Erasure Coding for Big-data Systems: Theory and Practice

Rashmi Vinayak

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2016-155
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-155.html

September 14, 2016

Erasure Coding for Big-data Systems: Theory and Practice

by

Rashmi Korlakai Vinayak

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering – Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Kannan Ramchandran, Chair
Professor Ion Stoica
Professor Randy Katz
Professor Rhonda Righter

Fall 2016

Erasure Coding for Big-data Systems: Theory and Practice

Abstract

Erasure Coding for Big-data Systems: Theory and Practice

by

Rashmi Korlakai Vinayak

Doctor of Philosophy in Engineering – Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Kannan Ramchandran, Chair

Big-data systems enable storage and analysis of massive amounts of data, and are fueling the data revolution that is impacting almost all walks of human endeavor today. The foundation of any big-data system is a large-scale, distributed, data storage system. These storage systems are typically built out of inexpensive and unreliable commodity components, which in conjunction with numerous other operational glitches make unavailability events the norm rather than the exception.

In order to ensure data durability and service reliability, data needs to be stored redundantly. While the traditional approach towards this objective is to store multiple replicas of the data, today's unprecedented data growth rates mandate more efficient alternatives. Coding theory, and erasure coding in particular, offers a compelling alternative by making optimal use of the storage space. For this reason, many data-center scale distributed storage systems are beginning to deploy erasure coding instead of replication. This paradigm shift has opened up exciting new challenges and opportunities both on the theoretical as well as the system design fronts. Broadly, this thesis addresses some of these challenges and opportunities by contributing in the following two areas:

- **Resource-efficient distributed storage codes and systems:** Although traditional erasure codes optimize the usage of storage space, they result in a significant increase in the consumption of other important cluster resources such as the network bandwidth, input-output operations on the storage devices (I/O), and computing resources (CPU). This thesis considers the problem of constructing codes, and designing and building storage systems, that reduce the usage of I/O, network, and CPU resources while not compromising on storage efficiency.

- **New avenues for erasure coding in big-data systems:** In big-data systems, the usage of erasure codes has largely been limited to disk-based storage systems, and furthermore, primarily towards achieving space-efficient fault tolerance—in other words, to durably store "cold" (less-frequently accessed) data. This thesis takes a step forward in exploring new avenues for erasure coding—in particular for "hot" (more-frequently accessed) data—by showing how erasure coding can be employed to improve load balancing, and to reduce the (median and tail) latencies in data-intensive cluster caches.

An overarching goal of this thesis is to bridge theory and practice. Towards this goal, we present new code constructions and techniques that possess attractive theoretical guarantees. We also design and build systems that employ the proposed codes and techniques. These systems exhibit significant benefits over the state-of-the-art in evaluations that we perform in real-world settings, and are also slated to be a part of the next release of Apache Hadoop.

Contents

Acknowledgments

First of all, I would like to express my deepest gratitude to my advisor Kannan Ramchandran for his continued support and guidance throughout my PhD studies. Kannan provided me with ample freedom in choosing the problems to work on all the while also providing the necessary guidance. The confidence that he vested in me right from the beginning of my PhD has been a constant source of motivation.

I would also like to thank my dissertation committee members Ion Stocia, Randy Katz and Rhonda Righter, for their valuable feedback during and after my qualifying examination as well as on the final dissertation. I also had the opportunity to closely collaborate with Ion, which was a great learning experience.

Early on in my PhD, I did a summer internship at Facebook in their data warehouse team, which helped me in understanding and appreciating the real-world systems and their requirements. This understanding has inspired and guided my research in many ways. I am very grateful to my mentors and collaborators at Facebook, Hairong Kuang, Dhruba Borthakur, and Dikang Gu, for enabling this learning experience. I also spent two wonderful summers interning at Microsoft Research Redmond which provided me with an opportunity to explore topics completely different from my PhD research area. This added to the breadth of my learning and research. I am thankful to my mentors and managers at Microsoft Research: Ran Gilad-Bachrach, Cheng Huang and Jin Li.

This work would not have been possible without a number of amazing collaborators that I had the fortune to work with: Nihar B. Shah, Preetum Nakkiran, Jingyan Wang, Dikang Gu, Hairong Kuang, Dhruba Borthakur, Mosharaf Chowdhury, and Jack Kosaian. I have also had the opportunity to collaborate and interact with amazing people on works which are not covered in this thesis: Kangwook Lee, Giulia Fanti, Asaf Cidon, Mitya Voloshchuk, Kai Zheng, Zhe Zhang, Jack Liuquan, Ankush Gupta, Diivanand Ramalingam, Jeff Lievense, Rishi Sharma, Tim Brown, and Nader Behdin.

I was a graduate student instructor (GSI) for two courses at Berkeley, and I was very lucky to have amazing professors as my mentors in both the courses — Anant Sahai and Thomas Courtade. I learnt a lot from both of them and this experience has further inspired me to pursue a career that also involves teaching. In fact, with Anant we almost completely revamped the EE121 course bringing in current research topics, and this was a great learning experience in course design.

I am deeply indebted to all the amazing teachers I had here at UC Berkeley, Jean Walrand, Ion Stoica, Randy Katz, Martin J. Wainright, Michael Jordan, Bin Yu, Ali Ghodsi, Michael Mahoney, Laurent El Ghaoui, Kurt Keutzer and Steve Blank. I am also very grateful to P.

Vijay Kumar, who was my masters' advisor at Indian Institute of Science, prior to joining UC Berkeley, from whom I have learnt a lot both about research and life in general.

I am grateful to Facebook, Microsoft Research, and Google, for financially supporting my PhD studies through their gracious fellowships.

I am thankful to all the members of the Berkeley Laboratory of Information and System Sciences (BLISS) (formely Wireless Foundations, WiFo) for providing a fun-filled and inspiring work environment: Sameer Pawar, Venkatesan Ekambaram, Hao Zhang, Giulia Fanti, Dimitris Papailiopoulos, Sahand Negahban, Guy Bresler, Pulkit Grover, Salim El Rouayheb, Longbo Huang, I-Hsiang Wang, Naveen Goela, Se Yong Park, Sreeram Kannan, Nebojsa Milosavljevic, Joseph Bradley, Soheil Mohajer, Kate Harrison, Nima Noorshams, Seyed Abolfazl Motahari, Gireeja Ranade, Eren Sasoglu, Barlas Oguz, Sudeep Kamath, Poling Loh, Kangwook Lee, Kristen Woyach, Vijay Kamble, Stephan Adams, Nan Ma, Nihar B. Shah, Ramtin Pedarsani, Ka Kit Lam, Vasuki Narasimha Swamy, Govinda Kamath, Fanny Yang, Dileep M. K., Steven Clarkson, Reza Abbasi Asl, Vidya Muthukumar, Ashwin Pananjady, Dong Yin, Orhan Ocal, Payam Delgosha, Yuting Wei, Ilan Shomorony, Reinhard Heckel, Yudong Chen, Simon Li, Raaz Dwivedi, Sang Min Han, and Soham Phade. I am also thankful to my friends at AMPLab where I started spending more time towards the end of my PhD: Neeraja Yadwadkar, Wenting Zheng, Arka Aloke Bhattacharya, Sara Alspaugh, Kristal Curtis, Orianna Dimasi, Virginia Smith, Shivaram Venkataraman, Evan Sparks, Qifan Pu, Anand Iyer, Becca Roelofs, K. Shankari, Anurag Khandelwal, Elaine Angelino, Daniel Haas, Philipp Moritz, Ahmed El Alaoui, and Aditya Ramdas. Many friends have enriched my life outside of work at Berkeley. Many thanks to each and everyone of you.

Thanks to all the members of the Women In Computer Science and Electrical Engineering (WICSE) for the wonderful conversations over our weekly lunches. Thanks to Sheila Humphreys, the former Director of Diversity in the EECS department and an ardent supporter of WICSE, for her never-ending enthusiasm and for constantly inspiring WICSE members.

I would also like to express gratitude to Shirley Salanio, Kim Kail, Jon Kuroda, Kattt Atchley, Boban Zarkovich, Carlyn Chinen, and all the amazing EECS staff who were always happy to help.

I am deeply grateful to my family for their unconditional love and support over the years. My mother, Jalajakshi Hegde, has always encouraged me to aim higher and higher goals. She has made countless sacrifices to ensure that her children received the best education and that they focused on studies more than anything else. Right from my childhood, she instilled a deep appreciation for education and its ability to transform ones life. This has been the driving force throughout my life. My younger sister, Ramya, has always been a very supportive companion right from childhood. I am thankful to her for being there for me whenever I have needed her support. Finally and most importantly, the love, support, and understanding from my husband Nihar Shah has been the backbone of my graduate studies. He made the journey so much more fun and has helped me go through the ups and downs of research without getting carried away. Thank you Nihar for being the amazing person that you are!

Chapter 1

Introduction

We are now living in the age of big data. People, enterprises and “smart” things are generating enormous amounts of digital data; and the insights derived from this data have shown the potential to impact almost all aspects of human endeavor from science and technology to commerce and governance. This data revolution is being enabled by so-called “big-data systems”, which make it possible to store and analyze massive amounts of data.

With the advent of big data, there has been a paradigm shift in the way computing infrastructure is evolving: cheap, failure-prone, moderately-powerful commodity components are being extensively employed in building warehouse-scale distributed computing infrastructure in contrast to the paradigm of high-end supercomputers which are built out of expensive, highly reliable, and powerful components. A typical big-data system comprises a distributed storage layer which allows one to store and access enormous amounts of data, an execution layer which orchestrates execution of tasks and manages the run-time environment, and an application layer comprising of various applications which allows users to manipulate and analyze data. Thus, the distributed storage layer forms the foundation on which big-data systems function, and this layer will be the focus of the thesis.

In a distributed storage system, data is stored using a distributed file system (DFS) that spreads data across a cluster consisting of hundreds to thousands of servers connected through a networking infrastructure. Such clusters are typically built out of commodity components, and failures are the norm rather than the exception in their day-to-day operation [37, 55]. [1] There are numerous sources of malfunctioning that can lead to data becoming unavailable from time-to-time, such as hardware failures, software bugs, maintenance shutdowns, power failures, issues in networking components etc. In the face of such incessant

[1]We will present our measurements on unavailability events from Facebook’s data warehouse cluster in production in Chapter 3.

unavailability events, it is the responsibility of the DFS to ensure that the data is stored in a reliable and durable fashion. Equally important is its responsibility to ensure that the performance guarantees in terms of latencies (both median and tail latencies) are met. [2] Hence the DFS has to ensure that any requested data is available to be accessed without much delay. In order to meet these objectives, distributed file systems store data *redundantly*, monitor the system to keep track of unavailable data, and recover the unavailable data to maintain the targeted redundancy level.

A typical approach for introducing redundancy in distributed storage systems has been to *replicate* the data [57, 158], that is, to store multiple copies of the data on distinct servers spread across different failure domains. While the simplicity of the replication strategy is appealing, the rapid growth in the amount of data needing to be stored has made storing multiple copies of the data an expensive solution. The volume of data needing to be stored is growing at a rapid rate, surpassing the efficiency rate corresponding to Moore's law for storage devices. Thus, in spite of the continuing decline in the cost of storage devices, replication is too extravagant a solution for large-scale storage systems.

Coding theory (and erasure coding specifically) offers an attractive alternative for introducing redundancy by making more efficient use of the storage space in providing fault tolerance. For this reason, large-scale distributed storage systems are increasingly turning towards erasure coding, with traditional Reed-Solomon (RS) codes being the popular choice. For instance, Facebook HDFS [66], Google Colossus [34], and several other systems [181, 147] employ RS codes. RS codes make optimal use of storage resources in the system for providing fault tolerance. This property makes RS codes appealing for large-scale, distributed storage systems where storage capacity is a critical resource.

Under traditional erasure codes such as RS codes, redundancy is introduced in the following manner: A file to be stored is divided into equal-sized units. These units are grouped into sets of k each, and for each such set of k units, r *parity* units (which are some mathematical functions of the k original units) are computed. The set of these $(k + r)$ units constitute a *stripe*. The data and parity units belonging to a stripe are placed on different servers, typically chosen from different failure domains. The parity units possess the property that any k out the $(k + r)$ units in a stripe suffice to recover the original data. Thus, failure of any r units in a stripe can be tolerated without any data loss.

Broadly, this thesis derives its motivation from the following two considerations:

- Although traditional erasure codes optimize the usage of storage space, they result in a significant increase in the usage of other important cluster resources such as the

[2]Such guarantees are typically termed as service level agreements (SLAs).

network bandwidth, the input-output operations on the storage devices (I/O), and the computing resources (CPU): in large-scale distributed storage systems, operations to recover unavailable data are almost constantly running in the background. Since there are no replicas in an erasure-coded system, a reconstruction operation necessitates reading and downloading data from several other units from the stripe. This results in significant amount of I/O and network transfers. Further, the encoding/decoding operations increase the usage of computational resources. In this thesis, we will consider the problem of constructing codes, and designing and building storage systems that reduce the usage of network, I/O, and CPU resources while not compromising on storage efficiency.

- In big-data systems, the usage of erasure codes has been largely limited to disk-based storage systems and primarily towards achieving fault tolerance in a space-efficient manner. This is identical to how erasure codes are employed in communication channels for reliably transmitting bits at the highest possible bit rate. Given the complexity of big-data systems and the myriad metrics of interest, erasure coding has the potential to impact big-data systems beyond the realm of disk-based storage systems and for goals beyond just fault tolerance. This potential has largely been unexplored. This thesis takes a step forward in exploring new avenues for erasure coding by showing how they can be employed to improve load balancing, and to reduce the median and tail latencies in data-intensive cluster caches.

1.1 Thesis goals and contributions

The goals of this thesis are three fold:

1. Construct practical distributed-storage-codes that optimize various system resources such as storage, I/O, network and CPU.
2. Design and build distributed storage systems that employ these new code constructions in order to translate their promised theoretical gains to gains in real-world systems.
3. Explore new avenues for the applicability of erasure codes in big-data systems (beyond fault tolerance and beyond disk-based storage), and design and build systems that validate the performance benefits that codes can realize in these new settings.

An overarching objective of this thesis is to bridge theory and practice. Towards this objective and the aforementioned goals, this thesis makes the following contributions:

- We present our measurements from Facebook's data warehouse cluster in production, focusing on important relevant attributes related to erasure coding such as statistics related to data unavailability in the cluster and how erasure coding impacts resource consumption in the cluster. This provides an insight into how large-scale distributed storage systems in production are employing traditional erasure codes. The measurements presented also serve as a concrete, real-world example motivating the work on optimizing resource consumption in erasure-coded distributed storage systems.

- We present a new framework for constructing distributed storage codes, which we call the *piggybacking* framework, that offers a rich design space for constructing storage codes optimizing for I/O and network usage while retaining the storage efficiency offered by RS codes. We illustrate the power of this framework by constructing explicit storage codes that feature the minimum usage of I/O and network bandwidth among all existing solutions among three classes of codes. One of these classes addresses the constraints arising out of system considerations in big-data systems, thus leading to a practical code construction easily deployable in real-world systems. In addition, we show how the piggybacking framework can be employed to enable efficient reconstruction of the parity units in existing codes that were originally designed to address the reconstruction of only the data units.

- We present *Hitchhiker*, a resource-efficient erasure-coded storage system that reduces both network and disk traffic during reconstruction by 25% to 45% without requiring any additional storage and maintaining the same level of fault-tolerance as RS-based systems. Hitchhiker accomplishes this with the aid of the following two components: (i) an erasure code built on top of RS codes using the Piggybacking framework, (ii) a disk layout (or data placement) technique that translates the savings in network traffic offered by the code to savings in disk traffic (and disk seeks) as well. The proposed data-placement technique for reducing the number of seeks is applicable in general to a broad class of storage codes, and is therefore of independent intellectual interest. We implement Hitchhiker on top of the Hadoop Distributed File System, and evaluate it on Facebook's data warehouse cluster in production with real-time traffic showing significant reduction in time taken to read data and perform computations during reconstruction along with the reduction in network and disk traffic.

- We present erasure codes aimed at jointly optimizing the usage of storage, network, I/O and CPU resources. First, we design erasure codes that are simultaneously optimal in terms of I/O, storage, and network usage. Here, we present a transformation that can be employed on existing classes of storage codes called *minimum-storage-regenerating codes* [41] in order to optimize their usage of I/O while retaining their optimal usage of storage and network. Through evaluations on Amazon EC2, we show that our proposed design results in a significant reduction in IOPS (that is, input-output operations per second) during reconstructions: a 5× reduction for typical parameters. Second, we show that optimizing I/O and CPU go hand-in-hand in these resource-efficient distributed storage codes, by showing that the transformation that optimizes I/O also sparsifies the code thereby reducing the computational complexity.

- In big-data systems, erasure codes have been primarily employed for reliably storing "cold" (less-frequently accessed) data in a storage efficient manner. We explore how erasure coding can be employed for improving performance in serving "hot" (more-frequently accessed) data. Data-intensive clusters rely on in-memory object caching to maximize the number of requests that can be served from memory in the presence of popularity skew, background load imbalance, and server failures. For improved load-balancing and reduced I/O latency, these caches typically employ selective replication, where the number of cached replicas of an object is proportional to its popularity. We show that erasure coding can be effectively employed to provide improved load-balancing under skewed popularity, and to reduce both median and tail latencies in cluster caches. We present *EC-Cache*, a load-balanced, high-performance cluster cache that employs erasure coding as a critical component of its data-serving path. We implement EC-Cache over Alluxio, a popular cluster cache, and through evaluations on Amazon EC2, show that EC-Cache improves load balancing by a factor of 3.3× and reduces the median and tail read latencies by more than 2×, while using the same amount of memory. We also show that the benefits offered by EC-Cache are further amplified in the presence of imbalance in the background network load.

1.2 Organization

The organization of the rest of the thesis is as follows.

Chapter 2 introduces the background and terminology that will be used in the upcoming chapters. A high-level overview of the landscape of the related literature is also provided in this chapter. More extensive discussions on the related works in relation to the contributions of this thesis are provided in the respective chapters.

Chapter 3 presents our measurements and observations from Facebook's data warehouse cluster in production focusing on various aspects related to erasure coding. This chapter is based on joint work with Nihar Shah, Dikang Gu, Hairong Kuang, Dhruba

Borthakur, and Kannan Ramchandran, and has been presented at USENIX HotStorage 2013 [131].

Chapter 4 presents the Piggybacking framework and code constructions based on this framework. This chapter is based on joint work with Nihar Shah and Kannan Ramchandran. The results in this chapter have been presented in part at IEEE International Symposium on Information Theory (ISIT) 2013 [128].

Chapter 5 changes gears from theory to systems and presents Hitchhiker and its evaluation on Facebook's data warehouse cluster. This chapter is based on joint work with Nihar Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran, and has been presented at ACM SIGCOMM 2014 [134].

Chapter 6 deals with constructing codes that jointly optimize storage, network, I/O and CPU resources. This chapter is based on joint work with Preetum Nakkiran, Jingyan Wang, Nihar B. Shah, and Kannan Ramchandran. The results in this chapter have been presented in part at USENIX Conference File and Storage Technologies (FAST) 2015 [136] and in part at IEEE ISIT 2016 [105].

Chapter 7 presents EC-Cache, a cluster cache that employs erasure coding for load balancing and for reducing median and tail latencies. This chapter is based on joint work with Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. The results in this chapter will be presented at the USENIX Symposium on Operating Systems Design and Implementation (OSDI) 2016 [135].

Chapter 2

Background and Related Work

In this chapter, we will first introduce the notation, terminology and some background material that we will refer to in the upcoming chapters. We will then provide a high-level overview of the landscape of the related literature. More extensive discussions on the related works in relation to the contributions of this thesis are provided in the respective chapters.

2.1 Notation and terminology

We will start by presenting some notation and terminology. In this thesis, we will be dealing with erasure codes in the context of computer systems, and hence we will start our introduction to erasure coding from this context rather than from the context of classical coding theory.

Erasure codes

In the systems context, erasure coding can be viewed as an operation that takes k units of data and generates $n = (k + r)$ units of data that are functions of the original k data units. Typically, in the codes employed in storage systems, the first k of the resultant n units are identical to the original k units. These units are called *data* units (or *systematic*) units. The r additional units generated are called *parity* units. The parity units are some mathematical functions of the data units, and thus contain redundant information associated with the data units. This set of $n = (k + r)$ units is called a *stripe*.

In the coding theory literature, an erasure code is associated to the two parameters n and k introduced above, and a code is referred to as an $(n,\ k)$ code. On the other hand, in the computer systems literature, an erasure code is identified with the two parameters k and r introduced above, and a code is referred to as a $(k,\ r)$ code. We will use both of these notations, as appropriate.

The computations performed to obtain the parity units from the data units is based on what is called *finite-field-arithmetic*. Such an arithmetic is defined over a set of elements called a *finite field*. The defined arithmetic operations on the elements of a finite field result in an element within the finite field. The number of elements in a finite field is referred to as its *size*. A finite field of size q is denoted as $\mathbb{F}_q$. Elements of $\mathbb{F}_q$ can be represented using bit vector of length $\lceil \log_2(q) \rceil$. In practice, the field size is usually chosen to be a power of two, so that the elements of the field can be efficiently represented using bit vectors leading to efficient implementations. In this thesis, we will not use any specific properties of the finite fields, and for simplicity, the reader may choose to consider usual arithmetic without any loss in comprehension.

Erasure coding in distributed storage systems

A file to be stored is first divided into equal-sized units that are also called as *blocks*. These units are then grouped into sets of k each, and for each such set of k units r additional units are computed using a (k, r) erasure code. The data and parity units belonging to a stripe are placed on different servers (also referred as *nodes*), typically chosen from different failure domains. In a distributed storage system, a node stores a large number of units belonging to different stripes. Each stripe is conceptually independent and identical, and hence, without loss of generality, we will typically consider only a single stripe. Given the focus on a single stripe, with a slight abuse of terminology, we will at times (interchangeably) refer to an unit as a *node* (since only one unit from a particular stripe will be stored on any given node).

Maximum-Distance-Separable (MDS) codes

If the erasure code employed is a *Maximum-Distance-Separable* (MDS) code [100], the storage system will be optimal in utilizing the storage space for providing fault tolerance. Specifically, under a $(k,\ r)$ MDS code, each node stores a $(\frac{1}{k})^{\text{th}}$ fraction of the data, and has the property that the entire data can be decoded from any k out of the n $(= k + r)$ nodes. Consequently, such a code can tolerate the failure of *any* r of the n nodes without any data loss. A single stripe of a $(k = 4,\ r = 2)$ MDS code is depicted in Figure 2.1, where $\{a_1, a_2, a_3, a_4\}$ are the finite field elements corresponding to the data that is encoded. Observe that each node stores $\frac{1}{4}^{\text{th}}$ fraction of the total data, and all the data can be recovered from the data stored

Unit 1	a_1
Unit 2	a_2
Unit 3	a_3
Unit 4	a_4
Unit 5	$\sum_{i=1}^{4} a_i$
Unit 6	$\sum_{i=1}^{4} i a_i$

Figure 2.1: A stripe of a ($k=4$, $r=2$) MDS code, with four data units and two parity units.

in any four nodes.

Code rate and Storage overhead

The *redundancy* or the *storage overhead* of a code is the ratio of the physical storage space consumed to the actual (logical) size of the data stored, i.e.,

$$\text{Storage overhead or redundancy} = \frac{n}{k}. \tag{2.1}$$

The redundancy factor thus reflects the additional storage space used by the code.

Following the terminology in the communications literature, the *rate* of an MDS storage code is defined as

$$\text{Rate} = \frac{k}{n}. \tag{2.2}$$

Thus, the rate of a code has an inverse relationship with the redundancy of the code: high-rate codes have low redundancy and vice versa.

Systematic codes

In general, all the n units in a stripe of a code can be parity units, that is, functions of the data units. Codes which have the property that the original k data units are available in uncoded form among the n units in a stripe are called *systematic* codes. Systematic codes have the advantage that the read requests to any of the data units can be served without having to perform any decoding operation. The example depicted in Figure 2.1 is a systematic code as the original data units are available in uncoded form in the first four units.

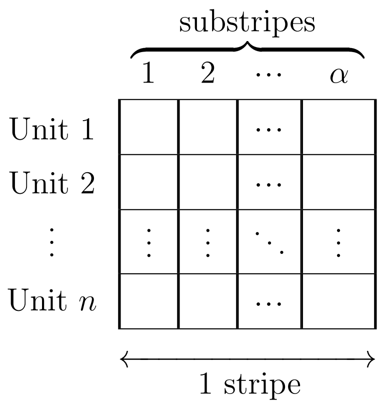


Figure 2.2: One stripe of a vector storage code consisting of α substripes. Each cell of the table corresponds to an element from the finite field of operation for the code. When restricting attention to only a single stripe, each row corresponds to a unit. Each column corresponds to a substripe. Here the number of substripes is equal to α.

Vector codes and number of substripes

The MDS code depicted in Figure 2.1 consists of a single element from the finite field over which the code is constructed in each unit. In general, each individual unit can be a vector of elements from the finite field over which the code is constructed. Such codes are called *vector codes*. Each stripe of a vector code can be viewed as consisting of one or more *substripes*. Figure 2.2 depicts a single stripe of a vector code with α substripes. In the literature, the number of substripes is also referred by the term *subpacketization*.

2.2 Problem of reconstructing erasure-coded data

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 unavailability of one or more nodes in the stripe. In order to maintain the targeted 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 *reconstruction* operation. We will also use the terms *recovery* and *repair* operations interchangeably to refer to reconstruction operations. In large-scale systems, such reconstruction operations are run as background jobs. Reconstruction operations also have a second, foreground application,

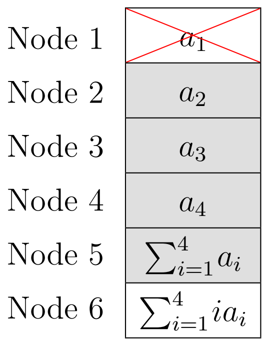


Figure 2.3: The traditional MDS code reconstruction framework: the first node (storing the finite field element a_1) is reconstructed by downloading the finite field elements from nodes 2, 3, 4 and 5 (highlighted in gray) from which a_1 can be recovered. In this example, the amount of data read and transferred is $k = 4$ times the amount of data being reconstructed.

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 reconstructing the requisite data on the fly, that is, the reconstruction operation is run immediately as a foreground job.

Under replication, a reconstruction operation is carried out by copying the desired data from one of it replicas and creating a new replica on another node in the system. Here, the the amount of data read and transferred is equal to the amount of data being reconstructed. However, under erasure coding, there are no replicas, and hence, the part of the data that is unavailable needs to be reconstructed through a decoding operation. Under the traditional reconstruction framework for MDS codes, reconstruction 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. An example illustrating such a traditional reconstruction operation is depicted in Figure 2.3. In this example, data from nodes $\{2, 3, 4, 5\}$ are used to reconstruct the data in node 1.

The nodes helping in a reconstruction operation are termed the *helper nodes.* Each helper node transfers some data, which is in general a function of the data stored in it, to aid in the reconstruction of the failed or otherwise unavailable node. The data from all the helper nodes is downloaded at a server and a decoding operation is performed to recover the data that was stored on the failed or otherwise unavailable node. Thus a reconstruction operation generates data transfers through the interconnecting network, which consume the network bandwidth of the cluster. The amount of data transfer involved in a reconstruction operation is also referred interchangeably by the terms *reconstruction bandwidth* or *data download* for reconstruction. A reconstruction operation also generates input-output operations at the storage devices at the helper nodes. The amount of data that is required to be read from the storage devices at the helper nodes is referred to as the amount of *data read* or *data accessed* or I/O. In the example depicted in Figure 2.3, nodes $\{2, 3, 4, 5\}$ are the helper nodes, and the amount of network transfer and I/O involved in the reconstruction are both four (finite field elements). If q is the size of the finite field over which the code is constructed, this corresponds to $\log_q(4)$ bits.

Thus, while traditional erasure codes provide significant increase in storage efficiency as compared to replication, they can result in significant increase in the amount of network transfers and I/O during reconstruction of failed or otherwise unavailable nodes.

2.3 Related literature

In this section, we provide a high-level overview of the landscape of the related literature. More extensive discussions on the related works in relation to the contributions of this thesis are provided in the respective chapters.

Reconstruction-efficient codes

There has been considerable amount of work in the recent past on constructing resource-efficient codes for distributed storage.

Regenerating codes

In a seminal work [41], Dimakis et al., introduced the regenerating codes model, which optimizes the amount of data downloaded during repair operations. In [41], the authors provide a network-flow (cutset) based lower 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. In [41, 182], the authors showed the theoretical existence of codes meeting the cutset bound for the functional repair setting. We will consider a more stringent requirement termed *exact* repair, wherein the reconstructed node is required to be identical to the failed node.

The cutset bound on the repair bandwidth 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 [41]. Two important points on the trade-off are its end points termed the Minimum-Storage-Regenerating (MSR) and the Minimum-Bandwidth-Regenerating (MBR) points. MSR codes are MDS, and thus minimize the amount of storage space consumed. For this minimal amount of storage space, MSR codes 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 under the requirement of exact repair [127, 149, 26, 25, 115, 169]. It has also been shown that the intermediate points are not achievable for exact repair [149], and that there is a non-vanishing gap at these intermediate points between the cutset bound and what is achievable [172]. Recently, there are a number of works on characterizing tighter outer bounds for the intermediate points [145, 46, 123, 43] and constructing codes for these points [60, 173, 45].

There have been several works on constructing explicit MSR and MBR codes [132, 127, 149, 156, 164, 115, 169, 26, 185]. MSR codes are of particular interest since they are MDS. The Product-Matrix MSR codes [127] are explicit, practical MSR code constructions which have linear number of substripes. However, these codes have a low rate (i.e., high redundancy), requiring a storage overhead of $\left(2 - \frac{1}{n}\right)$ or higher. In [25], the authors show the existence of high-rate MSR codes as the number of substripes approaches infinity. The MSR constructions presented in [115, 169, 26, 185] 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 has been shown 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 [168]. In [61], the authors 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. In [27, 144], MSR codes with polynomial number of substripes are presented for the setting of constant (high) rate greater than $\frac{2}{3}$.

Improving reconstruction efficiency in existing codes

Binary MDS codes have received special attention in the literature due to their extensive use in disk array systems, for instance, EVEN-ODD and RDP codes [21, 35]. The EVEN-ODD and RDP codes have been optimized for reconstruction in [178] and [183] respectively. A recent work [64] presents a reconstruction framework for traditional Reed-Solomon codes for reducing the amount of data transfer during reconstruction by downloading elements from a sub-field rather than the finite field over which the code is constructed. This work optimizes only the amount of data downloaded and not the amount of data read during reconstruction.

Computer-search based techniques

In [89], the authors present a search-based approach to find reconstruction 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 reconstruction-efficient MDS code construction based on RS codes, called Rotated-RS, with the number of substripes being 2. However, it supports at most 3 parities, and moreover, its fault-tolerance capability is established via a computer search.

Other settings

There are several other works in the literature that construct reconstruction-efficient codes that do not directly fall into the three areas listed above. These include codes that provide security [118, 148, 133, 159, 48, 138, 62, 170, 36, 83, 59, 75], co-operative reconstruction [88, 157, 70, 95, 80], different cost models [153, 3], oblivious updates [106, 107], and others [126, 154, 113].

Optimizing locality of reconstruction

Reconstruction-locality, that is the number of nodes contacted during a reconstruction operation, is another metric of interest that has been studied extensively in the recent literature [110, 58, 116, 84, 114, 167, 161]. However, all the code constructions under this umbrella trade the MDS property (that is storage efficiency) in order to achieve a locality smaller than k.

Erasure codes in storage systems

Since decades, disk arrays have employed erasure codes to achieve space-efficient fault-tolerance in the form of Redundant Array of Inexpensive Disks (RAID) systems [117]. The benefits of erasure coding over replication for providing fault tolerance in distributed storage systems has also been well studied [191, 180], and erasure codes have been employed in many settings such as network-attached-storage systems [2], peer-to-peer storage systems [93, 142], etc. Recently, erasure coding is being increasing deployed in datacenter-scale distributed storage systems [51, 66, 34, 104, 181, 74] to achieve fault tolerance while minimizing storage requirements.

Chapter 3

Measurements from Facebook's data warehouse cluster in production

In this chapter, we present our measurements from Facebook's data warehouse cluster in production that stores hundreds of petabytes of data across a few thousand machines. We present statistics related to data unavailability in the cluster and how erasure coding impacts resource consumption in the cluster. This study serves two purposes: (i) it provides an insight into how large-scale distributed storage systems in production are employing traditional erasure codes, and (ii) the measurements reveal that there is a significant increase in the network traffic due to the recovery operations of erasure-coded data, thus serving as a real-world motivation for the following chapters. To the best of our knowledge, this is the first study in the literature that looks at the effect of the reconstruction operations of erasure-coded data on the usage of network resources in data centers.

3.1 Overview of the warehouse cluster

In this section, we provide a brief description of Facebook's data warehouse cluster in production (based on the system in production in 2013), on which we performed the measurements presented in this chapter.

Before delving into the details of the data warehouse cluster, we will introduce some terms related to data-center architecture in general. In data centers, typically, the computing and the networking equipment are housed within *racks*. A rack is a group of 30-40 servers connected to a common network switch. This switch is termed the *top-of-rack* (TOR) switch.

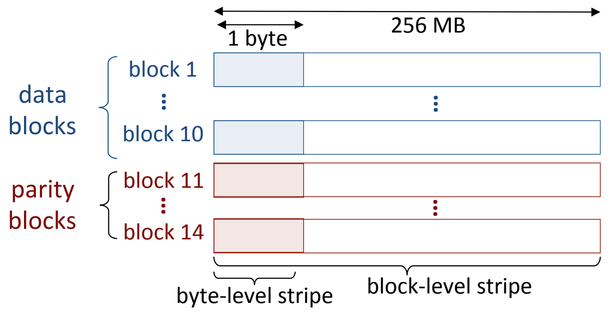


Figure 3.1: Erasure coding across blocks: 10 data blocks encoded using $(k = 10, r = 4)$ RS code to generate 4 parity blocks.

The top-of-rack switches from all the racks are interconnected through one or more layers of higher level switches (aggregation switches and routers) to provide connectivity from any server to any other server in the data center. Typically, the top-of-rack and higher level switches are heavily oversubscribed.

Facebook's data warehouse cluster in production is a Hadoop cluster, where the storage layer is the Hadoop Distributed File System (HDFS). The cluster comprises of two HDFS clusters, which we shall refer to as clusters A and B. In terms of the physical size, the two cluster together stores hundreds of petabytes of data, and the storage capacity used in the clusters is growing at a rate of a few petabytes every week. The cluster stores data across a few thousand machines, each of which has a storage capacity of 24-36TB. The data stored in this cluster is immutable until it is deleted, and is compressed prior to being stored in the cluster.

Since the amount of data stored is very large, the cost of operating the cluster is dominated by the cost of the storage capacity. The most frequently accessed data is stored as 3 replicas, to allow for efficient scheduling of the map-reduce jobs. In order to save on the storage costs, the data which has not been accessed for more than three months is stored as a $(k = 10, r = 4)$ Reed-Solomon (RS) code. The cluster stores more than ten petabytes of RS-coded data. Since the employed RS code has a redundancy of only 1.4, this results in huge savings (multiple petabytes) in storage capacity as compared to 3-way replication.

3.2 Erasure coding in the warehouse cluster

We shall now delve deeper into details of the RS-coded data in the cluster. A file or a directory to be stored is first partitioned into *blocks* of size 256MB. These blocks are grouped into sets of 10 blocks each; every set is then encoded with a $(k = 10, r = 4)$ RS code to obtain 4

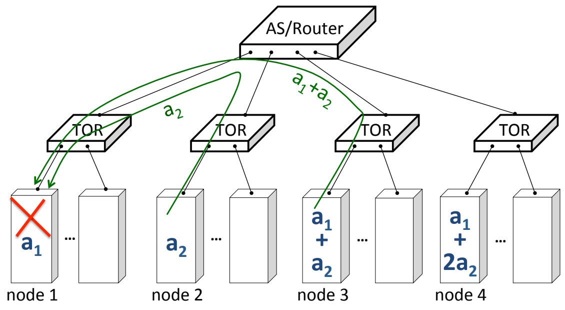


Figure 3.2: Recovery of erasure-coded data: a single missing block (a_1) is recovered by reading a block each at nodes 2 and 3 and transferring these two blocks through the top-of-rack (TOR) switches and the aggregation switch (AS).

parity blocks. As illustrated in Figure 3.1, one byte each at corresponding locations in the 10 data blocks are encoded to generate the corresponding bytes in the 4 parity blocks. The set of these 14 blocks constitutes a stripe of blocks. The 14 blocks belonging to a particular stripe are placed on 14 distinct (randomly chosen) machines. In order to secure the data against rack failures, these machines are typically chosen from distinct racks.

To recover a missing or otherwise unavailable block, any 10 of the remaining 13 blocks of its stripe are read and downloaded. Since each block is placed on a different rack, these transfers take place through the top-of-rack switches. This consumes cross-rack bandwidth that is typically heavily oversubscribed in most data centers including the one studied here. Figure 3.2 illustrates the recovery operation through an example with $(k = 2,\ r = 2)$ code. Here the first data unit a_1 (stored in server/node 1) is being recovered by downloading data from node 2 and node 3.

3.3 Measurements and observations

In this section, we present our measurements from the Facebook's data warehouse cluster, and analyze them to study the impact of recovery of RS-coded data on the network infrastructure.

Unavailability statistics

We begin with some statistics on machine unavailability events. Figure 3.3 plots the number of machines that were unavailable for more than 15 minutes in a day, over the period 22nd January to 24th February 2013 (15 minutes is the default wait time of the cluster to flag a

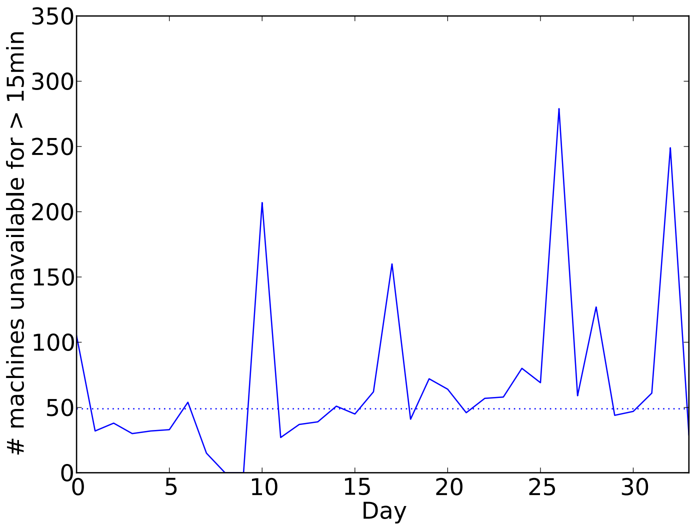


Figure 3.3: The number of machines unavailable for more than 15 minutes in a day. The dotted line represents the median value which is 52.

machine as unavailable). We observe that the median is more than 50 machine-unavailability events per day. This reasserts the necessity of redundancy in the data for both reliability and availability. A subset of these events ultimately trigger recovery operations.

Number of missing blocks in a stripe

Table 3.1 below shows the percentage of stripes with different number of blocks missing (on average). These statistics are based on data collected over a period of 6 months.

Number of missing blocks	Percentage of stripes
1	98.08 %
2	1.87 %
3	0.036 %
4	9×10^{-4} %
>5	9×10^{-6} %

Table 3.1: Percentage of stripes with different numbers of missing blocks (average of data collected over a period of six months).

We can see that one block missing in a stripe is the most dominant case: 98 % of all the

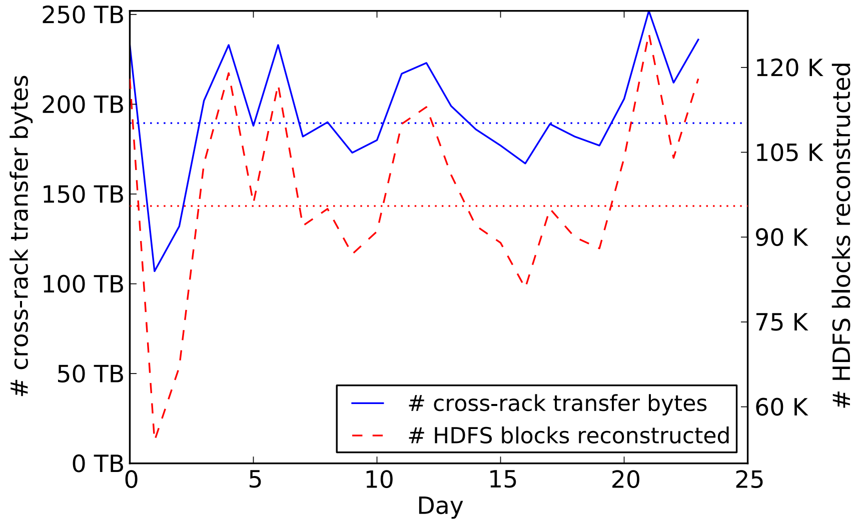


Figure 3.4: RS-coded HDFS blocks reconstructed and cross-rack bytes transferred for recovery operations per day, over a duration of around a month. The dotted lines represent the median values.

stripes with missing blocks have only one block missing. *Thus recovering from single failures is by-far the most common scenario.*

We now move on to measurements pertaining to the recovery operations for RS-coded data in the cluster. The analysis below is based on the data collected from Cluster A for the first 24 days of February 2013.

Number of block recoveries

Figure 3.4 shows the number of block recoveries triggered each day. A median of $95,500$ blocks of RS-coded data are recovered each day.

Cross-rack network transfers

We measured the number of bytes transferred across racks for the recovery of RS-coded blocks. The measurements, aggregated per day, are depicted in Figure 3.4. As shown in the figure, a median of more than 180 TB and a maximum of 250 TB of data is transferred through the top-of-rack switches every day solely for the purpose of recovering RS-coded data. The recovery operations, thus, consume a large amount of cross-rack bandwidth, thereby rendering the bandwidth unavailable for the foreground map-reduce jobs.

3.4 Summary

In this chapter, we presented our measurements and observations from Facebook's data warehouse cluster in production that stores hundreds of petabytes of data across a few thousand machines, focusing on the unavailability statistics and the impact of using erasure codes (in particular, RS codes) on the network infrastructure of a data center. In our measurements, we observed a median of more than 50 machine unavailability events per day. These unavailablility measurements corroborate the previous reports from other data centers regarding unavailabities being the norm rather than the exception. The analysis of the measurements also revealed that for the erasure-coded data, single failures in a stripe is by far the most dominant scenario. Motivated by this, we will be focusing on optimizing for reconstruction of single failure in a stripe in the following chapters. We also observed that the large amount of download performed by the RS-encoded data during reconstruction of missing blocks consumes a significantly high amount of network bandwidth and in particular, puts additional burden on the already oversubscribed top-of-rack switches. This provides a real-world motivation for our work on reconstruction-efficient erasure codes for distributed storage presented in the next few chapters.

Chapter 4

Piggybacking framework for constructing distributed storage codes

In this chapter, we present a new framework for constructing distributed storage codes, which we call the *Piggybacking* framework, that offers a rich design space for constructing storage codes optimizing for I/O and network usage while retaining the storage efficiency offered by traditional MDS codes. We also present three code constructions based on the Piggybacking framework that target different settings. In addition, we show how the piggybacking framework can be employed to enable efficient reconstruction of the parity units in existing codes that were originally designed to address the reconstruction of only the data units.

4.1 Introduction

A primary contributor to the cost of any large-scale storage system 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, we focus on codes that are MDS and have a high rate. (Recall from Chapter 2 that MDS codes are optimal in utilizing the storage space in providing reliability, and that codes with high rate have a small storage overhead factor.)

Further, recall from the measurements from Facebook's data warehouse cluster in production presented in Chapter 3, that the scenario of a single node failure in a stripe is by far the most prevalent. With this motivation, we will focus on optimizing for the case of a single failure in a stripe.

There has been considerable recent work in the area of designing reconstruction-efficient codes for distributed storage systems; these works are discussed in detail in Section 4.2. Of particular interest are the family Minimum Storage Regenerating (MSR) codes, which are MDS. While the papers [41, 182] showed the theoretical existence of MSR codes, several recent works [127, 156, 164, 115, 169, 26] have presented explicit MSR constructions for a wide range of parameters. Product-Matrix codes [127], and MISER codes [156, 164] are explicit MSR code constructions which have a number of substripes linear 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 [115, 169, 26] are high rate, but, necessarily require the number of substripes to be exponential in k. In fact, it has been shown in [168], 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: (i) A large number of substripes results in a large number of fragmented reads which is detrimental to the read-latency performance of disks, (ii) A large number of substripes, as a minor side effect, also restricts the minimum size of files that can be handled by the code. This restricts the range of file sizes that the storage system can handle. We refer the reader to Chapter 5 to see how the number of substripes manifests as practical challenges when employed in a distributed storage system. In addition to this practical motivation, there is also considerable theoretical interest in constructing reconstruction-efficient codes that are MDS, high-rate and have a smaller number of substripes [27, 61, 144]. To the best of our knowledge, the only explicit codes that meet these requirements are the Rotated-RS [89] codes and the (reconstruction-optimized) EVENODD [21, 178] and RDP [35, 183] 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 (reconstruction-optimized) EVENODD and RDP codes exist only for $r = 2$.

Furthermore, in any code, the amount of data read during a reconstruction operation is atleast as much as the amount of data downloaded, but in general, the amount of data read can be significantly higher than the amount downloaded. This is because, many codes, including the 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.

Here, we investigate the problem of constructing distributed storage codes that are efficient with respect to both the amount of data read and downloaded during reconstruction, while satisfying the constraints of (i) being MDS, (ii) having a high-rate, and (iii) having a small (constant or linear) 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.[1] 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 reconstruction 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 $r \geq 2$. The typical savings in the average amount of data read and downloaded during reconstruction 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.

(Class 2) Binary MDS codes with the lowest average amount of data read and downloaded for reconstruction Binary MDS codes are extensively used in disk arrays [21, 35]. Using the piggybacking framework, we construct binary MDS codes that, to the best of our knowledge, result in the lowest average amount of data read and downloaded for reconstruction among all existing binary MDS codes for $r \geq 3$. Our codes support all the parameters for which binary MDS codes are known to exist. The codes constructed here also optimize the reconstruction of parity nodes 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 [110, 58, 116, 84, 161] 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 $(k+1)$ [2]. To the best of our knowledge, the amount of data read and downloaded during reconstruction in the presented code is the smallest among all known explicit, exact-repair, minimum-locality, MDS codes for more than three parities (i.e., $(n-k) > 3$).

[1]Although we focus on MDS codes, it turns out that the piggybacking framework can be employed with non-MDS codes as well.

[2]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.

(Class 4) A method for reducing the amount of data read and downloaded 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 [89, 26, 155, 179, 115] 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 also retaining the efficiency of the repair of systematic nodes.

Organization

The rest of the chapter is organized as follows. A discussion on related literature is provided in Section 4.2. Section 4.3 presents two examples that highlight the key ideas behind the Piggybacking framework. Section 4.4 introduces the general piggybacking framework. Sections 4.5 and 4.6 present code designs and reconstruction algorithms based on the piggybacking framework, special cases of which result in classes 1 and 2 discussed above. Section 4.7 provides a piggyback design which enables a small repair-locality in MDS codes (Class 3). Section 4.8 provides a comparison of Piggyback codes with various other codes in the literature. Section 4.9 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).

4.2 Related work

Recall from Chapter 2 that MSR codes are MDS, and hence they are of particular relevance to this chapter. There have been several works on constructing explicit MSR codes [127, 156, 164, 26, 115, 169]. The Product-Matrix MSR codes [127] are explicit, practical MSR code constructions which have linear number of substripes. However, these codes have a low rate (i.e., high redundancy), requiring a storage overhead of $\left(2-\frac{1}{n}\right)$ or higher. In [25], the authors show the existence of high-rate MSR codes as the number of substripes approaches infinity. The MSR constructions presented in [115, 169, 26] 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 has been shown 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 [168]. In [61], the authors 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. In [27, 144], MSR codes with polynomial number of substripes are presented for the setting of constant (high) rate greater than $\frac{2}{3}$. On the other hand, 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.

In [89], the authors present a search-based approach to find reconstruction 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 reconstruction-efficient MDS code construction based on RS codes, called Rotated-RS, with the number of substripes being 2. However, it supports at most 3 parities, and moreover, its fault-tolerance capability is established via a computer search. In contrast, 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 codes have received special attention due to their extensive use in disk array systems [21, 35]. The EVEN-ODD and RDP codes have been optimized for reconstruction in [178] and [183] respectively. In [47], the authors present a binary MDS code construction for 2 parities that achieve the regenerating codes bound for repair of systematic nodes. In comparison, the repair-optimized binary MDS code constructions based on the Piggybacking framework provide as good or better savings for greater than 2 parities and also address the repair of both systematic and parity nodes.

Repair-locality, that is the number nodes contacted during the repair operation, is another metric of interest in distributed storage systems. Optimizing codes for repair-locality has been extensively studied [110, 58, 116, 84, 161] in the recent past. However, all the code constructions under this umbrella give up on the MDS property to get smaller locality than k.

Most of the related works discussed above take the conceptual approach of constructing vector codes in order to improve the efficiency of reconstruction with respect to the amount of data downloaded or both the amount of data read and downloaded. In a recent work [64], the authors present a reconstruction framework for (scalar) Reed-Solomon codes for reducing the amount of data downloaded by downloading elements from a subfield rather than the finite field over which the code is constructed. This work optimizes only the amount of data downloaded and not the amount of data read.

4.3 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 reconstruction of systematic nodes.

	An MDS Code	
Node 1	a_1	b_1
Node 2	a_2	b_2
Node 3	a_3	b_3
Node 4	a_4	b_4
Node 5	$\sum_{i=1}^4 a_i$	$\sum_{i=1}^4 b_i$
Node 6	$\sum_{i=1}^4 ia_i$	$\sum_{i=1}^4 ib_i$

(a)

	Intermediate Step	
Node 1	a_1	b_1
Node 2	a_2	b_2
Node 3	a_3	b_3
Node 4	a_4	b_4
Node 5	$\sum_{i=1}^4 a_i$	$\sum_{i=1}^4 b_i$
Node 6	$\sum_{i=1}^4 ia_i$	$\sum_{i=1}^4 ib_i + \sum_{i=1}^2 ia_i$

(b)

	Piggybacked Code	
Node 1	a_1	b_1
Node 2	a_2	b_2
Node 3	a_3	b_3
Node 4	a_4	b_4
Node 5	$\sum_{i=1}^4 a_i$	$\sum_{i=1}^4 b_i$
Node 6	$\sum_{i=3}^4 ia_i - \sum_{i=1}^4 ib_i$	$\sum_{i=1}^4 ib_i + \sum_{i=1}^2 ia_i$

(c)

Figure 4.1: An example illustrating efficient repair of systematic nodes using the piggybacking framework. Two instances of a $(n=6, k=4)$ MDS code are piggybacked to obtain a new $(n=6, k=4)$ 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.

Example 1. *Consider two instances of a* $(n = 6, k = 4)$ *MDS code as shown in Fig. 4.1a, with the* 8 *message symbols* $\{a_i\}_{i=1}^4$ *and* $\{b_i\}_{i=1}^4$ *(each column of Fig. 4.1a depicts a single instance of the code). One can verify that the message can be recovered from the data of any* 4 *nodes. The first step of piggybacking involves adding* $\sum_{i=1}^2 ia_i$ *to the second symbol of node* 6 *as shown in Fig. 4.1b. The second step in this construction involves subtracting the second symbol of node* 6 *in the code of Fig. 4.1b from its first symbol. The resulting code is shown in Fig. 4.1c. This code has* 2 *substripes (the number of columns in Fig. 4.1c).*

We now present the repair algorithm for the piggybacked code of Fig. 4.1c. Consider the repair of node 1*. Under our repair algorithm, the symbols* b_2, b_3, b_4 *and* $\sum_{i=1}^4 b_i$ *are download from the other nodes, and* b_1 *is decoded. In addition, the second symbol* $(\sum_{i=1}^4 ib_i + \sum_{i=1}^2 ia_i)$ *of node* 6 *is downloaded. Subtracting out the components of* $\{b_i\}_{i=1}^4$ *gives the piggyback* $\sum_{i=1}^2 ia_i$*. Finally, the symbol* a_2 *is downloaded from node* 2 *and subtracted to obtain* a_1*. Thus, node* 1 *is repaired by reading only* 6 *symbols which is* 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.*

The piggybacked code is MDS, and the entire message can be recovered from any 4 *nodes as follows. If node* 6 *is one of these four nodes, then add its second symbol to its first, to recover the code of Fig. 4.1b. Now, the decoding algorithm of the original code of Fig, 4.1a is employed to first recover* $\{a_i\}_{i=1}^4$*, which then allows for removal of the piggyback* $(\sum_{i=1}^2 ia_i)$ *from the second substripe, making the remainder identical to the code of Fig. 4.1c.*

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	$\sum_{i=3}^{4} ia_i - \sum_{i=1}^{4} ib_i$	$\sum_{i=1}^{4} ib_i + \sum_{i=1}^{2} ia_i$	$\sum_{i=3}^{4} ic_i - \sum_{i=1}^{4} id_i$	$\sum_{i=1}^{4} id_i + \sum_{i=1}^{2} ic_i$

Figure 4.2: An example illustrating the mechanism of repair of parity nodes under the piggybacking framework, using two instances of the code in Fig. 4.1c. A shaded cell indicates a modified symbol.

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

Example 2. *The code depicted in Fig. 4.2 takes two instances of the code in Fig. 4.1c, and adds the second symbol of node* 6, $(\sum_{i=1}^{4} ib_i + \sum_{i=1}^{2} ia_i)$ *(which belongs to the first instance), to the third symbol of node* 5 *(which belongs to the second instance). This code has* 4 *substripes (the number of columns in Fig. 4.2). In this code, repair of the second parity node involves downloading* $\{a_i, c_i, d_i\}_{i=1}^{4}$ *and the modified symbol* $(\sum_{i=1}^{4} c_i + \sum_{i=1}^{4} ib_i + \sum_{i=1}^{2} ia_i)$, *using which the data of node* 6 *can be recovered. The repair of the second parity node thus requires read and download of only* 13 *symbols instead of the entire message of size* 16*. The first parity is repaired by downloading all* 16 *message symbols. Observe that in the code of Fig. 4.1c, the first symbol of node* 5 *is never used for the repair of any of the systematic nodes. Thus the modification in Fig. 4.2 does not change the algorithm or the efficiency of the repair of systematic nodes. The code retains 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. 4.1c, 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. 4.1c.*

4.4 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

$\{f_i\}_{i=1}^n$: each of which takes a message $\mathbf{u}$ as input and encodes it to n coded symbols $\{f_1(\mathbf{u}), \ldots, f_n(\mathbf{u})\}$. Node i ($1 \leq i \leq n$) stores the data $f_i(\mathbf{u})$.

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

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

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 thus:

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

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 ($2 \leq i \leq n$), 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 chapter).

The piggybacking procedure described above was followed in Example 1 to obtain the code of Fig. 4.1b from that in Fig. 4.1a. Subsequently, in Example 2, this procedure was followed again to obtain the code of Fig. 4.2 from that in Fig. 4.1c.

The piggybacking framework also allows 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. 4.1b) 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. 4.1c is obtained from Fig. 4.1b via an invertible transformation of the data of node 6.

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

Theorem 1. *Let $U_1, \ldots, U_\alpha$ 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 the random variable corresponding to the (encoded) data stored in node i under the piggybacked version of that code. Then for any subset of nodes $S \subseteq \{1, \ldots, n\}$,*

$$I\left(\{Y_i\}_{i\in S}; U_1, \ldots, U_\alpha\right) \geq I\left(\{X_i\}_{i\in S}; U_1, \ldots, U_\alpha\right) \ . \tag{4.1}$$

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

$$\begin{aligned}
I\left(\{Y_i\}_{i\in S}\,;\,U_1,\ldots,U_\alpha\right) &= I\left(\{\mathbf{f}(U_j)+\mathbf{g}_j(U_1,\ldots,U_{j-1})\}_{j=1}^{\alpha}\;;\;U_1,\ldots,U_\alpha\right)\\
&= \sum_{\ell=1}^{\alpha} I\left(\{\mathbf{f}(U_j)+\mathbf{g}_j(U_1,\ldots,U_{j-1})\}_{j=1}^{\alpha}\,;\,U_\ell\;\middle|\;U_1,\ldots,U_{\ell-1}\right)\\
&= \sum_{\ell=1}^{\alpha} I\left(\mathbf{f}(U_\ell)\;,\;\{\mathbf{f}(U_j)+\mathbf{g}_j(U_1,\ldots,U_{j-1})\}_{j=\ell+1}^{\alpha}\,;\,U_\ell\;\middle|\;U_1,\ldots,U_{\ell-1}\right)\\
&\geq \sum_{\ell=1}^{\alpha} I\left(\mathbf{f}(U_\ell)\;;\;U_\ell\mid U_1,\ldots,U_{\ell-1}\right)\\
&= \sum_{\ell=1}^{\alpha} I\left(\mathbf{f}(U_\ell)\;;\;U_\ell\right) \qquad (4.2)\\
&= I\left(\{X_i\}_{i\in S}\,;U_1,\ldots,U_\alpha\right)\;. \qquad (4.3)
\end{aligned}$$

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

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

Proof. Directly follows from Theorem 1. □

Notational Conventions For simplicity of exposition, we shall assume throughout this section that the base codes are linear, scalar, MDS and systematic. Using vector codes (such as EVENODD or RDP) as base codes is a straightforward extension. The base code operates on a k-length message vector, with each symbol of this vector drawn from some finite field. The number of instances of the base code during piggybacking is denoted by 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 $\{\mathbf{p}_1^T\mathbf{a},\ldots,\mathbf{p}_r^T\mathbf{a}\}$.

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 code constructions presented here possesses the property that the repair of any node entails reading of only as much data as what has to be downloaded. [3] This property is called *repair-by-transfer* [149]. 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.

4.5 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. We will first present the piggyback design for optimizing the repair of systematic nodes, and then move on to the repair of parity nodes.

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 $\frac{k}{r}$. 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 $i = 1, \ldots, r$.

Define the $k-$length vectors $\mathbf{q}_2, \ldots, \mathbf{q}_{r+1}, \mathbf{v}_r$ to be the following projections of vector $\mathbf{p}_r$:

[3] 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.

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

Also, let

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

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

The base code is piggybacked in the following manner:

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

Fig. 4.1b depicts an example of such a piggybacking.

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

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

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 $\{a_i\}_{i\in S_1\backslash\{\ell\}}$ 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 $(k + \frac{k}{r})$ (in comparison, the size of the message is $2k$).

Case 2 ($S = S_r$): As in the previous case, $\mathbf{b}$ is completely decoded by downloading $\{b_1, \ldots, b_{\ell-1}, b_{\ell+1}, \ldots, b_k,\ \mathbf{p}_1^T\mathbf{b}\}$. 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 $\mathbf{b}$, $\mathbf{v}^T\mathbf{b}$ 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 $(k + \frac{k}{r} + r - 2)$.

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 described previously), 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

$$t_1 = \cdots = t_{r-1} = \left\lceil \frac{k}{r} + \frac{r-2}{2r} \right\rceil := t, \tag{4.4}$$

$$t_r = k - (r-1)t\ . \tag{4.5}$$

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

$$\gamma_1^{\text{sys}} = \frac{1}{2k^2}\left[(k - t_r)(k + t) + t_r(k + t_r + r - 2)\right]\ . \tag{4.6}$$

This quantity is plotted in Fig. 4.5a for several values of the system parameters n and k.

Repair of parity nodes

We shall now piggyback the code constructed in Section 4.5 to introduce efficiency in the repair of parity nodes, while also retaining the efficiency in the repair of systematic nodes. Observe that in the code of Section 4.5, 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 4.5. 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 below.

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

This completes the encoding procedure. The code of Fig. 4.2 is an example of this design.

As shown in Section 4.4, the piggybacked code retains the MDS property of the base code. In addition, the repair of systematic nodes is identical to the that in the code of Section 4.5, since the symbol modified in this piggybacking was never read for the repair of any systematic node in the code of Section 4.5.

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 above) 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.

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

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

This quantity is plotted in Fig. 4.5b for several values of the system parameters n and k with $m = 4$.

4.6 Piggybacking design 2

The design presented in this section provides a higher efficiency of repair as compared to the previous design. On the downside, it requires a larger number of substripes: the minimum number of substripes required under the design of Section 4.5 is 2 and under that of Section 4.5 is 4, while that required in the design of this section is $(2r-3)$. We first illustrate this piggybacking design with an example.

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*

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

Now piggyback the base code in the following manner

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

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

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

This completes the encoding procedure.

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

The general design is as follows. 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 of the $\frac{k}{r-1}$ sets consist of the first $\frac{k}{r-1}$ nodes, the next set consists of the next $\frac{k}{r-1}$ nodes and so on. Define k-length vectors $\{\mathbf{v}_i,\ \hat{\mathbf{v}}_i\}_{i=2}^{r}$ as

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

Further, define k-length vectors $\{\mathbf{q}_{i,j}\}_{i=2,j=1}^{r,r-1}$ as

$$\mathbf{q}_{i,j} = \begin{bmatrix} 0 & & & & & \\ & \ddots & & & & \\ & & 0 & & & \\ & & & 1 & & \\ & & & & \ddots & \\ & & & & & 1 \\ & & & & & & 0 \\ & & & & & & & \ddots \\ & & & & & & & & 0 \end{bmatrix} \mathbf{p}_i$$

where the positions of the ones on the diagonal of the $(k \times k)$ diagonal matrix depicted correspond to the nodes in the j^{th} group. It follows that

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

Parity node $(k+i)$, $i \in \{2, \ldots, r\}$, is then piggybacked to store

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

Following this, an invertible linear combination is performed at 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 the node $(k+i)$, $i \in \{2, \ldots, r\}$, stores

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

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.

When k is not a multiple of $(r-1)$, the k systematic nodes are divided into $(r-1)$ sets as follows. Let

$$t_\ell = \left\lfloor \frac{k}{r-1} \right\rfloor , \qquad t_h = \left\lceil \frac{k}{r-1} \right\rceil , \qquad t = (k-(r-1)t_\ell) \ . \tag{4.7}$$

The first t sets are chosen of size t_h each and the remaining $(r-1-t)$ sets have size t_ℓ each. The systematic symbols in the first $(r-1)$ substripes are piggybacked onto the parity symbols (except the first parity) of the last $(r-1)$ stripes. For repair of any failed systematic node $\ell \in \{1,\ldots,k\}$, the last $(r-2)$ substripes are decoded completely by reading the remaining systematic and the first parity symbols from each. To obtain the remaining $(r-1)$ symbols of the failed node, the $(r-1)$ parity symbols that have piggyback vectors (i.e., $\mathbf{q}$'s and $\mathbf{v}$'s) with a non-zero value of the ℓ^{th} element are downloaded. By design, these piggyback vectors have non-zero components only along the systematic nodes in the same set as node ℓ. Downloading and subtracting these other systematic symbols gives the desired data.

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

$$\gamma_2^{\text{sys}} \quad = \quad \frac{1}{(2r-3)k^2}\left[t((r-2)k+(r-1)t_h)+(k-t)((r-2)k+(r-1)t_\ell)\right] \ . \tag{4.8}$$

This quantity is plotted in Fig. 4.5a for several values of the system parameters n and k.

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 4.5. 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 4.5, 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 4.5, higher a value of m results in a lesser amount of data read and downloaded for repair. In such a design, the average data read γ_2^{par} for repair of parity nodes, as a fraction of the total message symbols, is

$$\gamma_2^{\text{par}} = \frac{1}{r} + \frac{r-1}{(2r-3)kmr}\left[(m+1)(r-2)k + (m-1)(r-2)(r-1) + \left\lceil\frac{m}{2}\right\rceil k + \left\lfloor\frac{m}{2}\right\rfloor (r-1)\right] . \tag{4.9}$$

This quantity is also plotted in Fig. 4.5b for various values of the system parameters n and k with $m = 4$.

4.7 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 $(k+1)$. [4] To the best of our knowledge, the amount of data read and downloaded during reconstruction in the presented code is the smallest among all known explicit, exact-repair, minimum-locality, MDS codes for more than three parities (i.e., $(r > 3)$.

This design involves two levels of piggybacking, and these are illustrated in the following two example constructions. The first example considers $\alpha = 2m$ instances of the base code and shows the first level of piggybacking, for any positive integer $m > 1$ (as we will see, higher values of m will result in repair with a smaller amount of data read and download). The second example uses two instances of this code and adds the second level of piggybacking. We note that this design deals with the repair of only the systematic nodes.

Example 4. *Consider any* $(n = 11,\ k = 8)$ *MDS code as the base code, and take* 4 *instances of this code. Divide the systematic nodes into two sets as follows,* $S_1 = \{1, 2, 3, 4\}$, $S_2 = \{5, 6, 7, 8\}$. *We then add the piggybacks as shown in Fig. 4.3. Observe that in this design, the piggybacks added to an even substripe is a function of symbols in its immediately previous (odd) substripe from only the systematic nodes in the first set* S_1*, while the piggybacks added to an odd substripe are functions of symbols in its immediately previous (even) substripe from only the systematic nodes in the second set* S_2.

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

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

Figure 4.3: Example illustrating first level of piggybacking in design 3. The piggybacks in the even substripes (in blue) are a function of only the systematic nodes $\{1, \dots, 4\}$ (also in blue), and the piggybacks in odd substripes (in green) are a function of only the systematic nodes $\{5, \dots, 8\}$ (also in green). This code requires an average 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 $\ell \in \{1, \dots, 4\}$ in the first set. For instance, say $\ell = 1$, then $\{b_2, \dots, b_8,\ \mathbf{p}_1^T\mathbf{b}\}$ and $\{d_2, \dots, d_8,\ \mathbf{p}_1^T\mathbf{d}\}$ are downloaded, and $\{\mathbf{b},\ \mathbf{d}\}$ (i.e., the messages in the even substripes) are decoded. It now remains to recover the symbols a_1 and c_1 (belonging to the odd substripes). The second symbol $(\mathbf{p}_2^T\mathbf{b} + a_1 + a_2)$ from node 10 is downloaded and $\mathbf{p}_2^T\mathbf{b}$ subtracted out to obtain the piggyback $(a_1 + a_2)$. Now a_1 can be recovered by downloading and subtracting out a_2. The fourth symbol from node 10, $(\mathbf{p}_2^T\mathbf{d} + c_1 + c_2)$, is also downloaded and $\mathbf{p}_2^T\mathbf{d}$ subtracted out to obtain the piggyback $(c_1 + c_2)$. Finally, c_1 is recovered by downloading and subtracting out c_2. Thus, node 1 is repaired by reading a total of 20 symbols (in comparison, the total total message size is 32) and with a locality of 9. The repair of node 2 can be carried out in an identical manner. The two other nodes in the first set, nodes 3 and 4, can be repaired in a similar manner by reading the second and fourth symbols of node 11 which have their piggybacks. Thus, repair of any node in the first group requires reading and downloading a total of 20 symbols and with a locality of 9.

Now we consider the repair of any node $\ell \in \{5, \dots, 8\}$ in the second set S_2. For instance, consider $\ell = 5$. The symbols $\{a_1, \dots, a_8,\ \mathbf{p}_1^T\mathbf{a}\} \setminus \{a_5\}$, $\{c_1, \dots, c_8,\ \mathbf{p}_1^T\mathbf{c}\} \setminus \{c_5\}$ and $\{d_1, \dots, d_8,\ \mathbf{p}_1^T\mathbf{d}\} \setminus \{d_5\}$ are downloaded in order to decode a_5, c_5, and d_5. From node 10, the symbol $(\mathbf{p}_2^T\mathbf{c} + b_5 + b_6)$ is downloaded and $\mathbf{p}_2^T\mathbf{c}$ is subtracted out. Then, b_5 is recovered *by downloading and subtracting out* b_6. *Thus, node* 5 *is recovered by reading a total of* 26 *symbols and with a locality 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. A higher value of* m *(i.e., a higher number of substripes) would lead to a further reduction in the read and download (the last substripe cannot be piggybacked and hence mandates a greater read and download; this is a boundary case, and its contribution to the overall read reduces with an increase in* m*).*

Example 5. *In this example, we illustrate the second level of piggybacking which further reduces the amount of data read during repair of systematic nodes as compared to Example 4. Consider* $\alpha = 8$ *instances of an* $(n = 13,\ k = 10)$ *MDS code. Partition the systematic nodes into three sets* $S_1 = \{1, \ldots,\ 4\}$, $S_2 = \{5, \ldots, 8\}$, $S_3 = \{9,\ 10\}$ *(for readers having access to Fig. 4.4 in color, these nodes are coloured blue, green, and red respectively). We first add piggybacks of the data of the first* 8 *nodes onto the parity nodes* 12 *and* 13 *on similar lines as in Example 4 (see Fig. 4.4). We now add piggybacks for the symbols stored in systematic nodes in the third set, i.e., nodes* 9 *and* 10. *To this end, we parititon the* 8 *substripes into two groups of size four each (indicated by white and gray shades respectively in Fig. 4.4). The symbols of nodes* 9 *and* 10 *in the first four substripes are piggybacked onto the last four substripes of the first parity node, as shown in Fig. 4.4 (in red color).*

We now present the algorithm for repair of systematic nodes under this piggyback code. The repair algorithm for the systematic nodes $\{1, \ldots, 8\}$ *in the first two sets closely follows the repair algorithm illustrated in Example 4. Suppose* $\ell \in S_1$, *say* $\ell = 1$. *By construction, the piggybacks corresponding to the nodes in* S_1 *are present in the parities of even substripes. From the even substripes, the remaining systematic symbols,* $\{b_i, d_i, f_i, h_i\}_{i=\{2,\ldots,10\}}$, *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. Again, using the MDS property of the base code, one recovers* $\mathbf{f}$ *and* $\mathbf{h}$. *It now remains to recover* $\{a_1,\ c_1,\ e_1,\ g_1\}$. *To this end, we download the symbols in the even substripes of node* 12, $\{\mathbf{p}_2^T\mathbf{b}{+}a_1 + a_2,\ \mathbf{p}_2^T\mathbf{d}{+}c_1 + c_2,\ \mathbf{p}_2^T\mathbf{f}{+}e_1 + e_2,\ \mathbf{p}_2^T\mathbf{h}{+}g_1 + g_2\}$, *which have piggybacks with the desired symbols. By subtracting out previously downloaded data, we obtain the piggybacks* $\{a_1 + a_2, c_1 + c_2, e_1 + e_2, g_1 + g_2\}$. *Finally, by downloading and subtracting* a_2, c_2, e_2, g_2, *we recover* a_1, c_1, e_1, g_1. *Thus, node* 1 *is recovered by reading* 48 *symbols, which is* 60% *of the total message size. Observe that the repair of node* 1 *was accomplished 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 corre-*

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

Figure 4.4: An example illustrating piggyback design 3, with $k = 10$, $n = 13$, $\alpha = 8$. The piggybacks in the first parity node (in red) are functions of the data of nodes $\{8,\ 9\}$ alone. In the remaining parity nodes, the piggybacks in the even substripes (in blue) are functions of the data of nodes $\{1, \ldots, 4\}$ (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 4.3). The piggybacks in node 11 piggyback the first set of 4 substripes (white background) onto the second set of number of substripes (gray background)

.

sponding piggybacks, however, the total number of symbols read is 64 *(since the last substripe cannot be piggybacked; such was the case in Example 4 as well).*

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

Code	k, r supported	Number of substripes
Rotated RS [89]	$r \in \{2,3\}, k \leq 36$	2
EVENODD, RDP [21, 35, 178, 183]	$r = 2$	k
Piggyback 1	$r \geq 2$	$2m$
Piggyback 2	$r \geq 3$	$(2r-3)m$
Piggyback 3	$r \geq 2$	$4m$

Table 4.1: Summary of supported parameters for explicit code constructions under Class 1, which are codes that are MDS, have high-rate, and have a small (constant or linear in k) number of substripes. For the Piggyback codes, the value of m can be chosen to be any positive integer.

recover the desired data $\{a_9, b_9, c_9, d_9\}$. *Thus, node* 9 *is recovered by reading and downloading* 48 *symbols. Observe that the locality of repair is* $(k+1) = 11$. *Node* 10 *is repaired in a similar manner.*

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

$$t_1 = \left\lceil \frac{1}{2r-1} \right\rceil \ , \quad t_2 = \left\lceil \frac{r-1}{2r-1} \right\rceil \ , \quad t_3 = \left\lfloor \frac{r-1}{2r-1} \right\rfloor \ . \tag{4.10}$$

Then the average amount of data read and downloaded γ_3^{sys} for repair of systematic nodes, as a fraction of the total message symbols $4mk$, is

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

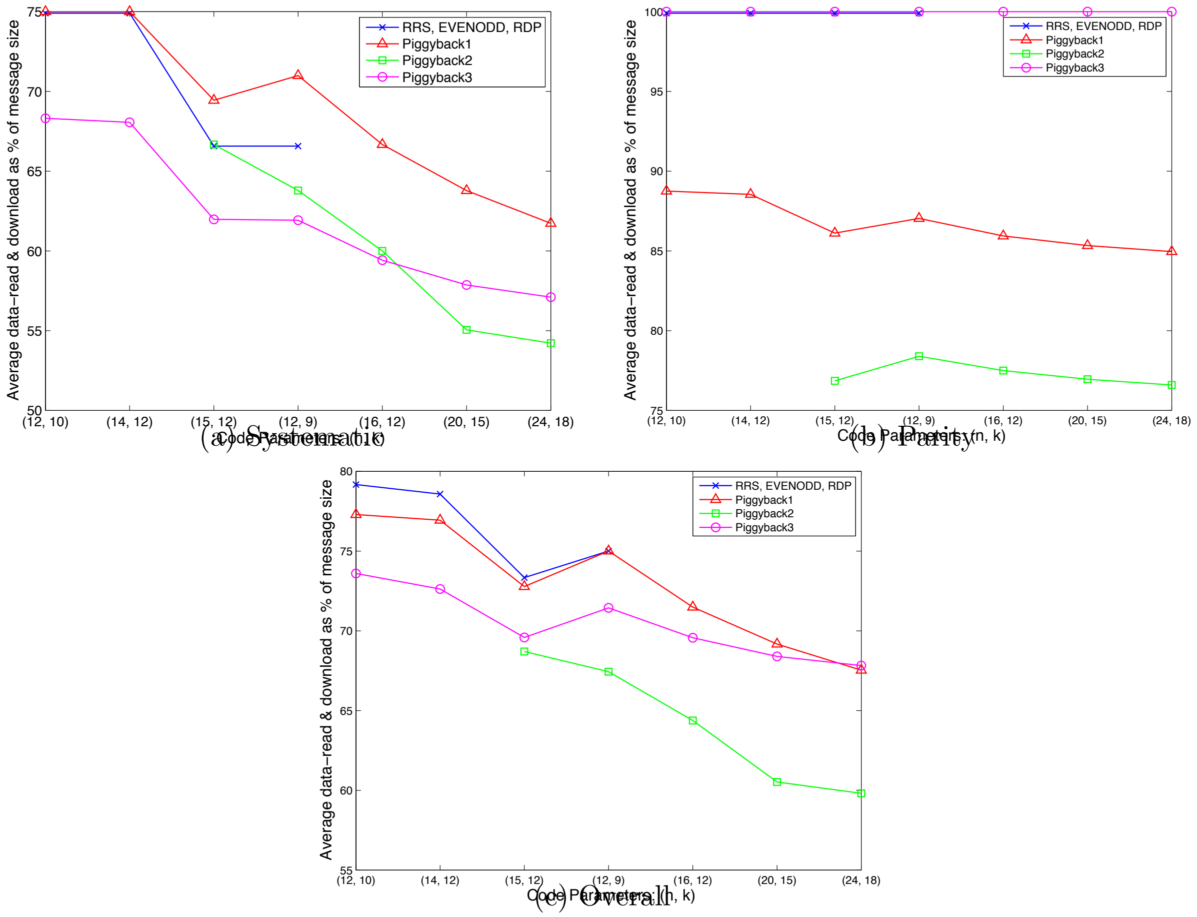
This quantity is plotted in Fig. 4.5a for several values of the system parameters n and k with $m = 16$.

4.8 Comparative study

We now compare the average amount of data read and downloaded during repair under the piggybacking constructions with existing explicit constructions in the 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 4.1). A summary of the supported parameters of various existing explicit constructions for Class 1 and the piggybacking constructions is provided in Table 4.1. Fig. 4.5 presents plots comparing the amount of data read and downloaded during repair under these codes. Each of the codes compared in Fig. 4.5 have the property of *repair-by-transfer* [149], i.e., they read only those symbols that they wish to download. Consequently, the amount of data read and downloaded are equal under these codes. The average amount of data read and downloaded during reconstruction of systematic nodes is compared in Fig. 4.5a, that during reconstruction of parity nodes in Fig. 4.5b, and the average taking all nodes into account is compared in Fig. 4.5c. Although the goal of 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, $4(2r-3)$ for Piggyback 2, and 16 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 $r=3$. 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 codes, we propose using Piggybacking designs 1 and 2 with traditional binary MDS array codes as the underlying base codes (for instance, [22]). This provides repair-efficient, binary, MDS codes for all parameters where traditional binary MDS array codes exist. The (repair-optimized) EVENODD and RDP codes [183, 178] and the code presented in [47] are binary MDS codes and each of these codes exists only for $r=2$. Furthermore, (repair-optimized) EVENODD and the construction in [47] consider reconstruction 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. 4.5. For $r=2$, considering only the reconstruction of systematic nodes, the construction in [47] achieves the minimum amount of data read and download that of 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 4.7. Class 4, which corresponds to optimizing reconstruction of parity nodes in codes that optimize reconstruction of only the systematic nodes will be considered in the following section.



(a) Systematic (b) Parity

(c) Overall

Figure 4.5: 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 [89], and (repair-optimized) EVENODD and RDP codes [183, 178]. 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.

4.9 Repairing parities in existing codes that address only systematic repair

Several codes proposed in the literature [89, 26, 155, 115] can efficiently repair only the systematic nodes, and require the download of the entire message for repair of any parity node. In this section, we piggyback these codes to reduce the read and download during repair of parity nodes, while also retaining the efficiency of repair of systematic nodes. This piggybacking design is first illustrated with the help of an example.

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)

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

(b)

Table 4.2: 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: (a) An existing code [155] originally designed to address repair of only systematic nodes; (b) Piggybacking to also optimize repair of parity nodes. See Example 6 for more details.

Example 6. *Consider the code depicted in Fig. 4.2a, originally proposed in [155]. This is an MDS code with parameters ($n = 4,\ k = 2$), and the message comprises four symbols a_1, a_2, b_1 and b_2 over finite field $\mathbb{F}_5$. The code can repair any systematic node with an optimal data read and download. Node 1 is repaired by reading and downloading the symbols a_2, $(3a_1 + 2b_1 + a_2)$ and $(3a_1 + 4b_1 + 2a_2)$ from nodes 2, 3 and 4 respectively; node 2 is repaired by reading and downloading the symbols b_1, $(b_1 + 2a_2 + 3b_2)$ and $(b_1 + 2a_2 + b_2)$ from nodes 1, 3 and 4 respectively. The amount of data read and downloaded in these two cases are the minimum possible. However, under this code, the repair of parity nodes with reduced data read has not been addressed.*

In this example, we piggyback the code of Fig. 4.2a 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. 4.2b. This code is obtained by piggybacking on the first parity symbol of the last two instance, as shown in Fig. 4.2b. In this piggybacked code, repair of systematic nodes follow the same algorithm as in the base code, i.e., repair of node 1 is accomplished by downloading the first and third symbols of the remaining three nodes, while the repair of node 2 is performed by downloading the second and fourth symbols of the remaining nodes. One can easily verify that the data obtained in each of these two cases are equivalent to what would have been obtained in the code of Fig. 4.2a in the absence of piggybacking. Thus the repair of the systematic nodes remains optimal. Now consider repair of the second parity node, i.e., node 4. The code (Fig. 4.2a), as proposed in [155], would require reading 8 symbols (which is the size of the entire message) for this repair. However, the piggybacked version of Fig. 4.2b 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. 4.2a carries over to Fig. 4.2b as discussed in Section 4.4.*

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 for some $(k\mu \times \mu)$ matrix P_j. Fig. 4.6a 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*

$$RQ_x^{(i)} = Q_y^{(i)} S \qquad \forall\ i \in \{1, \ldots, k\}\ . \tag{4.11}$$

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 (4.11) 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. 4.6b.

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

(a) Two instances of the vector base code.

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

(b) Illustrating the piggybacking design of Proposition 1. Parities $(k+1)$ and $(k+2)$ respectively correspond to $(k+x)$ and $(k+y)$ of the proposition.

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

(c) Piggybacking the regenerating code constructions of [26, 155, 179, 115] for efficient parity repair.

Figure 4.6: Piggybacking for efficient parity-repair in existing codes originally constructed for repair of only systematic nodes.

Under a piggybacking as described in the lemma, the repair of parity node $(k+y)$ can be made more efficient by exploiting the fact that the parity node $(k+x)$ now stores the piggybacked symbol $(\mathbf{b}^T P_x + \mathbf{a}^T P_y R)$. We now demonstrate the use of this design by making the repair of parity nodes efficient in the explicit MDS 'regenerating code' constructions of [26, 155, 179, 115] which address the repair of only the systematic nodes. These codes have the property that

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

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

Example 7. *The piggybacking of (two instances) of any such code [26, 155, 179, 115] is shown in Fig. 4.6c (for the case $r = 3$). As discussed previously, the MDS property and the property of efficient repair of systematic nodes is retained upon piggybacking. The repair of parity node $(k+1)$ in this example is carried out by downloading all the $2k\mu$ symbols. On the other hand, repair of node $(k+2)$ is accomplished by reading and downloading $\mathbf{b}$ from the systematic nodes, $(\mathbf{b}^T P_1 + \mathbf{a}^T P_2 + \mathbf{a}^T P_3)$ from the first parity node, and $(\mathbf{a}^T P_3)$ from the third parity node. This gives the two desired symbols $\mathbf{a}^T P_2$ and $\mathbf{b}^T P_2$. Repair of the third parity is performed in an identical manner, except that $\mathbf{a}^T P_2$ is downloaded from the second parity node. The average amount of data read and downloaded for the repair of parity nodes, as a fraction of the size $k\mu$ of the message, is thus*

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

which translates to a saving of around 33%.

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

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

sets of equal sizes (or nearly equal sizes if r is not a multiple of g). Within each set, the encoding procedure of Fig. 4.6c is performed separately. The first parity in each group is repaired by downloading all the data from the systematic nodes. On the other hand, as in Example 7, the repair of any other parity node is performed by reading $\mathbf{b}$ from the systematic nodes, the second (which is piggybacked) symbol of the first parity node of the set, and the first symbols of all other parity nodes in the set. Assuming the g sets have equal number of nodes (i.e., ignoring rounding effects), the average amount of data read and downloaded for the repair of parity nodes, as a fraction of the size $k\mu$ of the message, is

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

4.10 Summary

In this chapter, we presented the Piggybacking framework for designing storage codes that are efficient in both the amount of data read and downloaded during reconstruction, while being MDS, high-rate, and having a small number of substripes and a small field size. Under this setting, to the best of our knowledge, Piggyback codes achieve the minimum amount of data read and downloaded for reconstruction among all existing 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 reconstruction among all existing solutions for two important settings in addition to the one mentioned above. Furthermore, we showed 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.

In the following chapter, we present the design, implementation, and evaluation of an erasure-coded storage system, which we call *Hitchhiker*, that employs Piggyback codes.

Chapter 5

Hitchhiker: a resource-efficient erasure coded storage system

In this chapter, we change gears from theory to systems and present *Hitchhiker*, a resource-efficient erasure-coded storage system that reduces the usage of both I/O and network during reconstruction by 25% to 45% without requiring any additional storage and maintaining the same level of fault-tolerance as RS-based systems. Hitchhiker employs the Piggybacking framework presented in Chapter 4, and uses a novel disk layout (or data placement) technique to translate the gains promised in theory to gains in real-world systems. Hitchhiker has received considerable attention from the industry, and is being incorporated into the next release of Apache Hadoop Distributed File system (Apache HDFS).

5.1 Introduction

As discussed in the previous chapters, erasure coding offers a storage-efficient alternative for introducing redundancy as compared to replication. For this reason, several large-scale distributed storage systems [57, 66] now deploy erasure codes, that provide higher reliability at significantly lower storage overheads, with the most popular choice being the family of Reed-Solomon (RS) codes [140].

We posit that the primary reason that makes RS codes particularly attractive for large-scale distributed storage systems is its two following properties:
P1: Storage optimality. A primary contributor to the cost of any large-scale storage system 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. Recall from Chapter 2 that codes MDS codes are optimal in utilizing the storage space in providing reliability. RS codes are MDS, and hence a $(k,\ r)$ RS code entails the minimum storage overhead among all $(k,\ r)$ erasure codes that tolerate any r failures.
P2: Generic applicability. RS codes can be constructed for arbitrary values of the parameters $(k,\ r)$, thus allowing complete flexibility in the design of the system.

As we have seen in the previous chapters, although RS codes improve storage efficiency, they cause a *significant increase in the disk and network traffic*. In addition to the increased toll on network and disk resources, the significant increase in the amount of data to be read and downloaded during reconstruction affects RS-based storage systems in two ways. First, it *drastically hampers the read performance of the system in "degraded mode"*, i.e., when there is a read request for a data unit that is missing or unavailable. Serving such a request is called a 'degraded read'. In a replication based system, degraded reads can be performed very efficiently by serving it from one of the replicas of the requisite data. On the other hand, the high amount of data read and downloaded as well as the computational load for reconstructing any data block in an RS-based system *increases the latency of degraded reads*. Second, it *increases the recovery time of the system*: recovering a failed machine or decommissioning a machine takes significantly longer time than in a replication-based system. Based on conversations with teams from multiple enterprises that deploy RS codes in their storage systems, we gathered that *this increased disk and network traffic and its impact on degraded reads and recovery is indeed a major concern*, and is one of the bottlenecks to erasure coding becoming more pervasive in large-scale distributed storage systems.

The problem of reducing the amount of data required to be downloaded for reconstruction in erasure-coded systems has received considerable attention in the recent past both in the theory and the systems literature [149, 115, 169, 129, 146, 73, 127, 49, 150, 178, 183, 81, 85, 72]. However, all existing practical solutions either demand additional storage space [149, 146, 73, 49, 127, 150]), or are applicable in very limited settings [89, 178, 183, 72]. For example, [73, 146, 49] add at least 25% to 50% more parity units to the code, thereby increasing the storage overheads, [127, 150] necessitate a high redundancy of $r \geq (k-1)$ in the system, while [89, 178, 183, 72] operate in a limited setting allowing only two or three parity units.

In this chapter, we present *Hitchhiker*, an erasure-coded storage system that bridges this gap between theory and practice. Hitchhiker *reduces both network and disk traffic during reconstruction by* 25% *to* 45% *without requiring any additional storage and maintaining the same level of fault-tolerance as RS-based systems.*[1] Furthermore, Hitchhiker can be used with any choice of the system parameters k and r, thus retaining both the attractive prop-

[1]It takes a free-ride on top of the RS-based system, retaining all its desired properties, and hence the name 'Hitchhiker'.

Storage optimality and generic applicability:

Storage requirement	Same (optimal)
Supported parameters	All
Fault tolerance	Same (optimal)

Data reconstruction:

Data downloaded (i.e., network traffic)	35% less
Data read (i.e., disk traffic)	35% less
Data read time (median)	31.8% less
Data read time (95th percentile)	30.2% less
Computation time (median)	36.1% less

Encoding:

Encoding time (median)	72.1% more

Table 5.1: Performance of *Hitchhiker* as compared to Reed-Solomon-based system for default HDFS parameters.

erties of RS codes described earlier. Hitchhiker accomplishes this with the aid of two novel components: (i) a new encoding and decoding technique that builds on top of RS codes and reduces the amount of download required during reconstruction of missing or otherwise unavailable data, (ii) a novel disk layout technique that ensures that the savings in network traffic offered by the new code is translated to savings in disk traffic as well.

In proposing the new storage code, we make use of the 'Piggybacking framework' presented in Chapter 4. Specifically, we employ the Piggybacking design 1 to construct an erasure code that reduces the amount of data required during reconstruction while maintaining the storage optimality and generic applicability of RS codes. Our construction offers the choice of either using it with only XOR operations resulting in significantly faster computations or with finite field arithmetic for a greater reduction in the disk and network traffic during reconstruction. Interestingly, we also show that the XOR-only design can match the savings of non-XOR design if the underlying RS code satisfies a certain simple condition. The proposed codes also help reduce the computation time during reconstruction as compared to the existing RS codes.

The proposed storage codes reduce the amount of download required for data reconstruction, and this directly translates to reduction in network traffic. We then propose a novel disk layout (or data placement) technique which ensures that the savings in network resources are also translated to savings in disk resources. In fact, this technique is applicable to a number of other recently proposed storage codes [89, 127, 149, 150, 115, 169, 72] as well, and hence is also of independent interest.

Hitchhiker optimizes reconstruction of a single unit in a stripe without compromising any of the two properties of RS-based systems. Reconstruction of single units in a stripe is the most common reconstruction scenario in practice, as validated by our measurements from Facebook's data-warehouse cluster which reveal that 98.08% of all recoveries involve recovering a single unit in a stripe (see Chapter 3 and Section 5.7). Moreover, at any point in time, Hitchhiker can alternatively perform the (non-optimized) reconstruction of single or multiple units as in RS-based systems by connecting to *any* k of the remaining units. It follows that optimizations or solutions proposed outside the erasure coding component of a storage system (e.g., [20, 102, 33]) can be used in conjunction with Hitchhiker by simply treating Hitchhiker as functionally equivalent to an RS code, thereby allowing for the benefits of both solutions.

We have implemented Hitchhiker in the Hadoop Distributed File System (HDFS). HDFS is one of the most popular open-source distributed file systems with widespread adoption in the industry. For example, multiple tens of Petabytes are being stored via RS encoding in HDFS at Facebook, a popular social-networking company.

We evaluated Hitchhiker on two clusters in Facebook's data centers, with the default HDFS parameters of $(k = 10, r = 4)$. We first deployed Hitchhiker on a test cluster comprising 60 machines, and verified that the savings in the amount of download during reconstruction is as guaranteed by theory. We then evaluated various metrics of Hitchhiker on the data-warehouse cluster in production consisting of multiple thousands of machines, in the presence of ongoing real-time traffic and workloads. We observed that Hitchhiker reduces the time required for reading data during reconstruction by 32%, and reduces the computation time during reconstruction by 36%. Table 5.1 details the comparison between Hitchhiker and RS-based systems with respect to various metrics for $(k = 10, r = 4)$.[2] Based on our measurements [131] of the amount of data transfer for reconstruction of RS-encoded data in the data-warehouse cluster at Facebook (discussed above), employing Hitchhiker would save close to 62TB of disk and cross-rack traffic every day while retaining the same storage overhead, reliability, and system parameters.

[2]More specifically, these numbers are for the 'Hitchhiker-XOR+' version of Hitchhiker.

5.2 Related work

Erasure codes have many pros and cons over replication [143, 180]. The most attractive feature of erasure codes is that while replication entails a minimum of $2\times$ storage redundancy, erasure codes can support significantly smaller storage overheads for the same levels of reliability. Many storage systems thus employ erasure codes for various application scenarios [142, 20, 51, 147]. Traditional erasure codes however face the problem of inefficient reconstruction. To this end, several works (e.g., [20, 102, 33]) propose system level solutions that can be employed to reduce data transfer during reconstruction operations, such as caching the data read during reconstruction, batching multiple recovery operations in a stripe, or delaying the recovery operations. While these solutions consider the erasure code as a black-box, Hitchhiker modifies this black box, employing the new erasure code constructed using the Piggybacking framework presented in Chapter 4 to address the reconstruction problem. Note that Hitchhiker retains all the properties of the underlying RS-based system. Hence any solution proposed outside of the erasure-code module can be employed in conjunction with Hitchhiker to benefit from both the solutions.

The problem of reducing the amount of data accessed during reconstruction through the design of new erasure codes has received much attention in the recent past [115, 129, 146, 73, 127, 49, 150, 178, 183, 72]. However, all existing practical solutions either require the inclusion of additional parity units, thereby increasing the storage overheads [146, 73, 49, 127, 150], or are applicable in very limited settings [89, 178, 183, 72].

The idea of connecting to more machines and downloading smaller amounts of data from each node was proposed in [41] as a part of the 'regenerating codes model'. However, all existing practical constructions of regenerating codes necessitate a high storage redundancy in the system, e.g., codes in [127] require $r \geq (k-1)$. Rotated-RS [89] is another class of codes proposed for the same purpose. However, it supports at most 3 parities, and moreover, its fault tolerance is established via a computer search. Recently, optimized recovery algorithms [178, 183] have been proposed for EVENODD and RDP codes, but they support only 2 parities. For the parameters where [89, 178, 183] exist, Hitchhiker performs at least as good, while also supporting an arbitrary number of parities. An erasure-coded storage system which also optimizes for data download during reconstruction is presented in [72]. While this system achieves minimum possible download during reconstruction, it supports only 2 parities. Furthermore, [72] requires decode operation to be performed for every read request since it cannot reconstruct an identical version of a failed unit but only reconstruct a functionally equivalent version.

The systems proposed in [73, 146, 49] employ another class of codes called local-repair codes to reduce the number blocks accessed during reconstruction. This, in turn, also reduces the total amount of data read and downloaded during reconstruction. However, these codes necessitate addition of at least 25% to 50% more parity units to the code, thereby increasing the storage space requirements.

5.3 Background and notation

In this section, we will briefly discuss some background material, while also introducing the notation and terminology that will be used throughout this chapter.

Reed-Solomon (RS) codes

A $(k,\ r)$ RS code [140] encodes k data bytes into r parity bytes. The code operates on each set of k bytes independently and in an identical fashion. Each such set on which independent and identical operations are performed by the code is called a stripe. Figure 5.1 depicts ten units of data encoded using a $(k = 10,\ r = 4)$ RS code that generates four parity units. Here, $a_1, \ldots, a_{10}$ are one byte each, and so are $b_1, \ldots, b_{10}$. Note that the code is operating independently and identically on the two columns, and hence each of the two columns constitute a stripe of this code. We use the notation $\boldsymbol{a} = [a_1\ \cdots\ a_{10}]$ and $\boldsymbol{b} = [b_1\ \cdots\ b_{10}]$. Each of the functions f_1, f_2, f_3 and f_4, called *parity functions*, operate on $k = 10$ data bytes to generate the $r = 4$ parities. The output of each of these functions comprises one byte. These functions are such that one can reconstruct $\boldsymbol{a}$ from any 10 of the 14 bytes $\{a_1, \ldots, a_{10}, f_1(\boldsymbol{a}), \ldots, f_4(\boldsymbol{a})\}$. In general, a $(k,\ r)$ RS code has r parity functions generating the r parity bytes such that all the k data bytes are recoverable from any k of the $(k + r)$ bytes in a stripe.

In the above discussion, each unit is considered indivisible: all the bytes in a unit share the same fate, i.e., all the bytes are either available or unavailable. Hence, a reconstruction operation is always performed on one or more entire units. A unit may be the smallest granularity of the data handled by a storage system, or it may be a data chunk that is always written onto the same disk block.

Reconstruction of any unit under the traditional RS framework is performed in the following manner. Both the stripes of any 10 of the remaining 13 units are accessed. The RS code guarantees that any desired data can be obtained from any 10 of the units, allowing for reconstruction of the requisite unit from this accessed data. This reconstruction operation in the RS code requires accessing a total of 20 bytes from the other units.

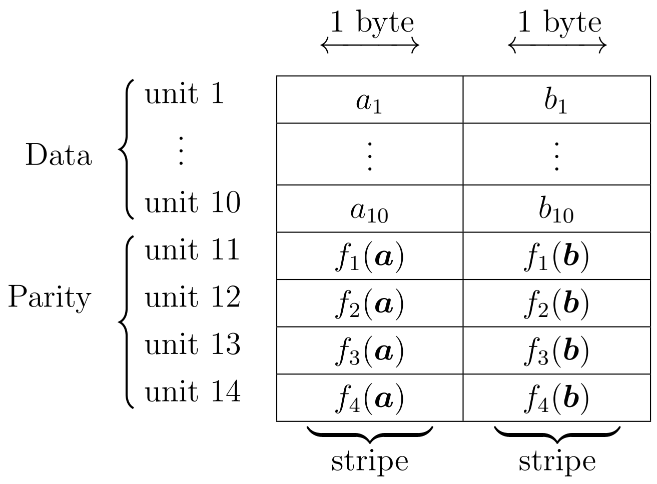


Figure 5.1: Two stripes of a (k=10, r=4) Reed-Solomon (RS) code. Ten units of data (first ten rows) are encoded using the RS code to generate four parity units (last four rows).

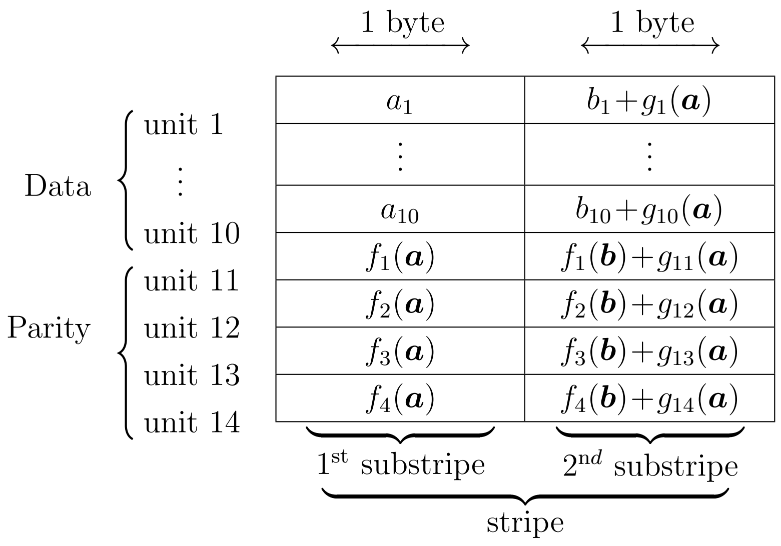
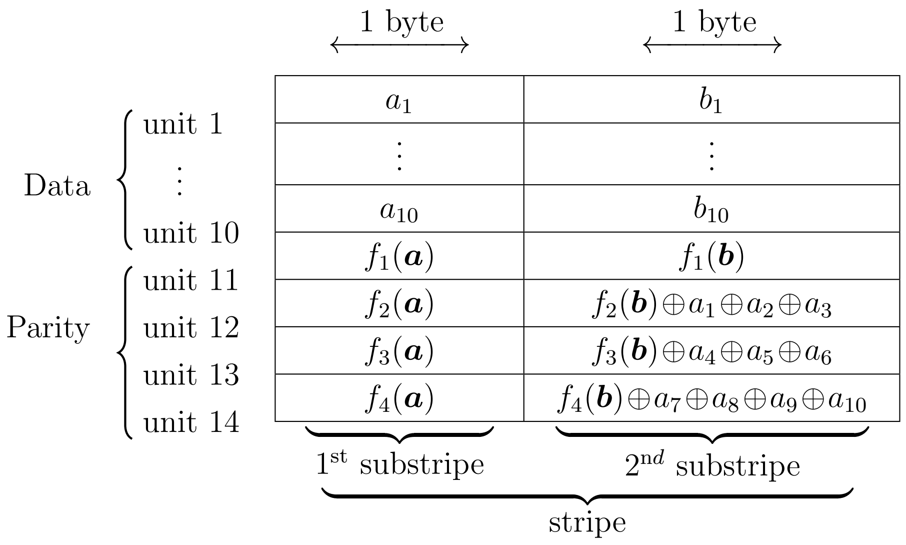

1 byte
1 byte
Data
unit 1
⋮
unit 10
Parity
unit 11
unit 12
unit 13
unit 14
a_1
$b_1+g_1(\boldsymbol{a})$
a_{10}
$b_{10}+g_{10}(\boldsymbol{a})$
$f_1(\boldsymbol{a})$
$f_1(\boldsymbol{b})+g_{11}(\boldsymbol{a})$
$f_2(\boldsymbol{a})$
$f_2(\boldsymbol{b})+g_{12}(\boldsymbol{a})$
$f_3(\boldsymbol{a})$
$f_3(\boldsymbol{b})+g_{13}(\boldsymbol{a})$
$f_4(\boldsymbol{a})$
$f_4(\boldsymbol{b})+g_{14}(\boldsymbol{a})$
1st substripe
2nd substripe
stripe

Figure 5.2: The Piggybacking framework for parameters (k=10, r=4) with a systematic RS code as the base code. Each row represents one unit of data.



unit 1	a_1	b_1
$\vdots$	$\vdots$	$\vdots$
unit 10	a_{10}	b_{10}
unit 11	$f_1(\boldsymbol{a})$	$f_1(\boldsymbol{b})$
unit 12	$f_2(\boldsymbol{a})$	$f_2(\boldsymbol{b}) \oplus a_1 \oplus a_2 \oplus a_3$
unit 13	$f_3(\boldsymbol{a})$	$f_3(\boldsymbol{b}) \oplus a_4 \oplus a_5 \oplus a_6$
unit 14	$f_4(\boldsymbol{a})$	$f_4(\boldsymbol{b}) \oplus a_7 \oplus a_8 \oplus a_9 \oplus a_{10}$

Figure 5.3: Hitchhiker-XOR code for (k=10, r=4). Each row represents one unit of data.

Piggybacking RS codes

We will now briefly review the Piggybacking framework, specifically Piggybacking design 1, presented in Chapter 4. Here we will employ the Piggybacking framework with RS codes as the underlying base codes. For the sake of simplicity, with a slight abuse of terminology, we will refer to Piggybacking design 1 as the Piggybacking framework throughout this chapter.

The Piggybacking framework operates on pairs of stripes of an RS code (e.g., the pair of columns depicted in Figure 5.1). The framework allows for arbitrary functions of the data pertaining to one stripe to be added to the second stripe. This is depicted in Figure 5.2 where arbitrary functions $g_1, \ldots, g_{14}$ of the data of the first stripe of the RS code $\boldsymbol{a}$ are added to the second stripe of the RS code. Each of these functions outputs values of size one byte. The Piggybacking framework performs independent and identical operations on pairs of columns, and hence a stripe consists of two columns. The constituent columns of a stripe will be referred to as *substripes* (see Figure 5.2).

Irrespective of the choice of the Piggybacking functions $g_1, \ldots, g_{14}$, the code retains the fault tolerance and the storage efficiency of the underlying RS code. To see the fault tolerance, recall that RS codes allow for tolerating failure of any r units. In our setting of ($k = 10$, $r = 4$), this amounts to being able to reconstruct the entire data from any 10 of the 14 units in that stripe. Now consider the code of Figure 5.2, and consider any 10 units (rows). The first column of Figure 5.2 is identical to the first stripe (column) of the RS code of Figure 5.1, which allows for reconstruction of $\boldsymbol{a}$ from these 10 units. Access to $\boldsymbol{a}$ now allows us to compute the values of the functions $g_1(\boldsymbol{a}), \ldots, g_{14}(\boldsymbol{a})$, and subtract the respective functions out from the second columns of the 10 units under consideration. What remains are some 10 bytes out of $\{b_1, \ldots, b_{10}, f_1(\boldsymbol{b}), \ldots, f_4(\boldsymbol{b})\}$. This is identical to the second stripe (column) of the RS code in Figure 5.1, which allows for the reconstruction of $\boldsymbol{b}$. It follows that the code of Figure 5.2 can also tolerate the failure of any $r = 4$ units. We will now argue storage efficiency. Each function g_i outputs one byte of data. Moreover, the operation of adding this function to the RS code is performed via finite field arithmetic [100], and hence the result also comprises precisely one byte. Thus the amount of storage is not increased upon performing these operations. It is easy to see that each of these arguments extend to any generic values of the parameters k and r.

5.4 Hitchhiker's erasure code

The proposed code has three versions, two of which require only XOR operations in addition to encoding of the underlying RS code. The XOR-only feature of these erasure codes significantly reduces the computational complexity of decoding, making degraded reads and failure recovery faster (Section 5.7). Hitchhiker's erasure code optimizes only the reconstruction of data units; reconstruction of parity units is performed as in RS codes.

The three versions of Hitchhiker's erasure code are described below. Each version is first illustrated with an example for the parameters ($k = 10,\ r = 4$), followed by the generalization to arbitrary values of the parameters. Without loss of generality, the description of the codes' operations considers only a single stripe (comprising two substripes). Identical operations are performed on each stripe of the data.

Hitchhiker-XOR

As compared to a ($k = 10, r = 4$) RS code, Hitchhiker-XOR saves 35% in the amount of data required during the reconstruction of the first six data units and 30% during the reconstruction of the remaining four data units.

Encoding

The code for ($k = 10,\ r = 4$) is shown in Figure 5.3. The figure depicts a single stripe of this code, comprising two substripes. The encoding operation in this code requires only XOR

unit 1	a_1	b_1
$\vdots$	$\vdots$	$\vdots$
unit 10	a_{10}	b_{10}
unit 11	$f_1(\boldsymbol{a})$	$f_1(\boldsymbol{b})$
unit 12	$\bigoplus_{i=4}^{10} a_i \oplus \bigoplus_{i=1}^{10} b_i$	$\bigoplus_{i=1}^{10} b_i \oplus \bigoplus_{i=1}^{3} a_i$
unit 13	$f_3(\boldsymbol{a})$	$f_3(\boldsymbol{b}) \oplus \bigoplus_{i=4}^{6} a_i$
unit 14	$f_4(\boldsymbol{a})$	$f_4(\boldsymbol{b}) \oplus \bigoplus_{i=7}^{9} a_i$

Figure 5.4: Hitchhiker-XOR+ for $(k=10, r=4)$. Parity 2 of the underlying RS code is all-XOR.

unit 1	a_1	b_1
$\vdots$	$\vdots$	$\vdots$
unit 10	a_{10}	b_{10}
unit 11	$f_1(\boldsymbol{a})$	$f_1(\boldsymbol{b})$
unit 12	$f_2(0,0,0,a_4,\ldots,a_{10}) \oplus f_2(\boldsymbol{b})$	$f_2(\boldsymbol{b}) \oplus f_2(a_1,a_2,a_3,0,\ldots,0)$
unit 13	$f_3(\boldsymbol{a})$	$f_3(\boldsymbol{b}) \oplus f_2(0,0,0,a_4,a_5,a_6,0,\ldots,0)$
unit 14	$f_4(\boldsymbol{a})$	$f_4(\boldsymbol{b}) \oplus f_2(0,\ldots,0,a_7,a_8,a_9,0)$

Figure 5.5: Hitchhiker-nonXOR code for $(k=10, r=4)$. This can be built on any RS code. Each row is one unit of data.

operations in addition to the underlying RS encoding.

Reconstruction

First consider reconstructing the first unit. This requires reconstruction of $\{a_1, b_1\}$ from the remaining units. Hitchhiker-XOR accomplishes this using only 13 bytes from the other units: the bytes belonging to both the substripes of units $\{2, 3\}$ and the bytes belonging to only the second substripe of units $\{4, \ldots, 12\}$. These 13 bytes are $\{a_2, a_3, b_2, b_3 \ldots, b_{10}, f_1(\boldsymbol{b}), f_2(\boldsymbol{b}) \oplus a_1 \oplus a_2 \oplus a_3\}$. The decoding procedure comprises three steps. Step 1: observe that the 10 bytes $\{b_2, \ldots, b_{10}, f_1(\boldsymbol{b})\}$ are identical to the corresponding 10 bytes in the RS encoding of $\boldsymbol{b}$ (Figure 5.1). RS decoding of these 10 bytes gives $\boldsymbol{b}$ (and this includes one of the desired bytes b_1). Step 2: XOR $f_2(\boldsymbol{b})$ with the second byte $(f_2(\boldsymbol{b}) \oplus a_1 \oplus a_2 \oplus a_3)$ of the 12th unit. This gives $(a_1 \oplus a_2 \oplus a_3)$. Step 3: XORing this with a_2 and a_3 gives a_1. Thus both a_1 and b_1 are reconstructed by using only 13 bytes, as opposed to 20 bytes in RS codes, resulting in a saving of 35%.

Let us now consider the reconstruction of any unit $i \in \{1, \ldots, 10\}$, which requires reconstruction of $\{a_i, b_i\}$. We shall first describe what data (from the other units) is required for the reconstruction, following which we describe the decoding operation. Any data unit $i \in \{1, 2, 3\}$ is reconstructed using the following 13 bytes: the bytes of both the substripes of units $\{1, 2, 3\} \backslash \{i\}$, and the bytes belonging to only the second substripe from units $\{4, \ldots, 12\}$.[3] Any data unit $i \in \{4, 5, 6\}$ is also reconstructed using only 13 bytes: the bytes belonging to both the substripes of units $\{4, 5, 6\} \backslash \{i\}$, and the bytes belonging to only the second substripe of units $\{1, 2, 3, 7, \ldots, 11, 13\}$. Any data unit $i \in \{7, 8, 9, 10\}$ is reconstructed using 14 bytes: both substripes of units $\{7, 8, 9, 10\} \backslash \{i\}$, and only the second substripe of units $\{1, \ldots, 6, 11, 14\}$.

Three-step decoding procedure:
Step 1: The set of 10 bytes $\{b_1, \ldots, b_{10}, f_1(\boldsymbol{b})\} \backslash \{b_i\}$ belonging to the second substripe of the units $\{1, \ldots, 11\} \backslash \{i\}$ is identical to the 10 corresponding encoded bytes in the RS code. Perform RS decoding of these 10 bytes to get $\boldsymbol{b}$ (which includes one of the desired bytes b_i).
Step 2: In the other bytes accessed, subtract out all components that involve $\boldsymbol{b}$.
Step 3: XOR the resulting bytes to get a_i.

Observe that during the reconstruction of any data unit, the remaining data units *do not* perform any computation. This property is termed 'repair-by-transfer' [149], and it carries over to all three versions of Hitchhiker's erasure code.

Hitchhiker-XOR+

Hitchhiker-XOR+ further reduces the amount of data required for reconstruction as compared to Hitchhiker-XOR, and employs only additional XOR operations. It however requires the underlying RS code to possess a certain property. This property, which we term the *all-XOR-parity* property, requires at least one parity function of the RS code to be an XOR of all the data units. That is, a $(k,\ r)$ RS code satisfying all-XOR-parity will have one of the r parity bytes as an XOR of all the k data bytes. For $(k = 10,\ r = 4)$, Hitchhiker-XOR+ requires 35% lesser data for reconstruction of any of the data units as compared to RS codes.

Encoding

The $(k = 10,\ r = 4)$ Hitchhiker-XOR+ code is shown in Figure 5.4. The Hitchhiker-XOR+ code is obtained by performing one additional XOR operation on top of Hitchhiker-XOR: in the second parity of Hitchhiker-XOR, the byte of the second substripe is XORed onto

[3]For any set $\mathcal{A}$ and any element $i \in \mathcal{A}$, the notation $\mathcal{A} \backslash \{i\}$ denotes all elements of $\mathcal{A}$ except i.

the byte of the first substripe to give Hitchhiker-XOR+. The underlying RS code in this example satisfies the all-XOR-parity property with its second parity function f_2 being an XOR of all the inputs.

We now argue that this additional XOR operation does not violate the fault tolerance level and storage efficiency. To see fault tolerance, observe that the data of the second parity unit of Hitchhiker-XOR+ can always be converted back to that under Hitchhiker-XOR by XORing its second substripe with its first substripe. It follows that the data in any unit under Hitchhiker-XOR+ is equivalent to the data in the corresponding unit in Hitchhiker-XOR. The fault tolerance properties of Hitchhiker-XOR thus carry over to Hitchhiker-XOR+. Storage efficiency is retained because the additional XOR operation does not increase the space requirement.

Decoding

The recovery of any unit i requires 13 bytes from the other units. The choice of the bytes to be accessed depends on the value of i, and is described below. The bytes required for the reconstruction of any data unit $i \in \{1, \ldots, 6\}$ are identical to that in Hitchhiker-XOR. Any data unit $i \in \{7, 8, 9\}$ is reconstructed using the following 13 bytes: the bytes of both substripes of units $\{7, 8, 9\}\backslash\{i\}$, and the bytes of only the second substripes of units $\{1, \ldots, 6, 10, 11, 14\}$. The tenth unit is also reconstructed using only 13 bytes: the bytes of only the second substripes of units $\{1, \ldots, 9, 11, 13, 14\}$, and the byte of only the first substripe of unit 12. The decoding procedure that operates on these 13 bytes is identical to the three-step decoding procedure described above.

Hitchhiker-nonXOR

We saw that Hitchhiker-XOR+ results in more savings as compared to Hitchhiker-XOR, but requires the underlying RS code to have the all-XOR-parity property. Hitchhiker-nonXOR presented here guarantees the same savings as Hitchhiker-XOR+ even when the underlying RS code does not possess the all-XOR-parity property, but at the cost of additional finite-field arithmetic. Hitchhiker-nonXOR can thus be built on top of any RS code. It offers a saving of 35% during the reconstruction of any data unit.

Encoding

The code for $(k = 10,\ r = 4)$ is shown in Figure 5.5. As in Hitchhiker-XOR, in the second parity, the first byte is XORed with the second byte. The final value of the second parity as shown in Figure 5.5 is a consequence of the fact that $f_2(\boldsymbol{a}) \oplus f_2(a_1, a_2, a_3, 0, \ldots, 0) = f_2(0, 0, 0, a_4, \ldots, a_{10})$ due to the linearity of RS encoding (this is discussed in greater detail in Section 5.6).

Decoding

Recovery of any unit requires only 13 bytes from other units. This set of 13 bytes is the same as in Hitchhiker-XOR+. The decoding operation is a three-step procedure. The first two steps are identical to the first two steps of the decoding procedure of Hitchhiker-XOR described above. The third step is slightly different, and requires an RS decoding operation (for units 1 to 9), as described below.

During reconstruction of any unit $i \in \{1, 2, 3\}$, the output of the second step is the set of three bytes $\{a_1, a_2, a_3, f_1(a_1, a_2, a_3, 0, \ldots, 0)\}\backslash\{a_i\}$. This is equivalent to having some 10 of the 11 bytes of the set $\{a_1, a_2, a_3, 0, \ldots, 0, f_1(a_1, a_2, a_3, 0, \ldots, 0)\}$. Now, this set of 11 bytes is equal to the set of first 11 bytes of the RS encoding of $\{a_1, a_2, a_3, 0, \ldots, 0\}$. An RS decoding operation thus gives $\{a_1, a_2, a_3\}$ which contains the desired byte a_i. Recovery of any other unit $i \in \{4, \ldots, 9\}$ follows along similar lines.

During the reconstruction of unit 10, the output of the second step is $f_1(0, \ldots, 0, a_{10})$. Hence the third step involves only a single (finite-field) multiplication operation.

Generalization to any $(k,\ r)$

The encoding and decoding procedures for the general case follow along similar lines as the examples discussed above, and are formally described below. In each of the three versions, the amount of data required for reconstruction is reduced by 25% to 45% as compared to RS codes, depending on the values of the parameters k and r. For instance, $(k = 6, r = 3)$ provides a saving of 25% with Hitchhiker-XOR and 34% with Hitchhiker-XOR+ and Hitchhiker-nonXOR; $(k = 20, r = 5)$ provides a savings of 37.5% with Hitchhiker-XOR and 40% with Hitchhiker-XOR+ and Hitchhiker-nonXOR.

Hitchhiker-XOR: The encoding procedure of Hitchhiker-XOR first divides the k data units into $(r-1)$ disjoint sets of roughly equal sizes. For instance, in the $(k{=}10, r{=}4)$ code of Figure 5.3, the three sets are units $\{1, 2, 3\}$, units $\{4, 5, 6\}$ and units $\{7, 8, 9, 10\}$. For each set $j \in \{1, \ldots, r-1\}$, the bytes of the first substripe of all units in set j are XORed, and the resultant is XORed with the second substripe of the $(j+1)^{\text{th}}$ parity unit.

Reconstruction of a data unit belonging to any set j requires the bytes of both the substripes of the other data units in set j, only the second byte of all other data units,

and the second bytes of the first and $(j+1)^{\text{th}}$ parity units. The decoding procedure for reconstruction of any data unit i is executed in three steps:
Step 1: The k bytes $\{b_1, \ldots, b_k, f_1(\boldsymbol{b})\} \backslash \{b_i\}$ belonging to the second substripe of the units $\{1, \ldots, k+1\} \backslash \{i\}$ are identical to the k corresponding encoded bytes in the underlying RS code. Perform RS decoding of these k bytes to get $\boldsymbol{b}$ (which includes one of the desired bytes b_i).
Step 2: In the other bytes accessed, subtract out all components that involve $\boldsymbol{b}$.
Step 3: XOR the resulting bytes to get a_i.

If the size of a set is s, reconstruction of any data unit in this set requires $(k+s)$ bytes (as compared to $2k$ under RS codes).

Hitchhiker-XOR+: Assume without loss of generality that, in the underlying RS code, the all-XOR property is satisfied by the second parity. The encoding procedure first selects a number $\ell \in \{0, \ldots, k\}$ and partitions the first $(k-\ell)$ data units into $(r-1)$ sets of roughly equal sizes. On these $(k-\ell)$ data units and r parity units, it performs an encoding identical to that in Hitchhiker-XOR. Next, in the second parity unit, the byte of the second substripe is XORed onto the byte of the first substripe.

Reconstruction of any of the first $(k-\ell)$ data units is performed in a manner identical to that in Hitchhiker-XOR. Reconstruction of any of the last ℓ data units requires the byte of the first substripe of the second parity and the bytes of the second substripes of all other units. The decoding procedure remains identical to the three-step procedure of Hitchhiker-XOR stated above.

For any of the first $(k-\ell)$ data units, if the size of its set is s then reconstruction of that data unit requires $(k+s)$ bytes (as compared to $2k$ under RS). The reconstruction of any of the last ℓ units requires $(k+r+\ell-2)$ bytes (as compared to $2k$ under RS). The parameter ℓ can be chosen to minimize the average or maximum data required for reconstruction as per the system requirements.

Hitchhiker-nonXOR: The encoding procedure is identical to that of Hitchhiker-XOR+, except that instead of XORing the bytes of the first substripe of the data units in each set, these bytes are encoded using the underlying RS encoding function considering all other data units that do not belong to the set as zeros.

The collection of data bytes required for the reconstruction of any data unit is identical to that under Hitchhiker-XOR+. The decoding operation for reconstruction is a three-step procedure. The first two steps are identical to the first two steps of the decoding procedure of Hitchhiker-XOR described above. The third step requires an RS decoding operation (recall from the $(k=10,\ r=4)$ case described above). In particular, the output of the second step when reconstructing a data unit i will be equal to k bytes that would have been obtained from the RS encoding of the data bytes in the units belonging to that set with all other data bytes set to zero. An RS decoding operation performed on these bytes now gives a_i, thus recovering the i^{th} data unit (recall that b_i is reconstructed in Step 1 itself.) The data access patterns during reconstruction and the amount of savings under Hitchhiker-nonXOR are identical to that under Hitchhiker-XOR+.

5.5 "Hop-and-Couple" for disk efficiency

The description of the codes in Section 5.3 and Section 5.4 considers only two bytes per data unit. We now move on to consider the more realistic scenario where each of the k data units to be encoded is larger (than two bytes). In the encoding process, these k data units are first partitioned into stripes, and identical encoding operations are performed on each of the stripes. The RS code considers one byte each from the k data units as a stripe. On the other hand, Hitchhiker's erasure code has two substripes within a stripe (Section 5.4), and hence it *couples* pairs of bytes within each of the k data units to form the substripes of a stripe. We will shortly see that the choice of the bytes to be coupled plays a crucial role in determining the efficiency of disk reads during reconstruction.

A natural strategy for forming the stripes for Hitchhiker's erasure code is to couple adjacent bytes within each unit, with the first stripe comprising the first two bytes of each of the units, the second stripe comprising the next two bytes, and so on. Figure 5.6a depicts such a method of coupling for ($k = 10, r = 4$). In the figure, the bytes accessed during the reconstruction of the first data unit are shaded. This method of coupling, however, results in highly discontiguous reads during reconstruction: alternate bytes are read from units 4 to 12 as shown in the figure. This high degree of discontinuity is detrimental to disk read performance, and forfeits the potential savings in disk IO during data reconstruction. The issue of discontiguous reads due to coupling of adjacent bytes is not limited to the reconstruction of the first data unit - it arises during reconstruction of any of the data units.

In order to ensure that the savings offered by Hitchhiker's erasure codes in the amount of data read during reconstruction are effectively translated to gains in disk read efficiency, we propose a coupling technique for forming stripes that we call *hop-and-couple.* This technique aims to minimize the degree of discontinuity in disk reads during the reconstruction of data units. The hop-and-couple technique couples a byte with another byte within the same unit that is a certain distance ahead (with respect to the natural ordering of bytes within a unit), i.e., it couples bytes after "hopping" a certain distance. We term this distance as the *hop-length.* This technique is illustrated in Figure 5.6b, where the hop-length is chosen to

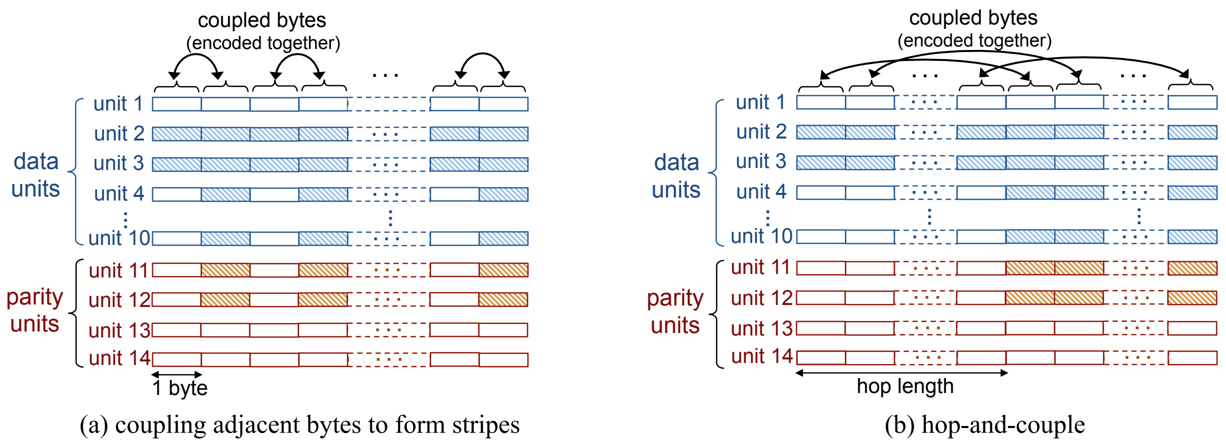


Figure 5.6: Two ways of coupling bytes to form stripes for Hitchhiker's erasure code. The shaded bytes are read and downloaded for the reconstruction of the first unit. While both methods require the same amount of data to be read, the reading is discontiguous in (a), while (b) ensures that the data to be read is contiguous.

be half the size of a unit.

The hop-length may be chosen to be any number that divides $\frac{B}{2}$, where B denotes the size of each unit. This condition ensures that all the bytes in the unit are indeed coupled. Coupling adjacent bytes (e.g., Figure 5.6a) is a degenerate case where the hop-length equals 1. The hop-length significantly affects the contiguity of the data read during reconstruction of the data units, in that the data is read as contiguous chunks of size equal to the hop-length. For Hitchhiker's erasure codes, a hop-length of $\frac{B}{2}$ minimizes the total number of discontiguous reads required during the reconstruction of data units. While higher values of hop-length reduces the number of discontiguous reads, it results in bytes further apart being coupled to form stripes. This is a trade-off to be considered when choosing the value of the hop-length, and is discussed further in Section 5.8.

We note that the reconstruction operation under RS codes reads the entire data from k of the units, and hence trivially, the reads are contiguous. On the other hand, any erasure code that attempts to make reconstruction more efficient by downloading *partial* data from the units (e.g.,[89, 127, 149, 150, 115, 169, 72]) will encounter the issue of discontiguous reads, as in Hitchhiker's erasure code. Any such erasure code would have multiple (say, α) substripes in every stripe and would read a subset of these substripes from each of the units during reconstruction. The hop-and-couple technique can be applied to any such erasure code to translate the network savings to disk savings as well. The hop-length can be chosen to be any number that divides $\frac{B}{\alpha}$. As in the case of Hitchhiker's erasure codes (where $\alpha = 2$), this condition ensures that all the bytes are indeed coupled. If the bytes to be coupled are chosen with hop-length equal to B/α, then the hop-and-couple technique would ensure that all the bytes of a substripe are contiguous within any unit. Reading a substripe from a unit would then result in a contiguous disk read, thereby minimizing the total degree of discontiguity in disk reads during reconstruction.

5.6 Implementation

We have implemented Hitchhiker in the Hadoop Distributed File System (HDFS). HDFS-RAID [67] is a module in HDFS that deals with erasure codes and is based on [51]. This module forms the basis for the erasure-coded storage system employed in the data-warehouse cluster at Facebook, and is open sourced under Apache. HDFS-RAID deployed at Facebook is based on RS codes, and we will refer to this system as RS-based HDFS-RAID. Hitchhiker builds on top of RS codes, and the present implementation uses the RS encoder and decoder modules of RS-based HDFS-RAID as its building blocks.

Brief description of HDFS-RAID

HDFS stores each file by dividing it into *blocks* of a certain size. By default, the size of each block is 256 MB, and this is also the value that is typically used in practice. In HDFS, three replicas of each block are stored in the system by default. HDFS-RAID offers RS codes as an alternative to replication for maintaining redundancy.

The relevant modules of HDFS-RAID and the execution flows for relevant operations are depicted in Figure 5.7. The *RAID-Node* manages all operations related to the use of erasure codes in HDFS. It has a list of files that are to be converted from the replicated state to the erasure-coded state, and periodically performs encoding of these files via MapReduce jobs. Sets of k blocks from these files are encoded to generate r parity blocks each.[4] Once the r parity blocks of a set are successfully written into the file system, the replicas of the k data blocks of that set are deleted. The MapReduce job calls the *Encoder* of the erasure code to perform encoding. The Encoder uses the *Parallel-Reader* to read the data from the k blocks that are to be encoded. The Parallel-Reader opens k parallel streams, issues HDFS read requests for these blocks, and puts the data read from each stream into different buffers. Typically, the buffers are 1 MB each. When one buffer-sized amount of data is read from each of the k blocks, the Encoder performs the computations pertaining to the encoding operation. This process is repeated until the entire data from the k blocks are encoded.

[4]In the context of HDFS, the term block corresponds to a unit and we will use these terms interchangeably.

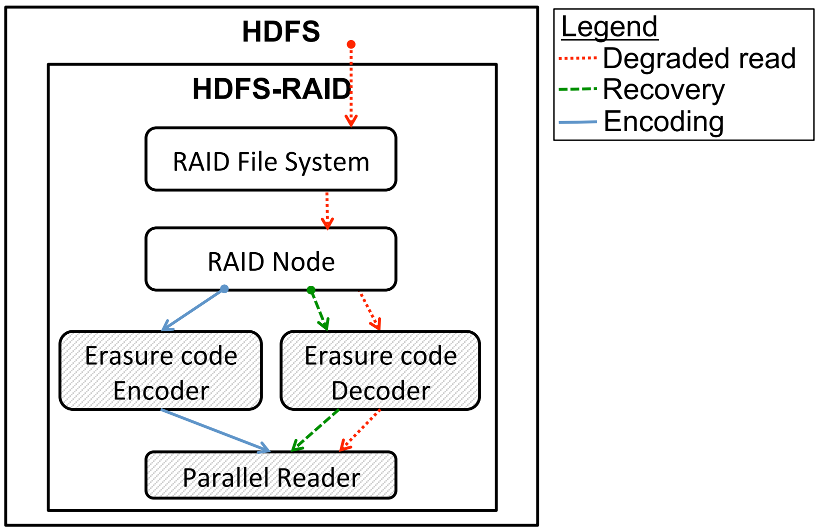


Figure 5.7: Relevant modules in HDFS-RAID. The execution flow for encoding, degraded reads and reconstruction operations are shown. Hitchhiker is implemented in the shaded modules.

The RAID-Node also handles recovery operations, i.e., reconstructing missing blocks in order to maintain the reliability of the system. The RAID-Node has a list of blocks that are missing and need to be *recovered*. It periodically goes through this list and reconstructs the blocks by executing a MapReduce job. The MapReduce job calls the *Decoder* of the erasure code to perform the reconstruction operation. The Decoder uses the Parallel-Reader to read the data required for reconstruction. For an RS code, data from k blocks belonging to the same set as that of the block under reconstruction is read in parallel. As in the encoding process, data from the k blocks are read into buffers, and the computations are performed once one buffer size amount of data is read from each of the k blocks. This is repeated until the entire block is reconstructed.

HDFS directs any request for a *degraded read* (i.e., a read request for a block that is unavailable) to the *RAID File System*. The requested block is reconstructed on the fly by the RAID-Node via a MapReduce job. The execution of this reconstruction operation is identical to that in the reconstruction process discussed above.

Hitchhiker in HDFS

We implemented Hitchhiker in HDFS making use of the new erasure code constructed using the Piggybacking framework (Section 5.4) and the hop-and-couple technique (Section 5.5). We implemented all three versions of the proposed storage code: Hitchhiker-XOR, Hitchhiker-XOR+, and Hitchhiker-nonXOR. This required implementing new Encoder, Decoder and Parallel-Reader modules (shaded in Figure 5.7), entailing $7k$ lines of code. The implementation details described below pertain to parameters ($k = 10,\ r = 4$) which are the default parameters in HDFS. We emphasize that, however, Hitchhiker is generic and supports all values of the parameters k and r.

Hitchhiker-XOR and Hitchhiker-XOR+

The implementation of these two versions of Hitchhiker is exactly as described in Section 5.4 and Section 5.4 respectively.

Hitchhiker-nonXOR

Hitchhiker-nonXOR requires finite field arithmetic operations to be performed in addition to the operations performed for the underlying RS code. We now describe how our implementation executes these operations.

Encoder: As seen in Figure 5.5, in addition to the underlying RS code, the Encoder needs to compute the functions $f_2(a_1,a_2,a_3,0...,0)$, $f_2(0,...,0,a_4,a_5,a_6,0...,0)$, and $f_2(0,...,0,a_7,a_8,a_9,0)$, where f_2 is the second parity function of the RS code. One way to perform these computations is to simply employ the existing RS encoder. This approach, however, involves computations that are superfluous to our purpose: The RS encoder uses an algorithm based on polynomial computations [100] that inherently computes *all* the four parities $f_1(\boldsymbol{x})$, $f_2(\boldsymbol{x})$, $f_3(\boldsymbol{x})$, and $f_4(\boldsymbol{x})$ for any given input $\boldsymbol{x}$. On the other hand, we require only one of the four parity functions for each of the three distinct inputs $(a_1,a_2,a_3,0...,0)$, $(0,...,0,a_4,a_5,a_6,0...,0)$ and $(0,...,0,a_7,a_8,a_9,0)$. Furthermore, these inputs are sparse, i.e., most of the input bytes are zeros.

Our Encoder implementation takes a different approach, exploiting the fact that RS codes have the property of *linearity*, i.e., each of the parity bytes of an RS code can be written as a linear combination of the data bytes. Any parity function f_ℓ can thus be specified as $f_\ell(\boldsymbol{x}) = \bigoplus_{j=1}^{10} c_{\ell,j}x_j$ for some constants $c_{\ell,1},\ldots,c_{\ell,10}$. We first inferred the values of these constants for each of the parity functions (from the existing "black-box" RS encoder) in the following manner. We first fed the input $[1\ 0\ \cdots\ 0]$ to the encoder. This gives the constants $c_{1,1},\ldots,c_{4,1}$ as the values of the four parities respectively in the output from the encoder. The values of the other constants were obtained in a similar manner by feeding different unit vectors as inputs to the encoder. Note that obtaining the values of these constants from the "black-box" RS encoder is a one-time task. With these constants, the encoder computes the desired functions simply as linear combinations of the given data units (for example, we compute $f_2(a_1,a_2,a_3,0\ldots,0)$ simply as $c_{2,1}a_1 \oplus c_{2,2}a_2 \oplus c_{2,3}a_3$).

Decoder: As seen in Section 5.4, during reconstruction of any of the first nine units, the first byte is reconstructed by performing an RS decoding of the data obtained in the intermediate step. One straightforward way to implement this is to use the existing RS decoder for this operation. However, we take a different route towards a computationally cheaper option. We make use of two facts: (a) Observe from the description of the reconstruction process (Section 5.4) that for this RS decoding operation the data is known to be 'sparse', i.e., contains many zeros, and (b) the RS code has the linearity property, and furthermore, any linear combination of zero-valued data is zero. Motivated by this observation, our Decoder simply inverts this sparse linear combination to reconstruct the first substripe in a more efficient manner.

Hop-and-couple

Hitchhiker uses the proposed hop-and-couple technique to couple bytes during encoding. We use a hop-length of half a block since the granularity of data read requests and recovery in HDFS is typically an entire block. As discussed in Section 5.5, this choice of the hop-length minimizes the number of discontiguous reads. However, the implementation can be easily extended to other hop-lengths as well.

When the block size is larger than the buffer size, coupling bytes that are half a block apart requires reading data from two different locations within a block. In Hitchhiker, this is handled by the Parallel-Reader.

Data read patterns during reconstruction

During reconstruction in Hitchhiker, the choice of the blocks from which data is read, the seek locations, and the amount of data read are determined by the identity of the block being reconstructed. Since Hitchhiker uses the hop-and-couple technique for coupling bytes to form stripes during encoding, the reads to be performed during reconstruction are always contiguous within any block (Section 5.5).

For reconstruction of any of the first nine data blocks in Hitchhiker-XOR+ or Hitchhiker-nonXOR or for reconstruction of any of the first six data blocks in Hitchhiker-XOR, Hitchhiker reads and downloads two full blocks (i.e., both substripes) and nine half blocks (only the second substripes). For example, the read patterns for decoding blocks 1 and 4 are as shown in Figure 5.8a and 5.8b respectively. For reconstruction of any of the last four data blocks in Hitchhiker-XOR, Hitchhiker reads and downloads three full blocks (both substripes) and nine half blocks (only the second substripes). For recovery of the tenth data

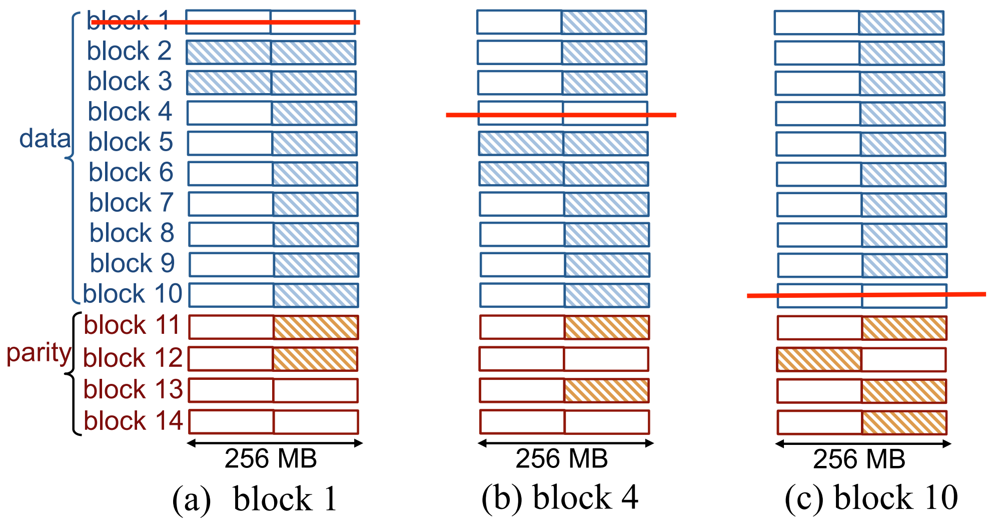


Figure 5.8: Data read patterns during reconstruction of blocks 1, 4 and 10 in Hitchhiker-XOR+: the shaded bytes are read and downloaded.

block in Hitchhiker-XOR+ or Hitchhiker-nonXOR, Hitchhiker reads half a block each from the remaining thirteen blocks. This read pattern is shown in Figure 5.8c.

5.7 Evaluation

Evaluation setup and metrics

We evaluate Hitchhiker using two HDFS clusters at Facebook: (i) the data-warehouse cluster in production comprising multiple thousands of machines, with ongoing real-time traffic and workloads, and (ii) a smaller test cluster comprising around 60 machines. In both the clusters, machines are connected to a rack switch through 1Gb/s Ethernet links. The higher levels of the network tree architecture have 8Gb/s Ethernet connections.

We compare Hitchhiker and RS-based HDFS-RAID in terms of the time taken for computations during encoding and reconstruction, the time taken to read the requisite data during reconstruction, and the amount of data that is read and transferred during reconstruction.[5] Note that in particular, the computation and the data read times during reconstruction are vital since they determine the performance of the system during degraded reads and recovery.

[5]Both systems read the same data during encoding, and are hence trivially identical on this metric.

Evaluation methodology

We first deployed HDFS-RAID with both Hitchhiker and the RS-based system on a 60-machine test cluster at Facebook in order to verify Hitchhiker's end-to-end functionality. We created multiple files with a block size of 256MB, and encoded them separately using Hitchhiker and RS-based HDFS-RAID. The block placement policy of HDFS-RAID ensures that the 14 blocks of an encoded stripe are all stored on different machines. We then forced some of these machines to become unavailable by stopping HDFS related scripts running on them, and collected the logs pertaining to the MapReduce jobs that performed the reconstruction operations. We verified that all reconstruction operations were successful. We also confirmed that the amount of data read and downloaded in Hitchhiker was 35% lower than in RS-based HDFS-RAID, as guaranteed by the proposed codes. We do not perform timing measurements on the test cluster since the network traffic and the workload on these machines do not reflect the real (production) scenario. Instead, we evaluated these metrics directly on the production cluster itself as discussed below.

The encoding and reconstruction operations in HDFS-RAID, including those for degraded reads and recovery, are executed as MapReduce jobs. The data-warehouse cluster does not make any distinction between the MapReduce jobs fired by the RAID-Node and those fired by a user. This allows us to perform evaluations of the timing metrics for encoding, recovery, and degraded read operations by running them as MapReduce jobs on the production cluster. Thus evaluation of all the timing metrics is performed in the presence of real-time production traffic and workloads.

For all the evaluations, we consider the encoding parameters ($k = 10,\ r = 4$), a block size of 256 MB (unless mentioned otherwise), and a buffer size of 1 MB. These are the default parameters of HDFS-RAID. Moreover, these are the parameters employed in the data-warehouse cluster in production at Facebook to store multiple tens of Petabytes.

In the evaluations, we show results of the timing metrics for *one buffer size*. This is sufficient since the same operations are performed repeatedly on one buffer size amount of data until an entire block is processed.

Computation time for degraded reads & recovery

Figure 5.9 shows the comparison of computation time during data reconstruction (from 200 runs each). Note that (i) in both Hitchhiker-XOR+ and Hitchhiker-nonXOR, reconstruction of any of the first nine data blocks entails same amount of computation, (ii) reconstruction of any data block in Hitchhiker-XOR is almost identical to reconstruction of block 1 in

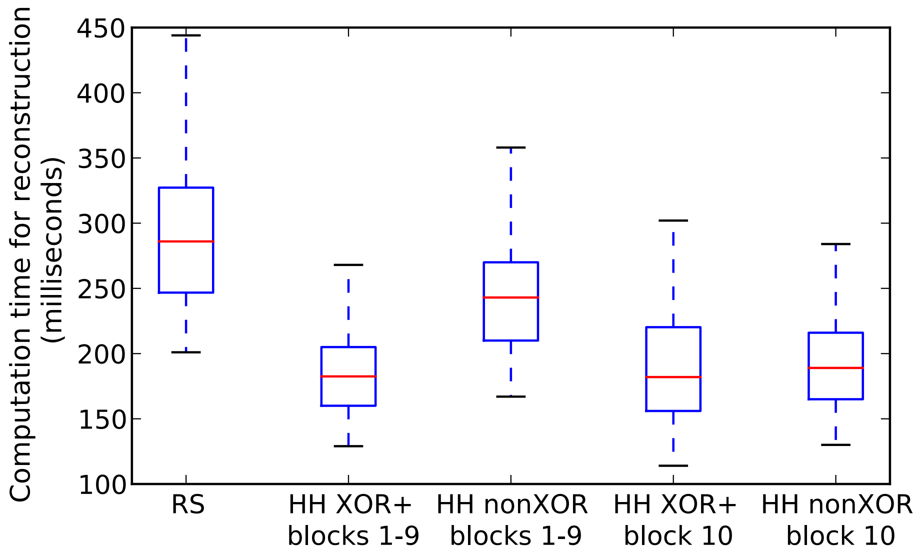


Figure 5.9: A box plot comparing the computation time for reconstruction of 1MB (buffer size) from 200 runs each on Facebook's data-warehouse cluster with real-time production traffic and workloads. (HH = Hitchhiker.)

Hitchhiker-XOR+. Hence, for brevity, no separate measurements are shown.

We can see that Hitchhiker's erasure codes perform faster reconstruction than RS-based HDFS-RAID for any data block. This is because (recall from Section 5.4) the reconstruction operation in Hitchhiker requires performing the resource intensive RS decoding only for *half* the substripes as compared to RS-based HDFS-RAID. In Hitchhiker, the data in the other half of substripes is reconstructed by performing either a few XOR operations (under Hitchhiker-XOR and Hitchhiker-XOR+) or a few finite-field operations (under Hitchhiker-nonXOR). One can also see that Hitchhiker-XOR+ has 25% lower computation time during reconstruction as compared to Hitchhiker-nonXOR for the first nine data blocks; for block 10, the time is almost identical in Hitchhiker-XOR+ and Hitchhiker-nonXOR (as expected from theory (Section 5.4)).

Read time for degraded reads & recovery

Figure 5.12a and Figure 5.12b respectively compare the median and the 95th percentile of the read times during reconstruction for three different block sizes: 4MB, 64MB, and 256MB in Hitchhiker-XOR+. The read patterns for reconstruction of any of the first nine blocks are identical (Section 5.4).

In the median read times for reconstruction of blocks 1-9 and block 10 respectively, in

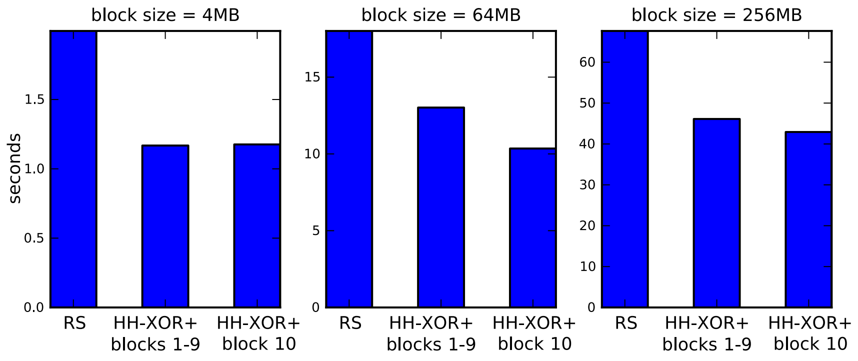


Figure 5.10: Median

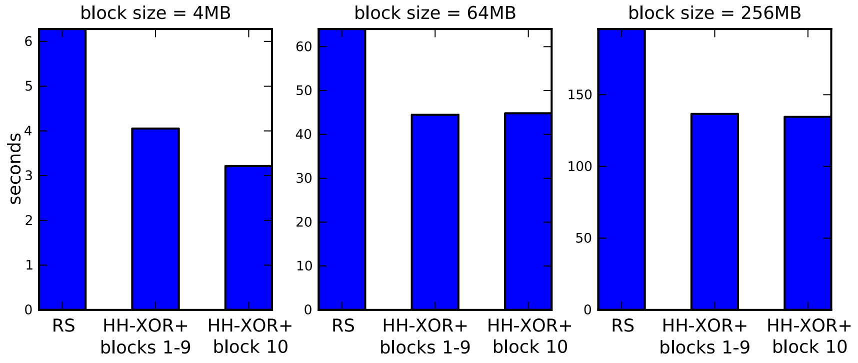


Figure 5.11: 95$^{\text{th}}$ percentile

Figure 5.12: Total read time (in seconds) during reconstruction from 200 runs each on Facebook's data-warehouse cluster with real-time production traffic and workloads. (HH = Hitchhiker.)

comparison to RS-based HDFS-RAID, we observed a reduction of 41.4% and 41% respectively for 4MB block size, 27.7% and 42.5% for 64MB block size, and 31.8% and 36.5% for 256MB block size. For the $95th$ percentile of the read time we observed a reduction of 35.4% and 48.8% for 4MB, 30.5% and 29.9% for 64MB, and 30.2% and 31.2% for 256MB block sizes.

The read pattern of Hitchhiker-nonXOR is identical to Hitchhiker-XOR+, while that of Hitchhiker-XOR is the same for the first six blocks and almost the same for the remaining four blocks. Hence for brevity, we plot the statistics only for Hitchhiker-XOR+.

Although Hitchhiker reads data from more machines as compared to RS-based HDFS-RAID, we see that it gives a superior performance in terms of read latency during reconstruction. The reason is that Hitchhiker reads only *half* a block size from most of the machines it connects to (recall from Section 5.6) whereas RS-based HDFS-RAID reads entire blocks.

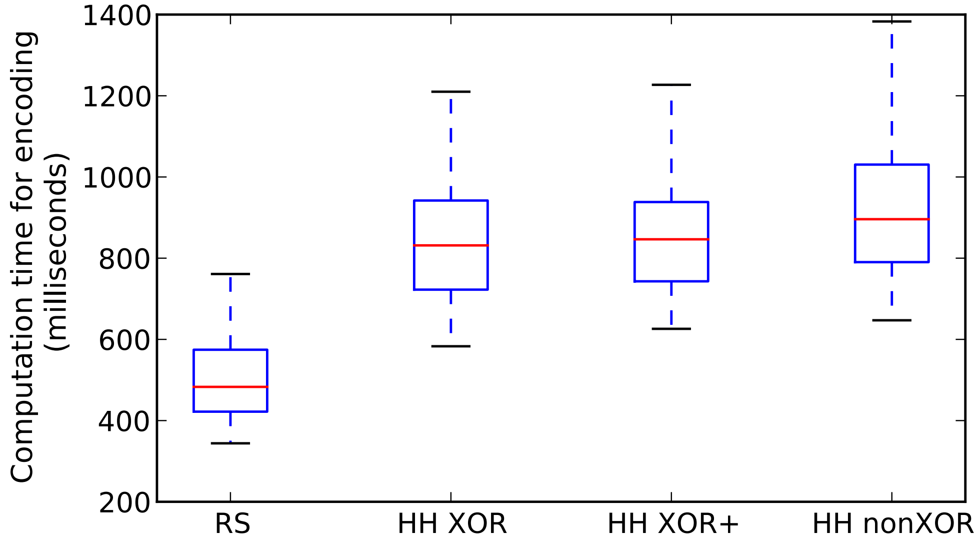


Figure 5.13: A box plot comparing the computation time for encoding of 1MB (buffer size) from 200 runs each on Facebook's data-warehouse cluster with real-time production traffic and workloads. (HH = Hitchhiker.)

Computation time for encoding

Figure 5.13 compares the computation time for the encoding operation. Hitchhiker entails higher computational overheads during encoding as compared to RS-based systems, and Hitchhiker-XOR+ and Hitchhiker-XOR are faster than the Hitchhiker-nonXOR. This is expected since Hitchhiker's encoder performs computations in addition to those of RS encoding. Section 5.8 discusses the tradeoffs between higher encoding time and savings in other metrics.

5.8 Discussion

In this section, we discuss the various tradeoffs afforded by Hitchhiker.

a) Three versions of the code: During encoding and reconstruction, Hitchhiker-XOR requires performing only XOR operations in addition to the operations of the underlying RS code. Hitchhiker-XOR+ allows for more efficient reconstruction in terms of network and disk resources in comparison to Hitchhiker-XOR, while still using only XOR operations in addition to the RS code's operations. Hitchhiker-XOR+, however, requires the underlying RS code to satisfy the all-XOR-parity property (Section 5.4). Hitchhiker-nonXOR provides the same efficiency in reconstruction as Hitchhiker-XOR+ without imposing the all-XOR-parity requirement on the RS code, but entails additional finite-field arithmetic during encoding and reconstruction.

b) Connecting to more machines during reconstruction: Reconstruction in RS-coded systems requires connecting to exactly k machines, whereas Hitchhiker entails connecting to more than k machines. In certain systems, depending on the setting at hand, this may lead to an increase in the read latency during reconstruction. However, in our experiments on the production data-warehouse cluster at Facebook, we saw no such increase in read latency. On the contrary, we consistently observed a significant reduction in the latency due to the significantly lower amounts of data required to be read and downloaded in Hitchhiker (see Figure 5.12).

c) Option of operating as an RS-based system: The storage overheads and fault tolerance of Hitchhiker are identical to RS-based systems. Moreover, a reconstruction operation in Hitchhiker can alternatively be performed as in RS-based systems by downloading any k entire blocks. Hitchhiker thus provides an option to operate as an RS-based system when necessary, e.g., when increased connectivity during reconstruction is not desired. This feature also ensures Hitchhiker's compatibility with other alternative solutions proposed outside the erasure-coding component (e.g., [20, 102, 33]).

d) Choice of hop-length: As discussed in Section 5.5, a larger hop-length leads to more contiguous reads, but requires coupling of bytes that are further apart. Recall that reconstructing any byte also necessitates reconstructing its coupled byte. In a scenario where only a part of a block may need to be reconstructed, all the bytes that are coupled with the bytes of this part must also be reconstructed even if they are not required. A lower hop-length reduces the amount such unnecessary reconstructions.

5.9 Summary

In this chapter, we presented a systematically-designed, novel storage system, called Hitchhiker, that "rides" on top of existing Reed-Solomon based erasure-coded systems utilizing the Piggybacking framework presented in Chapter 4. Hitchhiker retains the key benefits of RS-coded systems over replication-based counterparts, namely that of (i) optimal storage space needed for a targeted level of reliability, as well as (ii) complete flexibility in the design choice for the system parameters. We show how Hitchhiker can *additionally* reduce both network traffic and disk traffic by 25% to 45% over that of RS-coded systems during reconstruction of missing or otherwise unavailable data. Further, our implementation and evaluation of Hitchhiker on two HDFS clusters at Facebook also reveals savings of 36% in the computation time and 32% in the time taken to read data during reconstruction.

Hitchhiker is being incorporated into the next release of Apache Hadoop Distributed File system (Apache HDFS).

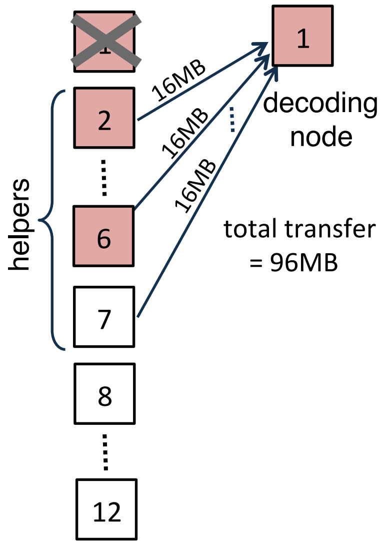
Chapter 6

Optimizing I/O and CPU along with storage and network bandwidth

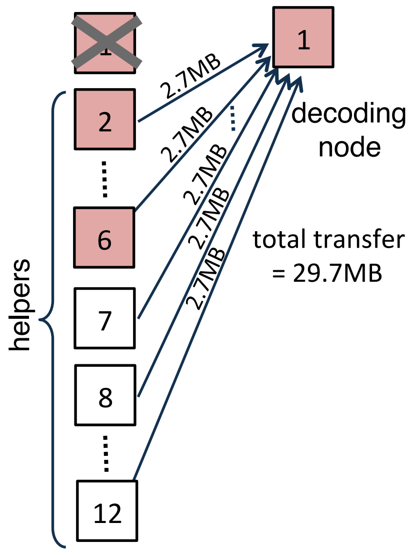
In this chapter, we present erasure codes aimed at jointly optimizing the usage of storage, network, I/O and CPU resources. First, we design erasure codes that are simultaneously optimal in terms of I/O, storage, and network usage. As a part of this design, we present a transformation that can be employed on existing classes of storage codes called *minimum-storage-regenerating codes* [41] in order to optimize their usage of I/O while retaining their optimal usage of storage and network. Through evaluations on Amazon EC2, we show that the proposed design results in a significant reduction in IOPS (I/O operations per second) during reconstructions: a 5× reduction for typical parameters. Second, we show that optimizing I/O and CPU go hand-in-hand in these resource-efficient distributed storage codes, by showing that our transformation that optimizes I/O also sparsifies the code thereby reducing the computational complexity.

6.1 Introduction

Recall from Chapter 2 that under RS codes, redundancy is introduced in the following manner: a file to be stored is divided into equal-sized units, which we will call *blocks*. Blocks are grouped into sets of k each, and for each such set of k blocks, r parity blocks are computed. The set of these $(k + r)$ blocks consisting of both the data and the parity blocks constitute a stripe. The data and parity blocks belonging to a stripe are placed on different nodes in the storage network, and these nodes are typically chosen from different racks.



(a) RS



(b) Minimum-storage-regenerating (MSR) code

Figure 6.1: Amount of data transfer involved in reconstruction of block 1 for an RS code with $k = 6$, $r = 6$ and an MSR code with $k = 6$, $r = 6$, $d = 11$, with blocks of size 16MB. The data blocks are shaded.

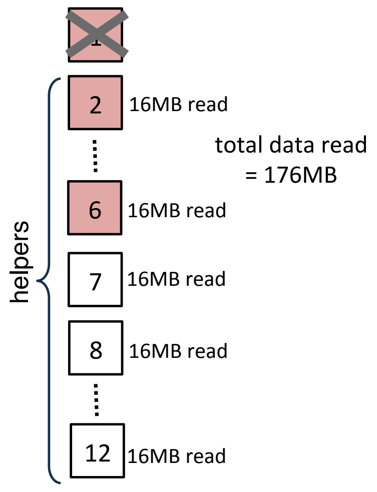
Also, recall that under traditional repair of RS codes, a missing block is replaced by downloading *all* data from (any) k other blocks in the stripe and decoding the desired data from it. Let us look at an example. Consider an RS code with $k = 6$ and $r = 6$. While this code has a storage overhead of 2×, it offers orders of magnitude higher reliability than 3× replication. Figure 6.1a depicts a stripe of this code, and also illustrates the reconstruction of block 1 using the data from blocks 2 to 7. Here, blocks 2 to 7 are the helpers. In this example, in order to reconstruct a 16 MB block, $6 \times 16MB = 96$ MB of data is read from disk at the helpers and transferred across the network to the node performing the decoding computations. In general, the disk read and the network transfer overheads during a reconstruction operation is k *times* that under replication. Consequently, under RS codes, reconstruction operations result in a large amount of disk I/O and network transfers, putting a huge burden on these system resources.

Minimum-storage-regenerating (MSR) codes [41] are a recently proposed framework for distributed storage codes that optimize storage and network-bandwidth usage during reconstruction operations. Under an MSR code, an unavailable block is reconstructed by downloading a small fraction of the data from any d $(> k)$ blocks in the stripe, in a manner that the total amount of data transferred during reconstruction is lower than that in RS codes. Let us consider an example with $k = 6$, $r = 6$ and $d = 11$. For these parameters, reconstruction of block 1 under an MSR code is illustrated in Figure 6.1b. In this example, the total network transfer is only 29.7 MB as opposed to 96 MB under RS. MSR codes are optimal with respect to storage and network transfers: the storage capacity and reliability is identical to that under RS codes, while the network transfer is significantly lower than that under RS and in fact, is the minimum possible under any code. However, in general, MSR codes do not optimize the usage of I/Os. The I/O overhead during a reconstruction operation in a system employing an MSR code is, in general, higher than that in a system employing RS code. This is illustrated in Figure 6.2a which depicts the amount data read from disks for reconstruction of block 1 under a practical construction of MSR codes called product-matrix-MSR (PM-MSR) codes. This entails reading 16 MB at each of the 11 helpers, totaling 176 MB of data read. While under RS codes, the amount of data read is only 96 MB.

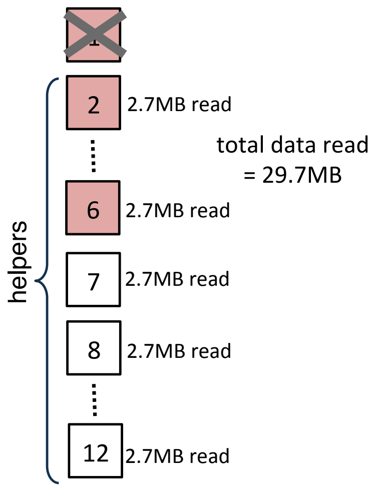
I/Os are a valuable resource in storage systems. With the increasing speeds of newer generation network interconnects and the increasing storage capacities of individual storage devices, I/O is becoming the primary bottleneck in the performance of disk-based storage systems. Moreover, many applications that the storage systems serve today are I/O bound, for example, applications that serve a large number of user requests [18]. Motivated by the increasing importance of I/O, we investigate practical erasure codes for storage systems that are also I/O optimal along with storage and network-bandwidth optimality.

Furthermore, one of the most frequent operations performed in distributed storage systems is encoding of the new data entering the system. The CPU cost of encoding is a non-issue when using replication, but can be significant when using erasure codes. This can also slow down the data ingestion process. Thus it is desirable for the storage code to possess a low-complexity, fast encoding operation. For any linear code, the encoding operation can be represented as a matrix-vector multiplication between a matrix termed as the *generator matrix* of the code and a vector consisting of data symbols to be encoded [100]. This encoding operation will be fast and less CPU intensive if the generator matrix is sparse, since this reduces the number of computations to be performed.

In this chapter, we consider the problem of designing practical erasure codes that are simultaneously optimal in terms of I/O, storage, and network bandwidth while also supporting fast encoding. To this end, we first identify two properties that aid in transforming MSR codes to be disk-read optimal during reconstruction while retaining their storage and network optimality. We show that a class of powerful practical constructions for MSR codes, the product-matrix-MSR codes (PM-MSR) [127], indeed satisfy these desired properties. We then present an algorithm to transform *any* MSR code satisfying these properties into a code that is optimal in terms of the amount of data read from disks. We apply our transformation



(a) PM-MSR



(b) PM-RBT

Figure 6.2: Amount of data read from disks during reconstruction of block 1 for $k = 6$, $r = 6$, $d = 11$ with blocks of size 16 MB.

to PM-MSR codes and call the resulting I/O optimal codes as PM-RBT codes. Figure 6.2b depicts the amount of data read for reconstruction of block 1 for the example parameters: PM-RBT entails reading only 2.7 MB at each of the 11 helpers, totaling 29.7 MB of data read as opposed to 176 MB under PM-MSR. We note that the PM-MSR codes operate in the regime $r \geq k - 1$, and consequently the PM-RBT codes also operate in this regime.

We implement PM-RBT codes in C/C++, and show through experiments on Amazon EC2 instances that our approach results in a $5\times$ reduction in I/Os consumed during reconstruction as compared to the original product-matrix codes, for a typical set of parameters. For general parameters, the number of I/Os consumed would reduce approximately by a factor of $(d - k + 1)$. For typical values of d and k, this can result in substantial gains. We then show that if the relative frequencies of reconstruction of blocks are different, a more holistic system-level design of helper assignments is needed to optimize I/Os across the entire system. Such situations are common: for instance, in a system that deals with degraded reads as well as node failures, the data blocks will be reconstructed more frequently than the parity blocks. We pose this problem as an optimization problem and present an algorithm to obtain the optimal solution to this problem. We evaluate our helper assignment algorithm through simulations using data from experiments on Amazon EC2 instances.

We then establish a general connection between optimizing I/O and reducing the encoding complexity, by showing that a specific transformation that we present for I/O optimization in MSR codes leads to sparsity in the generator matrix of the code. Through evaluations on

Amazon EC2 using PM-MSR codes, we show that such a transformation leads to approximately $k\times$ reduction in the encoding complexity and thus a $k\times$ speed up in the computation time of the encoding process.

PM-RBT codes presented in this chapter provide higher savings in I/O and network bandwidth as compared to Piggybacked-RS codes presented in Chapter 5. On the other hand, Piggybacked-RS codes have the advantage of being applicable for all values of k and r, whereas PM-RBT codes are only applicable for $d \geq (2k - 2)$ and thereby necessitate a storage overhead of atleast $(2 - \frac{1}{k})$.

6.2 Related work

In [71], the authors build a file system based on the minimum-bandwidth-regenerating (MBR) code constructions of [149]. While this system minimizes network transfers and the amount of data read during reconstruction, it mandates additional storage capacity to achieve the same. That is, the system is not optimal with respect to storage-reliability tradeoff. The storage systems proposed in [73, 146, 49] employ a class of codes called local-repair codes which optimize the number of blocks accessed during reconstruction. This, in turn, also reduces the amount of disk reads and network transfers. However, these systems also necessitate an increase in storage-space requirements in the form of at least 25% to 50% additional parities. In [92], authors present a system which combines local-repair codes with the graph-based MBR codes presented in [149]. This work also necessitates additional storage space. The goal of the present chapter is to optimize I/Os consumed during reconstruction *without* losing the optimality with respect to storage-reliability tradeoff.

In [72] and [11], the authors present storage systems based on random network-coding that optimize resources consumed during reconstruction. Here the data that is reconstructed is not identical and is only "functionally equivalent" to the failed data. As a consequence, the system is not systematic, and needs to execute the decoding procedure for serving every read request. The present chapter designs codes that are systematic, allowing read requests during the normal mode of operation to be served directly without executing the decoding procedure.

In [150], the authors consider the theory behind reconstruction-by-transfer for MBR codes, which as discussed earlier are not optimal with respect to storage-reliability tradeoff. Some of the techniques employed in the current chapter are inspired by the techniques introduced in [150]. In [178] and [183], the authors present optimizations to reduce the amount of data read for reconstruction in array codes with two parities. The code constructions provided in [44, 119, 91, 160] optimize both I/O and network bandwidth during reconstruction. However, these codes are not MDS and thus necessitate additional storage overhead.

[89] presents a search-based approach to find reconstruction symbols that optimize I/O for arbitrary binary erasure codes, but this search problem is shown to be NP-hard. In [89], authors also present Rotated-RS codes which reduce the amount of data read during rebuilding. However, the reduction achieved is significantly lower than that in PM-RBT.

Several works (e.g., [20, 102, 33]) have proposed system-level solutions to reduce network and I/O consumption for reconstruction, such as caching the data read during reconstruction, batching multiple reconstruction operations, and delaying the reconstruction operations. While these solutions consider the erasure code as a black-box, our work optimizes this black-box and can be used in conjunction with these system-level solutions.

The computational complexity of the encoding operation can be reduced by decreasing the field size over which the code is constructed as well as by making the generator matrix of the code sparse. There have been a number of recent works [47, 69, 137] on constructing MSR codes with small field size requirement. In this chapter, we look at sparsifying the generator matrix of MSR codes.

6.3 Background

Notation and terminology

We will refer the smallest granularity of the data in the system as a *symbol*. The actual size of a symbol is dependent on the finite-field arithmetic that is employed. For simplicity, the reader may consider a symbol to be a single byte. A vector will be column vector by default. We will use boldface to denote column vectors, and use T to denote a matrix or vector transpose operation.

We will now introduce some more terminology in addition to that introduced in Section 6.1. This terminology is illustrated in Figure 6.3. Let n $(= k + r)$ denote the total number of blocks in a stripe. In order to encode the k data blocks in a stripe, each of the k data blocks are first divided into smaller units consisting of α symbols each. A set of α symbols from each of the k blocks is encoded to obtain the corresponding set of α symbols in the parity blocks. We call the set of data and parities at the granularity of α symbols as a *byte-level* stripe. Thus in a byte-level stripe, $B = k\alpha$ original data symbols (α symbols from each of the k original data blocks) are encoded to generate $n\alpha$ symbols (α symbols for each of the n encoded blocks). The data symbols in different byte-level stripes are encoded

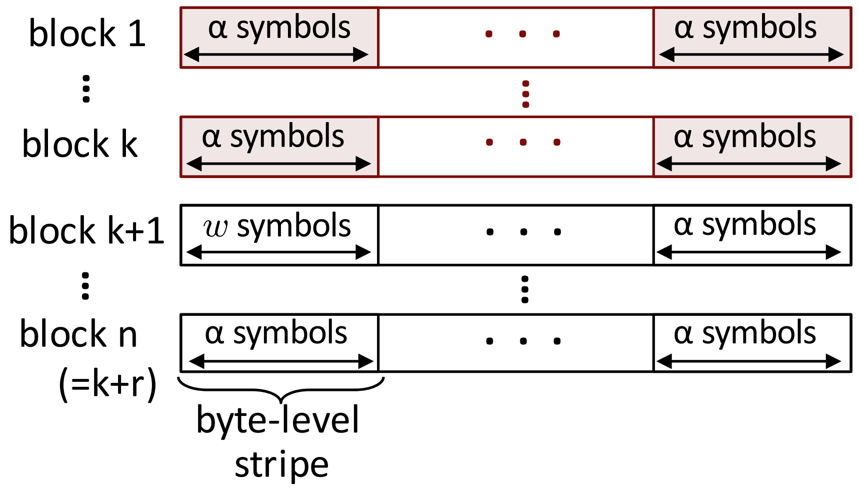


Figure 6.3: Illustration of notation: hierarchy of symbols, byte-level stripes and block-level stripe. The first k blocks shown shaded are systematic.

independently in an identical fashion. Hence in the rest of the chapter, for simplicity of exposition, we will assume that each block consists of a single byte-level stripe. We denote the B original data symbols as $\{m_1, \ldots, m_B\}$. For $i \in \{1, \ldots, n\}$, we denote the α symbols stored in block i by $\{s_{i1}, \ldots, s_{i\alpha}\}$. The value of α is called the *number of substripes.*

During reconstruction of a block, the other blocks from which data is accessed are termed the *helpers* for that reconstruction operation (see Figure 6.1). The accessed data is transferred to a node that performs the decoding operation on this data in order to recover the desired data. This node is termed the *decoding node.*

Linear and systematic codes

A code is said to be *linear*, if all operations including encoding, decoding and reconstruction can be performed using linear operations over the finite field. Any linear code can be represented using a $(nw \times B)$ matrix G, called its *generator matrix.* The nw encoded symbols can be obtained by multiplying the generator matrix with a vector consisting of the B message symbols:

$$G \begin{bmatrix} m_1 \; m_2 \; \cdots \; m_B \end{bmatrix}^T . \tag{6.1}$$

We will consider only linear codes in this chapter.

In most systems, the codes employed have the property that the original data is available in unencoded (i.e., raw) form in some k of the n blocks. This property is appealing since it allows for read requests to be served directly without having to perform any decoding operations. Codes that possess this property are called *s*ystematic codes. We will assume without loss of generality that the first k blocks out of the n encoded blocks store the unencoded data. These blocks are called *systematic* blocks. The remaining $(r = n - k)$ blocks store the encoded data and are called *parity* blocks. For a linear systematic code, the generator matrix can be written as

$$\begin{bmatrix} I \\ \hat{G} \end{bmatrix}, \tag{6.2}$$

where I is a $(B \times B)$ identity matrix, and $\hat{G}$ is a $((nw - B) \times B)$ matrix. The codes presented in this chapter are systematic.

Optimality of storage codes

A storage code is said to be optimal with respect to the storage-reliability tradeoff if it offers maximum fault tolerance for the storage overhead consumed. Recall from Chapter 2 that MDS codes are optimal with respect to the storage-reliability tradeoff. RS codes are MDS and hence RS codes are optimal with respect to the storage-reliability tradeoff.

In [41], the authors introduced another dimension of optimality for storage codes, that of reconstruction bandwidth, by providing a lower bound on the amount of data that needs to be transferred during a reconstruction operation. Codes that meet this lower bound are termed optimal with respect to storage-bandwidth tradeoff.

Minimum Storage Regenerating codes

In addition to the parameters k, r, and α, a minimum-storage-regenrating (MSR) code [41] is associated with two other parameters d and β. The parameter $d\,(> k)$ refers to the number of helpers used during a reconstruction operation, and the parameter β refers to the number of symbols downloaded from each helper. We will refer to such an MSR code as $[n, k, d](\alpha, \beta)$ MSR code. Throughout this chapter, we consider MSR codes with $\beta = 1$. For an $[n, k, d](\alpha, \beta = 1)$ MSR code, the number of substripes α is given by $\alpha = d - k + 1$. Thus, each byte-level stripe stores $B = k\alpha = k(d - k + 1)$ original data symbols.

We now describe the reconstruction process under the framework of MSR codes. A block to be reconstructed can choose *any* d other blocks as helpers from the remaining $(n-1)$ blocks in the stripe. Each of these d helpers compute some function of the α symbols stored whose resultant is a single symbol (for each byte-level stripe). Note that the symbol computed and transferred by a helper may, in general, depend on the choice of $(d-1)$ other helpers.

Consider the reconstruction of block f. Let D denote that set of d helpers participating in this reconstruction operation. We denote the symbol that a helper block h transfers to aid in the reconstruction of block f when the set of helpers is denoted by D as t_{hfD}. This resulting symbol is transferred to the decoding node, and the d symbols received from the helpers are used to reconstruct block f.

Like RS codes, MSR codes are optimal with respect to the storage-reliability tradeoff. Furthermore, they meet the lower bound on the amount of data transfer for reconstruction, and hence are optimal with respect to storage-bandwidth tradeoff as well.

Product-Matrix-MSR Codes

Product-matrix-MSR codes are a class of practical constructions for MSR codes that were proposed in [127]. These codes are linear. We consider the systematic version of these codes where the first k blocks store the data in an unencoded form. We will refer to these codes as *PM-vanilla* codes.

PM-vanilla codes exist for all values of the system parameters k and d satisfying $d \geq (2k-2)$. In order to ensure that, there are atleast d blocks that can act as helpers during reconstruction of any block, we need atleast $(d+1)$ blocks in a stripe, i.e., we need $n \geq (d+1)$. It follows that PM-vanilla codes need a storage overhead of

$$\frac{n}{k} \geq \left(\frac{2k-1}{k}\right) = 2 - \frac{1}{k}. \tag{6.3}$$

We now briefly describe the reconstruction operation in PM-vanilla codes to the extent that is required for the exposition of this chapter. We refer the interested reader to [127] for more details on how encoding and decoding operations are performed. Every block i $(1 \leq i \leq n)$ is assigned a vector $\mathbf{g_i}$ of length α, which we will call the *reconstruction vector* for block i. Let $\mathbf{g_i} = [g_{i1}, \ldots, g_{i\alpha}]^T$. The n $(= k+r)$ vectors, $\{\mathbf{g_1}, \ldots, \mathbf{g_n}\}$, are designed such that any α of these n vectors are linearly independent.

During reconstruction of a block, say block f, each of the chosen helpers, take a linear combination of their α stored symbols with the reconstruction vector of the failed block, $\mathbf{g_f}$, and transfer the result to the decoding node. That is, for reconstruction of block f, helper block h computes and transfers the symbol

$$t_{hfD} = \sum_{j=1}^{\alpha} s_{hj} g_{fj} \ , \tag{6.4}$$

where $\{s_{h1}, \ldots, s_{h\alpha}\}$ are the α symbols stored in block h, and D denotes the set of helpers.

PM-vanilla codes are optimal with respect to the storage-reliability tradeoff and storage-bandwidth tradeoff. However, PM-vanilla codes are not optimal with respect to the amount of data read during reconstruction: The values of most coefficients g_{fj} in the reconstruction vector are non-zero. Since the corresponding *symbol* s_{hj} must be read for every g_{fj} that is non-zero, the absence of sparsity in g_{fj} results in a large I/O overhead during the rebuilding process, as illustrated in Figure 6.2a (and experimentally evaluated in Figure 6.7).

6.4 Optimizing I/O during reconstruction

We will now employ the PM-vanilla codes to construct codes that optimize I/Os during reconstruction, while retaining optimality with respect to storage, reliability and network-bandwidth. In this section, we will optimize the I/Os *locally* in individual blocks, and Section 6.5 will build on these results to design an algorithm to optimize the I/Os *globally* across the entire system. The resulting codes are termed the *PM-RBT* codes. We note that the methods described here are more broadly applicable to other MSR codes as discussed subsequently.

Reconstruct-by-transfer

Under an MSR code, during a reconstruction operation, a helper is said to perform *reconstruct-by-transfer* (RBT) if it does not perform any computation and merely transfers one its stored symbols (per byte-level stripe) to the decoding node.[1] In the notation introduced in Section 6.3, this implies that $\mathbf{g_f}$ in (6.4) is a unit vector, and

$$t_{hfD} \in \{s_{h1}, \ldots, s_{h\alpha}\} \ .$$

We call such a helper as an *RBT-helper*. At an RBT-helper, the amount of data read from the disks is *equal* to the amount transferred through the network.

During a reconstruction operation, a helper reads requisite data from the disks, computes (if required) the desired function, and transfers the result to the decoding node. It follows that the amount of network transfer performed during reconstruction forms a lower bound on the amount of data read from the disk at the helpers. Thus, a lower bound on the

[1]This property was originally titled 'repair-by-transfer' in [149] since the focus of that paper was primarily on node failures. We consider more general reconstruction operations that include node-repair, degraded reads etc., and hence the slight change in nomenclature.

network transfers is also a lower bound on the amount of data read. On the other hand, MSR codes are optimal with respect to network transfers during reconstruction since they meet the associated lower bound [41]. It follows that, under an MSR code, an RBT-helper is optimal with respect to the amount of data read from the disk.

We now present our technique for achieving the reconstruct-by-transfer property in MSR codes.

Achieving reconstruct-by-transfer

Towards the goal of designing reconstruct-by-transfer codes, we first identify two properties that we would like a helper to satisfy. We will then provide an algorithm to convert any (linear) MSR code satisfying these two properties into one that can perform reconstruct-by-transfer at such a helper.

Property 1*:* The function computed at a helper is independent of the choice of the remaining $(d-1)$ helpers. In other words, for any choice of h and f, t_{hfD} is independent of D (recall the notation t_{hfD} from Section 6.3).

This allows us to simplify the notation by dropping the dependence on D and referring to t_{hfD} simply as t_{hf}.

Property 2*:* Assume Property 1 is satisfied. Then the helper would take $(n-1)$ linear combinations of its own data to transmit for the reconstruction of the other $(n-1)$ blocks in the stripe. We want every α of these $(n-1)$ linear combinations to be linearly independent.

We now show that under the product-matrix-MSR (PM-vanilla) codes, every helper satisfies the two properties enumerated above. Recall from Equation (6.4), the computation performed at the helpers during reconstruction in PM-vanilla codes. Observe that the right hand side of Equation (6.4) is independent of 'D', and therefore the data that a helper transfers during a reconstruction operation is dependent only on the identity of the helper and the block being reconstructed. The helper, therefore, does not need to know the identity of the other helpers. It follows that PM-vanilla codes satisfy Property 1.

Let us now investigate Property 2. Recall from Equation (6.4), the set of $(n-1)$ symbols, $\{t_{h1}, \ldots, t_{h(h-1)}, t_{h(h+1)}, \ldots, t_{hn}\}$, that a helper block h transfers to aid in reconstruction of each the other $(n-1)$ blocks in the stripe. Also, recall that the reconstruction vectors $\{\mathbf{g}_1, \ldots, \mathbf{g}_n\}$ assigned to the n blocks are chosen such that every α of these vectors are linearly independent. It follows that for every block, the $(n-1)$ linear combinations that it computes and transfers for the reconstruction of the other $(n-1)$ blocks in the stripe have the property of any α being independent. PM-vanilla codes thus satisfy Property 2 as well.

PM-vanilla codes are optimal with respect to the storage-bandwidth tradeoff (Section 6.3). However, these codes are not optimized in terms of I/O. As we will show through experiments on the Amazon EC2 instances (Section 6.7), PM-vanilla codes, in fact, have a higher I/O overhead as compared to RS codes. In this section, we will make use of the two properties listed above to transform the PM-vanilla codes into being I/O optimal for reconstruction, while retaining its properties of being storage and network optimal. While we focus on the PM-vanilla codes for concreteness, we remark that the technique described is generic and can be applied to any (linear) MSR code satisfying the two properties listed above.

Algorithm 1 Algorithm to achieve reconstruct-by-transfer at helpers

```
Encode the data in $k$ data blocks using PM-vanilla code to obtain the $n$ encoded blocks
for every block $h$ in the set of $n$ blocks
    Let $\{i_{h1}, \ldots, i_{h\alpha}\}$ denote the set of $\alpha$ blocks that block $h$ will help to reconstruct by transfer
    Let $\{s_{h1}, \ldots, s_{h\alpha}\}$ denote the set of $\alpha$ symbols that block $h$ stores under $PM - vanilla$
    Compute $[s_{h1} \ \cdots \ s_{h\alpha}] \, [\mathbf{g}_{i_{h1}} \ \cdots \ \mathbf{g}_{i_{h\alpha}}]$ and store the resulting $\alpha$ symbols in block $h$ instead of the original $\alpha$ symbols
```

Under our algorithm, each block will function as an RBT-helper for some α other blocks in the stripe. For the time being, let us assume that for each helper block, the choice of these α blocks is given to us. Under this assumption, Algorithm 1 outlines the procedure to convert the PM-vanilla code (or in general any linear MSR code satisfying the two aforementioned properties) into one in which every block can function as an RBT-helper for α other blocks. Section 6.5 will subsequently provide an algorithm to make the choice of RBT-helpers for each block to optimize the I/O cost across the entire system.

Let us now analyze Algorithm 1. Observe that each block still stores α symbols and hence Algorithm 1 does not increase the storage requirements. Further, recall from Section 6.3 that the reconstruction vectors $\{\mathbf{g}_{i_{h1}}, \cdots, \mathbf{g}_{i_{h\alpha}}\}$ are linearly independent. Hence the transformation performed in Algorithm 1 is an invertible transformation within each block. Thus the property of being able to recover all the data from any k blocks continues to hold as under PM-vanilla codes, and the transformed code retains the storage-reliability optimality.

Let us now look at the reconstruction process in the transformed code given by Algorithm 1. The symbol transferred by any helper block h for the reconstruction of any block f remains identical to that under the PM-vanilla code, i.e., is as given by the right hand side of Equation (6.4). Since the transformation performed in Algorithm 1 is invertible within each block, such a reconstruction is always possible and entails the minimum network transfers. Thus, the code retains the storage-bandwidth optimality as well. Observe that for a block f in the set of the α blocks for which a block h intends to function as an RBT-helper, block h now directly stores the symbol $t_{hf} = [s_{h1} \; \cdots \; s_{h\alpha}]\mathbf{g}_f$. As a result, whenever called upon to help block f, block h can directly read and transfer this symbol, thus performing a reconstruct-by-transfer operation. As discussed in Section 6.4, by virtue of its storage-bandwidth optimality and reconstruct-by-transfer, the transformed code from Algorithm 1 (locally) optimizes the amount of data read from disks at the helpers. We will consider optimizing I/O across the entire system in Section 6.5.

Making the code systematic

Transforming the PM-vanilla code using Algorithm 1 may result in a loss of the systematic property. A further transformation of the code, termed 'symbol remapping', is required to make the transformed code systematic. Symbol remapping [127, Theorem 1] involves transforming the original data symbols $\{m_1, \ldots, m_B\}$ using a bijective transformation before applying Algorithm 1, as described below.

Since every step of Algorithm 1 is linear, the encoding under Algorithm 1 can be represented by a generator matrix, say G_{Alg1}, of dimension $(n\alpha \times B)$ and the encoding can be performed by the matrix-vector multiplication: $G_{Alg1} \left[\; m_1 \; m_2 \; \cdots \; m_B \; \right]^T$. Partition G_{Alg1} as

$$G_{Alg1} = \begin{bmatrix} G_1 \\ G_2 \end{bmatrix} , \tag{6.5}$$

where G_1 is a $(B \times B)$ matrix corresponding to the encoded symbols in the first k systematic blocks, and G_2 is an $((n\alpha - B) \times B$ matrix. The symbol remapping step to make the transformed code systematic involves multiplication by G_1^{-1}. The invertibility of G_1 follows from the fact that G_1 corresponds to the encoded symbols in the first k blocks and all the encoded symbols in any set of k blocks are linearly independent. Thus the entire encoding process becomes

$$\begin{bmatrix} G_1 \\ G_2 \end{bmatrix} G_1^{-1} \begin{bmatrix} m_1 \\ m_2 \\ \vdots \\ m_B \end{bmatrix} = \begin{bmatrix} I \\ G_2 G_1^{-1} \end{bmatrix} \begin{bmatrix} m_1 \\ m_2 \\ \vdots \\ m_B \end{bmatrix} , \tag{6.6}$$

where I is the $(B \times B)$ identity matrix. We can see that the symbol remapping step followed by Algorithm 1 makes the first k blocks systematic.

Since the transformation involved in the symbol remapping step is invertible and is applied to the data symbols before the encoding process, this step does not affect the performance with respect to storage, reliability, and network and I/O consumption during reconstruction.

Making reads sequential

Optimizing the amount of data read from disks might not directly correspond to optimized I/Os, unless the data read is *sequential*. In the code obtained from Algorithm 1, an RBT-helper reads one symbol per byte-level stripe during a reconstruction operation. Thus, if one chooses the α symbols belonging to a byte-level stripe within a block in a contiguous manner, as in Figure 6.3, the data read at the helpers during reconstruction operations will be fragmented. In order to the read sequential, we employ the hop-and-couple technique introduced in Chapter 5. Recall that the basic idea behind this technique is to choose symbols that are farther apart within a block to form the byte-level stripes. If the number of substripes of the code is α, then choosing symbols that are a $\frac{1}{\alpha}$ fraction of the block size away will make the read at the helpers during reconstruction operations sequential. Note that this technique does not affect the natural sequence of the raw data in the data blocks, so the normal read operations can be served directly without any sorting. In this manner, we ensure that reconstruct-by-transfer optimizes I/O at the helpers along with the amount of data read from the disks.

6.5 Optimizing RBT-helper assignment

In Algorithm 1 presented in Section 6.4, we assumed the choice of the RBT-helpers for each block to be given to us. Under any such given choice, we saw how to perform a *local* transformation at each block such that reconstruction-by-transfer could be realized under that assignment. In this section, we present an algorithm to make this choice such that the I/Os consumed during reconstruction operations is optimized *globally* across the entire system. Before going into any details, we first make an observation which will motivate our approach.

A reconstruction operation may be instantiated in any one of the following two scenarios:

- Failures: When a node fails, the reconstruction operation restores the contents of that node in another storage nodes in order to maintain the system reliability and availability. Failures may entail reconstruction operations of either systematic or parity blocks.

Algorithm 2 Algorithm for optimizing RBT-helper assignment

```
//To compute number of RBT-helpers for each block
Set num_rbt_helpers[block] = 0 for every block
for total_rbt_help = $n\alpha$ to 1
  for block in all blocks
    if num_rbt_helpers[block] ¡ n-1
      Set improvement[block] = Cost(num_rbt_helpers[block]) - Cost(num_rbt_helpers[block
]+1)
    else
      Set improvement[block] = -1
  Let max_improvement be the set of blocks with the maximum value of improvement
  Let this_block be a block in max_improvement with the largest value of num_rbt_helpers
  Set num_rbt_helpers[this_block] = num_rbt_helpers[this_block]+1

//To select the RBT-helpers for each block
Call the Kleitman-Wang algorithm [90] to generate a digraph on $n$ vertices with incoming
degrees num_rbt_helpers and all outgoing degrees equal to $\alpha$
for every edge $i \to j$ in the digraph
  Set block $i$ as an RBT-helper to block $j$
```

- *D*egraded reads: When a read request arrives, the systematic block storing the requested data may be busy or unavailable. The request is then served by calling upon a reconstruction operation to recover the desired data from the remaining blocks. This is called a degraded read. Degraded reads entail reconstruction of only the systematic blocks.

A system may be required to support either one or both of these scenarios, and as a result, the importance associated to the reconstruction of a systematic block may often be higher than the importance associated to the reconstruction of a parity block.

We now present a simple yet general model that we will use to optimize I/Os holistically across the system. The model has two parameters, δ and p. The relative importance between systematic and parity blocks is captured by the first parameter δ. The parameter δ takes a value between (and including) 0 and 1, and the cost associated with the reconstruction of any parity block is assumed to be δ times the cost associated to the reconstruction of any systematic block. The "cost" can be used to capture the relative difference in the frequency of reconstruction operations between the parity and systematic blocks or the preferential treatment that one may wish to confer to the reconstruction of systematic blocks in order to serve degraded reads faster.

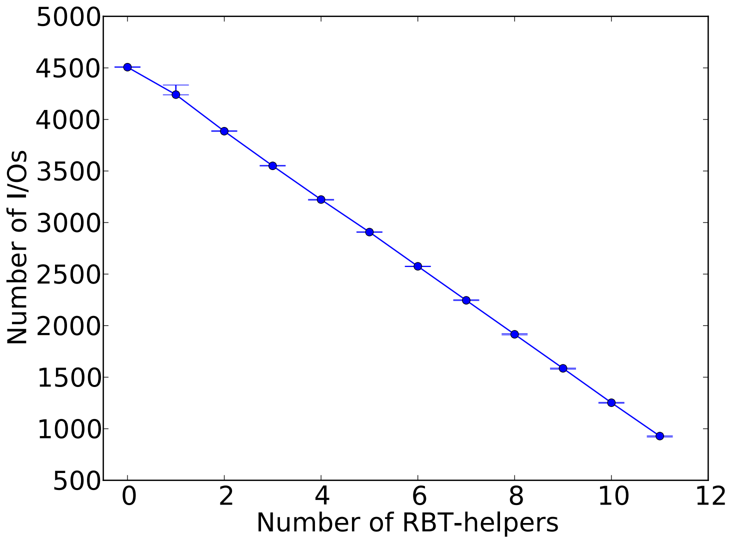
When reconstruction of any block is to be carried out, either for repairing a possible failure or for a degraded read, not all remaining blocks may be available to help in the reconstruction process. The parameter p ($0 \leq p \leq 1$) aims to capture this fact: when the reconstruction of a block is to be performed, every other block may individually be unavailable with a probability p independent of all other blocks. Our intention here is to capture the fact that if a block has certain number of helpers that can function as RBT-helpers, not all of them may be available when reconstruction is to be performed.

We performed experiments on Amazon EC2 measuring the number of I/Os performed for reconstruction when precisely j ($0 \leq j \leq d$) of the available helpers are RBT-helpers and the remaining $(d - j)$ helpers are non-RBT-helpers. The non-RBT-helpers do not perform reconstruct-by-transfer, and are hence optimal with respect to network transfers but not the I/Os. The result of this experiment aggregated from 20 runs is shown in Figure 6.4a, where we see that the number of I/Os consumed reduces linearly with an increase in j. We also measured the maximum of the time taken by the d helper blocks to complete the requisite I/O, which is shown Figure 6.4b. Observe that as long as $j < d$, this time decays very slowly upon increase in j, but reduces by a large value when j crosses d. The reason for this behavior is that the time taken by the non-RBT-helpers to complete the required I/O is similar, but is much larger than the time taken by the RBT-helpers.

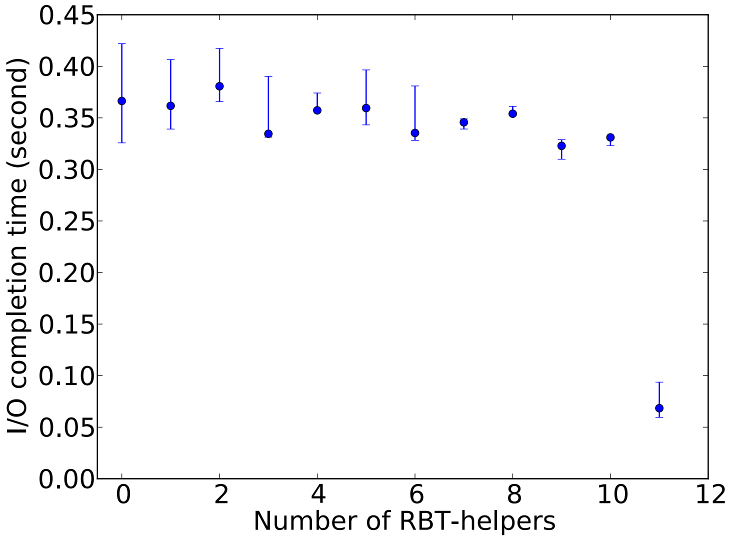
Algorithm 2 takes the two parameters δ and p as inputs and assigns RBT-helpers to all the blocks in the stripe. The algorithm optimizes the expected cost of I/O for reconstruction across the system, and furthermore subject to this minimum cost, minimizes the expected time for reconstruction. The algorithm takes a greedy approach in deciding the *number* of RBT-helpers for each block. Observing that, under the code obtained from Algorithm 1, each block can function as an RBT-helper for at most α other blocks, the total RBT-helping capacity in the system is $n\alpha$. This total capacity is partitioned among the n blocks as follows. The allocation of each unit of RBT-helping capacity is made to the block whose expected reconstruction cost will reduce the most with the help of this additional RBT-helper. The expected reconstruction cost for any block, under a given number of RBT-helper blocks, can be easily computed using the parameter p; the cost of a parity block is further multiplied by δ. Once the number of RBT-helpers for each block is obtained as above, all that remains is to make the choice of the RBT-helpers for each block. In making this choice, the only constraints to be satisfied are the number of RBT-helpers for each block as determined above and that no block can help itself. The Kleitman-Wang algorithm [90] facilitates such a construction.

The following theorem provides rigorous guarantees on the performance of Algorithm 2.

Theorem 1. *For any given (δ, p), Algorithm 2 minimizes the expected amount of disk reads*



(a) Total number of I/Os consumed



(b) Maximum of the I/O completion times at helpers

Figure 6.4: Reconstruction under different number of RBT-helpers for $k = 6$, $d = 11$, and a block size of 16 MB.

for reconstruction operations in the system. Moreover, among all options resulting in this minimum disk read, the algorithm further chooses the option which minimizes the expected time of reconstruction.

The proof proceeds by first showing that the expected reconstruction cost of any particular block is convex in the number of RBT-helpers assigned to it. It then employs this convexity, along with the fact that the expected cost must be non-increasing in the number of assigned RBT-helpers, to show that no other assignment algorithm can yield a lower expected cost. We omit the complete proof of the theorem due to space constraints.

The output of Algorithm 2 for $n = 15$, $k = 6$, $d = 11$ and $(\delta = 0.25, p = 0.03)$ is illustrated in Fig 6.5. Blocks $1, \ldots, 6$ are systematic and the rest are parity blocks. Here, Algorithm 2 assigns 12 RBT-helpers to each of the systematic blocks, and 11 and 7 RBT-helpers to the first and second parity blocks respectively.

The two 'extremities' of the output of Algorithm 2 form two interesting special cases:

1. Systematic (SYS): All blocks function as RBT-helpers for the first $\min(k, \alpha)$ systematic blocks.
2. Cyclic (CYC): Block $i \in \{1, \ldots, n\}$ functions as an RBT-helper for blocks $\{i+1, \ldots, i+\alpha\} \bmod n$.

Algorithm 2 will output the SYS pattern if, for example, reconstruction of parity blocks incur negligible cost (δ is close to 0) or if $\delta < 1$ and p is large. Algorithm 2 will output the CYC pattern if, for instance, the systematic and the parity blocks are treated on equal footing (δ is close to 1), or in low churn systems where p is close to 0.

While Theorem 1 provides mathematical guarantees on the performance of Algorithm 2, Section 6.7 will present an evaluation of its performance via simulations using data from Amazon EC2 experiments.

6.6 Reducing computational complexity of encoding

In this section, we show how transforming MSR codes to achieve repair-by-transfer also makes their generator matrix sparse, thereby reducing the computational complexity (and hence CPU usage) of encoding and also making the encoding process faster.

Let $\mathcal{C}$ be a linear systematic MSR $[n, k, d](\alpha, \beta = 1)$, with $(n\alpha \times B)$ generator matrix G. Let $G^{(i)}$ be the $(\alpha \times B)$ submatrix of the generator matrix G corresponding to the i-th node.

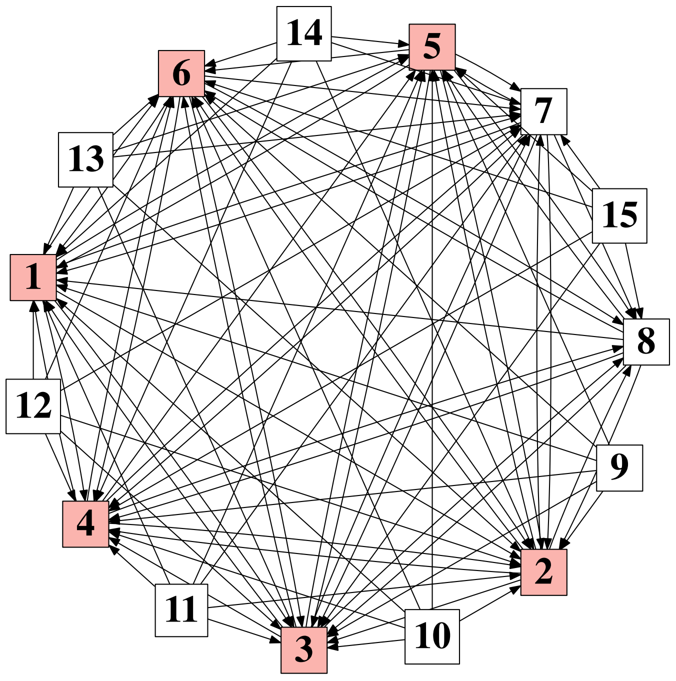


Figure 6.5: The output of Algorithm 2 for the parameters $n = 15$, $k = 6$, $d = 11$, and $(\delta = 0.25, p = 0.03)$ depicting the assignment of RBT-helpers. The directed edges from a block indicate the set of blocks that it helps to reconstruct-by-transfer. Systematic blocks are shaded.

Theorem 2. *If $\mathcal{C}$ supports repair of a systematic node ν via RBT with helper nodes comprising the remaining $(k-1)$ systematic nodes and $d-(k-1)=\alpha$ other parity nodes, then for each parity i, the corresponding generator-submatrix $G^{(i)}$ has a row with at most d non-zero entries.*

Proof. Say systematic node 0 fails, and is repaired via RBT by the $(k-1)$ other systematic nodes, and α other parity nodes. Each helper will send one of its α stored symbols. For the systematic helpers, these symbols correspond directly to message symbols – let S be the set of these message symbol indices. Notice that S is disjoint from the symbols that node 0 stores. For the parity helpers, each transferred symbol is a linear combination of message symbols. We claim that these linear combinations cannot be supported on more than the message symbols that node 0 stores, and the set S. That is, in total the support size can be at most $\alpha + (k-1) = d$.

Intuitively, Theorem 2 holds because the symbols from systematic helpers can only "cancel interference" in $(k-1)$ coordinates (of S), and the α parity helpers must allow the repair of node 0's α coordinates, and thus cannot contain more interference. This concept of interference-alignment is made precise in [156], and our Theorem 2 follows from "Property 2 (Necessity of Interference Alignment)" proved in Section VI.D of [156]. □

Theorem 3. *If $\mathcal{C}$ supports the* RBT-SYS pattern*, then for each parity i, the corresponding generator-submatrix $G^{(i)}$ has* $\min(\alpha, k)$ *rows that have at most d non-zero entries. In particular, if $d \leq (2k-1)$, then all rows of G are d-sparse.*

Proof. In the *RBT-SYS pattern*, each parity node i helps the first α nodes via RBT, including $\min(\alpha, k)$ systematic nodes. In repair of a systematic node, the row of $G^{(i)}$ corresponding to the RBT symbol sent is d-sparse (by Theorem 2). This is true for each of the symbols sent during repair of each of the systematic nodes. These transferred symbols correspond to distinct symbols stored in node i, by Property 3, Section 6 of [156], which states that these symbols must be linearly independent. Therefore, $\min(\alpha, k)$ rows of $G^{(i)}$ are d-sparse.

In particular, for an MSR code, $d \leq (2k-1)$ implies $\alpha \leq k$, so all rows of G are d-sparse in this regime. □

6.7 Implementation and evaluation

Implementation and evaluation setting

We have implemented the PM-vanilla codes [127] and the PM-RBT codes (Section 6.4 and Section 6.5) in C/C++. In our implementation, we make use of the fact that the PM-vanilla and the PM-RBT codes are both linear. That is, we compute the matrices that represent the encoding and decoding operations under these codes and execute these operations with a single matrix-vector multiplication. We employ the Jerasure2 [121] and GF-Complete [122] libraries for finite-field arithmetic operations also for the RS encoding and decoding operations.

We performed all the evaluations, except the encoding and decoding time evaluations, on Amazon EC2 instances of type m1.medium (with 1 CPU, 3.75 GB memory, and attached to 410 GB of hard disk storage). We chose m1.medium type since these instances have hard-disk storage. We evaluated the encoding and decoding performance on instances of type m3.medium which run on an Intel Xeon E5-2670v2 processor with a 2.5GHz clock speed. All evaluations are single-threaded.

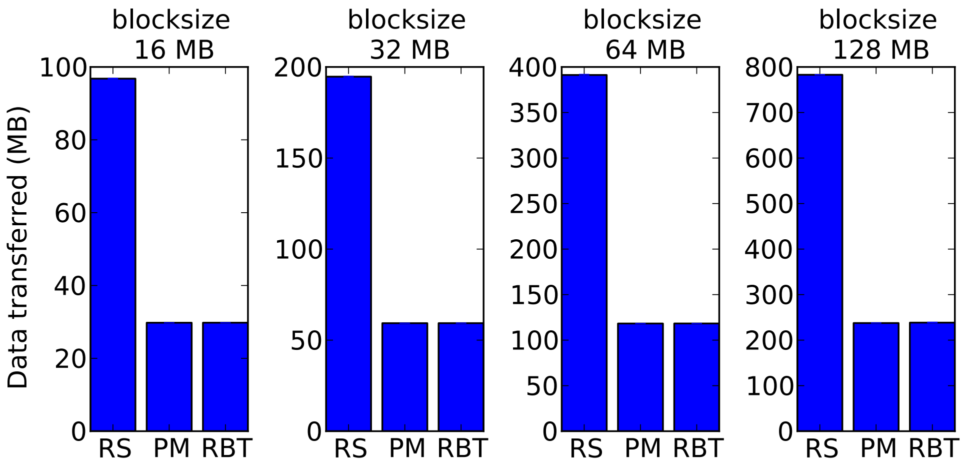


Figure 6.6: Total amount of data transferred across the network from the helpers during reconstruction. Y-axes scales vary across plots.

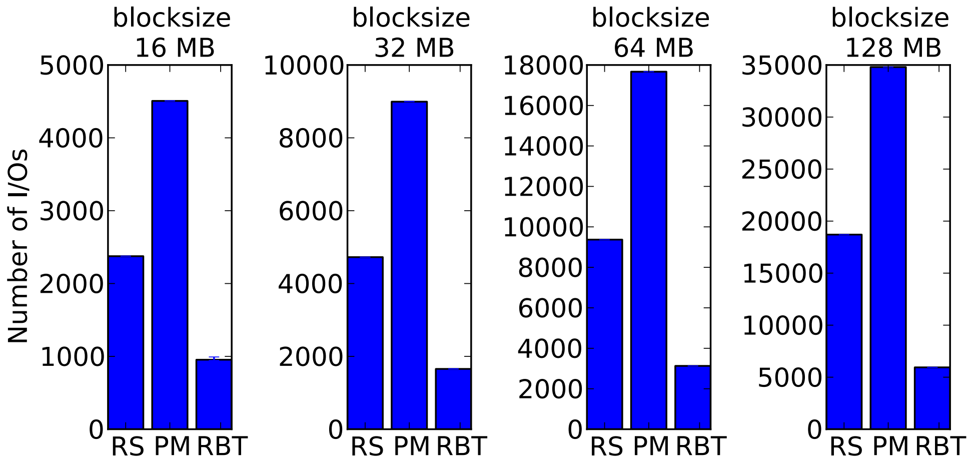
All the plots are from results aggregated over 20 independent runs showing the median values with 25th and 75th percentiles. In the plots, PM refers to PM-vanilla codes and RBT refers to PM-RBT codes. Unless otherwise mentioned, all evaluations on reconstruction are for ($n = 12,\ k = 6,\ d = 11$), considering reconstruction of block 1 (i.e., the first systematic block) with all $d = 11$ RBT-helpers. We note that all evaluations except the one on decoding performance (Section 6.7) are independent of the identity of the block being reconstructed.

Data transfers across the network

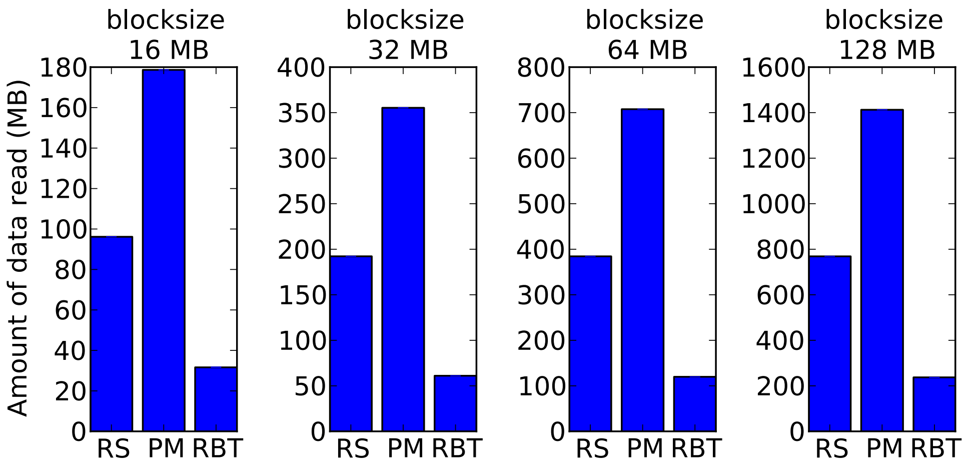
Figure 6.6 compares the total amount of data transferred from helper blocks to the decoding node during reconstruction of a block. We can see that, both PM-vanilla and PM-RBT codes have identical and significantly lower amount of data transferred across the network as compared to RS codes: the network transfers during the reconstruction for PM-vanilla and PM-RBT are about $4x$ lower than that under RS codes.

Data read and number of I/Os

A comparison of the total number of disk I/Os and the total amount of data read from disks at helpers during reconstruction are shown in Figure 6.7a and Figure 6.7b respectively. We observe that the amount of data read from the disks is as given by the theory across all block sizes that we experimented with. We can see that while PM-vanilla codes provide significant



(a) Total number of disk I/Os consumed



(b) Total amount of data read from disks

Figure 6.7: Total number of disk I/Os and total amount of data read from disks at the helpers during reconstruction. Y-axes scales vary across plots.

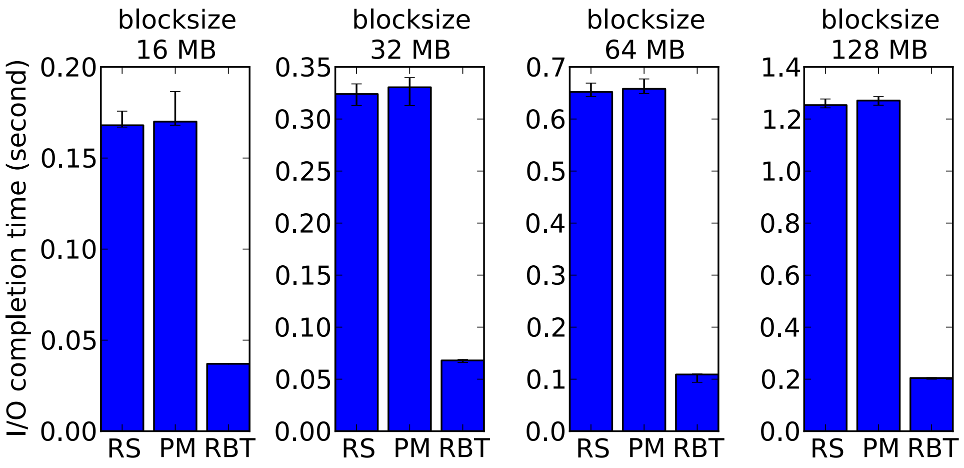


Figure 6.8: Maximum of the I/O completion times at helpers. Y-axes scales vary across plots.

savings in network transfers during reconstruction as compared to RS as seen in Figure 6.6, they result in an increased number of I/Os. Furthermore, the PM-RBT code leads to a significant reduction in the number of I/Os consumed and the amount of data read from disks during reconstruction. For all the block sizes considered, we observed approximately a $5x$ reduction in the number of I/Os consumed under the PM-RBT as compared to PM-vanilla (and approximately $3\times$ reduction as compared to RS).

I/O completion time

The I/O completion times during reconstruction are shown in Figure 6.8. During a reconstruction operation, the I/O requests are issued in parallel to the helpers. Hence we plot the the maximum of the I/O completion times from the $k = 6$ helpers for RS coded blocks and the maximum of the I/O completion times from $d = 11$ helpers for PM-vanilla and PM-RBT coded blocks. We can see that PM-RBT code results in approximately $5\times$ to $6\times$ reduction I/O completion time.

Decoding performance

We measure the decoding performance during reconstruction in terms of the amount of data of the failed/unavailable block that is decoded per unit time. We compare the decoding speed for various values of k, and fix $n = 2k$ and $d = 2k - 1$ for both PM-vanilla and PM-RBT. For reconstruction under RS codes, we observed that a higher number of systematic helpers

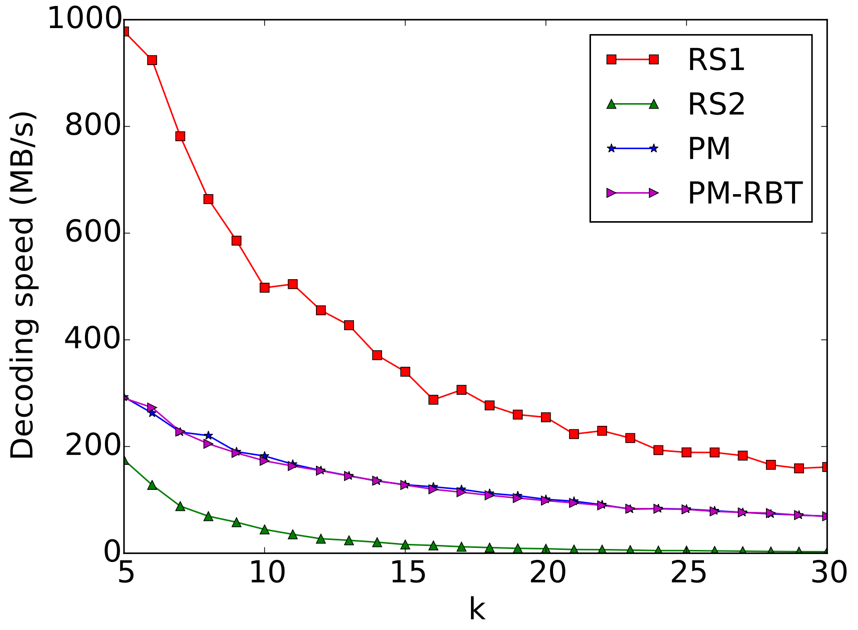


Figure 6.9: Comparison of decoding speed during reconstruction for various values of k with $n = 2k$, and $d = 2k - 1$ for PM-vanilla and PM-RBT.

results in a faster decoding process, as expected. In the plots discussed below, we will show two extremes of this spectrum: (RS1) the best case of helper blocks comprising all the existing $(k - 1) = 5$ systematic blocks and one parity block, and (RS2) the worst case of helper blocks comprising all the $r = 6$ parity blocks.

Figure 6.9 shows a comparison of the decoding speed during reconstruction of block 0. We see that the best case (RS1) for RS is the fastest since the operation involves only substitution and solving a small number of linear equations. On the other extreme, the worst case (RS2) for RS is much slower than PM-vanilla and PM-RBT. The actual decoding speed for RS would depend on the number of systematic helpers involved and the performance would lie between the RS1 and RS2 curves. We can also see that the transformations introduced here to optimize I/Os does not affect the decoding performance: PM-vanilla and PM-RBT have roughly the same decoding speed. In our experiments, we also observed that in both PM-vanilla and PM-RBT, the decoding speeds were identical for the n blocks.

Encoding performance

We measure the encoding performance in terms of the amount of data encoded per unit time. A comparison of the encoding speed for varying values of k is shown in Figure 6.10. Here we

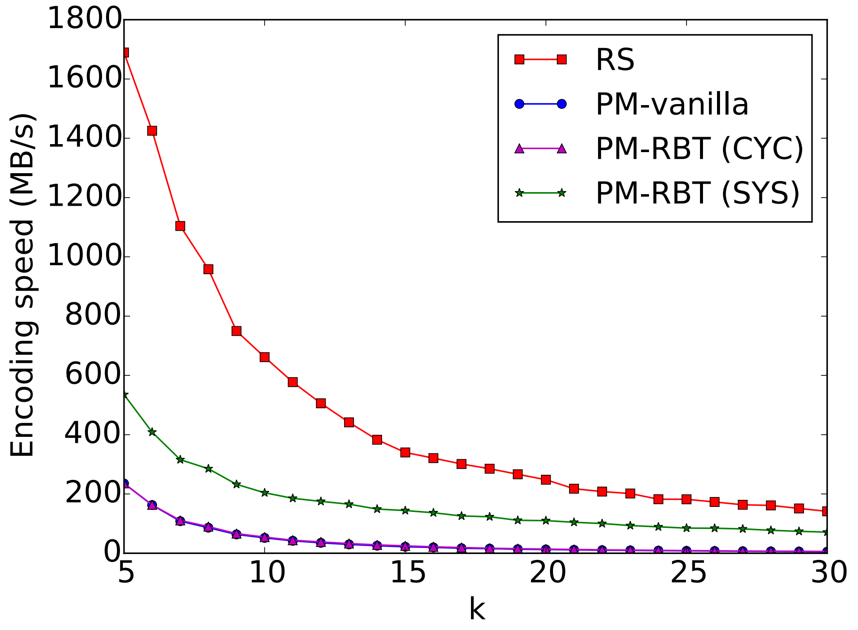


Figure 6.10: Encoding speed for various values of k with $n = 2k$, and $d = 2k - 1$ for PM-vanilla and PM-RBT.

fix $d = 2k-1$ and $n = 2k$ and vary the values of k. The lower encoding speeds for PM-vanilla and PM-RBT as compared to RS are expected since the encoding complexity of these codes is higher. In RS codes, computing each encoded symbol involves a linear combination of only k data symbols, which incurs a complexity of $O(k)$, whereas in PM-vanilla and PM-RBT with CYC RBT-helper pattern each encoded symbol is a linear combination of $k\alpha$ symbols which incurs a complexity of $\Theta(k^2)$.

We observe that encoding under PM-RBT with the SYS RBT-helper pattern (Section 6.5) is significantly faster than that under the PM-vanilla code. This is because the generator matrix of the code under the SYS RBT-helper pattern is sparse (i.e., has many zero-valued entries); this reduces the number of finite-field multiplication operations, that are otherwise computationally heavy. Thus, PM-RBT with SYS RBT-helper pattern results in faster encoding as compared to PM-vanilla codes, in addition to minimizing the disk I/O during reconstruction. From the encoding time data corresponding to Figure 6.10, we observe that the resulting encoding speed is approximately $k\times$, as expected. Such sparsity does not arise under the CYC RBT-helper pattern, and hence its encoding speed is almost identical to PM-vanilla.

Remark: The reader may observe that the speed (MB/s) of encoding in Figure 6.10 is faster than that of decoding during reconstruction in Figure 6.9. This is because encoding

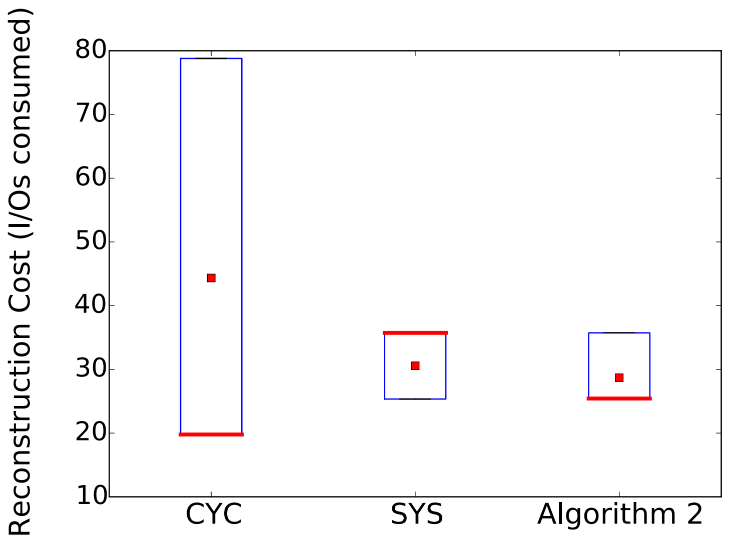


Figure 6.11: A box plot of the reconstruction cost for different RBT-helper assignments, for $\delta = 0.25$, $p = 0.03$, $n = 15$, $k = 6$, $d = 11$, and block size of 16 MB. In each box, the mean is shown by the small (red) square and the median is shown by the thick (red) line.

addresses k blocks at a time while the decoding operation addresses only a single block.

RBT-helper assignment algorithm

As discussed earlier in Section 6.5, we conducted experiments on EC2 performing reconstruction using different number of RBT-helpers (see Figure 6.4). We will now evaluate the performance of the helper assignment algorithm, Algorithm 2, via simulations employing the measurements obtained from these experiments. The plots of the simulation results presented here are aggregated from one million runs of the simulation. In each run, we failed one of the n blocks chosen uniformly at random. For its reconstruction operation, the remaining $(n-1)$ blocks were made unavailable (busy) with a probability p each, thereby also making some of the RBT-helpers assigned to this block unavailable. In the situation when only j RBT-helpers are available (for any j in $\{0,\ldots,d\}$), we obtained the cost of reconstruction (in terms of number of I/Os used) by sampling from the experimental values obtained from our EC2 experiments with j RBT-helpers and $(d-j)$ non-RBT-helpers (Figure 6.4a). The reconstruction cost for parity blocks is weighted by δ.

Figure 6.11 shows the performance of the RBT-helper assignment algorithm for the parameter values $\delta = 0.25$ and $p = 0.03$. The plot compares the performance of three possible choices of helper assignments: the assignment obtained by Algorithm 2 for the chosen parameters (shown in Figure 6.5), and the two extremities of Algorithm 2, namely SYS and

CYC. We make the following observations from the simulations. In the CYC case, the unweighted costs for reconstruction are homogeneous across systematic and parity blocks due to the homogenity of the CYC pattern, but upon reweighting by δ, the distribution of costs become (highly) bi-modal. In Figure 6.11, the performance of SYS and the solution obtained from Algorithm 2 are comparable, with the output of Algorithm 2 slightly outperforming SYS. This is as expected since for the given choice of parameter values $\delta = 0.25$ and $p = 0.03$, the output of Algorithm 2 (see Figure 6.5) is close to SYS pattern.

6.8 Summary

With rapid increases in the network-interconnect speeds and the advent of high-capacity disks, disk I/O is increasingly becoming the bottleneck in many large-scale distributed storage systems. A family of erasure codes called minimum-storage-regeneration (MSR) codes has recently been proposed as a superior alternative to the popular Reed-Solomon codes in terms of storage, fault-tolerance and network-bandwidth consumed. However, existing practical MSR codes do not address the critically growing problem of optimizing for I/Os. In this chapter, we considered the problem of optimizing MSR codes for the usage of storage device I/Os, while simultaneously retaining their optimality in terms of both storage, reliability and network-bandwidth.

The proposed solution is based on the identification of two key properties of existing MSR codes that can be exploited to make them I/O optimal. We presented an algorithm to transform MSR codes satisfying these properties into I/O optimal codes while retaining their storage and network optimality. Through an extensive set of experiments on Amazon EC2, using Product-Matrix-MSR codes, we have shown that transformed codes (PM-RBT) result in significant reduction in the I/O consumed (more than $5\times$ for typical parameters). Additionally, we also presented an optimization framework for helper assignment to attain a system-wide optimal solution.

Further, we established a general connection between optimizing I/O and reducing the encoding complexity in such MSR codes, by showing that the a specific transformation that we present for I/O optimization in MSR codes leads to sparsity in the generator matrix of the code. Through evaluations on EC2 using PM-MSR codes, we have shown that such a transformation leads to approximately $k\times$ speed up in the encoding computation time.

Chapter 7

EC-Cache: Load-balanced low-latency cluster caching using erasure coding

In the preceding four chapters, the focus has been on addressing the critical shortcomings of the traditional erasure codes for application in distributed storage systems. Specifically, we presented new code constructions and some techniques for existing code constructions for optimizing the usage of I/O, CPU and network resources while maintaining the storage efficiency offered by traditional codes. We also designed and built erasure-coded storage systems employing the proposed codes and techniques along with system-level optimizations to ensure that the promised theoretical gains translate to real-world system gains. In this chapter, we change gears to explore new opportunities for erasure coding in big-data systems, going beyond its traditional application in disk-based storage systems and going beyond its usage as a tool for achieving fault tolerance. Specifically, we show how erasure coding can be employed in in-memory (caching) systems for load balancing and improving read latency performance.

7.1 Introduction

In recent years, in-memory analytics [188, 97, 124, 14, 186] has gradually replaced disk-based solutions [12, 24, 39] as the primary toolchain for high-performance data analytics in big-data systems. The root cause behind the transition is simple: in-memory I/O is multiple orders of magnitude faster than that involving disks. In the architecture of big-data systems, the in-memory (caching) layer lies between the storage layer and the execution layer, providing fast access to the data for tasks/queries run by the execution layer. Such an in-memory system is a distributed cache making use of the memory space across all the servers in the cluster, and is also called a 'cluster cache'. For instance, Alluxio (formerly known as Tachyon [97]) is a popular cluster cache employed between execution frameworks such as Spark [188] and Hadoop Mapreduce [39] and storage systems such as the Hadoop Distributed File System (HDFS) [24] and Amazon S3 [6].

There are three primary challenges towards effectively using cluster caches for high-performance data-analytics:

(i) Judiciously determining which data items to cache and which ones to evict. Caching and eviction algorithms have been well studied in a number of past works, for example see [13, 19, 7, 125].

(ii) Increasing the effective memory capacity to be able to cache more data. Sampling [1, 86, 124] and compression [171, 17] are some of the popular approaches employed to increase the effective memory capacity.

(iii) Improving the I/O performance for data that is cached in memory.

It is the third challenge of improving the I/O performance for objects that are cached in memory that will be the focus of the current chapter. One of the key challenges towards this is the heavy skew in the popularity of these objects [9, 77]. A skew in the popularity of objects creates significant load imbalance across the servers in the cluster [9, 78]. The load imbalance results in over-provisioning in order to accommodate the peaks in the load distribution. The load imbalance also adversely affects the latency performance as heavily loaded servers will have poor response times. Consequently, load imbalance is one of the key challenges toward improving the performance of cluster caches.

A popular approach employed to address the challenge of popularity skew is *selective replication* [109, 9], which replicates objects based on their popularity; i.e., it creates more replicas for more popular objects as compared to the less popular ones. However, due to the limited amount of available memory, selective replication falls short in practice, for instance as shown for networked caches employed for web services [78]. While typical caches used in web services cache small-sized objects in the range of a few bytes to few kilobytes, data-intensive cluster caches used for data analytics [97, 7] store larger objects in the range of tens to hundreds of megabytes (Section 7.4). The larger sizes of the objects allow us to take a novel approach, using *erasure coding*, toward load balancing and improving I/O performance in cluster caches.

We present EC-Cache, an in-memory object cache that leverages *online* erasure coding – that is, the data is never stored in a decoded form – to provide better load balancing and latency performance. We show through extensive system evaluation that EC-Cache

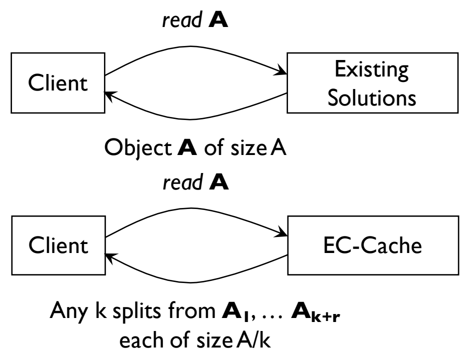


Figure 7.1: EC-Cache splits individual objects and encodes them to enable parallelism and late binding during individual reads.

can outperform the optimal selective replication mechanism while using the same amount of memory.

EC-Cache employs erasure coding and its properties toward load balancing and reducing I/O latency in the following manner:

(a) Self-Coding and load spreading: EC-Cache employs Maximum-Distance-Separable (MDS) codes. Recall from Chapter 2 that a $(k,\ r)$ MDS code encodes k data units and generates r parity units, and that any k of the $(k+r)$ total units are sufficient to decode the original k data units. Erasure coding is traditionally employed in disk-based systems to provide fault-tolerance in a storage efficient manner. In many such systems [73, 15, 130, 104], erasure coding is applied *across objects*: k objects are encoded to generate r additional objects. Read requests to an object are served from the original object unless it is missing. If the object is missing, it is reconstructed using the parities. In such a configuration, reconstruction using parities incurs a huge amount of bandwidth overhead [130, 15]; hence, coding across objects is not useful for load balancing or improving I/O performance. In contrast, EC-Cache divides *individual* objects into k *splits* and creates r additional parities. Read requests to an object are served by reading any k of the $(k+r)$ splits and decoding them to recover the desired object (Figure 7.1). This approach provides multiple benefits: First, spreading the load of read requests across both data and parity splits results in better load balancing under skewed popularity. Second, reading/writing in parallel from multiple splits provides better I/O performance due to parallelism during both reading and writing. Third, decoding the object using the parities does not incur any additional bandwidth overhead.

(b) Late Binding: Under self-coding, an object can be reconstructed from *any* k of its $(k+r)$ splits. This allows us to leverage the power of choices: instead of reading from exactly k splits, we read from $(k + \Delta)$ splits (where $\Delta \leq r$) and wait for the reading of *any* k splits to complete. Late binding makes EC-Cache resilient to background load imbalance and unforeseen stragglers, which are common in data intensive clusters [189, 8].

We have implemented EC-Cache over Alluxio (formerly called Tachyon) [97] using Intel's ISA-L library [79]. EC-Cache can be used directly as a caching layer in front of object stores such as Amazon S3 [6], Windows Azure Storage [28], OpenStack Swift [111]. It can also be used in front of cluster file systems such as HDFS [24], GFS [57], and Cosmos [29] by considering each block of a distributed file as an individual object.

We evaluated EC-Cache by deploying it on Amazon EC2 using synthetic workloads as well as a production workload from a 3000-machine cluster at Facebook. We observed that, for synthetic workloads, EC-Cache improves the median and the tail read latencies as compared to the optimal selective replication scheme by a factor greater than $2\times$, and improves the load distribution by more than $3\times$, while using the same amount of memory [1]. For production workloads, EC-Cache's reductions in latency range from $1.33\times$ for 1 MB objects to $5.5\times$ for 100 MB objects with an upward trend. We note that using the parameter values $k = 10$ and $\Delta = 1$ suffices to avail these benefits. In other words, a bandwidth overhead of at most 10% can lead to more than 50% reduction in the median and tail latencies. EC-Cache performs even better in the presence of an imbalance in the background network load, which is a common occurrence in data-intensive clusters [31]. Finally, EC-Cache performs well over a wide range of parameter settings.

Despite its effectiveness, our current implementation of EC-Cache offers advantages only for objects greater than 1 MB due to the overheads of creating $(k+\Delta)$ parallel TCP connections for each read. However, small objects form a negligible fraction of the cache footprint in data-intensive workloads (Section 7.4). Consequently, EC-Cache simply replicates objects smaller than this threshold to minimize the overhead. In other words, EC-Cache is designed to operate in the bandwidth-limited regime, where most data-intensive clusters operate; it is not ideal for key-value stores, where the number of requests handled per second is often the bottleneck.

7.2 Related literature

The focus on this chapter is to demonstrate and validate a new application for erasure coding, specifically in in-memory caching systems toward load balancing and reducing read latencies. The basic building blocks employed in EC-Cache are simple and have been studied

[1]This evaluation is in terms of the percent imbalance metric described in Section 7.7.

and employed in various other systems and settings. We borrow and build on the large body of existing work in this area.

Erasure coding in storage systems Since decades, disk arrays have employed erasure codes to achieve space-efficient fault-tolerance in the form of Redundant Array of Inexpensive Disks (RAID) systems [117]. The benefits of erasure coding over replication for providing fault tolerance in distributed storage systems has also been well studied [191, 180], and erasure codes have been employed in many settings such as network-attached-storage systems [2], peer-to-peer storage systems [93, 142], etc. Recently, erasure coding is being increasing deployed for storing relatively *cold* data in datacenter-scale distributed storage systems [104, 181, 73] to achieve fault tolerance while minimizing storage requirements. While some of the storage systems [171, 131, 104] encode across objects, some employ self-coding [181, 174]. However, the purpose of erasure coding in these systems is to achieve storage-efficient fault tolerance while the focus of EC-Cache is on load balancing and reducing median and tail read latencies.

Analyzing performance of erasure-coded storage systems Recently, there have been a number of theoretical works analyzing file-download times under certain queuing models for erasure-coded storage systems [76, 151, 82, 98, 184]. There are also a number of works studying scheduling and placement algorithms for erasure-coded storage systems [53, 165].

Late binding Many systems have employed the technique of sending additional/redundant requests or running redundant jobs to reign in tail latency in various settings [120, 10, 176, 38, 163, 54]. The effectiveness of late binding for load balancing and scheduling has been well known and well utilized in many systems [103, 112, 175]. Recently, there have been a number of theoretical works that analyze the performance of redundant requests [151, 98, 152, 82, 177, 56]. Specifically, [151, 82, 98] analyze the average content-download delay for erasure-coded storage systems using late binding under certain queuing models.

In-memory key-value stores A large body of work in recent years has focused on building high-performance in-memory key-value stores [101, 52, 99, 42, 109, 139]. EC-Cache focuses on a different workload where object sizes are much larger than sizes those allowed in these key-value stores. A recent work [190] employs erasure coding to provide fault tolerance in in-memory key-value stores. In EC-Cache, we use erasure codes toward providing load balancing and reducing I/O latency.

7.3 Background and motivation

This section provides a brief overview of object stores (e.g., Amazon S3 [6], Windows Azure Storage [28], OpenStack Swift [111], and Ceph [181]) and in-memory caching solutions (e.g., Tachyon/Alluxio [97] and PACMan [7]) used in modern data-intensive clusters which serve as an interface to these object stores. We discuss the tradeoffs and challenges faced in these systems followed by the opportunities for improvements over the state-of-the-art.

Cluster caching for object stores

Cloud object stores [6, 28, 111, 181] provide a simple PUT/GET interface to store and retrieve arbitrary objects at an attractive price point. In recent years, due to the rapid increase in the datacenter bandwidth [16, 162], cloud tenants are increasingly relying on these object stores as their primary storage solutions instead of cluster file systems such as HDFS [24]. For example, Netflix has been exclusively using Amazon S3 since 2013 [65].

Despite their prevalence, these object stores can rarely offer end-to-end non-blocking connectivity at cloud scale (unlike FDS [108]). Hence, for high-performance applications, in-memory storage systems [97] are becoming immensely popular as caching layers over these object stores. Objects stored in such systems can range from a few kilobytes to several gigabytes [104, 97, 7]. In contrast, key-value stores [94, 52, 42, 99]) deal with values ranging from a few bytes to few kilobytes.

Naturally, the highest-order metric for these caching systems is the I/O performance. Load balancing and fault-tolerance act as means toward lowering access latencies – the former increases I/O parallelism and mitigates hotspots, while the latter decreases disk hits in presence of failures.

Challenges in object caching

In-memory object caches face unique tradeoffs and challenges due to workload variations and dynamic infrastructure in large-scale deployments.

Popularity Skew Recent studies from production clusters show that the popularity of objects in cluster caches are heavily skewed [9, 77], which creates significant load imbalance across the storage workers in the cluster. This hurts I/O performance and also requires over-provisioning the cluster to accommodate the peaks in the load distribution. Unsurprisingly, load imbalance has been reported to be one of the key challenges toward improving the performance of cluster caches [78, 68].

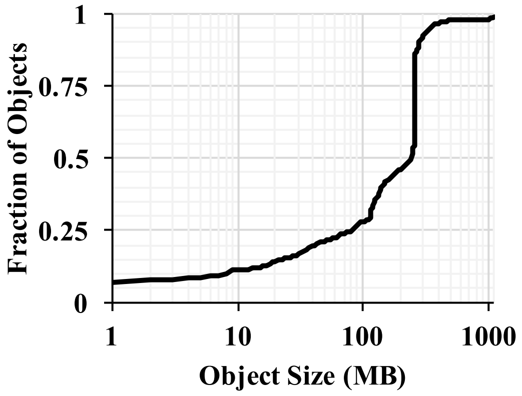
Background Load Imbalance Across the Infrastructure In addition to skews in object popularity, network interfaces – and I/O subsystems in general – throughout the cluster experience massive load fluctuations due to background activities [31]. Predicting and reacting to these variations in time is difficult. Even with selective replication, performance can deteriorate significantly if the source of an object suddenly becomes a hotspot.

Tradeoff Between Memory Efficiency, Fault Tolerance, and I/O Performance Given that memory is a constrained and expensive resource, existing solutions either sacrifice fault tolerance (i.e., no redundancy) to increase memory efficiency [97, 188], or incur high memory overheads (e.g., three-way replication) to provide fault tolerance [141, 187]. However, in caches, fault tolerance and I/O performance are inherently tied together since failures result in disk I/O activities; thereby, also increasing latency.

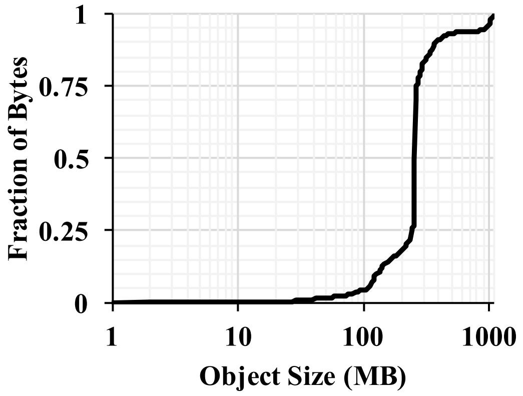
Potential for Benefits

Due to the challenges of popularity skew, background load imbalance, and failures, having a single copy of the objects is not sufficient to provide good performance. Replication schemes that treat all objects alike do not perform well under popularity skew as they result in wastage of memory on not-so-popular objects. Selective replication [109, 68, 9], where additional replicas of hot objects are cached, only provides coarse-grained support: each replica incurs an additional memory overhead of $1\times$ while only providing fractional benefits in terms of latency and load balancing. Selective replication has been shown to fall short in terms of both load balancing and read performance [78].

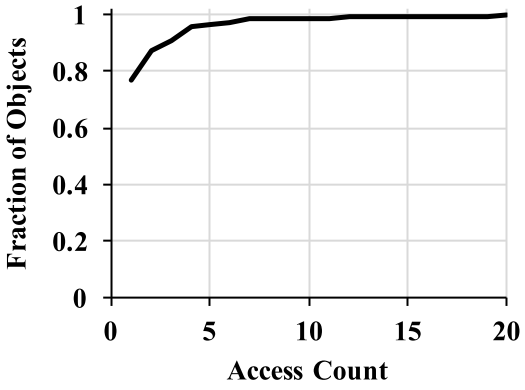
Selective replication along with object splitting (all splits of the same object have the same replication factor) does not solve the problem either. While such splitting provides better load balancing and opportunities for read parallelism, it cannot exploit late binding without incurring high memory and bandwidth overheads. As shown in Section 7.7, contacting multiple servers to read the splits severely affect the tail latencies, and late binding is necessary to reign in tail latencies. Hence, under splitting and selective replication, each object will need at least $2\times$ memory overhead. Furthermore, in order to make use of late binding, one must read multiple copies of each split resulting in at least $2\times$ bandwidth overhead.



(a) Object size (X-axis in log-scale)



(b) Object footprint (X-axis in log-scale)



(c) Object access characteristics [7, Figure 9]

Figure 7.2: Characteristics of object reads in the Facebook data analytics cluster. We observe that (a) objects vary widely in size; (b) small objects have even smaller footprint; and (c) a small fraction of the objects are accessed the most. Note that the X-axes are in log-scale in (a) and (b).

7.4 Analysis of production workload

Object stores are gaining popularity as the primary data storage solution for data analytics pipelines (e.g., at Netflix [50, 65]). As EC-Cache is designed to cater to these use cases, in order to obtain a better understanding of the requirements, we analyzed a trace consisting of millions of reads in a 3000-machine analytics cluster at Facebook. The trace was collected in October 2010, and it consisted of a mix of batch and interactive MapReduce analytics jobs generated from Hive queries. Each task accesses one or more blocks of a file and thus here object corresponds to a block. The default block size for the HDFS installation in this cluster was 256 MB, and the corresponding network had a 10 : 1 oversubscription ratio. Our goal behind analyzing these traces is to highlight characteristics – distributions of object sizes, their relative impact, access characteristics, and the nature of imbalance in I/O utilizations

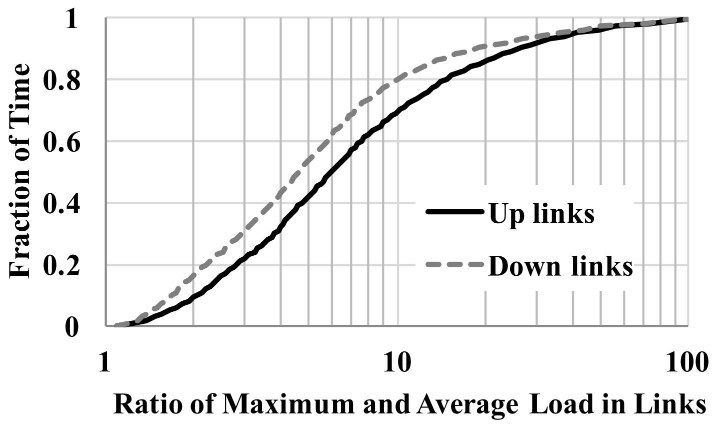


Figure 7.3: Imbalance in utilizations (averaged over 10-second intervals) of up and down oversubscribed links in Facebook clusters due to data analytics workloads. The X-axis is in log-scale.

– that enable us to make realistic assumptions in our analysis, design, and evaluation.

Large objects are prevalent

Data-intensive jobs in production clusters are known to follow skewed distributions in terms of their input size, output size, and number of tasks [7, 30, 32]. We observe a similar skewed pattern (Figure 7.2): only 7% (11%) of the objects read are smaller than 1 (10) MB, and their total size in terms of storage usage is miniscule. Furthermore, 28% of the objects are less than 100 MB in size with less than 5% storage footprint. Note that a large fraction of the blocks in the Facebook cluster are 256 MB in size, which corresponds to the vertical segment in Figure 7.2a.

Skew in popularity of objects

Next, we focus on object popularity/access patterns. As noted in prior works [30, 97, 7, 104, 9], the popularity of objects follow a Zipf-like skewed pattern, that is, a small fraction of the objects are highly popular. Figure 7.2c [7, Figure 9] plots the object access characteristics. (Note that this measurement does not include objects that were never accessed.) Here, the most popular 5% of the objects are seven times more popular than the bottom three-quarters [7].

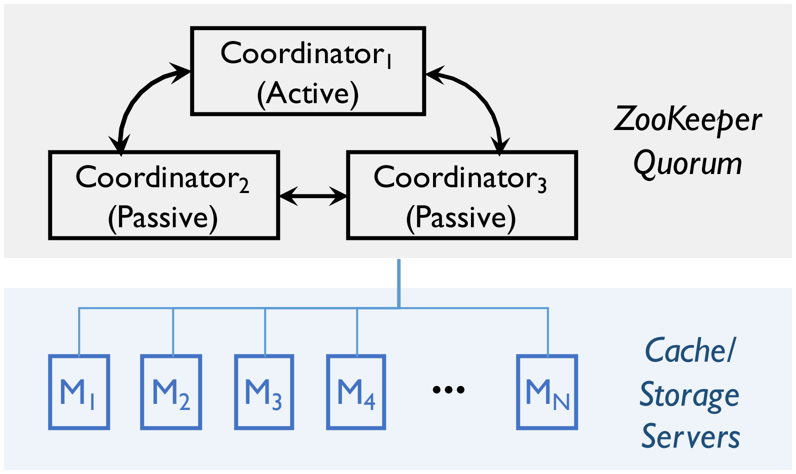


Figure 7.4: Alluxio architecture.

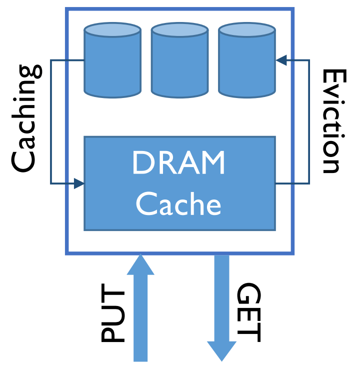
Network load imbalance is inherent

As also observed in prior studies [31, 87, 63, 23], we found that traffic from data analytics workloads is significantly imbalanced across the oversubscribed links in the network. Furthermore, the network imbalances are time varying. The root causes behind such imbalances include, among others, skew in application-level communication patterns [23, 87, 96], rolling upgrades and maintenance operations [23], and imperfect load balancing inside multipath datacenter networks [4]. We measured the network imbalance as the ratio of the maximum and the average utilizations across all oversubscribed links[2] in the Facebook cluster (Figure 7.3). This ratio was more than 4.5× more than 50% of the time for both up and downlinks, indicating a significant imbalance. More importantly, the maximum utilization was significantly high for a large fraction of the time, thereby increasing the possibility of congestion. For instance, the maximum uplink utilization was more than 50% of the capacity for more than 50% of the time. Since operations on object stores must go over the network, network hotspots can significantly impact their performance. This impact is more significant in-memory object caches where the network is the primary bottleneck.

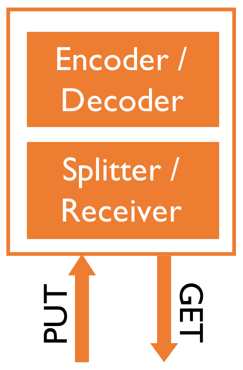
7.5 EC-Cache design overview

The primary objective of EC-Cache is to enable load-balancing and high-performance I/O in cluster caches. This section provides a high-level overview of EC-Cache's architecture.

[2]Links connecting top-of-the-rack (ToR) switches to the core.



(a) Backend server



(b) EC-Cache client

Figure 7.5: Roles in EC-Cache: (a) backend servers manage interactions between caches and persistent storage for client libraries; (b) EC-Cache clients perform encoding and decoding during writes and reads.

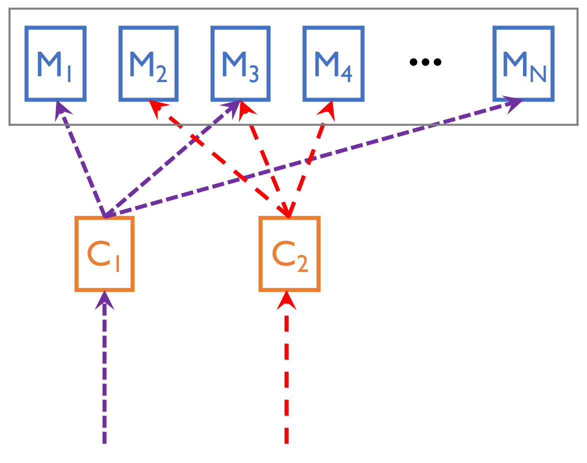
Overall architecture

We have implemented EC-Cache on top of the Alluxio (formerly known as Tachyon [97]) codebase — Alluxio is a popular caching solution for big data clusters. Consequently, EC-Cache shares some high-level similarities with Alluxio's architecture, such as a centralized architecture (Figure 7.4). Throughout the rest of the section, we highlight EC-Cache's key components that are different from Alluxio.

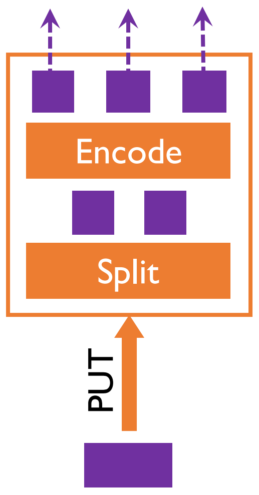
EC-Cache stores objects in a flat address space. An object is a byte sequence with a unique name (e.g., 128-bit GUID) as its key. Each object is divided into k splits and encoded using a Reed-Solomon code to add r parities. Each of these $(k+r)$ splits are cached on independent backend servers.

Backend Storage Servers Both in-memory and on-disk storage in each server is managed by an EC-Cache worker that responds to read and write requests for splits from clients (Figure 7.5a). Backend servers are unaware of object-level erasure coding introduced by EC-Cache. They also take care of caching and eviction of objects to and from memory using the least-recently-used (LRU) heuristic [97].

EC-Cache Client Library EC-Cache applications use a simple PUT-GET client API to store and retrieve objects (Figure 7.5b). As with backend servers, erasure coding remains transparent to the applications.



(a)



(b)

Figure 7.6: Writes to EC-Cache. (a) Two concurrent writes with $k = 2$ and $r = 1$. (b) Steps involved in an individual write in the client library with $k = 2$ and $r = 1$.

EC-Cache departs significantly from Alluxio in two major ways in its design of the user-facing client library. First, EC-Cache's client library exposes a significantly narrower interface for object-level operations as compared to Alluxio's file-level interface. Second, EC-Cache's client library takes care of splitting and encoding during writes and of reception and decoding during reads instead of writing and reading entire objects.

EC-Cache Coordinator EC-Cache also has a coordinator that has limited involvement during normal operations: it maintains an updated list of the active backend servers and manages object metadata. In addition, the EC-Cache coordinator performs several periodic background tasks such as garbage cleaning and split recovery after server failures.

Writes

EC-Cache stores each object by dividing it into k splits and encoding these splits using a Reed-Solomon code to add r parities. It then distributes these $(k + r)$ splits across unique backend servers. Figure 7.6 depicts an example of object writes with $k = 2$ and $r = 1$ for both objects C_1 and C_2. EC-Cache uses Intel's ISA-L library [79] for the encoding operations.

A key issue in any distributed storage solution is that of data placement and rendezvous, that is, where to write and where to read from. The fact that each object is further divided

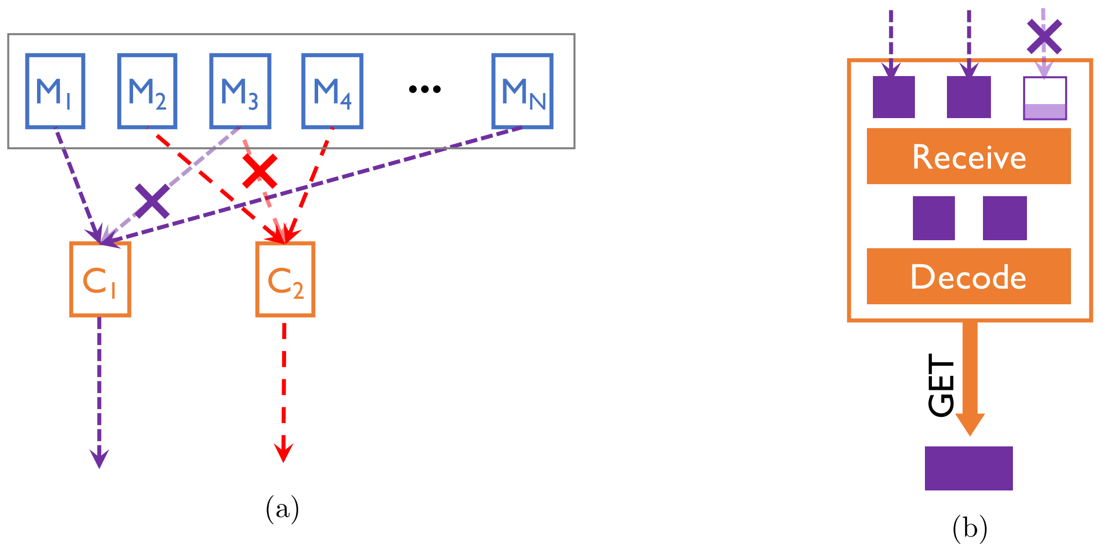


Figure 7.7: Reads from EC-Cache. (a) Two concurrent reads with $k = 2$ and $r = 1$. (b) Steps involved in an individual read in the client library, with $k = 2$ and $r = 1$. Because 2 out of 3 reads from the backend server is enough, the remaining ones are cancelled (crossed).

into $(k + r)$ splits in EC-Cache only magnifies its impact. For the same reason, metadata management is also an important issue in our design.

Similar to Alluxio and most other storage systems [57, 24, 97], the EC-Cache coordinator determines and manages the locations of the splits. Each write is preceded by an interaction with the coordinator server that determines where to write for all the $(k+r)$ splits. Similarly, each reader receives the locations of the splits in a single interaction with the coordinator. This approach provides the maximum flexibility in load-aware data placement [31], and it takes only a small fraction of time in comparison to actual object writes and reads. We plan to study fully decentralized placement schemes [108] in the future.

The total amount of additional metadata due to splitting of objects is small. For each object, EC-Cache stores its associated k and r values and the associated $(k + r)$ server locations (32-bit unsigned integers). This forms only a small fraction of the total metadata size of an object. Since EC-Cache does not track lineage information, the amount of metadata for each object is often smaller than that in Alluxio.

Reads

The key advantage of EC-Cache comes into picture during the read operations. Instead of reading from a single replica, the EC-Cache client library reads from some $(k + \Delta)$ splits (out of the $(k+r)$ total splits of the object) in parallel. This provides two benefits. First, it exploits I/O parallelism and distributes the load across many backend servers. Second, it can complete reading as soon as any k out of $(k+\Delta)$ splits arrive, thereby avoiding stragglers. Once some k splits of the objects have been received, the decoding operation is performed to obtain the object. EC-Cache uses Intel's ISA-L library [79] for the decoding operations. Each additional read introduces an additional bandwidth usage of at most B/k for each object.

Figure 7.7a provides an example of a reading operation over the objects stored in the example presented in Figure 7.6, where both the objects have $k = 2$ and $r = 1$. Although 2 splits are enough to complete each read, in this example, EC-Cache is performing an additional reads (that is, $\Delta = 1$). Since both objects had one split in server M_3, reading from that server may be slow. However, instead of waiting for that read, EC-Cache will proceed as soon as it receives the other 2 splits (Figure 7.7b) and decode them to complete the object reads.

7.6 Analysis

In this section, we provide an analytical explanation for the benefits offered by EC-Cache.

Impact on Load Balancing

Consider a cluster with S servers and F objects. For simplicity, let us first assume that all objects are equally popular. Under selective replication, each object is placed on a server chosen uniformly at random out of the S servers. For simplicity, first consider EC-Cache where each split of a object is placed on a server chosen uniformly at random (neglecting the fact that each split is placed on a unique server). The total load on a server equals the sum of the loads on each of the splits stored on that server. Thus the load on each server is a random variable. Without loss of generality, let us consider the load on any particular server and denote the corresponding random variable by L.

The variance of L has a direct impact on the load imbalance in the cluster – intuitively, a higher variance of L, implies a load on the maximally loaded server in comparison to the average load, and consequently, a higher load imbalance.

Under this simplified setting, the following result holds.

Theorem 1 For the setting described above:

$$\frac{\text{Var}(L_{\text{EC-Cache}})}{\text{Var}(L_{\text{Selective Replication}})} = \frac{1}{k}.$$

Proof: Let $w > 0$ denote the popularity of each of the files. The random variable $L_{\text{Selective Replication}}$ is distributed as a Binomial random variable with F trials and success probability $\frac{1}{S}$, scaled by w. On the other hand, $L_{\text{EC-Cache}}$ is distributed as a Binomial random variable with kF trials and success probability $\frac{1}{S}$, scaled by $\frac{w}{k}$. Thus we have

$$\frac{\text{Var}(L_{\text{EC-Cache}})}{\text{Var}(L_{\text{Selective Replication}})} = \frac{\left(\frac{w}{k}\right)^2 (kF)\frac{1}{S}\left(1-\frac{1}{S}\right)}{w^2 F\frac{1}{S}\left(1-\frac{1}{S}\right)} = \frac{1}{k},$$

thereby proving our claim. □

Intuitively, the splitting action of EC-Cache leads to a smoother spreading of the load in comparison to selective replication. One can further extend Theorem 1 to accommodate a skew in the popularity of the objects. Such an extension leads to an identical result on the ratio of the variances. Additionally, the fact that each split of a object in EC-Cache is placed on a unique server only helps in spreading the load better, leading to even better load balancing.

Impact on Latency

Next, we focus on how object splitting impacts read latencies. Under selective replication, a read request for an object is served by reading the object from a server. We first consider naive EC-Cache without any additional reads. Under naive EC-Cache, a read request for an object is served by reading k of its splits in parallel from k servers and performing a decoding operation. Let us also assume that the time taken for decoding is negligible compared to the time taken to read the splits.

Intuitively, one may expect that reading splits in parallel from different servers will reduce read latencies due to the parallelism. While this reduction indeed occurs for the average/median latencies, the tail latencies behave in an opposite manner due to the presence of stragglers.

In order to obtain a better understanding of the aforementioned phenomenon, let us consider the following simplified model. Consider a parameter $p \in [0, 1]$ and assume that for any request, a server becomes a straggler with probability p, independent of all else. There are two primarily contributing factors to the distributions of the latencies under selective replication and EC-Cache:

(a) Proportion of stragglers: Under selective replication, the fraction of requests that hit stragglers is p. On the other hand, under EC-Cache, a read request for an object will face a straggler if any of the k servers from where splits are being read becomes a straggler. Hence, a higher fraction $\left(1-(1-p)^k\right)$ of read requests can hit stragglers under naive EC-Cache.

(b) Latency conditioned on absence/presence of stragglers: If a read request does not face stragglers, the time taken for serving a read request is significantly smaller under EC-Cache as compared to selective replication, because of the parallel reading of the smaller size splits. On the other hand, in the presence of a straggler in the two scenarios, the time taken for reading under EC-Cache is about as large as that under selective replication.

Putting the aforementioned two factors together we get that the relatively higher likelihood of a straggler under EC-Cache increases the number of read requests incurring a higher latency. The read requests that do not encounter any straggler incur a lower latency as compared to selective replication. These two factors explain the decrease in the median and mean latencies, and the increase in the tail latencies.

In order to alleviate the impact on tail latencies, we use additional reads and late binding in EC-Cache. Reed-Solomon codes, by virtue of being MDS, have the property that any k of the collection of all splits of an object suffice to decode the object. We exploit this property by reading more than k splits in parallel, and using the k splits that are read first. It is well-known that such additional reads help in mitigating the straggler problem and alleviate the affect on tail latencies [112, 175, 38, 151, 82, 98].

7.7 Evaluation

We evaluated EC-Cache through a series of experiments on Amazon EC2 [5] clusters using synthetic workloads and traces from Facebook production clusters. The highlights of the evaluation results are:

- For skewed popularity distributions, EC-Cache improves load balancing over selective replication by 3.3× while using the same amount of additional extra memory. EC-Cache also decreases the median latency by 2.64× and the 99.9th percentile latency by 1.79×.
- For skewed popularity distributions *and* in the presence of background load imbalance, EC-Cache decreases the 99.9th percentile latency over selective replication by 2.56× while maintaining the same benefits in median latency and load balancing as in the case without background load imbalance.
- EC-Cache's improvements over selective replication keeps increasing as object sizes keep increasing in production traces; e.g., 5.5× at median for 100 MB objects with an upward

trend.

- EC-Cache outperforms selective replication across a wide range of values of k, r, and Δ.

Methodology

Cluster Unless otherwise specified, our experiments use 55 `c4.8xlarge` EC2 instances. 25 of these machines act as backend servers for EC-Cache, each with 8 GB cache space, and 30 machines generate thousands of read requests to EC-Cache. All these machines were instantiated in the same Amazon Virtual Private Cloud (VPC) with 10 Gbps enhanced networking enabled; we observed around 5 Gbps bandwidth between machines in the VPC using `iperf`. As mentioned earlier, we implemented EC-Cache on Alluxio [97], which, in turn, uses HDFS [24] as its persistence layer and runs on the 25 backend servers. We used DFS-Perf [40] to generate the workload on the 30 client machines.

Metrics Our primary metrics for comparison are *latency* in reading objects and *load imbalance* across the backend servers.

Given a workload, we consider mean, median, and high-percentile latencies. We measure improvements in latency as:

$$\text{Latency Improvement} = \frac{\text{Latency w/ Compared Scheme}}{\text{Latency w/ EC-Cache}}$$

If the value of this "latency improvement" is greater (or smaller) than one, EC-Cache is better (or worse).

We measure load imbalance using the percent imbalance metric λ defined as follows:

$$\lambda = \left(\frac{L_{\text{max}} - L_{\text{avg}\star}}{L_{\text{avg}\star}}\right) * 100, \tag{7.1}$$

where L_{max} is the load on the server which is maximally loaded and $L_{\text{avg}\star}$ is the load on any server under an *oracle* scheme where the total load is equally distributed among all the servers without any overhead. λ measures the percentage of additional load on the maximally loaded server as compared to the ideal average load. For the percent imbalance metric λ, a lower value is better. As EC-Cache operates in the bandwidth-limited regime, the load on a server is the total amount of data read from that server.

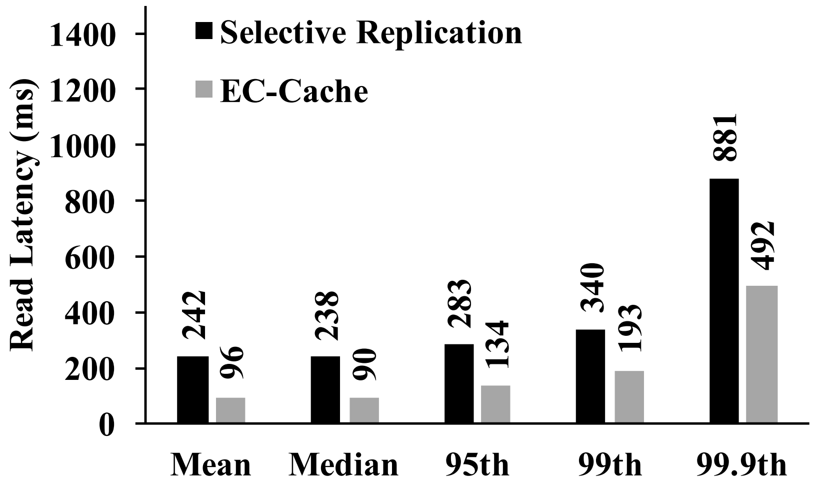


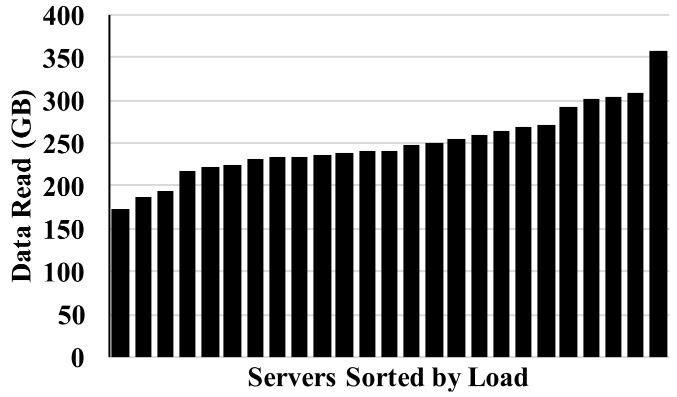
Figure 7.8: Read latencies under skewed popularity of objects.

Skew Resilience

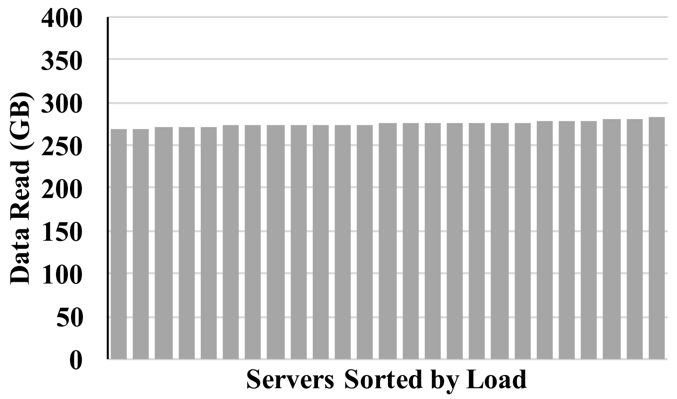
We begin by evaluating the performance of EC-Cache in the presence of skews in object popularity.

We consider a Zipf distribution, which is common in many real-world object popularity distributions [7, 97, 9]. In particular, we consider a skew parameter of 0.9 (i.e., high skew). EC-Cache uses the parameter values $k = 10$, and $\Delta = 1$. We allow both selective replication and EC-Cache to use a 15% memory overhead to handle the skew. Both schemes make use of the skew information to decide how to allocate the allowed memory overhead among different objects in an identical manner: flattening out the skew starting from the most popular objects until the allowed memory overhead is consumed. Moreover, both use uniform random placement policy to evenly distribute the objects across memory servers. The size of each object considered in this experiment is 40 MB. We present results for varying object sizes observed in the Facebook trace, and also perform a sensitivity analysis with respect to all the above parameters.

Latency Characteristics Figure 7.8 compares the mean, median, and tail latencies of EC-Cache and selective replication. We observe that EC-Cache improves median and mean latencies by factors of 2.64× and 2.52×, respectively. EC-Cache outperforms selective replication at high percentiles as well, improving the latency by a factor of 1.76× at the 99th percentile and by a factor of 1.79× at the 99.9th percentile.



(a) Selective replication



(b) EC-Cache with $k = 10$

Figure 7.9: A comparison of the load distribution across servers in terms of amount of data read from each server.

Load Balancing Characteristics Figure 7.9 presents the distribution of loads across servers. (In the figure, the servers are sorted based on their load). The percent imbalance metric λ observed for selective replication and EC-Cache in this experiment are 43.45% and 13.14% respectively.

Decoding Overhead During Reads We observe that the time taken to decode during the reads is approximately 30% of the total time taken to complete a read request. Despite this overhead, we see (Figure 7.8) that EC-Cache provides a significant reduction in both median and tail latencies. Our current implementation uses only a single thread for performing the decoding operation. Importantly, the underlying erasure codes permit the decoding operation to be made embarrassingly parallel, potentially allowing for a linear speed up and consequently further improving the latency performance of EC-Cache.

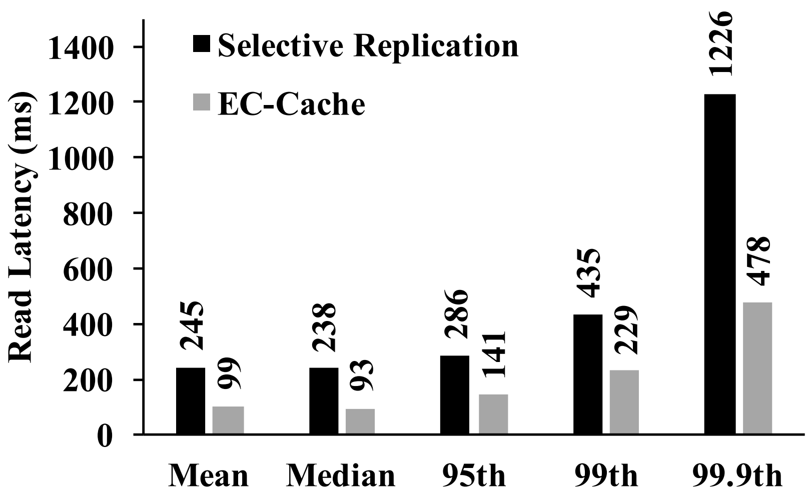


Figure 7.10: Read latencies under skewed popularity of objects in the presence of background traffic from big data workload.

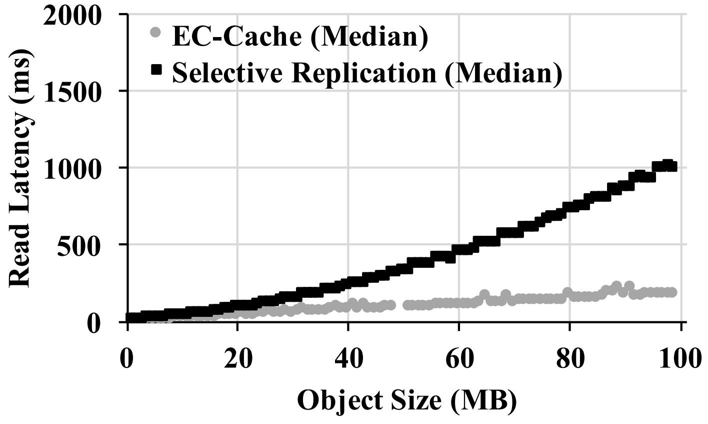
Impact of Background Load Imbalance

We now investigate the performance of EC-Cache in the presence of a background network load, specifically in the presence of an imbalance in background traffic. For this experiment, we used the same skew and parameter settings as in the previous experiments. We generated background traffic using a big data workload with network traffic characteristics as described in Section 7.4, that is, the network interfaces of a handful of the machines were overloaded.

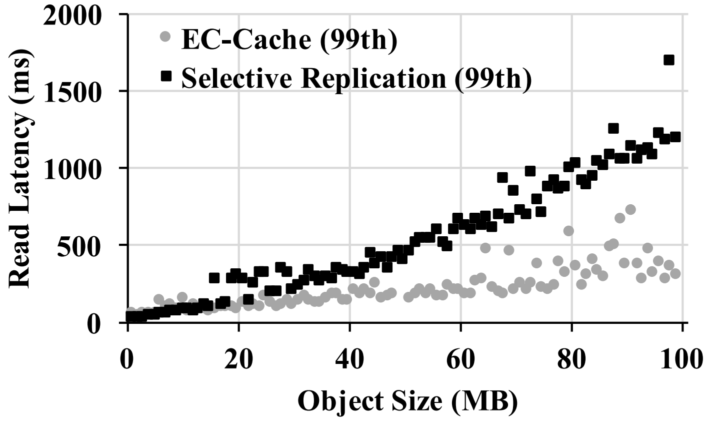
Latency Characteristics Figure 7.10 compares the mean, median, and tail latencies using both EC-Cache and selective replication. We observe that, EC-Cache improves the median and median latencies by factos of 2.56× and 2.47× respectively.

Importantly, at higher percentiles, the benefits offered by EC-Cache over selective replication are even more than that observed above in the absence of background load imbalance. In particular, EC-Cache outperforms selective replication by a factor of 1.9× at the 99th percentile and by a factor of 2.56× at the 99.9th percentile. The reason for such an improved performance of EC-Cache is that while selective replication gets stuck in few of the overloaded backend servers, EC-Cache remains almost impervious to such imbalance due to late binding.

Load Balancing Characteristics The percent imbalance metric λ for selective replication and EC-Cache are similar to that reported above in the absence of background load



(a) Median latency



(b) 99th percentile latency

Figure 7.11: Comparison of EC-Cache and selective replication read latencies over varying object sizes in the Facebook production trace. EC-Cache advantages keep improving for larger object sizes.

imbalance. This is because with the same setting as earlier, the background load imbalance does not affect the load distribution arising from read requests.

Performance on Production Workload

So far we have focused on EC-Cache's performance on skewed popularity distributions and background load imbalance for a fixed object size. Here, we compare EC-Cache against selective replication on the workload collected from Facebook (details in Section 7.4), where objects sizes vary widely.

Figure 7.11 presents the median and the 99th percentile read latencies for objects of

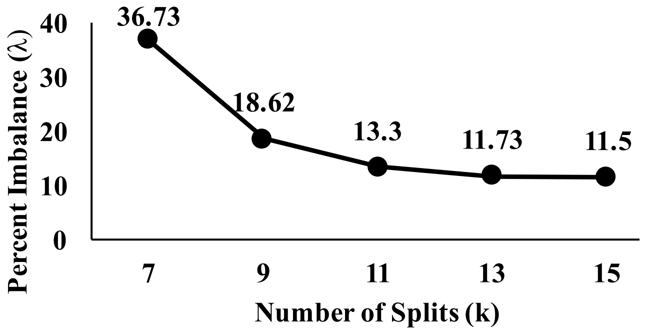


Figure 7.12: Percent imbalance metric (λ) for various values of the number of splits (k). The imbalance decreases as objects are divided into more splits (up to a certain threshold).

different sizes (starting from 1 MB) in the Facebook workload. Note that EC-Cache resorts to replication for objects smaller than 1 MB to avoid communication overheads.

We make two primary observations. First, EC-Cache's median improvements over selective replication steadily increases with the object size; e.g., EC-Cache is 1.33$\times$ faster for 1 MB-sized objects, which improves to 5.5$\times$ for 100 MB-sized objects and beyond. Second, EC-Cache's 99th percentile improvements over selective replication kick off when object sizes grow beyond 10 MB. This is because the overheads of creating $(k + \Delta)$ connections affects few of the reads in EC-Cache. Beyond 10 MB, the connection overhead gets amortized, and EC-Cache's improvements over selective replication even in tail percentile latencies steadily increases from 1.25$\times$ to 3.85$\times$ for 100 MB objects.

Sensitivity Analysis

Here, we analyze the effects of the choice of EC-Cache's parameters on its performance. We present the results for 10 MB objects (instead of 40 MB as above) in order to highlight the effects of all the parameters more clearly and to be able to sweep for a wide range of parameters.

Number of splits k

We begin with the number of splits k.

Load Balancing Characteristics The percent imbalance metric for varying values of k with $\Delta = 1$ are shown in Figure 7.12. We can see that the load balancing effect improves as the value of k is increased. There are two reasons for this phenomenon: (i) A higher the

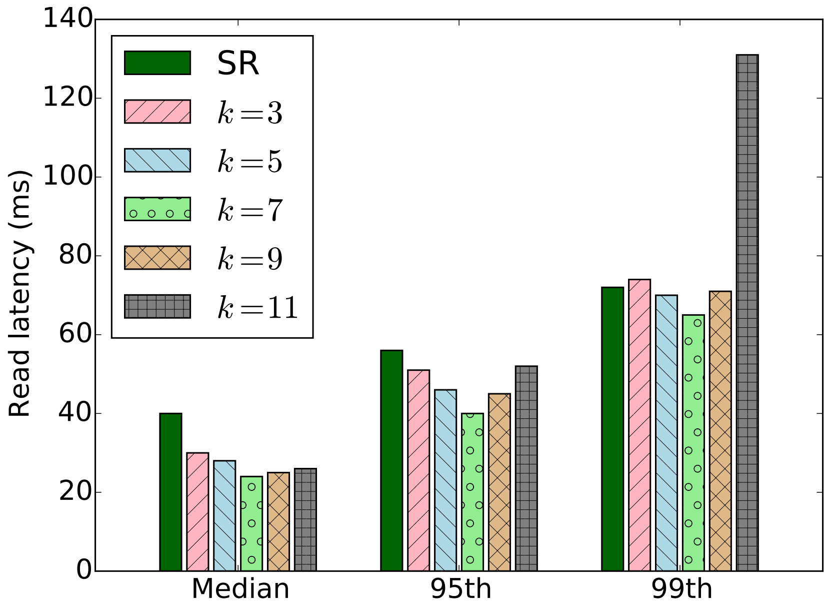


Figure 7.13: Impact of number of splits k on the read latency. SR refers to selective replication.

value of k leads to a smaller granularity of individual splits, thereby resulting in a greater smoothing on the load under skewed popularity; and (ii) the load overhead due to additional reads varies inversely with the value of k. This trend is as expected by our theoretical analysis (Section 7.6).

Latency Characteristics Figure 7.13 shows a comparison of median and tail percentile of the read latencies for different values of k. The corresponding values for selective replication is also provided for comparison. We see that parallelism helps to improve the median latencies, but with diminishing returns. However, higher values of k lead to worse tail read latencies as a result of the straggler effect discussed earlier in Section 7.6. Hence, for values of k higher than 10, we need more than one additional reads to reign in the tail percentile latencies. This effect is studied in the subsequent evaluation.

Additional Reads Δ

Figure 7.14 shows the CDF of the read latencies from about $160,000$ reads for selective replication and EC-Cache with $k = 10$ with and without additional reads. We can see that EC-Cache without any additional reads performs quite well in terms of the median latency,

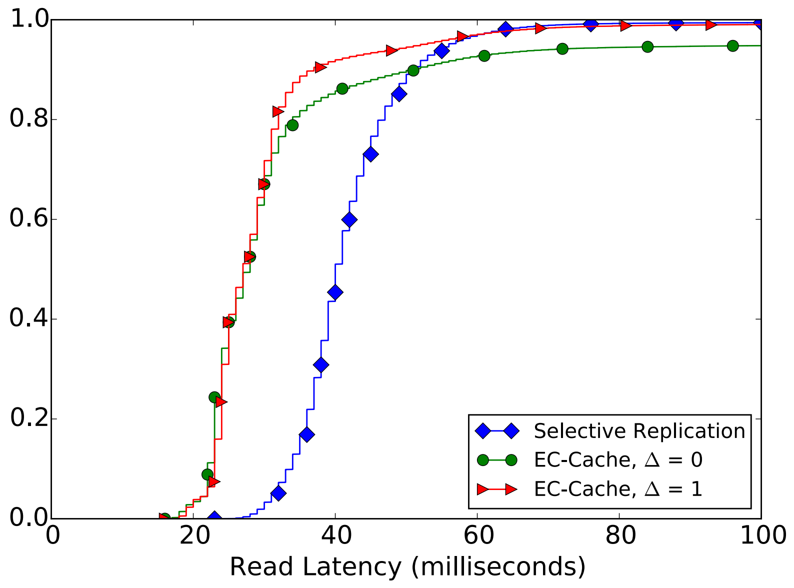


Figure 7.14: CDF of read latencies showing the need for additional reads in reining in tail latencies in EC-Cache.

but becomes severely inferior at latencies above the 80th percentile. Here, adding just one additional read helps EC-Cache tame the negative effect of the fan-out on the tail latencies.

The impact of higher number of additional reads for $k = 12$, and an object size of $20MB$ is shown in Figure 7.15. (We choose $k = 12$ instead of 10 since the effect of more additional reads is more prominent for higher values of k.) The results plotted in Figure 7.15 show that the first one or two additional read provide a significant reduction in the tail latencies while subsequent additional reads provide little additional benefits. In general, having too many additional reads would start hurting the performance since they would necessitate a proportional increase in the communication and bandwidth overheads.

Memory Overhead

In the previous evaluations, we compared EC-Cache and selective replication with a fixed memory overhead of 15%. Given a fixed amount of total memory, increasing memory overhead allows a scheme to store more redundant objects but fewer unique objects. Here, we varied memory overhead and evaluated the latency and load balancing characterisitics of selective replication and EC-Cache.

We observed that the relative difference in terms of latency between EC-Cache and selective replication remained similar to that shown in Figure 7.8 – EC-Cache provided a

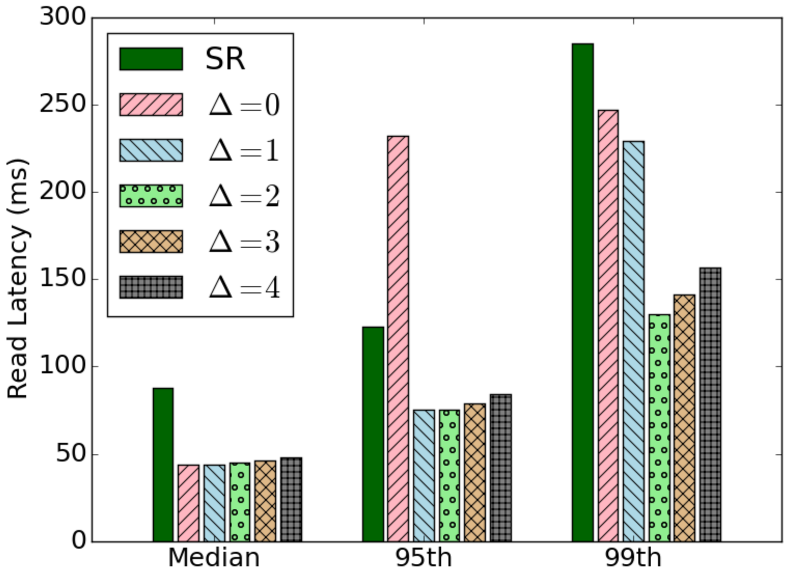


Figure 7.15: Impact of number of additional reads on read latency. SR refers to selective replication.

significant reduction in the median and tail latencies as compared to selective replication even for higher memory overhead.

However, in terms of load balancing, the gap between EC-Cache and selective replication decreased with increasing memory overhead. This is because EC-Cache was almost balanced even with just 15% memory overhead (Figure 7.9) with little room for further improvement. In contrast, selective replication became more balanced, reducing the relative gap from EC-Cache

Write Performance

Figure 7.16 shows a comparison of the average write times. The time taken to write an object in EC-Cache involves the time to encode and the time to write out the splits to different workers, and the figure depicts the breakdown of the write time in terms of these two components. We observe that EC-Cache is faster than selective replication for objects larger than 40 MB, supplementing its faster performance in terms of the read times observed earlier. EC-Cache performs worse for smaller objects due to the overheads of connecting to several machines in parallel. Finally, we observe that the time taken for encoding is less than 10% of the total write time, regardless of the object size.

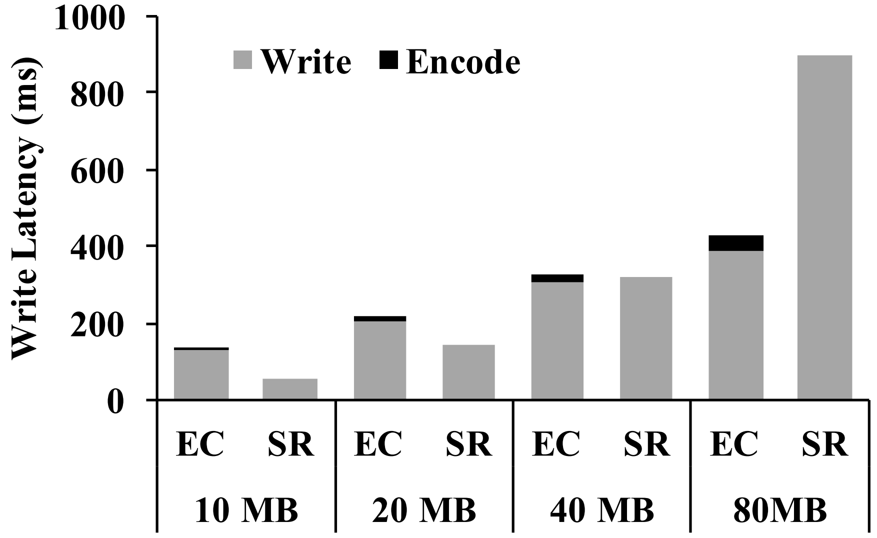


Figure 7.16: Comparison of writing times (and encoding time for EC-Cache) for different object sizes.

7.8 Discussion

While EC-Cache outperforms existing solutions both in terms of latency and load balancing, our current implementation has several known limitations. We envisage that addressing these limitations will further improve the performance of EC-Cache.

Networking Overheads. A key reason behind EC-Cache not being as effective for smaller objects is its networking overheads. More specifically, creating many TCP connections takes a non-negligible fraction (few milliseconds) of a read's duration. Instead, using long-running, reusable connections may allow us to support even smaller objects. Furthermore, multiplexing will also help in decreasing the total number of TCP connections in the cluster.

Reducing Bandwidth Overhead. EC-Cache has at most 10% bandwidth overheads in our present setup. While this overhead does not significantly impact performance during off-peak hours, it can have a non-negligible impact during the peak. In order to address this issue, one may additionally employ proactive cancellation [38, 112] that can help reduce the bandwidth overheads due to additional reads.

Time Varying Skew. EC-Cache can handle time-varying popularity skew and load imbalance by changing the number of parities. However, we have not yet implemented this feature due to a limitation posed by Alluxio. In our current implementation, we store individual splits of an object as part of the file abstraction in Alluxio to reduce metadata overheads (Section 7.5). Since Alluxio does not currently offer support for appending to a file once the file is closed (`ALLUXIO-25` [166]), at present we are unable to dynamically change the number of parities, and hence to adapt to time-varying skew. Assuming that the underlying support for appending is obtained, EC-Cache will respond to time-varying skew better as compared to selective replication since the overhead of any object can be changed in fractional increments in EC-Cache as opposed to the limitation of having only integral increments in selective replication.

7.9 Summary

In this chapter, we presented EC-Cache, a cluster cache that employs erasure coding for load balancing and for reducing the I/O latencies under skewed object popularity. Through extensive evaluations, we showed that, as compared to the state-of-the art, EC-Cache improves load balancing by a factor of 3.3× and reduces the median and tail read latencies by more than 2×, while using the same amount of memory. EC-Cache does so using 10% additional bandwidth and a small increase in the amount of stored metadata. The benefits offered by EC-Cache are further amplified in the presence of background network load imbalance.

Chapter 8

Conclusions and Future Directions

As we look to scale next-generation big-data systems, a primary challenge is that of sustaining the massive growth in the volume of data needing to be stored and retrieved reliably and efficiently. Replication of data, while ideal from the viewpoint of its simplicity and flexible access, is clearly not a sustainable option for all but a small fraction of the data. Not surprisingly, therefore, erasure-coded storage systems have recently received more traction, despite their shortcomings. This has opened up many exciting challenges and opportunities in the theoretical as well as systems fronts. In this thesis, we addressed some of these challenges and explored some new opportunities. However, a lot still remains to be done, and we discuss potential future directions below.

Advancing theory and practice in tandem for erasure-coded storage systems

As discussed in the thesis, there has been a significant amount of recent work in the coding theory community on constructing reconstruction-optimal MDS codes. However, most of the constructions are theoretical, falling short in one or the other way in being readily applicable to real-world systems. For instance, many constructions either need very high field size, or require very large number of substripes, or support only a limited set of parameters that do not cover the parameters of interest in many real-world systems. In this thesis, we proposed the Piggybacking framework that enabled construction of practical, high-rate, reconstruction-efficient MDS codes with a small field size, a small number of substripes, and without any restrictions on the code parameters. However, the amount of network bandwidth and I/O consumed by the proposed constructions are not optimal, that is, they do not meet the theoretical minimum established by the cutset lower bound [41]. It is of both theoretical and practical interest to construct high-rate, MDS codes with a small field size and a small number of substripes that can achieve better reconstruction efficiency reducing the gap between Piggyback code designs presented in this thesis and the cutset lower bound.

While the above discussion points to approaching the gap from the theoretical perspective, similar efforts are needed from the systems side as well. The current generation of distributed storage systems have been designed for replication. Although traditional Reed-Solomon codes are being deployed recently, there is no general framework for more advanced erasure codes. In this thesis, we presented Hitchhiker built over HDFS, which employed Piggybacked-RS codes with just 2 substripes and same field size as RS codes. Despite the relative simplicity of Hitchhiker's erasure code and its similarity to RS codes, a slew of novel system level optimizations, such as the hop-and-couple technique, were necessary to translate the gains promised in theory to real system gains. In order to reap the full benefits of the recent and ongoing theoretical advances, the next-generation distributed storage systems need to be designed to natively and holistically support superior, sophisticated code constructions.

Non-traditional applications of erasure codes

In this thesis, we explored new avenues of applicability of erasure codes in big-data systems, going beyond their conventional application domain of disk-based storage systems and as well as going beyond their traditional objective of fault tolerance. We presented EC-Cache, a cluster cache that employs erasure codes for load balancing and for reducing both the median and tail latencies for data cached in memory. This is just the tip of the iceberg, and there are many more interesting and exciting questions that remain to be answered.

In what follows, we discuss two specific potential future directions for erasure-coded in-memory systems. In EC-Cache, we focused on improving the performance for the working datasets that fit in the memory of the cluster cache. Situations where the working dataset does not fit in the memory trigger the caching and eviction dynamics. EC-Cache currently employs the standard least-recently-used (LRU) eviction policy. Its not known whether the LRU policy is optimal, and designing optimal caching/eviction algorithms suited for erasure-coded in-memory caching systems is a challenging open problem. A second interesting direction of future work is that of efficiently handling time-varying popularity of objects. In general, we expect erasure coding to be better suited to respond to time-varying skew than selective replication due to its ability to vary the overhead of an object in fractional increments as opposed to the limitation of having only integral increments in selective replication. It remains to investigate this conjecture as well as to design optimal algorithms for handling time-varying popularity skew in cluster caches.

Finally, in-memory caching is one of the numerous places where erasure coding has the potential to significantly enhance performance of big-data systems. Further opportunities for using erasure-coding as a tool in other layers of big-data systems, such as the execution layer and the application layer, remain to be explored.

List of Figures

List of Tables

Bibliography

[1] Sameer Agarwal et al. “BlinkDB: Queries with bounded errors and bounded response times on very large data”. In: *EuroSys*. 2013.

[2] Marcos Aguilera, Ramaprabhu Janakiraman, and Lihao Xu. “Using erasure codes efficiently for storage in a distributed system”. In: *International Conference on Dependable Systems and Networks (DSN’05)*. 2005, pp. 336–345.

[3] S. Akhlaghi, A. Kiani, and M. R. Ghanavati. “A Fundamental Trade-off Between The Download Cost And Repair Bandwidth In Distributed Storage Systems”. In: *Proc. IEEE International Symposium on Network Coding (NetCod)*. Toronto, June 2010.

[4] Mohammad Alizadeh et al. “CONGA: Distributed Congestion-Aware Load Balancing for Datacenters”. In: *SIGCOMM*. 2014.

[5] *Amazon EC2*. `http://aws.amazon.com/ec2`.

[6] *Amazon Simple Storage Service*. `http://aws.amazon.com/s3`.

[7] G. Ananthanarayanan et al. “PACMan: Coordinated Memory Caching for Parallel Jobs”. In: *NSDI*. 2012.

[8] G. Ananthanarayanan et al. “Reining in the Outliers in MapReduce Clusters using Mantri”. In: *OSDI*. 2010.

[9] G. Ananthanarayanan et al. “Scarlett: Coping with Skewed Popularity Content in MapReduce Clusters”. In: *EuroSys*. 2011.

[10] G. Ananthanarayanan et al. “Why let resources idle? Aggressive Cloning of Jobs with Dolly”. In: *USENIX HotCloud*. June 2012.

[11] Fabien André et al. “Archiving cold data in warehouses with clustered network coding”. In: *Proceedings of the Ninth European Conference on Computer Systems*. ACM. 2014, p. 21.

[12] *Apache Hadoop*. `http://hadoop.apache.org`.

[13] Martin Arlitt et al. “Evaluating content management techniques for web proxy caches”. In: *ACM SIGMETRICS Performance Evaluation Review* 27.4 (2000), pp. 3–11.

[14] Michael Armbrust et al. “Spark SQL: Relational data processing in Spark”. In: *SIGMOD.* 2015.

[15] M. Asteris et al. “XORing Elephants: Novel Erasure Codes for Big Data”. In: *PVLDB.* 2013.

[16] *AWS Innovation at Scale.* `https://www.youtube.com/watch?v=JIQETrFC_SQ`.

[17] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. “The datacenter as a computer: An introduction to the design of warehouse-scale machines”. In: *Synthesis lectures on computer architecture* 8.3 (2013), pp. 1–154.

[18] Doug Beaver et al. “Finding a Needle in Haystack: Facebook’s Photo Storage.” In: *OSDI.* Vol. 10. 2010, pp. 1–8.

[19] Laszlo A. Belady. “A study of replacement algorithms for a virtual-storage computer”. In: *IBM Systems journal* 5.2 (1966), pp. 78–101.

[20] R. Bhagwan et al. “Total Recall: System Support for Automated Availability Management”. In: *Proc. 1st conference on Symposium on Networked Systems Design and Implementation (NSDI).* 2004.

[21] M. Blaum et al. “EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures”. In: *IEEE Transactions on Computers* 44.2 (1995), pp. 192–202.

[22] Mario Blaum, Jehoshua Bruck, and Alexander Vardy. “MDS array codes with independent parity symbols”. In: *Information Theory, IEEE Transactions on* 42.2 (1996), pp. 529–542.

[23] P. Bodik et al. “Surviving Failures in Bandwidth-Constrained Datacenters”. In: *SIGCOMM.* 2012.

[24] Dhruba Borthakur. *The Hadoop Distributed File System: Architecture and Design.* Hadoop Project Website. 2007.

[25] Viveck R Cadambe et al. “Asymptotic interference alignment for optimal repair of MDS codes in distributed storage”. In: *Information Theory, IEEE Transactions on* 59.5 (2013), pp. 2974–2987.

[26] 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: *IEEE International Symposium on Information Theory (ISIT).* 2011, pp. 1225–1229.

[27] V.R. Cadambe et al. "Polynomial Length MDS Codes With Optimal Repair in Distributed Storage". In: *Forty Fifth Asilomar Conference on Signals, Systems and Computers*. Nov. 2011, pp. 1850–1854.

[28] B. Calder et al. "Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency". In: *SOSP*. 2011.

[29] R. Chaiken et al. "SCOPE: Easy and Efficient Parallel Processing of Massive Datasets". In: *VLDB*. 2008.

[30] Yanpei Chen, Sara Alspaugh, and Randy Katz. "Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads". In: 2012.

[31] Mosharaf Chowdhury, Srikanth Kandula, and Ion Stoica. "Leveraging Endpoint Flexibility in Data-Intensive Clusters". In: *SIGCOMM*. 2013.

[32] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. "Efficient Coflow Scheduling with Varys". In: *SIGCOMM*. 2014.

[33] Byung-Gon Chun et al. "Efficient replica maintenance for distributed storage systems". In: *NSDI*. Vol. 6. 2006, pp. 225–264.

[34] *Colossus, successor to Google File System.* *http://static.googleusercontent.com/media/research.google.com/en/us/university/relations/facultysummit2010/storage_architecture_and_challenges.pdf*.

[35] P. Corbett et al. "Row-Diagonal Parity for Double Disk Failure Correction". In: *Proc. 3rd USENIX Conference on File and Storage Technologies (FAST)*. 2004, pp. 1–14.

[36] Son Hoang Dau, Wentu Song, and Chau Yuen. "On block security of regenerating codes at the MBR point for distributed storage systems". In: *Information Theory (ISIT), 2014 IEEE International Symposium on*. IEEE. 2014, pp. 1967–1971.

[37] Jeffrey Dean. "Evolution and future directions of large-scale storage and computation systems at Google". In: *Proceedings of the 1st ACM symposium on Cloud computing*. ACM. 2010, pp. 1–1.

[38] Jeffrey Dean and Luiz André Barroso. "The tail at scale". In: *Communications of the ACM* 56.2 (2013), pp. 74–80.

[39] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *OSDI*. 2004.

[40] *DFS-Perf*. http://pasa-bigdata.nju.edu.cn/dfs-perf.

[41] A. G. Dimakis et al. "Network Coding for Distributed Storage Systems". In: *IEEE Transactions on Information Theory* 56.9 (2010), pp. 4539–4551.

[42] Aleksandar Dragojević et al. “FaRM: Fast Remote Memory”. In: *NSDI*. 2014.

[43] Iwan M Duursma. “Shortened regenerating codes”. In: *arXiv preprint arXiv:1505.00178* (2015).

[44] Salim El Rouayheb and Kannan Ramchandran. “Fractional Repetition Codes for Repair in Distributed Storage Systems”. In: *Allerton Conference on Control, Computing, and Communication*. Urbana-Champaign, Sept. 2010.

[45] Mehran Elyasi and Soheil Mohajer. “New exact-repair codes for distributed storage systems using matrix determinant”. In: *2016 IEEE International Symposium on Information Theory (ISIT)*. IEEE. 2016, pp. 1212–1216.

[46] Mehran Elyasi, Soheil Mohajer, and Ravi Tandon. “Linear exact repair rate region of (k+ 1, k, k) distributed storage systems: A new approach”. In: *2015 IEEE International Symposium on Information Theory (ISIT)*. IEEE. 2015, pp. 2061–2065.

[47] Eyal En Gad et al. “Repair-optimal MDS array codes over GF (2)”. In: *Information Theory Proceedings (ISIT), 2013 IEEE International Symposium on*. IEEE. 2013, pp. 887–891.

[48] Toni Ernvall et al. “Capacity and security of heterogeneous distributed storage systems”. In: *Selected Areas in Communications, IEEE Journal on* 31.12 (2013), pp. 2701–2709.

[49] Kyumars Sheykh Esmaili, Lluis Pamies-Juarez, and Anwitaman Datta. “CORE: Cross-object redundancy for efficient data repair in storage systems”. In: *IEEE International Conference on Big data*. 2013, pp. 246–254.

[50] *Evolution of the Netflix Data Pipeline*. `http://techblog.netflix.com/2016/02/evolution-of-netflix-data-pipeline.html`.

[51] B. Fan et al. “DiskReduce: RAID for data-intensive scalable computing”. In: *Proceedings of the 4th Annual Workshop on Petascale Data Storage*. ACM. 2009, pp. 6–10.

[52] Bin Fan, David G Andersen, and Michael Kaminsky. “MemC3: Compact and concurrent memcache with dumber caching and smarter hashing”. In: *NSDI*. 2013.

[53] Ulric J Ferner, Emina Soljanin, and Muriel Médard. “Why Reading Patterns Matter in Storage Coding &amp; Scheduling Design”. In: *2015 IEEE 8th International Conference on Cloud Computing*. IEEE. 2015, pp. 357–364.

[54] Tobias Flach et al. “Reducing Web Latency: the Virtue of Gentle Aggression”. In: *ACM SIGCOMM*. 2013.

[55] D. Ford et al. “Availability in globally distributed storage systems”. In: *USENIX Symposium on Operating Systems Design and Implementation.* 2010.

[56] Kristen Gardner et al. “Reducing latency via redundant requests: Exact analysis”. In: *ACM SIGMETRICS Performance Evaluation Review* 43.1 (2015), pp. 347–360.

[57] S. Ghemawat, H. Gobioff, and S.T. Leung. “The Google file system”. In: *ACM SIGOPS Operating Systems Review.* Vol. 37. 5. ACM. 2003, pp. 29–43.

[58] P. Gopalan et al. “On the locality of codeword symbols”. In: *IEEE Transactions on Information Theory* (Nov. 2012).

[59] Sreechakra Goparaju, Salim El Rouayheb, and Robert Calderbank. “Can linear minimum storage regenerating codes be universally secure?” In: *2015 49th Asilomar Conference on Signals, Systems and Computers.* IEEE. 2015, pp. 549–553.

[60] Sreechakra Goparaju, Salim El Rouayheb, and Robert Calderbank. “New codes and inner bounds for exact repair in distributed storage systems”. In: *2014 IEEE International Symposium on Information Theory.* IEEE. 2014, pp. 1036–1040.

[61] Sreechakra Goparaju, Itzhak Tamo, and Robert Calderbank. “An improved subpacketization bound for minimum storage regenerating codes”. In: *Information Theory, IEEE Transactions on* 60.5 (2014), pp. 2770–2779.

[62] Sreechakra Goparaju et al. “Data secrecy in distributed storage systems under exact repair”. In: *Network Coding (NetCod), 2013 International Symposium on.* IEEE. 2013, pp. 1–6.

[63] Albert Greenberg et al. “VL2: A Scalable and Flexible Data Center Network”. In: *SIGCOMM.* 2009.

[64] Venkatesan Guruswami and Mary Wootters. “Repairing Reed-solomon codes”. In: *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing.* ACM. 2016, pp. 216–226.

[65] *Hadoop Platform as a Service in the Cloud.* http://techblog.netflix.com/2013/01/hadoop-platform-as-service-in-cloud.html.

[66] *HDFS and Erasure Codes (HDFS-RAID).* *http://hadoopblog.blogspot.com/2009/08/hdfs-and-erasure-codes-hdfs-raid.html.*

[67] *HDFS-RAID. http://wiki.apache.org/hadoop/HDFS-RAID.*

[68] Yu-Ju Hong and Mithuna Thottethodi. “Understanding and mitigating the impact of load imbalance in the memory caching tier”. In: *Proceedings of the 4th annual Symposium on Cloud Computing.* ACM. 2013, p. 13.

[69] Hanxu Hou et al. “BASIC regenerating code: Binary addition and shift for exact repair”. In: *IEEE International Symposium on Information Theory (ISIT)*. IEEE. 2013, pp. 1621–1625.

[70] Y. Hu et al. “Cooperative Recovery of Distributed Storage Systems from Multiple Losses with Network Coding”. In: *IEEE Journal on Selected Areas in Communication* 28.2 (Feb. 2010), pp. 268–276.

[71] Y. Hu et al. “NCFS: On the practicality and extensibility of a network-coding-based distributed file system”. In: *International Symposium on Network Coding (NetCod)*. Beijing, July 2011.

[72] Yuchong Hu et al. “NCCloud: Applying network coding for the storage repair in a cloud-of-clouds”. In: *USENIX FAST*. 2012.

[73] C. Huang et al. “Erasure Coding in Windows Azure Storage”. In: *Proc. USENIX Annual Technical Conference (ATC)*. 2012.

[74] Cheng Huang et al. “Erasure Coding in Windows Azure Storage”. In: *USENIX ATC*. 2012.

[75] Kun Huang, Udaya Parampalli, and Ming Xian. “On Secrecy Capacity of Minimum Storage Regenerating Codes”. In: *arXiv preprint arXiv:1505.01986* (2015).

[76] L. Huang et al. “Codes Can Reduce Queueing Delay in Data Centers”. In: (July 2012).

[77] Qi Huang et al. “An analysis of Facebook photo caching”. In: *SOSP*. 2013.

[78] Qi Huang et al. “Characterizing load imbalance in real-world networked caches”. In: *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*. ACM. 2014, p. 8.

[79] *Intel Storage Acceleration Library (Open Source Version)*. `https://goo.gl/zkVl4N`.

[80] Steve Jiekak and Nicolas Le Scouarnec. “CROSS-MBCR: Exact minimum bandwith coordinated regenerating codes”. In: *arXiv preprint arXiv:1207.0854* (2012).

[81] Steve Jiekak et al. “Regenerating codes: A system perspective”. In: *ACM SIGOPS Operating Systems Review* 47.2 (2013), pp. 23–32.

[82] Gauri Joshi, Yanpei Liu, and Emina Soljanin. “On the delay-storage trade-off in content download from coded distributed storage systems”. In: *Selected Areas in Communications, IEEE Journal on* 32.5 (2014), pp. 989–997.

[83] Swanand Kadhe and Alex Sprintson. “On a weakly secure regenerating code construction for minimum storage regime”. In: *Communication, Control, and Computing (Allerton), 2014 52nd Annual Allerton Conference on*. IEEE. 2014, pp. 445–452.

[84] Govinda M Kamath et al. "Codes with local regeneration". In: *Information Theory and Applications Workshop (ITA), 2013*. IEEE. 2013, pp. 1–5.

[85] Govinda M Kamath et al. "Explicit MBR all-symbol locality codes". In: *IEEE International Symposium on Information Theory*. 2013, pp. 504–508.

[86] Srikanth Kandula et al. "Quickr: Lazily Approximating Complex AdHoc Queries in BigData Clusters". In: *SIGMOD*. 2016.

[87] Srikanth Kandula et al. "The Nature of Datacenter Traffic: Measurements and Analysis". In: *IMC*. 2009.

[88] Anne-Marie Kermarrec, Nicolas Le Scouarnec, and Gilles Straub. "Repairing multiple failures with coordinated and adaptive regenerating codes". In: *2011 International Symposium on Networking Coding*. IEEE. 2011, pp. 1–6.

[89] O. Khan et al. "Rethinking erasure codes for cloud file systems: minimizing I/O for recovery and degraded reads". In: *Proc. Usenix Conference on File and Storage Technologies (FAST)*. 2012.

[90] Daniel J Kleitman and Da-Lun Wang. "Algorithms for constructing graphs and digraphs with given valences and factors". In: *Discrete Mathematics* 6.1 (1973), pp. 79–88.

[91] J.C. Koo and JT Gill. "Scalable constructions of fractional repetition codes in distributed storage systems". In: *Communication, Control, and Computing (Allerton), 2011 49th Annual Allerton Conference on*. IEEE. 2011, pp. 1366–1373.

[92] M Nikhil Krishnan et al. "Evaluation of Codes with Inherent Double Replication for Hadoop". In: (2014).

[93] John Kubiatowicz et al. "Oceanstore: An architecture for global-scale persistent storage". In: *ACM Sigplan Notices* 35.11 (2000), pp. 190–201.

[94] Avinash Lakshman and Prashant Malik. "Cassandra: a decentralized structured storage system". In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40.

[95] Nicolas Le Scouarnec. "Exact scalar minimum storage coordinated regenerating codes". In: *IEEE International Symposium on Information Theory Proceedings (ISIT)*. 2012, pp. 1197–1201.

[96] Jeongkeun Lee et al. "Application-driven bandwidth guarantees in datacenters". In: *SIGCOMM*. 2014.

[97] Haoyuan Li et al. "Tachyon: Reliable, memory speed storage for cluster computing frameworks". In: *SoCC*. 2014.

[98] G. Liang and U.C. Kozat. “FAST CLOUD: Pushing the Envelope on Delay Performance of Cloud Storage with Coding”. In: *arXiv:1301.1294* (Jan. 2013).

[99] Hyeontaek Lim et al. “MICA: A holistic approach to fast in-memory key-value storage”. In: *NSDI*. 2014.

[100] Shu Lin and Daniel Costello. *Error control coding*. Prentice-hall Englewood Cliffs, 2004.

[101] *MemCached*. `http://www.memcached.org`.

[102] James Mickens and Brian Noble. “Exploiting availability prediction in distributed systems”. In: *NSDI*. 2006.

[103] Michael Mitzenmacher, Andrea W. Richa, and Ramesh Sitaraman. “The Power of Two Random Choices: A Survey of Techniques and Results”. In: *Handbook of Randomized Computing* (1 2001), pp. 255–312.

[104] Subramanian Muralidhar et al. “f4: Facebook’s warm BLOB storage system”. In: *OSDI*. 2014.

[105] Preetum Nakkiran, KV Rashmi, and Kannan Ramchandran. “Optimal systematic distributed storage codes with fast encoding”. In: *2016 IEEE International Symposium on Information Theory (ISIT)*. IEEE.

[106] Preetum Nakkiran, Nihar B Shah, and KV Rashmi. “Fundamental limits on communication for oblivious updates in storage networks”. In: *2014 IEEE Global Communications Conference*. IEEE. 2014, pp. 2363–2368.

[107] Preetum Nakkiran et al. “Optimal Oblivious Updates in Distributed Storage Networks”. In: (2016).

[108] Ed Nightingale et al. “Flat Datacenter Storage”. In: *OSDI*. 2012.

[109] Rajesh Nishtala et al. “Scaling Memcache at Facebook”. In: *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 2013, pp. 385–398.

[110] Frederique Oggier and Anwitaman Datta. “Self-repairing homomorphic codes for distributed storage systems”. In: *INFOCOM, 2011 Proceedings IEEE*. 2011, pp. 1215–1223.

[111] *OpenStack Swift*. `http://swift.openstack.org`.

[112] Kay Ousterhout et al. “Sparrow: Distributed, low latency scheduling”. In: *SOSP*. 2013.

[113] Lluis Pamies-Juarez, Cyril Guyot, and Robert Mateescu. "Spider Codes: Practical erasure codes for distributed storage systems". In: *2016 IEEE International Symposium on Information Theory (ISIT)*. IEEE. 2016, pp. 1207–1211.

[114] Lluis Pamies-Juarez, Henk DL Hollmann, and Frédérique Oggier. "Locally repairable codes with multiple repair alternatives". In: *Information Theory Proceedings (ISIT), 2013 IEEE International Symposium on*. IEEE. 2013, pp. 892–896.

[115] D Papailiopoulos, A Dimakis, and V Cadambe. "Repair Optimal Erasure Codes through Hadamard Designs". In: *IEEE Transactions on Information Theory* 59.5 (May 2013), pp. 3021–3037.

[116] D.S. Papailiopoulos and A.G. Dimakis. "Locally repairable codes". In: *Information Theory Proceedings (ISIT), 2012 IEEE International Symposium on*. IEEE. 2012, pp. 2771–2775.

[117] D. A. Patterson, G. Gibson, and R. H. Katz. "A Case for Redundant Arrays of Inexpensive Disks (RAID)". In: *Proc. ACM SIGMOD international conference on management of data*. Chicago, USA, June 1988, pp. 109–116.

[118] S. Pawar, S. El Rouayheb, and K. Ramchandran. "Securing dynamic distributed storage systems against eavesdropping and adversarial attacks". In: *IEEE Transactions on Information Theory* 57.10 (2011), pp. 6734–6753.

[119] S. Pawar et al. "DRESS Codes for the Storage Cloud: Simple Randomized Constructions". In: *Proc. IEEE International Symposium on Information Theory (ISIT)*. St. Petersburg, Aug. 2011.

[120] Mikko Juhani Pitkänen and Jörg Ott. "Redundancy and distributed caching in mobile dtns". In: *ACM/IEEE MobiArch*. 2007.

[121] James S Plank and Kevin M Greenan. *Jerasure: A library in C facilitating erasure coding for storage applications–version 2.0*. Tech. rep. Technical Report UT-EECS-14-721, University of Tennessee, 2014.

[122] James S Plank et al. *GF-Complete: A Comprehensive Open Source Library for Galois Field Arithmetic, Version 1.0*. Tech. rep. CS-13-716. University of Tennessee, 2013.

[123] N Prakash and M Nikhil Krishnan. "The Storage-Repair-Bandwidth Trade-off of Exact Repair Linear Regenerating Codes for the Case $d = k = n - 1$". In: *IEEE International Symposium on Information Theory (ISIT)*. 2015.

[124] *Presto: Distributed SQL Query Engine for Big Data*. `https://prestodb.io`.

[125] Qifan Pu et al. "FairRide: Near-Optimal, Fair Cache Sharing". In: *NSDI*. 2016.

[126] K. V. Rashmi, N. B. Shah, and P. V. Kumar. "Enabling Node Repair in Any Erasure Code for Distributed Storage". In: *Proc. IEEE International Symposium on Information Theory (ISIT).* July 2011.

[127] 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". In: *IEEE Transactions on Information Theory* 57.8 (Aug. 2011), pp. 5227–5239.

[128] K. V. Rashmi, N. B. Shah, and K. Ramchandran. "A Piggybacking Design Framework for Read-and Download-efficient Distributed Storage Codes". In: *IEEE International Symposium on Information Theory.* July 2013.

[129] K. V. Rashmi, N. B. Shah, and K. Ramchandran. "A Piggybacking Design Framework for Read-and Download-efficient Distributed Storage Codes". In: *IEEE International Symposium on Information Theory.* 2013.

[130] K. V. Rashmi et al. "A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers". In: *SIGCOMM.* 2015.

[131] K. V. Rashmi et al. "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.* June 2013.

[132] K. V. Rashmi et al. "Explicit Construction of Optimal Exact Regenerating Codes for Distributed Storage". In: *Proc. 47th Annual Allerton Conference on Communication, Control, and Computing.* Urbana-Champaign, Sept. 2009, pp. 1243–1249.

[133] K. V. Rashmi et al. "Regenerating Codes for Errors and Erasures in Distributed Storage". In: *Proc. International Symposium on Information Theory.* July 2012.

[134] KV Rashmi et al. "A Hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers". In: *Proceedings of the 2014 ACM conference on SIGCOMM.* ACM. 2014, pp. 331–342.

[135] KV Rashmi et al. "EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16).* 2016.

[136] KV Rashmi et al. "Having your cake and eating it too: jointly optimal erasure codes for I/O, storage, and network-bandwidth". In: *13th USENIX Conference on File and Storage Technologies (FAST 15).* 2015, pp. 81–94.

[137] Netanel Raviv, Natalia Silberstein, and Tuvi Etzion. "Constructions of high-rate MSR codes over small fields". In: *arXiv preprint arXiv:1505.00919* (2015).

[138] Ankit Singh Rawat et al. “Optimal locally repairable and secure codes for distributed storage systems”. In: *Information Theory, IEEE Transactions on* 60.1 (2014), pp. 212–236.

[139] *Redis*. `http://www.redis.io`.

[140] I.S. Reed and G. Solomon. “Polynomial Codes over certain Finite Fields”. In: *Journal of the Society for Industrial and Applied Mathematics* 8.2 (1960), pp. 300–304.

[141] Robbert van Renesse and Fred B. Schneider. “Chain Replication for Supporting High Throughput and Availability”. In: *OSDI*. 2004.

[142] S. Rhea et al. “Pond: The OceanStore prototype”. In: *Proc. 2nd USENIX conference on File and Storage Technologies (FAST)*. 2003, pp. 1–14.

[143] R. Rodrigues and B. Liskov. “High Availability in DHTs: Erasure coding vs. Replication”. In: *Proc. International Workshop on Peer-To-Peer Systems*. Feb. 2005, pp. 226–239.

[144] Birenjith Sasidharan, Gaurav Kumar Agarwal, and P Vijay Kumar. “A high-rate MSR code with polynomial sub-packetization level”. In: *2015 IEEE International Symposium on Information Theory (ISIT)*. IEEE. 2015, pp. 2051–2055.

[145] Birenjith Sasidharan, Kaushik Senthoor, and P Vijay Kumar. “An improved outer bound on the storage-repair-bandwidth tradeoff of exact-repair regenerating codes”. In: *Information Theory (ISIT), 2014 IEEE International Symposium on*. IEEE. 2014, pp. 2430–2434.

[146] Mahesh Sathiamoorthy et al. “XORing Elephants: Novel Erasure Codes for Big Data”. In: *VLDB Endowment*. 2013.

[147] *Seamless Reliability. http://www.cleversafe.com/overview/reliable*. Feb. 2014.

[148] N. B. Shah, K. V. Rashmi, and P. V. Kumar. “Information-theoretically Secure Regenerating Codes for Distributed Storage”. In: *Proc. Globecom*. Houston, Dec. 2011.

[149] N. B. Shah et al. “Distributed Storage Codes with Repair-by-Transfer and Non-achievability of Interior Points on the Storage-Bandwidth Tradeoff”. In: *IEEE Transactions on Information Theory* 58.3 (Mar. 2012), pp. 1837–1852.

[150] Nihar B. Shah. “On Minimizing Data-read and Download for Storage-Node Recovery”. In: *IEEE Communications Letters* (2013).

[151] Nihar B Shah, Kangwook Lee, and Kannan Ramchandran. “The MDS queue: Analysing the latency performance of erasure codes”. In: *2014 IEEE International Symposium on Information Theory*. IEEE. 2014, pp. 861–865.

[152] Nihar B Shah, Kangwook Lee, and Kannan Ramchandran. "When do redundant requests reduce latency?" In: *IEEE Transactions on Communications* 64.2 (2016), pp. 715–722.

[153] Nihar B. Shah, K. V. Rashmi, and P. Vijay Kumar. "A Flexible Class of Regenerating Codes for Distributed Storage". In: *Proc. IEEE International Symposium on Information Theory (ISIT)*. Austin, June 2010, pp. 1943–1947.

[154] Nihar B Shah, KV Rashmi, and Kannan Ramchandran. "One extra bit of download ensures perfectly private information retrieval". In: *2014 IEEE International Symposium on Information Theory*. IEEE. 2014, pp. 856–860.

[155] Nihar B. Shah et al. "Explicit Codes Minimizing Repair Bandwidth for Distributed Storage". In: *Proc. IEEE Information Theory Workshop (ITW)*. Cairo, Jan. 2010.

[156] Nihar B. Shah et al. "Interference Alignment in Regenerating Codes for Distributed Storage: Necessity and Code Constructions". In: *IEEE Transactions on Information Theory* 58.4 (Apr. 2012), pp. 2134–2158.

[157] Kenneth W Shum and Yuchong Hu. "Exact minimum-repair-bandwidth cooperative regenerating codes for distributed storage systems". In: *Information Theory Proceedings (ISIT), 2011 IEEE International Symposium on*. IEEE. 2011, pp. 1442–1446.

[158] Konstantin Shvachko et al. "The Hadoop distributed file system". In: *IEEE Symposium on Mass Storage Systems and Technologies (MSST)*. 2010.

[159] N. Silberstein, A.S. Rawat, and S. Vishwanath. "Adversarial Error Resilience in Distributed Storage Using MRD Codes and MDS Array Codes". In: *arXiv:1202.0800* (2012).

[160] Natalia Silberstein and Tuvi Etzion. "Optimal fractional repetition codes based on graphs and designs". In: *IEEE Transactions on Information Theory* 61.8 (2015), pp. 4164–4180.

[161] Natalia Silberstein, Ankit Singh Rawat, and Sriram Vishwanath. "Error-Correcting Regenerating and Locally Repairable Codes via Rank-Metric Codes". In: *Information Theory, IEEE Transactions on* 61.11 (2015), pp. 5765–5778.

[162] Arjun Singh et al. "Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network". In: SIGCOMM. 2015.

[163] Christopher Stewart, Aniket Chakrabarti, and Rean Griffith. "Zoolander: Efficiently meeting very strict, low-latency SLOs". In: *USENIX ICAC*. 2013.

[164] C. Suh and Kannan Ramchandran. "Exact-Repair MDS Code Construction Using Interference Alignment". In: *IEEE Transactions on Information Theory* (Mar. 2011), pp. 1425–1442.

[165] Yin Sun et al. "Provably delay efficient data retrieving in storage clouds". In: *2015 IEEE Conference on Computer Communications (INFOCOM)*. IEEE. 2015, pp. 585–593.

[166] *Support append operation after completing a file.* `https://alluxio.atlassian.net/browse/ALLUXIO-25`.

[167] Itzhak Tamo and Alexander Barg. "A family of optimal locally recoverable codes". In: *IEEE Transactions on Information Theory* 60.8 (2014), pp. 4661–4676.

[168] Itzhak Tamo, Zhiying Wang, and Jehoshua Bruck. "Access Versus Bandwidth in Codes for Storage". In: *IEEE Transactions on Information Theory* 60.4 (2014), pp. 2028–2037.

[169] Itzhak Tamo, Zhiying Wang, and Jehoshua Bruck. "Zigzag codes: MDS array codes with optimal rebuilding". In: *Information Theory, IEEE Transactions on* 59.3 (2013), pp. 1597–1616.

[170] Ravi Tandon et al. "Towards Optimal Secure Distributed Storage Systems with Exact Repair". In: *arXiv preprint arXiv:1310.0054* (2013).

[171] Ashish Thusoo et al. "Data Warehousing and Analytics Infrastructure at Facebook". In: *SIGMOD*. 2010.

[172] Chao Tian. "Characterizing the rate region of the (4, 3, 3) exact-repair regenerating codes". In: *Selected Areas in Communications, IEEE Journal on* 32.5 (2014), pp. 967–975.

[173] Chao Tian et al. "Layered exact-repair regenerating codes via embedded error correction and block designs". In: *IEEE Transactions on Information Theory* 61.4 (2015), pp. 1933–1947.

[174] Cristian Ungureanu et al. "HydraFS: A High-Throughput File System for the HYDRAstor Content-Addressable Storage System." In: *FAST*. Vol. 10. 2010, pp. 225–239.

[175] Shivaram Venkataraman et al. "The Power of Choice in Data-Aware Cluster Scheduling". In: *OSDI*. 2014.

[176] A. Vulimiri et al. "More is Less: Reducing Latency via Redundancy". In: *11th ACM Workshop on Hot Topics in Networks*. Oct. 2012, pp. 13–18.

[177] Da Wang, Gauri Joshi, and Gregory Wornell. "Efficient task replication for fast response times in parallel computation". In: *ACM SIGMETRICS Performance Evaluation Review*. Vol. 42. 1. ACM. 2014, pp. 599–600.

[178] Z. Wang, A. Dimakis, and J. Bruck. "Rebuilding for Array Codes in Distributed Storage Systems". In: *Workshop on the Application of Communication Theory to Emerging Memory Technologies (ACTEMT)*. Dec. 2010.

[179] Z. Wang, I. Tamo, and J. Bruck. "On codes for optimal rebuilding access". In: *Communication, Control, and Computing (Allerton), 2011 49th Annual Allerton Conference on*. 2011, pp. 1374–1381.

[180] Hakim Weatherspoon and John D Kubiatowicz. "Erasure coding vs. replication: A quantitative comparison". In: *International Workshop on Peer-to-Peer Systems*. Springer, 2002, pp. 328–337.

[181] Sage A Weil et al. "Ceph: A scalable, high-performance distributed file system". In: *OSDI*. 2006.

[182] Yunnan Wu. "Existence and Construction of Capacity-Achieving Network Codes for Distributed Storage". In: *IEEE Journal on Selected Areas in Communications*. Vol. 28. 2. 2010, pp. 277–288.

[183] L. Xiang et al. "Optimal recovery of single disk failure in RDP code storage systems". In: *ACM SIGMETRICS*. Vol. 38. 1. 2010, pp. 119–130.

[184] Yu Xiang et al. "Joint latency and cost optimization for erasurecoded data center storage". In: *ACM SIGMETRICS Performance Evaluation Review* 42.2 (2014), pp. 3–14.

[185] Min Ye and Alexander Barg. "Explicit constructions of MDS array codes and RS codes with optimal repair bandwidth". In: *2016 IEEE International Symposium on Information Theory (ISIT)*. IEEE. 2016, pp. 1202–1206.

[186] Yuan Yu et al. "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language". In: *OSDI*. 2008.

[187] M. Zaharia et al. "Discretized Streams: Fault-Tolerant Stream Computation at Scale". In: *SOSP*. 2013.

[188] M. Zaharia et al. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". In: *NSDI*. 2012.

[189] Matei Zaharia et al. "Improving MapReduce Performance in Heterogeneous Environments". In: *OSDI*. 2008.

[190] Heng Zhang, Mingkai Dong, and Haibo Chen. "Efficient and available in-memory KV-store with hybrid erasure coding and replication". In: *14th USENIX Conference on File and Storage Technologies (FAST 16)*. 2016, pp. 167–180.

[191] Zhe Zhang et al. “Does erasure coding have a role to play in my data center”. In: *Microsoft research MSR-TR-2010* 52 (2010).