

# A Piggybacking Design Framework for Read-and Download-efficient Distributed Storage Codes

K. V. Rashmi, Nihar B. Shah, Kannan Ramchandran, *Fellow, IEEE*  
 Department of Electrical Engineering and Computer Sciences  
 University of California, Berkeley.  
 {rashmikv, nihar, kannanr}@eecs.berkeley.edu

## Abstract

We present a new *piggybacking* framework for designing distributed storage codes that are efficient in data-read and download required during node-repair. We illustrate the power of this framework by constructing classes of explicit codes that entail the smallest data-read and download for repair among all existing solutions for three important settings: (a) codes meeting the constraints of being Maximum-Distance-Separable (MDS), high-rate and having a small number of substripes, arising out of practical considerations for implementation in data centers, (b) binary MDS codes for all parameters where binary MDS codes exist, (c) MDS codes with the smallest repair-locality. In addition, we employ this framework to enable efficient repair of parity nodes in existing codes that were originally constructed to address the repair of only the systematic nodes. The basic idea behind our framework is to take multiple instances of existing codes and add carefully designed functions of the data of one instance to the other. Typical savings in data-read during repair is 25% to 50% depending on the choice of the code parameters.

## I. INTRODUCTION

Distributed storage systems today are increasingly employing erasure codes for data storage, since erasure codes provide much better storage efficiency and reliability as compared to replication-based schemes [1]–[3]. Frequent failures of individual storage nodes in these systems mandate schemes for efficient repair of failed nodes. In particular, upon failure of a node, it is replaced by a new node, which must obtain the data that was previously stored in the failed node by reading and downloading data from the remaining nodes. Two primary metrics that determine the efficiency of repair are the amount of data read at the remaining nodes (termed *data-read*) and the amount of data downloaded from them (termed *data-download* or simply the *download*).

	An MDS Code		Intermediate Step		Piggybacked Code	
Node 1	$a_1$	$b_1$	$a_1$	$b_1$	$a_1$	$b_1$
Node 2	$a_2$	$b_2$	$a_2$	$b_2$	$a_2$	$b_2$
Node 3	$a_3$	$b_3$	$a_3$	$b_3$	$a_3$	$b_3$
Node 4	$a_4$	$b_4$	$a_4$	$b_4$	$a_4$	$b_4$
Node 5	$\sum_{i=1}^4 a_i$	$\sum_{i=1}^4 b_i$	$\sum_{i=1}^4 a_i$	$\sum_{i=1}^4 b_i$	$\sum_{i=1}^4 a_i$	$\sum_{i=1}^4 b_i$
Node 6	$\sum_{i=1}^4 ia_i$	$\sum_{i=1}^4 ib_i$	$\sum_{i=1}^4 ia_i$	$\sum_{i=1}^4 ib_i + \sum_{i=1}^2 ia_i$	$\sum_{i=3}^4 ia_i - \sum_{i=1}^4 ib_i$	$\sum_{i=1}^4 ib_i + \sum_{i=1}^2 ia_i$
	(a)		(b)		(c)	

Fig. 1: An example illustrating efficient repair of systematic nodes using the piggybacking framework. Two instances of a (6,4) MDS code are piggybacked to obtain a new (6,4) MDS code that achieves 25% savings in data-read and download in the repair of any systematic node. A highlighted cell indicates a modified symbol.

Node 1	$a_1$	$b_1$	$c_1$	$d_1$
Node 2	$a_2$	$b_2$	$c_2$	$d_2$
Node 3	$a_3$	$b_3$	$c_3$	$d_3$
Node 4	$a_4$	$b_4$	$c_4$	$d_4$
Node 5	$\sum_{i=1}^4 a_i$	$\sum_{i=1}^4 b_i$	$\sum_{i=1}^4 c_i + \sum_{i=1}^4 ib_i + \sum_{i=1}^2 ia_i$	$\sum_{i=1}^4 d_i$
Node 6	$\sum_{i=3}^4 ia_i - \sum_{i=1}^4 ib_i$	$\sum_{i=1}^4 ib_i + \sum_{i=1}^2 ia_i$	$\sum_{i=3}^4 ic_i - \sum_{i=1}^4 id_i$	$\sum_{i=1}^4 id_i + \sum_{i=1}^2 ic_i$

Fig. 2: An example illustrating of the mechanism of repair of parity nodes under the piggybacking framework, using two instances of the code of Fig. 1c. A shaded cell indicates a modified symbol.

In this paper, we present a new framework, which we call the *piggybacking* framework, for design of repair-efficient storage codes. In a nutshell, this framework considers multiple instances of an existing code, and the piggybacking operation adds (carefully designed) functions of the data of one instance to the other. We design these functions with the goal of reducing the data-read and download requirements during repair. Piggybacking preserves many of the properties of the underlying code such as the minimum distance and the field of operation.

We need to introduce some notation and terminology at this point. Let  $n$  denote the number of (storage) nodes and assume that the nodes have equal storage capacities. The data to be stored across these nodes is termed the *message*. A *Maximum-Distance-Separable (MDS)* code is associated to another parameter  $k$ : an  $[n, k]$  MDS code guarantees that the message can be recovered from *any*  $k$  of the  $n$  nodes, and requires a storage capacity of  $\frac{1}{k}$  of the size of the message at every node. It follows that an MDS code can tolerate the failure of *any*  $(n - k)$  of the nodes without suffering any permanent data-loss. A *systematic* code is one in which  $k$  of the nodes store parts of the message without any coding. These  $k$  nodes are termed the systematic nodes and the remaining  $(n - k)$  nodes are termed the parity nodes. We denote the number of parity nodes by  $r = (n - k)$ . We shall assume without loss of generality that in a systematic code, the first  $k$  nodes are systematic. The *number of substripes* of a (vector) code is defined as the length of the vector of symbols that a node stores in a single instance of the code.

The piggybacking framework offers a rich design space for constructing codes for various different settings. We illustrate the power of this framework by providing the following four classes of explicit code constructions in this paper.

*(Class 1) A class of codes meeting the constraints of being MDS, high-rate, and having a small number of substripes, with the smallest known average data-read for repair:* A major component of the cost of current day data-centers which store enormous amounts of data is the storage hardware. This makes it critical for any storage code to minimize the storage space utilization. In light of this, it is important for the erasure code employed to be MDS and have a high-rate (i.e., a small storage overhead). In addition, practical implementations also mandate a small number of substripes. There has recently been considerable work on the design of distributed storage codes with efficient data-read during repair [4]–[26]. However, to the best of our knowledge, the only explicit codes that meet the aforementioned requirements are the Rotated-RS [8] codes and the (repair-optimized) EVENODD [25], [27] and RDP [26], [28] codes. Moreover, Rotated-RS codes exist only for  $r \in \{2, 3\}$  and  $k \leq 36$ ; the (repair-optimized) EVENODD and RDP codes exist only for  $r = 2$ . Through our piggybacking framework, we construct a class of codes that are MDS, high-rate, have a small number of substripes, and require the least amount of data-read and download for repair among all other known codes in this class. An appealing feature of our codes is that they support all values of the system parameters  $n$  and  $k$ .

*(Class 2) Binary MDS codes with the lowest known average data-read for repair, for all parameters where binary MDS codes exist:* Binary MDS codes are extensively used in disk arrays [27], [28]. Through our piggybacking framework, we construct binary MDS codes that require the lowest known average data-read for repair among all

existing binary MDS codes [8], [25]–[29]. Furthermore, unlike the other codes and repair algorithms [8], [25]–[29] in this class, the codes constructed here also optimize the repair of parity nodes (along with that of systematic nodes). Our codes support all the parameters for which binary MDS codes are known to exist.

(Class 3) *Efficient repair MDS codes with smallest possible repair-locality*: Repair-locality is the number of nodes that need to be read during repair of a node. While several recent works [20]–[23] present codes optimizing on locality, these codes are not MDS and hence require additional storage overhead for the same reliability levels as MDS codes. In this paper, we present MDS codes with efficient repair properties that have the smallest possible repair-locality for an MDS code.

(Class 4) *A method of reducing data-read and download for repair of parity nodes in existing codes that address only the repair of systematic nodes*: The problem of efficient node-repair in distributed storage systems has attracted considerable attention in the recent past. However, many of the codes proposed [7]–[9], [19], [30] have algorithms for efficient repair of *only* the systematic nodes, and require the download of the entire message for repair of any parity node. In this paper, we employ our piggybacking framework to enable efficient repair of parity nodes in these codes, while also retaining the efficiency of repair of systematic nodes. The corresponding piggybacked codes enable an average saving of 25% to 50% in the amount of download and read required for repair of parity nodes.

The following examples highlight the key ideas behind the piggybacking framework.

*Example 1: This example illustrates one method of piggybacking for reducing data-read during systematic node repair. Consider two instances of a (6, 4) MDS code as shown in Fig. 1a, with the 8 message symbols  $\{a_i\}_{i=1}^4$  and  $\{b_i\}_{i=1}^4$  (each column of Fig. 1a depicts a single instance of the code). One can verify that the message can be recovered from the data of any 4 nodes. The first step of piggybacking involves adding  $\sum_{i=1}^2 ia_i$  to the second symbol of node 6 as shown in Fig. 1b. The second step in this construction involves subtracting the second symbol of node 6 in the code of Fig. 1b from its first symbol. The resulting code is shown in Fig. 1c. This code has 2 substripes (the number of columns in Fig. 1c).*

*We now present the repair algorithm for the piggybacked code of Fig. 1c. Consider the repair of node 1. Under our repair algorithm, the symbols  $b_2, b_3, b_4$  and  $\sum_{i=1}^4 b_i$  are download from the other nodes, and  $b_1$  is decoded. In addition, the second symbol ( $\sum_{i=1}^4 ib_i + \sum_{i=1}^2 ia_i$ ) of node 6 is downloaded. Subtracting out the components of  $\{b_i\}_{i=1}^4$  gives the piggyback  $\sum_{i=1}^2 ia_i$ . Finally, the symbol  $a_2$  is downloaded from node 2 and subtracted to obtain  $a_1$ . Thus, node 1 is repaired by reading only 6 symbols which is 75% of the total size of the message. Node 2 can be repaired in a similar manner. Repair of nodes 3 and 4 follows on similar lines except that the first symbol of node 6 is read instead of the second.*

*The piggybacked code is MDS, and the entire message can be recovered from any 4 nodes as follows. If node 6 is one of these four nodes, then add its second symbol to its first, to recover the code of Fig. 1b. Now, the decoding algorithm of the original code of Fig. 1a is employed to first recover  $\{a_i\}_{i=1}^4$ , which then allows for removal of the piggyback ( $\sum_{i=1}^2 ia_i$ ) from the second substripe, making the remainder identical to the code of Fig. 1c.*

*Example 2: This example illustrates the use of piggybacking to reduce data-read during the repair of parity nodes. The code depicted in Fig. 2 takes two instances of the code of Fig. 1c, and adds the second symbol of node 6, ( $\sum_{i=1}^4 ib_i + \sum_{i=1}^2 ia_i$ ) (which belongs to the first instance), to the third symbol of node 5 (which belongs to the second instance). This code has 4 substripes (the number of columns in Fig. 2). In this code, repair of the second parity node involves downloading  $\{a_i, c_i, d_i\}_{i=1}^4$  and the modified symbol ( $\sum_{i=1}^4 c_i + \sum_{i=1}^4 ib_i + \sum_{i=1}^2 ia_i$ ), using which the data of node 6 can be recovered. The repair of the second parity node thus requires read and download of only 13 symbols instead of the entire message of size 16. The first parity is repaired by downloading all 16 message symbols. Observe that in the code of Fig. 1c, the first symbol of node 5 is not used for repair of any of the systematic nodes. Thus the modification in Fig. 2 does not change the algorithm or the efficiency of the repair of systematic nodes. The code retains its MDS property: the entire message can be recovered from any 4 nodes by*

first decoding  $\{a_i, b_i\}_{i=1}^4$  using the decoding algorithm of the code of Fig. 1a, which then allows for removal of the piggyback  $(\sum_{i=1}^4 ib_i + \sum_{i=1}^2 ia_i)$  from the second instance, making the remainder identical to the code of Fig. 1a.

Our piggybacking framework, enhances existing codes by adding piggybacks from one instance onto the other. The design of these piggybacks determine the properties of the resulting code. In this paper, we provide a few designs of piggybacking and specialize it to existing codes to obtain the four specific classes mentioned above. This framework, while being powerful, is also simple, and easily amenable for code constructions in other settings and scenarios.

The rest of the paper is organized as follows. Section II introduces the general piggybacking framework. Sections III and IV then present code designs and repair-algorithms based on this framework, special cases of which result in classes 1 and 2 discussed above. Section V provides piggyback design which result in low-repair locality along with low data-read and download. Section VI provides a comparison of these codes and various other codes in the literature. Section VII demonstrates the use of piggybacking to enable efficient parity repair in existing codes that were originally constructed for repair of only the systematic nodes. Section VIII draws conclusions.

Readers interested only in repair locality may skip Sections III and IV, and readers interested only in the mechanism of imbibing efficient parity-repair in existing codes optimized for systematic-repair may skip Sections III, IV, and V without any loss in continuity.

## II. THE PIGGYBACKING FRAMEWORK

The piggybacking framework operates on an existing code, which we term the *base code*. The choice of the base code is arbitrary. The base code is associated to  $n$  encoding functions  $\{f_i\}_{i=1}^n$ : it takes the message  $\mathbf{u}$  as input and encodes it to  $n$  coded symbols  $\{f_1(\mathbf{u}), \dots, f_n(\mathbf{u})\}$ . Node  $i$  ( $1 \leq i \leq n$ ) stores the data  $f_i(\mathbf{u})$ .

The piggybacking framework operates on multiple instances of the base code, and embeds information about one instance into other instances in a specific fashion. Consider  $\alpha$  instances of the base code. The encoded symbols in  $\alpha$  instances of the base code are

Node 1	$f_1(\mathbf{a})$	$f_1(\mathbf{b})$	$\dots$	$f_1(\mathbf{z})$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
Node $n$	$f_n(\mathbf{a})$	$f_n(\mathbf{b})$	$\dots$	$f_n(\mathbf{z})$

where  $\mathbf{a}, \dots, \mathbf{z}$  are the (independent) messages encoded under these  $\alpha$  instances.

We shall now describe the piggybacking of this code. For every  $i$ ,  $2 \leq i \leq \alpha$ , one can add an arbitrary function of the message symbols of all previous instances  $\{1, \dots, (i-1)\}$  to the data stored under instance  $i$ . These functions are termed *piggyback* functions, and the values so added are termed *piggybacks*. Denoting the piggyback functions by  $g_{i,j}$  ( $i \in \{2, \dots, \alpha\}$ ,  $j \in \{1, \dots, n\}$ ), the piggybacked code is thus:

Node 1	$f_1(\mathbf{a})$	$f_1(\mathbf{b}) + g_{2,1}(\mathbf{a})$	$f_1(\mathbf{c}) + g_{3,1}(\mathbf{a}, \mathbf{b})$	$\dots$	$f_1(\mathbf{z}) + g_{\alpha,1}(\mathbf{a}, \dots, \mathbf{y})$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
Node $n$	$f_n(\mathbf{a})$	$f_n(\mathbf{b}) + g_{2,n}(\mathbf{a})$	$f_n(\mathbf{c}) + g_{3,n}(\mathbf{a}, \mathbf{b})$	$\dots$	$f_n(\mathbf{z}) + g_{\alpha,n}(\mathbf{a}, \dots, \mathbf{y})$

The decoding properties (such as the minimum-distance or the MDS nature) of the base code are retained upon piggybacking. In particular, the piggybacked code allows for decoding of the entire message from any set of nodes from which the base code allowed decoding. To see this, consider any set of nodes from which the message can be recovered in the base code. Observe that the first column of the piggybacked code is identical to a single instance of the base code. Thus  $\mathbf{a}$  can be recovered directly using the decoding procedure of the base code. The piggyback functions  $\{g_{2,i}(\mathbf{a})\}_{i=1}^n$  can now be subtracted from the second column. The remainder of this column

is precisely another instance of the base code, allowing recovery of  $\mathbf{b}$ . Continuing in the same fashion, for any instance  $i$  ( $2 \leq i \leq n$ ), the piggybacks (which are always a function of previously decoded instances  $\{1, \dots, i-1\}$ ) can be subtracted out to obtain the base code of that instance which can be decoded.

The decoding properties of the code are thus not hampered by the choice of the piggyback functions  $g_{i,j}$ 's. This allows for flexibility in the choice of the piggyback functions, and these need to be picked cleverly to achieve the desired goals (such as efficient repair, which is the focus of this paper).

The piggybacking procedure described above was followed in Example 1 to obtain the code of Fig. 1b from Fig. 1a. Subsequently, in Example 2, this procedure was followed again to obtain the code of Fig. 2 from Fig. 1c.

The piggybacking framework also allows any invertible linear transformation of the data stored in any individual node. In other words, each node of the piggybacked code (e.g., each row in Fig. 1b) can separately undergo an invertible transformation. Clearly, any invertible transformation of data within the nodes does not alter the decoding capabilities of the code, i.e., the message can still be recovered from any set of nodes from which it could be recovered in the base code. In Example 1, the code of Fig. 1c is obtained from Fig. 1b via an invertible transformation of the data of node 6.

The following theorem formally proves that piggybacking does not reduce the amount of information stored in any subset of nodes.

*Theorem 1:* Let  $U_1, \dots, U_\alpha$  be random variables corresponding to the messages associated to the  $\alpha$  instances of the base code. For  $i \in \{1, \dots, n\}$ , let  $X_i$  denote the data stored in node  $i$  under the base code. Let  $Y_i$  denote the encoded symbols stored in node  $i$  under the piggybacked version of that code. Then for any subset of nodes  $S \subseteq \{1, \dots, n\}$ ,

$$I(\{Y_i\}_{i \in S}; U_1, \dots, U_\alpha) \geq I(\{X_i\}_{i \in S}; U_1, \dots, U_\alpha) . \quad (1)$$

The proof of this theorem is provided in the appendix.

*Corollary 2:* Piggybacking a code does not decrease its minimum distance; piggybacking an MDS code preserves the MDS property.

*Notational Conventions:* For simplicity of exposition, we shall assume throughout this section that the base codes are linear, scalar, MDS and systematic. Using vector codes (such as EVENODD or RDP) as base codes is a straightforward extension. The base code operates on a  $k$ -length message vector, with each symbol of this vector drawn from some finite field. The number of instances of the base code during piggybacking is denoted by  $\alpha$ , and  $\{\mathbf{a}, \mathbf{b}, \dots\}$  shall denote the  $k$ -length message vectors corresponding to the  $\alpha$  instances. Since the code is systematic, the first  $k$  nodes store the elements of the message vector. We use  $\mathbf{p}_1, \dots, \mathbf{p}_r$  to denote the  $r$  encoding vectors corresponding to the  $r$  parity symbols, i.e., if  $\mathbf{a}$  denotes the  $k$ -length message vector then the  $r$  parity nodes under the base code store  $\mathbf{p}_1^T \mathbf{a}, \dots, \mathbf{p}_r^T \mathbf{a}$ .

The transpose of a vector or a matrix will be indicated by a superscript  $T$ . Vectors are assumed to be column vectors. For any vector  $\mathbf{v}$  of length  $\kappa$ , we denote its  $\kappa$  elements as  $\mathbf{v} = [v_1 \ \dots \ v_\kappa]^T$ , and if the vector itself has an associated subscript then we its elements as  $\mathbf{v}_i = [v_{i,1} \ \dots \ v_{i,\kappa}]^T$ .

Each of the explicit codes constructed in this paper possess the property that the repair of any node entails reading of only as much data as what has to be downloaded.<sup>1</sup> This property is called *repair-by-transfer* [24]. Thus the amounts of data-read and download are equal under our codes, and hence we shall use the same notation  $\gamma$  to denote both these quantities.

<sup>1</sup>In general, the amount of download lower bounds the amount of read, and the download could be strictly smaller if a node passes a (non-injective) function of the data that it stores.

### III. PIGGYBACKING DESIGN 1

In this section, we present our first design of piggyback functions and associated repair algorithms. This design allows one to reduce data-read and download during repair while having a small number of substripes. For instance, when the number of substripes is small as 2, we can achieve a 25 to 35% savings during repair of systematic nodes. We shall first present the piggyback design for optimizing the repair of systematic nodes, and then move on to the repair of parity nodes.

#### A. Efficient repair of systematic nodes

This design operates on  $\alpha = 2$  instances of the base code. We first partition the  $k$  systematic nodes into  $r$  sets,  $S_1, \dots, S_r$ , of equal size (or nearly equal size if  $k$  is not a multiple of  $r$ ). For ease of understanding, let us assume that  $k$  is a multiple of  $r$ , which fixes the size of each of these sets as  $\frac{k}{r}$ . Then, let  $S_1 = \{1, \dots, \frac{k}{r}\}$ ,  $S_2 = \{\frac{k}{r} + 1, \dots, \frac{2k}{r}\}$  and so on, with  $S_i = \{\frac{(i-1)k}{r} + 1, \dots, \frac{ik}{r}\}$  for  $i = 1, \dots, r$ .

Define the following  $k$ -length vectors:

$$\begin{aligned} \mathbf{q}_2 &= [ p_{r,1} \cdots p_{r,\frac{k}{r}} \quad 0 \cdots \quad \cdots \quad \cdots \quad \cdots 0 \quad ]^T \\ \mathbf{q}_3 &= [ \quad 0 \cdots 0 \quad p_{r,\frac{k}{r}+1} \cdots p_{r,\frac{2k}{r}} \quad 0 \cdots \quad \cdots \quad \cdots 0 \quad ]^T \\ &\vdots \\ \mathbf{q}_r &= [ \quad 0 \cdots \quad \cdots \quad \cdots 0 \quad p_{r,\frac{k}{r}(r-2)+1} \cdots p_{r,\frac{k}{r}(r-1)} \quad 0 \cdots 0 \quad ]^T \\ \mathbf{q}_{r+1} &= [ \quad 0 \cdots \quad \cdots \quad \cdots \quad \cdots 0 \quad \cdots 0 \quad p_{r,\frac{k}{r}(r-1)+1} \cdots p_{r,k} \quad ]^T . \end{aligned}$$

Also, let

$$\begin{aligned} \mathbf{v}_r &= \mathbf{p}_r - \mathbf{q}_r \\ &= [ \quad p_{r,1} \cdots \quad \cdots \quad \cdots p_{r,\frac{k}{r}(r-2)} \quad 0 \cdots 0 \quad p_{r,\frac{k}{r}(r-1)+1} \cdots p_{r,k} \quad ]^T . \end{aligned}$$

Note that each element  $p_{i,j}$  is non-zero since the base code is MDS. We shall use this property during repair operations.

The base code is piggybacked in the following manner:

Node 1	$a_1$	$b_1$
$\vdots$	$\vdots$	$\vdots$
Node k	$a_k$	$b_k$
Node k+1	$\mathbf{p}_1^T \mathbf{a}$	$\mathbf{p}_1^T \mathbf{b}$
Node k+2	$\mathbf{p}_2^T \mathbf{a}$	$\mathbf{p}_2^T \mathbf{b} + \mathbf{q}_2^T \mathbf{a}$
$\vdots$	$\vdots$	$\vdots$
Node k+r	$\mathbf{p}_r^T \mathbf{a}$	$\mathbf{p}_r^T \mathbf{b} + \mathbf{q}_r^T \mathbf{a}$

Fig. 1b depicts an example of such a piggybacking.

We shall now perform an invertible transformation of the data stored in node  $(k+r)$ . In particular, the first symbol of node  $(k+r)$  in the code above is replaced with the difference of this symbol from its second symbol, i.e., node  $(k+r)$  now stores

$$\text{Node } k+r \quad \boxed{\mathbf{v}_r^T \mathbf{a} - \mathbf{p}_r^T \mathbf{b} \quad \mathbf{p}_r^T \mathbf{b} + \mathbf{q}_r^T \mathbf{a}}$$

The other symbols in the code remain intact. This completes the description of the encoding process.

Next, we present the algorithm for repair of any systematic node  $\ell$  ( $\in \{1, \dots, k\}$ ). This entails recovery of the two symbols  $a_\ell$  and  $b_\ell$  from the remaining nodes.

*Case 1* ( $\ell \notin S_r$ ): Without loss of generality let  $\ell \in S_1$ . The  $k$  symbols  $\{b_1, \dots, b_{\ell-1}, b_{\ell+1}, \dots, b_k, \mathbf{p}_1^T \mathbf{b}\}$  are downloaded from the remaining nodes, and the entire vector  $\mathbf{b}$  is decoded (using the MDS property of the base code). It now remains to recover  $a_\ell$ . Observe that the  $\ell^{\text{th}}$  element of  $\mathbf{q}_2$  is non-zero. The symbol  $(\mathbf{p}_2^T \mathbf{b} + \mathbf{q}_2^T \mathbf{a})$  is downloaded from node  $(k+2)$ , and since  $\mathbf{b}$  is completely known,  $\mathbf{p}_2^T \mathbf{b}$  is subtracted from the downloaded symbol to obtain the piggyback  $\mathbf{q}_2^T \mathbf{a}$ . The symbols  $\{a_i\}_{i \in S_1 \setminus \{\ell\}}$  are also downloaded from the other systematic nodes in set  $S_1$ . The specific (sparse) structure of  $\mathbf{q}_2$  allows for recovering  $a_\ell$  from these downloaded symbols. Thus the total data-read and download during the repair of node  $\ell$  is  $(k + \frac{k}{r})$  (in comparison, the size of the message is  $2k$ ).

*Case 2* ( $S = S_r$ ): As in the previous case,  $\mathbf{b}$  is completely decoded by downloading  $\{b_1, \dots, b_{\ell-1}, b_{\ell+1}, \dots, b_k, \mathbf{p}_1^T \mathbf{b}\}$ . The first symbol  $(\mathbf{v}_r^T \mathbf{a} - \mathbf{p}_r^T \mathbf{b})$  of node  $(k+r)$  is downloaded. The second symbols  $\{\mathbf{p}_i^T \mathbf{b} + \mathbf{q}_i^T \mathbf{a}\}_i$  stored in the parities  $i \in \{(k+2), \dots, (k+r-1)\}$  are also downloaded, and are then subtracted from the first symbol of node  $(k+r)$ . This gives  $(\mathbf{q}_{r+1}^T \mathbf{a} + \mathbf{w}^T \mathbf{b})$  for some vector  $\mathbf{w}$ . Using the previously decoded value of  $\mathbf{b}$ ,  $\mathbf{v}^T \mathbf{b}$  is removed to obtain  $\mathbf{q}_{r+1}^T \mathbf{a}$ . Observe that the  $\ell^{\text{th}}$  element of  $\mathbf{q}_{r+1}$  is non-zero. The desired symbol  $a_\ell$  can thus be recovered by downloading  $\{a_{\frac{k}{r}(r-1)+1}, \dots, a_{\ell-1}, a_{\ell+1}, \dots, a_k\}$  from the other systematic nodes in  $S_r$ . The total data-read and download required in recovering node  $\ell$  is  $(k + \frac{k}{r} + r - 2)$ .

Observe that the repair of systematic nodes in the last set  $S_r$  requires more read and download as compared to repair of systematic nodes in the other sets. Given this observation, we do not choose the sizes of the sets to be equal (as described previously), and instead optimize the sizes to minimize the average read and download required. For  $i = 1, \dots, r$ , denoting the size of the set  $S_i$  by  $t_i$ , the optimal sizes of the sets turn out to be

$$t_1 = \dots = t_{r-1} = \left\lceil \frac{k}{r} + \frac{r-2}{2r} \right\rceil := t, \quad (2)$$

$$t_r = k - (r-1)t. \quad (3)$$

The amount of data read and downloaded for repair of any systematic node in the first  $(r-1)$  sets is  $(k+t)$ , and the last set is  $(k+t_r+r-2)$ . Thus, the average data-read and download  $\gamma_1^{\text{sys}}$  for repair of systematic nodes, as a fraction of the total number  $2k$  of message symbols, is

$$\gamma_1^{\text{sys}} = \frac{1}{2k^2} [(k-t_r)(k+t) + t_r(k+t_r+r-2)]. \quad (4)$$

This quantity is plotted in Fig. 5a for various values of the system parameters  $n$  and  $k$ .

### B. Reducing data-read during repair of parity nodes

We shall now piggyback the code constructed in Section III-A to introduce efficiency in the repair of parity nodes, while also retaining the efficiency in the repair of systematic nodes. Observe that in the code of Section III-A, the first symbol of node  $(k+1)$  is never read for repair of any systematic node. We shall add piggybacks to this unused parity symbol to aid in the repair of other parity nodes.

This design employs  $m$  instances of the piggybacked code of Section III-A. The number of substripes in the resultant code is thus  $2m$ . The choice of  $m$  can be arbitrary, and higher values of  $m$  result in greater repair-efficiency. For every instance  $i \in \{2, 4, \dots, 2m-2\}$ , the  $(r-1)$  parity symbols in nodes  $(k+2)$  to  $(k+r)$  are summed up. The result is added as a piggyback to the  $(i+1)^{\text{th}}$  symbol of node  $(k+1)$ . The resulting code, when  $m = 2$ , is shown below.

Node 1	$a_1$	$b_1$	$c_1$	$d_1$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
Node k	$a_k$	$b_k$	$c_k$	$d_k$
Node k+1	$\mathbf{p}_1^T \mathbf{a}$	$\mathbf{p}_1^T \mathbf{b}$	$\mathbf{p}_1^T \mathbf{c} + \sum_{i=2}^r (\mathbf{p}_i^T \mathbf{b} + \mathbf{q}_i^T \mathbf{a})$	$\mathbf{p}_1^T \mathbf{d}$
Node k+2	$\mathbf{p}_2^T \mathbf{a}$	$\mathbf{p}_2^T \mathbf{b} + \mathbf{q}_2^T \mathbf{a}$	$\mathbf{p}_2^T \mathbf{c}$	$\mathbf{p}_2^T \mathbf{d} + \mathbf{q}_2^T \mathbf{c}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
Node k+r-1	$\mathbf{p}_{r-1}^T \mathbf{a}$	$\mathbf{p}_{r-1}^T \mathbf{b} + \mathbf{q}_{r-1}^T \mathbf{a}$	$\mathbf{p}_{r-1}^T \mathbf{c}$	$\mathbf{p}_{r-1}^T \mathbf{d} + \mathbf{q}_{r-1}^T \mathbf{c}$
Node k+r	$\mathbf{v}_r^T \mathbf{a} - \mathbf{p}_r^T \mathbf{b}$	$\mathbf{p}_r^T \mathbf{b} + \mathbf{q}_r^T \mathbf{a}$	$\mathbf{v}_r^T \mathbf{c} - \mathbf{p}_r^T \mathbf{d}$	$\mathbf{p}_r^T \mathbf{d} + \mathbf{q}_r^T \mathbf{c}$

This completes the encoding procedure. The code of Fig. 2 is an example of this design.

As shown in Section II, the piggybacked code retains the MDS property of the base code. In addition, the repair of systematic nodes is identical to the that in the code of Section III-A, since the symbol modified in this piggybacking was never read for the repair of any systematic node in the code of Section III-A.

We now present an algorithm for efficient repair of parity nodes under this piggyback design. The first parity node is repaired by downloading all  $2mk$  message symbols from the systematic nodes. Consider repair of some other parity node, say node  $\ell \in \{k+2, \dots, k+r\}$ . All message symbols  $\mathbf{a}, \mathbf{c}, \dots$  in the odd substripes are downloaded from the systematic nodes. All message symbols of the last substripe (e.g., message  $\mathbf{d}$  in the  $m=2$  code shown above) are also downloaded from the systematic nodes. Further, the  $\{3^{\text{rd}}, 5^{\text{th}}, \dots, (2m-1)^{\text{th}}\}$  symbols of node  $(k+1)$  (i.e., the symbols that we modified in the piggybacking operation above) are also downloaded, and the components corresponding to the already downloaded message symbols are subtracted out. By construction, what remains in the symbol from substripe  $i$  ( $i \in \{3, 5, \dots, 2m-1\}$ ) is the piggyback. This piggyback is a sum of the parity symbols of the substripe  $(i-1)$  from the last  $(r-1)$  nodes (including the failed node). The remaining  $(r-2)$  parity symbols belonging to each of the substripes  $\{2, 4, \dots, 2m-2\}$  are downloaded and subtracted out, to recover the data of the failed node. The procedure described above is illustrated via the repair of node 6 in Example 2.

The average data-read and download  $\gamma_1^{\text{par}}$  for repair of parity nodes, as a fraction of the total message symbols, is

$$\gamma_1^{\text{par}} = \frac{1}{2kr} \left[ 2k + (r-1) \left( \left(1 + \frac{1}{m}\right) k + \left(1 - \frac{1}{m}\right) (r-1) \right) \right].$$

This quantity is plotted in Fig. 5b for various values of the system parameters  $n$  and  $k$ .

#### IV. PIGGYBACKING DESIGN 2

The design presented in this section provides a higher efficiency of repair as compared to the previous design. On the downside, it requires a larger number of substripes: the minimum number of substripes required under the design of Section III-A is 2 and under that of Section III-B is 4, while that required in the design of this section is  $(2r-3)$ . The following example illustrates this piggybacking design.

*Example 3: Consider some  $(n=13, k=10)$  MDS code as the base code, and consider  $\alpha = (2r-3) = 3$  instances of this code. Divide the systematic nodes into two sets of sizes 5 each as  $S_1 = \{1, \dots, 5\}$  and  $S_2 =$*

$\{6, \dots, 10\}$ . Define 10-length vectors  $\mathbf{q}_2$ ,  $\mathbf{v}_2$ ,  $\mathbf{q}_3$  and  $\mathbf{v}_3$  as

$$\begin{aligned}\mathbf{q}_2 &= [p_{2,1} \ \cdots \ p_{2,5} \ 0 \ \cdots \ 0] \\ \mathbf{v}_2 &= [0 \ \cdots \ 0 \ p_{2,6} \ \cdots \ p_{2,10}] \\ \mathbf{q}_3 &= [0 \ \cdots \ 0 \ p_{3,6} \ \cdots \ p_{3,10}] \\ \mathbf{v}_3 &= [p_{3,1} \ \cdots \ p_{3,5} \ 0 \ \cdots \ 0]\end{aligned}$$

Now piggyback the base code in the following manner

Node 1	$a_1$	$b_1$	$c_1$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
Node 10	$a_{10}$	$b_{10}$	$c_{10}$
Node 11	$\mathbf{p}_1^T \mathbf{a}$	$\mathbf{p}_1^T \mathbf{b}$	$\mathbf{p}_1^T \mathbf{c}$
Node 12	$\mathbf{p}_2^T \mathbf{a}$	$\mathbf{p}_2^T \mathbf{b} + \mathbf{q}_2^T \mathbf{a}$	$\mathbf{p}_2^T \mathbf{c} + \mathbf{q}_2^T \mathbf{b} + \mathbf{q}_2^T \mathbf{a}$
Node 13	$\mathbf{p}_3^T \mathbf{a}$	$\mathbf{p}_3^T \mathbf{b} + \mathbf{q}_3^T \mathbf{a}$	$\mathbf{p}_3^T \mathbf{c} + \mathbf{q}_3^T \mathbf{b} + \mathbf{q}_3^T \mathbf{a}$

Next, we take invertible transformations of the (respective) data of nodes 12 and 13. The second symbol of node  $i \in \{12, 13\}$  in the new code is the difference between the second and the third symbols of node  $i$  in the code above. The fact that  $(\mathbf{p}_2 - \mathbf{q}_2) = \mathbf{v}_2$  and  $(\mathbf{p}_3 - \mathbf{q}_3) = \mathbf{v}_3$  results in the following code

Node 1	$a_1$	$b_1$	$c_1$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
Node 10	$a_{10}$	$b_{10}$	$c_{10}$
Node 11	$\mathbf{p}_1^T \mathbf{a}$	$\mathbf{p}_1^T \mathbf{b}$	$\mathbf{p}_1^T \mathbf{c}$
Node 12	$\mathbf{p}_2^T \mathbf{a}$	$\mathbf{v}_2^T \mathbf{b} - \mathbf{p}_2^T \mathbf{c}$	$\mathbf{p}_2^T \mathbf{c} + \mathbf{q}_2^T \mathbf{b} + \mathbf{q}_2^T \mathbf{a}$
Node 13	$\mathbf{p}_3^T \mathbf{a}$	$\mathbf{v}_3^T \mathbf{b} - \mathbf{p}_3^T \mathbf{c}$	$\mathbf{p}_3^T \mathbf{c} + \mathbf{q}_3^T \mathbf{b} + \mathbf{q}_3^T \mathbf{a}$

This completes the encoding procedure.

We now present an algorithm for (efficient) repair of any systematic node, say node 1. The 10 symbols  $\{c_2, \dots, c_{10}, \mathbf{p}_1^T \mathbf{c}\}$  are downloaded, and  $\mathbf{c}$  is decoded. It now remains to recover  $a_1$  and  $b_1$ . The third symbol ( $\mathbf{p}_2^T \mathbf{c} + \mathbf{q}_2^T \mathbf{b} + \mathbf{q}_2^T \mathbf{a}$ ) of node 12 is downloaded and  $\mathbf{p}_2^T \mathbf{c}$  subtracted out to obtain  $(\mathbf{q}_2^T \mathbf{b} + \mathbf{q}_2^T \mathbf{a})$ . The second symbol ( $\mathbf{v}_3^T \mathbf{b} - \mathbf{p}_3^T \mathbf{c}$ ) from node 13 is downloaded and  $(-\mathbf{p}_3^T \mathbf{c})$  is subtracted out from it to obtain  $\mathbf{v}_3^T \mathbf{b}$ . The specific (sparse) structure of  $\mathbf{q}_2$  and  $\mathbf{v}_3$  allows for decoding  $a_1$  and  $b_1$  from  $(\mathbf{q}_2^T \mathbf{b} + \mathbf{q}_2^T \mathbf{a})$  and  $\mathbf{v}_3^T \mathbf{b}$ , by downloading and subtracting out  $\{a_i\}_{i=2}^5$  and  $\{b_i\}_{i=2}^5$ . Thus, the repair of node 1 involved reading and downloading 20 symbols (in comparison, the size of the message is  $k\alpha = 30$ ). The repair of any other systematic node follows a similar algorithm, and results in the same amount of data-read.

The general design is as follows. Consider  $(2r - 3)$  instances of the base code, and let  $\mathbf{a}_1, \dots, \mathbf{a}_{2r-3}$  be the messages associated to the respective instances. First divide the  $k$  systematic nodes into  $(r - 1)$  equal sets (or nearly equal sets if  $k$  is not a multiple of  $(r - 1)$ ). Assume for simplicity of exposition that  $k$  is a multiple of  $(r - 1)$ . The first of the  $\frac{k}{r-1}$  sets consist of the first  $\frac{k}{r-1}$  nodes, the next set consists of the next  $\frac{k}{r-1}$  nodes and so on. Define  $k$ -length vectors  $\{\mathbf{v}_i, \hat{\mathbf{v}}_i\}_{i=2}^r$  as

$$\begin{aligned}\mathbf{v}_i &= \mathbf{a}_{r-1} + i\mathbf{a}_{r-2} + i^2\mathbf{a}_{r-3} + \cdots + i^{r-2}\mathbf{a}_1 \\ \hat{\mathbf{v}}_i &= \mathbf{v}_i - \mathbf{a}_{r-1} = i\mathbf{a}_{r-2} + i^2\mathbf{a}_{r-3} + \cdots + i^{r-2}\mathbf{a}_1.\end{aligned}$$



While we only discussed the repair of systematic nodes for this code, the repair of parity nodes can be made efficient by considering  $m$  instances of this code. A procedure analogous to that described in Section III-B is followed, where the odd instances are piggybacked on to the succeeding even instances. As in Section III-B, higher a value of  $m$  results in a lesser amount of data-read and download for repair. In such a design, the average data-read and download  $\gamma_2^{\text{par}}$  for repair of parity nodes, as a fraction of the total message symbols, is

$$\gamma_2^{\text{par}} = \frac{1}{r} + \frac{r-1}{2r-3} \left[ (m+1)(r-2)k + (m-1)(r-2)(r-1) + \left\lceil \frac{m}{2} \right\rceil k + \left\lfloor \frac{m}{2} \right\rfloor (r-1) \right]. \quad (7)$$

This quantity is also plotted in Fig. 5b for various values of the system parameters  $n$  and  $k$ .

## V. PIGGYBACKING DESIGN 3

In this section, we present a piggybacking design to construct MDS codes with a primary focus on the *locality* of repair. The locality of a repair operation is defined as the number of nodes that are contacted during the repair operation. The codes presented here perform the efficient repair of any systematic node with the smallest possible locality for any MDS code, which is equal to  $(k+1)$ .<sup>2</sup> The amount of read and download is the smallest among all known MDS codes with this locality, when  $(n-k) > 2$ .

This design involves two levels of piggybacking, and these are illustrated in the following two example constructions. The first example considers  $\alpha = 2m$  instances of the base code and shows the first level of piggybacking, for any arbitrary choice of  $m > 1$ . Higher values of  $m$  result in repair with a smaller read and download. The second example uses two instances of this code and adds the second level of piggybacking. We note that this design deals with the repair of only the systematic nodes.

*Example 4:* Consider any  $(n = 11, k = 8)$  MDS code as the base code, and take 4 instances of this code. Divide the systematic nodes into two sets as follows,  $S_1 = \{1, 2, 3, 4\}$ ,  $S_2 = \{5, 6, 7, 8\}$ . We then add the piggybacks as shown in Fig. 3. Observe that in this design, the piggybacks added to an even substripe is a function of symbols in its immediately previous (odd) substripe from only the systematic nodes in the first set  $S_1$ , while the piggybacks added to an odd substripe are functions of symbols in its immediately previous (even) substripe from only the systematic nodes in the second set  $S_2$ .

<sup>2</sup>A locality of  $k$  is also possible, but this necessarily mandates the download of the entire data, and hence we do not consider this option.

Node 1	$a_1$	$b_1$	$c_1$	$d_1$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
Node 4	$a_4$	$b_4$	$c_4$	$d_4$
Node 5	$a_5$	$b_5$	$c_5$	$d_5$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
Node 8	$a_8$	$b_8$	$c_8$	$d_8$
Node 9	$\mathbf{p}_1^T \mathbf{a}$	$\mathbf{p}_1^T \mathbf{b}$	$\mathbf{p}_1^T \mathbf{c}$	$\mathbf{p}_1^T \mathbf{d}$
Node 10	$\mathbf{p}_2^T \mathbf{a}$	$\mathbf{p}_2^T \mathbf{b} + a_1 + a_2$	$\mathbf{p}_2^T \mathbf{c} + b_5 + b_6$	$\mathbf{p}_2^T \mathbf{d} + c_1 + c_2$
Node 11	$\mathbf{p}_3^T \mathbf{a}$	$\mathbf{p}_3^T \mathbf{b} + a_3 + a_4$	$\mathbf{p}_3^T \mathbf{c} + b_7 + b_8$	$\mathbf{p}_3^T \mathbf{d} + c_3 + c_4$

Fig. 3: Example illustrating first level of piggybacking in design 3. The piggybacks in the even substripes (in blue) are a function of only the systematic nodes  $\{1, \dots, 4\}$  (also in blue), and the piggybacks in odd substripes (in green) are a function of only the systematic nodes  $\{5, \dots, 8\}$  (also in green). This code requires an average data-read and download of only 71% of the message size for repair of systematic nodes.

We now present the algorithm for repair of any systematic node. First consider the repair of any systematic node  $\ell \in \{1, \dots, 4\}$  in the first set. For instance, say  $\ell = 1$ , then  $\{b_2, \dots, b_8, \mathbf{p}_1^T \mathbf{b}\}$  and  $\{d_2, \dots, d_8, \mathbf{p}_1^T \mathbf{d}\}$  are downloaded, and  $\{\mathbf{b}, \mathbf{d}\}$  (i.e., the messages in the even substripes) are decoded. It now remains to recover the symbols  $a_1$  and  $c_1$  (belonging to the odd substripes). The second symbol  $(\mathbf{p}_2^T \mathbf{b} + a_1 + a_2)$  from node 10 is downloaded and  $\mathbf{p}_2^T \mathbf{b}$  subtracted out to obtain the piggyback  $(a_1 + a_2)$ . Now  $a_1$  can be recovered by downloading and subtracting out  $a_2$ . The fourth symbol from node 10,  $(\mathbf{p}_2^T \mathbf{d} + c_1 + c_2)$ , is also downloaded and  $\mathbf{p}_2^T \mathbf{d}$  subtracted out to obtain the piggyback  $(c_1 + c_2)$ . Finally,  $c_1$  is recovered by downloading and subtracting out  $c_2$ . Thus, node 1 is repaired by reading a total of 20 symbols (in comparison, the total total message size is 32). The repair of node 2 can be carried out in an identical manner. The two other nodes in the first set, nodes 3 and 4, can be repaired in a similar manner by reading the second and fourth symbols of node 11 which have their piggybacks. Thus, repair of any node in the first group requires reading and downloading a total of 20 symbols.

Now we consider the repair of any node  $\ell \in \{5, \dots, 8\}$  in the second set  $S_2$ . For instance, consider  $\ell = 5$ . The symbols  $\{a_1, \dots, a_8, \mathbf{p}_1^T \mathbf{a}\} \setminus \{a_5\}$ ,  $\{c_1, \dots, c_8, \mathbf{p}_1^T \mathbf{c}\} \setminus \{c_5\}$  and  $\{d_1, \dots, d_8, \mathbf{p}_1^T \mathbf{d}\} \setminus \{d_5\}$  are downloaded in order to decode  $a_5, c_5$ , and  $d_5$ . From node 10, the symbol  $(\mathbf{p}_2^T \mathbf{c} + b_5 + b_6)$  is downloaded and  $\mathbf{p}_2^T \mathbf{c}$  is subtracted out. Then,  $b_5$  is recovered by downloading and subtracting out  $b_6$ . Thus, node 5 is recovered by reading a total of 26 symbols. Recovery of other nodes in  $S_2$  follows on similar lines.

The average amount of data read and downloaded during the recovery of systematic nodes is 23, which is 71% of the message size. A higher value of  $m$  (i.e., a higher number of substripes) would lead to a further reduction in the read and download (the last substripe cannot be piggybacked and hence mandates a greater read and download; this is a boundary case, and its contribution to the overall read reduces with an increase in  $m$ ).

*Example 5:* In this example, we illustrate the second level of piggybacking which further reduces the amount of data-read during repair of systematic nodes as compared to Example 4. Consider  $\alpha = 8$  instances of an  $(n = 13, k = 10)$  MDS code. Partition the systematic nodes into three sets  $S_1 = \{1, \dots, 4\}$ ,  $S_2 = \{5, \dots, 8\}$ ,  $S_3 = \{9, 10\}$  (for readers having access to Fig. 4 in color, these nodes are coloured blue, green, and red respectively). We first add piggybacks of the data of the first 8 nodes onto the parity nodes 12 and 13 exactly as

Node 1	$a_1$	$b_1$	$c_1$	$d_1$	$e_1$	$f_1$	$g_1$	$h_1$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
Node 4	$a_4$	$b_4$	$c_4$	$d_4$	$e_4$	$f_4$	$g_4$	$h_4$
Node 5	$a_5$	$b_5$	$c_5$	$d_5$	$e_5$	$f_5$	$g_5$	$h_5$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
Node 8	$a_8$	$b_8$	$c_8$	$d_8$	$e_8$	$f_8$	$g_8$	$h_8$
Node 9	$a_9$	$b_9$	$c_9$	$d_9$	$e_9$	$f_9$	$g_9$	$h_9$
Node 10	$a_{10}$	$b_{10}$	$c_{10}$	$d_{10}$	$e_{10}$	$f_{10}$	$g_{10}$	$h_{10}$
Node 11	$\mathbf{p}_1^T \mathbf{a}$	$\mathbf{p}_1^T \mathbf{b}$	$\mathbf{p}_1^T \mathbf{c}$	$\mathbf{p}_1^T \mathbf{d}$	$\mathbf{p}_1^T \mathbf{e} + a_9 + a_{10}$	$\mathbf{p}_1^T \mathbf{f} + b_9 + b_{10}$	$\mathbf{p}_1^T \mathbf{g} + c_9 + c_{10}$	$\mathbf{p}_1^T \mathbf{h} + g_9 + g_{10}$
Node 12	$\mathbf{p}_2^T \mathbf{a}$	$\mathbf{p}_2^T \mathbf{b} + a_1 + a_2$	$\mathbf{p}_2^T \mathbf{c} + b_5 + b_6$	$\mathbf{p}_2^T \mathbf{d} + c_1 + c_2$	$\mathbf{p}_2^T \mathbf{e}$	$\mathbf{p}_2^T \mathbf{f} + e_1 + e_2$	$\mathbf{p}_2^T \mathbf{g} + f_5 + f_6$	$\mathbf{p}_2^T \mathbf{h} + g_1 + g_2$
Node 13	$\mathbf{p}_3^T \mathbf{a}$	$\mathbf{p}_3^T \mathbf{b} + a_3 + a_4$	$\mathbf{p}_3^T \mathbf{c} + b_7 + b_8$	$\mathbf{p}_3^T \mathbf{d} + c_3 + c_4$	$\mathbf{p}_3^T \mathbf{e}$	$\mathbf{p}_3^T \mathbf{f} + e_3 + e_4$	$\mathbf{p}_3^T \mathbf{g} + f_7 + f_8$	$\mathbf{p}_3^T \mathbf{h} + g_3 + g_4$

Fig. 4: An example illustrating piggyback design 3, with  $k = 10$ ,  $n = 13$ ,  $\alpha = 8$ . The piggybacks in the first parity node (in red) are functions of the data of nodes  $\{8, 9\}$  alone. In the remaining parity nodes, the piggybacks in the even substripes (in blue) are functions of the data of nodes  $\{1, \dots, 4\}$  (also in blue), and the piggybacks in the odd substripes (in green) are functions of the data of nodes  $\{5, \dots, 8\}$  (also in green), and the piggybacks in red (also in red). The piggybacks in nodes 12 and 13 are identical to that in Example 4 (Fig 3). The piggybacks in node 11 piggyback the first set of 4 substripes (white background) onto the second set of number of substripes (gray background)

done in Example 4 (see Fig. 4). We now add piggybacks for the symbols stored in systematic nodes in the third set, i.e., nodes 9 and 10. To this end, we partition the 8 substripes into two groups of size four each (indicated by white and gray shades respectively in Fig. 4). The symbols of nodes 9 and 10 in the first four substripes are piggybacked onto the last four substripes of the first parity node, as shown in Fig. 4 (in red color).

We now present the algorithm for repair of systematic nodes under this piggyback code. The repair algorithm for the systematic nodes  $\{1, \dots, 8\}$  in the first two sets closely follows the repair algorithm illustrated in Example 4. Suppose  $\ell \in S_1$ , say  $\ell = 1$ . By construction, the piggybacks corresponding to the nodes in  $S_1$  are present in the parities of even substripes. From the even substripes, the remaining systematic symbols,  $\{b_i, d_i, f_i, h_i\}_{i=\{2, \dots, 10\}}$ , and the symbols in the first parity,  $\{\mathbf{p}_1^T \mathbf{b}, \mathbf{p}_1^T \mathbf{d}, \mathbf{p}_1^T \mathbf{f} + b_9 + b_{10}, \mathbf{p}_1^T \mathbf{h} + d_9 + d_{10}\}$ , are downloaded. Observe that, the first two parity symbols downloaded do not have any piggybacks. Thus, using the MDS property of the base code,  $\mathbf{b}$  and  $\mathbf{d}$  can be decoded. This also allows us to recover  $\mathbf{p}_1^T \mathbf{f}$ ,  $\mathbf{p}_1^T \mathbf{h}$  from the symbols already downloaded. Again, using the MDS property of the base code, one recovers  $\mathbf{f}$  and  $\mathbf{h}$ . It now remains to recover  $\{a_1, c_1, e_1, g_1\}$ . To this end, we download the symbols in the even substripes of node 12,  $\{\mathbf{p}_2^T \mathbf{b} + a_1 + a_2, \mathbf{p}_2^T \mathbf{d} + c_1 + c_2, \mathbf{p}_2^T \mathbf{f} + e_1 + e_2, \mathbf{p}_2^T \mathbf{h} + g_1 + g_2\}$ , which have piggybacks with the desired symbols. By subtracting out previously downloaded data, we obtain the piggybacks  $\{a_1 + a_2, c_1 + c_2, e_1 + e_2, g_1 + g_2\}$ . Finally, by downloading and subtracting  $a_2, c_2, e_2, g_2$ , we recover  $a_1, c_1, e_1, g_1$ . Thus, node 1 is recovered by reading 48 symbols, which is 60% of the total message size. Observe that the repair of node 1 was accomplished by downloading data from only  $(k+1) = 11$  other nodes. Every node in the first set can be repaired in a similar manner. Repair of the systematic nodes in the second set is performed in a similar fashion by utilizing the corresponding piggybacks, however, the total number of symbols read is 64 (since the last substripe cannot be piggybacked; such was the case in Example 4 as well).

We now present the repair algorithm for systematic nodes  $\{9, 10\}$  in the third set  $S_3$ . Let us suppose  $\ell = 9$ . Observe that the piggybacks corresponding to node 9 fall in the second group (i.e., the last four) of substripes. From the last four substripes, the remaining systematic symbols  $\{e_i, f_i, g_i, h_i\}_{i=\{1, \dots, 8, 10\}}$ , and the symbols in the second parity  $\{\mathbf{p}_1^T \mathbf{e}, \mathbf{p}_1^T \mathbf{f} + e_1 + e_2, \mathbf{p}_1^T \mathbf{g} + f_1 + f_2, \mathbf{p}_1^T \mathbf{h} + g_1 + g_2\}$  are downloaded. Using the MDS property of the base code, one recovers  $\mathbf{e}, \mathbf{f}, \mathbf{g}$  and  $\mathbf{h}$ . It now remains to recover  $a_9, b_9, c_9$  and  $d_9$ . To this end, we download  $\{\mathbf{p}_1^T \mathbf{e} + a_9 + a_{10}, \mathbf{p}_1^T \mathbf{f} + b_9 + b_{10}, \mathbf{p}_1^T \mathbf{g} + c_9 + c_{10}, \mathbf{p}_1^T \mathbf{h} + d_9 + d_{10}\}$  from node 11. Subtracting out the previously downloaded data, we obtain the piggybacks  $\{a_9 + a_{10}, b_9 + b_{10}, c_9 + c_{10}, d_9 + d_{10}\}$ . Finally, by downloading and subtracting out  $\{a_{10}, b_{10}, c_{10}, d_{10}\}$ , we recover the desired data  $\{a_9, b_9, c_9, d_9\}$ . Thus, node 9 is recovered by reading and downloading 48 symbols. Observe that the repair process involved reading data from only  $(k+1) = 11$  other nodes. Node 10 is repaired in a similar manner.

For general values of the parameters,  $n$ ,  $k$ , and  $\alpha = 4m$  for some integer  $m > 1$ , we choose the size of the three sets  $S_1, S_2$ , and  $S_3$ , so as to make the number of systematic nodes involved in each piggyback equal or nearly equal. Denoting the sizes of  $S_1, S_2$  and  $S_3$ , by  $t_1, t_2$ , and  $t_3$  respectively, this gives

$$t_1 = \left\lceil \frac{1}{2r-1} \right\rceil, \quad t_2 = \left\lceil \frac{r-1}{2r-1} \right\rceil, \quad t_3 = \left\lceil \frac{r-1}{2r-1} \right\rceil. \quad (8)$$

Then the average data-read and download  $\gamma_3^{\text{sys}}$  for repair of systematic nodes, as a fraction of the total message symbols  $4mk$ , is

$$\gamma_3^{\text{sys}} = \frac{1}{4mk^2} \left[ t_1 \left( \frac{k}{2} + \frac{t_1}{2} \right) + t_2 \left( \frac{k}{2} + \frac{t_2}{2(r-1)} \right) + t_3 \left( \left( \frac{1}{2} + \frac{1}{m} \right) k + \left( \frac{1}{2} - \frac{1}{m} \right) \frac{t_3}{(r-1)} \right) \right]. \quad (9)$$

This quantity is plotted in Fig. 5a for various values of the system parameters  $n$  and  $k$ .

## VI. COMPARISON OF DIFFERENT CODES

We now compare the average data-read and download entailed during repair under the piggyback constructions with various other storage codes in the literature. As discussed in Section I, practical considerations in data centers

require the storage codes to be MDS, high-rate, and have a small number of substripes. The table below compares different explicit codes designed for efficient repair, with respect to whether they are MDS or not, the parameters they support and the number of substripes. Shaded cells indicate a violation of the aforementioned requirements. The parameter  $m$  associated to the piggyback codes can be chosen to have any value  $m \geq 1$ . The base code for each of the piggyback constructions is a Reed-Solomon code [31].

Code	MDS	$k, r$ supported	Number of substripes
High-rate Regenerating [7], [9]	Y	$r \in \{2, 3\}$	$r^{\frac{k}{r}}$
Product-Matrix MSR [5]	Y	$r \geq k - 1$	$r$
Local Repair [20]–[23]	N	all	1
Rotated RS [8]	Y	$r \in \{2, 3\}, k \leq 36$	2
EVENODD, RDP [25]–[28]	Y	$r = 2$	$k$
Piggyback 1	Y	all	$2m$
Piggyback 2	Y	$r \geq 3$	$(2r - 3)m$
Piggyback 3	Y	all	$4m$

The piggyback, rotated-RS, (repair-optimized) EVENODD and RDP codes satisfy the desired conditions. Fig. 5 shows a plot comparing the repair properties of these codes. The plot corresponds to the number of substripes being 8 in Piggyback 1 and Rotated-RS,  $4(2r - 3)$  in Piggyback 2, and 16 in the Piggyback 3 codes. We observe from the plot that piggyback codes require a lesser (average) data-read and download as compared to Rotated-RS, (repair-optimized) EVENODD and RDP.

## VII. REPAIRING PARITIES IN EXISTING CODES THAT ADDRESS ONLY SYSTEMATIC REPAIR

Several codes proposed in the literature [7], [8], [19], [30] can efficiently repair only the systematic nodes, and require the download of the entire message for repair of any parity node. In this section, we piggyback these codes to reduce the read and download during repair of parity nodes, while also retaining the efficiency of repair of systematic nodes. This piggybacking design is first illustrated with the help of an example.

*Example 6:* Consider the code depicted in Fig. 6a, originally proposed in [19]. This is an MDS code with parameters  $(n = 4, k = 2)$ , and the message comprises four symbols  $a_1, a_2, b_1$  and  $b_2$  over finite field  $\mathbb{F}_5$ . The code can repair any systematic node with an optimal data-read and download. Node 1 is repaired by reading and downloading the symbols  $a_2, (3a_1 + 2b_1 + a_2)$  and  $(3a_1 + 4b_1 + 2a_2)$  from nodes 2, 3 and 4 respectively; node 2 is repaired by reading and downloading the symbols  $b_1, (b_1 + 2a_2 + 3b_2)$  and  $(b_1 + 2a_2 + b_2)$  from nodes 1, 3 and 4 respectively. The amount of data-read and downloaded in these two cases are the minimum possible. However, under this code, the repair of parity nodes with reduced data-read has not been addressed.

In this example, we piggyback the code of Fig. 6a to enable efficient repair of the second parity node. In particular, we take two instances of this code and piggyback it in a manner shown in Fig. 6b. This code is obtained by piggybacking on the first parity symbol of the last two instance, as shown in Fig. 6b. In this piggybacked code, repair of systematic nodes follow the same algorithm as in the base code, i.e., repair of node 1 is accomplished by downloading the first and third symbols of the remaining three nodes, while the repair of node 2 is performed by downloading the second and fourth symbols of the remaining nodes. One can easily verify that the data obtained in each of these two cases is identical to what would have been obtained in the code of Fig. 6a in the absence of piggybacking. Thus the repair of the systematic nodes remains optimal. Now consider repair of the second parity node, i.e., node 4. The code (Fig. 6a), as proposed in [19], would require reading 8 symbols (which is the size of the

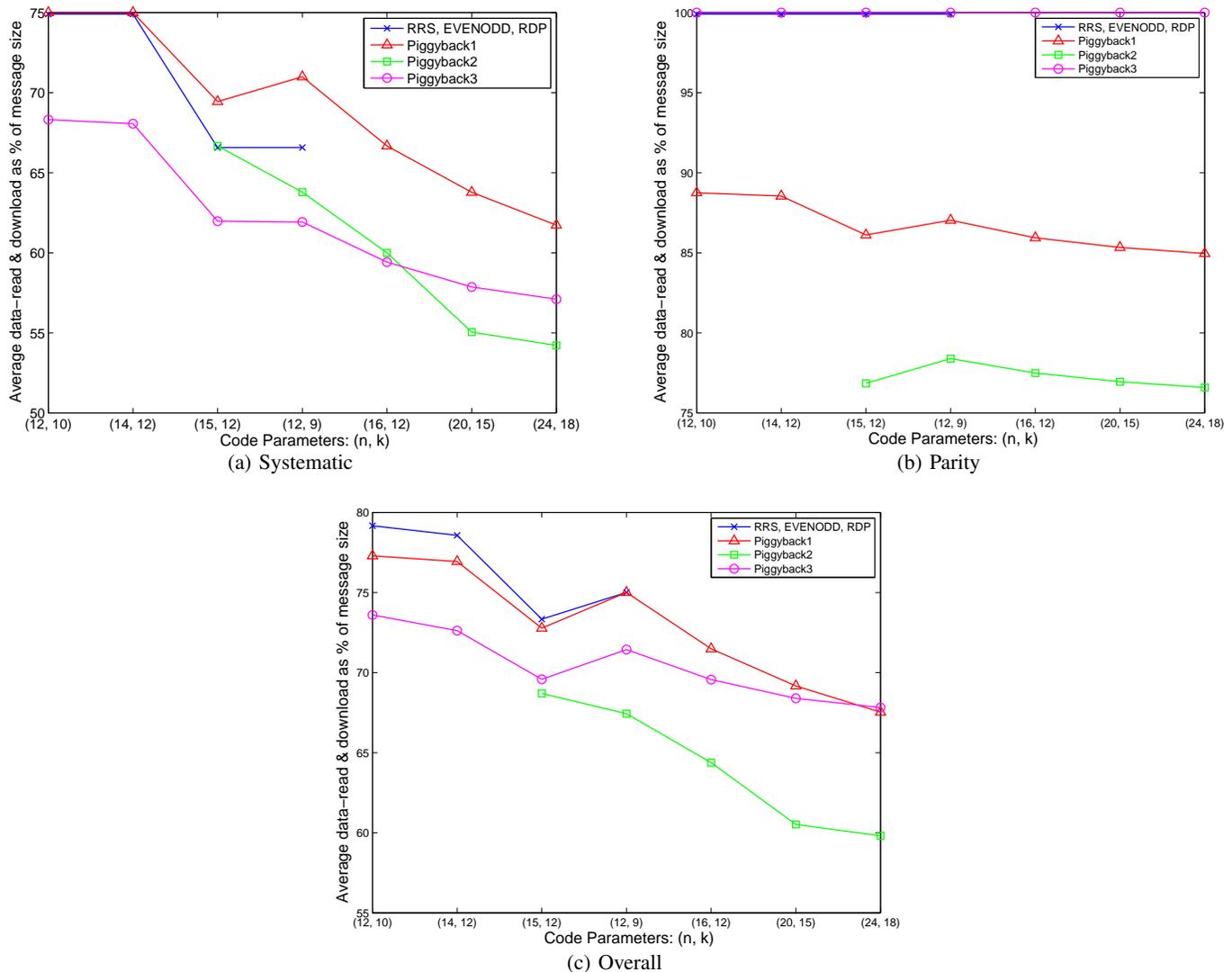


Fig. 5: Average data-read and download for repair of systematic, parity, and all nodes in the three piggybacking designs, Rotated-RS codes [8], and (repair-optimized) EVENODD and RDP codes [25], [26]. The (repair-optimized) EVENODD and RDP codes exist only for  $(n - k) = 2$ , and the data-read and download required for repair are identical to that of a rotated-RS codes with the same parameters. While the savings plotted correspond to a relatively small number of substripes, an increase in this number improves the performance of the piggybacked codes.

Node 1	$a_1$	$b_1$	$a_1$	$b_1$	$c_1$	$d_1$
Node 2	$a_2$	$b_2$	$a_2$	$b_2$	$c_2$	$d_2$
Node 3	$3a_1 + 2b_1 + a_2$	$b_1 + 2a_2 + 3b_2$	$3a_1 + 2b_1 + a_2$	$b_1 + 2a_2 + 3b_2$	$3c_1 + 2d_1 + c_2$ $+ (3a_1 + 4b_1 + 2a_2)$	$d_1 + 2c_2 + 3d_2$ $+ (b_1 + 2a_2 + b_2)$
Node 4	$3a_1 + 4b_1 + 2a_2$	$b_1 + 2a_2 + b_2$	$3a_1 + 4b_1 + 2a_2$	$b_1 + 2a_2 + b_2$	$3c_1 + 4d_1 + 2c_2$	$d_1 + 2c_2 + d_2$

(a) An existing code [19] originally designed to address repair of only systematic nodes

(b) Piggybacking to also optimize repair of parity nodes

Fig. 6: An example illustrating piggybacking to perform efficient repair of the parities in an existing code that originally addressed the repair of only the systematic nodes. See Example 6 for more details.

entire message) for this repair. However, the piggybacked version of Fig. 6b can accomplish this task by reading and downloading only 6 symbols:  $c_1, c_2, d_1, d_2, (3c_1+2d_1+c_2+3a_1+4b_1+2a_2)$  and  $(d_1+2c_2+3d_2+b_1+2a_2+b_2)$ . Here, the first four symbols help in the recovery of the last two symbols of node 4,  $(3c_1+4d_1+2c_2)$  and  $(d_1+2c_2+d_2)$ . Further, from the last two downloaded symbols,  $(3c_1+2d_1+c_2)$  and  $(d_1+2c_2+3d_2)$  can be subtracted out (using the known values of  $c_1, c_2, d_1$  and  $d_2$ ) to obtain the remaining two symbols  $(3a_1+4b_1+2a_2)$  and  $(b_1+2a_2+b_2)$ . Finally, one can easily verify that the MDS property of the code in Fig. 6a carries over to Fig. 6b as discussed in Section II.

We now present a general description of this piggybacking design. We first set up some notation. Let us assume that the base code is a vector code, under which each node stores a vector of length  $\mu$  (a scalar code is, of course, a special case with  $\mu = 1$ ). Let  $\mathbf{a} = [\mathbf{a}_1^T \ \mathbf{a}_2^T \ \dots \ \mathbf{a}_k^T]^T$  be the message, with systematic node  $i \in \{1, \dots, k\}$  storing the  $\mu$  symbols  $\mathbf{a}_i^T$ . Parity node  $(k+j)$ ,  $j \in \{1, \dots, r\}$ , stores the vector  $\mathbf{a}^T P_j$  of  $\mu$  symbols for some  $(k\mu \times \mu)$  matrix  $P_j$ . Fig. 7a illustrates this notation using two instances of such a (vector) code.

We assume that in the base code, the repair of any failed node requires only linear operations at the other nodes. More concretely, for repair of a failed systematic node  $i$ , parity node  $(k+j)$  passes  $\mathbf{a}^T P_j Q_j^{(i)}$  for some matrix  $Q_j^{(i)}$ .

The following lemma serves as a building block for this design.

*Lemma 1:* Consider two instances of any base code, operating on messages  $\mathbf{a}$  and  $\mathbf{b}$  respectively. Suppose there exist two parity nodes  $(k+x)$  and  $(k+y)$ , a  $(\mu \times \mu)$  matrix  $R$ , and another matrix  $S$  such that

$$RQ_x^{(i)} = Q_y^{(i)}S \quad \forall i \in \{1, \dots, k\}. \quad (10)$$

Then, adding  $\mathbf{a}^T P_y R$  as a piggyback to the parity symbol  $\mathbf{b}^T P_x$  of node  $(k+x)$  (i.e., changing it from  $\mathbf{b}^T P_x$  to  $(\mathbf{b}^T P_x + \mathbf{a}^T P_y R)$ ) does not alter the amount of read or download required during repair of any systematic node.

*Proof:* Consider repair of any systematic node  $i \in \{1, \dots, k\}$ . In the piggybacked code, we let each node pass the same linear combinations of its data as it did under the base code. This keeps the amount of read and download identical to the base code. Thus, parity node  $(k+x)$  passes  $\mathbf{a}^T P_x Q_x^{(i)}$  and  $(\mathbf{b}^T P_x + \mathbf{a}^T P_y R)Q_x^{(i)}$ , while parity node  $(k+y)$  passes  $\mathbf{a}^T P_y Q_y^{(i)}$  and  $\mathbf{b}^T P_y Q_y^{(i)}$ . From (10) we see that the data obtained from parity node  $(k+y)$  gives access to  $\mathbf{a}^T P_y Q_y^{(i)}S = \mathbf{a}^T P_y R Q_x^{(i)}$ . This is now subtracted from the data downloaded from node  $(k+x)$  to obtain  $\mathbf{b}^T P_x Q_x^{(i)}$ . At this point, the data obtained is identical to what would have been obtained under the repair algorithm of the base code, which allows the repair to be completed successfully. ■

An example of such a piggybacking is depicted in Fig. 7b.

Under a piggybacking as described in the lemma, the repair of parity node  $(k+y)$  can be made more efficient by exploiting the fact that the parity node  $(k+x)$  now stores the piggybacked symbol  $(\mathbf{b}^T P_x + \mathbf{a}^T P_y R)$ . We now demonstrate the use of this design by making the repair of parity nodes efficient in the explicit MDS ‘regenerating code’ constructions of [7], [9], [19], [30] which address the repair of only the systematic nodes. These codes have the property that

$$Q_x^{(i)} = Q_i \quad \forall i \in \{1, \dots, k\}, \quad \forall x \in \{1, \dots, r\}$$

i.e., the repair of any systematic node involves every parity node passing the same linear combination of its data (and this linear combination depends on the identity of the systematic node being repaired). It follows that in these codes, the condition (10) is satisfied for every pair of parity nodes with  $R$  and  $S$  being identity matrices.

*Example 7:* The piggybacking of (two instances) of any such code [7], [9], [19], [30] is shown in Fig. 7c (for the case  $r = 3$ ). As discussed previously, the MDS property and the property of efficient repair of systematic nodes is retained upon piggybacking. The repair of parity node  $(k+1)$  in this example is carried out by downloading all the  $2k\mu$  symbols. On the other hand, repair of node  $(k+2)$  is accomplished by reading and downloading  $\mathbf{b}$  from the systematic nodes,  $(\mathbf{b}^T P_1 + \mathbf{a}^T P_2 + \mathbf{a}^T P_3)$  from the first parity node, and  $(\mathbf{a}^T P_3)$  from the third parity

Node 1	$\mathbf{a}_1^T$	$\mathbf{b}_1^T$
$\vdots$	$\vdots$	$\vdots$
Node k	$\mathbf{a}_k^T$	$\mathbf{b}_k^T$
Node k+1	$\mathbf{a}^T P_1$	$\mathbf{b}^T P_1$
Node k+2	$\mathbf{a}^T P_2$	$\mathbf{b}^T P_2$
$\vdots$	$\vdots$	$\vdots$
Node k+r	$\mathbf{a}^T P_r$	$\mathbf{b}^T P_r$

(a) Two instances of the vector base code.

	$\mathbf{a}_1^T$	$\mathbf{b}_1^T$
	$\vdots$	$\vdots$
	$\mathbf{a}_k^T$	$\mathbf{b}_k^T$
	$\mathbf{a}^T P_1$	$\mathbf{b}^T P_1 + \mathbf{a}^T P_2 R$
	$\mathbf{a}^T P_2$	$\mathbf{b}^T P_2$
	$\vdots$	$\vdots$
	$\mathbf{a}^T P_r$	$\mathbf{b}^T P_r$

(b) Illustrating the piggybacking stated in Lemma 1. The parities  $(k+1)$  and  $(k+2)$  respectively correspond to  $(k+x)$  and  $(k+y)$  of the Lemma.

Node 1	$\mathbf{a}_1^T$	$\mathbf{b}_1^T$
$\vdots$	$\vdots$	$\vdots$
Node k	$\mathbf{a}_k^T$	$\mathbf{b}_k^T$
Node k+1	$\mathbf{a}^T P_1$	$\mathbf{b}^T P_1 + \mathbf{a}^T P_2 + \mathbf{a}^T P_3$
Node k+2	$\mathbf{a}^T P_2$	$\mathbf{b}^T P_2$
Node k+3	$\mathbf{a}^T P_3$	$\mathbf{b}^T P_3$

(c) Piggybacking the ‘regenerating code’ constructions of [7], [9], [19], [30] for efficient parity repair

Fig. 7: Piggybacking for efficient parity-repair in existing codes originally constructed for repair of only systematic nodes.

node. This gives the two desired symbols  $\mathbf{a}^T P_2$  and  $\mathbf{b}^T P_2$ . Repair of the third parity is performed in an identical manner, except that  $\mathbf{a}^T P_2$  is downloaded from the second parity node. The average amount of download and read for the repair of parity nodes, as a fraction of the size  $k\mu$  of the message, is thus

$$\frac{2k+2}{3k}$$

which translates to a saving of around 33%.

In general, the set of  $r$  parity nodes is partitioned into

$$g = \left\lfloor \frac{r}{\sqrt{k+1}} \right\rfloor$$

sets of equal sizes (or nearly equal sizes if  $r$  is not a multiple of  $g$ ). Within each set, the encoding procedure of Fig. 7c is performed separately. The first parity in each group is repaired by downloading all the data from the systematic nodes. On the other hand, as in Example 7, the repair of any other parity node is performed by reading  $\mathbf{b}$  from the systematic nodes, the second (which is piggybacked) symbol of the first parity node of the set, and the first symbols of all other parity nodes in the set. Assuming the  $g$  sets have equal number of nodes (i.e., ignoring rounding effects), the average amount of read and download for the repair of parity nodes, as a fraction of the size  $k\mu$  of the message, is

$$\frac{1}{2} + \frac{k + \left(\frac{r}{g} - 1\right)^2}{2k \left(\frac{r}{g}\right)}.$$

## VIII. CONCLUSIONS AND OPEN PROBLEMS

We present a new *piggybacking* framework for designing storage codes that require low data-read and download during repair of failed nodes. This framework operates on multiple instances of existing codes and cleverly adds functions of the data from one instance onto the other, in a manner that preserves properties such as minimum distance and the finite field of operation, while enhancing the repair-efficiency. We illustrate the power of this framework by using it to design the most efficient codes (to date) for three important settings. In the paper, we also show how this framework can enhance the efficiency of existing codes that focus on the repair of only systematic nodes, by piggybacking them to also enable efficient repair of parity nodes.

This simple-yet-powerful framework provides a rich design space for construction of storage codes. In this paper, we provide a few designs of piggybacking and specialize it to existing codes to obtain the four specific classes of

code constructions. We believe that this framework has a greater potential, and clever designs of other piggybacking functions and application to other base codes could potentially lead to efficient codes for various other settings as well. Further exploration of this rich design space is left as future work. Finally, while this paper presented only achievable schemes for data-read efficiency during repair, determining the optimal repair-efficiency under these settings remains open.

## REFERENCES

- [1] D. Borthakur, "HDFS and Erasure Codes (HDFS-RAID)," 2009. [Online]. Available: <http://hadoopblog.blogspot.com/2009/08/hdfs-and-erasure-codes-hdfs-raid.html>
- [2] D. Ford, F. Labelle, F. Popovici, M. Stokely, V. Truong, L. Barroso, C. Grimes, and S. Quinlan, "Availability in globally distributed storage systems," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.
- [3] "Erasure codes: the foundation of cloud storage," Sep. 2010. [Online]. Available: <http://blog.cleversafe.com/?p=508>
- [4] K. V. Rashmi, N. B. Shah, P. V. Kumar, and K. Ramchandran, "Explicit construction of optimal exact regenerating codes for distributed storage," in *Proc. 47th Annual Allerton Conference on Communication, Control, and Computing*, Urbana-Champaign, Sep. 2009, pp. 1243–1249.
- [5] K. V. Rashmi, N. B. Shah, and P. V. Kumar, "Optimal exact-regenerating codes for the MSR and MBR points via a product-matrix construction," *IEEE Transactions on Information Theory*, vol. 57, no. 8, pp. 5227–5239, Aug. 2011.
- [6] I. Tamo, Z. Wang, and J. Bruck, "MDS array codes with optimal rebuilding," in *Proc. IEEE International Symposium on Information Theory (ISIT)*, St. Petersburg, Jul. 2011.
- [7] V. Cadambe, C. Huang, and J. Li, "Permutation code: optimal exact-repair of a single failed node in MDS code based distributed storage systems," in *IEEE International Symposium on Information Theory (ISIT)*, 2011, pp. 1225–1229.
- [8] O. Khan, R. Burns, J. Plank, W. Pierce, and C. Huang, "Rethinking erasure codes for cloud file systems: minimizing I/O for recovery and degraded reads," in *Proc. Usenix Conference on File and Storage Technologies (FAST)*, 2012.
- [9] Z. Wang, I. Tamo, and J. Bruck, "On codes for optimal rebuilding access," in *Communication, Control, and Computing (Allerton), 2011 49th Annual Allerton Conference on*, 2011, pp. 1374–1381.
- [10] S. Jiekak, A. Kermarrec, N. Scouarnec, G. Straub, and A. Van Kempen, "Regenerating codes: A system perspective," *arXiv:1204.5028*, 2012.
- [11] A. S. Rawat, O. O. Koyluoglu, N. Silberstein, and S. Vishwanath, "Optimal locally repairable and secure codes for distributed storage systems," *arXiv:1210.6954*, 2012.
- [12] K. Shum and Y. Hu, "Functional-repair-by-transfer regenerating codes," in *IEEE International Symposium on Information Theory (ISIT)*, Cambridge, Jul. 2012, pp. 1192–1196.
- [13] S. El Rouayheb and K. Ramchandran, "Fractional repetition codes for repair in distributed storage systems," in *Allerton Conference on Control, Computing, and Communication*, Urbana-Champaign, Sep. 2010.
- [14] O. Olmez and A. Ramamoorthy, "Repairable replication-based storage systems using resolvable designs," *arXiv:1210.2110*, 2012.
- [15] Y. S. Han, H.-T. Pai, R. Zheng, and P. K. Varshney, "Update-efficient regenerating codes with minimum per-node storage," *arXiv:1301.2497*, 2013.
- [16] B. Gastón, J. Pujol, and M. Villanueva, "Quasi-cyclic regenerating codes," *arXiv:1209.3977*, 2012.
- [17] B. Sasidharan and P. V. Kumar, "High-rate regenerating codes through layering," *arXiv:1301.6157*, 2013.
- [18] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," *IEEE Transactions on Information Theory*, vol. 56, no. 9, pp. 4539–4551, 2010.
- [19] N. B. Shah, K. V. Rashmi, P. V. Kumar, and K. Ramchandran, "Explicit codes minimizing repair bandwidth for distributed storage," in *Proc. IEEE Information Theory Workshop (ITW)*, Cairo, Jan. 2010.
- [20] F. Oggier and A. Datta, "Self-repairing homomorphic codes for distributed storage systems," in *INFOCOM, 2011 Proceedings IEEE*, 2011, pp. 1215–1223.
- [21] P. Gopalan, C. Huang, H. Simitci, and S. Yekhanin, "On the locality of codeword symbols," *IEEE Transactions on Information Theory*, Nov. 2012.
- [22] D. Papailiopoulos and A. Dimakis, "Locally repairable codes," in *Information Theory Proceedings (ISIT), 2012 IEEE International Symposium on*. IEEE, 2012, pp. 2771–2775.
- [23] G. M. Kamath, N. Prakash, V. Lalitha, and P. V. Kumar, "Codes with local regeneration," *arXiv:1211.1932*, 2012.
- [24] N. B. Shah, K. V. Rashmi, P. V. Kumar, and K. Ramchandran, "Distributed storage codes with repair-by-transfer and non-achievability of interior points on the storage-bandwidth tradeoff," *IEEE Transactions on Information Theory*, vol. 58, no. 3, pp. 1837–1852, Mar. 2012.
- [25] Z. Wang, A. G. Dimakis, and J. Bruck, "Rebuilding for array codes in distributed storage systems," in *Workshop on the Application of Communication Theory to Emerging Memory Technologies (ACTEMT)*, Dec. 2010.
- [26] L. Xiang, Y. Xu, J. Lui, and Q. Chang, "Optimal recovery of single disk failure in RDP code storage systems," in *ACM SIGMETRICS*, vol. 38, no. 1, 2010, pp. 119–130.
- [27] M. Blaum, J. Brady, J. Bruck, and J. Menon, "EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures," *IEEE Transactions on Computers*, vol. 44, no. 2, pp. 192–202, 1995.

- [28] P. Corbett, B. English, A. Goel, T. Gracanac, S. Kleiman, J. Leong, and S. Sankar, “Row-diagonal parity for double disk failure correction,” in *Proc. 3rd USENIX Conference on File and Storage Technologies (FAST)*, 2004, pp. 1–14.
- [29] M. Blaum, J. Bruck, and A. Vardy, “MDS array codes with independent parity symbols,” *Information Theory, IEEE Transactions on*, vol. 42, no. 2, pp. 529–542, 1996.
- [30] D. Papailiopoulos, A. Dimakis, and V. Cadambe, “Repair optimal erasure codes through hadamard designs,” in *Proc. 47th Annual Allerton Conference on Communication, Control, and Computing*, Sep. 2011, pp. 1382–1389.
- [31] I. Reed and G. Solomon, “Polynomial codes over certain finite fields,” *Journal of the Society for Industrial & Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

## APPENDIX

*Proof of Theorem 1:* Let us restrict our attention to only the nodes in set  $S$ , and let  $|S|$  denote the size of this set. From the description of the piggybacking framework above, the data stored in instance  $j$  ( $1 \leq j \leq \alpha$ ) under the base code is a function of  $U_j$ . This data can be written as a  $|S|$ -length vector  $\mathbf{f}(U_j)$  with the elements of this vector corresponding to the data stored in the  $|S|$  nodes in set  $S$ . On the other hand, the data stored in instance  $j$  of the piggybacked code is of the form  $(\mathbf{f}(U_j) + \mathbf{g}_j(U_1, \dots, U_{j-1}))$  for some arbitrary (vector-valued) functions ‘ $\mathbf{g}$ ’. Now,

$$I(\{Y_i\}_{i \in S}; U_1, \dots, U_\alpha) = I(\{\mathbf{f}(U_j) + \mathbf{g}_j(U_1, \dots, U_{j-1})\}_{j=1}^\alpha; U_1, \dots, U_\alpha) \quad (11)$$

$$= \sum_{\ell=1}^{\alpha} I(\{\mathbf{f}(U_j) + \mathbf{g}_j(U_1, \dots, U_{j-1})\}_{j=1}^\alpha; U_\ell \mid U_1, \dots, U_{\ell-1}) \quad (12)$$

$$= \sum_{\ell=1}^{\alpha} I(\mathbf{f}(U_\ell), \{\mathbf{f}(U_j) + \mathbf{g}_j(U_1, \dots, U_{j-1})\}_{j=\ell+1}^\alpha; U_\ell \mid U_1, \dots, U_{\ell-1}) \quad (13)$$

$$\geq \sum_{\ell=1}^{\alpha} I(\mathbf{f}(U_\ell); U_\ell \mid U_1, \dots, U_{\ell-1}) \quad (14)$$

$$= \sum_{\ell=1}^{\alpha} I(\mathbf{f}(U_\ell); U_\ell) \quad (15)$$

$$= I(\{X_i\}_{i \in S}; U_1, \dots, U_\alpha) . \quad (16)$$

where the last two equations follow from the fact that the messages  $U_\ell$  of different instances  $\ell$  are independent. ■