node failure recovery in distributed storage systems. *Computer Networks*, 113:84 – 93, 2017.

[21] May Itani, Sanaa Sharafeddine, and Islam ElKabani. Dynamic multiple node failure recovery in distributed storage systems. *Ad Hoc Networks*, 72:1 – 13, 2018.

[22] Cheng Huang, Minghua Chen, and Jin Li. Pyramid codes: Flexible schemes to trade space for access efficiency in reliable data storage systems. *Trans. Storage*, 9(1):3:1–3:28, March 2013.

[23] K. Kralevska, D. Gligoroski, and H. Øverby. Balanced locally repairable codes. In *Int. Sym. on Turbo Codes and Iterative Inf. Processing (ISTC)*, pages 280–284, Sept 2016.

[24] L. Pamies-Juarez, H. D. L. Hollmann, and F. Oggier. Locally repairable codes with multiple repair alternatives. In *Information Theory Proceedings (ISIT), 2013 IEEE International Symposium on*, pages 892–896, July 2013.

[25] S. Kadhe and A. Sprintson. Codes with unequal locality. In *IEEE Int. Symp. on Inf. Theory (ISIT)*, pages 435–439, July 2016.

[26] X. Zhang, S. Sprouse, and I. Ilani. A flexible and low-complexity local erasure recovery scheme. *IEEE Communications Letters*, 20(11):2129–2132, Nov 2016.

[27] H. Park, D. Lee, and J. Moon. Ldpc code design for distributed storage: Balancing repair bandwidth, reliability, and storage overhead. *IEEE Transactions on Communications*, 66(2):507–520, Feb 2018.

[28] A. S. Rawat, O. O. Koyluoglu, N. Silberstein, and S. Vishwanath. Optimal locally repairable and secure codes for distributed storage systems. *IEEE Trans. on Inf. Theory*, 60(1):212–236, 2014.

[29] I. Ahmad and C. C. Wang. When locally repairable codes meet regenerating codes - what if some helpers are unavailable. In *IEEE Int. Symp. on Inf. Theory (ISIT)*, pages 849–853, June 2015.

[30] T. Ernvall, T. Westerbck, R. Freij-Hollanti, and C. Hollanti. A connection between locally repairable codes and exact regenerating codes. In *IEEE Int. Symp. on Inf. Theory (ISIT)*, pages 650–654, July 2016.

[31] Mingyuan Xia, Mohit Saxena, Mario Blaum, and David A. Pease. A tale of two erasure codes in HDFS. In *13th USENIX Conference on File and Storage Technologies (FAST)*, pages 213–226, 2015.

[32] S. B. Balaji, M. Nikhil Krishnan, M. Vajha, V. Ramkumar, B. Sasidharan, and P. Vijay Kumar. Erasure Coding for Distributed Storage: An Overview. *ArXiv e-prints*, June 2018.

[33] C. Tian, J. Li, and X. Tang. A generic transformation for optimal repair bandwidth and rebuilding access in mds codes. In *IEEE Int. Symposium on Inf. Theory (ISIT)*, pages 1623–1627, June 2017.

[34] Tom White. *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.