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

K. V. Rashmi, Nihar B. Shah, and Kannan Ramchandran, *Fellow, IEEE*

***Abstract*— Erasure codes are being extensively deployed in distributed storage systems instead of replication to achieve fault tolerance in a storage efficient manner. While traditional erasure codes are storage efficient, they can result in a significant increase in the amount of data access and downloaded during rebuilding of failed or otherwise unavailable nodes. In this paper, we present a new framework, which we call *piggybacking*, for constructing distributed storage codes that are efficient in the amount of data read and downloaded during rebuilding, while meeting requirements arising out of system considerations in data centers—maximum-distance-separability (MDS), high-rate, and a small number of so-called substripes. Under this setting, to the best of our knowledge, piggyback codes achieve the minimum average amount of data access and downloaded during rebuilding among all existing explicit solutions. The piggybacking framework also offers a rich design space for constructing codes for a variety of other settings. In particular, we construct codes that require minimum amount of data access and downloaded for rebuilding among all existing solutions for: 1) binary MDS array codes with more than two parities and 2) MDS codes with the smallest locality during rebuilding. In addition, we show how piggybacking can be employed to enable efficient repair of parity nodes in codes that address the rebuilding of only systematic nodes. The basic idea behind the piggybacking framework is to take multiple instances of existing codes and add carefully designed functions of the data from one instance to the others. This framework provides 25% to 50% savings in the average amount of data access and downloaded during rebuilding depending on the choice of the code parameters.**

***Index Terms*— Distributed storage, erasure codes, repair, bandwidth and I/O, code-design framework.**

I. Introduction

LARGE-SCALE distributed storage systems are the building blocks of big-data systems that enable us to store and analyze massive amounts of data. The main goal of such storage systems is to store the data ensuring its reliability and availability in the face of many temporary and permanent failures that occur in their day-to-day operations. In our earlier work [2], we measured failure statistics in Facebook's data warehouse cluster in production. We observed that a median of 50 server-unavailability events occur every day in a cluster comprising multiple thousand servers. Hence, it is imperative that the data be stored in a redundant fashion in order to ensure that the data is reliable and readily available to the applications that wish to consume it.

The traditional industry standard in distributed storage systems, such as Hadoop Distributed File System, Google File System, Windows Azure, etc. [3]–[5], was to employ triple replication, that is, to store three copies of all the data. This redundancy mechanism incurs a storage overhead factor of 3. With the exponential increase in the amount of data being stored, $3\times$ replication has become an expensive proposition. For this reason, distributed storage systems are increasingly turning towards erasure coding as an efficient alternative to replication [5]–[7], with Reed-Solomon (RS) codes [8] being the most popular choice. RS codes are Maximum-Distance-Separable (MDS), and thus make optimal use of storage resources for providing reliability. This makes RS codes attractive for large-scale, distributed storage systems where storage capacity is a critical resource [9]. For instance, Facebook has reported savings of multiple Petabytes of storage space by employing RS codes instead of replication in their data warehouse cluster [10].

An erasure code consists of a total of n units, each of which are functions of the data. In the parlance of information theory, the data that is to be stored in the system is also called the *message*. Our primary focus will be on the setting where k of these units are data units (also called *systematic* units) that store the data in an uncoded form, and $r(= n - k)$ parity units that contain redundant information (or parity functions) associated with the systematic units.[1] We call this set of $n = (k + r)$ units as a *stripe*. In a distributed storage system, a node stores a large number of systematic and parity units belonging to different stripes. Each stripe is conceptually independent and identical, and hence, without loss of generality, we will consider only a single stripe. Given our focus on a single stripe, with a slight abuse of terminology, we will refer to each individual unit as a node. If the erasure code employed is an MDS code, the system will be optimal in terms of storage efficiency. Specifically, in an (n, k) MDS code, each node stores a $(\frac{1}{k})$ fraction of the data, and has the property

Manuscript received March 29, 2016; accepted September 13, 2016. Date of publication June 15, 2017; date of current version August 16, 2017. This work was supported in part by NSF under Grant 1409135, in part by MURI CHASE under Grant 556016, and an Award through Cisco. K. V. Rashmi was supported in part by a Facebook Fellowship and in part by a Microsoft Research Ph.D. Fellowship. N. B. Shah was supported in part by a Microsoft Research Ph.D. Fellowship. This paper was presented in part at the 2013 IEEE International Symposium on Information Theory [1].

The authors are with the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA 94720 USA (e-mail: rashmikv@eecs.berkeley.edu; nihar@eecs.berkeley.edu; kannanr@eecs.berkeley.edu).

Communicated by M. Schwartz, Associate Editor for Coding Techniques.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIT.2017.2715043

[1] We focus on this setting since practical systems today generally prefer to possess a copy of the data in an uncoded form. That said, our results also carry over to settings where each of the n units are some functions of the data.

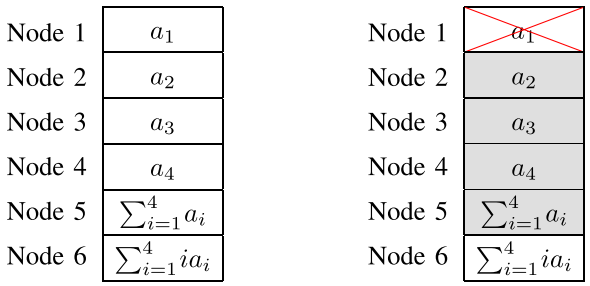

Fig. 1. (a) A stripe of a $(6, 4)$ MDS code. (b) The traditional MDS code repair framework: the first node (storing the finite field element a_1) is repaired by downloading the finite field elements from the $k = 4$ nodes 2, 3, 4 and 5 (highlighted in gray) from which a_1 can be recovered. Here, the amount of data read and downloaded is $k = 4$ times the amount of data being repaired.

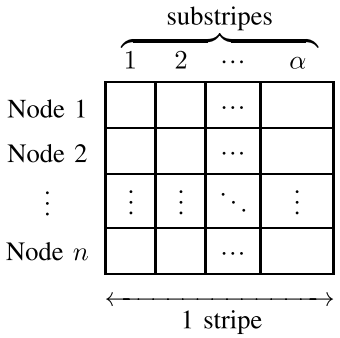

Fig. 2. One stripe of a vector storage code consisting of α substripes. Each cell of the table corresponds to an element from the finite field of operation for the code. When restricting attention to only a single stripe, each row corresponds to a node. Each column corresponds to a substripe. Here the number of substripes is equal to α.

that the entire data can be decoded from any k out of the n nodes. Consequently, such a code can tolerate the failure of *any* $r(= n - k)$ of the n nodes. A single stripe of a $(6, 4)$ MDS codes is depicted in Fig. 1a

A primary contributor to the cost of large-scale storage systems is the storage hardware. Further, several auxiliary costs, such as those of networking equipment, physical space, and cooling, grow proportionally with the amount of storage used. As a consequence, it is critical for any storage code to minimize the storage space consumed. With this motivation of not compromising on storage efficiency, in this paper, we focus on codes that are MDS and have a high rate (that is, a small storage overhead factor).

The frequent temporary and permanent failures that occur in data centers render many parts of the data unavailable from time to time. Recall that we are focusing on a single stripe, and in this case, unavailability pertains to the unavailability of one or more nodes in the stripe. In order to maintain the desired level of reliability and availability, a missing node needs to be replaced by a new node by recreating the data that was stored in it, with the help of the remaining nodes in the stripe. We will call such an operation as a *repair* operation. In large-scale systems, such repair operations are run as background jobs. Repair operations also have a second, foreground application that of *degraded reads*. Large-scale storage systems often receive read requests for data that may be unavailable at that point in time. Such read requests are called degraded reads. Degraded reads are served by repairing the requisite data on the fly, that is, the repair operation is run immediately as a foreground job.

In large-scale distributed systems, unavailability events are the norm rather than the exception [2], [6]. Consequently, when designing storage codes, in addition to storage efficiency, it is important to consider the metric of repair efficiency. The repair efficiency is the efficiency with respect to system resources consumed during repair operations. While traditional framework of MDS codes, such as Reed-Solomon codes, are optimal with respect to storage efficiency, they are inefficient with respect to repair efficiency and put a huge burden on the I/O resources and the network bandwidth of the system [2]. Under the traditional framework of MDS codes, repair of a node is achieved by downloading the data stored in any k of the remaining nodes in the stripe, performing the decoding operation, and retaining only the requisite data corresponding to the failed node. Thus the amount of data read and downloaded during such a repair operation is k times the size of the data being repaired. An example illustrating such a traditional repair operation is depicted in Fig. 1b.

In our earlier work [2], we observed, from measurements on the Facebook data warehouse cluster in production storing multiple tens of Petabytes of RS encoded data, that the scenario of a single node failure in a stripe is by far the most prevalent (more than 98% of all repairs). With this motivation, we focus on optimizing for the case of a single failure in a stripe.

There has been considerable recent work in the area of designing repair-efficient codes for distributed storage systems; these works are discussed in detail in Section II. Of particular interest are the family of so-called *regenerating codes* [11] that aim to optimize the amount of data downloaded during repair operations. These codes are vector (or array) codes, wherein each individual unit is a vector of elements from the finite field over which the code is constructed. Thus each stripe of the code can be viewed as consisting of multiple substripes as depicted in Fig. 2. The number of substripes in a stripe is equal to the size of the vector that constitutes a unit of the code. We are interested in the so called Minimum Storage Regenerating (MSR) subclass of regenerating codes, since this subclass is MDS. The papers [11], [12] showed theoretical existence of MSR codes, and several subsequent works [13]–[18] have presented explicit MSR constructions for a wide range of parameters. Of these, product-matrix codes [13] and MISER codes [14], [15] are explicit MSR code constructions which have a linear number of substripes in k. However, these codes have a low rate: MISER codes require a storage overhead of 2 and product-matrix codes require a storage overhead of $\left(2 - \frac{1}{n}\right)$. The constructions provided in [16]–[18] are high rate, but necessarily require the number of substripes to be exponential in k. In fact, it has been shown in [19], that an exponential number of substripes (more specifically $r^{\frac{k}{r}}$) is a fundamental limitation of any high-rate MSR code optimizing both the amount of data read and downloaded.

The requirement of a large number of substripes presents multiple challenges on the systems front. The primary issue is that a large number of substripes results in a large number of fragmented reads which is detrimental to the performance of disks [20]. An additional, secondary issue is that a large number of substripes also restricts the minimum size of files that can be handled by the code (and hence by the storage system). In addition to this practical motivation, there is also considerable theoretical interest in constructing repair-efficient codes that are MDS, high-rate and have a smaller number of substripes [21]–[24]. To the best of our knowledge, the only explicit codes that meet these requirements are the Rotated-RS [25] codes and the (repair-optimized) EVENODD [26], [27] and RDP [28], [29] codes, all of which are MDS, high-rate and have either a constant or linear (in k) number of substripes. However, Rotated-RS codes exist only for $r \in \{2, 3\}$ and $k \leq 36$, and the (repair-optimized) EVENODD and RDP codes exist only for $r = 2$.

In addition to the metric of the amount of data downloaded for repair, another important metric of interest is the amount of data read from storage devices during repair operations. In general, the amount of data read for a repair operation under any code is necessarily lower bounded by the amount of data downloaded, and furthermore, the amount of data read may be significantly higher than the amount downloaded. This is because, many codes (including regenerating codes), allow each node to read all its data, perform some computations, and transfer only the result. Our interest is in reducing both the amount of data read and downloaded.

To summarize, the two conditions arising from systems and (prior) theory conflict with each other. The systems perspective prefers a small number of substripes [20], whereas any high-rate MDS code that is optimal in terms of data read and download during repair must necessarily have an exponential number of substripes. Consequently, driven by the motivation of practical implementability, this paper focuses on constructing codes with a small number of substripes, and hence are necessarily suboptimal in terms of the amount of data read and downloaded during repair.

In this paper, we investigate the problem of constructing distributed storage codes that are efficient with respect to both the amount of data read and downloaded during repair, while satisfying the constraints of (i) being MDS, (ii) having a high-rate, and (iii) having a small (constant or linear in k) number of substripes. To this end, we present a new framework for constructing distributed storage codes, which we call the *piggybacking* framework. In a nutshell, the piggybacking framework considers multiple instances of an existing code, and a *piggybacking* operation adds (carefully designed) functions of the data from one instance to another.[2] The framework preserves many useful properties of the underlying code such as the minimum distance and the finite field of operation.

The piggybacking framework facilitates construction of codes that are MDS, high-rate, and have a constant (as small as 2) number of substripes, with the smallest average amount of data read and downloaded during repair among all other known codes in this class. An appealing feature of piggyback codes is that they support all values of the code parameters k and $r \geq 2$. [3] The typical savings in the average amount of data read and downloaded during repair is 25% to 50% depending on the choice of the code parameters. In addition to the aforementioned class of codes (which we will term as *Class 1*), the piggybacking framework offers a rich design space for constructing codes for a wide variety of other settings. We illustrate the power of this framework by providing the following three additional classes of explicit code constructions in this paper.

(Class 2) Binary MDS array codes with the lowest average amount of data read and download for repair: Binary MDS array codes [26], [28] are extensively used in disk systems for local storage. Using the piggybacking framework, we construct binary MDS array codes that, to the best of our knowledge, result in the lowest average amount of data read and downloaded for repair among all existing binary MDS array codes for more than two parities (i.e., $r > 2$). Our codes support all the parameters for which binary MDS array codes are known to exist. The codes constructed here reduce the amount of data read and downloaded for repair of the parity nodes as well along with that of systematic nodes.

(Class 3) MDS codes with smallest possible repair locality: Repair locality is the number of nodes that need to be contacted during a repair operation. Several recent works [30]–[36] present codes optimizing for repair locality. However, these codes are not MDS. Given our focus on MDS codes, we use the piggybacking framework to construct MDS codes that have the smallest possible repair locality of $(k + 1)$.[4] To the best of our knowledge, the amount of data read and downloaded during repair in the presented code is the smallest among all known explicit, exact-repair, minimum-locality, MDS codes for more than three parities (i.e., $r > 3$).

(Class 4) A method for reducing the amount of data read and download for repair of parity nodes in existing codes that address only repair of systematic nodes: The problem of efficient node-repair in distributed storage systems has attracted considerable recent attention. However, many of the proposed codes [16], [18], [25], [37], [38] have algorithms for efficient repair of *only* the systematic nodes. We show how the proposed piggybacking framework can be employed to enable efficient repair of parity nodes in such codes, while retaining the efficiency of the repair of systematic nodes.

To be clear, the focus of this paper is on explicit code constructions and not on lower bounds or optimality guarantees. We present the piggybacking framework that enables constructing codes that are efficient with respect to both read and download during repair, while satisfying the constraints of (i) being MDS, (ii) having a high-rate, and (iii) having a small (constant or linear in k) number of substripes. The lower bounds on the amount of data read and downloaded during

[2] Although we focus on MDS codes in this paper, the piggybacking framework can be employed with non-MDS codes as well.

[3] When $r = 1$, it can be easily shown that MDS codes need to read and download all the remaining data during any repair operation.

[4] A locality of k is also possible in MDS codes, but this necessarily mandates the download of the entire data, and hence we do not consider this option.

	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. 3. 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 the amount of data read and downloaded during the repair of any systematic node. A highlighted cell indicates a modified symbol compared to the previous step.

repair under the piggybacking framework, and in general, for distributed storage codes satisfying the above constraints is still open. We refer the reader to a companion paper [20] which builds a practical system making use of the code construction framework presented in this paper. A brief discussion on this system is provided below.

A. Impact on Practice

Since its initial conference publication [1], the Piggybacking framework has attracted considerable interest from the industry. In a companion work [20], we have incorporated one of the classes (Class 1 discussed above) of storage codes constructed using the piggybacking framework on Facebook's distributed file system and this enhanced system is termed *Hitchhiker*. Hitchhiker has been evaluated on Facebook's data warehouse cluster in production. These system evaluation results [20] show significant reduction in network usage, data read and transfer time, and computation time during repair. Piggyback codes have also been incorporated into the open source *Apache Hadoop* [39] as a part of the Hadoop Distributed File System. Apache Hadoop is the most popular big-data storage and analytics platform in the industry today, used by more than a thousand enterprises and powering tens of thousands of clusters.

B. Examples

We now present two examples that highlight the key ideas behind the piggybacking framework. The first example illustrates a method of piggybacking for reducing the amount of data read and downloaded during repair of systematic nodes.

Example 1: Consider two instances of a $(6, 4)$ MDS code as shown in Fig. 3a, with the 8 message symbols $\{a_i\}_{i=1}^{4}$ and $\{b_i\}_{i=1}^{4}$ (each column of Fig. 3a 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. 3b. The second step in this construction involves subtracting the second symbol of node 6 in the code of Fig. 3b from its first symbol. The resulting code is shown in Fig. 3c. This code has 2 substripes (the number of columns in Fig. 3c).

We now present the repair algorithm for the piggybacked code in Fig. 3c. 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 downloaded 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 25% smaller than the total size of the message. Node 2 is 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.

Let us now show that the piggybacked code in Fig. 3c is MDS, and that the entire message can be recovered from any 4 nodes. If node 6 is one of these four nodes, then add its second symbol to its first, to recover the code of Fig. 3b. Otherwise the data obtained from the four nodes in Fig. 3c is identical to that in the code of Fig. 3b. Now in the code of Fig. 3b, the decoding algorithm of the original code of Fig. 3a is employed on the first column to 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 MDS code of Fig. 3a.

The second example below illustrates the use of piggybacking to reduce the amount of data read and downloaded during repair of parity nodes.

Example 2: The code depicted in Fig. 4 takes two instances of the code in Fig. 3c, 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. 4). 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. 3c, the first symbol of node 5 is never used for the repair of any of the systematic nodes. Thus the modification in Fig. 4 does not change the algorithm or the efficiency of the repair of systematic nodes. The code retains

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. 4. An example illustrating the mechanism of repair of parity nodes under the piggybacking framework, using two instances of the code in Fig. 3c. A shaded cell indicates a modified symbol.

the 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. 3c, 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. 3c.

C. Organization

The rest of the paper is organized as follows. A discussion on related literature is provided in Section II. Section III introduces the general piggybacking framework. Sections IV and V present code designs and repair algorithms based on the piggybacking framework, special cases of which result in classes 1 and 2 discussed above. Section VI provides a piggyback design which enables a small repair-locality in MDS codes (Class 3). Section VII provides a comparison of Piggyback codes with various other codes in the literature. Section VIII demonstrates the use of piggybacking to enable efficient parity repair in existing codes that were originally constructed for repair of only the systematic nodes (Class 4). Section IX draws conclusions and discusses open problems.

II. Related Literature

There has been considerable amount of work in the recent past in the area of designing codes for distributed storage systems. Dimakis *et al.* [11] proposed the popular regenerating codes model, which optimizes the amount of data downloaded during repair operations. They provide a network coding based cutset bound for the amount of data download during what is called a *functional* repair. Under functional repair, the repaired node is only functionally equivalent to the failed node. Dimakis *et al.* [11] and Wu [12] showed the theoretical existence of codes meeting the cutset bound for the functional repair setting. In this paper, we consider the more stringent requirement wherein the repaired node is required to be identical to the failed node, also known as *exact* repair in the literature.

The cutset bound presented in [11] leads to a trade-off between the storage space used per node and the bandwidth consumed for repair operations, and this trade-off is called the storage-bandwidth tradeoff. Two important points on the trade-off are its end-points, termed the Minimum-Storage-Regenerating (MSR) and the Minimum-Bandwidth-Regenerating (MBR) points. As discussed in Section I, MSR codes are MDS and hence minimize the amount of storage space consumed; for this minimal storage space, they also minimize the amount of data downloaded during repair. On the other hand, MBR codes achieve the minimum possible download during repair while compromising on the storage space consumption. It has been shown that the MSR and MBR points are achievable even for exact repair [13], [16]–[18], [40], [41]. It has also been shown that the intermediate points are not achievable for exact repair [41], and that there is a non-vanishing gap at these intermediate points between the cutset bound and what is achievable [42].

Several works in the recent past address the problem of constructing explicit MSR and MBR codes [13]–[18], [41]. MSR codes are of particular interest to this paper since they are MDS. The Product-Matrix MSR codes [13] are explicit, practical MSR code constructions which have linear (in k) number of substripes. However, these codes have a low rate (that is, high redundancy), requiring a storage overhead of $\left(2 - \frac{1}{n}\right)$ or higher. Cadambe *et al.* [40] show the existence of high-rate MSR codes as the number of substripes approaches infinity. The MSR constructions presented in [16]–[18] are high rate and have a finite number of substripes. However, the number of substripes in these constructions is exponential in k. In fact, it is shown in [19] that exponential number of substripes (more specifically $r^{\frac{k}{r}}$) is necessary for high-rate MSR codes optimizing both the amount of data read and downloaded. Goparaju *et al.* [22] present a lower bound on the number of substripes for MSR codes which optimize only for data download and do not optimize for data read. The works [21], [23] construct MSR codes with polynomial number of substripes for the setting of constant (high) rate greater than $\frac{2}{3}$. In comparison, the piggybacking framework allows for construction of repair-efficient codes with respect to the amount of data read and downloaded that are MDS, high-rate, and have a constant (as small as 2) number of substripes.

Under the regenerating codes model, each node is allowed to read all its data and perform some computations before transferring the data. Hence the amount of data read during repair can be much higher than in traditional MDS codes. This issue is addressed in [43], where a technique is provided to transform certain classes of MSR codes such that the amount of data read is also optimized along with the amount of data downloaded.

The work [25] presents a search-based approach to find repair symbols that optimize I/O for arbitrary binary erasure codes, but this search problem is shown to be NP-hard. The authors also present a repair-efficient MDS code construction based on RS codes, called Rotated-RS, with the number of substripes as small as 2. However, this construction supports at most 3 parities, and moreover, its fault-tolerance capability is established via a computer search. In contrast, the piggybacking framework is applicable for all parameters, and piggybacked RS codes achieve same or better savings in the amount of data read and download.

Binary MDS array codes have received special attention due to their extensive use in disk array systems [26], [28]. The EVEN-ODD and RDP codes have been optimized for repair in the works [27] and [29] respectively. The work [44] presents a binary MDS array code construction for 2 parities that achieve the regenerating codes bound for repair of systematic nodes. In comparison, the Piggybacking framework based binary MDS array code constructions presented in this paper provide as good or better savings for greater than 2 parities and also address the repair of both systematic and parity nodes.

Another metric of interest in distributed storage systems is that of *repair-locality*, that is, the number nodes contacted during the repair operation. The problem of constructing codes that optimize repair-locality has been extensively studied [30]–[36] in the recent past. While these constructions enjoy a low locality and download during repair, the constructions under this umbrella (must necessarily) relinquish the MDS property in order to get a locality any smaller than k.

Most related works discussed above take the conceptual approach of constructing vector codes in order to improve the efficiency of repair with respect to the amount of data downloaded or both the amount of data read and downloaded. The works [45], [46] present repair algorithms for (scalar) Reed-Solomon codes for reducing the amount of data downloaded during repair by downloading elements from a subfield rather than the finite field over which the code is constructed. While these algorithms optimize the amount of data downloaded, they do not address the amount of data read during repair.

Finally, subsequent to the conference publication of piggybacking framework [1], the Piggybacking design framework has been used in [47] to minimize the amount of download for repair of the parity nodes (meeting the cutset bound) in MSR codes that optimize for repair of only the systematic nodes.

III. The Piggybacking Framework

The piggybacking framework operates on an existing code, which we term the *base code*. Any code can be used as the base code. The base code is associated with n encoding functions, $\{f_i\}_{i=1}^{n}$. For every $i \in \{1, \ldots, n\}$, the encoding function f_i, takes the message $\mathbf{u}$ as input and outputs a coded symbol $f_i(\mathbf{u})$. This symbol $f_i(\mathbf{u})$ is then stored in node i.

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 α instances of the base code. Letting $\mathbf{a}, \ldots, \mathbf{z}$ denote the (independent) messages encoded under these α instances, the encoded symbols in the α instances of the base code can be written as in Fig. 5a.

We shall now describe *piggybacking* of this code. For every $i, 2 \leq i \leq \alpha$, one can add an arbitrary function of the message symbols of all the previous instances $\{1, \ldots, (i-1)\}$ 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 $g_{i,j} (i \in \{2, \ldots, \alpha\}, j \in \{1, \ldots, n\})$, the piggybacked code is shown in Fig.5b.

The decoding properties (such as the minimum distance or the MDS property) 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 would have allowed decoding. To see why this holds, consider any set of nodes from which the message can be recovered under 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 $\{g_{2,i}(\mathbf{a})\}_{i=1}^{n}$ in the second column can now be subtracted out, and what remains in this column is precisely another instance of the base code, allowing recovery of $\mathbf{b}$. Continuing in the same fashion, for any instance i $(2 \leq i \leq \alpha)$, the piggybacks (which are always a function of previously decoded instances $\{1, \ldots, i-1\}$) can be subtracted out to obtain an instance of the base code 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 arbitrary choice of the piggyback functions, and these functions need to be picked judiciously 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. 3b from that in Fig. 3a. Subsequently, in Example 2, this procedure was followed again to obtain the code of Fig. 4 from that in Fig. 3c.

The piggybacking framework also allows for any invertible transformation of the data stored in *any* individual node. In other words, each node of the piggybacked code (e.g., each row in Fig. 3b) can independently undergo *any* invertible transformation. A 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. This is because, the transformation can be inverted as the first step followed by the usual decoding procedure. In Example 1, the code of Fig. 3c is obtained from Fig. 3b 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 $U_1, \ldots, U_\alpha$ be random variables corresponding to the messages associated to the α instances of the base code. For $i \in \{1, \ldots, n\}$, let X_i denote the random variable corresponding to the (encoded) data stored in node i under the base code. Let Y_i denote the random variable variable corresponding to the (encoded) data stored in node i under the piggybacked version of that code. Then for any

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

(a)

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

(b)

Fig. 5. The general piggybacking framework. (a) α instances of the base code. (b) Piggybacked code.

subset of nodes $S \subseteq \{1, \ldots, n\}$,

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

Proof: Please see Appendix. ∎

We also have the following immediate consequences of Theorem 1.

Remark 1: Piggybacking a code does not reduce its minimum distance. Hence, piggybacking an MDS code preserves the MDS property.

Notational Conventions: For simplicity of exposition, in the description of the piggybacking designs in the next four sections, we will assume that the base codes are linear, scalar, MDS and systematic. Using vector (or array) codes (such as EVENODD or RDP codes) as base codes is a straightforward extension. We will use the terms "vector codes" and "array codes" interchangeably.

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 α, and $\{\mathbf{a}, \mathbf{b}, \ldots\}$ shall denote the k-length message vectors corresponding to the α instances. Since the code is systematic, the first k nodes store the elements of the message vector. We use $\mathbf{p}_1, \ldots, \mathbf{p}_r$ to denote the r *encoding vectors* corresponding to the r parities, i.e., if $\mathbf{a}$ denotes the k-length message vector then the r parity nodes under the base code store $\{\mathbf{p}_1^T\mathbf{a}, \ldots, \mathbf{p}_r^T\mathbf{a}\}$. Note that in any MDS base code, every entry of each of the encoding vectors $\mathbf{p}_1, \ldots, \mathbf{p}_r$ must necessarily be non-zero.

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 κ, we denote its κ elements as $\mathbf{v} = [v_1 \ \cdots \ v_\kappa]^T$, and if the vector itself has an associated subscript then we denote its elements as $\mathbf{v}_i = [v_{i,1} \ \cdots \ v_{i,\kappa}]^T$.

Each of the explicit codes constructed in this paper possesses the property that the repair of any node entails reading of only as much data as what has to be downloaded.[5] This property is called *repair-by-transfer* [41]. Thus the amount of data read and the amount of data downloaded are equal under the presented code constructions, and hence we shall use the same notation γ to denote both these quantities.

IV. Piggybacking Design 1

In this section, we present the first design of piggyback functions and the associated repair algorithms, which allows one to reduce the amount of data read and downloaded during repair while having a (small) constant number of substripes. For instance, even when the number of substripes is as small as 2, this piggybacking design can achieve a 25% to 50% saving in the amount of data read and downloaded during the repair of systematic nodes, depending on the values of the parameters k and r. We will first present the piggyback design for optimizing the repair of systematic nodes, and then move on to the repair of parity nodes.

[5]In general, the amount of data downloaded lower bounds the amount of data read, and the amount of data downloaded could be strictly smaller than that the amount of data read if a node passes a (non-injective) function of the data that it stores.

A. General Design

We begin with a design optimizing repair of systematic nodes, following which we then also optimize the repair of parity nodes.

1) 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, $S_1, \ldots, S_r$, 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 to $\frac{k}{r}$ (the sizes of the sets in general scenarios is derived in Section IV-B). Then, let $S_1 = \{1, \ldots, \frac{k}{r}\}$, $S_2 = \{\frac{k}{r} + 1, \ldots, \frac{2k}{r}\}$ and so on, with $S_i = \{\frac{(i-1)k}{r} + 1, \ldots, \frac{ik}{r}\}$ for every $i \in \{1, \ldots, r\}$.

Define the $k-$length vectors $\mathbf{q}_2, \ldots, \mathbf{q}_{r+1}, \mathbf{v}_r$ to be the projections of the encoding vector $\mathbf{p}_r$ as shown at the top of the next page. Note that each element $p_{i,j}$ in the above vectors 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. 3b 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:

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

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 ℓ ($\in \{1, \ldots, k\}$). This entails recovery of the two symbols a_ℓ and b_ℓ from the remaining nodes.

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

Case 1 ($\ell \notin S_r$): Without loss of generality, let $\ell \in S_1$. The k symbols $\{b_1, \ldots, b_{\ell-1}, b_{\ell+1}, \ldots, b_k,\ \mathbf{p}_1^T\mathbf{b}\}$ 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_ℓ. Observe that the ℓ^{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_ℓ from these downloaded symbols. Thus the total amount of data read and downloaded during the repair of node ℓ is $(k + |S_1|)$ (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 $\{b_1, \ldots, b_{\ell-1}, b_{\ell+1}, \ldots, b_k,\ \mathbf{p}_1^T\mathbf{b}\}$. The first symbol $(\mathbf{v}_r^T\mathbf{a} - \mathbf{p}_r^T\mathbf{b})$ of node $(k+r)$ is downloaded. The second symbols $\{\mathbf{p}_i^T\mathbf{b} + \mathbf{q}_i^T\mathbf{a}\}_{i\in\{2,\ldots,r-1\}}$ of the parities $(k+2), \ldots, (k+r-1)$ 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 $\mathbf{b}$, the term $\mathbf{w}^T\mathbf{b}$ is removed to obtain $\mathbf{q}_{r+1}^T\mathbf{a}$. Observe that the ℓ^{th} element of $\mathbf{q}_{r+1}$ is non-zero. The desired symbol a_ℓ can thus be recovered by downloading $\{a_{\frac{k}{r}(r-1)+1}, \ldots, a_{\ell-1}, a_{\ell+1}, \ldots, a_k\}$ from the other systematic nodes in S_r. Thus the total amount of data read and downloaded for recovering node ℓ is $(k + |S_r| + r - 2)$.

2) Repair of Parity Nodes: We now piggyback the code constructed in Section IV-A1 to introduce efficiency in the repair of parity nodes, while retaining the efficiency in the repair of systematic nodes. Observe that in the code of Section IV-A1, 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 IV-A1. 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 even substripe $i \in \{2, 4, \ldots, 2m-2\}$, the $(r-1)$ parity symbols in nodes $(k+2)$ to $(k+r)$ are summed up, and 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 in Fig. 6.
This completes the encoding procedure. The code of Fig. 4 is an example of this design.

As shown in Section III, the piggybacked code retains the MDS property of the base code. In addition, the repair of systematic nodes is dentical to that in the code of Section IV-A1, since the symbol modified in this piggybacking was never read for the repair of any systematic node in the code of Section IV-A1.

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 any other parity node, say node $\ell \in \{k+2, \ldots, k+r\}$. All message symbols $\mathbf{a}, \mathbf{c}, \ldots$ 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 in Fig. 6) are also downloaded from the systematic nodes. Further, the $\{3^{\text{rd}}, 5^{\text{th}}, \ldots, (2m-1)^{\text{th}}\}$ 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 $(\in \{3, 5, \ldots, 2m-1\})$ 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 $\{2, 4, \ldots, 2m-2\}$ 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.

B. Analysis

We begin with an analysis of the repair of systematic nodes. Observe that the repair of systematic nodes in the last set S_r requires larger amount of data to be read and downloaded as compared to the repair of systematic nodes in the other sets. Given this observation, we do not choose the sizes of the sets to be equal (as was considered in Section IV-A1 for simplicity), and instead optimize the sizes to minimize the average read and download required. For $i = 1, \ldots, r$, denoting the size of the set S_i by t_i, the optimal sizes of the sets turn out to be

$$
\begin{aligned}
t_1 = \cdots = t_{r-1} &= \left\lceil \frac{k}{r} + \frac{r-2}{2r} \right\rceil \ := t, \\
t_r &= k - (r-1)t.
\end{aligned}
$$

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 amount of data read and downloaded γ_1^{sys} for repair of systematic nodes, as a fraction of the total number $2k$ of message symbols, is

$$
\gamma_1^{\text{sys}} = \frac{1}{2k^2}\left[(k-t_r)(k+t) + t_r(k+t_r+r-2)\right]. \qquad (2a)
$$

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	$\mathbf{v}_r^T\mathbf{a}-\mathbf{p}_r^T\mathbf{b}$	$\mathbf{p}_r^T\mathbf{b}+\mathbf{q}_r^T\mathbf{a}$	$\mathbf{v}_r^T\mathbf{c}-\mathbf{p}_r^T\mathbf{d}$	$\mathbf{p}_r^T\mathbf{d}+\mathbf{q}_r^T\mathbf{c}$

Fig. 6. General construction of Piggybacking design 1 to include repair of parity nodes.

This quantity is plotted in Fig. 12a for several values of the system parameters n and k.

The average amount of data read and downloaded γ_1^{par} for repair of parity nodes, as a fraction of the total message symbols, is

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

This quantity is plotted in Fig. 12b for several values of the system parameters n and k with $m = 4$.

V. Piggybacking Design 2

In this section we present our second piggybacking design, that is applicable to every value of k and $r \geq 3$. This design provides a higher efficiency of repair as compared to piggybacking design 1 from the previous section. This improved efficiency is at the expense of the number of substripes: the minimum number of substripes required in this design is $(2r-3)$, whereas in comparison the minimum number of substripes required in the designs of Section IV-A1 and Section IV-A1 are 2 and 4 respectively.

A. Example

We begin with an example of 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 $S_1 = \{1, \ldots, 5\}$ and $S_2 = \{6, \ldots, 10\}$. Define 10-length vectors $\mathbf{q}_2$, $\mathbf{v}_2$, $\mathbf{q}_3$ and $\mathbf{v}_3$ as

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

As the first step, piggyback the base code as shown in Fig. 7a.

In the second step, we take invertible transformations of the (respective) data of nodes 12 and 13 as follows. 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 shown in Fig. 7. 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 code as shown in Fig. 7b. This completes the encoding procedure.

We now present an algorithm for efficient repair of any systematic node, say node 1. The 10 symbols $\{c_2, \ldots, c_{10},\ \mathbf{p}_1^T\mathbf{c}\}$ 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 and downloaded.

B. General Design

We now move on to present the general design. Consider $(2r-3)$ instances of the base code, and let $\mathbf{a}_1, \ldots \mathbf{a}_{2r-3}$ 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 set consists 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

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

Further, define k-length vectors $\{\mathbf{q}_{i,j}\}_{i=2,j=1}^{r,r-1}$ where vector q_{ij} is formed from the vector $\mathbf{p}_i$ obtained by setting the entries outside set j to zero, that is,

$$\mathbf{q}_{i,j} = [\underbrace{0 \ldots 0}_{\frac{(j-1)k}{r-1}}\ \underbrace{p_{i,\frac{(j-1)k}{r-1}+1} \cdots p_{i,\frac{jk}{r-1}}}_{\frac{k}{r-1}}\ \underbrace{0 \ldots 0}_{\frac{(r-1-j)k}{r-1}}].$$

It follows that

$$\sum_{j=1}^{r-1} \mathbf{q}_{i,j} = \mathbf{p}_i \qquad \forall\ i \in \{2, \ldots, r\}.$$

Given these preliminaries, we now present the code construction. For every $i \in \{2, \ldots, r\}$, in the first step, parity node

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	$\mathbf{p}_3^T\mathbf{a}$	$\mathbf{p}_3^T\mathbf{b}+\mathbf{q}_3^T\mathbf{a}$	$\mathbf{p}_3^T\mathbf{c}+\mathbf{q}_3^T\mathbf{b}+\mathbf{q}_3^T\mathbf{a}$

(a)

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

(b)

Fig. 7. Example of Piggybacking desgin 2. (a) First step. (b) Second step.

$\mathbf{p}_i^T\mathbf{a}_1$	$\cdots$	$\mathbf{p}_i^T\mathbf{a}_{r-2}$	$\mathbf{p}_i^T\mathbf{a}_{r-1}+(p_i-q_{i,i-1})^T\hat{\mathbf{v}}_i$	$\mathbf{p}_i^T\mathbf{a}_r+\mathbf{q}_{i,1}^T\mathbf{v}_i$	$\cdots$	$\mathbf{p}_i^T\mathbf{a}_{r+i-3}+\mathbf{q}_{i,i-2}^T\mathbf{v}_i$	$\mathbf{p}_i^T\mathbf{a}_{r+i-2}+\mathbf{q}_{i,i}^T\mathbf{v}_i$	$\cdots$	$\mathbf{p}_i^T\mathbf{a}_{2r-3}+\mathbf{q}_{i,r-1}^T\mathbf{v}_i$
$\overbrace{\quad}^{r-2}$			$\overbrace{\quad}^{1}$			$\overbrace{\quad}^{i-2}$	$\overbrace{\quad}^{r-i}$		

(a)

$\mathbf{p}_i^T\mathbf{a}_1$	$\cdots$	$\mathbf{p}_i^T\mathbf{a}_{r-2}$	$\mathbf{q}_{i,i-1}^T\mathbf{a}_{r-1}-\sum_{j=r}^{2r-3}\mathbf{p}_i^T\mathbf{a}_j$	$\mathbf{p}_i^T\mathbf{a}_r+\mathbf{q}_{i,1}^T\mathbf{v}_i$	$\cdots$	$\mathbf{p}_i^T\mathbf{a}_{r+i-3}+\mathbf{q}_{i,i-2}^T\mathbf{v}_i$	$\mathbf{p}_i^T\mathbf{a}_{r+i-2}+\mathbf{q}_{i,i}^T\mathbf{v}_i$	$\cdots$	$\mathbf{p}_i^T\mathbf{a}_{2r-3}+\mathbf{q}_{i,r-1}^T\mathbf{v}_i$

(b)

Fig. 8. The $(2r-3)$ symbols stored in the parity node $(k+i)$ for $i \in \{2,\ldots,r\}$ under Piggybacking design 2. (a) First step. (b) Second step.

$(k+i)$ is piggybacked to store the $(2r-3)$ symbols as shown in Fig. 8a

The second step in the encoding procedure is to take an invertible linear combination within each of the nodes $\{k+2,\ldots,k+r\}$. The transform subtracts the last $(r-2)$ substripes from the $(r-1)^{\text{th}}$ substripe, following which each node $(k+i)$, $i \in \{2,\ldots,r\}$, stores the $(2r-3)$ symbols as shown in Fig. 8b.

Let us now see how repair of a systematic node is performed. Consider repair of node ℓ. First, from nodes $\{1,\ldots,k+1\}\backslash\{\ell\}$, all the data in the last $(r-2)$ substripes is downloaded and the data $\mathbf{a}_r,\ldots,\mathbf{a}_{2r-3}$ is recovered. This also provides us with a part of the desired data $\{a_{r,\ell},\ldots,a_{2r-3,\ell}\}$. Next, observe that in each parity node $\{k+2,\ldots,k+r\}$, there is precisely one symbol whose '$\mathbf{q}$' vector has a non-zero ℓ^{th} component. From each of these nodes, the symbol having this vector is downloaded, and the components along $\{\mathbf{a}_r,\ldots,\mathbf{a}_{2r-3}\}$ are subtracted out. Further, we download all symbols from all other systematic nodes in the same set as node ℓ, and subtract this out from the previously downloaded symbols. This leaves us with $(r-1)$ independent linear combinations of $\{a_{1,\ell},\ldots,a_{r-1,\ell}\}$ from which the desired data is decoded.

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 as in Piggyback design 1 described in Section IV-A2. Note that in Piggyback design 2, the first parities of the first $(r-1)$ substripes are never read for repair of any systematic node. We shall add piggybacks to these unused parity symbols to aid in the repair of other parity nodes. As in Section IV-A2, when a substripe x is piggybacked onto substripe y, the last $(r-1)$ parity symbols in substripe x are summed up and the result is added as a piggyback onto the first parity in substripe y. In this design, the last $(r-2)$ substripes of an odd instance are piggybacked onto the first $(r-2)$ substripes of the subsequent even instance respectively. This leaves the first parity in the $(r-1)^{\text{th}}$ substripe unused in each of the instances. We shall piggyback onto these parity symbols in the following manner: the $(r-1)^{\text{th}}$ substripe in each of the odd instances is piggybacked onto the $(r-1)^{\text{th}}$ substripe of the subsequent even instance. As in Section IV-A2, higher a value of m results in a lesser amount of data read and downloaded for repair.

C. Analysis

We begin our analysis by first deriving the amount of data read and download for repair of systematic nodes. Recall that the k systematic nodes are divided into $(r-1)$, and for a systematic node belonging to a set of size x, the amount of data read and downloaded during repair is given by $(r-2)k+(r-1)x$.

Since the repair algorithm is identical to all the sets, by symmetry, having the $(r-1)$ sets of equal or nearly equal size would minimize the average amount of data read and downloaded. When k is not a multiple of $(r-1)$, we divide the k systematic nodes into $(r-1)$ sets of nearly equal size as follows. Let

$$t_\ell = \left\lfloor \frac{k}{r-1} \right\rfloor, \quad t = (k-(r-1)t_\ell).$$

We first assign t_ℓ systematic nodes to each of the $(r-1)$ sets. This leaves t systematic nodes unassigned. We then assign each of these t remaining systematic nodes to t arbitrarily chosen sets. Without loss of generality, we choose these sets

as the first t sets. Thus, we have the first t sets to be of size $(t_\ell + 1)$ each and the remaining $(r - 1 - t)$ sets to be of size t_ℓ each.

The average data read and download γ_2^{sys} for repair of systematic nodes, as a fraction of the total message symbols $(2r - 3)k$ is as shown in (3a). This quantity is plotted in Fig. 12a for several values of the system parameters n and k.

Let us now consider the repair of the parity nodes. In a manner analogous to the analysis of piggybacking design 1, we obtain that the average data read γ_2^{par} for repair of parity nodes, as a fraction of the total message symbols is as shown in (3b). This quantity is also plotted in Fig. 12b for various values of the system parameters n and k with $m = 4$.

VI. 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 efficient repair of any of the systematic nodes with the smallest possible locality for any MDS code, which is equal to $(k + 1)$. [6] The minimum number of substripes used in this design is 4, and in general, this design can be extended to number of substripes being $2m$ for any positive integer $m \geq 2$. To the best of our knowledge, the amount of data read and downloaded during repair in this code is the smallest among all known explicit, exact-repair, minimum-locality, MDS codes when there are more than three parities.

A. Examples

This design involves two levels of piggybacking, and these are illustrated in the following two example constructions. The first example considers $\alpha = 4$ instances of the base code and shows the first level of piggybacking. 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, $S_1 = \{1, 2, 3, 4\},\ S_2 = \{5, 6, 7, 8\}$. We then add the piggybacks as shown in Fig. 9. 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.

Node 1	a_1	b_1	c_1	d_1
⋮	⋮	⋮	⋮	⋮
Node 4	a_4	b_4	c_4	d_4
Node 5	a_5	b_5	c_5	d_5
⋮	⋮	⋮	⋮	⋮
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. 9. 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 $\{1, \ldots, 4\}$ (also in blue), and the piggybacks in odd substripes (in green) are a function of only the systematic nodes $\{5, \ldots, 8\}$ (also in green). This code requires an average amount of 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 $\ell \in \{1, \ldots, 4\}$ in the first set. For instance, say $\ell = 1$, then $\{b_2, \ldots, b_8,\ \mathbf{p}_1^T\mathbf{b}\}$ and $\{d_2, \ldots, d_8,\ \mathbf{p}_1^T\mathbf{d}\}$ are downloaded, and the messages in the even substripes $\{\mathbf{b},\ \mathbf{d}\}$ 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) and with a locality of 9 $(= k + 1)$. The repair of node 2 is carried out in an identical manner. The two other nodes in the first set, nodes 3 and 4, are 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 set requires reading and downloading a total of 20 symbols and with a locality of 9.

Now we consider the repair of any node $\ell \in \{5, \ldots, 8\}$ in the second set S_2. For instance, consider $\ell = 5$. The symbols $\{a_1, \ldots, a_8,\ \mathbf{p}_1^T\mathbf{a}\} \setminus \{a_5\}$, $\{c_1, \ldots, c_8,\ \mathbf{p}_1^T\mathbf{c}\} \setminus \{c_5\}$ and $\{d_1, \ldots, d_8,\ \mathbf{p}_1^T\mathbf{d}\} \setminus \{d_5\}$ 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 and with a locality

[6] 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.

$$\gamma_2^{\text{sys}} = \frac{1}{(2r-3)k^2}\left[t\,(t_\ell+1)\,((r-2)k+(r-1)(t_\ell+1)) + (r-1-t)t_\ell((r-2)k+(r-1)t_\ell)\right]. \tag{3a}$$

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

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	$\mathbf{p}_1^T\mathbf{a}$	$\mathbf{p}_1^T\mathbf{b}$	$\mathbf{p}_1^T\mathbf{c}$	$\mathbf{p}_1^T\mathbf{d}$	$\mathbf{p}_1^T\mathbf{e}$	$\mathbf{p}_1^T\mathbf{f}+b_9+b_{10}$	$\mathbf{p}_1^T\mathbf{g}+c_9+c_{10}$	$\mathbf{p}_1^T\mathbf{h}+d_9+d_{10}$
Node 12	$\mathbf{p}_2^T\mathbf{a}$	$\mathbf{p}_2^T\mathbf{b}+a_1+a_2$	$\mathbf{p}_2^T\mathbf{c}+b_5+b_6$	$\mathbf{p}_2^T\mathbf{d}+c_1+c_2$	$\mathbf{p}_2^T\mathbf{e}+d_5+d_6$	$\mathbf{p}_2^T\mathbf{f}+e_1+e_2$	$\mathbf{p}_2^T\mathbf{g}+f_5+f_6$	$\mathbf{p}_2^T\mathbf{h}+g_1+g_2$
Node 13	$\mathbf{p}_3^T\mathbf{a}$	$\mathbf{p}_3^T\mathbf{b}+a_3+a_4$	$\mathbf{p}_3^T\mathbf{c}+b_7+b_8$	$\mathbf{p}_3^T\mathbf{d}+c_3+c_4$	$\mathbf{p}_3^T\mathbf{e}+d_7+d_8$	$\mathbf{p}_3^T\mathbf{f}+e_3+e_4$	$\mathbf{p}_3^T\mathbf{g}+f_7+f_8$	$\mathbf{p}_3^T\mathbf{h}+g_3+g_4$

Fig. 10. 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 $\{1, \ldots, 4\}$ (also in blue), and the piggybacks in the odd substripes (in green) are functions of the data of nodes $\{5, \ldots, 8\}$ (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 9). The piggybacks in node 11 piggyback the first set of 4 substripes (white background) onto the second set of substripes (gray background) except the first substripe.

of 9. 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.

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 $S_1 = \{1, \ldots, 4\}$, $S_2 = \{5, \ldots, 8\}$, $S_3 = \{9, 10\}$ (for readers having access to Fig. 10 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 on similar lines as in Example 4 (see Fig. 10). 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 parititon the 8 substripes into two groups of size four each (indicated by white and gray shades respectively in Fig. 10). The symbols of nodes 9 and 10 in the first four substripes are piggybacked onto the last four substripes of the first parity node except the first substripe, as shown in Fig. 10 (in red color).

We now present the algorithm for repair of systematic nodes under this piggyback code. The repair algorithm for the systematic nodes $\{1, \ldots, 8\}$ in the first two sets closely follows the repair algorithm illustrated in Example 4. Consider the repair of any node $\ell \in S_1$; assume without loss of generality that $\ell = 1$. 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, $\{b_i, d_i, f_i, h_i\}_{i=\{2,\ldots,10\}}$, 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 by subtracting out the piggyback added. 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 the symbols a_2, c_2, e_2, g_2 from node 2 and subtracting them out from the data previously downloaded, we recover the symbols 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 with a locality of $(k + 1) = 11$. 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 56 (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 without loss of generality that node $\ell = 9$ is to be repaired. Observe that the piggybacks corresponding to node 9 fall in the last three substripes. From the first and the fifth substripes, the remaining systematic symbols $\{a_i, e_i\}_{i=\{1,\ldots,8,10\}}$, and the symbols in the first parity $\{\mathbf{p}_1^T\mathbf{a}, \mathbf{p}_1^T\mathbf{e}\}$ are downloaded. Using the MDS property of the base code, one recovers $\mathbf{a}$ and $\mathbf{e}$. Then from the last three substripes, the remaining systematic symbols $\{f_i, g_i, h_i\}_{i=\{1,\ldots,8,10\}}$, and the symbols in the second parity $\{\mathbf{p}_2^T\mathbf{f} + e_1 + e_2,\ \mathbf{p}_2^T\mathbf{g} + f_5 + f_6,\ \mathbf{p}_2^T\mathbf{h} + g_1 + g_2,\}$ are downloaded. By subtracting out the already decoded symbols and using the MDS property of the base code, one recovers $\mathbf{f}$, $\mathbf{g}$ and $\mathbf{h}$. It now remains to recover b_9, c_9 and d_9. To this end, we download $\{\mathbf{p}_1^T\mathbf{f} + b_9 + b_{10},\ \mathbf{p}_1^T\mathbf{g} + c_9 + c_{10},\ \mathbf{p}_1^T\mathbf{h} + d_9 + d_{10}\}$ from node 11. Subtracting out the previously decoded data, we obtain the piggybacks $\{b_9 + b_{10},\ c_9 + c_{10},\ d_9 + d_{10}\}$. Finally, by downloading and subtracting out $\{b_{10}, c_{10}, d_{10}\}$, we recover the desired data

$\mathbf{p}_1^T\mathbf{a}$	$\mathbf{p}_1^T\mathbf{b}$	$\mathbf{p}_1^T\mathbf{c}$	$\mathbf{p}_1^T\mathbf{d}$	$\mathbf{p}_1^T\mathbf{e}$	$\mathbf{p}_1^T\mathbf{f}+\sum_{i\in\mathcal{S}_3} b_i$	$\mathbf{p}_1^T\mathbf{g}+\sum_{i\in\mathcal{S}_3} c_i$	$\mathbf{p}_1^T\mathbf{h}+\sum_{i\in\mathcal{S}_3} d_i$

(a)

$\mathbf{p}_j^T\mathbf{a}$	$\mathbf{p}_j^T\mathbf{b}+\sum_{i\in\mathcal{S}_1^{(j-1)}} a_i$	$\mathbf{p}_j^T\mathbf{c}+\sum_{i\in\mathcal{S}_2^{(j-1)}} b_i$	$\mathbf{p}_j^T\mathbf{d}+\sum_{i\in\mathcal{S}_1^{(j-1)}} c_i$	$\mathbf{p}_j^T\mathbf{e}+\sum_{i\in\mathcal{S}_2^{(j-1)}} d_i$	$\mathbf{p}_j^T\mathbf{f}+\sum_{i\in\mathcal{S}_1^{(j-1)}} e_i$	$\mathbf{p}_j^T\mathbf{g}+\sum_{i\in\mathcal{S}_2^{(j-1)}} f_i$	$\mathbf{p}_j^T\mathbf{h}+\sum_{i\in\mathcal{S}_1^{(j-1)}} g_i$

(b)

Fig. 11. Symbols stored in parity nodes under Piggyback design 3. (a) The 8 symbols stored in node (k+1). (b) The 8 symbols stored in node (k+j) for every j ∈ {2,...,r}.

$\{a_9, b_9, c_9, d_9\}$. Thus, node 9 is recovered by reading and downloading 56 symbols. Observe that the locality of repair is $(k+1) = 11$. Node 10 is repaired in a similar manner.

B. Construction for General $(n,\ k)$

As in the example earlier, we partition the set of k systematic nodes into three sets $\mathcal{S}_1$, $\mathcal{S}_2$ and $\mathcal{S}_3$. Let $t_1 = |\mathcal{S}_1|$, $t_2 = |\mathcal{S}_2|$ and $t_3 = |\mathcal{S}_3|$ denote the sizes of sets. The precise values of t_1, t_2 and t_3 are specified in the analysis presented in Section VI-C; given these sizes, the partition can be made in an arbitrary manner. For simplicity of exposition, we will assume that t_1 and t_2 are divisible by $r-1$ (otherwise, one can take suitable floors and ceilings in the expressions to follow).

The construction is a direct generalization of Example 5 presented in the previous section. We first describe the general construction for $\alpha = 8$ substripes of the base code, corresponding to the message (vectors of) symbols $\mathbf{a}, \ldots, \mathbf{h}$ respectively. We later explain extending this construction to $\alpha = 2m$ substripes for any $m \geq 2$ in Section VI-D. First, partition set $\mathcal{S}_1$ into $r-1$ arbitrary subsets of equal sizes, and call these subsets $\mathcal{S}_1^{(1)}, \ldots, \mathcal{S}_1^{(r-1)}$. Likewise partition set $\mathcal{S}_2$ into $r-1$ arbitrary subsets of equal sizes, and call these subsets $\mathcal{S}_2^{(1)}, \ldots, \mathcal{S}_2^{(r-1)}$.

Node $(k+1)$ now stores the 8 symbols shown in Fig. 11a, and for every $j \in \{2, \ldots, r\}$, node $(k+j)$ stores the 8 symbols shown in Fig. 11b. This completes the description of the encoding procedure.

Let us now see how an efficient repair of the systematic nodes can be performed. We begin with the repair of any node in set $\mathcal{S}_1$. Consider any node $\ell \in \mathcal{S}_1$ and assume without loss of generality that $\ell \in \mathcal{S}_1^{j-1}$ for some $j \in \{2, \ldots, r\}$. The repair process entails connecting to the $(k+1)$ nodes $\{1, \ldots, \ell-1, \ell+1, \ldots, k, k+1, k+j\}$, and reading and downloading the following data. First, download the data in the second, fourth, sixth and eighth substripes of nodes $\{1, \ldots, \ell-1, \ell+1, \ldots, k, k+1\}$, from which the data $\mathbf{b}$, $\mathbf{d}$, $\mathbf{f}$ and $\mathbf{h}$ are recovered. This also includes the desired symbols $\{b_\ell, d_\ell, f_\ell, h_\ell\}$. Next, from node $(k+j)$, download the four symbols $\big\{\mathbf{p}_j^T\mathbf{b} + \sum_{i\in\mathcal{S}_1^{(j-1)}} a_i,\ \mathbf{p}_j^T\mathbf{d} + \sum_{i\in\mathcal{S}_1^{(j-1)}} c_i,\ \mathbf{p}_j^T\mathbf{f} + \sum_{i\in\mathcal{S}_1^{(j-1)}} e_i,\ \mathbf{p}_j^T\mathbf{h} + \sum_{i\in\mathcal{S}_1^{(j-1)}} g_i\big\}$. Subtracting out the components associated to $\{\mathbf{b}, \mathbf{d}, \mathbf{f}, \mathbf{h}\}$, then yields $\big\{\sum_{i\in\mathcal{S}_1^{(j-1)}} a_i,\ \sum_{i\in\mathcal{S}_1^{(j-1)}} c_i,\ \sum_{i\in\mathcal{S}_1^{(j-1)}} e_i,\ \sum_{i\in\mathcal{S}_1^{(j-1)}} g_i\big\}$. Finally, downloading the symbols $\{a_i, c_i, e_i, g_i\}_{i\in\mathcal{S}_1^{(j-1)}\setminus\{\ell\}}$ and subtracting them out yields the remaining desired symbols $(a_\ell, c_\ell, e_\ell, g_\ell)$.

We now present the repair procedure for repair of any node in set $\mathcal{S}_2$. Consider any node $\ell \in \mathcal{S}_2$ and assume without loss of generality that $\ell \in \mathcal{S}_2^{j-1}$ for some $j \in \{2, \ldots, r\}$. The repair process entails connecting to the $(k+1)$ nodes $\{1, \ldots, \ell-1, \ell+1, \ldots, k, k+1, k+j\}$, and reading and downloading the following data. First, download the data in the first, third, fifth, seventh and eighth substripes of nodes $\{1, \ldots, \ell-1, \ell+1, \ldots, k, k+1\}$, from which the data $\mathbf{a}$, $\mathbf{c}$, $\mathbf{e}$, $\mathbf{g}$ and $\mathbf{h}$ are recovered. This also includes the desired symbols $\{a_\ell, c_\ell, e_\ell, g_\ell, h_\ell\}$. Next, from node $(k+j)$, download the three symbols $\big\{\mathbf{p}_j^T\mathbf{c} + \sum_{i\in\mathcal{S}_2^{(j-1)}} b_i,\ \mathbf{p}_j^T\mathbf{e} + \sum_{i\in\mathcal{S}_2^{(j-1)}} d_i,\ \mathbf{p}_j^T\mathbf{g} + \sum_{i\in\mathcal{S}_2^{(j-1)}} f_i\big\}$. Subtracting out the components associated to $\{\mathbf{c}, \mathbf{g}\}$, then yields $\big\{\sum_{i\in\mathcal{S}_2^{(j-1)}} b_i,\ \sum_{i\in\mathcal{S}_2^{(j-1)}} d_i,\ \sum_{i\in\mathcal{S}_2^{(j-1)}} f_i\big\}$. Finally, downloading the symbols $\{b_i, d_i, f_i\}_{i\in\mathcal{S}_2^{(j-1)}\setminus\{\ell\}}$ and subtracting them out yields the remaining desired symbols (b_ℓ, d_ℓ, f_ℓ).

Finally, the repair of any node $\ell \in \mathcal{S}_3$ is executed by connecting to the $(k+1)$ nodes $\{1, \ldots, \ell-1, \ell+1, \ldots, k, k+1, k+2\}$. The first and fifth substripes of each of the nodes $\{1, \ldots, \ell-1, \ell+1, \ldots, k, k+1\}$ is downloaded, which yields the data $\mathbf{a}$ and $\mathbf{e}$. Next, the sixth, seventh and eighth substripes of nodes $\{1, \ldots, \ell-1, \ell+1, \ldots, k, k+2\}$ are downloaded, and subtracting out the components along $\mathbf{e}$ from this data yields $\{\mathbf{f}, \mathbf{g}, \mathbf{h}\}$. This data also includes the desired symbols $\{a_\ell, e_\ell, f_\ell, g_\ell, h_\ell\}$. Now, the three symbols $\big\{\mathbf{p}_j^T\mathbf{f} + \sum_{i\in\mathcal{S}_3} e_i,\ \mathbf{p}_j^T\mathbf{g} + \sum_{i\in\mathcal{S}_3} f_i,\ \mathbf{p}_j^T\mathbf{h} + \sum_{i\in\mathcal{S}_3} g_i\big\}$ are downloaded from node $(k+2)$, and canceling out the terms associated to $\mathbf{f}$, $\mathbf{g}$ and $\mathbf{h}$ yields $\big\{\sum_{i\in\mathcal{S}_3} e_i,\ \sum_{i\in\mathcal{S}_3} f_i,\ \sum_{i\in\mathcal{S}_3} g_i\big\}$. Finally, downloading the symbols $\{b_i, c_i, d_i\}_{i\in\mathcal{S}_3\setminus\{\ell\}}$ and subtracting them out yields the remaining desired symbols (b_ℓ, c_ℓ, d_ℓ).

C. Analysis

We continue to assume that t_1 and t_2 are both divisible by $(r-1)$. Also, we set $m = 4$, that is, $\alpha = 8$. Then one can verify that the repair of any node in set $\mathcal{S}_1$ entails a read and download of $4k + \frac{4t_1}{r-1}$ symbols, the repair of any node in set $\mathcal{S}_2$ requires a read and download of $5k + \frac{3t_2}{r-1}$ symbols, and the repair of any node in set $\mathcal{S}_3$ requires a read and download of $5k + 3t_3$ symbols. The average read and download required to repair a systematic node, in comparison to the total message

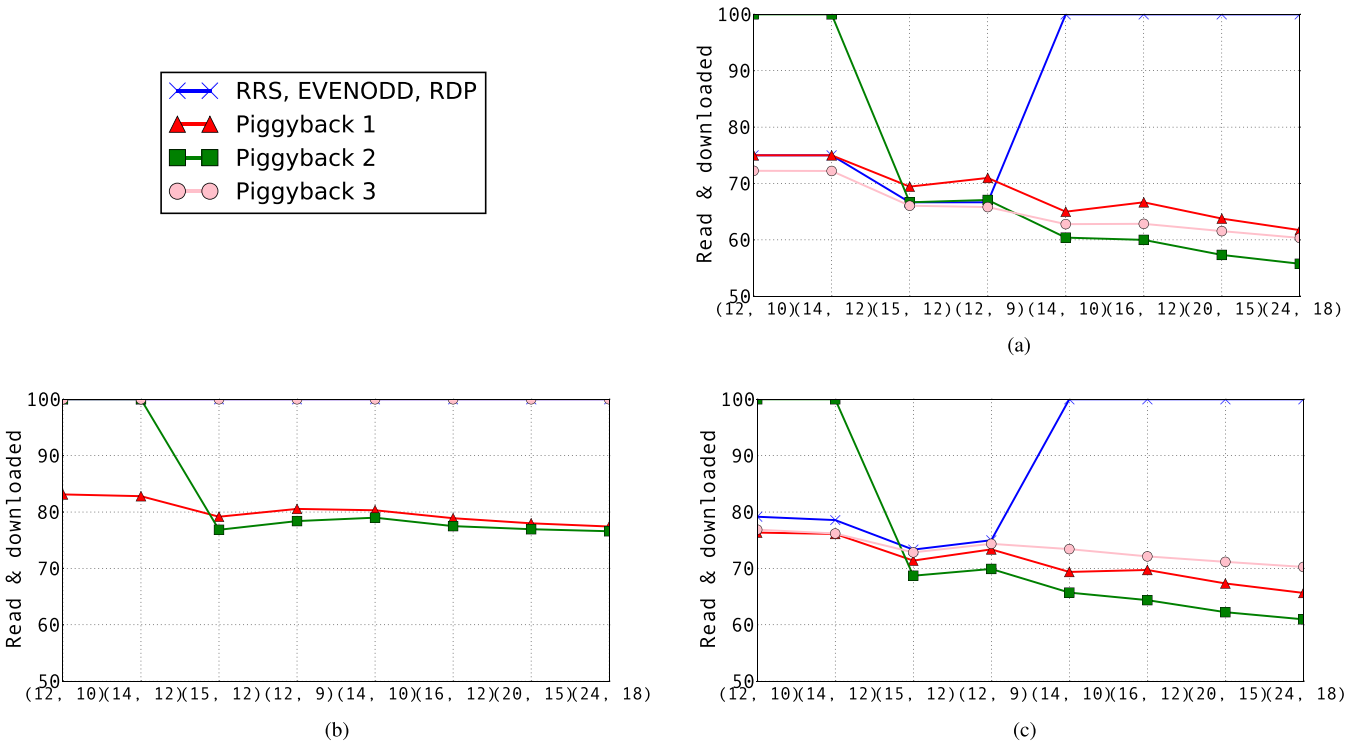

Fig. 12. Average amount of data read and downloaded for repair of systematic nodes, parity nodes, and all nodes in the three piggybacking designs, Rotated-RS codes [25], and (repair-optimized) EVENODD and RDP codes [27], [29]. The (repair-optimized) EVENODD and RDP codes exist only for $r = 2$, and the amount of data read and downloaded for repair is identical to that of a Rotated-RS code with the same parameters. (a) Systematic. (b) Parity. (c) Overall.

size $8k$ equals:

$$\gamma_3^{\text{sys}} = \frac{1}{8k}\left[t_1\left(4k+\frac{4t_1}{r-1}\right) + t_2\left(5k+\frac{3t_2}{r-1}\right) + t_3\left(5k+3t_3\right)\right]. \tag{4}$$

One may further optimize this expression over the choices of t_1, t_2 and t_3. A choice that is consistent with the construction in Example 5 is

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

The average read and download for repair of systematic nodes under this construction, γ_3^{sys}, is plotted in Fig. 12a for several values of the system parameters n and k. The parity nodes here are repaired by reading and downloading all the message symbols, and hence we have $\gamma_3^{\text{par}} = 1$.

D. Extending the number of substripes

As a final remark, we note that Piggyback desgin 3 described above can be extended to any value of $\alpha = 2m$ with $m \geq 2$ in a straightforward manner. First, the nodes in sets $\mathcal{S}_1$ and $\mathcal{S}_2$ are piggybacked onto the parity nodes $\{(k+2), \ldots, (k+r)\}$ in a manner identical to the above description in Section VI-B. That is, for nodes in set $\mathcal{S}_1$, symbols from odd substripes are piggybacked onto the immediately following even substripes, and for nodes in set $\mathcal{S}_2$, symbols from even substripes are piggybacked onto the immediately following odd substripes. Next, for the nodes in set $\mathcal{S}_3$, symbols in the first m substripes are piggybacked onto the last m substripes of the first parity node except for the symbols in the first substripe which are not piggybacked. Given this piggybacking, the repair of nodes in set $\mathcal{S}_2$ becomes more efficient for higher values of m. This is due to the fact that for any value of m, the last substripe is not piggybacked and hence mandates a greater read and download; this is a boundary case, and its contribution to the overall read and download reduces with an increase in m.

VII. Comparative Study

We now compare the average amount of data read and downloaded during repair under the piggybacking constructions with existing explicit constructions in prior literature. We first consider Class 1, which corresponds to codes that are MDS, have high-rate, and have a small (constant or linear in k) number of substripes (recall from Section I). A summary of the supported parameters of various existing explicit constructions for Class 1 and the piggybacking constructions is provided in Table I.

Fig. 12 presents plots comparing the amount of data read and downloaded during repair under these codes. Each of the codes compared in Fig. 12 have the property of *repair-by-transfer* [41], i.e., they read only those symbols that they wish to download. [7] Consequently, the amount of data read and downloaded are equal under these codes. The average amount of data read and downloaded during repair of systematic nodes is compared in Fig. 12a, that during repair of parity nodes in Fig. 12b, and the average taking all nodes into account is compared in Fig. 12c. Although the goal of

[7] In prior literature, this property has also been called by the term *help-by-transfer*.

TABLE I

SUMMARY OF SUPPORTED PARAMETERS FOR EXPLICIT CODE CONSTRUCTIONS UNDER CLASS 1, WHICH ARE CODES THAT ARE MDS, ALLOW HIGH-RATE, AND HAVE A SMALL (CONSTANT OR LINEAR IN k) NUMBER OF SUBSTRIPES

Code	k, r supported	Number of substripes
Rotated RS [25]	$r \in \{2, 3\}, k \leq 36$	2
EVENODD, RDP [26]–[29]	$r = 2$	$k-1, k$
Piggyback 1	$r \geq 2$	$2m$, $m \geq 1$
Piggyback 2	$r \geq 3$	$(2r-3)m$, $m \geq 1$
Piggyback 3	$r \geq 2$	$2m$, $m \geq 2$

Node 1	a_1	b_1
Node 2	a_2	b_2
Node 3	$3a_1+2b_1+a_2$	$b_1+2a_2+3b_2$
Node 4	$3a_1+4b_1+2a_2$	$b_1+2a_2+b_2$

(a)

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

(b)

Fig. 13. 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. (a) An existing code [37] originally designed to address repair of only systematic nodes. (b) Piggybacking to make the repair of parity nodes more efficient.

Piggyback design 3 is to minimize the repair locality, it also reduces the amount of data read and downloaded during repair and hence we include it in these comparison plots. In the plots, we consider the following number of substripes: 8 for Piggyback 1 and Rotated-RS codes (abbreviated as RRS in Fig. 12), $4(2r-3)$ for Piggyback 2, and 8 for Piggyback 3. The number of substripes for EVENODD and RDP codes are $(k-1)$ and k respectively. Note that the (repair-optimized) EVENODD and RDP codes exist only for $r = 2$, and Rotated-RS codes exist only for $r = \{2, 3\}$. The amount of data read and downloaded for repair under the (repair-optimized) EVENODD and RDP codes is identical to that of a Rotated-RS code with the same parameters.

To address Class 2, which corresponds to binary MDS array codes, we propose using Piggybacking designs 1 and 2 with traditional binary MDS array codes as the underlying base codes (such as [48]). This provides repair-efficient, binary, MDS codes for all parameters where traditional binary MDS array codes exist. The (repair-optimized) EVENODD and RDP codes [27], [29] and the code presented in [44] are binary MDS array codes and each of these codes exists only for $r = 2$. Furthermore, (repair-optimized) EVENODD and the construction in [44] consider repair of only the systematic nodes. We have already seen the comparison between the savings from the piggyback constructions and the (repair-optimized) EVENODD and RDP codes in Fig. 12. For $r = 2$, considering only the repair of systematic nodes, the construction in [44] achieves the minimum amount of data read and download which is 50% of the message size while requiring an exponential in k (more specifically, 2^{k-1}) number of substripes.

Class 3, which corresponds to repair-efficient MDS codes with smallest possible repair locality of $(k+1)$, was addressed by the Piggybacking design 3 in Section VI. Class 4, which corresponds to optimizing repair of parity nodes in codes that optimize repair of only the systematic nodes is presented in the following section.

VIII. REPAIRING PARITIES IN EXISTING CODES THAT ADDRESS ONLY SYSTEMATIC REPAIR

Several codes proposed in the literature [16], [18], [25], [37] 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 show how to piggyback these codes to reduce the read and download during repair of parity nodes, while retaining the efficiency of repair of systematic nodes.

A. Example

We begin with an illustrative example.

Example 6: Consider the code depicted in Fig. 13a, originally proposed in [37]. 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 amount of 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. 13a 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. 13b. This code is obtained by piggybacking on the first parity symbols of the second instance, as shown in Fig. 13b. 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 three nodes. One can easily verify that the data obtained in each of these two cases are equivalent to what would have been obtained in the code of Fig. 13a in the absence of piggybacking by first repairing the first instance and then subtracting out the piggybacks. Thus the repair of the systematic nodes remains optimal. Now consider repair of the second parity node, i.e., node 4. The code of Fig. 13a, as originally proposed in [37], would require reading 8 symbols (which is the size of the entire message) for this repair. However, the piggybacked version in Fig. 13b 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. 13a carries over to Fig. 13b in a manner discussed in Section III.

B. General Design

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 μ (a scalar code is, of course, a special case with $\mu = 1$). Let $\mathbf{a} = [\mathbf{a}_1^T \ \mathbf{a}_2^T \ \cdots \ \mathbf{a}_k^T]^T$ be the message, with systematic node $i \in \{1, \ldots, k\}$ storing the μ symbols $\mathbf{a}_i^T$. Parity node $(k + j)$, $j \in \{1, \ldots, r\}$, stores the vector $\mathbf{a}^T P_j$ of μ symbols, where P_j is some $(k\mu \times \mu)$ matrix determined by the base code. Fig. 14a 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 proposition serves as a building block for this design.

Proposition 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

$$R Q_x^{(i)} = Q_y^{(i)} S \quad \forall\ i \in \{1, \ldots, k\}. \tag{5}$$

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 $i \in \{1, \ldots, k\}$. 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 data read and downloaded 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 (5) 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. 14b.

Under a piggybacking as described in the above proposition, 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 [16], [18], [37], [38] which address the repair of only the systematic nodes. These codes have the property that

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

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 (5) is satisfied for every pair of parity nodes with R and S being identity matrices.

The r parity nodes are first partitioned (in an arbitrary manner) into t sets, $\{\mathcal{S}_1, \ldots, \mathcal{S}_t\}$, each of which contains at least two nodes. The value of t and the sizes of these sets are chosen subsequently in Section VIII-C. The following procedure is applied to each set $\mathcal{S} \in \{\mathcal{S}_1, \ldots, \mathcal{S}_t\}$. Choose (arbitrarily) any one node $s \in \mathcal{S}$ in this set. Every node $j \in \mathcal{S}\backslash\{s\}$ stores data as in the base code, that is, stores the data $\mathbf{a}^T P_j$ and $\mathbf{b}^T P_j$. On the other hand, the node $s \in \mathcal{S}$ stores the data $\mathbf{a}^T P_s$ and $(\mathbf{b}^T P_s + \sum_{j \in \mathcal{S}\backslash\{s\}} \mathbf{a}^T P_j)$. This completes the description of the encoding process. The code for the example of $r = 3$ and $t = 1$ is depicted in Fig. 14c; in this example, we have $\mathcal{S} = \{1, 2, 3\}$ and $s = 1$.

We now describe the repair process. Proposition 1 guarantees that the repair of systematic nodes occur with the same efficiency as in the underlying base code. Now, to see the improved repair of parity nodes, consider any set $\mathcal{S} \in \{\mathcal{S}_1, \ldots, \mathcal{S}_t\}$. Let $s \in \mathcal{S}$ denote the node that was chosen in this set during the encoding process. This node s is repaired by downloading all the data from the systematic nodes. Now consider the repair of any other node $j \in \mathcal{S}\backslash\{s\}$. This repair is performed by reading and downloading the symbols $\mathbf{b}_1, \ldots \mathbf{b}_k$ from the systematic nodes, the symbol $(\mathbf{b}^T P_s + \sum_{j \in \mathcal{S}\backslash\{s\}} \mathbf{a}^T P_j)$ from node s, and the symbols $\{\mathbf{a}^T P_\ell\}_{\ell \in \mathcal{S}\backslash\{j,s\}}$ from the remaining nodes in set $\mathcal{S}$. The symbols $\mathbf{b}_1, \ldots \mathbf{b}_k$ are used to obtain half of the desired data $\mathbf{b}^T P_j$. These symbols are also used to remove the component $\mathbf{b}^T P_s$ from $(\mathbf{b}^T P_s + \sum_{j \in \mathcal{S}\backslash\{s\}} \mathbf{a}^T P_j)$ to obtain $(\sum_{j \in \mathcal{S}\backslash\{s\}} \mathbf{a}^T P_j)$. Finally,

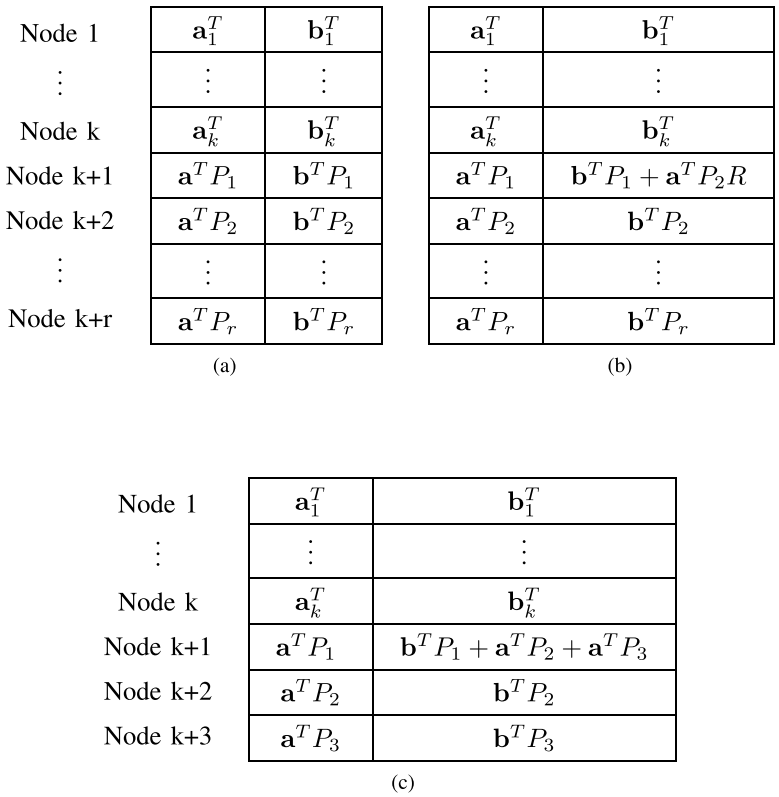

Fig. 14. Piggybacking for efficient repair of parity nodes in existing codes originally constructed for repair of only systematic nodes. (a) Two instances of the base (vector) code. (b) Illustrating the piggybacking design stated in Proposition 1. The parities $(k+1)$ and $(k+2)$ respectively correspond to $(k+x)$ and $(k+y)$ of the Proposition. (c) Piggybacking the regenerating code constructions of [18], [37], [38], [49] for efficient parity repair.

the symbols $\{\mathbf{a}^T P_\ell\}_{\ell \in \mathcal{S}\setminus\{j,s\}}$ are subtracted out from this data, and what remains is precisely the other half $\mathbf{a}^T P_j$ of the desired data.

For the example code of Fig. 14c with $r = 3$ and $t = 1$, one can calculate that the repair of nodes $(k+2)$ and $(k+3)$ entail a read and download of $(k+2)$ symbols each, as opposed to reading and downloading the entire message of $2k$ symbols.

C. Analysis

We now compute the total amount of data read and downloaded during repair of any parity node. Observe that the total amount of data read and downloaded for repair of node s is exactly $2\ k\mu$. On the other hand, every node $j \in \mathcal{S}\setminus\{s\}$ is repaired by reading and downloading $(k+|\mathcal{S}|-1)\mu$ symbols. The average amount of data read and downloaded for repair of any node in set $\mathcal{S}$ then equals

$$\frac{2k + (|\mathcal{S}|-1)(k+|\mathcal{S}|-1)}{|\mathcal{S}|}\mu.$$

Finally, it remains to specify the choice of the number of sets t and the sizes of each set. To this end, one can show that the value of $|\mathcal{S}|$ that minimizes the expression above is $|\mathcal{S}| = \sqrt{k+1}$. We must thus choose the sets so that the size of each set is as close to $\sqrt{k+1}$ as possible.

Assuming for simplicity of exposition that $\sqrt{k+1}$ is an integer, and that r is a multiple of $\sqrt{k+1}$, we choose the number of sets $t = \frac{r}{\sqrt{k+1}}$ and the size of each set as $\sqrt{k+1}$. This choice results in an average data read and download for the repair of parity nodes, as a fraction of the size $2k\mu$ of the message, as

$$\frac{2\sqrt{k+1}+k-2}{2k}.$$

IX. Conclusions and Open Problems

We present a new framework, which we call the *piggybacking* framework, for designing storage codes that are efficient in both the amount of data read and downloaded during repair, while being MDS, high-rate, and having a small number of substripes. Under this setting, to the best of our knowledge, piggyback codes achieve the minimum amount of data read and downloaded for repair among all existing explicit solutions. The piggybacking framework operates on multiple instances of existing codes and adds carefully designed 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. We illustrate the power of this framework by constructing classes of explicit codes that entail, to the best of our knowledge, the smallest amount of data read and downloaded for repair among all existing solutions for two important settings in addition to the one mentioned above. Furthermore, we show how the piggybacking framework can be employed to enable efficient repair of parity nodes in existing codes that were originally constructed to address the repair of only the systematic nodes.

Piggyback codes have also received considerable attention from practitioners: In a companion work [20], we present a

system *Hitchhiker* where we have incorporated one of the classes (Class 1) of storage codes constructed using the piggybacking framework on Facebook's distributed file system. Hitchhiker has been evaluated on Facebook's data warehouse cluster in production showing significant benefits [20]. Piggyback codes have also been incorporated into the open source Apache Hadoop [39] which is the most widely used big-data storage and analytics platform in the industry.

The simple-yet-powerful piggybacking 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. A further exploration of this rich design space is left as future work. Finally, while this paper presented achievable schemes for data read and download efficiency during repair, determining the optimal amount of data read and download required under this framework, and in general, for distributed storage codes under the constraints discussed above still remains open.

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 $\left(\mathbf{f}(U_j) + \mathbf{g}_j(U_1, \ldots, U_{j-1})\right)$ for some arbitrary (vector-valued) functions $\mathbf{g}_j$. Now,

$$
\begin{aligned}
&I\left(\{Y_i\}_{i\in S};\ U_1, \ldots, U_\alpha\right) \\
&= I\left(\left\{\mathbf{f}(U_j) + \mathbf{g}_j(U_1, \ldots, U_{j-1})\right\}_{j=1}^{\alpha}; U_1, \ldots, U_\alpha\right) \\
&= \sum_{\ell=1}^{\alpha} I\left(\left\{\mathbf{f}(U_j) + \mathbf{g}_j(U_1, \ldots, U_{j-1})\right\}_{j=1}^{\alpha};\ U_\ell \right. \\
&\qquad\qquad \left. \Big|\ U_1, \ldots, U_{\ell-1}\right) \\
&= \sum_{\ell=1}^{\alpha} I\left(\mathbf{f}(U_\ell)\ ,\ \left\{\mathbf{f}(U_j) + \mathbf{g}_j(U_1, \ldots, U_{j-1})\right\}_{j=\ell+1}^{\alpha}; U_\ell \right. \\
&\qquad\qquad \left. \Big|\ U_1, \ldots, U_{\ell-1}\right) \\
&\geq \sum_{\ell=1}^{\alpha} I\left(\mathbf{f}(U_\ell);\ U_\ell \mid U_1, \ldots, U_{\ell-1}\right) \\
&= \sum_{\ell=1}^{\alpha} I\left(\mathbf{f}(U_\ell);\ U_\ell\right) \\
&= I\left(\{X_i\}_{i\in S}; U_1, \ldots, U_\alpha\right),
\end{aligned}
$$

where the last two equations follow from the fact that the messages U_ℓ of different instances ℓ are independent. ∎

REFERENCES

[1] K. V. Rashmi, N. B. Shah, and K. Ramchandran, "A piggybacking design framework for read-and download-efficient distributed storage codes," in *Proc. IEEE Int. Symp. Inf. Theory*, Jul. 2013, pp. 331–335.

[2] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster," in *Proc. USENIX HotStorage*, Jun. 2013.

[3] S. Ghemawat, H. Gobioff, and S. Leung, "The Google file system," *ACM SIGOPS Operating Syst. Rev.*, vol. 37, no. 5, pp. 29–43, 2003.

[4] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proc. IEEE Symp. Mass Storage Syst. Technol. (MSST)*, May 2010, pp. 1–10.

[5] B. Calder *et al.*, "Windows Azure storage: A highly available cloud storage service with strong consistency," in *Proc. ACM Symp. Oper. Syst. Principles*, 2011, pp. 143–157.

[6] D. Ford *et al.*, "Availability in globally distributed storage systems," in *Proc. USENIX Symp. Oper. Syst. Design Implement.*, 2010, pp. 1–7.

[7] *HDFS and Erasure Codes (HDFS-RAID)*, accessed on Jun. 15, 2017. [Online]. Available: http://hadoopblog.blogspot.com/2009/08/hdfs-and-erasure-codes-hdfs-raid.html

[8] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *J. Soc. Ind. Appl. Math.*, vol. 8, no. 2, pp. 300–304, Jun. 1960.

[9] *Facebook's Approach to Big Data Storage Challenge*, accessed on Jun. 15, 2017. [Online]. Available: http://www.slideshare.net/Hadoop_Summit/facebooks-approach-to-big-data-storage-challenge

[10] *HDFS RAID*, accessed on Jun. 15, 2017. [Online]. Available: http://www.slideshare.net/ydn/hdfs-raid-facebook

[11] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," *IEEE Trans. Inf. Theory*, vol. 56, no. 9, pp. 4539–4551, Sep. 2010.

[12] Y. Wu, "Existence and construction of capacity-achieving network codes for distributed storage," *IEEE J. Sel. Areas Commun.*, vol. 28, no. 2, pp. 277–288, Feb. 2010.

[13] 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 Trans. Inf. Theory*, vol. 57, no. 8, pp. 5227–5239, Aug. 2011.

[14] N. B. Shah, K. V. Rashmi, P. V. Kumar, and K. Ramchandran, "Interference alignment in regenerating codes for distributed storage: Necessity and code constructions," *IEEE Trans. Inf. Theory*, vol. 58, no. 4, pp. 2134–2158, Apr. 2012.

[15] C. Suh and K. Ramchandran, "Exact-repair MDS code construction using interference alignment," *IEEE Trans. Inf. Theory*, vol. 52, no. 3, pp. 1425–1442, Mar. 2011.

[16] D. Papailiopoulos, A. G. Dimakis, and V. Cadambe, "Repair optimal erasure codes through Hadamard designs," *IEEE Trans. Inf. Theory*, vol. 59, no. 5, pp. 3021–3037, May 2013.

[17] I. Tamo, Z. Wang, and J. Bruck, "Zigzag codes: MDS array codes with optimal rebuilding," *IEEE Trans. Inf. Theory*, vol. 59, no. 3, pp. 1597–1616, Mar. 2013.

[18] V. R. Cadambe, C. Huang, and J. Li, "Permutation code: Optimal exact-repair of a single failed node in MDS code based distributed storage systems," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Aug. 2011, pp. 1225–1229.

[19] I. Tamo, Z. Wang, and J. Bruck, "Access versus bandwidth in codes for storage," *IEEE Trans. Inf. Theory*, vol. 60, no. 4, pp. 2028–2037, Apr. 2014.

[20] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A Hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers," in *Proc. ACM SIGCOMM*, 2014, pp. 331–342.

[21] V. R. Cadambe, C. Huang, J. Li, and S. Mehrotra, "Polynomial length MDS codes with optimal repair in distributed storage," in *Proc. Conf. Rec. 45th Asilomar Conf. Signals, Syst. Comput. (ASILOMAR)*, Nov. 2011, pp. 1850–1854.

[22] S. Goparaju, I. Tamo, and R. Calderbank, "An improved sub-packetization bound for minimum storage regenerating codes," *IEEE Trans. Inf. Theory*, vol. 60, no. 5, pp. 2770–2779, May 2014.

[23] B. Sasidharan, G. K. Agarwal, and P. V. Kumar, "A high-rate MSR code with polynomial sub-packetization level," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jun. 2015, pp. 2051–2055.

[24] V. Guruswami and A. S. Rawat, "MDS code constructions with small sub-packetization and near-optimal repair bandwidth," in *Proc. ACM-SIAM Symp. Discrete Algorithms*, 2017, pp. 2109–2122.

[25] 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 Conf. File Storage Technol. (FAST)*, 2012, p. 20.
[26] M. Blaum, J. Brady, J. Bruck, and J. Menon, "EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures," *IEEE Trans. Comput.*, vol. 44, no. 2, pp. 192–202, Feb. 1995.
[27] Z. Wang, A. Dimakis, and J. Bruck, "Rebuilding for array codes in distributed storage systems," in *Proc. Workshop Appl. Commun. Theory Emerg. Memory Technol. (ACTEMT)*, Dec. 2010, pp. 1905–1909.
[28] P. Corbett *et al.*, "Row-diagonal parity for double disk failure correction," in *Proc. USENIX Conf. File Storage Technol. (FAST)*, 2004, pp. 1–14.
[29] L. Xiang, Y. Xu, J. Lui, and Q. Chang, "Optimal recovery of single disk failure in RDP code storage systems," in *Proc. ACM SIGMETRICS*, 2010, pp. 119–130.
[30] F. Oggier and A. Datta, "Self-repairing homomorphic codes for distributed storage systems," in *Proc. IEEE INFOCOM*, Apr. 2011, pp. 1215–1223.
[31] P. Gopalan, C. Huang, H. Simitci, and S. Yekhanin, "On the locality of codeword symbols," *IEEE Trans. Inf. Theory*, vol. 58, no. 11, pp. 6925–6934, Nov. 2012.
[32] D. S. Papailiopoulos and A. G. Dimakis, "Locally repairable codes," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Oct. 2012, pp. 5843–5855.
[33] G. M. Kamath, N. Prakash, V. Lalitha, and P. V. Kumar, "Codes with local regeneration," in *Proc. Inf. Theory Appl. Workshop (ITA)*, 2013, pp. 1606–1610.
[34] L. Pamies-Juarez, H. D. Hollmann, and F. Oggier, "Locally repairable codes with multiple repair alternatives," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jul. 2013, pp. 892–896.
[35] I. Tamo and A. Barg, "A family of optimal locally recoverable codes," *IEEE Trans. Inf. Theory*, vol. 60, no. 8, pp. 4661–4676, Aug. 2014.
[36] N. Silberstein, A. S. Rawat, and S. Vishwanath, "Error-correcting regenerating and locally repairable codes via rank-metric codes," *IEEE Trans. Inf. Theory*, vol. 61, no. 11, pp. 5765–5778, Nov. 2015.
[37] N. B. Shah, K. V. Rashmi, P. V. Kumar, and K. Ramchandran, "Explicit codes minimizing repair bandwidth for distributed storage," in *Proc. IEEE Inf. Theory Workshop (ITW)*, Jan. 2010, pp. 1–5.
[38] Z. Wang, I. Tamo, and J. Bruck, "On codes for optimal rebuilding access," in *Proc. Allerton Conf. Commun., Control, Comput.*, 2011, pp. 1374–1381.
[39] *Apache Hadoop Release 3.0.0-Alpha1*, accessed on Jun. 15, 2017. [Online]. Available: https://github.com/apache/hadoop/tree/release-3.0.0-alpha1-RC0
[40] V. R. Cadambe, S. A. Jafar, H. Maleki, K. Ramchandran, and C. Suh, "Asymptotic interference alignment for optimal repair of MDS codes in distributed storage," *IEEE Trans. Inf. Theory*, vol. 59, no. 5, pp. 2974–2987, May 2013.
[41] 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 Trans. Inf. Theory*, vol. 58, no. 3, pp. 1837–1852, Mar. 2012.
[42] C. Tian, "Characterizing the rate region of the (4, 3, 3) exact-repair regenerating codes," *IEEE J. Sel. Areas Commun.*, vol. 32, no. 5, pp. 967–975, May 2014.
[43] K. Rashmi, P. Nakkiran, J. Wang, N. B. Shah, and K. Ramchandran, "Having your cake and eating it too: Jointly optimal erasure codes for I/O, storage, and network-bandwidth," in *Proc. USENIX Conf. File Storage Technol. (FAST)*, 2015, pp. 81–94.
[44] E. E. Gad, R. Mateescu, F. Blagojevic, C. Guyot, and Z. Bandic, "Repair-optimal MDS array codes over GF(2)," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jul. 2013, pp. 887–891.
[45] K. Shanmugam, D. S. Papailiopoulos, A. G. Dimakis, and G. Caire, "A repair framework for scalar mds codes," *IEEE J. Sel. Areas Commun.*, vol. 32, no. 5, pp. 998–1007, May 2014.
[46] V. Guruswami and M. Wootters, "Repairing Reed-Solomon codes," in *Proc. ACM Symp. Theory Comput.*, 2016, pp. 216–226.
[47] B. Yang, X. Tang, and J. Li, "A systematic piggybacking design for minimum storage regenerating codes," *IEEE Trans. Inf. Theory*, vol. 61, no. 11, pp. 5779–5786, Nov. 2015.
[48] M. Blaum, J. Bruck, and A. Vardy, "MDS array codes with independent parity symbols," *IEEE Trans. Inf. Theory*, vol. 42, no. 2, pp. 529–542, Mar. 1996.
[49] D. S. Papailiopoulos, A. G. Dimakis, and V. R. Cadambe, "Repair optimal erasure codes through Hadamard designs," in *Proc. Allerton Conf. Commun., Control, Comput.*, Sep. 2011, pp. 1382–1389.

K. V. Rashmi is a postdoctoral researcher in the EECS department at UC Berkeley, where she received her Ph.D. in 2016. She received the Eli Jury Award 2016 from EECS UC Berkeley for "outstanding achievement in the area of systems, communications, control, or signal processing." Rashmi is also the recipient of the IEEE Data Storage Best Paper and Best Student Paper Awards for the years 2011/2012, the Facebook Fellowship 2012-13, the Microsoft Research Ph.D. Fellowship 2013-15, and the Google Anita Borg Memorial Scholarship 2015-16. Her research interests lie in the theoretical and system challenges that arise in storage and analysis of big data.

Nihar B. Shah is a Ph.D. candidate in the EECS department at the University of California, Berkeley. He is a recipient of the David J. Sakrison memorial prize 2017 from EECS Berkeley for "truly outstanding and innovative research" in his Ph.D. dissertation. He is also a recipient of the Microsoft Research Ph.D. Fellowship 2014-16, the Berkeley Fellowship 2011-13, the IEEE Data Storage Best Paper and Best Student Paper Awards for the years 2011/2012, and the SVC Aiya Medal from the Indian Institute of Science for the best master's thesis in the department. His research interests include statistics and machine learning, with a current focus on applications to learning from people.

Kannan Ramchandran (Ph.D.: Columbia University, 1993) is a Professor of Electrical Engineering and Computer Science at UC Berkeley, where he has been since 1999. Prior to that, he was on the faculty at the University of Illinois from 1993 to 1999. Prof. Ramchandran is a recipient of the 2017 IEEE Kobayashi Computers and Communications award for his pioneering contributions to the theory and practice of distributed storage coding and distributed compression. He is a Fellow of the IEEE, has published extensively in his field, and holds over a dozen patents. He has received several awards for his research and teaching including an IEEE Information Theory Society and Communication Society Joint Best Paper award for 2012, an IEEE Communication Society Data Storage Best Paper award in 2010, two Best Paper awards from the IEEE Signal Processing Society in 1993 and 1999, an Okawa Foundation Prize for outstanding research at Berkeley in 2001, and a Departmental Outstanding Teaching Award at Berkeley in 2009.