Doctoral thesis

Designing storage codes for heterogeneity: theory and practice

Francisco Maturana

CMU-CS-23-134

September, 2023

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:

Rashmi Vinayak, Chair
Gregory R. Ganger
Ryan O'Donnell
Muriel Médard, Massachusetts Institute of Technology

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored in part by the National Science Foundation under award numbers 1901410, 1943409, and 1956271, in part by a Google Faculty Research award, and in part by a Facebook distributed systems research award. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Coding theory, Data storage, Distributed storage systems, Erasure coding

Abstract

Data is at the heart of many of the services that society relies on. As a consequence, distributed storage systems (DSSs), which store data, are a fundamental part of most applications. Because of their essential role, these systems need to be extremely reliable. On top of this, DSSs need to be able to scale and grow incrementally to satisfy the increasing demand for storage. It is common for DSS deployments to be massive in size: just a single deployment can often store Petabytes of data and manage tens of thousands of disks. Supporting large-scale systems requires a large amount of resources (such as hardware, energy, physical space, and personnel) and results in significant capital and operating expenditures. For this reason, making these systems efficient is important as even small improvements can have a large impact.

DSSs commonly use *erasure coding* to tolerate failures. Typically, the operator of the DSS will choose the type of erasure code and its parameters based on the expected cluster conditions and their target metrics. However, cluster conditions are vastly heterogeneous and subject to significant variations across time. Current systems handle this by using extremely conservative amounts of redundancy and by relying on unplanned interventions, both of which are very costly. This thesis focuses on solving this problem by making DSSs more robust, by enabling them to automatically adapt to heterogeneity and variations across time and space.

To make progress towards our goal, we develop and use tools from both Coding Theory and Computer Systems. Our approach goes in both directions: we model the system, identify the fundamental theoretical questions at its heart, and apply the answers to these questions to develop better systems.

The first part focuses on variations over time that affect the DSS. We propose *convertible codes*, a theoretical framework for studying code conversion, i.e. the process of converting already-encoded data into a different encoding. Using this framework, we derive lower bounds on the cost of conversion for multiple metrics and propose optimal code constructions. Additionally, we design two DSSs, named Pacemaker and Tiger, that manage data and carefully choose *when* and *how* to convert data to guarantee a target level of reliability (in spite of the variations in the failure rate of disks) without overwhelming the cluster.

The second part focuses on heterogeneity across different parts of the DSS. We consider the setting of a geo-distributed storage system, where latencies between nodes vary significantly and the cost of sending data across the wide-area network (WAN) is important. We model the problem theoretically and study the tradeoff between storage overhead and WAN bandwidth usage. Using this model, we propose a construction for codes that jointly minimize storage overhead and WAN bandwidth usage. Finally, we use this theoretical framework to design and implement a strongly-consistent geo-distributed storage system that co-optimizes its erasure code and configuration to minimize cost.

Acknowledgements

I would like to express my deepest gratitude to my advisor and collaborator Prof. Rashmi Vinayak: not only was her help and guidance essential for the research in this thesis, but her kindness and cordiality also made my PhD journey very enjoyable and gratifying. I would like to deeply thank my thesis committee for their valuable comments, which helped shape this thesis. I would also like to thank all my collaborators whose work is featured in this thesis, and from whom I learnt many useful things (alphabetically): Sanjith Athlur, Mosharaf Chowdhury, Gregory R. Ganger, Saurabh Kadekodi, Harsha V. Madhyastha, Arif Merchant, V. S. Chaitanya Mukka, Suhas Jayaram Subramanya, Muhammed Uluyol, Juncheng Yang. In addition, I would like to thank the many people who helped me throughout this journey: Michael Rudow, for many useful technical discussions, reviewing many of my drafts, and being a personal friend; Andrew Park and Wenting Zheng, for collaborating with me in a project not included in this thesis; Austin Ramos, Timothy Kim, Dax Vandevoorde, Shaobo Guan, Jiaan Dai, Xuren Zhou, Jiaqi Zuo, Sai Kiriti Badam, and Jiongtao Ye, for their help in past or ongoing storage systems projects; Saransh Chopra and Justin Zhang, for continuing my work on some of the theoretical problems presented in this thesis; Jean-Sébastien Légaré and Andrew Warfield, for mentoring me during my internship at Amazon; Lluis Pamies-Juarez, Mustafa Uysal, and Arif Merchant, for mentoring me during my internship at Google and for facilitating access to some of the data used in this thesis; Keith Smith, Tim Emami, Jason Henessey, and Peter Macko, for mentoring me during my internship at Netapp and for facilitating access to some of the data used in this thesis; Cristian Riveros, Domagoj Vrgoč, and Marcelo Arenas, for helping me get started in computer science research before starting my PhD. I would also like to thank all the anonymous reviewers at various conferences and journals, whose comments helped me improve the work contained in these pages. I am thankful to all the thinkers and researchers that came before me, for sharing their knowledge with the rest of the world and enabling me to make this research, and for all those that will come after me (especially those that will cite my work).

I am also thankful to the following organizations for providing the funding to

support my research: the National Science Foundation through award numbers 1901410, 1943409, and 1956271, Google through the Faculty Research award, and Meta through the Facebook distributed systems research award.

Finally, I am deeply grateful to my family for their continued support, and to my friends, for cheering me up and helping me keep my sanity throughout the PhD: Gaurav Manek, Po Bhattacharyya, Jordan Barría, Pablo Guarda, Francisca Espinoza, Silvana Juri, Vicente Christian, Esteban Iglesias, Vicente Baeza, Chuli, Sebastián Salata, Francisco Carrasco, Alberto Croquevielle, Sebastián De Vidts, David Fuller, Carlos Brunner, Miguel Fadić, and many, many more.

Contents

Introduction

Many of today's most important and popular applications require storing ever-increasing amounts of data. Such storage needs far surpass what can be handled by a single machine, and thus many applications have to rely on distributed storage systems (DSSs), which store data across large numbers of devices. Because of the essential role they play in supporting other applications, DSSs need to be extremely reliable: they must guarantee that data can be readily accessed when needed, and that it will not be irrecoverably lost. Additionally, because of the constant increase in the demand for storage, DSSs need to be able to scale and allow operators to easily increase their capacity by adding more storage devices.

These strict requirements, and the logistical challenges attached to managing and running the required hardware, make DSSs hard to operate. As a consequence, most large-scale DSSs are operated by organizations that store very large amounts of data and/or offer storage services to other organizations. Therefore, it is common for DSSs deployments to be massive in size: just a single deployment can often store Petabytes of data and manage tens of thousands of disks. Supporting such a large number of disks requires specialized infrastructure capable of powering and cooling the disks as well as the machines that read, write, process, and communicate the data stored in them. DSSs thus consume vast amounts of resources in the form of hardware, electricity, computational resources, network traffic, etc. Given this situation, a very active line of research is to make these systems more efficient, as even small reductions in resource usage can have big impacts due to the sheer scale of these systems.

One key challenge that DSSs must handle is failure tolerance. In large-scale systems, failures are common and unavoidable events. Therefore, DSSs need to

employ techniques that allow them to gracefully handle and repair failures. This is typically expressed through two properties: durability, which is the ability to prevent data from becoming irrecoverably lost; and availability, which is the ability to guarantee that data can be accessed (in a reasonable amount of time). The simplest way to provide these properties is to replicate the data across different disks that are unlikely to fail together. However, at such large scales replication is economically unfeasible because it doubles or triples (or more) the amount of resources needed.

A more efficient alternative to replication is erasure coding, which can achieve the same level of failure tolerance with much lower storage overhead. On the flip side, the use of erasure codes in storage applications brings up a variety of other considerations which are not present with replication: e.g. encoding/decoding complexity, the repair of failed nodes, cost of updating data, impact of stragglers on latency, etc. However, in most cases the benefits of erasure coding in DSSs far outweigh the costs of dealing with their complications, and for this reason it is very common nowadays for DSSs to use erasure codes. The operator of a DSS typically chooses the type of erasure code and its parameters according to the expected operating conditions (such as the node failure rate, the workload, the network topology, etc.), and their target metrics (such as durability, availability, storage overhead, etc.).

One important unresolved problem that DSSs have to face in practice is having to adapt to the heterogeneities in the environments where they operate, and their variations across time. Current systems handle these differences by using extremely conservative amounts of redundancy when initially encoding the date, and by relying on unplanned manual interventions. Both of these are very costly: the unnecessary redundancy consumes a lot of resources and manual interventions, because they are unplanned, disrupt the normal functioning of the system and require significant effort. My thesis focuses on solving this problem: making DSSs more robust, by enabling them to automatically adapt to heterogeneity and variations across time.

To make progress towards making DSSs more robust to variations, we develop and use tools from both the Coding Theory and Computer Systems research. Our approach goes in both directions: we model different aspects of the problem and identify the fundamental theoretical questions at their heart, and we apply the answers

to these questions to develop better systems.

In the first part (Part I), we focus on enabling DSSs to adapt to *heterogeneity across time*. Data in DSSs typically lives for long periods of time, and over this period of time the cluster environment can vary significantly. This means that the erasure code that was used when the data was originally stored may become unsuitable due to variations in the cluster environment. For example, the failure rate of disks varies as they age, and the popularity of each file changes over time. DSSs can adapt to such variations via *redundancy tuning*, i.e., automatically converting already-encoded data into a different erasure code that is suitable. Despite bringing significant benefits, redundancy tuning can be hard to adopt in practice because existing DSSs are not designed to support it. The default approach to conversion is to read, decode, re-encode, and write data, which is very costly. Therefore, if performed carelessly, the work required for conversion can easily overwhelm the cluster. To bridge this gap, we approach the problem from two complementing directions: we design erasure codes that reduce the cost of conversion, and we design systems that manage the data and can effectively and efficiently decide *when* and *how* to convert data.

On the theoretical side, we develop the *convertible codes* framework, which allows us to precisely study the fundamental costs of converting data between two erasure codes with different parameters. We consider two types of cost: *access cost*, which measures the number of nodes that have to be read or written during conversion, and *conversion bandwidth*, which measures the amount of data that needs to be transferred between nodes during conversion. For each of these costs, we derive lower bounds on the cost of conversion, and we construct erasure codes that can perform conversion more efficiently than the default approach, in many cases achieving the optimal cost. In our optimization of these costs, we first focus on the so-called *maximum-distance separable* (MDS) codes, which are commonly used in practice because they achieve the minimum storage-overhead for a given level of failure tolerance. Then, we focus on a different class of codes known as *locally repairable codes* (LRCs), which are also widely used in practice and trade off store-overhead for better repair performance (which translates into higher availability).

On the practical side, we focus on designing DSSs that efficiently manage and

convert data to guarantee a target level of reliability in spite of variation in the failure rate of disks. A previous system called HeART [1] showed that this style of redundancy tuning can yield savings in storage space of up to 16% compared to a system that uses a single fixed erasure code chosen to tolerate the highest failure rate observed. However, trace-based simulations show that the work required by HeART can completely overwhelm a cluster for periods of days or weeks. We propose two systems that improve upon HeART by providing similar savings in storage space but without overwhelming the cluster. We first propose *Pacemaker*, which employs two main strategies: (1) it places stripes across disks that have similar failure rates, and (2) it proactively transitions data by observing trends in failure rates and anticipating large transitions. While Pacemaker is effective at avoiding overwhelming the cluster, it poses undesirable constraints on data placement. To address this, we propose Tiger, which is able to avoid transition overload without additional placement restrictions. To achieve this, Tiger introduces a new abstraction called *eclectic stripe*, which can precisely measure the reliability of a stripe composed of devices with different failure rates. This is enabled by a new system architecture and a set of techniques which allow Tiger to reliably manage and transition eclectic stripes without large overheads.

In the second part (Part II), we focus on enabling DSSs to adapt to *heterogeneity across space*. The setting that we consider is that of a geo-distributed storage system, in which users in different parts of the world to read and write to a shared set of objects. For example, this could correspond to data stored in a cloud storage system used in a collaborative application, such as Google Docs [2], or Overleaf [3]. Two important costs in this setting are: (1) storage overhead, and (2) wide-area network (WAN) bandwidth. One important objective of the system is to provide good read latency to users, which is typically achieved by placing replicas of the objects at data sites in different locations. To reduce the storage overhead, recent work [4–6] has proposed using Reed-Solomon codes instead. This, however, leads to high WAN bandwidth usage, because all parities of the code have to be updated whenever a write is made. Inspired by this problem, we pose the following question: *is it possible to design codes that minimize both storage overhead and WAN bandwidth required by updates?* To answer this question, we model the problem from a coding-theoretical

perspective. A key characteristic of this problem is the heterogeneity across space: users will typically contact the data sites that are closest to them, and some regions are better connected than others. To capture this, we use the notion of *access sets*, i.e, sets of nodes that must be able to decode the store object. Given a collection of access sets, we derive lower bounds on storage overhead and update cost, and study the tradeoffs between the two of them. Since WAN bandwidth tends to be more costly than storage in practice, we focus on codes that first minimize update cost, and then minimize storage overhead subject to that, which we term *minimum update cost* (MUC) codes. We fully characterize the update cost and storage overhead of MUC codes and provide a randomized construction. Then, we propose *Pudu*, a novel strongly-consistent geo-distributed storage system. Pudu implements and integrates the MUC framework to tailor the system's erasure code. Using this approach, Pudu is able to minimize the resource-cost of the system by co-optimizing both the design of the erasure code, and the design of the consensus protocol. This allows Pudu to achieve lower costs that were unachievable by prior systems.

The rest of this document is divided as follows. Part I focuses on heterogeneity across time. Chapters 1 to 4 focus on convertible codes, and Chapter 5 focuses on adapting distributed storage systems to perform erasure code changes more efficiently. Part II focuses on heterogeneity across space: Chapter 6 focuses on the theoretical results on codes with minimum update cost, and Chapter 7 incorporates these codes into a strongly-consistent geo-distributed storage system. Finally, Part III discusses future directions for the topics explored in this thesis.

Part I

Dynamic storage codes for change across time

In the first part of the thesis, we concern ourselves with *changes in storage codes across time*. In other words, we focus on studying erasure codes and distributed storage systems that change the encoding of data throughout time. We approach this subject from two perspectives: a coding-theoretical perspective, focused on the design of storage codes, and a systems perspective, focused on the design of a distributed storage system that adapts throughout time.

Chapters 1 to 4 are dedicated to *convertible codes*. In Chapter 1, we introduce the code conversion problem, and propose the convertible codes framework: using this framework, we study the access cost of the merge regime. In Chapter 2, we study the access cost of conversion in MDS codes. In Chapter 3, we introduce the *conversion bandwidth* cost metric, and study it in MDS codes. In Chapter 4, we go beyond MDS codes, and study the conversion bandwidth of *locally-repairable codes*.

Chapter 5 is dedicated to the concept of *disk-adaptive redundancy tuning*, and to system designs that implement it. First in Section 5.1, we propose Pacemaker, a distributed storage system designed to automatically change the encoding of data in response to changes in disk failure rates without overwhelming the system. Then in Section 5.10, we propose Tiger, a distributed storage system which is also designed to automatically adapt to changes in failure rate, but imposes fewer constraints and is more robust than Pacemaker.

Chapter 1

The convertible codes framework: enabling efficient conversion of coded data in distributed storage

This chapter is based on work from [7], done in collaboration with K. V. Rashmi.

1.1 Introduction

Erasure codes have become an essential tool for protecting against node failures in distributed storage systems [8–14]. Under erasure coding, a set of k data symbols to be stored is encoded using an $[n, k]$ code to generate n coded symbols, called a *codeword* (or *stripe*). Each of the n symbols in a codeword is stored on a different storage node, and the system as a whole typically contains several independent codewords distributed across different subsets of storage nodes in the cluster.

A key factor that determines the choice of parameters n and k is the failure rate of the storage devices. It has been shown that failure rates of storage devices in large-scale storage systems can vary significantly over time and that changing the code rate, by changing n and k, in response to these variations yields substantial savings in storage space and hence the operating costs [1]. For example, in [1], the authors show that an 11% to 44% reduction in storage space can be achieved by tailoring

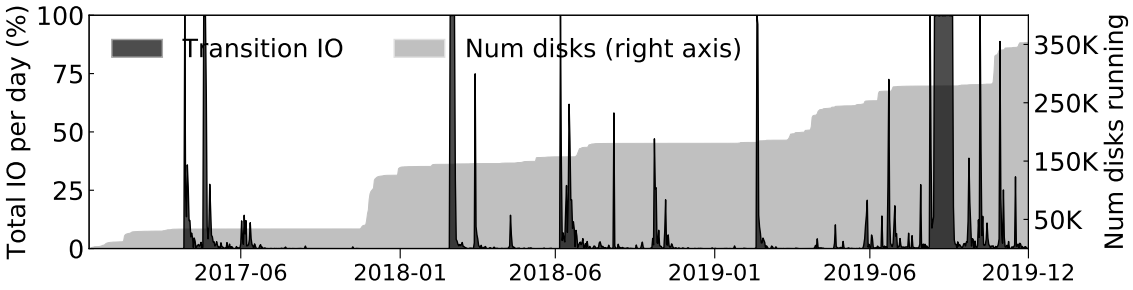


Figure 1.1: (From [15].) The left y-axis shows the percentage of disk IO utilized by conversion (called "transition" in [15]) against time simulated from a trace of a production Google cluster. The right y-axis shows the size of the cluster in number of disks against time. Code conversions can result in big spikes in disk IO consumption that can overwhelm the cluster for several days.

n and k to changes in observed device failure rates. Such a reduction in storage space requirement translates to significant savings in the cost of resources and energy consumed in large-scale storage systems. It is natural to think of potentially achieving such a change in code rate by changing only n while keeping k fixed. However, due to several practical system constraints, changing code rate in storage systems often necessitates change in both the parameters n and k [1]. We refer the reader to [1] for a more detailed discussion on the practical benefits and constraints of adapting the erasure-code parameters to the variations in failure rates in storage systems.

Changing n and k for codewords in a storage system, from $[n^I, k^I]$, to $[n^F, k^F]$, would involve converting already encoded data from one code to another. Clearly, it is always possible to *re-encode* the data in a codeword according to a new code by accessing (and decoding if necessary) all the original message symbols. However, such an approach, which we call the *default approach*, requires accessing a large number of symbols (for example, for MDS codes, the initial value of k number of symbols need to be accessed from each codeword), reading out all the data, transferring over the network, and re-encoding. Such conversions can generate a large amount of load on cluster resources, which adversely affects the foreground operations of the cluster. Figure 1.1 shows the IO load that would be caused by code conversions on a Google cluster with multiple hundreds of thousands of disks [15]. As seen from the figure,

IO load from conversions can easily overwhelm the cluster for long periods of time. Furthermore, in some cases conversions might need to be performed in an expedited manner, for example, to avoid the risk of data loss when facing an unexpected rise in failure rate.

High IO load is problematic for such conversions because it slows down conversion as well as other important cluster processes, such as serving client requests. While recent work [15] has initiated a study on systems techniques to mitigate the spikes in the IO load caused by conversions, the total amount of work necessary for conversion still remains considerably high and these systems techniques introduce restrictions on other operations of the cluster such as data placement. Given that the root cause of the problem is the high resource overhead involved in performing conversions on the underlying code, we investigate the problem from a fundamental theoretical perspective.

There are also several other reasons to perform code conversions in storage systems. One may convert data that is frequently read into a code with a small k (in order to improve the performance of reconstructions) and convert data that is infrequently read into a code with large k (to achieve lower storage overhead). In addition, code conversions may need to be performed to keep the total size of the encoded data under a given threshold, or to maximize the reliability given the available storage space.

To the best of our knowledge, the existing literature [16–19] which formally studies the problem of changing the length and dimension of already encoded data does so from the perspective of the so-called *scaling problem*. The scaling problem [16] refers to the problem of evenly redistributing each codeword in a distributed storage system when additional nodes are added to the system and the level of failure tolerance (specifically, $(n-k)$) is kept constant. Some works [17, 19] generalize the scaling problem to broader cases where $(n-k)$ need not remain constant. However, even in cases where the scaling problem could be used to perform code conversion, it has several drawbacks that make it inefficient for conversion. For example, using the approach of scaling to achieve conversion requires accessing every symbol in each codeword and performing a significant amount of data movement to keep the amount

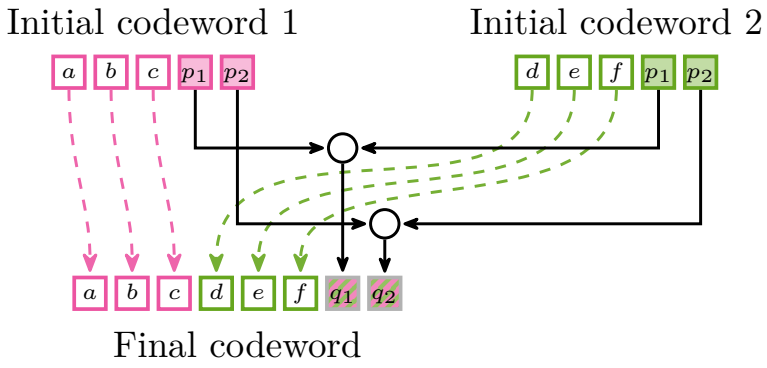


Figure 1.2: Example of code conversion: two codewords of a $[5, 3]$ MDS code are converted into one codeword of a $[8, 6]$ MDS code. Unshaded boxes represent data symbols, and shaded boxes represent parity symbols. Some of the initial symbols are kept unchanged in the final codewords, as shown by the dashed arrows. Some initial symbols are read and downloaded (solid arrows). The downloaded data is then used to compute and write the remaining symbols in the final codewords.

stored in each node the same. While these costs are necessary to fulfill the goals of the scaling problem, they are unnecessary to achieve code conversion, making this approach inefficient. A more detailed discussion of the scaling problem and other related work is provided in Section 1.2.1.

In this chapter, we propose a theoretical framework to model the code conversion problem. Our approach is based on the insight that the problem of changing code parameters in a storage system can be viewed as converting *multiple codewords* of an $[n^I, k^I]$ code (denoted by $\mathcal{C}^I$) into (potentially multiple) codewords of an $[n^F, k^F]$ code (denoted by $\mathcal{C}^F$)[1], with desired constraints on decodability, such as both codes satisfying the maximum distance separability (MDS) property. To address the problem of code conversion, we then introduce a new class of codes, which we call *convertible codes*, that allow for resource-efficient conversions. The general formulation of code conversions provides a powerful framework to theoretically study convertible codes.

We now present an example to elucidate the concept of code conversion in the

[1]The superscripts I and F stand for initial and final respectively, representing the initial and final state of the conversion.

convertible codes framework.

Example 1.1: Consider conversion from an $[n^I = 5, k^I = 3]$ code $\mathcal{C}^I$ to an $[n^F = 8, k^F = 6]$ code $\mathcal{C}^F$. We will focus on the number of symbols read, i.e. *read access cost*, and on the number symbols written, i.e. *write access cost*, for conversion. The default approach to conversion is to read $k^I = 3$ symbols from each of the two initial codewords belonging to $\mathcal{C}^I$, decoding the original data, and using it to write two symbols of the final codeword belonging to $\mathcal{C}^F$, while keeping the three read symbols from each initial codeword unchanged as symbols of the final codeword. Thus, the default approach has a read access cost of 6 and write access cost of 2.

In the convertible codes framework, this conversion is achieved by converting two codewords of the initial code into a single codeword of the final code, as depicted in Figure 1.2. This approach uses specially designed systematic codes $\mathcal{C}^I$ and $\mathcal{C}^F$. Let $\mathbb{F}_q$ be the finite field of size $q = 37$. Let $a, b, c \in \mathbb{F}_q$ be the data symbols of the first initial codeword, $d, e, f \in \mathbb{F}_q$ be the data symbols of the second initial codeword. Let $p_1, p_2 : \mathbb{F}_q^3 \to \mathbb{F}_q$ be the parity functions for the initial code $\mathcal{C}^I$, and $q_1, q_2 : \mathbb{F}_q^6 \to \mathbb{F}_q$ be the parity functions for the final code $\mathcal{C}^F$. The parity functions are chosen as below:

$$p_1(a, b, c) = a + b + c, \quad p_2(a, b, c) = a + 2b + 4c.$$

This is an example of the general construction presented in Section 2.2. The conversion procedure keeps the data symbols from each initial codeword unchanged in the final codeword, and then constructs the first (resp. second) parity of the final codeword as a linear combination of the first (resp. second) parity of each initial codeword. The final parity functions are chosen to satisfy the equation below:

$$\begin{aligned} q_1(a, b, c, d, e, f) &= p_1(a, b, c) + p_1(d, e, f), \\ q_2(a, b, c, d, e, f) &= p_2(a, b, c) + 8p_2(d, e, f). \end{aligned}$$

It is straightforward to check that the initial and final codes defined by these parity functions have the MDS property. This conversion procedure requires reading two symbols from each initial codeword and writing two symbols, resulting in a total read

access cost of 4 and a write access cost of 2, a reduction of $33.\overline{3}\%$ in the read access cost as compared to the default approach. This is also the the minimum possible read cost, as will be shown in Section 2.1. ▸

We begin the exploration of convertible codes by focusing on MDS codes in Chapters 2 and 3, and then we focus on LRCs (Locally Repairable Codes) in Chapter 4. These two classes of codes are two of the most commonly used in distributed storage systems, and thus are good subjects of study. Furthermore, we consider two notions of conversion cost: 1) *access cost* (studied in Chapter 2), which corresponds to the number of nodes that are read or written during conversion; and 2) *conversion bandwidth* (studied in Chapter 3), which corresponds to the amount of network bandwidth used during conversion.

To simplify our analysis of convertible codes, we divide the space of possible parameters into subsets that we call *regimes*. In particular, we consider: 1) the *merge regime*, which consists of conversions that merge multiple codewords into a single one (i.e. $k^F = \lambda^I k^I$ for integer $\lambda^I \geq 2$); 2) the *split regime*, which consists of conversions that split a single codeword into multiple ones (i.e. $k^I = \lambda^F k^F$ for integer $\lambda^F \geq 2$); and 3) the *general regime*, where k^I and k^F are arbitrary.

Throughout our analysis of convertible codes, we assume the values of the parameters (n^I, k^I) and (n^F, k^F) are known and fixed. Similarly, we assume that data undergoes a single conversion. However, in practice the value of (n^F, k^F) might not be known at the time of code construction, since it depends on the future failure rates of storage devices, or multiple conversion might be executed over the same data at different points in time. Throughout the thesis, we also discuss and address these problems: we show how to construct convertible codes which support conversion for multiple possible values of (n^F, k^F) simultaneously, or support multiple consecutive conversions for a sequence of parameters $(n_1, k_1), (n_2, k_2), (n_3, k_3), \ldots$, and so on.

1.2 Related work, background and notation

In this section, we place convertible codes within the larger context of traditional codes and more recent works on codes for distributed storage. Then, we review some basic concepts and notation that will be used throughout this thesis.

1.2.1 Related Work

MDS erasure codes, such as Reed-Solomon codes [20], are widely used in storage systems because they achieve the optimal tradeoff between failure tolerance and storage overhead [21, 22]. However, the use of erasure codes in storage systems raises a host of other aspects to optimize for. Several works in the literature have studied these aspects and proposed codes that optimize them.

One aspect of storage codes that received considerable attention early on is the computational overhead involved in encoding and decoding data. *Array codes* [23–26] are usually designed to use XOR operations exclusively, which are faster to execute, and aim to decrease the complexity of encoding and decoding.

Another aspect of storage codes that has received considerable attention in the recent past is related to the resource overhead associated with repair of failed nodes. Several approaches have been proposed to alleviate this problem. Dimakis et al. [27] proposed a new class of codes called *regenerating codes* that minimize the amount of network bandwidth consumed during repair operations. Under the regenerating codes model [27], each symbol (i.e., node) is represented as an α-dimensional vector over a finite field. During repair of a failed node, downloading elements of the finite field (i.e., "sub-symbols") is allowed as opposed to the whole vector (i.e., one "entire" symbol). This line of research has led to several constructions [28–47], generalizations [48–50], and more efficient repair algorithms for Reed-Solomon codes [39, 51–57]. Several of these constructions [31, 40, 58–60] minimize the amount of IO consumed during repairs in addition to minimizing the network bandwidth consumption. Like regenerating codes, convertible codes optimized for *conversion bandwidth* also aim to minimize IO and network bandwidth, but for code conversion instead of repair. It has been shown

that meeting the lower bound on the network bandwidth required by repair when MDS property and high rate are desired necessitates large sub-packetization [58, 60–62], which negatively affects certain key performance metrics in storage systems [12, 13]. To overcome this issue, several works [63, 64] have proposed code constructions that relax the requirement of meeting lower bounds on IO and bandwidth for repair operations in order to reduce the degree of sub-packetization.

The challenge of code repair has also been addressed by another class of codes, called *locally repairable codes* (LRCs) [65–81]. These codes focus on the locality of codeword symbols during repair, that is, the number of nodes that need to be accessed when repairing a single failure. LRCs improve repair performance, since missing information can be recovered by accessing a small subset of symbols. LRCs and convertible codes optimized for access cost both aim to minimize the number of symbols that need to be accessed, albeit for different operations in storage systems.

Recent literature on storage codes has also considered the problem of redistributing data when additional devices are added to a distributed storage system, which is known as the *scaling problem* [16, 17, 19, 82–90]. The setting considered consists of an n node distributed storage system where the data is encoded using an $[n, k]$ MDS code, where the n symbols of each codeword are spread across evenly on all the n nodes in the system. Then, s new empty nodes are added to the system, and the data (which was encoded under an $[n, k]$ MDS code) needs to be updated to an $[n' = n + s, k' = k + s]$ MDS code. The central goal of this problem is to evenly redistribute each codeword across all n' nodes while reducing the total amount of data transferred across nodes and ensuring the MDS property holds. In some cases, it is additionally required that the ratio of data to parity in each node is the same (e.g. [87]). Some works consider more general scaling scenarios: for example [19] considers the case where $k < k'$ and $n < n'$, and [17] considers arbitrary $n' > k'$. The scaling problem is fundamentally different from the conversion problem that we study in this thesis because of the need to evenly redistribute data across nodes under scaling. Hence, some of the key constraints and limitations of the scaling problem do not apply to code conversion. For example, scaling necessitates modifying every node in the system (incurring a high access cost) and necessitates transfer of data not for

the purpose of conversion (i.e. changing n and k) but for the purpose of rebalancing the amount of data stored by each codeword in a given node. On the other hand, under the code conversion problem, we do not impose any requirements on data balancing. This is because, typically, large-scale distributed storage systems balance data across nodes at a higher level rather than at the level of each codeword [8, 11].

Several works have studied scenarios where encoded data is transformed to conform to a different code. In [91, 92], the authors propose a two-stage encoding process, where in the first stage data is encoded using a $[n, k]$ MDS code, and in the second stage $(n' - n)$ additional parities are generated to form a codeword from a $[n', k]$ MDS code. This process can be seen as a special case of convertible codes, i.e. an $(n, k; n', k)$ convertible code. In [93], the authors propose a distributed storage system which alternates between two specific erasure codes in response to variations in workload. In [94], the authors propose a scheme for changing the parameters of an erasure code in the context of coded matrix multiplication.

In [95], which appeared after the publication of the conference paper that this chapter is based on [96], the authors propose a code construction for improving the efficiency of conversion. This construction performs conversion by acting on initial codewords that are encoded differently, i.e. a different $(k^I \times n^I)$ generator matrix is used for each initial codeword. The focus of Wu et al. [95] is on a practical code construction for a specific parameter regime and they do not investigate theoretical modeling and fundamental limits. All the lower bounds derived in our work continue to hold even if each codeword is encoded differently (i.e. they also apply to the setting considered in [95]). The approach of using multiple different initial codes has the advantage of simplifying the code construction: a final MDS code $\mathcal{C}^F$ is chosen first, and then the encoding of each initial codeword is chosen to fit $\mathcal{C}^F$. However, such an approach has several disadvantages. First, conversion can only happen among specific groups of initial codewords, making the conversion process more rigid as codewords cannot be freely chosen. Second, this approach increases the overhead of codeword management, as the system needs to keep track of the code of each codeword. Third, it only considers one specific known value for the final parameters (n^F, k^F). On the other hand, the framework of convertible codes that we propose

allows one to choose any set of initial codewords for conversion (since they all use the same code), is independent of data placement, and the proposed code constructions support access-optimal conversion for any (n^F, k^F) in a set of possible final parameter values.

1.2.2 Background

In this subsection we introduce some basic definitions and notation related to linear codes. Let $\mathbb{F}_q$ be a finite field of size q. An $[n, k]$ linear code $\mathcal{C}$ over $\mathbb{F}_q$ is a k-dimensional subspace $\mathcal{C} \subseteq \mathbb{F}_q^n$. Here, n is called the length of the code, and k is called the dimension of the code. A *generator matrix* of an $[n, k]$ linear code $\mathcal{C}$ over $\mathbb{F}_q$ is a $k \times n$ matrix $\mathbf{G}$ over $\mathbb{F}_q$ such that the rows of $\mathbf{G}$ form a basis of the subspace $\mathcal{C}$. A $k \times n$ generator matrix $\mathbf{G}$ is said to be *systematic* if it has the form $\mathbf{G} = [\mathbf{I} \mid \mathbf{P}]$, where $\mathbf{I}$ is the $k \times k$ identity matrix and $\mathbf{P}$ is a $k \times (n-k)$ matrix. Even though the generator matrix of a code $\mathcal{C}$ is not unique, we will sometimes associate a code $\mathcal{C}$ to a specific generator matrix $\mathbf{G}$, which will be clear from context. The encoding of a message $\mathbf{m} \in \mathbb{F}_q^k$ under an $[n, k]$ code $\mathcal{C}$ with generator matrix $\mathbf{G}$ is denoted $\mathcal{C}(\mathbf{m}) = \mathbf{m}^T\mathbf{G}$.

Let $[n]$ denote the set $\{1, 2, \ldots, n\}$ for $n \geq 1$, and the empty set for $n \leq 0$. A linear code $\mathcal{C}$ is *maximum distance separable* (MDS) if the minimum distance of the code is the maximum possible:

$$\text{min-dist}(\mathcal{C}) = \min_{c \neq c' \in \mathcal{C}} |\{i \in [n] : c_i \neq c'_i\}| = n - k + 1,$$

where $c_i \in \mathbb{F}_q$ denotes the i-th coordinate of c. Equivalently, a linear code $\mathcal{C}$ is MDS if and only if every $k \times k$ submatrix of its generator matrix $\mathbf{G}$ is non-singular [97].

A matrix M is said to be *superregular* if every square submatrix of M is nonsingular[2]. The following property is a key property that will be used in this thesis.

[2]This definition of superregularity is stronger than the definition introduced in [98] in the context of convolutional codes.

Proposition 1.1 ([97]). *Let $\mathcal{C}$ be an $[n, k]$ code with generator matrix $G = [I|P]$. Then $\mathcal{C}$ is MDS if and only if P is superregular.*

Let $\mathbf{v} \in \mathbb{F}_q^n$ be a vector. We interpret vectors as column vectors by convention. We denote the transpose of a vector (or matrix) as $\mathbf{v}^T$. Given a set of coordinates $\mathcal{I} \subseteq [n]$, we denote the projection of $\mathbf{v}$ to the coordinates in $\mathcal{I}$ as $\mathbf{v}|_{\mathcal{I}} \in \mathbb{F}_q^{|\mathcal{I}|}$. For a set of vectors $\mathcal{V}$ we define $\text{proj}_{\mathcal{I}}(\mathcal{V}) = \{\mathbf{v}|_{\mathcal{I}} \mid v \in \mathcal{V}\}$.

We use the following notation for submatrices: let M be a $n \times m$ matrix, the submatrix of M defined by row indices $\{i_1, \ldots, i_a\} \subseteq [n]$ and column indices $\{j_1, \ldots, j_b\} \subseteq [m]$ is denoted by $M[i_1, \ldots, i_a; j_1, \ldots, j_b]$. For conciseness, we use $*$ to denote all row or column indices, e.g., $M[*; j_1, \ldots, j_b]$ denotes the submatrix composed by columns $\{j_1, \ldots, j_b\}$, and $M[i_1, \ldots, i_a; *]$ denotes the submatrix composed by rows $\{i_1, \ldots, i_a\}$.

1.3 A framework for studying code conversions

In this section, we formally define the new framework for studying *code conversions* and introduce *convertible codes*. While we use the notation of linear codes introduced in Section 1.2.2, the framework introduced in this section can be applied to arbitrary (not necessarily linear) codes. Suppose one wants to convert data that is already encoded using an $[n^I, k^I]$ initial code $\mathcal{C}^I$ into data encoded using an $[n^F, k^F]$ final code $\mathcal{C}^F$ where both codes are over the same field $\mathbb{F}_q$. In the initial and final configurations, the system must store the same information, but encoded differently. In order to capture the changes in the dimension of the code during conversion, we consider $M = \text{lcm}(k^I, k^F)$ number of "message" symbols (i.e., the data to be stored) over a finite field $\mathbb{F}_q$, denoted by $\mathbf{m} \in \mathbb{F}_q^M$. This corresponds to $\lambda^I = M/k^I$ codewords in the initial configuration and $\lambda^F = M/k^F$ codewords in the final configuration. Let $r^I = (n^I - k^I)$ and $r^F = (n^F - k^F)$.

Figure 1.3 shows the conversion process for general initial and final codes. We note that this *need for considering multiple codewords in order to capture the smallest instance of the problem* deviates from existing literature on the code repair (e.g., [27,

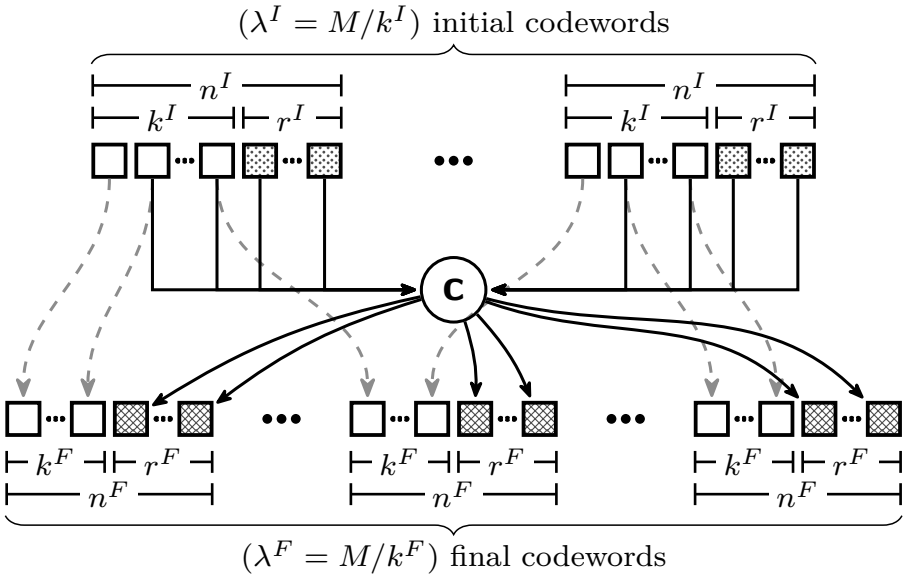


Figure 1.3: Conversion from $[n^I, k^I]$ initial code to $[n^F, k^F]$ final code. Each box denotes a symbol, and they are grouped into codewords. Dotted boxes denote retired symbols, and cross-hatched boxes denote new symbols. The **c** node denotes the location where new symbols are computed from the symbols read during conversion. Solid arrows denote a transfer of symbols (read or write) and dashed arrows denote unchanged symbols.

28, 52, 63]) and code locality (e.g., [65, 70, 80]), where a single codeword is sufficient to capture the problem.

Since there are multiple codewords, we first specify an *initial partition* $\mathcal{P}^I$ and a *final partition* $\mathcal{P}^F$ of the set $[M]$, which map the message symbols of $\mathbf{m}$ to their corresponding initial and final codewords. The initial partition $\mathcal{P}^I = \{P_1^I, \ldots, P_{\lambda^I}^I\}$ is composed of λ^I disjoint subsets of size $|P_i^I| = k^I$ $(i \in [\lambda^I])$, and the final partition $\mathcal{P}^F = \{P_1^F, \ldots, P_{\lambda^F}^F\}$ is composed of λ^F disjoint subsets of size $|P_j^F| = k^F$ $(j \in [\lambda^F])$. In the initial (respectively, final) configuration, the data indexed by each subset $P_i^I \in \mathcal{P}^I$ (respectively, $P_j^F \in \mathcal{P}^F$) is encoded using the code $\mathcal{C}^I$ (respectively, $\mathcal{C}^F$). The codewords $\{\mathcal{C}^I(\mathbf{m}|_{P_i^I}) \mid P_i^I \in \mathcal{P}^I\}$ are referred to as *initial codewords*, and the codewords $\{\mathcal{C}^F(\mathbf{m}|_{P_j^F}) \mid P_j^F \in \mathcal{P}^F\}$ are referred to as *final codewords*. The descriptions of the initial and final partitions and codes, along with the conversion procedure, define a convertible code. We now proceed to define conversions and convertible codes formally.

Definition 1.1 (Code conversion): A *conversion* from an initial code $\mathcal{C}^I$ to a final code $\mathcal{C}^F$ with initial partition $\mathcal{P}^I$ and final partition $\mathcal{P}^F$ is a procedure, denoted by $T_{\mathcal{C}^I \to \mathcal{C}^F}$, that for any $\mathbf{m}$, takes the set of initial codewords $\{\mathcal{C}^I(\mathbf{m}|_{P_i^I}) \mid P_i^I \in \mathcal{P}^I\}$ as input, and outputs the corresponding set of final codewords $\{\mathcal{C}^F(\mathbf{m}|_{P_j^F}) \mid P_j^F \in \mathcal{P}^F\}$. ▸

Definition 1.2 (Convertible code): An $(n^I, k^I; n^F, k^F)$ convertible code over $\mathbb{F}_q$ is defined by: (1) a pair of codes $(\mathcal{C}^I, \mathcal{C}^F)$ where $\mathcal{C}^I$ is an $[n^I, k^I]$ code over $\mathbb{F}_q$ and $\mathcal{C}^F$ is an $[n^F, k^F]$ code over $\mathbb{F}_q$; (2) a pair of partitions $\mathcal{P}^I, \mathcal{P}^F$ of $[M = \text{lcm}(k^I, k^F)]$ such that each subset in $\mathcal{P}^I$ is of size k^I and each subset in $\mathcal{P}^F$ is of size k^F; and (3) a conversion procedure $T_{\mathcal{C}^I \to \mathcal{C}^F}$ that on input $\{\mathcal{C}^I(\mathbf{m}|_{P_i^I}) \mid P_i^I \in \mathcal{P}^I\}$ outputs $\{\mathcal{C}^F(\mathbf{m}|_{P_j^F}) \mid P_j^F \in \mathcal{P}^F\}$, for any $\mathbf{m} \in \mathbb{F}_q^M$. ▸

Typically, additional constraints would be imposed on $\mathcal{C}^I$ and $\mathcal{C}^F$, for example, decodability constraints such as requiring both codes to be MDS.

The cost of conversion is determined by the cost of the conversion procedure $T_{\mathcal{C}^I \to \mathcal{C}^F}$, as a function of the parameters $(n^I, k^I; n^F, k^F)$. Towards minimizing the overhead of the conversion, our general objective is to design codes $(\mathcal{C}^I, \mathcal{C}^F)$, partitions

$(\mathcal{P}^I, \mathcal{P}^F)$ and conversion procedure $T_{\mathcal{C}^I \to \mathcal{C}^F}$ that satisfy Definition 1.2 and minimize the conversion cost for given parameters $(n^I, k^I; n^F, k^F)$, subject to desired decodability constraints on $\mathcal{C}^I$ and $\mathcal{C}^F$.

Depending on the relative importance of various resources in the cluster, one might be interested in optimizing the conversion with respect to various types of costs such as symbol access, computation (CPU), communication (network bandwidth), read/writes (disk IO), etc., or a combination of these costs. The general formulation of code conversions above provides a powerful framework to theoretically reason about convertible codes.

To decide whether or not a conversion procedure is efficient, we need to measure its cost. Two kinds of cost have been consider the access cost of code conversion, which measures the number of symbols that are affected by the conversion.

Definition 1.3 (Access cost)**:** The *read access cost* of a conversion procedure is defined as the total number of symbols read during the procedure. Similarly, the *write access cost* of a conversion procedure is the total number of symbols written during the procedure. The *access cost* of a conversion procedure is the sum of its read and write access costs. The access cost of a convertible code is the access cost of its conversion procedure. ▸

Definition 1.4 (Conversion bandwidth)**:** The *read conversion bandwidth* of a conversion procedure is defined as the total size of the data read from the initial codewords during conversion. Similarly, the *write conversion bandwidth* of a conversion procedure is defined as total size of the data written to the final codewords during conversion. The *conversion bandwidth* of a conversion procedure is the sum of its read and write conversion bandwidths. The conversion bandwidth of a convertible code is the conversion bandwidth of its conversion procedure. ▸

Both of these costs are important in practice, but which one is more important will depend on the specifics of the system and its workloads. Reducing access cost makes code conversion less disruptive, reduces the its tail latency, and allows the unaffected symbols to remain available for normal operation. Reducing conversion

bandwidth reduces the IO of disks and the amount of network traffic. Reducing either access cost or conversion bandwidth will also reduce the amount of computation and communication required in contrast to the default approach. Given the definition of codes given above, it would seem as though both access cost and conversion bandwidth are equivalent, however, we will show in Chapter 3 that by considering a class of codes known as *vector codes*, we can explicitly minimize conversion bandwidth.

In order to understand the necessary access cost of conversion, we classify symbols into three categories: (1) *unchanged symbols*, which refers to symbols in the initial codewords that remain *as is* in the final codewords; (2) *retired symbols*, which refers to the remaining symbols of the initial codewords that are discarded; and (3) *new symbols*, which refers the symbols in the final stripes which are not unchanged (and therefore must be written during conversion). For example, in Figure 1.3, unchanged symbols are unshaded, retired symbols in the initial codewords are dotted, and new symbols in the final codewords are cross-hatched.

Having unchanged symbols has many practical benefits, because when conversion is implemented, such symbols can stay in the same location and only their corresponding metadata needs to be updated. We introduce the following definition to capture codes that maximize the number of such symbols.

Definition 1.5 (Stable convertible code)**:** An $(n^I, k^I; n^F, k^F)$ MDS convertible code is said to be *stable* if it uses the maximum number of unchanged symbols over all $(n^I, k^I; n^F, k^F)$ MDS convertible codes. ▸

In the following chapters, we will see that stable convertible codes play an important role in minimizing access cost and conversion bandwidth.

The convertible codes framework defined in this work is flexible and allows for the initial and final codes to have any parameters and be of any kind. Our goal will be to find codes that minimize the access cost and conversion bandwidth.

Definition 1.6 (Access-optimal)**:** A convertible code is said to be *access-optimal* (over a class of codes) if and only if it attains the minimum access cost possible (in that class of codes). ▸

Table 1.1: Notation used in this thesis.

$\square^I$	Related to initial code	$\square^F$	Related to final code
$n^\diamondsuit$	Code length, number of symbols	$k^\diamondsuit$	Code dimension, number of message symbols
$\lambda^\diamondsuit$	Number of codewords	$\mathcal{C}^\diamondsuit$	Code
$\mathcal{P}^\diamondsuit$	Partition of $[k^\diamondsuit]$	$\mathbf{G}^\diamondsuit$	Generator matrix of $\mathcal{C}^\diamondsuit$
$\mathbf{P}^\diamondsuit$	Parity matrix of $\mathcal{C}^\diamondsuit$	$\mathbf{m}$	Message
$\mathcal{S}_i^\diamondsuit$	Encoding vectors (codeword i)	$\mathcal{S}^\diamondsuit$	All initial/final encoding vectors
$\mathcal{U}_{i,j}$	Unchanged vectors ($= \mathcal{S}_i^I \cap \mathcal{S}_j^F$)	$\mathcal{D}_i$	Read access set (initial codeword i)
$\mathcal{A}_i$	Accessed vectors (initial codeword i)	$\mathcal{N}$	New encoding vectors ($= \mathcal{S}^F \setminus \mathcal{S}^I$)

Definition 1.7 (Bandwidth-optimal)**:** A convertible code is said to be *bandwidth-optimal* (over a class of codes) if and only if it attains the minimum conversion bandwidth possible (in that class of codes). ▸

In practice, the final parameters (n^F, k^F) might not be known at the time of code construction because they might depend on future failure rates. To address this, we also consider designing codes which have the ability to be converted to multiple final codes of different length and dimension with optimal access cost. This way, instead of having to decide (n^F, k^F) in advance, the user can specify a subset $S \subseteq (\mathbb{N} \times \mathbb{N})$ of possible values for the pair (n^F, k^F) and construct an initial code with the ability to be converted to an $[n^F, k^F]$ final code for *any* $(n^F, k^F) \in S$. At the time of conversion, the user simply chooses the desired pair from S and converts. We introduce the following definition to help describe such codes.

Definition 1.8 (Optimally convertible)**:** A $[n^I, k^I]$ MDS code $\mathcal{C}^I$ is said to be access/bandwidth-optimally convertible if and only if it is the initial code of an access/bandwidth-optimal $(n^I, k^I; n^F, k^F)$ convertible code. ▸

1.3.1 Notation for linear convertible codes

In this chapter, we focus exclusively on convertible codes where $\mathcal{C}^I$ and $\mathcal{C}^F$ are linear. To this end, we introduce some notation for describing and analyzing this class of codes. Table 1.1 summarizes the most important notation used for easy reference.

Let $\Diamond \in \{I, F\}$. The generator matrix of $\mathcal{C}^\Diamond$ is a $(k^\Diamond \times n^\Diamond)$ matrix $\mathbf{G}^\Diamond = [\mathbf{g}_1^\Diamond \cdots \mathbf{g}_{n^\Diamond}^\Diamond]$, where $\mathbf{g}_j^\Diamond \in \mathbb{F}_q^{k^\Diamond}$ $(j \in [n^\Diamond])$ denotes the j-th *encoding vector* of $\mathcal{C}^\Diamond$. Consequently, with a given partition $\mathcal{P}^\Diamond = \{P_i^\Diamond\}_{i=1}^{\lambda^\Diamond}$, the j-th symbol of the i-th codeword corresponds to $(\mathbf{m}|_{P_i^\Diamond})^T \mathbf{g}_j^\Diamond$.

In order to analyse linear convertible codes, we also view each code symbol in relation to the whole message $\mathbf{m}$. Accordingly, we view the j-th symbol of the i-th initial codeword as $\mathbf{m}^T \tilde{\mathbf{g}}_{i,j}^\Diamond$, where the encoding vector $\tilde{\mathbf{g}}_{i,j}^\Diamond \in \mathbb{F}_q^M$ is defined to be equal to $\mathbf{g}_j^\Diamond$ for coordinates in $P_i^\Diamond$, i.e. $\tilde{\mathbf{g}}_{i,j}^\Diamond|_{P_i^\Diamond} = \mathbf{g}_j^\Diamond$, and equal to 0 everywhere outside of $P_i^\Diamond$. Note that $\mathbf{m}^T \tilde{\mathbf{g}}_{i,j}^\Diamond = (\mathbf{m}|_{P_i^\Diamond})^T \mathbf{g}_j^\Diamond$ for all $i \in [\lambda^\Diamond]$ and $j \in [n^\Diamond]$. In general, we will refer to a code symbol and its corresponding encoding vector interchangeably.

Let $\mathcal{S}_i^\Diamond = \{\tilde{\mathbf{g}}_{i,j}^\Diamond \mid j \in [n^\Diamond]\}$ denote the encoding vectors of codeword $i \in [\lambda^\Diamond]$, and let $\mathcal{S}^\Diamond = \bigcup_{i \in [\lambda^\Diamond]} \mathcal{S}_i^\Diamond$. Define $\mathcal{U}_{i,j} = (\mathcal{S}_i^I \cap \mathcal{S}_j^F)$ denoting the unchanged symbols that form part of initial codeword i and final codeword j. If $\lambda^I = 1$ or $\lambda^F = 1$, then we omit the corresponding index from $\mathcal{U}_{i,j}$ for simplicity. Let $\mathcal{U} = (\mathcal{S}^I \cap \mathcal{S}^F)$ denote all unchanged vectors. We define the *read access set* of a convertible code as a set of tuples $\mathcal{D} \in [\lambda^I] \times [n^I]$, where $(i, j) \in \mathcal{D}$ corresponds to the j-th symbol of initial codeword i. Furthermore, we use $\mathcal{D}_i = \{j \mid (i, j) \in \mathcal{D}\}$, $\forall i \in [\lambda^I]$ to denote the symbols read from initial codeword i. Note that the read access cost is given by $|\mathcal{D}|$. Let $\mathcal{A}_i = \{\tilde{\mathbf{g}}_{i,j}^I \mid j \in \mathcal{D}_i\}$ denote the encoding vectors of the symbols from initial codeword $i \in [\lambda^I]$ that are part of the read access set $\mathcal{D}$, and define $\mathcal{A} = \{\tilde{\mathbf{g}}_{i,j}^I \mid (i, j) \in \mathcal{D}\}$ as the set of all encoding vectors of the symbols in the read access set. Finally, let $\mathcal{N} = (\mathcal{S}^F \setminus \mathcal{S}^I)$ denote the new vectors. Notice that it must hold that $\mathcal{N} \subseteq \text{span}(\mathcal{A})$, since the new vectors are obtained as linear combinations of the encoding vectors of the symbols in the read access set.

1.3.2 The case of $k^I = k^F$

Before diving into the study of convertible codes, we briefly study the exceptional conversion case where $k^I = k^F$. We typically do not consider conversion with parameters $k^I = k^F$ as part of the merge or split regime because it does not have the same behavior. However, we analyze this case here for completeness. Observe

that in the case where $n^I \geq n^F$, conversion for any MDS code can be carried out with zero access cost and conversion bandwidth by simply retiring any $(n^I - n^F)$ symbols. In the complementary case where $n^I < n^F$, it is necessary to access at least k^I symbols and write at least $(n^F - n^I)$ symbols (i.e. it is not possible to beat the default approach in terms of access cost). This is apparent from the fact that in an $[n, k]$ MDS code, any subset of $k - 1$ symbols gives no information about any one of the remaining symbols. The minimization of conversion bandwidth in the where $n^I < n^F$ is non-trivial: it can be considered as an special case of a *regenerating code* [27], and it has also been studied on previous work [92].

Chapter 2

Access-cost of convertible codes: fundamental limits and optimal constructions

This chapter is based on work from [7], done in collaboration with K. V. Rashmi; and [99], done in collaboration with V. S. Chaitanya Mukka and K. V. Rashmi.

There are several ways in which one might measure the cost of conversion. In this chapter, we will focus on *access cost*, which is measured in terms of the total number of symbols that need to be accessed during conversion. In particular, by the end of this chapter we will have presented:

1. lower bounds on the access cost of conversion for linear MDS codes *for all valid parameters*, that is, all $n^I, k^I, n^F, k^F \in \mathbb{N}$ such that $n^I > k^I$ and $n^F > k^F$.

2. *explicit constructions* of linear MDS convertible codes that achieve these lower bounds, and are thus optimal in terms of access-cost.

To achieve this, we divide the problem space (the set of possible parameters) into three regimes:

- the merge regime, where multiple codewords are combined into a single one (i.e. $k^F = \lambda^I k^I$ for an integer $\lambda^I \geq 2$);

Table 2.1: Optimal access cost for different regimes, assuming $r^F \leq \min\{k^I, k^F\}$. When $r^F > k^I$ or $r^F > k^F$, the optimal access cost is the same as the default approach.

Regime	Access cost ($r^I < r^F$)	Access cost ($r^I \geq r^F$)	Default approach
Merge regime	$\lambda^I k^I$	$\lambda^I r^F$	$\lambda^I k^I$
Split regime	$\lambda^F k^F$	$(\lambda^F - 1)k^F + r^F$	$\lambda^F k^F$
General regime	$\text{lcm}(k^I, k^F)$	$\lambda^I r^F + (\lambda^I \bmod \lambda^F)$ $(k^I - \max\{k^F \bmod k^I, r^F\})$	$\text{lcm}(k^I, k^F)$

- the split regime, where a single codeword is split into multiple ones (i.e. $k^I = \lambda^F k^F$ for an integer $\lambda^F \geq 2$);
- the general regime, where k^I and k^F take arbitrary values.

We prove access-cost lower bounds and constructions for each of these cases separately. One surprising aspect is that the lower bounds and constructions for the merge regime and split regime are directly used in proving a lower bound and designing the construction for the general regime. Interestingly, one of the degrees-of-freedom in the design of convertible codes (called "partitions", described in Chapter 1), which is inconsequential in the split and merge regimes, turns out to be crucial in the general regime. The proposed construction for access-optimal convertible codes for the general regime builds on the constructions for split and merge regimes, while separately optimizing along this additional degree-of-freedom. We summarize the results of this chapter in Table 2.1.

2.1 Lower bounds on the access cost of convertible codes in the merge regime

In this section, we focus on studying the merge regime. Recall, from Section 1.3, that the merge regime corresponds to conversion where multiple codewords are combined

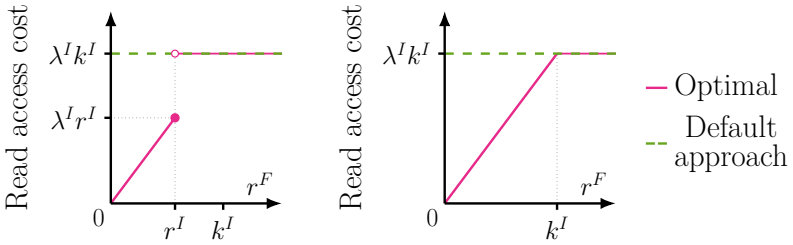


Figure 2.1: Comparison of the read access cost of the optimal conversion of a $(n^I, k^I; n^F, k^F = \lambda^I k^I)$ convertible code and the default approach for a variable value of r^F (x-axis) when $r^I < k^I$ (left side) and $r^I \geq k^I$ (right side). When $r^F < \min\{k^I, r^I\}$, optimal conversion achieves lower cost than the default approach, and when $r^F > \min\{k^I, r^I\}$, the default approach is (trivially) optimal. The optimal write access cost in the merge regime is always r^F.

Table 2.2: Access cost savings for different example parameters.

$[n^I, k^I] \Rightarrow [n^F, k^F]$	Optimal read access cost	Default read access cost	Write access cost (either approach)	Cost reduction
$[14, 10] \Rightarrow [22, 20]$	4	20	2	72.7%
$[9, 6] \Rightarrow [14, 12]$	4	12	2	57.1%
$[5, 3] \Rightarrow [11, 9]$	6	9	2	27.8%
$[9, 5] \Rightarrow [14, 10]$	8	10	4	14.3%
$[6, 4] \Rightarrow [11, 8]$	8	8	3	0.0%

into a single codeword (i.e. $k^F = \lambda^I k^I$ for an integer $\lambda^I \geq 2$). This implies that $M = k^F$ and $\lambda^F = 1$.

In this section, we present lower bounds on the access cost of linear MDS convertible codes in the merge regime. Our main result is summarized by the following theorem, which will be proved at the end of this section.

Theorem 2.1. *For all linear MDS $(n^I, k^I; n^F, k^F = \lambda^I k^I)$ convertible codes, the read access cost of conversion is at least $\lambda^I \min\{k^I, r^F\}$ and the write access cost is at least r^F. Furthermore, if $r^I < r^F$, the read access cost of conversion is at least $\lambda^I k^I$.* ⊡

As we will show in Section 2.2, this lower bound is achievable and it therefore corresponds to the optimal access cost in the merge regime. Figure 2.1 shows a plot comparing the optimal access cost against the access cost of the default approach for different parameter values, and Table 2.2 shows these costs for some concrete conversion examples.

We break down the proof of this result into four steps:

1. We show that in the merge regime, all possible pairs of partitions $\mathcal{P}^I$ and $\mathcal{P}^F$ partitions are equivalent up to relabeling, and hence do not need to be specified (Lemma 2.2).

2. An upper bound on the maximum number of unchanged symbols is proved. As described in Definition 1.5, convertible codes that meet this bound are called stable (Lemma 2.3).

3. Lower bounds on the access cost of linear MDS convertible codes are proved under the added restriction that the codes are stable (Lemmas 2.4 and 2.5 and Theorem 2.6).

4. The stability restriction is removed, by showing that non-stable linear MDS convertible codes necessarily incur higher access cost, and hence it suffices to consider only stable MDS convertible codes (Lemma 2.8 and Theorem 2.1).

In general, partitions need to be specified since they indicate how message symbols from the initial codewords are mapped into the final codewords. However in the merge

regime, the choice of the partitions are equivalent, and hence are inconsequential as shown below.

Lemma 2.2. *For every $(n^I, k^I; n^F, k^F = \lambda^I k^I)$ convertible code, all possible pairs of initial and final partitions $(\mathcal{P}^I, \mathcal{P}^F)$ are equivalent up to relabeling of symbols.*

Proof. We have that $k^I \mid k^F$. Thus $\lambda^F = (M/k^F) = 1$ and $\mathcal{P}^F = \{[M]\}$ always holds. Because of this, all data will be mapped to the same final codeword, regardless of the initial partition. Therefore, for any two partitions $\mathcal{P}^I$ and $\mathcal{P}^{I'}$, there exists some permutation σ of $[\lambda^I k^I]$ such that $\mathcal{P}^{I'} = \{\sigma(P) \mid P \in \mathcal{P}^I\}$, i.e., different partitions differ only on the way symbols are labeled. □

Since one of the terms in access cost is the number of new symbols, a natural way to reduce access cost is to maximize the number of unchanged symbols. However, there is a limit on the number of symbols that can remain unchanged which is characterized below.

Lemma 2.3. *In an MDS $(n^I, k^I; n^F, k^F = \lambda^I k^I)$ convertible code, there can be at most k^I unchanged symbols from each initial codeword.*

Proof. By the MDS property of $\mathcal{C}^I$ every subset of k^I+1 symbols is linearly dependent. Hence, there can be at most k^I unchanged symbols from each initial codeword for $\mathcal{C}^F$ to be MDS. In other words, $|\mathcal{U}_i| \leq k^I$ for all $i \in [\lambda^I]$. □

This implies that there are at most $\lambda^I k^I$ unchanged symbols and at least r^F new symbols in total. Thus, the number of symbols that need to be written in a stable code is at least r^F.

Now, we focus on bounding the total number of symbols read, that is, the size of the read access sets. The general strategy we use to obtain bounds on the size of read access sets is to consider a specially chosen set of k^F encoding vectors from the final codeword, which by the MDS property of the final code is linearly independent. We then use the fact that final codewords are the result of conversion to identify the encoding vectors in each initial codeword that span the selected final encoding vectors. The MDS property of the initial code and the fact that different initial codewords

contain different information will allow us to derive a lower bound on the number of read symbols in each initial codeword.

Intuitively, having more new symbols means that more symbols have to be read in order to construct them, resulting in higher access cost. With this intuition in mind, we first focus on stable convertible codes, which minimize the number of new symbols (Definition 1.5). We first prove lower bounds on the access cost of stable linear MDS convertible codes, and then show that the minimum access cost of conversion in MDS codes without this stability property can only be higher. The first lower bound on the size of each $\mathcal{D}_i$ ($i \in [\lambda^I]$) is given by the interaction between new symbols and the MDS property.

Lemma 2.4. *For every linear stable MDS $(n^I, k^I; n^F, k^F = \lambda^I k^I)$ convertible code, the read access set $\mathcal{D}_i$ from each initial codeword $i \in [\lambda^I]$ satisfies $|\mathcal{D}_i| \geq \min\{k^I, r^F\}$.*

Proof. For convenience, readers can recall the notation from Table 1.1. By the MDS property, every subset $\mathcal{V} \subseteq \mathcal{S}^F$ of size at most $k^F = \lambda^I k^I$ is linearly independent. For any initial codeword $i \in [\lambda^I]$, take the set of all unchanged encoding vectors from other codewords $\cup_{\ell \neq i} \mathcal{U}_\ell$, and additionally pick any subset of new encoding vectors $\mathcal{W} \subseteq \mathcal{N}$ of size $|\mathcal{W}| = \min\{k^I, r^F\}$. The following holds for set $\mathcal{V} = (\cup_{\ell \neq i} \mathcal{U}_\ell \cup \mathcal{W})$:

$$\mathcal{V} \subseteq \mathcal{S}^F \text{ and } |\mathcal{V}| = (\lambda^I - 1)k^I + \min\{k^I, r^F\} \leq k^F.$$

Therefore, all the encoding vectors in $\mathcal{V}$ are linearly independent.

Notice that the encoding vectors in $(\mathcal{V} \setminus \mathcal{W})$ contain no information about initial codeword i and complete information about every other initial codeword $\ell \neq i$. Therefore, the information about initial codeword i in each encoding vector in $\mathcal{W}$ has to be linearly independent since, otherwise, $\mathcal{V}$ could not be linearly independent. Formally, it must be the case that $\mathcal{W}_i = \text{proj}_{P_i^I}(\mathcal{W})$ has rank equal to $\min\{k^I, r^F\}$ (recall that P_i^I is the set of symbols corresponding to initial codeword i). However, by definition, the subset $\mathcal{W}_i$ must be contained in the span of $\mathcal{A}_i$. Therefore, the rank of $\mathcal{A}_i$ is at least that of $\mathcal{W}_i$, which implies that $|\mathcal{D}_i| \geq \min\{k^I, r^F\}$. □

We next show that when the number of new symbols r^F is greater than r^I in

a MDS stable convertible code in the merge regime, then the default approach is optimal in terms of access cost.

Lemma 2.5. *For every linear stable MDS $(n^I, k^I; n^F, k^F = \lambda^I k^I)$ convertible code, if $r^I < r^F$ then the read access set $\mathcal{D}_i$ from each initial codeword $i \in [\lambda^I]$ satisfies $|\mathcal{D}_i| \geq k^I$.*

Proof. When $r^F \geq k^I$, this lemma is equivalent to Lemma 2.4, so assume $r^I < r^F < k^I$. From the proof of Lemma 2.4, for every initial codeword $i \in [\lambda^I]$ it holds that $|\mathcal{D}_i| \geq r^F$. Since $r^F > r^I$, this implies that $\mathcal{D}_i$ must contain at least one index of an unchanged encoding vector.

Choose a subset of at most $k^F = \lambda^I k^I$ encoding vectors from $\mathcal{S}^F$, which must be linearly independent by the MDS property. In this subset, include all the unchanged encoding vectors from the other initial codewords, $\cup_{\ell \neq i} \mathcal{U}_\ell$. Then, choose all the unchanged encoding vectors from initial codeword i that are accessed during conversion, $\mathcal{W}_1 = (\mathcal{A}_i \cap \mathcal{U}_i)$. For the remaining vectors (if any), choose an arbitrary subset of new encoding vectors, $\mathcal{W}_2 \subseteq \mathcal{N}$, such that:

$$|\mathcal{W}_2| = \min\{k^I - |\mathcal{W}_1|, r^F\}. \tag{2.1}$$

It is easy to check that the subset $\mathcal{V} = (\cup_{\ell \neq i} \mathcal{U}_\ell \cup \mathcal{W}_1 \cup \mathcal{W}_2)$ is of size at most $k^F = \lambda^I k^I$, and therefore it is linearly independent. This choice of $\mathcal{V}$ follows from the idea that the information contributed by $\mathcal{W}_1$ to the new encoding vectors is already present in the unchanged encoding vectors, which will be at odds with the linear independence of $\mathcal{V}$.

Since the elements of $\mathcal{W}_1$ and $\mathcal{W}_2$ are the only encoding vectors in $\mathcal{V}$ that contain information from initial codeword i, it must be the case that $\widetilde{\mathcal{W}} = (\text{proj}_{P_i^I}(\mathcal{W}_1) \cup \text{proj}_{P_i^I}(\mathcal{W}_2))$ has rank $(|\mathcal{W}_1| + |\mathcal{W}_2|)$. Moreover, $\widetilde{\mathcal{W}}$ is contained in the span of $\mathcal{A}_i$ by definition, so it holds that:

$$|\mathcal{D}_i| \geq |\mathcal{W}_1| + |\mathcal{W}_2|. \tag{2.2}$$

From Equation (2.1), there are two cases:

Case 1: $(k^I - |\mathcal{W}_1|) \leq r^F$. Then $|\mathcal{W}_2| = (k^I - |\mathcal{W}_1|)$ and by Equation (2.2) it holds that:

$$|\mathcal{D}_i| \geq (|\mathcal{W}_1| + |\mathcal{W}_2|) = k^I. \tag{2.3}$$

Case 2: $(k^I - |\mathcal{W}_1|) > r^F$. Then $|\mathcal{W}_2| = r^F$ and by Equation (2.2) it holds that:

$$|\mathcal{D}_i| \geq |\mathcal{W}_1| + r^F. \tag{2.4}$$

Notice that there are only r^I retired (i.e. not unchanged) encoding vectors in codeword i. Since every accessed encoding vector is either in $\mathcal{W}_1$ or is a retired encoding vector, it holds that:

$$|\mathcal{D}_i| \leq |\mathcal{W}_1| + r^I. \tag{2.5}$$

By combining Equation (2.4) and Equation (2.5), we arrive at the contradiction $r^F \leq r^I$, which occurs because there are not enough retired symbols in the initial codeword i to ensure that the final code has the MDS property. Therefore, case 1 must always hold, and $|\mathcal{D}_i| \geq k^I$. □

Combining the above results leads to the following theorem on the lower bound of read access set size of linear stable MDS convertible codes.

Theorem 2.6. *For all stable linear MDS $(n^I, k^I; n^F, k^F = \lambda^I k^I)$ convertible codes with read access set $\mathcal{D}$, it holds that $|\mathcal{D}| \geq \lambda^I \min\{k^I, r^F\}$. Furthermore, if $r^I < r^F$, then $|\mathcal{D}| \geq k^F$.*

Proof. Follows directly from Lemma 2.4 and Lemma 2.5. □

We next show that this lower bound generally applies even for non-stable convertible codes by proving that increasing the number of new symbols from the minimum possible does not decrease the lower bound on the size of the read access set $\mathcal{D}$.

Lemma 2.7. *The lower bounds on the size of the read access set from Theorem 2.6 hold for **all** linear MDS $(n^I, k^I; n^F, k^F = \lambda^I k^I)$ convertible codes.*

Proof. We show that, even for non-stable convertible codes, that is, when there are more than r^F new symbols, the bounds on the read access set $\mathcal{D}$ from Theorem 2.6 still hold.

Case 1: $r^I \geq r^F$. Let $i \in [\lambda^I]$ be an arbitrary initial codeword. We lower bound the size of $\mathcal{D}_i$ by invoking the MDS property on a subset $\mathcal{V} \subseteq \mathcal{S}^F$ of size $|\mathcal{V}| = \lambda^I k^I$ that minimizes the size of the intersection $|\mathcal{V} \cap \mathcal{U}_i|$. There are exactly r^F encoding vectors in $(\mathcal{S}^F \setminus \mathcal{V})$, so the minimum size of the intersection $|\mathcal{V} \cap \mathcal{U}_i|$ is $\max\{|\mathcal{U}_i| - r^F, 0\}$. Clearly, the subset $\text{proj}_{P_i^I}(\mathcal{V})$ has rank k^I due to the MDS property. Therefore, it holds that $|\mathcal{D}_i| + \max\{|\mathcal{U}_i| - r^F, 0\} \geq k^I$. By reordering, the following is obtained:

$$|\mathcal{D}_i| \geq k^I - \max\{|\mathcal{U}_i| - r^F, 0\} \geq \min\{r^F, k^I\},$$

which means that the bound on $\mathcal{D}_i$ established in Lemma 2.4 continues to hold for non-stable codes.

Case 2: $r^I < r^F$. Let $i \in [\lambda^I]$ be an arbitrary initial codeword, let $\mathcal{W}_1 = (\mathcal{A}_i \cap \mathcal{U}_i)$ be the unchanged encoding vectors that are accessed during conversion, and let $\mathcal{W}_2 = (\mathcal{U}_i \setminus \mathcal{W}_1)$ be the unchanged encoding vectors that are *not* accessed during conversion. Consider the subset $\mathcal{V} \subseteq \mathcal{S}^F$ of $|\mathcal{V}| = k^F$ encoding vectors from the final codeword such that $\mathcal{V} \supseteq \mathcal{W}_1$ and the size of the intersection $\mathcal{W}_3 = (\mathcal{V} \cap \mathcal{W}_2)$ is minimized. Since $\mathcal{V}$ may exclude at most r^F encoding vectors from the final codeword, it holds that:

$$|\mathcal{W}_3| = \max\{0, |\mathcal{W}_2| - r^F\}. \tag{2.6}$$

By the MDS property, $\mathcal{V}$ is a linearly independent set of encoding vectors of size k^F, and thus, must contain all the information to recover the contents of every initial codeword, and in particular, initial codeword i. Since all the information in $\mathcal{V}$ about codeword i is in either $\mathcal{W}_3$ or the accessed encoding vectors, it must hold that:

$$|\mathcal{D}_i| + |\mathcal{W}_3| \geq k^I. \tag{2.7}$$

From Equation (2.6), there are two cases:

Subcase 2.1: $(|\mathcal{W}_2| - r^F) \leq 0$. Then $|\mathcal{W}_3| = 0$, and by Equation (2.7) it holds

that $|\mathcal{D}_i| \geq k^I$, which matches the bound of Lemma 2.5.

Subcase 2.2: $(|\mathcal{W}_2| - r^F) > 0$. Then $|\mathcal{W}_3| = (|\mathcal{W}_2| - r^F)$, and by Equation (2.7) it holds that:

$$|\mathcal{D}_i| + |\mathcal{W}_2| - r^F \geq k^I. \tag{2.8}$$

The initial codeword i has $(k^I + r^I)$ symbols. By the principle of inclusion-exclusion we have that:

$$|\mathcal{D}_i| + |\mathcal{U}_i| - |\mathcal{W}_1| \leq k^I + r^I. \tag{2.9}$$

By using Equation (2.8), Equation (2.9) and the fact that $|\mathcal{W}_2| = (|\mathcal{U}_i| - |\mathcal{W}_1|)$, we conclude that $r^I \geq r^F$, which is a contradiction and means that subcase 2.1 always holds in this case. □

The above result, along with the fact that the lower bound in Theorem 2.6 is achievable (as will be shown in Section 2.2), implies that all access-optimal linear MDS convertible codes in the merge regime are stable.

Lemma 2.8. *All access-optimal linear MDS* $(n^I, k^I; n^F, k^F = \lambda^I k^I)$ *convertible codes are stable.*

Proof. Lemma 2.7 shows that the lower bound on the read access set $\mathcal{D}$ for stable linear MDS convertible codes continues to hold in the non-stable case. Furthermore, this bound is achievable by stable linear MDS convertible codes in the merge regime (as will be shown in Section 2.2). The number of new blocks written during conversion under stable MDS convertible codes is r^F. On the other hand, the number of new symbols under a non-stable convertible code is strictly greater than r^F. Thus, the overall access cost of a non-stable MDS $(n^I, k^I; n^F, k^F = \lambda^I k^I)$ convertible code is strictly greater than the access cost of an access-optimal $(n^I, k^I; n^F, k^F = \lambda^I k^I)$ convertible code. □

Thus, for MDS convertible codes in the merge regime, it suffices to focus only on stable codes. Combining all the results above, leads to the main theorem presented at the beginning of this section.

Proof of Theorem 2.1. Follows from Theorem 2.6 and Lemmas 2.7 and 2.8, and the fact that at least r^F new symbols must be written. □

Next, in Section 2.2 we show that the lower bound of Theorem 2.1 is achievable for all parameters. Thus, Theorem 2.1 implies that it is possible to perform conversion of MDS convertible codes in the merge regime with significantly less access cost than the default approach if and only if $r^F \leq r^I$ and $r^F < k^I$.

2.2 Achievability: Explicit access-optimal convertible codes in the merge regime

In this section, we present an explicit construction of access-optimal MDS convertible codes for all parameters in the merge regime. In other words, we present a construction that matches the access cost lower bound presented in Section 2.1. In Section 2.2.1, we present the construction of the generator matrices for the initial and final code. Then, in Section 2.2.2, we describe sufficient conditions for optimality and show that this construction satisfies these conditions and thus yields access-optimal convertible codes. Our constructions in this and the following section work over any finite field of sufficient size (which we explicitly specify), but for the sake of illustration we use prime fields in our examples.

2.2.1 Explicit construction of generator matrices

Recall that, in the merge regime, $k^F = \lambda^I k^I$, for an integer $\lambda^I \geq 2$, while $n^I > k^I$ and $n^F > k^F$ are arbitrary. Also, recall that $r^I = (n^I - k^I)$ and $r^F = (n^F - k^F)$. Notice that when $r^I < r^F$ or $k^I \leq r^F$, constructing an access-optimal convertible code is trivial, since the default approach to conversion is optimal. Thus, assume $r^F \leq \min\{r^I, k^I\}$.

Let $\mathbb{F}_q$ be a finite field of size $q = p^D$, where p is any prime (in particular, we can have $p = 2$, i.e. a binary field) and the degree D is determined by a function of the convertible code parameters (discussed later in this subsection). The degree D

required by this construction is $\mathcal{O}((\max\{n^I, n^F\})^3)$, that is, the field size requirement is exponential in the length of the code. Let θ be a primitive element of $\mathbb{F}_q$. Let $\mathbf{G}^I = [\mathbf{I}|\mathbf{P}^I]$ and $\mathbf{G}^F = [\mathbf{I}|\mathbf{P}^F]$ be systematic generator matrices of $\mathcal{C}^I$ and $\mathcal{C}^F$ over $\mathbb{F}_q$, where $\mathbf{P}^I$ is a $k^I \times r^I$ matrix and $\mathbf{P}^F$ is a $k^F \times r^F$ matrix.

Define entry (i, j) of $\mathbf{P}^I \in \mathbb{F}_q^{k^I \times r^I}$ as $\theta^{(i-1)(j-1)}$, where (i, j) ranges over $[k^I] \times [r^I]$. Entry (i, j) of $\mathbf{P}^F \in \mathbb{F}_q^{k^F \times r^F}$ is defined identically as $\theta^{(i-1)(j-1)}$, where (i, j) ranges over $[k^F] \times [r^F]$. That is, $\mathbf{P}^I$ and $\mathbf{P}^F$ are as follows:

$$\mathbf{P}^I = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \theta & \theta^2 & \cdots & \theta^{(r^I-1)} \\ 1 & \theta^2 & \theta^4 & \cdots & \theta^{2(r^I-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \theta^{(k^I-1)} & \theta^{2(k^I-1)} & \cdots & \theta^{(k^I-1)(r^I-1)} \end{bmatrix},$$

$$\mathbf{P}^F = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \theta & \theta^2 & \cdots & \theta^{(r^F-1)} \\ 1 & \theta^2 & \theta^4 & \cdots & \theta^{2(r^F-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \theta^{(k^F-1)} & \theta^{2(k^F-1)} & \cdots & \theta^{(k^F-1)(r^F-1)} \end{bmatrix}.$$

Notice that this construction is stable, because it is access-optimal (recall from Lemma 2.8). The unchanged symbols of the initial code are exactly the systematic symbols.

2.2.2 Proof of optimality

Recall from Proposition 1.1, that if the constructed code is to be MDS, then both $\mathbf{P}^I$ and $\mathbf{P}^F$ need to be superregular (every square submatrix of them is invertible). In addtion, to be access-optimal during conversion in the non-trivial case, the new symbols (corresponding to the columns of $\mathbf{P}^F$) have to be such that they can be generated by accessing r^F symbols from the initial codewords (corresponding to columns of $\mathbf{G}^I$).

During conversion, the encoding vectors of symbols from the initial codewords are represented as $\lambda^I k^I$-dimensional vectors, where each initial codeword occupies a disjoint subset of k^I coordinates. To capture this property, we introduce the following definition.

Definition 2.1 (**t-column block-constructible**)**:** We will say that an $n \times m_1$ matrix M_1 is *t-column constructible* from an $n \times m_2$ matrix M_2 if and only if there exists a subset $S \subseteq \text{cols}(M_2)$ of size t, such that the m_1 columns of M_1 are in the span of S. We say that a $\lambda n \times m_1$ matrix M_1 is *t-column block-constructible* from an $n \times m_2$ matrix M_2 if and only if for every $i \in [\lambda^I]$, the submatrix $M_1[(i-1)n+1, \ldots, in; *]$ is t-column constructible from M_2. ▸

Theorem 2.9. *A systematic $(n^I, k^I; n^F, k^F = \lambda^I k^I)$ convertible code with $k^I \times r^I$ initial parity generator matrix $\mathbf{P}^I$ and $k^F \times r^F$ final parity generator matrix $\mathbf{P}^F$ is MDS and access-optimal, if the following two conditions hold: (1) if $r^I \geq r^F$ then $\mathbf{P}^F$ is r^F-column block-constructible from $\mathbf{P}^I$, and (2) $\mathbf{P}^I, \mathbf{P}^F$ are superregular.*

Proof. Follows from Proposition 1.1 and the fact that $\mathbf{P}^F$ must be generated by accessing just r^F symbols from each initial codeword (Lemma 2.4). □

Thus, we can reduce the problem of proving the optimality of a systematic MDS convertible code in the merge regime to that of showing that matrices $\mathbf{P}^I$ and $\mathbf{P}^F$ satisfy the two properties mentioned in Theorem 2.9.

We first show that the construction specified in Section 2.2.1 satisfies condition (1) of Theorem 2.9.

Lemma 2.10. *Let $\mathbf{P}^I, \mathbf{P}^F$ be as defined in Section 2.2.1. Then $\mathbf{P}^F$ is r^F-column block-constructible from $\mathbf{P}^I$.*

Proof. Consider the first r^F columns of $\mathbf{P}^I$, which we denote as $\mathbf{P}^I_{r^F} = \mathbf{P}^I[*; 1, \ldots, r^F]$.

Notice that $\mathbf{P}^F$ can be written as the following block matrix:

$$\mathbf{P}^F = \begin{bmatrix} \mathbf{P}^I_{r^F} \\ \mathbf{P}^I_{r^F} \operatorname{diag}(1, \theta^{k^I}, \theta^{2k^I}, \ldots, \theta^{(r^F-1)k^I}) \\ \mathbf{P}^I_{r^F} \operatorname{diag}(1, \theta^{2k^I}, \theta^{2\cdot 2k^I}, \ldots, \theta^{(r^F-1)2k^I}) \\ \vdots \\ \mathbf{P}^I_{r^F} \operatorname{diag}(1, \theta^{(\lambda^I-1)k^I}, \ldots, \theta^{(r^F-1)(\lambda^I-1)k^I}) \end{bmatrix},$$

where $\operatorname{diag}(a_1, a_2, \ldots, a_n)$ is the $n \times n$ diagonal matrix with $(a_1, \ldots, a_n)$ as the diagonal elements. From this representation, it is clear that $\mathbf{P}^F$ can be constructed from the the first r^F columns of $\mathbf{P}^I$. □

It only remains to show that the construction in Section 2.2.1 satisfies condition (2) of Theorem 2.9, that is, that $\mathbf{P}^I$ and $\mathbf{P}^F$ are superregular.

Lemma 2.11. *Let $\mathbf{P}^I, \mathbf{P}^F$ be as defined in Section 2.2.1. Then $\mathbf{P}^I$ and $\mathbf{P}^F$ are superregular, for sufficiently large field size.*

Proof. Let $\mathbf{R}$ be a $t \times t$ submatrix of $\mathbf{P}^I$ or $\mathbf{P}^F$, determined by the row indices $i_1 < i_2 < \cdots < i_t$ and the column indices $j_1 < j_2 < \cdots < j_t$, and denote entry (i, j) of $\mathbf{R}$ as $\mathbf{R}[i, j]$. The determinant of $\mathbf{R}$ is defined by the Leibniz formula:

$$\begin{aligned} \det(\mathbf{R}) &= \sum_{\sigma \in \mathrm{Perm}(t)} \operatorname{sgn}(\sigma) \prod_{l=1}^{t} \mathbf{R}[l, \sigma(l)] \\ &= \sum_{\sigma \in \mathrm{Perm}(t)} \operatorname{sgn}(\sigma) \theta^{E_\sigma} \end{aligned} \tag{2.10}$$

$$\text{where} \qquad E_\sigma = \sum_{l=1}^{t} (i_l - 1)(j_{\sigma(l)} - 1),$$

$\mathrm{Perm}(t)$ is the set of all permutations on t elements, and $\operatorname{sgn}(\sigma) \in \{-1, 1\}$ is the sign of permutation σ. Clearly, $\det(\mathbf{R})$ defines a univariate polynomial $f_{\mathbf{R}} \in \mathbb{F}_p[\theta]$. We will now show that $\deg(f_{\mathbf{R}}) = \sum_{l=1}^{t} (i_l - 1)(j_l - 1)$ by showing that there is a unique permutation $\sigma^* \in \mathrm{Perm}(t)$ for which E_{σ^*} achieves this value, and that this is the

maximum over all permutations in $\text{Perm}(t)$. This means that $f_{\mathbf{R}}$ has a leading term of degree E_{σ^*}.

To prove this statement, we show that any permutation $\sigma \in \text{Perm}(t) \backslash \{\sigma^*\}$ can be modified into a permutation σ' such that $E_{\sigma'} > E_\sigma$. Specifically, we show that $\sigma^* = \sigma_{\text{id}}$, the identity permutation. Consider $\sigma \in \text{Perm}(t) \backslash \{\sigma_{\text{id}}\}$: let a be the smallest index such that $\sigma(a) \neq a$, let $b = \sigma^{-1}(a)$, and let $c = \sigma(a)$. Let σ' be such that $\sigma'(a) = a$, $\sigma'(b) = c$, and $\sigma'(d) = \sigma(d)$ for $d \in [t] \backslash \{a, b\}$. In other words, σ' is the result of "swapping" the images of a and b in σ. Notice that $a < b$ and $a < c$. Then, we have that:

$$\begin{aligned} E_{\sigma'} - E_\sigma &= (i_a - 1)(j_a - 1) + (i_b - 1)(j_c - 1) - (i_a - 1)(j_c - 1) - (i_b - 1)(j_a - 1) \\ &= (i_b - i_a)(j_c - j_a) > 0 \end{aligned}$$

The last inequality comes from the fact that $a < b$ implies $i_a < i_b$ and $a < c$ implies $j_a < j_c$. Therefore, $\deg(f_{\mathbf{R}}) = \max_{\sigma \in \text{Perm}(t)} E_\sigma = E_{\sigma_{\text{id}}}$.

Let $E^*(\lambda^I, k^I, r^I, r^F)$ be the maximum degree of $f_{\mathbf{R}}$ over all submatrices $\mathbf{R}$ of $\mathbf{P}^I$ or $\mathbf{P}^F$. Then, $E^*(\lambda^I, k^I, r^I, r^F)$ corresponds to the diagonal with the largest elements in $\mathbf{P}^I$ or $\mathbf{P}^F$. In $\mathbf{P}^F$ this is the diagonal of the square submatrix formed by the bottom r^F rows. In $\mathbf{P}^I$ it can be either the diagonal of the square submatrix formed by the bottom r^I rows, or by the right k^I columns. Thus, we have that:

$$E^*(\lambda^I, k^I, r^I, r^F) = \max\{E_1, E_2, E_3\}$$

$$\begin{aligned}
\text{where } E_1 &= \sum_{i=0}^{r^F-1} i(\lambda^I k^I - r^F + i) \\
&= r^F(r^F-1)(3\lambda^I k^I - r^F - 1)/6, \\
E_2 &= \sum_{i=0}^{r^I-1} i(k^I - r^I + i) \\
&= r^I(r^I-1)(3k^I - r^I - 1)/6, \\
E_3 &= \sum_{i=0}^{k^I-1} i(r^I - k^I + i) \\
&= k^I(k^I-1)(3r^I - k^I - 1)/6.
\end{aligned}$$

Recall that we defined the field size as $q = p^D$ for any prime p. We set $D = (E^*(\lambda^I, k^I, r^I, r^F) + 1)$. Then, if $\det(\mathbf{R}) = 0$ for some submatrix $\mathbf{R}$, θ is a root of $f_{\mathbf{R}}$, which is a contradiction since θ is a primitive element and the minimal polynomial of θ over $\mathbb{F}_p$ has degree $D > \deg(f_{\mathbf{R}})$ [97]. □

Combining the above results leads to the following key result on the achievability of the lower bounds on access cost derived in Section 2.1.

Theorem 2.12. *The explicit construction provided in Section 2.2.1 yields access-optimal linear MDS convertible codes for all parameter values in the merge regime.*

Proof. Follows from Theorem 2.9, Lemma 2.10, and Lemma 2.11. □

The construction presented in this section is practical only for small values of the parameters since the required field size grows exponentially with the lengths of the initial and final codes. In Section 2.3 we present practical low-field-size constructions.

2.3 Low field-size convertible codes in the merge regime based on superregular Hankel arrays

In this section we present alternative constructions for $(n^I, k^I; n^F, k^F = \lambda^I k^I)$ convertible code that require a significantly lower (polynomial) field size than the construction

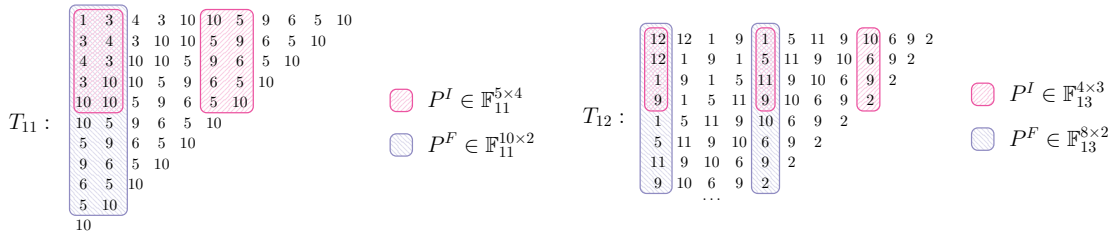
presented in Section 2.2. We start by explaining the key ideas behind these constructions and present two examples that represent two extremes of a tradeoff between field size and coverage of parameter values. In Section 2.3.1, we describe the general construction, which includes codes at the two extremes of the tradeoff and a sequence of constructions in between. In Section 2.3.2, we show that the proposed code construction can support access-optimal conversion even when parameters of the final code are a priori unknown.

The key idea behind our constructions is to take the matrices $\mathbf{P}^I$ and $\mathbf{P}^F$ as cleverly-chosen submatrices from a specially constructed triangular array of the following form:

$$T_m: \quad \begin{array}{cccccc} b_1 & b_2 & b_3 & \cdots & b_{m-1} & b_m \\ b_2 & b_3 & \cdots & \cdots & b_m & \\ b_3 & \vdots & \iddots & \iddots & & \\ \vdots & \vdots & \iddots & & & \\ b_{m-1} & b_m & & & & \\ b_m & & & & & \end{array} \tag{2.11}$$

with the property that every submatrix of T_m is superregular (the submatrix must lie completely within the triangular array). Here, (1) $(b_1, \ldots, b_m)$ are (not necessarily distinct) elements from $\mathbb{F}_q$, and (2) m is at most the field size q. The array T_m has Hankel form, that is, $T_m[i, j] = T_m[i-1, j+1]$, for all $i \in [2, m]$, $j \in [m-1]$. We denote T_m a *superregular Hankel array*. Such an array can be constructed by employing the algorithm proposed in [100] (where the algorithm was employed to generate generalized Cauchy matrices to construct generalized Reed-Solomon codes).

We construct the initial and final codes by taking submatrices $\mathbf{P}^I$ and $\mathbf{P}^F$ from superregular Hankel arrays in a special manner. This guarantees that $\mathbf{P}^I$ and $\mathbf{P}^F$ are superregular. In addition, we exploit the Hankel form of the array by carefully choosing the submatrices that form $\mathbf{P}^I$ and $\mathbf{P}^F$ to ensure that $\mathbf{P}^F$ is r^F-column block-constructible from $\mathbf{P}^I$. Given the way we construct these matrices and the properties of T_m, all the initial and final codes presented in this section turn out to be inside a well-studied class of codes known as (punctured) *generalized doubly-extended*



(a) Hankel-I construction. **(b)** Hankel-II construction.

Figure 2.2: Examples of constructions based on Hankel arrays: (a) Hankel-I construction parity generator matrices for systematic $(9, 5; 12, 10)$ convertible code. Notice how matrix $\mathbf{P}^F$ corresponds to the vertical concatenation of the first two columns and the last two columns of matrix $\mathbf{P}^I$. (b) Hankel-II construction parity generator matrices for systematic $(7, 4; 10, 8)$ convertible code. Notice how matrix $\mathbf{P}^F$ corresponds to the vertical concatenation of the first and second column of $\mathbf{P}^I$, and the second and third column of $\mathbf{P}^I$.

Reed-Solomon codes [100].

The above idea yields a sequence of constructions with a tradeoff between the field size and the maximum value of r^F supported. We first present two examples that correspond to the extreme ends of this tradeoff, which we call *Hankel-I* and *Hankel-II*. Construction Hankel-I, shown in Example 2.1, can be applied whenever $r^F \leq \lfloor r^I/\lambda^I \rfloor$, and requires a field size of $q \geq (\max\{n^I, n^F\} - 1)$. Construction Hankel-II, shown in Example 2.2, can be applied whenever $r^F \leq (r^I - \lambda^I + 1)$, and requires a field size of $q \geq k^I r^I$.

Throughout this section we will assume that $\lambda^I \leq r^I \leq k^I$. The ideas presented here are still applicable when $r^I > k^I$, but the constructions and analysis change in minor ways.

Example 2.1 (Hankel-I)**:** Consider the parameters $(9, 5; 12, 10)$ and the field $\mathbb{F}_{11}$ (any finite field of size at least 11 suffices, but we choose a prime field for ease of

explanation). Notice that these parameters satisfy:

$$r^F = 2 \leq \left\lfloor \frac{r^I}{\lambda^I} \right\rfloor = 2, \text{ and}$$
$$q = 11 \geq \max\{n^I, n^F\} - 1 = 11.$$

First, construct a superregular Hankel array of size $n^F - 1 = 11$, T_{11}, employing the algorithm in [100]. Then, divide the $r^I = 4$ initial parities into $\lambda^I = 2$ groups: encoding vectors of parities in the same group will correspond to contiguous columns of T_{11}. The submatrix $\mathbf{P}^I \in \mathbb{F}_{11}^{5\times 4}$ is formed from the top $k^I = 5$ rows and columns $1, 2, k^I + 1 = 6$ and $k^I + 2 = 7$ of T_{11}, as shown in Figure 2.2a. The submatrix $\mathbf{P}^F \in \mathbb{F}_{11}^{10\times 2}$ is formed from the top $k^I = 10$ rows and columns $1, 2$ of T_{11}, as shown in Figure 2.2a. Checking that these matrices are superregular follows from the superregularity of T_{11}. It is straightforward to check that both these matrices are superregular, which follows from the the superregularity of T_{11}. Furthermore, notice that the chosen parity matrices have the following structure:

$$\mathbf{P}^I = \begin{bmatrix} \top & \top & \top & \top \\ \mathbf{p_1} & \mathbf{p_2} & \mathbf{p_3} & \mathbf{p_4} \\ \bot & \bot & \bot & \bot \end{bmatrix}, \qquad \mathbf{P}^F = \begin{bmatrix} \top & \top \\ \mathbf{p_1} & \mathbf{p_2} \\ \bot & \bot \\ \top & \top \\ \mathbf{p_3} & \mathbf{p_4} \\ \bot & \bot \end{bmatrix}.$$

From this structure, it is clear that $\mathbf{P}^F$ is 2-column block-constructible from $\mathbf{P}^I$. Therefore, $\mathbf{P}^I$ and $\mathbf{P}^F$ satisfy the sufficient conditions of Theorem 2.9, and define an access-optimal convertible code. ▸

Example 2.2 (Hankel-II)**:** Consider parameters $(7, 4; 10, 8)$ and field $\mathbb{F}_{13}$ (any finite field of size at least 12 suffices, but we choose a prime field for ease of explanation). Notice that these parameters satisfy:

$$r^F = 2 \leq r^I - \lambda^I + 1 = 2 \quad \text{and} \quad q = 13 \geq k^I r^I = 12$$

First, construct a superregular Hankel array of size $k^I r^I = 12$, T_{12}, by choosing $q = 13$

$$\mathbf{P}^I = \left[\begin{array}{ccc|c|ccc|c|ccc} b_1 & \cdots & b_t & \cdots & b_{(i-1)k^I+1} & \cdots & b_{(i-1)k^I+t} & \cdots & b_{(s-1)k^I+1} & \cdots & b_{(s-1)k^I+t} \\ b_2 & \cdots & b_{t+1} & \cdots & b_{(i-1)k^I+2} & \cdots & b_{(i-1)k^I+t+1} & \cdots & b_{(s-1)k^I+2} & \cdots & b_{(s-1)k^I+t+1} \\ \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots \\ b_{k^I} & \cdots & b_{k^I+t-1} & \cdots & b_{ik^I} & \cdots & b_{ik^I+t-1} & \cdots & b_{sk^I} & \cdots & b_{sk^I+t-1} \end{array}\right]$$

$$\mathbf{P}^F = \left[\begin{array}{ccc|c|ccc|c|ccc} b_1 & \cdots & b_t & \cdots & b_{(i-1)k^I+1} & \cdots & b_{(i-1)k^I+t} & \cdots & b_{(s-\lambda^I)k^I+1} & \cdots & b_{(s-\lambda^I)k^I+t} \\ b_2 & \cdots & b_{t+1} & \cdots & b_{(i-1)k^I+2} & \cdots & b_{(i-1)k^I+t+1} & \cdots & b_{(s-\lambda^I)k^I+2} & \cdots & b_{(s-\lambda^I)k^I+t+1} \\ \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots \\ b_{\lambda^I k^I} & \cdots & b_{\lambda^I k^I+t-1} & \cdots & b_{(i+\lambda^I-1)k^I} & \cdots & b_{(i+\lambda^I-1)k^I+t-1} & \cdots & b_{sk^I} & \cdots & b_{sk^I+t-1} \end{array}\right].$$

Figure 2.3: Generator matrix for initial and final parities in Hankel$_s$ construction. The vertical bars separate groups of columns. In matrix $\mathbf{P}^I$, the index i ranges from 1 to s. In matrix $\mathbf{P}^F$, the index i ranges from 1 to $(s - \lambda^I + 1)$.

as the field size, and employing the algorithm in [100]. The submatrix $\mathbf{P}^I \in \mathbb{F}_{13}^{4\times 3}$ is formed by the top $k^I = 4$ rows and columns $\{1, (k^I + 1) = 5, (2k^I + 1) = 9\}$ of T_{12}, as shown in Figure 2.2b. The submatrix $\mathbf{P}^F \in \mathbb{F}_{13}^{8\times 2}$ is formed by the top $k^F = 8$ rows and columns $\{1, (k^I + 1) = 5\}$ of T_{12}, as shown in Figure 2.2b. It is easy to check that $\mathbf{P}^I$ and $\mathbf{P}^F$ are superregular, which follows from the superregularity of T_{12}. Furthermore, notice that the chosen parity matrices have the following structure:

$$\mathbf{P}^I = \begin{bmatrix} \top & \top & \top \\ \mathbf{p_1} & \mathbf{p_2} & \mathbf{p_3} \\ \bot & \bot & \bot \end{bmatrix}, \qquad \mathbf{P}^F = \begin{bmatrix} \top & \top \\ \mathbf{p_1} & \mathbf{p_2} \\ \bot & \bot \\ \top & \top \\ \mathbf{p_2} & \mathbf{p_3} \\ \bot & \bot \end{bmatrix}.$$

It is easy to see that $\mathbf{P}^F$ is 2-column block-constructible from $\mathbf{P}^I$. Therefore, $\mathbf{P}^I$ and $\mathbf{P}^F$ satisfy the sufficient conditions of Theorem 2.9, and define an access-optimal convertible code. ▸

2.3.1 General Hankel-array-based construction of convertible codes

In this subsection, we present a sequence of Hankel-array-based constructions of access-optimal MDS convertible codes. This sequence of constructions presents a tradeoff between field size and the range of r^F supported. To index the sequence we use $s \in \{\lambda^I, \lambda^I + 1, \ldots, r^I\}$ which corresponds to the number of groups into which the initial parity encoding vectors are divided. Given parameters $\{k^I, r^I, \lambda^I\}$ and a field $\mathbb{F}_q$, construction Hankel_s ($s \in \{\lambda^I, \lambda^I + 1, \ldots, r^I\}$) supports:

$$r^F \leq (s - \lambda^I + 1)\left\lfloor \frac{r^I}{s} \right\rfloor + \max\{(r^I \bmod s) - \lambda^I + 1, 0\},$$
$$\text{requiring} \quad q \geq \max\{sk^I + \left\lfloor \frac{r^I}{s} \right\rfloor - 1, n^I - 1\}.$$

Therefore, Hankel-I, from Example 2.1 corresponds to $\text{Hankel}_{\lambda^I}$ and Hankel-II from Example 2.2 corresponds to Hankel_{r^I}.

Construction of Hankel_s

Assume, for the sake of simplicity, that $k^I \geq r^I$, $s \mid r^I$ and let $t = (r^I/s)$. Now we describe how to construct $\mathbf{P}^I$ and $\mathbf{P}^F$ over a field $\mathbb{F}_q$ whenever:

$$r^F \leq (s - \lambda^I + 1)t \quad \text{and} \quad q \geq sk^I + t - 1.$$

Without loss of generality, we consider $r^F = (s - \lambda^I + 1)t$ (lesser values of r^F can be obtained by puncturing the final code, i.e., eliminating some of the final parities). Let T_m be as in Equation (2.11), with $m = (sk^I + t - 1)$. Divide the r^I initial parity encoding vectors into s disjoint sets $(S_1, S_2, \ldots, S_s)$ of size t each. We associate each set S_i ($i \in [s]$) with a set of column indices $\text{col}(S_i) = \{(i-1)k^I + 1, (i-1)k^I + 2, \ldots, (i-1)k^I + t\}$ of T_m. Matrix $\mathbf{P}^I$ is the submatrix formed by the top k^I rows and the columns indexed by the set $(\text{col}(S_1) \cup \cdots \cup \text{col}(S_s))$ of T_m. Matrix $\mathbf{P}^F$ is the submatrix formed by the top $\lambda^I k^I$ rows and the columns indexed by the set

$(\text{col}(S_1) \cup \cdots \cup \text{col}(S_{s-\lambda^I+1}))$ of T_m. The resulting matrices $\mathbf{P}^I$ and $\mathbf{P}^F$ are shown in Figure 2.3. In the case where $s \nmid r^I$, we form an additional set S_{s+1} with the remaining $(r^I \mod s)$ initial parity encoding vectors, and proceed as above.

Theorem 2.13. *Given parameters k^I, r^I, λ^I, and a field $\mathbb{F}_q$ Hankel$_s$ ($s \in \{\lambda^I, \ldots, r^I\}$) constructs an access-optimal $(n^I, k^I; n^F, k^F = \lambda^I k^I)$ convertible code if:*

$$r^F \leq (s - \lambda^I + 1)\left\lfloor \frac{r^I}{s} \right\rfloor + \max\{(r^I \bmod s) - \lambda^I + 1, 0\}$$
$$\text{and} \quad q \geq \max\{sk^I + \left\lfloor \frac{r^I}{s} \right\rfloor - 1, n^I - 1\}.$$

Proof. Consider the construction Hankel$_s$ described in this section, for some $s \in \{\lambda^I, \ldots, r^I\}$. The Hankel form of T_m and the manner in which $\mathbf{P}^I$ and $\mathbf{P}^F$ are constructed guarantees that the l-th column of $\mathbf{P}^F$ corresponds to the vertical concatenation of columns $\{l, l+t, \ldots, l+(\lambda^I-1)t\}$ of $\mathbf{P}^I$. Thus, $\mathbf{P}^F$ is r^F-column block-constructible from $\mathbf{P}^I$. Furthermore, since $\mathbf{P}^I$ and $\mathbf{P}^F$ are submatrices of T_m, they are superregular. Thus $\mathbf{P}^I$ and $\mathbf{P}^F$ satisfy both of the properties laid out in Theorem 2.9 and hence the convertible code constructed by Hankel$_s$ is access-optimal. □

Conversion procedure

During conversion, the k^I data symbols from each of the λ^I initial codewords remain unchanged, and become the $k^F = \lambda^I k^I$ data symbols from the final codeword. The r^F new (parity) symbols from the final codeword are constructed by accessing symbols from the initial codewords as detailed below. To construct the l-th new symbol (corresponding to the l-th column of $\mathbf{P}^F$, $l \in [r^F]$), read parity symbol $(l + (i-1)t)$ from each initial codeword $i \in [\lambda^I]$, and then sum the λ^I symbols read. The encoding vector of the new symbol will be equal to the sum of the encoding vectors of the symbols read. This is done for every new encoding vector $l \in [r^F]$.

2.3.2 Handling a priori unknown parameters

In practice, the final parameters (n^F, k^F) might be unknown at the time of code construction, as they might depend on the empirically observed failure rates. Thus, it is of interest to construct initial codes that are (n^F, k^F)-access-optimally convertible for all (n^F, k^F) in a given set. The general construction and the Hankel-array based constructions presented above indeed provide such a property.

Proposition 2.14. *Every initial code from an $(n^I, k^I; n^F, k^F = \lambda^I k^I)$ convertible code constructed using the constructions in this section and Section 2.2 is also $(n^{F'}, k^{F'})$-access-optimally convertible for any $k^{F'} = \lambda^{I'} k^I$ and $n^{F'} = (r^{F'} + k^{F'})$ with $0 \leq r^{F'} \leq r^F$ and $2 \leq \lambda^{I'} \leq \lambda^I$.*

Proof. The conversion procedure can be easily modified to take fewer initial codewords (i.e. by treating some of the initial codewords as all-zero codewords) or construct fewer parity symbols. Since the access cost associated with each initial codeword is $\min\{k^I, r^I\}$, and the access cost associated with every parity symbol is $\lambda^I + 1$, the resulting conversion procedure has optimal access cost. □

Thus, to support access-optimal conversion for all parameters $(n^F = \lambda^I k^I + r^F, k^F = \lambda^I k^I)$ in a given finite set of values for λ^I and r^F, it suffices to construct an access-optimal convertible code using the largest parameter λ^I and r^F in the set. Then, by Proposition 2.14, the initial code will support access-optimal conversion for all parameter values in the given set.

2.4 Split regime

The *split regime* of convertible codes corresponds to the case where a single initial codeword is split into multiple final codewords. This regime is, in some sense, the opposite of the merge regime, in which multiple initial codewords are combined into one final codeword. Specifically, an $(n^I, k^I; n^F, k^F)$ convertible code is in the split regime if $k^I = \lambda^F k^F$ for an integer $\lambda^F \geq 2$, with arbitrary n^I and n^F. Notice that in this regime we have that $M = \text{lcm}(k^I, k^F) = k^I$ and $\lambda^I = 1$.

First, in Section 2.4.1, we show a lower bound on access cost for the split regime. In Section 2.4.2 we show a matching upper bound on access cost by showing that for every systematic $[n^I, \lambda^F k^F]$ MDS code $\mathcal{C}$ there exists an access-optimal $(n^I, k^I = \lambda^F k^F; n^F, k^F)$ convertible code having $\mathcal{C}$ as its initial code by presenting a conversion procedure whose cost matches the lower bound.

2.4.1 Access cost lower bound for the split regime

In this subsection, we lower bound the access cost of conversion in the split regime. This is done by first showing a lower bound on write access cost, and then showing a lower bound on the read access cost of conversion.

The following fact simplifies the analysis of the split regime.

Proposition 2.15. *For a linear MDS $(n^I, k^I = \lambda^F k^F; n^F, k^F)$ convertible code, all possible pairs of initial and final partitions are equivalent (up to relabeling).*

Proof. There is only one possible initial partition $\mathcal{P}^I = \{[k^I]\}$, hence any two final partitions can be made equivalent by relabeling symbols. □

Therefore, we do not need to consider differences in partitions in our analysis of the split regime.

Proposition 2.16. *In a linear MDS $(n^I, k^I = \lambda^F k^F; n^F, k^F)$ convertible code, there are at most k^F unchanged symbols in each of the final codewords (i.e., at least r^F new symbols per codeword). Hence, there are at most k^I unchanged symbols in total.*

Proof. For any final codeword $i \in [\lambda^F]$, any subset $\mathcal{V} \subseteq \mathcal{S}_i^F$ of size at least $k^F + 1$ is linearly dependent due to the MDS property. Thus, $\mathcal{V} \subseteq \mathcal{S}^I$ contradicts the fact that $\mathcal{C}^I$ is MDS. Hence, each final codeword i has at most k^F unchanged symbols. □

Therefore, the total write access cost in the split regime is at least $\lambda^F r^F$.

Now we focus on bounding the read access cost. The general strategy we use to obtain bounds on read access cost is to consider a specially chosen set $\mathcal{W}$ of k^F symbols from a final codeword, which by the MDS property of the final code is enough

to decode all data in that codeword. We then use the fact that final codewords are the result of conversion to identify a set $\mathcal{V}$ of initial symbols that contain all the information contained in $\mathcal{W}$. The MDS property of the initial code constrains the information available in $\mathcal{V}$, which allows us to derive a lower bound on its size and thus a lower bound on the number of read symbols.

Lemma 2.17. *For all linear MDS $(n^I, k^I = \lambda^F k^F; n^F, k^F)$ convertible codes, the read access set $\mathcal{D}$ satisfies $|\mathcal{D}| \geq (\lambda^F - 1)k^F + \min\{r^F, k^F\}$.*

Proof. If $r^F \geq k^F$, then all data should be decodable by accessing only new symbols in the final codewords, and the result follows easily since all data must have been read to create the new symbols. Therefore, assume for the rest of this proof that $r^F < k^F$.

Suppose, for the sake of contradiction, that $|\mathcal{D}| < (\lambda^F - 1)k^F + r^F$. Let u be a symbol in some final codeword $i \in [\lambda^F]$ which is neither read nor written. Such a codeword and symbol exist since otherwise every symbol in the final codewords would be accessed (for either read or write) and thus $\mathcal{S}^F$ would be in the span of $\mathcal{A}$, which is a contradiction since $\text{rk}(\mathcal{S}^F) = k^I$.

Let $\mathcal{W}_1$ be a subset of symbols of the same final codeword i such that $\mathcal{W}_1 \subseteq \mathcal{N}_i$ and $|\mathcal{W}_1| = r^F$. Such a subset exists by virtue of Proposition 2.16. Further, let $\mathcal{W}_2 \subseteq \mathcal{S}_i^F \setminus (\mathcal{W}_1 \cup \{u\})$ be such that $|\mathcal{W}_2| = k^F - r^F$. Clearly $\mathcal{W} = \mathcal{W}_1 \cup \mathcal{W}_2$ is of size $|\mathcal{W}| = k^F$ and can reconstruct the contents of u, by the MDS property of the final code. In other words, $u \in \text{span}(\mathcal{W})$.

Let $\mathcal{W}_2' = (\mathcal{W}_2 \cap \mathcal{U}_i)$ be the unchanged symbols in $\mathcal{W}_2$. Since $\mathcal{W}_1$ and $\mathcal{W}_2 \setminus \mathcal{W}_2'$ only have new symbols, they are both contained in $\text{span}(\mathcal{A})$, therefore $\mathcal{W} \subseteq \text{span}(\mathcal{A} \cup \mathcal{W}_2')$. Notice that the subset $\mathcal{V} = (\mathcal{A} \cup \mathcal{W}_2')$ consists only of initial symbols. Furthermore, it holds that $\text{rk}(\mathcal{A}) \leq |\mathcal{D}|$ and $\text{rk}(\mathcal{W}_2') \leq |\mathcal{W}_2| = k^F - r^F < k^F$. Thus:

$$\text{rk}(\mathcal{V}) \leq \text{rk}(\mathcal{A}) + \text{rk}(\mathcal{W}_2') \leq |\mathcal{D}| + (k^F - r^F) < k^I.$$

This implies that $\mathcal{W}$ is spanned by less than k^I initial symbols (which do not include u). However, by the MDS property of the initial code, any subset of less than k^I

initial symbols that does not contain symbol u, has no information about u. This causes a contradiction with the fact that $u \in \text{span}(\mathcal{W}) \subseteq \text{span}(\mathcal{V})$. Thus, we must have $|\mathcal{D}| \geq (\lambda^F - 1)k^F + r^F$. □

It is easy to show that if we only read unchanged symbols, it is not possible to do better than the default approach. This follows from the fact that unchanged symbols are already present in the final codewords and hence using them to create the new symbols will contradict with the MDS property. Retired symbols, on the other hand, do not have this drawback. Thus, intuitively, based on Lemma 2.17, one might expect to achieve an efficient conversion by reading from the retired symbols. However, we next show that it is not possible to achieve lower read access cost than the default approach when $r^I < r^F$.

Lemma 2.18. *For all linear MDS $(n^I, k^I = \lambda^F k^F; n^F, k^F)$ convertible codes, if $r^I < r^F$ then the read access set $\mathcal{D}$ satisfies $|\mathcal{D}| \geq \lambda^F k^F$.*

Proof. Suppose, for the sake of contradiction, that $|\mathcal{D}| < \lambda^F k^F$. Let u be a symbol in some final codeword $i \in [\lambda^F]$ which is neither read nor written. Such a codeword and symbol always exist as described in the proof of Lemma 2.17. We will choose a subset of symbols $\mathcal{W} \subseteq \mathcal{S}_i^F$ of size $|\mathcal{W}| = k^F$. By the MDS property of the final code, symbol u is decodable from $\mathcal{W}$, i.e., $u \in \text{span}(\mathcal{W})$. There are two cases for the choice of $\mathcal{W}$ depending on the total number of accessed symbols in codeword i:
Case 1: If $|\mathcal{N}_i| + |\mathcal{U}_i \cap \mathcal{A}| \geq k^F$, then let $\mathcal{W} \subseteq \mathcal{N}_i \cup (\mathcal{U}_i \cap \mathcal{A})$. That is, $\mathcal{W}$ only contains symbols that are read or written. It is easy to see that $\mathcal{W} \subseteq \text{span}(\mathcal{A})$.

Clearly, $\mathcal{A}$ contains only initial symbols, and the following holds:

$$\text{rk}(\mathcal{A}) \leq |\mathcal{D}| < \lambda^F k^F = k^I.$$

However, this is a contradiction with the fact that $u \in \text{span}(\mathcal{W})$, since by the MDS property of the initial code, $\mathcal{A}$ contains no information about symbol u.
Case 2: If $|\mathcal{N}_i| + |\mathcal{U}_i \cap \mathcal{A}| < k^F$, then choose $\mathcal{W} = (\mathcal{W}_1 \cup \mathcal{W}_2)$, where $\mathcal{W}_1 = (\mathcal{N}_i \cup (\mathcal{U}_i \cap \mathcal{A}))$ and $\mathcal{W}_2$ is any subset of $(\mathcal{S}_i^F \setminus (\mathcal{W}_1 \cup \{u\}))$ of size $|\mathcal{W}_2| = k^F - |\mathcal{W}_1|$. That is, $\mathcal{W}$ contains all the symbols of final codeword i that are read or written

(in addition to other unchanged symbols distinct from u). It is easy to see that $\mathcal{W}_1 \subseteq \text{span}(\mathcal{A})$ and thus $\mathcal{W} \subseteq \text{span}(\mathcal{A} \cup \mathcal{W}_2)$. Furthermore, the subset $\mathcal{V} = (\mathcal{A} \cup \mathcal{W}_2)$ consists only of initial symbols.

Notice that there are at most $(|\mathcal{S}^I| - |\mathcal{U}_i|) = (k^I + r^I - |\mathcal{U}_i|)$ read symbols outside of final codeword i (i.e., in $\mathcal{S}^I \setminus \mathcal{U}_i$). Therefore, we can bound $\text{rk}(\mathcal{A})$ by $\text{rk}(\mathcal{A}) \leq k^I + r^I - |\mathcal{U}_i| + |\mathcal{U}_i \cap \mathcal{A}|$. On the other hand, it is clear that $\text{rk}(\mathcal{W}_2) \leq |\mathcal{W}_2| = k^F - |\mathcal{N}_i| - |\mathcal{U}_i \cap \mathcal{A}|$. Combining these, we get:

$$\begin{aligned}\text{rk}(\mathcal{V}) &\leq \text{rk}(\mathcal{A}) + \text{rk}(\mathcal{W}_2) \\ &\leq k^I + r^I + k^F - |\mathcal{U}_i| - |\mathcal{N}_i| \\ &\leq k^I + r^I - r^F \\ &< k^I.\end{aligned}$$

However, this is a contradiction with the fact that $u \in \text{span}(\mathcal{W}) \subseteq \text{span}(\mathcal{V})$, since by the MDS property of the initial codes, $\mathcal{V}$ contains no information about symbol u. □

By combining all the results in this subsection, we obtain the following lower bound on the access cost of conversion in the split regime.

Theorem 2.19. *The total access cost of any linear MDS $(n^I, k^I = \lambda^F k^F; n^F, k^F)$ convertible code is at least $(\lambda^F - 1)k^F + \min\{r^F, k^F\} + \lambda^F r^F$ if $r^I \geq r^F$, and at least $\lambda^F n^F$ otherwise.*

Proof. Follows from Proposition 2.16 and Lemmas 2.17 and 2.18. □

As we show in the next subsection, this lower bound is tight since it is achievable.

2.4.2 Access-optimal convertible codes for the split regime

In this subsection we present a construction of access-optimal convertible codes in the split regime. Under this construction, any systematic MDS code can be used as the initial code. The final code corresponds to the projection of the initial code onto the

coordinates of any k^F systematic symbols. Since our construction can be applied to existing codes and only specifies the conversion procedure, we introduce the following definition capturing the property of codes that can be converted efficiently.

Definition 2.2: A code $\mathcal{C}^I$ is (n^F, k^F)-*optimally convertible* if and only if there exists an $[n^F, k^F]$ code $\mathcal{C}^F$ (along with partitions and conversion procedure) that form an access-optimal $(n^I, k^I; n^F, k^F)$ convertible code. ▸

The conversion procedure that leads to optimal access cost (meeting the lower bound in Theorem 2.19) is as follows.

Conversion procedure: All the systematic symbols are used as unchanged symbols. When $r^I < r^F$ or $r^F \geq k^F$, the conversion is trivial since one cannot do better than the default approach. The conversion procedure for the nontrivial case proceeds as follows. For all but one final codeword, all unchanged symbols are read ($(\lambda^F - 1)k^F$ in total), and the new symbols are naively constructed from them. For the remaining final codeword, r^F retired symbols are read, and then the unchanged symbols from the other final codewords are used to remove their interference from the retired symbols to obtain r^F new symbols.

Theorem 2.20. *Every systematic linear MDS* $[n^I, k^I = \lambda^F k^F]$ *code* $\mathcal{C}^I$ *is* (n^F, k^F)-*optimally convertible.*

Proof. If $r^F > \min\{r^I, k^F\}$, then the default approach achieves the bound stated in Theorem 2.19. Thus, assume $r^F \leq \min\{r^I, k^F\}$. Let $\mathbf{G}^I = [\mathbf{I} \mid \mathbf{P}^I]$ be the generator matrix of $\mathcal{C}^I$ and assume symbols are numbered in the same order as the columns of $\mathbf{G}^I$. Define $\mathcal{C}^F$ as the code generated by the matrix formed by taking the first k^F rows of $\mathbf{G}^I$, and columns $1, \ldots, k^F$ and $k^I + 1, \ldots, k^I + r^F$. Let $(i-1)k^F + 1, \ldots, ik^F$ be the columns of the unchanged symbols corresponding to final codeword $i \in [\lambda^F]$. Consider the following conversion procedure: read the the subset of unchanged symbols $U = \{k^F + 1, \ldots, \lambda^F k^F\}$ and the retired symbols $R = \{k^I + 1, \ldots, k^I + r^F\}$. To construct the new symbols for codeword 1, simply project the symbols of R onto their first k^F coordinates by using symbols U. To construct the new symbols for

codeword $i \neq 1$, simply use then symbols in U. This conversion procedure reads a total of $|U| + |R| = (\lambda^F - 1)k^F + r^F$ symbols and writes a total of $\lambda^F r^F$ new symbols, which matches the bound from Theorem 2.19. □

Notice that convertible codes created using the construction above are stable. We show this property is, in fact, necessary.

Lemma 2.21. *All access-optimal convertible codes for the split regime are stable.*

Proof. Theorem 2.20 shows that there exist stable access-optimal codes for the split regime. Since any unstable convertible code must incur higher write access cost and at least as much read access cost, it cannot be access-optimal. □

2.5 General regime

In this section, we will study the general regime of convertible codes with arbitrary valid parameter values (i.e. any $n^I > k^I$ and $n^F > k^F$). Recall that the choice of partition functions was inconsequential in the split and merge regimes. In contrast, it turns out that *the choice of initial and final partitions play an important role in the general regime*. This makes the general regime significantly harder to analyze. We deal with this complexity by reducing conversion in the general regime to generalized versions of the split and merge conversions, and by *identifying the conditions on initial and final partitions to minimize total access cost.*

In Section 2.5.1, we explore a generalization of the split regime and of the merge regime. In Section 2.5.2, these generalizations are used to lower bound the access cost of conversion in the general regime. In Section 2.5.3, we describe a conversion procedure and construction for access-optimal conversion in the general regime which utilizes ideas from the constructions for generalizations of split and merge regimes.

2.5.1 Generalized split and merge regimes

The generalized split and merge regimes are similar to the split and merge regimes, except that the generalized variants allow for initial or final codewords of unequal

sizes. This flexibility enables the generalized split and merge regimes to be used as building blocks in the analysis of the general regime. In these generalized variants, the message length M is defined to be $\max\{k^I, k^F\}$ (which coincides with the definition of M in the split and merge regime), but now the sets in the initial and final partitions need not be all of the same size.

Since the initial (or final) codewords might be of different lengths, we define them as shortenings of a common code $\mathcal{C}$.

Definition 2.3: An s-shortening of an $[n, k]$ code $\mathcal{C}$ is the code $\mathcal{C}'$ formed by all the codewords in $\mathcal{C}$ that have 0 in a fixed subset of s positions, with those s positions deleted. ▸

Shortening a code has the effect of decreasing the length n and dimension k while keeping $(n-k)$ fixed. It can be shown that an s-shortening of an $[n, k]$ MDS code is an $[n-s, k-s]$ MDS code. *Lengthening* is the inverse operation of shortening, and has the effect of increasing length n and dimension k while keeping $(n-k)$ fixed. For linear codes, an s-lengthening of a code can be defined as adding s additional columns to its parity check matrix. Similarly, it can be shown that for an $[n, k]$ MDS code, there exists an s-lengthening of it that is an $[n+s, k+s]$ MDS code (assuming a large enough field size).

Generalized split regime

In the generalized split regime, $\lambda^I = 1$ is fixed, $\lambda^F > 1$ is arbitrary, and the final partition $\mathcal{P}^F = \{P_1^F, \ldots, P_{\lambda^F}^F\}$ is such that $|P_i^F| = k_i^F$ and $\sum_{i\in[\lambda^F]} k_i^F = k^I$. Let $k_*^F = \max_{i\in[\lambda^F]} k_i^F$. Then $\mathcal{C}^F$ is a $[n^F, k_*^F]$ MDS code, and the code corresponding to each final codeword is some fixed shortening of $\mathcal{C}^F$. In this case, we define $r^F = n^F - k_*^F$.

Definition 2.4: A $(n^I, k^I = \sum_{i=1}^{\lambda^F} k_i^F; n^F, \{k_i^F\}_{i=1}^{\lambda^F})$ convertible code for the generalized split regime is a variant of a convertible code defined by:

1. $\mathcal{C}^I$ and $\mathcal{C}^F$ as $[n^I, k^I]$ and $[n^F, k_*^F]$ codes, where $k_*^F = \max_{i\in[\lambda^F]} k_i^F$,

2. a partition $\mathcal{P}^F = \{P_1^F, \ldots, P_{\lambda^F}^F\}$ where $|P_i^F| = k_i^F$, and
3. a conversion procedure such that each final codeword i, is an s_i-shortening of $\mathcal{C}^F$ where $s_i = k_*^F - k_i^F$.

▸

The generalized split regime has an access cost lower bound similar to the split regime presented in Section 2.4. We show this by showing that a more efficient conversion procedure for the generalized split regime would imply the existence of a conversion procedure for split regime violating Theorem 2.19.

Theorem 2.22. *For all linear MDS* $(n^I, k^I = \sum_{i=1}^{\lambda^F} k_i^F; n^F, \{k_i^F\}_{i=1}^{\lambda^F})$ *convertible codes, the read access set* $\mathcal{D}$ *satisfies:*

$$|\mathcal{D}| \geq k^I - \max\{k_*^F - r^F, 0\}, \text{ where } k_*^F = \max_{i \in [\lambda^F]} k_i^F.$$

Proof. Suppose, for the sake of contradiction, that there exists a conversion procedure with read access cost $|\mathcal{D}| < k^I - \max\{k_*^F - r^F, 0\}$ for some convertible code in the generalized split regime with codes $\mathcal{C}^I$ and $\mathcal{C}^F$. We modify the initial code $\mathcal{C}^I$ by lengthening it to an $[n_s^I, k_s^I]$ MDS code $\mathcal{C}^s$, such that $k_s^I = \lambda^F k_*^F$ and $r^I = n^I - k^I = n_s^I - k_s^I$. This adds $\sum_{i=1}^{\lambda^F}(k_*^F - k_i^F) = (k_s^I - k^I)$ extra "pseudo-symbols" to the initial code, which we denote with $\mathcal{W}$.

We then define a new conversion procedure from code $\mathcal{C}^s$ to code $\mathcal{C}^F$ which uses the conversion procedure for the generalized split regime convertible code as a subroutine, and then simply reads all the added pseudo-symbols to construct the new symbols. This procedure only reads the read access set $\mathcal{D}$ from $\mathcal{C}^s$ along with the $(k_s^I - k^I)$ pseudo-symbols.

Hence, the total read access is,

$$\begin{aligned} |\mathcal{D} \cup \mathcal{W}| &< (k^I - \max\{k_*^F - r^F, 0\}) + (k_s^I - k^I) \\ &\leq (\lambda^F - 1)k_*^F + \min\{r^F, k_*^F\}. \end{aligned}$$

However, the codes $\mathcal{C}^s$ and $\mathcal{C}^F$ with the new conversion procedure clearly form an MDS $(n_s^I, k_s^I = \lambda^F k_*^F; n^F, k_*^F)$ convertible code. Therefore, this is in contradiction to Theorem 2.19. Then, it must hold that $|\mathcal{D}| \geq k^I - \max\{k_*^F - r^F, 0\}$. □

This lower bound is achievable for all pairs of initial and final parameters. Similar to the case of the split regime, shown in Section 2.4.2, we can use any systematic MDS codes as initial and final codes, and access all but a set of symbols of size k_*^F (forming the largest final codeword) to perform this conversion, as described below.

Conversion procedure: All the systematic symbols are used as unchanged symbols. When $r^I < r^F$ or $r^F \geq k_*^F$, the conversion is trivial since one cannot do better than the default approach. The conversion procedure for the nontrivial case proceeds as follows. For all but the largest final codeword, all unchanged symbols are read ($\lambda^F k^F - k_*^F$ in total), and the new symbols are naively constructed from them. For the largest final codeword, the r^F retired symbols are read, and then the unchanged symbols from the other final codewords are used to remove their interference from the retired symbols to obtain r^F new symbols.

Generalized merge regime

In the generalized merge regime, the sets in the initial partition need not be all of the same size. In this case, we fix $M = k^F$ and $\lambda^F = 1$, while $\lambda^I > 1$ is arbitrary. The initial partition $\mathcal{P}^I = \{P_1^I, \ldots, P_{\lambda^I}^I\}$ is such that $|P_i^I| = k_i^I$ and $\sum_{i\in[\lambda^I]} k_i^I = k^F$. Let $k_*^I = \max_{i\in[\lambda^I]} k_i^I$. Then $\mathcal{C}^I$ is a $[n^I, k_*^I]$ MDS code, $r^I = n^I - k_*^I$, and the code corresponding to each initial codeword is some fixed shortening of $\mathcal{C}^I$.

Definition 2.5: A $(n^I, \{k_i^I\}_{i=1}^{\lambda^I}; n^F, k^F = \sum_{i=1}^{\lambda^I} k_i^I)$ convertible code for the generalized merge regime is a variant of a convertible code defined by:

1. $\mathcal{C}^I, \mathcal{C}^F$ as $[n^I, k_*^I]$ and $[n^F, k^F]$ codes, where $k_*^I = \max_{i\in[\lambda^I]} k_i^I$
2. partition $\mathcal{P}^I = \{P_1^I, \ldots, P_{\lambda^I}^I\}$ where $|P_i^I| = k_i^I$, and
3. a conversion procedure such that each initial codeword i, is an s_i-shortening of $\mathcal{C}^I$ where $s_i = k_*^I - k_i^I$.

▸

The next theorem gives a lower bound on the read access cost of a $(n^I, \{k_i^I\}_{i=1}^{\lambda^I}; n^F, k^F = \sum_{i=1}^{\lambda^I} k_i^I)$ convertible code.

Theorem 2.23. *For all $(n^I, \{k_i^I\}_{i=1}^{\lambda^I}; n^F, k^F = \sum_{i=1}^{\lambda^I} k_i^I)$ convertible code, the following holds:*

$$|\mathcal{D}_i| \geq \min\{k_i^I, r^F\} \text{ for all } i \in [\lambda^I].$$

Furthermore, if $r^I < r^F$, then $|\mathcal{D}_i| \geq k_i^I$ for all $i \in [\lambda^I]$.

Proof. Follows from the proofs of Lemmas 10, 11, and 13 in [96], with some straightforward modifications to account for the difference in the number of symbols of each initial codeword. □

We can achieve this lower bound by shortening an access-optimal $(n^I, k_*^I; n_m^F, k_m^F)$ convertible code, where $k_m^F = \lambda^I k_*^I$ and $n_m^F = k_m^F + r^F$.

2.5.2 Access cost lower bound for the general regime

In this subsection, we study the access cost lower bound for conversions in the general regime (i.e., for all valid parameter values, $n^I > k^I$ and $n^F > k^F$). As in the merge and split regime, we show that when $r^I \geq r^F$, significant reduction in access cost can be achieved. However when $r^I < r^F$, one cannot do better than the default approach.

For an $(n^I, k^I; n^F, k^F)$ convertible code with $k^I \neq k^F$ and partitions $(\mathcal{P}^I, \mathcal{P}^F)$, let $k_{i,j} = |P_i^I \cap P_j^F|$ for $(i, j) \in [\lambda^I] \times [\lambda^F]$ and let $k_{i,*} = \max_{j \in [\lambda^F]} k_{i,j}$.

Lemma 2.24. *For all linear MDS $(n^I, k^I; n^F, k^F)$ convertible codes with $k^I \neq k^F$:*

$$|\mathcal{D}_i| \geq k^I - \max\{k_{i,*} - r^F, 0\} \text{ for all } i \in [\lambda^I].$$

Moreover, if $r^I < r^F$ then $|\mathcal{D}_i| \geq k^I$ for all $i \in [\lambda^I]$.

Proof. Let $i \in [\lambda^I]$ be an initial codeword. There are two cases.

Case $k_{i,*} < k^I$: In this case, we can reduce this conversion to a conversion in the generalized split regime by focusing on initial codeword i, and considering messages which are zero everywhere outside of P_i^I. This is equivalent to a $(n^I, k^I; k_{i,*} + r^F, \{k_{i,j}\}_{j=1}^{\lambda^F})$ convertible code. Then, the result follows from Theorem 2.22.
Case $k_{i,*} = k^I$: Let $j = \text{argmax}_{j' \in [\lambda^F]} k_{i,j'}$. In this case, we can reduce this conversion to conversion in the generalized merge regime by focusing on final codeword j, and considering messages which are zero everywhere outside of P_j^F. This is equivalent to a $(n^I, \{k_{i,j}\}_{i=1}^{\lambda^I}; n^F, k^F)$ convertible code. Then, the result follows from Theorem 2.23. □

We prove a lower bound on the total access cost of conversion in the general regime by using Lemma 2.24 on all initial codewords and finding a partition that minimizes the value of the sum.

Theorem 2.25. *For every linear MDS $(n^I, k^I; n^F, k^F)$ convertible code such that $k^I \neq k^F$, it holds that:*

$$|\mathcal{D}| \geq \lambda^I r^F + (\lambda^I \bmod \lambda^F)(k^I - \max\{k^F \bmod k^I, r^F\})$$

if $r^F < \min\{k^I, k^F\}$. Furthermore, if $r^I < r^F$ or $r^F \geq \min\{k^I, k^F\}$, then $|\mathcal{D}| \geq M$.

Proof. Clearly, it holds that $|\mathcal{D}| = \sum_{i=1}^{\lambda^I} |\mathcal{D}_i|$. Then, the case $r^I < r^F$ follows directly from Lemma 2.24. Otherwise, by the same lemma we have:

$$|\mathcal{D}| = \sum_{i=1}^{\lambda^I} |\mathcal{D}_i| \geq \sum_{i=1}^{\lambda^I} k^I - \max\{k_{i,*} - r^F, 0\}. \tag{2.12}$$

First, we consider the case $k^I > k^F$. Notice that in this case $(\lambda^I \bmod \lambda^F) = \lambda^I$ and $(k^F \bmod k^I) = k^F$. If $r^F \geq k^F$, then the result is trivial, so assume $r^F < k^F$. Since $k_{i,*} \leq k^F$ for all $i \in [\lambda^I]$, we have:

$$|\mathcal{D}| \geq \sum_{i=1}^{\lambda^I} k^I - \max\{k_{i,*} - r^F, 0\} \geq \lambda^I (k^I + r^F - k^F),$$

which proves the result.

Now, we consider the case $k^I < k^F$. Assume, for now, that the right hand side of Inequality 2.12 is minimized when:

$$k_{i,*} = \begin{cases} k^I, & \text{for } 1 \leq i \leq (\lambda^I - (\lambda^I \bmod \lambda^F)) \\ (k^F \bmod k^I), & \text{otherwise.} \end{cases} \tag{2.13}$$

Then, from Inequality 2.12 we have:

$$|\mathcal{D}| \geq \lambda^I k^I - (\lambda^I - (\lambda^I \bmod \lambda^F)) \max\{k^I - r^F, 0\} - (\lambda^I \bmod \lambda^F) \max\{(k^F \bmod k^I) - r^F, 0\} \tag{2.14}$$

If $r^F \geq k^I$, then the result is trivial, so assume $r^F < k^I$. Then, by manipulating the terms of Inequality 2.14, the result is obtained.

It only remains to prove that the right hand side of Inequality 2.12 is minimized when Equation (2.13) holds.

Notice that this is equivalent to showing that $s = \sum_{i=1}^{\lambda^I} \max\{k_{i,*} - r^F, 0\}$ is maximized by the proposed assignment. To prove this, we will show that any optimal assignment to the variables $k_{i,j}$ can be modified to be of the proposed form, without decreasing the value of the objective s. Firstly, it is straightforward to check that there exists a feasible assignment to the variables $k_{i,j}$ that satisfies the statement.

Suppose we have an optimal assignment for variable $k_{i,j}$ that is not of the proposed form and assume, without loss of generality, that $k_{1,*} \geq \cdots \geq k_{\lambda^I,*}$. Let $1 \leq i \leq (\lambda^I - (\lambda^I \bmod \lambda^F))$ be the least such that $k_{i,*} < k^I$, and let $j = \text{argmax}_{j' \in [\lambda^F]} k_{i,j'}$. It must hold that $k_{i,*} > \max\{r^F, k^F \bmod k^I\}$, otherwise this assignment could not be optimal. Notice that $k_{i',*} = k^I$ for all $i' < i$ and since $k^I \nmid (k^F - k_{i,*})$, there exists at least one $i' > i$ such that $k_{i',j} > 0$. Furthermore, there exists $j' \neq j$ such that $k_{i,j'} > 0$, since $k_{i,*} < k^I$. Then, we can "swap" elements from $k_{i,j'}$ with $k_{i',j}$. This increases $k_{i,*}$ and decreases $k_{i',*}$ by at most the same amount. Since $k_{i,*} > r^F$, this cannot decrease the value of the objective s. We can repeat this procedure until

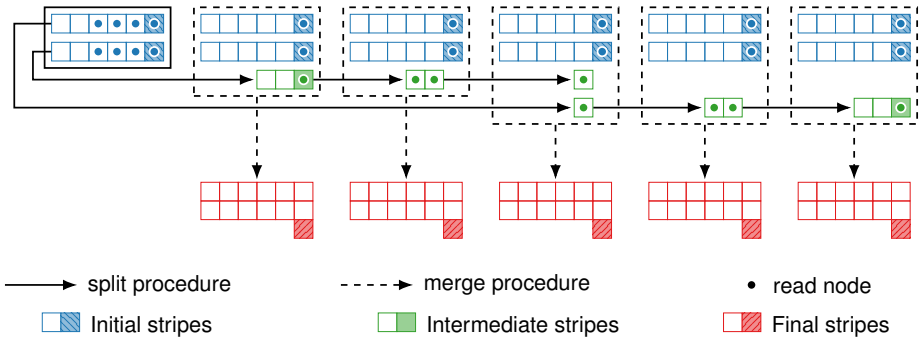


Figure 2.4: Conversion procedure from $[6, 5]$ to $[13, 12]$ ($\lambda^I = 12$ and $\lambda^F = 5$). Read access cost is 18 compared to 60 in the default approach (70% savings).

$k_{i,*} = k^I$ for all $1 \leq i \leq (\lambda^I - (\lambda^I \bmod \lambda^F))$.

Notice now that for every $(\lambda^I - (\lambda^I \bmod \lambda^F)) \leq i \leq \lambda^I$ it holds that:

$$k_{i,*} \leq k^F \bmod k^I \tag{2.15}$$

otherwise, there must exist some $j \in [\lambda^F]$ such that $\sum_{i=1}^{\lambda^I} k_{i,j} > k^F$. If $r^F < (k^F \bmod k^I)$, then Inequality 2.15 must hold with equality. Otherwise, each such $k_{i,*}$ will contribute exactly r^F to the objective s, so they can be modified to be of the desired form without decreasing s. □

2.5.3 Access-optimal convertible codes for the general regime

In this subsection we prove that the lower bound from Theorem 2.25 is achievable by presenting convertible code constructions that are access-optimal in the general regime. We first present the conversion procedure for our construction and then describe the construction of the initial and final codes that are compatible with this conversion procedure.

Conversion procedure

Conversion in the general regime can be achieved by combining the conversion procedures of codes in the generalized split and merge regimes. In the case where

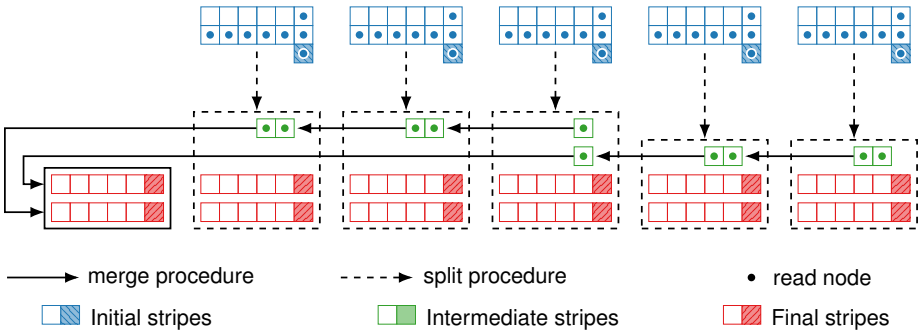


Figure 2.5: Conversion procedure from $[13, 12]$ to $[6, 5]$ ($\lambda^I = 5$ and $\lambda^F = 12$). Read access cost is 40, compared to 60 in the default approach ($33.\overline{3}\%$ savings in read access cost).

$r^I < r^F$, we access k^I symbols from each initial codeword and use the default approach. For the case where $r^I \geq r^F$, we present the conversion procedure by considering three cases: $k^I = k^F$, $k^I < k^F$, and $k^I > k^F$.

Case $k^I = k^F$: Notice that $n^I \geq n^F$ since $r^I \geq r^F$. This is a degenerate case where any n^F symbols from the initial codeword can be kept unchanged.

Case $k^I < k^F$: We will separate the symbols of initial codewords into λ^F disjoint groups with the same amount of information. This requires splitting some initial codewords into what we call *intermediate codewords*, which are then assigned to different groups. We will finally merge each group to form the λ^F final codewords. Specifically (see Figure 2.4):

1. Assign $\lfloor k^F/k^I \rfloor$ initial codewords to each group (dashed boxes in Figure 2.4).

2. Use an $(n^I, k^I; n^F, \{k_i^F\}_{i=1}^{\hat{\lambda}^F})$ conversion procedure to (generalized) split the $(\lambda^I \bmod \lambda^F)$ remaining initial codewords to obtain $\hat{\lambda}^F$ intermediate codewords, where $\hat{\lambda}^F = \lceil k^I/(k^F \bmod k^I) \rceil$, $k_i^F = (k^F \bmod k^I)$ for $i \in [\hat{\lambda}^F - 1]$, and $k_{\hat{\lambda}^F}^F = (k^F \bmod k^I)$ if $(k^F \bmod k^I) \mid k^I$ and $k_{\hat{\lambda}^F}^F = (k^I \bmod (k^F \bmod k^I))$ otherwise. Each intermediate codeword is assigned to a different group.

3. The conversion procedure for generalized merge is used to turn each codeword group into a single final codeword.

The total number of symbols read during conversion is:

$$\lambda^I r^F + (\lambda^I \bmod \lambda^F)(k^I - \max\{k^F \bmod k^I, r^F\}),$$

which matches Theorem 2.25.

Case $k^I > k^F$: Conversion occurs in two steps (see Figure 2.5):

1. First, use an $(n^I, k^I; n^F, \{k_i^F\}_{i=1}^{\hat{\lambda}^F})$ conversion procedure to (generalized) split each initial codeword, where $\hat{\lambda}^F = (\lceil k^I/k^F \rceil)$, $k_i^F = k^F$ for $i \in [\hat{\lambda}^F - 1]$ (corresponding to final codewords), and $k_{\hat{\lambda}^F}^F = k^F$ if $k^F \mid k^I$ (corresponding to another final codeword) and $k_{\hat{\lambda}^F}^F = (k^F \bmod k^I)$ otherwise (corresponding to an intermediate codeword).

2. Assemble the $\lambda^I(k^F \bmod k^I)$ remaining symbols from the intermediate codewords into $(\lambda^F \bmod \lambda^I)$ final codewords. This is done using the default approach, since all the remaining symbols would have been already accessed in the first step.

The total number of symbols read in this case during conversion is $\lambda^I(r^F + k^I - k^F)$, which matches Theorem 2.25.

Therefore, the total access cost of conversion when $r^I \geq r^F$ and $k^I \neq k^F$ is $(\lambda^I + \lambda^F)r^F + (\lambda^I \bmod \lambda^F)(k^I - \max\{k^F \bmod k^I, r^F\})$, while the access cost of the default approach is $\lambda^F n^F$.

Access-optimal construction

Since the conversion procedure in Section 2.5.3 is based on the generalized split and merge regimes, we only need to ensure that the constructed codes can perform those conversions with optimal access cost.

Theorem 2.26. *For all $k^F \leq k^I$, every systematic linear MDS $[n^I, k^I]$ code $\mathcal{C}^I$ is (n^F, k^F)-optimally convertible. For all $k^F \leq \lambda^F k^I$ with integer $\lambda^F > 2$, every access-optimal systematic linear MDS $(n^I, k^I; n^F, k^F = \lambda^F k^I)$ convertible code is (n^F, k^F)-optimally convertible.*

Proof. Recall, from Section 2.5.1 that any systematic $[n^I, k^I]$ code $\mathcal{C}^I$ can be used as the initial code of an access-optimal convertible code in the generalized split regime (i.e., an $(n^I, k^I = \sum_{i=1}^{\lambda^F} k_i^F; n^F, \{k_i^F\}_{i=1}^{\lambda^F})$ convertible code). Since the conversion procedure for the general regime in the case where $k^I > k^F$ only uses conversions from the generalized split regime and conversions from the generalized merge regime that can be carried out using the default approach, it is clear that any systematic code $\mathcal{C}^I$ can be used. Similarly, from Section 2.5.1 we know that any $[n^I, k^I]$ code $\mathcal{C}^I$ that is $(n^F, \lambda^F k^I)$-optimally convertible for an integer $\lambda^F \geq 2$ can achieve conversion with optimal access cost in a $(n^I, \{k_i^I\}_{i=1}^{\lambda^I}; n^F, k^F = \sum_{i=1}^{\lambda^I} k_i^I)$ convertible code, where $\lambda^I \leq \lambda^F$. Since the conversion procedure for the general regime in the case where $k^I < k^F$ only uses conversions from the generalized split and merge regimes, it is clear that any $(n^F, \lambda^F k^I)$-optimally convertible code $\mathcal{C}^I$ such that $\lambda^F \geq \lceil k^F/k^I \rceil$ can be used. □

Therefore, the constructions for the merge regime presented in [96] can be used to construct access-optimal convertible codes in the general regime.

Chapter 3

Bandwidth-cost of convertible codes: fundamental limits and optimal constructions

This chapter is based on work from [101, 102], done in collaboration with K. V. Rashmi.

In the preceding chapter, we measured cost in terms of the *access cost* of conversion, which corresponds to the number of codeword symbols accessed during conversion. Another important resource overhead incurred during conversion is that on the network bandwidth, which we call *conversion bandwidth*. In the system, this corresponds to the total amount of data transferred between nodes during conversion. Figure 3.1 depicts the conversion process from an $[n^I, k^I]$ initial code to an $[n^F, k^F]$ final code: the total amount of data read from the nodes γ_R corresponds to the read conversion bandwidth, and the total amount of data written back to the nodes γ_W corresponds to the write conversion bandwidth. Access-optimal convertible codes, by virtue of reducing the number of code symbols accessed, also reduce conversion bandwidth as compared to the default approach. However, it is not clear, a priori, whether these codes are also optimal with respect to conversion bandwidth.

In this chapter, we study the conversion bandwidth of code conversions. As in the previous chapter, we will focus on MDS codes, and study the merge and split regimes.

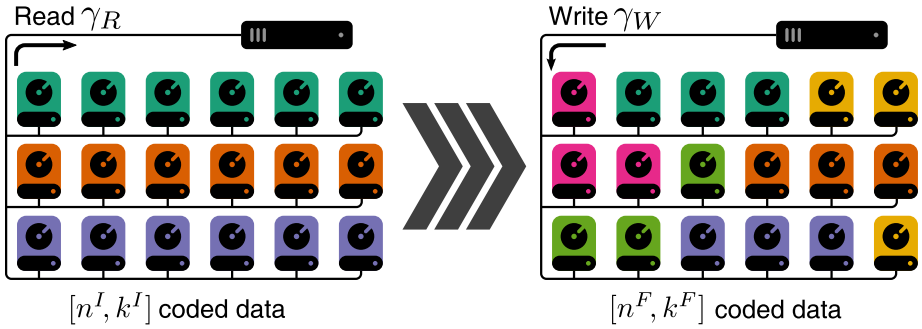


Figure 3.1: Conversion process of codewords of an $[n^I, k^I]$ initial code into codewords of an $[n^F, k^F]$ final code. In this figure, each color represents a different codeword. Code conversion is performed by downloading data from storage nodes to a central location, processing the data, and writing back the processed data to the nodes. The total amount of data read is denoted by γ_R, and the total amount of data written is denoted by γ_W.

For each of these regimes, we derive lower bounds on conversion bandwidth, and propose constructions that are more efficient than the default approach and access-optimal codes in terms of conversion bandwidth. Similar to access cost, conversion bandwidth costs behave differently depending on whether $r^I < r^F$ or $r^I \geq r^F$. However, in both cases it is possible to reduce conversion bandwidth compared to the default approach (unlike access cost, which could only be reduced when $r^I \geq r^F$).

To achieve these reductions in conversion bandwidth, it is necessary to use *vector codes*, where each symbol is a vector of α subsymbols. This is unlike the case of access cost, where scalar codes (each symbol is a scalar and $\alpha = 1$) were sufficient for constructing optimal codes.

In Table 3.1, we summarize the results shown in this chapter.

3.1 Additional background

In this section we introduce some additional concepts from the literature which are used in this chapter. We then do an overview of other related work.

Table 3.1: Comparison of the read conversion bandwidth of different approaches to the merge regime and split regime. In all cases, the write conversion bandwidth is $\lambda^F r^F \alpha$. We assume that $r^F \leq \min\{k^I, k^F\}$; when this condition does not hold, the default approach is bandwidth-optimal.

Merge regime

Approach	Read bandwidth ($r^I < r^F$)	Read bandwidth ($r^I \geq r^F$)
Default	$\lambda^I k^I \alpha$	$\lambda^I k^I \alpha$
Access optimal [96]	$\lambda^I k^I \alpha$	$\lambda^I r^F \alpha$
Bandwidth optimal	$\lambda^I k^I \alpha - \lambda^I r^I \alpha \left(\frac{k^I}{r^F} - 1\right)$	$\lambda^I r^F \alpha$

Split regime

Approach	Read bandwidth ($r^I < r^F$)	Read bandwidth ($r^I \geq r^F$)
Default	$\lambda^F k^F \alpha$	$\lambda^F k^F \alpha$
Access optimal [99]	$\lambda^F k^F \alpha$	$[(\lambda^F - 1)k^F + r^F]\alpha$
Bandwidth optimal	$\lambda^F k^F \alpha - r^I \alpha \left(\frac{k^F}{r^F} - 1\right)$	$\lambda^F r^F \alpha \frac{(\lambda^F - 1)k^F + r^I}{(\lambda^F - 1)r^F + r^I}$

3.1.1 Vector codes and puncturing

In this section we introduce the basic notation for vector codes, and generalize some definitions to the case of vector codes. Let $[i]$ denote the subset $\{1, 2, \ldots, i\}$, for a natural number i. An $[n, k, \alpha]$ vector code $\mathcal{C}$ over a finite field $\mathbb{F}_q$ is an injective mapping $\mathcal{C} : \mathbb{F}_q^{\alpha k} \to \mathbb{F}_q^{\alpha n}$. For a given codeword $\mathbf{c} = \mathcal{C}(\mathbf{m})$ and $i \in [n]$, define $\mathbf{c}_i = \mathcal{C}_i(\mathbf{m}) = (c_{\alpha(i-1)+1}, \ldots, c_{\alpha i})$ as the i-th *symbol* of $\mathbf{c}$, which is a vector of length α over $\mathbb{F}_q$. We refer to elements from the base field $\mathbb{F}_q$ as *subsymbols*. A code is said to be *systematic* if it always maps $\mathbf{m}$ to a codeword that contains all the subsymbols of $\mathbf{m}$ uncoded. In a *linear* $[n, k, \alpha]$ vector code $\mathcal{C}$, the encoding of message $\mathbf{m} \in \mathbb{F}_q^{k\alpha}$ is given by the mapping $\mathbf{m} \mapsto \mathbf{mG}$ where $\mathbf{G} \in \mathbb{F}_q^{k\alpha \times n\alpha}$ is called the *generator matrix* of $\mathcal{C}$, and the columns of $\mathbf{G}$ are called *encoding vectors*. The *minimum distance* of a vector code is defined as:

$$\text{dist}(\mathcal{C}) := \min_{\mathbf{m} \neq \mathbf{m}'} |\{i \in [n] : \mathcal{C}_i(\mathbf{m}) \neq \mathcal{C}_i(\mathbf{m}')\}| .$$

An $[n, k, \alpha]$ vector code $\mathcal{C}$ is said to be *maximum-distance-separable* (MDS) if $\text{dist}(\mathcal{C}) = n - k + 1$ (i.e., it achieves the Singleton bound [97]). MDS codes are commonly used in practice because they achieve the optimal tradeoff between storage overhead and failure tolerance.

A *scalar code* is a vector code with $\alpha = 1$. We will omit the parameter α when it is clear from context or when $\alpha = 1$. A *puncturing* of a vector code $\mathcal{C}$ is the resulting vector code after removing a fixed subset of symbols from every codeword.

3.1.2 Convertible codes [96, 99]

We recall a few definitions and results on access-optimal convertible codes from Chapter 2.

The access cost of a conversion procedure is the sum of the *read access cost*, i.e. the total number of code symbols read, and the *write access cost*, i.e. the total number of code symbols written. An *access-optimal convertible code* is a convertible code whose conversion procedure has the minimum access cost over all convertible

Symbol 1	$f_1(\mathbf{m}_1)$	$f_1(\mathbf{m}_2)$	$\cdots$	$f_1(\mathbf{m}_\alpha)$
Symbol 2	$f_2(\mathbf{m}_1)$	$f_2(\mathbf{m}_2)$	$\cdots$	$f_2(\mathbf{m}_\alpha)$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
Symbol n	$f_n(\mathbf{m}_1)$	$f_n(\mathbf{m}_2)$	$\cdots$	$f_n(\mathbf{m}_\alpha)$

(a) α instances of the base code

$f_1(\mathbf{m}_1)$	$f_1(\mathbf{m}_1) + g_{2,1}(\mathbf{m}_2)$	$\cdots$	$f_1(\mathbf{m}_\alpha) + g_{\alpha,1}(\mathbf{m}_1, \ldots, \mathbf{m}_\alpha)$
$f_2(\mathbf{m}_1)$	$f_2(\mathbf{m}_1) + g_{2,2}(\mathbf{m}_2)$	$\cdots$	$f_2(\mathbf{m}_\alpha) + g_{\alpha,2}(\mathbf{m}_1, \ldots, \mathbf{m}_\alpha)$
$\vdots$	$\vdots$	$\ddots$	$\vdots$
$f_n(\mathbf{m}_1)$	$f_n(\mathbf{m}_1) + g_{2,n}(\mathbf{m}_2)$	$\cdots$	$f_1(\mathbf{m}_\alpha) + g_{\alpha,n}(\mathbf{m}_1, \ldots, \mathbf{m}_\alpha)$

(b) Piggybacked code

Figure 3.2: Piggybacking framework [63] for constructing vector codes.

codes with given parameters $(n^I, k^I; n^F, k^F)$. Similarly, an $[n^I, k^I]$ code is said to be (n^F, k^F)-access-optimally convertible if it is the initial code of an access-optimal $(n^I, k^I; n^F, k^F)$ convertible code.

In practice, the values of n^F and k^F for the conversion might be unknown. Thus, constructing convertible codes which are simultaneously (n^F, k^F)-access-optimally convertible for several possible values of n^F and k^F is also important (as will be discussed in Section 3.4.2).

Though the definition of convertible codes allows for any kind of initial and final codes, this chapter focuses on MDS codes. A convertible code is said to be MDS when both $\mathcal{C}^I$ and $\mathcal{C}^F$ are MDS. The access cost lower bound for linear MDS convertible codes is summarized in Table 2.1. There are explicit constructions of access-optimal convertible codes for all valid parameters $(n^I, k^I; n^F, k^F)$ (described in Chapter 2). Notice that for the increasing-redundancy region $(r^I < r^F)$, read access cost is always M, which is the same as the default approach. In the decreasing-redundancy region $(r^I \geq r^F)$, on the other hand, one can achieve lower access cost than the default approach when $r^F < \min\{k^I, k^F\}$.

During conversion, code symbols from the initial codewords can play multiple roles: they can become part of different final codewords, their contents might be read or written, additional code symbols may be added and existing code symbols may be removed. Based on their role, code symbols can be divided into three groups: (1) *unchanged symbols*, which are present both in the initial and final codewords without any modifications; (2) *retired symbols*, which are only present in the initial codewords

but not in the final codewords; and (3) *new symbols*, which are present only in the final codewords but not in the initial codewords. Both unchanged and retired symbols may be read during conversion, and then linear combinations of data read are written into the new symbols.

The *merge regime* (Section 1.1) is a fundamental regime of convertible codes which corresponds to conversions which merge multiple initial codewords into a single final codeword. Thus, convertible codes in the merge regime are such that $k^F = \lambda^I k^I$ for some integer $\lambda^I \geq 2$, and $\lambda^F = 1$. We recall two lemmas from the previous chapter which will be useful for analyzing the merge regime in this chapter. First, Lemma 2.2, which notes that all data gets mapped to the same final stripe. Thus, the initial and final partition do not play an important role in the merge regime. And second, Lemma 2.3, which states that there can be at most k^I unchanged symbols in each initial codeword. This is because having more than k^I unchanged symbols in an initial codeword would contradict the MDS property. Recall that codes which have the maximum number of unchanged symbols are called *stable* (Definition 1.5).

Access-optimal convertible code for merge regime. When $r^I < r^F$, the default approach has optimal access cost, and so constructing an access-optimal code for this case is trivial. When $r^I \geq r^F$ and the code is in the merge regime, only r^F code symbols from each initial codeword need to be read. These symbols are then used to compute r^F new code symbols.

In Chapter 2, several constructions for access-optimal convertible codes in the merge regime were presented. Codes built using these constructions are (1) systematic, (2) linear, (3) during conversion only access r^F parities from each initial stripe, and (4) when constructed with a given value of $\lambda^I = \lambda$ and $r^F = r$, the initial $[n^I, k^I]$ code is (n^F, k^F)-access-optimally convertible for all $k^F = \lambda' k^I$ and $n^F = k^F + r'$ such that $1 \leq \lambda' \leq \lambda$ and $1 \leq r' \leq r$. In Section 3.4 we use an access-optimal convertible code in the merge regime as part of our construction of bandwidth-optimal convertible codes for the merge regime. Next, we give a brief summary of the general construction presented in Section 2.2.

Consider the case where $r^I \geq r^F$ and $r^F < k^I$ (otherwise, the construction is trivial). The codes $\mathcal{C}^I$ and $\mathcal{C}^F$ over finite field $\mathbb{F}_q$ are defined via the matrices

$\mathbf{G}^I = \left[\mathbf{I}_{k^I} \mid \mathbf{P}^I\right]$ and $\mathbf{G}^F = \left[\mathbf{I}_{k^F} \mid \mathbf{P}^F\right]$ where:

- $\mathbf{I}_k$ is the $k \times k$ identity matrix,
- $\alpha_1, \alpha_2, \ldots, \alpha_{r^I}$ are distinct elements from $\mathbb{F}_q$,
- $\mathbf{P}^I$ is the $k^I \times r^I$ Vandermonde matrix with evaluation points $(\alpha_1, \ldots, \alpha_{r^I})$,
- $\mathbf{P}^F$ is the $k^F \times r^F$ Vandermonde matrix with evaluation points $(\alpha_1, \ldots, \alpha_{r^F})$.

(In Section 2.2, α_i was chosen as θ^{i-1} for some primitive element $\theta \in \mathbb{F}_q$.) One important aspect of this construction is that, due to the nature of Vandermonde matrices, the i-th column of $\mathbf{P}^F$ is equal to the vertical concatenation of the respective i-th columns of $\mathbf{P}^I, \alpha_i^{k^I}\mathbf{P}^I, \ldots, \alpha_i^{(\lambda^I-1)k^I}\mathbf{P}^I$. This property ensures that each final parity can be constructed during conversion as a linear combination of one initial parity from each initial codeword. As shown in Chapter 2, this construction satisfies the properties (1–4) described above, and is MDS for appropriately chosen points α_i $(i \in [r^I])$ and sufficiently large $\mathbb{F}_q$.

Example 3.1 (Access-optimal code)**:** Consider the parameters $(n^I = 7, k^I = 4; n^F = 11, k^F = 8)$ over $\mathbb{F}_{17}$: the evaluation points $(\alpha_1 = 1, \alpha_2 = 2, \alpha_3 = 6)$ yield an MDS access-optimal code. It is easy to check that the codes defined by the following matrices are MDS:

$$\mathbf{P}^I = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 6 \\ 1 & 4 & 2 \\ 1 & 8 & 12 \end{bmatrix} \qquad \mathbf{P}^F = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 6 \\ 1 & 4 & 2 \\ 1 & 8 & 12 \\ 1 & 16 & 4 \\ 1 & 15 & 7 \\ 1 & 13 & 8 \\ 1 & 9 & 14 \end{bmatrix}$$

Now, suppose the data $(a_1, \ldots, a_4)$ and $(a_5, \ldots, a_8)$ are encoded with the initial code.

It is easy to check that the following holds:

$$(a_1, \ldots, a_4)\mathbf{P}^I + (a_5, \ldots, a_8)\mathbf{P}^I \begin{bmatrix} 1 & 0 & 0 \\ 0 & 16 & 0 \\ 0 & 0 & 4 \end{bmatrix} = (a_1, \ldots, a_8)\mathbf{P}^F.$$

▸

3.1.3 Network information flow

Network information flow [103] is a class of problems that model the transmission of information from sources to sinks in a point-to-point communication network. *Network coding* [104–108] is a generalization of store-and-forward routing, where each node in the network is allowed to combine its inputs using a code before communicating messages to other nodes. For the purposes of this chapter, an *information flow graph* is a directed acyclic graph $G = (V, E)$, where V is the set of nodes, $E \subseteq V \times V \times \mathbb{R}_{\geq 0}$ is the set of edges with non-negative capacities, and $(i, j, c) \in E$ represents that information can be sent noiselessly from node i to node j at rate c. Let $X_1, X_2, \ldots, X_m$ be mutually independent information sources with rates $x_1, x_2, \ldots, x_m$ respectively. Each information source X_i is associated with a source $s_i \in V$, where it is generated, and a sink $t_i \in V$, where it is required. In this chapter we mainly make use of the *information max-flow bound* [109] which indicates that it is impossible to transmit X_i at a higher rate than the maximum flow from s_i to t_i. In other words, $x_i \leq \text{max-flow}(s_i, t_i)$ for all $i \in [m]$ is a necessary condition for a network coding scheme satisfying all constraints to exist. In our analysis, we will consider s_i-t_i-cuts of the information flow graph, which give an upper bound on max-flow(s_i, t_i) and thus an upper bound on x_i as well. We will also utilize the fact that two independent information sources with the same source and sink can be considered as a single information source with rate equal to the sum of their rates.

In [27], information flow and network coding is applied to the *repair problem* in distributed storage systems. The repair problem is the problem of reconstructing a small number of failed code symbols in an erasure code (without having to decode

the full codeword). Dimakis et al. [27] use information flow to establish bounds on the storage size and repair network-bandwidth of erasure codes. In this work we use information flow to model the process of code conversion and establish lower bounds on the total amount of network bandwidth used during conversion.

3.1.4 Piggybacking framework for constructing vector codes

The *Piggybacking framework* [63, 110] is a framework for constructing new vector codes building on top of existing codes. The main technique behind the Piggybacking framework is to take an existing code as a *base code*, create a new vector code consisting of multiple instances of the base code (as described below), and then add carefully designed functions of the data (called *piggybacks*) from one instance to the others. These piggybacks are added in a way such that it retains the decodability properties of the base code (such as the MDS property). The piggyback functions are chosen to confer additional desired properties to the resulting code. In [63], the authors showcase the Piggybacking framework by constructing codes that are efficient in reducing bandwidth consumed in repairing codeword symbols.

More specifically, the Piggybacking framework works as follows. Consider a length n code defined by the function $f(\mathbf{m}) = (f_1(\mathbf{m}), f_2(\mathbf{m}), \ldots, f_n(\mathbf{m}))$. Now, consider α instances of this base code, each corresponding to a coordinate of the α-length vector of each symbol in the new vector code. Let $(\mathbf{m}_1, \mathbf{m}_2, \ldots, \mathbf{m}_\alpha)$ denote the independent messages encoded under these α instances, as shown in Figure 3.2a. For every i such that $2 \leq i \leq \alpha$, one can add to the data encoded in instance i an arbitrary function of the data encoded by instances $\{1, \ldots, (i-1)\}$. Such functions are called *piggyback functions*, and the piggyback function corresponding to code symbol $j \in [n]$ of instance $i \in \{2, \ldots, \alpha\}$ is denoted as $g_{i,j}$.

The decoding of the piggybacked code proceeds as follows. Observe that instance 1 does not have any piggybacks. First, instance 1 of the base code is decoded using the base code's decoding procedure in order to obtain $\mathbf{m}_1$. Then, $\mathbf{m}_1$ is used to compute and subtract any of the piggybacks $\{g_{2,i}(\mathbf{m}_1)\}_{i=1}^{n}$ from instance 2 and the base code's decoding can then be used to recover $\mathbf{m}_2$. Decoding proceeds like this, using the

data decoded from previous instances in order to remove the piggybacks until all instances have been decoded. It is clear that if an $[n, k, \alpha]$ vector code is constructed from an $[n, k]$ MDS code as the base code using the Piggybacking framework, then the resulting vector code is also MDS. This is because any set of k symbols from the vector code contains a set of k subsymbols from each of the α instances.

In this chapter, we use the Piggybacking framework to design a code where piggybacks store data which helps in making the conversion process efficient.

3.2 Modeling conversion for conversion bandwidth optimization

In this section, we model the conversion process as an information flow problem. We utilize this model primarily for deriving lower bounds on the total amount of information that needs to be transferred during conversion. Since our focus is on modeling the conversion process, we consider a single value for each of the final parameters n^F and k^F. This model continues to be valid for each individual conversion, even when the final parameters might take multiple values.

In the previous chapter (Chapter 2) we considered only *scalar codes*, where each code symbol corresponds to a scalar from a finite field $\mathbb{F}_q$. Considering scalar codes is sufficient when optimizing for access cost, which was the focus in that chapter, since the access cost is measured at the granularity of code symbols. However, when optimizing conversion bandwidth, vector codes can perform better than scalar codes since they allow partial download from a node. This allows conversion procedures to only download a fraction of a code symbol and thus only incur the conversion bandwidth associated with the size of that fraction. This can potentially lead to significant reduction in the total conversion bandwidth. For this reason, we consider the initial code $\mathcal{C}^I$ as an $[n^I, k^I, \alpha]$ MDS code and the final code $\mathcal{C}^F$ as an $[n^F, k^F, \alpha]$ MDS code, where $\alpha \geq 1$ is considered as a free parameter chosen to minimize conversion bandwidth. This move to vector codes is inspired by the work of Dimakis et al. [27] on regenerating codes, who showed the benefit of vector codes in reducing

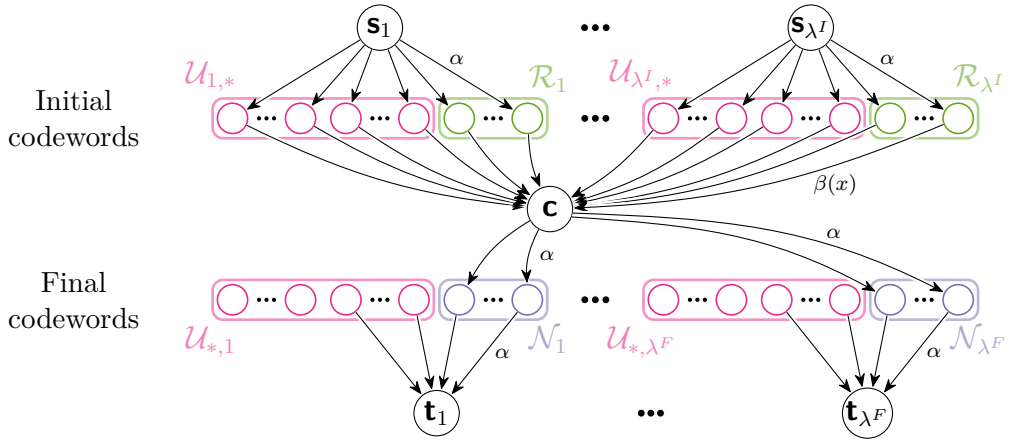


Figure 3.3: Information flow graph of conversion in the general case. Unchanged, retired, and new nodes are shown in different colors. Notice that each unchanged node in this figure is drawn twice: once in the initial codewords and once in the final codewords. These correspond to exactly the same node, but are drawn twice for clarity. Some representative edges are labeled with their capacities.

network bandwidth in the context of the repair problem. For MDS convertible codes, message size will be $B = M\alpha = \text{lcm}(k^I, k^F)\alpha$, which we interpret as a vector $\mathbf{m} \in \mathbb{F}_q^{M\alpha}$ composed of M symbols made up of α subsymbols each. We will denote the number of subsymbols downloaded from node s during conversion as $\beta(s) \leq \alpha$ and extend this notation to sets of nodes as $\beta(\mathcal{S}) = \sum_{s\in\mathcal{S}} \beta(s)$.

Consider an $(n^I, k^I; n^F, k^F)$ MDS convertible code with initial partition $\mathcal{P}_I = \{P_1^I, \ldots, P_{\lambda^I}^I\}$ and final partition $\mathcal{P}_F = \{P_1^F, \ldots, P_{\lambda^F}^F\}$. We model conversion using an information flow graph as the one shown in Figure 3.3 where message symbols are generated at source nodes, and sinks represent the decoding constraints of the final code. Symbols of message $\mathbf{m}$ are modeled as information sources $X_1, X_2, \ldots, X_M$ of rate α (over $\mathbb{F}_q$) each. For each initial codeword $i \in [\lambda^I]$, we include one source node s_i, where the information sources corresponding to the message symbols in P_i^I are generated. Each code symbol of initial codeword i is modeled as a node with an incoming edge from s_i. A *coordinator node* c models the central location where the contents of new symbols are computed, and it has incoming edges from all nodes in the initial codewords. During conversion, some of the initial code symbols will

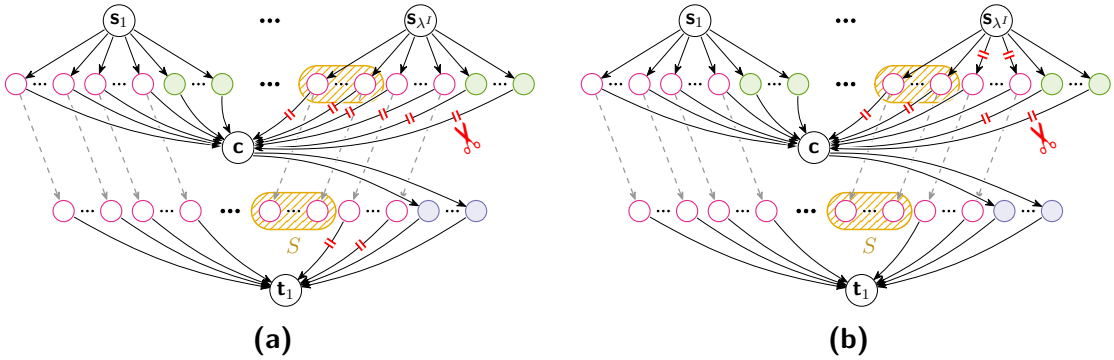


Figure 3.4: Information flow graph of conversion in the merge regime with two different cuts (used in proofs). For clarity, each unchanged node is drawn twice: once in the initial codewords and once in the final codeword. These two instances are connected by a dashed arrow. Marked edges denote a graph cut.

remain unchanged, some will be retired, and some new code symbols will be added. Thus, we also include the nodes corresponding to unchanged symbols in the final codewords (that is, every unchanged node is shown twice in Figure 3.3). Note that the unchanged nodes in the initial codewords and the unchanged nodes in the final codewords are identical, and thus do not add any conversion bandwidth. For each new symbol we add a node that connects to the coordinator node. From this point, we will refer to code symbols and their corresponding nodes interchangeably. For each final codeword $j \in [\lambda^F]$, we add a sink t_j which connects to some subset of nodes from final codeword j, and recovers the information sources corresponding to the message symbols in P_j^F.

Thus, the information flow graph for a convertible code comprises the following nodes:

- unchanged nodes $\mathcal{U}_{i,j} = \{u_{i,j,1}, \ldots, u_{i,j,|\mathcal{U}_{i,j}|}\}$ for all $i \in [\lambda^I],\ j \in [\lambda^F]$, which are present both in the initial and final codewords;
- retired nodes $\mathcal{R}_i = \{v_{i,1}, \ldots, v_{i,|\mathcal{R}_i|}\}$ for $i \in [\lambda^I]$, which are only present in the initial codewords;
- new nodes $\mathcal{N}_j = \{w_{j,1}, \ldots, w_{j,|\mathcal{N}_j|}\}$ for $j \in [\lambda^F]$, which are only present in the

final codewords;

- source nodes s_i for $i \in [\lambda^I]$, representing the data to be encoded;
- sink nodes t_j for $j \in [\lambda^F]$, representing the data decoded; and
- a coordinator node c.

In the information flow graph, information source X_l is generated at node s_i if and only if $l \in P_i^I$, and recovered at node t_l if and only if $l \in P_j^F$.

Throughout this chapter, we use the disjoint union symbol $\sqcup$ when appropriate to emphasize that the two sets in the union are disjoint. To simplify the notation, when $*$ is used as an index, it denotes the disjoint union of the indexed set over the range of that index, e.g. $\mathcal{U}_{*,j} = \bigsqcup_{i=1}^{\lambda^I} \mathcal{U}_{i,j}$.

The information flow graph must be such that the following conditions hold: (1) the number of nodes per initial codeword is n^I, i.e., $|\mathcal{U}_{i,*}| + |\mathcal{R}_i| = n^I$ for all $i \in [\lambda^I]$; and (2) the number of nodes per final codeword is n^F, i.e., $|\mathcal{U}_{*,j}| + |\mathcal{N}_j| = n^F$ for all $j \in [\lambda^F]$. Additionally, the information flow graph contains the following set of edges E, where a directed edge from node u to v with capacity δ is represented with the triple (u, v, δ):

- $\{(s_i, x, \alpha) : x \in \mathcal{U}_{i,*} \sqcup \mathcal{R}_i\} \subset E$ for each $i \in [\lambda^I]$, where the capacity corresponds to the size of the data stored on each node;
- $\{(x, c, \beta(x)) : x \in \mathcal{U}_{i,*} \sqcup \mathcal{R}_i\} \subset E$ for each $i \in [\lambda^I]$, where the capacity corresponds to the amount of data downloaded from node x;
- $\{(c, y, \alpha) : y \in \mathcal{N}_j\} \subset E$ for each $j \in [\lambda^F]$, where the capacity corresponds to the size of the data stored on each new node;
- $\{(y, t_j, \alpha) : y \in V_j\} \subset E$ for $V_j \subseteq \mathcal{U}_{*,j} \sqcup \mathcal{N}_j$ such that $|V_j| = k^F$, for all $j \in [\lambda^F]$, where the capacity corresponds to the size of the data stored on each node.

The sinks t_j represent the decoding constraints of the final code, and each choice of set V_j will represent a different choice of k^F code symbols for decoding the final

codeword. A necessary condition for a conversion procedure is to satisfy all sinks t_j for all possible $V_1, \ldots, V_{\lambda^F}$. The sets $\mathcal{U}_{i,j}, \mathcal{R}_i, \mathcal{N}_j$ and the capacities $\beta(x)$ are determined by the conversion procedure of the convertible code. Figure 3.3 shows the information flow graph of an arbitrary convertible code.

Definition 3.1 (Conversion bandwidth)**:** The *read conversion bandwidth* γ_R is the total amount of data transferred from the initial nodes to the coordinator node c. The *write conversion bandwidth* γ_W is the total amount of data transferred from the coordinator node c to the new nodes. The (total) *conversion bandwidth* γ is the sum of the read conversion bandwidth and the write conversion bandwidth. Formally:

$$\gamma_R := \beta(\mathcal{U}_{*,*} \sqcup \mathcal{R}_*)\,, \qquad \gamma_W := |\mathcal{N}_*|\alpha, \qquad \gamma := \gamma_R + \gamma_W. \tag{3.1}$$

▸

Once the structure of the graph is set and fixed, information flow analysis gives lower bounds on the capacities $\beta(x)$. Therefore, a part of our objective in designing convertible codes is to set $\mathcal{U}_{i,j}, \mathcal{R}_i, \mathcal{N}_j$ so as to minimize the lower bound on γ.

Notice that the conversion process, as defined above, is *not* a single-source multicast problem; therefore, the information max-flow bound is not guaranteed to be achievable. Nonetheless, information flow can be applied to obtain a lower bound (Section 3.3), which we show is achievable by providing a construction (Section 3.4).

Remark 3.1: In practice, conversion bandwidth can sometimes be further reduced by placing the coordinator node along with a new node and/or a retired node in the same server. One can even first split the coordinator node into several coordinator nodes, each processing data which is not used in conjunction with data processed by other coordinator nodes, and then place them in the same server as a new node and/or a retired node. Such "optimizations" do not fundamentally alter our result, and hence are left out in order to make the exposition clear. ▸

3.3 Optimizing conversion bandwidth in the merge regime

In this section, we use the information flow model presented in Section 3.2 to derive a lower bound on the conversion bandwidth for MDS codes in the merge regime. Recall from Section 3.1.2, that convertible codes in the merge regime are those where $k^F = \lambda^I k^I$ for some integer $\lambda^I \geq 2$, i.e., this regime corresponds to conversions where multiple initial codewords are merged into a single final codeword. As in the previous section, our analysis focuses on a single conversion, and thus a single value for the final parameters n^F and k^F. However, our analysis only depends on the conversion process itself; therefore, the bound on the bandwidth still applies even if we consider multiple conversions.

Consider an $(n^I, k^I; n^F, \lambda^I k^I)$ convertible code in the merge regime, for some integer $\lambda^I \geq 2$. Note that for all convertible codes in the merge regime, it holds that the number of final codewords is $\lambda^F = 1$. Since all initial and final partitions $(\mathcal{P}_I, \mathcal{P}_F)$ are equivalent up to relabeling in this regime (by Lemma 2.2), we can omit them from our analysis. Note also that all information sources are recovered at the same sink node, t_1. Thus, we may treat each source node s_i as having a single information source X_i of rate αk^I $(i \in [\lambda^I])$. For each source node and each sink node pair, we can invoke the information max-flow bound (Section 3.1.3) to derive an inequality. For conversion to be possible, the variable-capacity edges must take on values such that all these inequalities are simultaneously satisfied. Figure 3.4a shows the information flow graph for a convertible code in the merge regime.

First, we derive a general lower bound on conversion bandwidth in the merge regime by considering a simple cut in the information flow graph. Intuitively, this lower bound emerges from the fact that new nodes need to have a certain amount of information from each initial codeword in order to fulfill the MDS property of the final code. This lower bound depends on the number of unchanged nodes and achieves its minimum when the number of unchanged nodes is maximized. Recall from Section 3.1.2 that convertible codes with maximum number of unchanged nodes

are called *stable* convertible codes. Thus, the derived lower bound is minimized for stable convertible codes.

Lemma 3.2. *Consider an MDS* $(n^I, k^I; n^F, \lambda^I k^I)$ *convertible code. Then:*

$$\gamma_R \geq \lambda^I \alpha \min\{r^F, k^I\} \quad \textit{and} \quad \gamma_W \geq r^F \alpha,$$

where equality is only possible for stable codes.

Proof. We prove this inequality via an information flow argument. Let $i \in [\lambda^I]$ and consider the information source generated at source s_i. Let $S \subseteq \mathcal{U}_{i,1}$ be a subset of unchanged nodes from initial codeword i of size $\tilde{r}_i = \min\{r^F, |\mathcal{U}_{i,1}|\}$. Consider a sink t_1 that connects to nodes $\mathcal{U}_{*,1} \setminus S$. We choose the graph cut defined by nodes $\{s_i\} \sqcup \mathcal{U}_{i,1} \sqcup \mathcal{R}_i$ (see Figure 3.4a, which depicts the cut for $i = \lambda^I$). This cut yields the following inequality:

$$k^I \alpha \leq \max\{|\mathcal{U}_{i,1}| - r^F, 0\}\alpha + \beta(\mathcal{U}_{i,1} \sqcup \mathcal{R}_i)$$

$$\iff \beta(\mathcal{U}_{i,1} \sqcup \mathcal{R}_i) \geq (k^I + r^F - \max\{|\mathcal{U}_{i,1}|, r^F\})\alpha$$

This inequality must hold for every $i \in [\lambda^I]$ simultaneously; otherwise, it would be impossible for the sink to recover the full data. By summing this inequality over all sources $i \in [\lambda^I]$ and using the definition of γ (Equation (3.1)), we obtain:

$$\gamma \geq \sum_{i=1}^{\lambda^I} (k^I + r^F - \max\{|\mathcal{U}_{i,1}|, r^F\})\alpha + |\mathcal{N}_1|\alpha$$

By Lemma 2.3, $|\mathcal{U}_{i,1}| \leq k^I$. Therefore, it is clear that the right hand side achieves its minimum if and only if $|\mathcal{U}_{i,1}| = k^I$ for all $i \in [\lambda^I]$ (i.e. the code is stable). Lemma 2.3 also implies that $\gamma_W \geq r^F \alpha$, proving the lemma. □

Remark 3.3: Note that the conversion bandwidth lower bound described in Lemma 3.2 coincides with the access-cost lower bound described in Table 2.1 when $r^I \geq r^F$. This follows by recalling that each node corresponds to an α-length vector, and for scalar codes $\alpha = 1$. ▸

In particular, this implies that convertible codes in the merge regime which are access-optimal and have $r^I \geq r^F$ are also bandwidth-optimal (i.e. those in the decreasing-redundancy region). However, as we will show next, this property fails to hold when $r^I < r^F$ (that is, increasing-redundancy region).

We next derive a lower bound on conversion bandwidth which is tighter than Lemma 3.2 when $r^I < r^F$. Nevertheless, it allows for less conversion bandwidth usage than the access-optimal codes.

Intuitively, the data downloaded from retired nodes during conversion will be "more useful" than the data downloaded from unchanged nodes, since unchanged nodes already form part of the final codeword. At the same time, it is better to have the maximum amount of unchanged nodes per initial codeword (k^I) because this minimizes the number of new nodes that need to be constructed. However, this leads to fewer retired nodes per initial codeword (r^I). If the number of retired nodes per initial codeword is less than the number of new nodes ($r^I < r^F$), then conversion procedures are forced to download data from unchanged nodes. This is because one needs to download at least $r^F\alpha$ from each initial codeword (by Lemma 3.2). Since data from unchanged nodes is "less useful", more data needs to be downloaded in order to construct the new nodes.

As in the case of Lemma 3.2, this lower bound depends on the number of unchanged nodes in each initial codeword, and achieves its minimum in the case of stable convertible codes.

Lemma 3.4. *Consider an MDS $(n^I, k^I; n^F, \lambda^I k^I)$ convertible code, with parameters such that $r^I < r^F \leq k^I$. Then $\gamma_R \geq \lambda^I \alpha \left(r^I + k^I \left(1 - \frac{r^I}{r^F}\right)\right)$ and $\gamma_W \geq r^F \alpha$, where equality is only possible for stable codes.*

Proof. We prove this via an information flow argument. Let $i \in [\lambda^I]$ and consider the information source generated at source s_i. Let $S \subseteq \mathcal{U}_{i,1}$ be a subset of size $\tilde{r}_i = \min\{r^F, |\mathcal{U}_{i,1}|\}$. Consider a sink t_1 that connects to the nodes in $\mathcal{U}_{*,1} \setminus S$. Now, we choose a different cut from the one considered in Lemma 3.2, which allows to derive a tighter bound when $r^I < r^F$. We choose the graph cut defined by nodes $\{s_i\} \sqcup S \sqcup \mathcal{R}_i$ (see Figure 3.4b, which depicts the cut when $i = \lambda^I$). This yields the

following inequality:

$$k^I\alpha \leq (|\mathcal{U}_{i,1}| - \tilde{r}_i)\alpha + \beta(S) + \beta(\mathcal{R}_i)\,.$$

This inequality must hold for all possible $S \subseteq \mathcal{U}_{i,1}$ simultaneously; otherwise, there would exist at least one sink incapable of recovering the full data, which violates the MDS property. By rearranging this inequality and summing over all possible choices of subset S, we obtain the following inequality:

$$\binom{|\mathcal{U}_{i,1}|}{\tilde{r}_i}(k^I + \tilde{r}_i - |\mathcal{U}_{i,1}|)\alpha \leq \binom{|\mathcal{U}_{i,1}|-1}{\tilde{r}_i-1}\beta(\mathcal{U}_{i,1}) + \binom{|\mathcal{U}_{i,1}|}{\tilde{r}_i}\beta(\mathcal{R}_i)$$

$$\iff |\mathcal{U}_{i,1}|(k^I + \tilde{r}_i - |\mathcal{U}_{i,1}|)\alpha \leq \tilde{r}_i\beta(\mathcal{U}_{i,1}) + |\mathcal{U}_{i,1}|\beta(\mathcal{R}_i)\,. \tag{3.2}$$

Then, our strategy to obtain a lower bound is to find the minimum value for conversion bandwidth γ which satisfies Inequality (3.2) for all $i \in [\lambda^I]$, which can be formulated as the following optimization problem:

$$\begin{aligned} \text{minimize} \quad & \gamma = \textstyle\sum_{i\in\lambda^I}\left[\beta(\mathcal{U}_{i,1}) + \beta(\mathcal{R}_i)\right] + |\mathcal{N}_1|\alpha \\ \text{subject to} \quad & \text{Inequality (3.2), for all } i \in [\lambda^I] \\ & 0 \leq \beta(x) \leq \alpha, \text{ for all } x \in \mathcal{U}_{*,1} \sqcup \mathcal{R}_*. \end{aligned} \tag{3.3}$$

Intuitively, this linear program shows that it is preferable to download more data from retired nodes ($\beta(\mathcal{R}_i)$) than unchanged nodes ($\beta(\mathcal{U}_{i,1})$), since both have the same impact on γ but the contribution of $\beta(\mathcal{R}_i)$ towards satisfying Inequality (3.2) is greater than or equal to that of $\beta(\mathcal{U}_{i,1})$, because $\tilde{r}_i \leq |\mathcal{U}_{i,1}|$ by definition. Thus to obtain an optimal solution we first set $\beta(\mathcal{R}_i) = \min\{k^I + \tilde{r}_i - |\mathcal{U}_{i,1}|, |\mathcal{R}_i|\}\alpha$ to the maximum needed for all $i \in [\lambda^I]$, and then set:

$$\sum_{x\in\mathcal{U}_{i,1}} \beta(x) = \frac{\max\{\tilde{r}_i - r^I, 0\}|\mathcal{U}_{i,1}|\alpha}{\tilde{r}_i}, \qquad \text{for all } i \in [\lambda^I]$$

to satisfy the constraints. It is straightforward to check that this solution satisfies the KKT (Karush-Kuhn-Tucker) conditions, and thus is an optimal solution to Linear

program 3.3. By replacing these terms back into γ and simplifying we obtain the optimal objective value:

$$\gamma^* = \sum_{i=1}^{\lambda^I} \left[k^I - \min\{r^I, \tilde{r}_i\} \left(\frac{|\mathcal{U}_{i,1}|}{\tilde{r}_i} - 1 \right) \right] \alpha + |\mathcal{N}_1|\alpha$$

It is easy to show that the right hand side achieves its minimum if and only if $|\mathcal{U}_{i,1}| = k^I$ for all $i \in [\lambda^I]$ (i.e., the code is stable). This gives the following lower bound for conversion bandwidth:

$$\gamma \geq \lambda^I \alpha \left(r^I + k^I \left(1 - \frac{r^I}{r^F} \right) \right) + r^F \alpha.$$

Since we must write at least r^F parities, $\gamma_W \geq r^F \alpha$, which proves the lemma. □

By combining Lemmas 3.2 and 3.4 we obtain the following general lower bound on conversion bandwidth of MDS convertible codes in the merge regime.

Theorem 3.5. *For any MDS $(n^I, k^I; n^F, \lambda^I k^I)$ convertible code:*

$$\gamma_R \geq \begin{cases} \lambda^I \alpha \min\{k^I, r^F\}, & \text{if } r^I \geq r^F \text{ or } k^I \leq r^F, \\ \lambda^I \alpha \left(r^I + k^I \left(1 - \frac{r^I}{r^F} \right) \right), & \text{otherwise.} \end{cases}$$
$$\gamma_W \geq r^F \alpha.$$

where equality can only be achieved by stable convertible codes.

Proof. Follows from Lemmas 3.2 and 3.4. □

In Section 3.4, we show that the lower bound of Theorem 3.5 is indeed achievable for all parameter values in the merge regime, and thus it is tight. We will refer to convertible codes that meet this bound with equality as *bandwidth-optimal*.

Remark 3.6: Observe that the model above allows for nonuniform data download during conversion, that is, it allows the amount of data downloaded from each node during conversion to be different. If instead one were to assume uniform download,

i.e. $\beta(x) = \beta(y)$ for all $x, y \in \mathcal{U}_{*,*} \sqcup \mathcal{R}_*$, then a higher lower bound for conversion bandwidth γ is obtained (mainly due to Inequality (3.2) in the proof of Lemma 3.4). Since the lower bound of Theorem 3.5 is achievable, this implies that assuming uniform download necessarily leads to a suboptimal solution. ▸

Remark 3.7: The case where $k^I = k^F$ can be analyzed using the same techniques used in this section. In this case, $\lambda^I = 1$. There are some differences compared to the case of the merge regime: for example, in this case the number of unchanged nodes can be at most $\min\{n^I, n^F\}$ (in contrast to the $\lambda^I k^I$ maximum of the merge regime). So, conversion bandwidth in the case where $n^I \geq n^F$ is zero, since we can simply keep n^F nodes unchanged. In the case where $n^I < n^F$, the same analysis from Lemma 3.4 is followed, but the larger number of unchanged nodes will lead to a slightly different inequality. Thus, in the case of $k^I = k^F$ the lower bound on conversion bandwidth is:

$$\gamma \geq \begin{cases} 0, & \text{if } n^I \geq n^F \\ \alpha\left(k^I + r^I\right)\left(1 - \frac{r^I}{r^F}\right) + (r^F - r^I)\alpha, & \text{otherwise.} \end{cases}$$

Readers familiar with regenerating codes might notice that the above lower bound is equivalent to the lower bound on the repair bandwidth [27, 36] when $(r^F - r^I)$ symbols of an $[k^I + r^F, k^I]$ MDS code are to be repaired with the help of the remaining $(k^I + r^I)$ symbols. Note that this setting imposes a relaxed requirement of repairing only a specific subset of symbols as compared to regenerating codes which require optimal repair of all nodes. Yet, the lower bound remains the same. This is not surprising though, since it has been shown [29] that the regenerating codes lower bound for MDS codes applies even for repair of only a single specific symbol. ▸

3.4 Explicit construction of bandwidth-optimal MDS convertible codes in the merge regime

In this section, we present an explicit construction for bandwidth-optimal convertible codes in the merge regime. Our construction employs the Piggybacking framework [63].

Recall from Section 3.1.4 that the Piggybacking framework is a framework for constructing vector codes using an existing code as a base code and adding specially designed functions called piggybacks which impart additional properties to the resulting code. We use an access-optimal convertible code to construct the base code and design the piggybacks to help achieve minimum conversion bandwidth. First, in Section 3.4.1, we describe our construction of bandwidth-optimal convertible codes in the case where we only consider fixed unique values for the final parameters n^F and $k^F = \lambda^I k^I$. Then, in Section 3.4.2, we show that initial codes built with this construction are not only (n^F, k^F)-bandwidth-optimally convertible, but also simultaneously bandwidth-optimally convertible for multiple other values of the pair (n^F, k^F). Additionally, we present a construction which given any finite set of possible final parameter values (n^F, k^F), constructs an initial $[n^I, k^I]$ code which is simultaneously (n^F, k^F)-bandwidth-optimally convertible for every (n^F, k^F) in that set.

3.4.1 Bandwidth-optimal MDS convertible codes for fixed final parameters

The case where $r^F \geq k^I$ is trivial, since the default approach to conversion is bandwidth-optimal in this case. Therefore, in the rest of this section, we only consider $r^F < k^I$. Moreover, in the case where $r^I \geq r^F$ (decreasing-redundancy region), access-optimal convertible codes (for which explicit constructions are known) are also bandwidth-optimal. Therefore, we focus on the case $r^I < r^F$ (increasing-redundancy region).

We start by describing the base code used in our construction, followed by the design of piggybacks, and then describe the conversion procedure along with the role of piggybacks during conversion. To help illustrate the construction, we keep a running example showing each step.

Base code for piggybacking. As the base code for our construction, we use a *punctured* initial code of an access-optimal $(k^I + r^F, k^I; n^F, k^F)$ convertible code. Any

access-optimal convertible code can be used. However, as mentioned in Section 3.1.2, we assume that this convertible code is: (1) systematic, (2) linear, and (3) only requires accessing the first r^F parities from each initial codeword during access-optimal conversion. We refer to the $[k^I + r^F, k^I]$ initial code of this access-optimal convertible code as $\mathcal{C}^{I'}$, to its $[n^F, k^F]$ final code as $\mathcal{C}^{F'}$. Let $\mathcal{C}^{I''}$ be the punctured version of $\mathcal{C}^{I'}$ where the last $(r^F - r^I)$ parity symbols are punctured.

Example 3.2: Suppose we want to construct a bandwidth-optimal $(5, 4; 10, 8)$ convertible code over a finite field $\mathbb{F}_q$ (assume that q is sufficiently large). As a base code, we use a punctured access-optimal $(6, 4; 10, 8)$ convertible code. For this example, we use the code presented in Example 3.1 and puncture the last parity. Thus, $\mathcal{C}^{I'}$ is a $[6, 4]$ code, $\mathcal{C}^{F'}$ is a $[10, 8]$ code, and $\mathcal{C}^{I''}$ is a $[5, 4]$ code. ▸

Piggyback design. Now, we describe how to construct the $[n^I, k^I, \alpha]$ initial vector code $\mathcal{C}^I$ and the $[n^F, k^F, \alpha]$ final vector code $\mathcal{C}^F$ that make up the bandwidth-optimal $(n^I, k^I; n^F, \lambda^I k^I)$ convertible code.

The first step is to choose the value of α. Let us reexamine the lower bound derived in Theorem 3.5 for $r^I < r^F < k^I$, which is rewritten below in a different form.

$$\gamma \geq \lambda^I \left(r^I \alpha + k^I \left(1 - \frac{r^I}{r^F} \right) \alpha \right) + r^F \alpha.$$

We can see that one way to achieve this lower bound would be to download exactly $\beta_1 = \alpha$ subsymbols from each of the r^I retired nodes in the λ^I initial codewords, and to download $\beta_2 = (1 - r^I/r^F)\,\alpha$ subsymbols from each of the k^I unchanged nodes in the λ^I initial stripes. Thus, we choose $\alpha = r^F$, which is the smallest value that makes β_1 and β_2 integers, thus making:

$$\beta_1 = r^F \qquad \text{and} \qquad \beta_2 = (r^F - r^I).$$

The next step is to design the piggybacks. We first provide the intuition behind the design. Recall from above that we can download $\beta_2 = (r^F - r^I)$ subsymbols from each unchanged node and all the α subsymbols from each retired node. Hence,

we can utilize up to $\beta_2 = (r^F - r^I)$ coordinates from each of the r^I parity nodes for piggybacking. Given that there are precisely $(r^F - r^I)$ punctured symbols and α instances of $\mathcal{C}^{I''}$, we can store piggybacks corresponding to r^I instances of each of these punctured symbols. During conversion, these punctured symbols can be reconstructed and used for constructing the new nodes.

Consider a message $\mathbf{m} \in \mathbb{F}_q^{\lambda^I k^I \alpha}$ split into $\lambda^I \alpha$ submessages $\mathbf{m}_j^{(s)} \in \mathbb{F}_q^{k^I}$, representing the data encoded by instance $j \in [\alpha]$ of the base code in initial codeword $s \in [\lambda^I]$. Recall that $\mathcal{C}^{I''}$ is systematic by construction. Therefore, the submessage $\mathbf{m}_j^{(s)}$ will correspond to the contents of the j-th coordinate of the k^I systematic nodes in initial codeword s. Let $c_{i,j}^I(s)$ denote the contents of the j-th coordinate of parity symbol i in initial codeword s under code $\mathcal{C}^I$, and $c_{i,j}^F$ let denote the same for the single final codeword encoded under $\mathcal{C}^F$. These are constructed as follows:

$$c_{i,j}^I(s) = \begin{cases} \mathbf{m}_j^{(s)} \mathbf{p}_i^I, & \text{for} \quad \begin{array}{c} s \in [\lambda^I], \\ i \in [r^I], \\ 1 \leq j \leq r^I \end{array} \\ \mathbf{m}_j^{(s)} \mathbf{p}_i^I + \mathbf{m}_i^{(s)} \mathbf{p}_j^I, & \text{for} \quad \begin{array}{c} s \in [\lambda^I], \\ i \in [r^I], \\ r^I < j \leq r^F \end{array} \end{cases}$$

$$c_{i,j}^F = [\mathbf{m}_j^{(1)} \cdots \mathbf{m}_j^{(\lambda^I)}] \mathbf{p}_i^F, \qquad \text{for } i \in [r^F],\ j \in [r^F],$$

where $\mathbf{p}_i^I$ corresponds to the encoding vector of the i-th parity of $\mathcal{C}^{I'}$ and $\mathbf{p}_i^F$ corresponds to the encoding vector of the i-th parity of $\mathcal{C}^{F'}$. By using the access-optimal conversion procedure from the base code, we can compute $c_{i,j}^F = [\mathbf{m}_j^{(1)} \cdots \mathbf{m}_j^{(\lambda^I)}] \mathbf{p}_i^F$ from $\{\mathbf{m}_j^{(s)} \mathbf{p}_i^I : s \in [\lambda^I]\}$ for all $i \in [r^F]$ and $j \in [r^F]$. Notice that each initial codeword is independent and encoded in the same way (as required).

This piggybacking design, that of using parity code subsymbols of the base code as piggybacks, is inspired by one of the piggybacking designs proposed in [63], where it is used for efficiently reconstructing failed (parity) code symbols.

Example 3.2 (continued)**:** Let $\mathbf{p}_1^I, \mathbf{p}_2^I \in \mathbb{F}_q^{4 \times 1}$ be the encoding vectors for the parities of

initial codeword 1 ($\mathcal{C}^I$)

a_1	b_1
$\vdots$	$\vdots$
a_4	b_4
$\mathbf{a}^{(1)}\mathbf{p}_1^I$	$\mathbf{b}^{(1)}\mathbf{p}_1^I + \mathbf{a}^{(1)}\mathbf{p}_2^I$

initial codeword 2 ($\mathcal{C}^I$)

a_5	b_5
$\vdots$	$\vdots$
a_8	b_8
$\mathbf{a}^{(2)}\mathbf{p}_1^I$	$\mathbf{b}^{(2)}\mathbf{p}_1^I + \mathbf{a}^{(2)}\mathbf{p}_2^I$

final codeword ($\mathcal{C}^F$)

a_1	b_1
$\vdots$	$\vdots$
a_4	b_4
a_5	b_5
$\vdots$	$\vdots$
a_8	b_8
$\mathbf{ap}_1^F$	$\mathbf{bp}_1^F$
$\mathbf{ap}_2^F$	$\mathbf{bp}_2^F$

Figure 3.5: Example of a bandwidth-optimal $(5, 4; 10, 8)$ convertible code. Each block in this diagram represents a codeword, where each column corresponds to a distinct coordinate of the α-length vector ($\alpha = 2$ in this case), and each row corresponds to a node. The shaded rows correspond to retired nodes for the first two blocks (initial codewords), and new nodes for the third block (final codeword). For the initial codewords, text color is used emphasize the piggybacks. In the final codeword, text color is used to denote the base code subsymbol that is constructed from the piggybacks.

$\mathcal{C}^{I\prime}$, and $\mathbf{p}_1^F, \mathbf{p}_2^F \in \mathbb{F}_q^{8\times 1}$ be the encoding vector for the parities of $\mathcal{C}^{F\prime}$. Since $\alpha = r^F = 2$, we construct a $[5, 4, 2]$ initial vector code $\mathcal{C}^I$ and a $[10, 8, 2]$ final vector code $\mathcal{C}^F$. Let $\mathbf{a} = (a_1, \ldots, a_8)$ and $\mathbf{b} = (b_1, \ldots, b_8)$. Figure 3.5 shows the resulting piggybacked codes encoding submessages $\mathbf{a}^{(1)} = (a_1, \ldots, a_4), \mathbf{a}^{(2)} = (a_5, \ldots, a_8), \mathbf{b}^{(1)} = (b_1, \ldots, b_4), \mathbf{b}^{(2)} = (b_5, \ldots, b_8) \in \mathbb{F}_q^{1\times 4}$. Recall from Example 3.1, that $\mathbf{a}^{(1)}\mathbf{p}_i^I + \alpha_i^4\mathbf{a}^{(2)}\mathbf{p}_i^I = \mathbf{a}\mathbf{p}_i^F$ for $i \in \{1, 2\}$ (and equivalently for $\mathbf{b}$). ▸

Conversion procedure. Conversion proceeds as follows:

1. Download $D = \{\mathbf{m}_j^{(s)} : s \in [\lambda^I] \text{ and } r^I < j \leq r^F\}$, $C_1 = \{c_{i,j}^I(s) : s \in [\lambda^I],\ i \in [r^I],\ \text{and } 1 \leq j \leq r^I\}$, and $C_2 = \{c_{i,j}^I(s) : s \in [\lambda^I],\ i \in [r^I],\ \text{and } r^I < j \leq r^F\}$.
2. Recover the piggybacks $C_3 = \{\mathbf{m}_j^{(s)}\mathbf{p}_i^I : s \in [\lambda^I],\ r^I < i \leq r^F,\ \text{and } 1 \leq j \leq r^I\}$ by computing $\mathbf{m}_i^{(s)}\mathbf{p}_j^I$ from D and obtaining $\mathbf{m}_j^{(s)}\mathbf{p}_i^I = c_{j,i}^I(s) - \mathbf{m}_i^{(s)}\mathbf{p}_j^I$ using C_2.
3. Compute the remaining base code symbols from the punctured symbols $C_4 = \{\mathbf{m}_i^{(s)}\mathbf{p}_j^I : s \in [\lambda^I],\ r^I < i \leq r^F,\ \text{and } r^I < j \leq r^F\}$ using D.
4. Compute the parity nodes of the final codeword specified by the subsymbols $C_5 = \{c_{i,j}^F : i \in [r^F],\ j \in [r^F]\}$. This is done by using the conversion procedure from the access-optimal convertible code used as base code to compute C_5 from $C_1,\ C_2,\ C_3$, and C_4.

This procedure requires downloading β_1 subsymbols from each retired node and β_2 subsymbols from each unchanged node. Thus, the read conversion bandwidth is:

$$\begin{aligned}\gamma_R &= \lambda^I\left(r^I\beta_1 + k^I\beta_2\right)\\ &= \lambda^I\left(r^I\alpha + k^I\left(1 - \frac{r^I}{r^F}\right)\alpha\right).\end{aligned}$$

Additionally, $r^F\alpha$ write conversion bandwidth is required for the new nodes.

$$\gamma_W = r^F\alpha$$

Since $\gamma = \gamma_R + \gamma_W$, this matches Theorem 3.5.

Example 3.2 (continued)**:** During conversion, only 12 subsymbols need to be downloaded: $\mathbf{b}^{(1)}, \mathbf{b}^{(2)}$ and all the parity symbols from both codewords. From these subsymbols, we can recover the piggyback terms $\mathbf{a}^{(1)}\mathbf{p}_2^I$ and $\mathbf{a}^{(2)}\mathbf{p}_2^I$, and then compute $\mathbf{b}^{(1)}\mathbf{p}_2^I$ and $\mathbf{b}^{(2)}\mathbf{p}_2^I$ in order to reconstruct the second parity symbol of $\mathcal{C}^{I\prime}$. Finally, we use $\mathbf{a}^{(i)}\mathbf{p}_1^I, \mathbf{b}^{(i)}\mathbf{p}_1^I, \mathbf{a}^{(i)}\mathbf{p}_2^I, \mathbf{b}^{(i)}\mathbf{p}_2^I$ for $i \in \{1, 2\}$ with the conversion procedure from the access-optimal convertible code to compute the base code symbols $\mathbf{a}\ \mathbf{p}_1^F, \mathbf{a}\ \mathbf{p}_2^F, \mathbf{b}\ \mathbf{p}_1^F$ and $\mathbf{b}\ \mathbf{p}_2^F$ of the new nodes.

The default approach would require one to download 16 subsymbols in total from the initial nodes. Both approaches require downloading 4 subsymbols in total from the coordinator node to the new nodes. Thus, the proposed construction leads to 20% reduction in conversion bandwidth as compared to the default approach of reencoding. ▸

3.4.2 Convertible codes with bandwidth-optimal conversion for multiple final parameters

In practice, the final parameters n^F, k^F might depend on observations made after the initial encoding of the data and hence they may be unknown at code construction time. In particular, for a $(n^I, k^I; n^F, \lambda^I k^I)$ convertible code in the merge regime this means that the values of λ^I and $r^F = (n^F - k^F)$ are unknown.

To ameliorate this problem, we now present convertible codes which support bandwidth-optimal conversion *simultaneously* for multiple possible values of the final parameters. Recall property (4) of the access-optimal base code which we reviewed in Section 3.1.2: when constructed with a given value of $\lambda^I = \lambda$ and $r^F = r$, the initial $[n^I, k^I]$ code is (n^F, k^F)-access-optimally convertible for all $k^F = \lambda' k^I$ and $n^F = k^F + r'$ such that $1 \leq \lambda' \leq \lambda$ and $1 \leq r' \leq r$.

Supporting multiple values of λ^I

The construction from Section 3.4 with a particular value of $\lambda^I = \lambda$, intrinsically supports bandwidth-optimal conversion for any $\lambda^I = \lambda' < \lambda$. This is a consequence

of property (4) above, and can be done easily by considering one or multiple of the initial codewords as consisting of zeroes only, and ignoring them during conversion. From Theorem 3.5, it is easy to see that this modified conversion procedure achieves the optimal conversion bandwidth for the new parameter $\lambda^I = \lambda'$.

Supporting multiple values of r^F

We break this scenario into two cases:

Case 1 (supporting $r^F \leq r^I$): due to property (4) above, the base code used in the construction from Section 3.4 supports access-optimal conversion for any value of $r^F = r$ such that $r \leq r^I$. Using this property, one can achieve bandwidth optimality for any $r \leq r^I$ by simply using the access-optimal conversion on each of the α instances of the base code independently. The only difference is that some of the instances might have piggybacks, which can be simply ignored. The final code might still have these piggybacks, however they will still satisfy the property that the piggybacks in instance i $(2 \leq i \leq \alpha)$ only depend on data from instances $\{1, \ldots, (i-1)\}$. Thus, the final code will have the MDS property and the desired parameters.

Case 2 (supporting $r^F > r^I$): for supporting multiple values of $r^F \in \{r_1, r_2, \ldots, r_s\}$ such that $r_i > r^I$ $(i \in [s])$, we start with an access-optimal convertible code having $r^F = \max_i r_i$. Then we repeat the piggybacking step of the construction (see Section 3.4.1) for each r_i, using the resulting code from step i (with the punctured symbols from $\mathcal{C}^{I'}$ added back) as a base code for step $(i+1)$. Therefore, the resulting code will have $\alpha = \prod_{i=1}^{s} r_i$. Since the piggybacking step will preserve the MDS property of its base code, and the initial code used in the first piggyback step is MDS, it is clear that the initial code resulting from the last piggybacking step will also be MDS. Conversion for one of the supported $r^F = r_i$ is performed as described in Section 3.4.1 on each of the additional instances created by steps $(i+1), \ldots, s$ (i.e. $\prod_{i'=(i+1)}^{s} r_{i'}$ in total). As before, some of these instances after conversion will have piggybacks, which can be simply ignored, as the resulting code will continue to have the property that piggybacks from a given instance only depend on data from earlier instances.

initial codeword i ($\mathcal{C}^I$)

a_{4i-3}	b_{4i-3}	c_{4i-3}	d_{4i-3}	e_{4i-3}	f_{4i-3}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
a_{4i}	b_{4i}	c_{4i}	d_{4i}	e_{4i}	f_{4i}
$\mathbf{a}^{(i)}\mathbf{p}_1^I$	$\mathbf{b}^{(i)}\mathbf{p}_1^I + \mathbf{a}^{(i)}\mathbf{p}_2^I$	$\mathbf{c}^{(i)}\mathbf{p}_1^I + \mathbf{a}^{(i)}\mathbf{p}_2^I$	$\mathbf{d}^{(i)}\mathbf{p}_1^I + \mathbf{c}^{(i)}\mathbf{p}_2^I + \mathbf{b}^{(i)}\mathbf{p}_2^I$	$\mathbf{e}^{(i)}\mathbf{p}_1^I + \mathbf{a}^{(i)}\mathbf{p}_3^I$	$\mathbf{f}^{(i)}\mathbf{p}_1^I + \mathbf{e}^{(i)}\mathbf{p}_2^I + \mathbf{b}^{(i)}\mathbf{p}_3^I$

final codeword ($r^F = 2$)

a_1	b_1	c_1	d_1	e_1	f_1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
a_8	b_8	c_8	d_8	e_8	f_8
$\mathbf{ap}_1^F$	$\mathbf{bp}_1^F$	$\mathbf{cp}_1^F + \mathbf{a}^{(1)}\mathbf{p}_2^I + \mathbf{a}^{(2)}\mathbf{p}_2^I$	$\mathbf{dp}_1^F$	$\mathbf{ep}_1^F + \mathbf{a}^{(1)}\mathbf{p}_3^I + \mathbf{a}^{(2)}\mathbf{p}_3^I$	$\mathbf{fp}_1^F$
$\mathbf{ap}_2^F$	$\mathbf{bp}_2^F$	$\mathbf{cp}_2^F$	$\mathbf{dp}_2^F$	$\mathbf{ep}_2^F$	$\mathbf{fp}_2^F$

final codeword ($r^F = 3$)

a_1	b_1	c_1	d_1	e_1	f_1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
a_8	b_8	c_8	d_8	e_8	f_8
$\mathbf{ap}_1^F$	$\mathbf{bp}_1^F + \mathbf{a}^{(1)}\mathbf{p}_2^I + \mathbf{a}^{(2)}\mathbf{p}_2^I$	$\mathbf{cp}_1^F$	$\mathbf{dp}_1^F$	$\mathbf{ep}_1^F$	$\mathbf{fp}_1^F$
$\mathbf{ap}_2^F$	$\mathbf{bp}_2^F$	$\mathbf{cp}_2^F$	$\mathbf{dp}_2^F$	$\mathbf{ep}_2^F$	$\mathbf{fp}_2^F$
$\mathbf{ap}_3^F$	$\mathbf{bp}_3^F$	$\mathbf{cp}_3^F$	$\mathbf{dp}_3^F$	$\mathbf{ep}_3^F$	$\mathbf{fp}_3^F$

Figure 3.6: Example of a $[5, 4]$ MDS code that supports bandwidth-optimal conversion to multiple final codes. This code supports bandwidth-optimal conversion to a $[8+r, 8]$ MDS code for $r = 1, 2, 3$. Piggybacks from the first round ($r = 2$) are colored orange and piggybacks from the second round ($r = 3$) are colored magenta. In the possible final codewords, text color is used to show base code symbols which are directly computed from the corresponding piggybacks, or to denote leftover piggybacks that were not used during conversion.

Example 3.3 (bandwidth-optimal conversion for multiple final parameters)**:** In this example, we will extend the $(5, 4; 10, 8)$ convertible code from Example 3.2 ($r^F = 2$) to construct a code which additionally supports bandwidth-optimal conversion to an $[11, 8]$ MDS code ($r^F = 3$). Figure 3.6 shows initial codeword $i \in \{1, 2\}$ of the new initial vector code, which has $\alpha = 2 \cdot 3 = 6$. Here $\mathbf{a}^{(1)} = (a_1, \ldots, a_4)$, $\mathbf{a}^{(2)} = (a_5, \ldots, a_8) \in \mathbb{F}_q^{1\times 4}$, $\mathbf{a} = (a_1, \ldots, a_8) \in \mathbb{F}_q^{1\times 8}$, and similarly for $\mathbf{b}, \ldots, \mathbf{f}$. The vectors $\mathbf{p}_i^I \in \mathbb{F}_q^{4\times 1}$ are the encoding vectors of the initial code $\mathcal{C}^{I'}$ and $\mathbf{p}_i^F \in \mathbb{F}_q^{8\times 1}$ are encoding vectors of the final code $\mathcal{C}^{F'}$ ($i \in \{1, 2, 3\}$). Since the maximum supported r^F is 3, we start with an access-optimal $(7, 4; 11, 8)$ convertible code. Thus, $\mathcal{C}^{I'}$ is a $[7, 4]$ code, $\mathcal{C}^{F'}$ is a $[11, 8]$ code, and $\mathcal{C}^{I''}$ is a $[5, 4]$ code. In the first round of piggybacking we consider $r^F = 2$, which yields the code shown in Example 3.2. In the second round of piggybacking we consider $r^F = 3$ and piggyback the code resulting from the first round, which yields the code shown in Figure 3.6. Conversion for $r^F = 1$ proceeds by simply downloading the contents of the single parity node and using the access-optimal conversion procedure. Conversion for $r^F = 2$ proceeds by treating this code as three instances of the code from Example 3.2 and performing conversion for each one independently. Conversion for $r^F = 3$ proceeds by treating this code as a vector code with $\alpha = 3$ and base field $\mathbb{F}_{q^2}$ (i.e. each element is a vector over $\mathbb{F}_q$ of length 2). ▸

Remark 3.8 (Field size requirement)**:** The field size requirement for $\mathbb{F}_q$ of the constructions presented in this section is given by the field size requirement of the base code used. The currently lowest known field size requirement for an explicit construction of systematic linear access-optimal convertible codes in the merge regime is given by [96]. For typical parameters, this requirement is roughly $q \geq \Omega(2^{\lambda^I (n^I)^3})$. When $r^F \leq r^I - \lambda^I + 1$, this can be significantly reduced to $q \geq k^I r^I$. And when $r^F \leq \lfloor r^I / \lambda^I \rfloor$, this can be further reduced to $q \geq \max\{n^I, n^F\}$. ▸

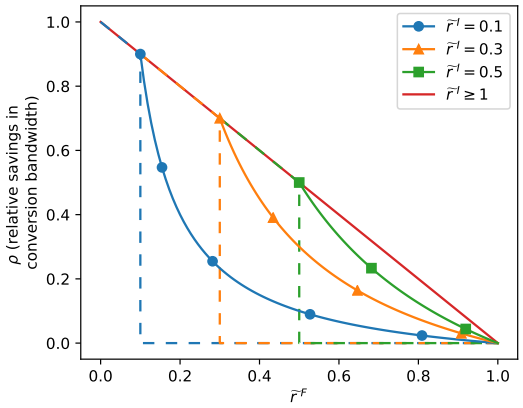


Figure 3.7: Achievable savings in conversion bandwidth by bandwidth-optimal convertible codes in comparison to the default approach to conversion. Here $\tilde{r}^I = r^I/k^I$ and $\tilde{r}^F = r^F/k^I$ are the initial and final redundancies, divided by the initial code dimension. Each curve shows the relative savings for a fixed value of $\tilde{r}^I$, as $\tilde{r}^F$ varies. Solid lines indicate bandwidth-optimal convertible codes, and dashed lines indicate access-optimal convertible codes. Notice that each curve overlaps with the red curve ($\tilde{r}^I \geq 1$) in the range $\tilde{r}^F \in (0, \tilde{r}^I]$.

3.5 Bandwidth savings of bandwidth-optimal convertible codes

In this section, we show the amount of savings in bandwidth that can be obtained by using bandwidth-optimal convertible codes in the merge regime, relative to the default approach to conversion. We present the amount of savings in terms of two ratios:

$$\tilde{r}^I = (r^I/k^I) \qquad \text{and} \qquad \tilde{r}^F = (r^F/k^I),$$

i.e. the initial and final amount of "redundancy" relative to the initial dimension of the code. For simplicity, we only consider the read conversion bandwidth (data sent from initial nodes to the coordinator node), since the write conversion bandwidth (data sent from the coordinator node to the new nodes) is fixed for stable convertible codes (specifically, it is equal to αr^F). Thus, the conversion bandwidth of the default approach is always $\lambda^I k^I \alpha$. Figure 3.7 shows the relative savings, i.e. the ratio between

the conversion bandwidth of optimal conversion and the conversion bandwidth of the default approach, for fixed values of $\tilde{r}^I \in (0, \infty)$ and varying $\tilde{r}^F \in (0, \infty)$.

Each curve shown in Figure 3.7 can be divided into three regions, depending on the value of $\tilde{r}^F$:

- **Region $0 < \tilde{r}^F \leq \tilde{r}^I$ and $\tilde{r}^F < 1$:** these conditions imply that $r^F \leq r^I$, so by Lemma 3.2 the conversion bandwidth is $\lambda^I r^F \alpha$, and the relative savings are:

$$\rho = 1 - \frac{\lambda^I r^F \alpha}{\lambda^I k^I \alpha} = 1 - \tilde{r}^F.$$

  This region corresponds to the decreasing-redundancy region, and in this region access-optimal convertible codes are also bandwidth-optimal. This region of the curve is linear, and the amount of savings is not affected by $\tilde{r}^I$.

- **Region $\tilde{r}^I < \tilde{r}^F < 1$:** this implies that $r^I \leq r^F \leq k^I$, and by Lemma 3.4 the conversion bandwidth is $\lambda^I \alpha(r^I + k^I(1 - r^I/r^F))$, and the relative savings are:

$$\rho = 1 - \frac{\lambda^I \alpha \left(r^I + k^I \left(1 - \frac{r^I}{r^F}\right)\right)}{\lambda^I k^I \alpha} = \tilde{r}^I \left(\frac{1}{\tilde{r}^F} - 1\right).$$

  This corresponds to the increasing-redundancy region, where access-optimal convertible codes provide no conversion bandwidth savings. Thus bandwidth-optimal convertible codes provide substantial savings in conversion bandwidth in this regime, compared to access-optimal convertible codes.

- **Region $\tilde{r}^F \geq 1$:** this implies that $r^F \geq k^I$ and by Lemma 3.2 a conversion bandwidth of $\lambda^I k^I \alpha$ is required. Thus no savings in conversion bandwidth are possible in this region.

Thus, bandwidth-optimal convertible codes allow for savings in conversion bandwidth on a much broader region relative to access-optimal convertible codes.

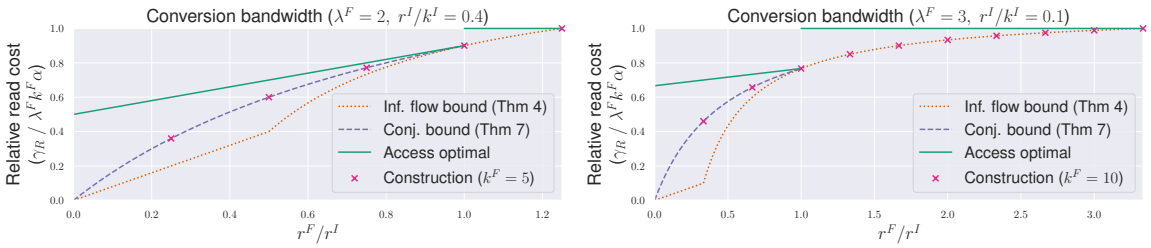


Figure 3.8: Read-conversion-bandwidth relative to the default approach (split regime). In each plot, the value of the parameters λ^F and the ratio r^I/k^I are fixed, and the value of the ratio r^F/r^I ranges in $(0, (\lambda^F r^I/k^I)^{-1}]$. By choosing this parametrization, the plotted curves are independent of the value of k^F. For illustration, markers are added on the points that can be achieved by the construction of Section 3.7 when k^F takes the given example value.

3.6 Conversion bandwidth of the split regime

In this section we analyze the conversion bandwidth required by MDS convertible codes in the split regime, i.e., the case where $k^I = \lambda^F k^F$ for some integer $\lambda^F \geq 2$.

In order to obtain a lower bound on the conversion bandwidth, we model split conversion as an information flow problem. In this model, we represent the flow of information during conversion as a DAG with edges with variable capacity that represent the transfer of data between nodes. Our objective is to set the capacity of edges in a way that minimizes the conversion bandwidth, while ensuring that the flow conditions necessary for conversion are met.

One challenge is that, as we will show, the bound we obtain through information flow is not achievable in general.[1] This bound is not achievable in general because retired symbols contain data that is associated with more than one final codeword. Thus, in order to make use of these symbols during conversion, we must also download enough data from unchanged symbols to remove the "interference" from other final codewords. To this end, we introduce a conjecture and derive from it a lower bound which, as we show in Section 3.7, is achievable.

[1]Split conversion corresponds to a *multi-source multicast* problem. In this case (unlike the *single-source* case) the information flow bound is not necessarily tight with respect to coding [109].

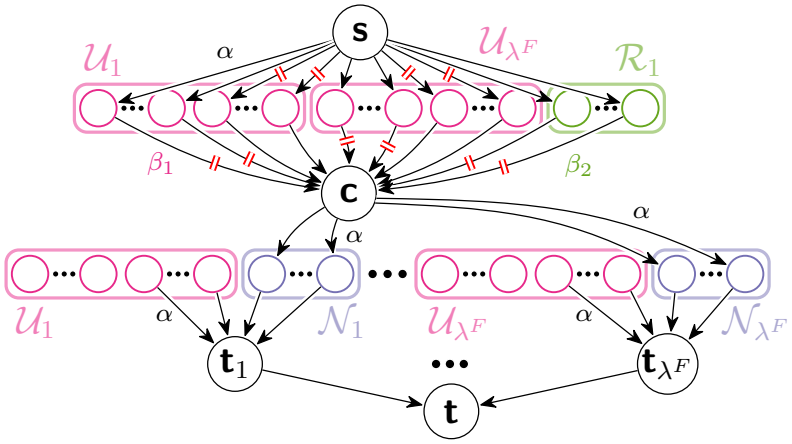


Figure 3.9: Information flow graph of split conversion. For clarity, each unchanged symbol is drawn twice, in order to show the initial configuration of the system in the top row of nodes, and the final configuration in the bottom row of nodes. The edges with a red mark depict a graph cut.

3.6.1 Information flow

We model the conversion process using the graph (see Figure 3.9) composed by the following nodes:

- source $\mathbf{s}$, representing the whole data $\mathbf{m} \in \mathbb{F}_q^{\alpha k^I}$;
- the set $\mathcal{U}_i$ for $i \in [\lambda^F]$, representing the unchanged symbols of final codeword i;
- the set $\mathcal{R}$ representing retired symbols;
- the set $\mathcal{N}_i$ for $i \in [\lambda^F]$, representing the new symbols of final codeword i;
- data collectors $\mathbf{t}_i$ for $i \in [\lambda^F]$ that represent the decoders for each final codeword;
- a central node $\mathbf{c}$ that computes the new symbols;
- a sink $\mathbf{t}$ collecting the data for all final codewords (i.e. $\mathbf{m}$).

Let (u, v, x) denote and edge from node u to node v with capacity $x \geq 0$. Nodes are connected by the following edges:

- $\{(\mathbf{s}, x, \alpha) \mid x \in \bigcup_i \mathcal{U}_i \cup \mathcal{R}\}$, representing the data stored in the initial symbols;
- $\{(x, \mathbf{c}, \beta_1) \mid x \in \bigcup_i \mathcal{U}_i\}$ representing the data downloaded from unchanged symbols;
- $\{(x, \mathbf{c}, \beta_2) \mid x \in \mathcal{R}\}$, representing the data downloaded from retired symbols;
- $\{(\mathbf{c}, x, \alpha) \mid x \in \bigcup_i \mathcal{N}_i\}$, representing the data written to the new symbols;
- $\{(x, \mathbf{t}_i, \alpha) \mid x \in V_i\}$ for $V_j \subseteq \bigcup_i (\mathcal{U}_i \cup \mathcal{N}_i)$ such that $|V_j| = k^F$ for $j \in [\lambda^F]$, representing decoding of final codeword i;
- $\{(\mathbf{t}_i, \mathbf{t}, \alpha k^F) \mid i \in [\lambda^F]\}$, representing the collection of all the decoded data.

In this chapter, we focus on stable codes (see Chapter 1). Therefore, we have that $|\mathcal{U}_i| = k^F$, $|\mathcal{R}| = r^I$, and $|\mathcal{N}_i| = r^F$ $(i \in [\lambda^F])$. The total conversion bandwidth γ will be given by the total size of the information communicated between nodes during conversion, which corresponds to the following equation:

$$\begin{gathered} \gamma := \gamma_R + \gamma_W, \\ \text{where } \gamma_R := \lambda^F k^F \beta_1 + r^I \beta_2 \text{ and } \gamma_W := \lambda^F r^F \alpha. \end{gathered} \tag{3.4}$$

We refer to γ_R as the *read conversion bandwidth* and to γ_W as the *write conversion bandwidth*. Our objective is to set (β_1, β_2) to minimize γ while ensuring an information flow of size αk^I (the size of the data $\mathbf{m}$) is feasible. Since γ_W is constant with respect to (β_1, β_2), our analysis will focus on γ_R.

Note that our model assumes a uniform amount of data downloaded from unchanged symbols and retired symbols. This is without loss of generality, since any stable convertible code with non-uniform downloads, can be made uniform by repeating the code a sufficient number of times and rotating the assignment of symbols to nodes with each repetition.

Our first lemma expresses the constraint which arises from considering the cut shown in Figure 3.9.

Lemma 3.9. *For all stable MDS* $(n^I, k^I = \lambda^F k^F; n^F, k^F)$ *convertible code:*

$$\lambda^F \min\{r^F, k^F\}\alpha \leq \lambda^F \min\{r^F, k^F\}\beta_1 + r^I \beta_2. \tag{3.5}$$

Proof. For each $j \in [\lambda^F]$, consider a sink $\mathbf{t}_j$ that connects to all symbols in a final codeword but a set $S_j \subseteq \mathcal{U}_j$ of size $\min\{k^F, r^F\}$. Consider the cut defined by $\{\mathbf{s}\} \cup \bigcup_{j=1}^{\lambda^F} S_j \cup \mathcal{R}$. This cut yields (3.5) after simplification. □

Using (3.4), we can show that when $r^F \geq k^F$, no savings in conversion bandwidth are possible over the default approach.

Corollary 3.10. *When* $r^F \geq k^F$*, we have* $\gamma_R \geq \lambda^F k^F \alpha$.

In other words, the default approach has optimal conversion bandwidth when $r^F \geq k^F$. For this reason, we will only focus on the case $r^F < k^F$.

To obtain a lower bound on γ, we will minimize it subject to (3.5) with β_1 and β_2 as variables.

Lemma 3.11. *Assume* $r^F < k^F$*. Then, the value of* γ *is minimized subject to* (3.5) *when:*

$$\beta_1 = \max\left\{1 - \frac{r^I}{\lambda^F r^F}, 0\right\}\alpha, \qquad \beta_2 = \min\left\{1, \frac{\lambda^F r^F}{r^I}\right\}\alpha.$$

Proof. As intuition, note that β_2 offers the better "bang for the buck" for satisfying (3.5), because each unit of β_2 contributes r^I costing r^I, while each unit of β_1 contributes $\lambda^F r^F$ costing $\lambda^F k^F$. Thus, in order to satisfy (3.5), it is better to increase β_2 first, and then increase β_1 if necessary. This approach leads to the proposed solution. It is straightforward to check that this solution satisfies the Karush-Kuhn-Tucker (KKT) conditions, and is thus an optimal solution. □

By replacing into (3.4), we obtain the following lower bound.

Theorem 3.12. *For all stable MDS* $(n^I, k^I = \lambda^F k^F; n^F, k^F)$ *convertible code:*

$$\gamma_R \geq \begin{cases} \lambda^F k^F \alpha - r^I \alpha \max\left\{\frac{k^F}{r^F} - 1, 0\right\} & \text{if } r^I \leq \lambda^F r^F, \\ \lambda^F \min\{r^F, k^F\}\alpha & \text{otherwise.} \end{cases}$$

Proof. Follows from Lemma 3.11 and case analysis. □

This bound shows that there is potential for conversion bandwidth savings when $k^F > r^F$, because the bound is strictly lower than the default approach ($\lambda^F k^F \alpha$) in this region. Unfortunately, this bound is not always achievable, as we see next.

For example, suppose we have have a stable convertible code with $k^F > r^F$, $r^I = \lambda^F r^F$ and that we set $\beta_1 = 0$ and $\beta_2 = \alpha$. This assignment satisfies Theorem 3.12 (and it is easy to check that it leads to a feasible flow in Figure 3.9). However, as shown by previous work on access cost of conversion [99], it is not possible to perform conversion in this case by accessing fewer than $(\lambda^F - 1)k^F + r^F$ symbols. Furthermore, it can be shown that any assignment that makes $\beta_1 > 0$ necessarily leads to a higher conversion bandwidth than the lower bound of Theorem 3.12. The fundamental problem in this case is that to create new symbols for a particular final codeword we need to remove the interference from all other final codewords. This is not possible if the conversion procedure does not access a sufficient number of symbols.

For this reason, we introduce the following conjecture, which lower bounds the amount of data that needs to be downloaded from unchanged symbols based on the above intuition.

Conjecture 3.13. *In the information flow model presented in this section, for all stable MDS $(n^I, k^I = \lambda^F k^F; n^F, k^F)$ convertible code we must have:*

$$\lambda^F \beta_1 \geq (\lambda^F - 1)\beta_2. \tag{3.6}$$

□?

We incorporate this constraint into the minimization of γ and obtain a different solution, which limits the amount of data downloaded from retired symbols when $r^I > r^F$.

Lemma 3.14. *Assume $r^F < k^F$. Then, the minimum value of γ subject to (3.5) and (3.6) is achieved by Lemma 3.11 when $r^I < r^F$, and otherwise by:*

$$\beta_1 = \frac{(\lambda^F - 1)r^F \alpha}{(\lambda^F - 1)r^F + r^I}, \qquad \beta_2 = \frac{\lambda^F r^F \alpha}{(\lambda^F - 1)r^F + r^I}.$$

Proof. As in the case of Lemma 3.11, it is intuitively better to increase β_2 rather than β_1. However, (3.6) gives an upper bound on β_2 in terms of β_1. Therefore, we set $\beta_2 = \min\left\{\alpha, \frac{\lambda^F}{\lambda^F-1}\beta_1\right\}$. We then replace β_2 in (3.5) and set β_1 in order to satisfy the inequality. When $r^I < r^F$, one can check that (3.6) is not tight, and thus we obtain the same values that Lemma 3.11. Otherwise, we obtain the stated values of β_1 and β_2. It is straightforward to check that this solution satisfies the KKT conditions, and is thus an optimal solution. □

By replacing back into (3.4), we obtain the following lower bound based on Conjecture 3.13.

Theorem 3.15. *If Conjecture 3.13 holds, then for all $(n^I, k^I = \lambda^F k^F; n^F, k^F)$ convertible code with $r^I \geq r^F$ and $r^F \leq k^F$:*

$$\gamma_R \geq \lambda^F r^F \alpha \frac{(\lambda^F - 1)k^F + r^I}{(\lambda^F - 1)r^F + r^I}.$$

Proof. Follows from Lemma 3.14. □

As we shall see in Section 3.7, the proposed constructions achieve the combination of the lower bounds of Theorem 3.12 and Theorem 3.15. Thus, we finish this section by comparing the conversion bandwidth of our approach with that of the default approach and existing convertible codes optimized for access cost [99]. Since in all approaches the write conversion bandwidth is equal $(\lambda^F r^F \alpha)$, we focus on the read conversion bandwidth. Table 3.1 includes the expressions for the read conversion bandwidth of different approaches. Figure 3.8 plots the lower bounds on read conversion bandwidth relative to the default approach for some example parameters. These results show that our approach can achieve significant savings in conversion bandwidth with respect to the default approach and access-optimal convertible codes.

3.7 Explicit constructions

In this section, we present constructions for convertible codes in the split regime that optimize for conversion bandwidth. The constructions employ the Piggybacking

framework [63].

Theorem 3.16. *The constructions presented in this section achieve the optimal conversion bandwidth when $r^I \leq r^F$. Furthermore, they achieve the optimal conversion bandwidth when $r^I > r^F$ if Conjecture 3.13 is true.*

Proof. Follows directly from the design and description of the constructions below. □

These construction require less conversion bandwidth than the default approach *and* the access optimal approach (regardless of Conjecture 3.13) as long as $r^F < k^F$ (Corollary 3.10). We begin by describing the base code used in both constructions, and then present the piggybacking constructions for the cases $r^I > r^F$ and $r^I \leq r^F$, respectively.

3.7.1 The base code

We utilize an $[n^I, k^I]$ systematic code with a Vandermonde matrix with evaluation points $(\xi_1, \ldots, \xi_{r^I})$ as the parity matrix. A code of this form is guaranteed to be MDS when choosing ξ_t ($t \in [r^I]$) and field size as specified by the general construction in [96]. Nonetheless, in practice it is often possible to search for ξ_t that generate an MDS code over a given finite field. Let $\mathbf{h}_t := (h_1^{(t)}, \ldots, h_{k^I}^{(t)})^T = (1, \xi_t, \ldots, \xi_t^{k^I-1})^T$ be parity encoding vector $t \in [r^I]$ of the base code. In our construction, we use the property that $(h_1^{(t)}, \ldots, h_{k^F}^{(t)}) = \xi_t^{-(i-1)k^F}(h_{(i-1)k^F+1}^{(t)}, \ldots, h_{ik^F}^{(t)})$ for all $t \in [r^I]$ and $i \in [\lambda^F]$.

3.7.2 Piggybacking construction (case $r^I > r^F$)

We now describe the construction (assuming $r^F < k^F$). Recall that during conversion, we download β_1 from each unchanged symbol, and β_2 from each retired symbol, which are set as discussed in Section 3.6. If we set the set the size of each symbol as $\alpha := ((\lambda^F - 1)r^F + r^I)$, then $\beta_1 := (\lambda^F - 1)r^F$ and $\beta_2 := \lambda^F r^F$. For simplicity, we

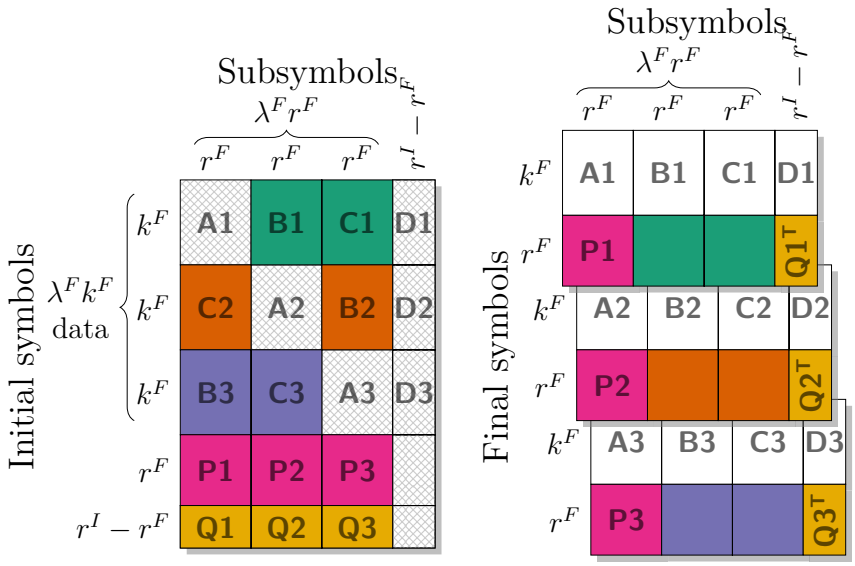


Figure 3.10: Diagram of convertible code construction in the split regime when $r^I > r^F$ and $\lambda^F = 3$.

divide α into blocks: for a given $\ell \in [\alpha]$ we define (ℓ_1, ℓ_2) as follows.

$$(\ell_1, \ell_2) := \begin{cases} \left(\left\lceil \frac{\ell}{r^F} \right\rceil, (\ell - 1 \bmod r^F) + 1\right) & \text{if } \ell \leq \lambda^F r^F, \\ (\lambda^F + 1,\ \ell - \lambda^F r^F) & \text{otherwise.} \end{cases}$$

To describe the encoding vectors of our code, we decompose each encoding vector of the base code into λ^F vectors of length k^F, corresponding to the data associated with each final codeword. Then, we represent each of these vector in the αk^I-dimensional space corresponding to the whole data $\mathbf{m}$ (by filling the additional dimensions with zeros). Specifically, we define $\mathbf{p}_{t,\ell}^{(i)} \in \mathbb{F}_q^{\alpha k^I}$ as the column vector such that $\mathbf{m}\mathbf{p}_{t,\ell}^{(i)}$ corresponds to the encoding of the data under the base code for parity $t \in [r^I]$, final codeword $i \in [\lambda^F]$, and instance $\ell \in [\alpha]$. We achieve this by setting $\mathbf{p}_{t,\ell}^{(i)}[(i-1)k^F\alpha + (j-1)\alpha + \ell] := \mathbf{h}_t[(i-1)k^F + j]$ for $j \in [k^F]$ and 0 everywhere else.

We specify how to construct $\mathbf{q}_{t,\ell}^{I} \in \mathbb{F}_q^{\alpha k^I}$, which is the encoding vector for instance $\ell \in [\alpha]$ of parity $t \in [r^I]$ of the initial codeword, and $\mathbf{q}_{t,\ell}^{F(i)} \in \mathbb{F}_q^{\alpha k^I}$ which is the encoding vector for instance $\ell \in [\alpha]$ of parity $t \in [r^F]$ of final codeword $i \in [\lambda^F]$. The

construction is designed so that the final codewords are all encoded under the same final code. Figure 3.10 shows a diagram for this construction. The construction has three important elements:

1. *Permutation*: In the initial code, the first λ^F blocks of the data symbols associated with final codeword i are circularly shifted to the right $i-1$ times (denoted with letters **A**-**C**). This reordering is logical (no data is moved) and used for describing the code only.

2. *Projection*: For parities 1 through r^F (**P** blocks), we use the base code without modification to encode each data column. During conversion, we download blocks $\{2, \ldots, \lambda^F\}$ from each data symbol (blocks **B** and **C**) and subtract their interference from the corresponding parity symbols to obtain the first block of each final codeword (**P**) blocks).

3. *Piggybacks*: For parities (r^F+1) through r^I (**Q** blocks), we use the base code and add piggybacks to block $\ell_1 \in [\lambda^F]$ that contain the subsymbols of block (λ^F+1) of final codeword ℓ_1 (transposed). During conversion, we recover the piggybacks by using the downloaded data (blocks **B** and **C**). Note that the piggybacks will still have extra data remaining from the unaccessed block (**A**). However, the final code can still be sequentially decoded (the same way that codes in the piggyback framework are decoded).

The remaining parity subsymbols are generated from the accessed data blocks (**B** and **C**). Finally, parity symbol $t \in [r^F]$ in final codeword $i \in [\lambda^F]$ is scaled by $\xi_t^{-(i-1)k^F}$ to ensure that all final codewords are encoded by the same final code (as described in Section 3.7.1). Let $\overrightarrow{\ell}(i) := ((\ell_1 - i \bmod \lambda^F)k^F + \ell_2)$ be the instance index after permutation. Then, the encoding vectors for the initial and final codes are defined as:

$$\mathbf{q}^I_{t,\ell} := \begin{cases} \sum_{i=1}^{\lambda^F} \mathbf{p}^{(i)}_{t,\overrightarrow{\ell}(i)} & \text{if } t \leq r^F,\ \ell_1 \leq \lambda^F, \\ \sum_{i=1}^{\lambda^F} \mathbf{p}^{(i)}_{t,\overrightarrow{\ell}(i)} + \mathbf{p}^{(\ell_1)}_{\ell_2,(\lambda^F-1)r^F+t} & \text{if } t > r^F,\ \ell_1 \leq \lambda^F, \\ \sum_{i=1}^{\lambda^F} \mathbf{p}^{(i)}_{t,\ell} & \text{otherwise.} \end{cases}$$

$$\mathbf{q}_{t,\ell}^{F(i)} := \begin{cases} \xi_t^{-(i-1)k^F} \mathbf{p}_{t,\ell}^{(i)} & \text{if } \ell_1 \leq \lambda^F, \\ \underbrace{\xi_t^{-(i-1)k^F}}_{\text{scaling factor}} (\mathbf{p}_{t,\ell}^{(i)} + \underbrace{\mathbf{p}_{r^F+\ell_2,t}^{(i)}}_{\text{extra data}}) & \text{otherwise.} \end{cases}$$

Notice that during conversion we only need to download $\beta_1 = (\lambda^F - 1)r^F$ subsymbols from each unchanged symbol and $\beta_2 = \lambda^F r^F$ subsymbols from each retired symbol, out of the $\alpha = ((\lambda^F - 1)r^F + r^I)$ subsymbols in each symbol. Therefore, this construction achieves the conversion bandwidth bound of Theorem 3.15. Furthermore, the constructed code is MDS because it uses the piggyback framework and the base code is MDS.

3.7.3 Piggybacking construction (case $r^I \leq r^F$)

The construction in the case when $r^I \leq r^F$ is similar. In this case, we set $\alpha := \lambda^F r^F$, and $\beta_1 := (\lambda^F r^F - r^I)$ and $\beta_2 := \lambda^F r^F$. We divide each symbol evenly into λ^F blocks of r^F columns. Thus, for a given $\ell \in [\alpha]$ we define (ℓ_1, ℓ_2) as follows.

$$\ell_1 := \left\lceil \frac{\ell}{r^F} \right\rceil,$$
$$\ell_2 := (\ell - 1 \bmod r^F) + 1.$$

Now we describe the construction. Figure 3.11 shows a diagram for this construction.

1. *Permutation*: In the initial code, the λ^F blocks of the data symbols associated with final codeword i are circularly shifted to the right $i - 1$ times (denoted with letters **A**-**C**). This reordering is logical (no data is moved) and used for describing the code only.

2. *Projection*: For the first r^I columns of each block, we use the base code without modification to encode each data column. During conversion, we download the data in the first r^I columns of blocks $\underbrace{\{2, \ldots, \lambda^F\}}_{\text{piggyback}}$ from each data symbol (blocks **B** and **C**) and subtract their interference from the corresponding parity

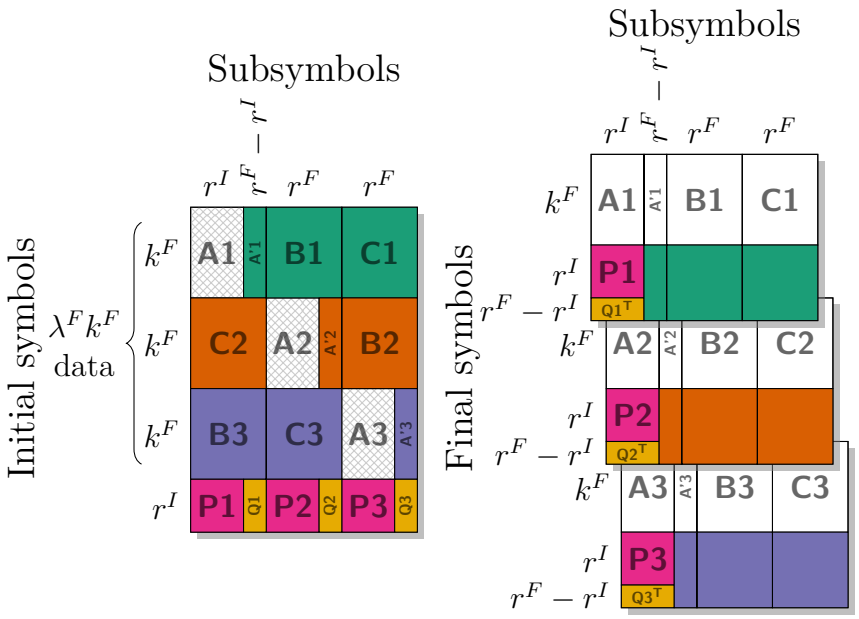


Figure 3.11: Diagram of convertible code construction in the split regime when $r^I \leq r^F$ and $\lambda^F = 3$.

symbols to obtain the first r^I columns of the first block of parities 1 through r^I in each final codeword (**P**) blocks).

3. *Piggybacks*: For columns $(r^I + 1)$ through r^F of each block (**Q** blocks), we use the base code and add piggybacks that contain the subsymbols of the first r^I columns of the first block of parities (r^I+1) through r^F in each final codeword ℓ_1 (transposed). During conversion, we download all the data in the corresponding columns (blocks **A'**, **B** and **C**) and recover the piggybacks.

The remaining parity subsymbols are generated from the accessed data blocks (**A'**, **B**, and **C**). Finally, parity symbol $t \in [r^F]$ is scaled by $\xi_t^{-(i-1)k^F}$. Let $\overrightarrow{\ell}(i)$ and $\mathbf{p}_{t,\ell}^{(i)}$ be defined as before (in the case $r^I > r^F$). Then, the encoding vectors for the initial and final codes are defined as:

$$\mathbf{q}^{I}_{t,\ell} := \begin{cases} \sum_{i=1}^{\lambda^F} \mathbf{p}^{(i)}_{t,\overrightarrow{\ell}_{(i)}} & \text{if } \ell_2 \leq r^I, \\ \sum_{i=1}^{\lambda^F} \mathbf{p}^{(i)}_{t,\overrightarrow{\ell}_{(i)}} + \underbrace{\mathbf{p}^{(\ell_1)}_{\ell_2,t}}_{\text{piggyback}} & \text{otherwise.} \end{cases}$$

$$\mathbf{q}^{F(i)}_{t,\ell} := \underbrace{\xi_t^{-(i-1)k^F}}_{\text{scaling factor}} \mathbf{p}^{(i)}_{t,\ell}$$

Notice that during conversion we only need to download $\beta_1 = (\lambda^F r^F - r^I)$ subsymbols from each unchanged symbol and $\beta_2 = \lambda^F r^F$ subsymbols from each retired symbol, out of the $\alpha = \lambda^F r^F$ subsymbols in each symbol. Therefore, this construction achieves the conversion bandwidth bound of Theorem 3.15. Furthermore, the constructed code is MDS because it uses the piggyback framework and the base code is MDS.

Chapter 4

Locally repairable convertible codes

This chapter is based on work from [111], done in collaboration with K. V. Rashmi.

The previous two chapters of this thesis focus on conversion of *maximum-distance-separable* (MDS) codes from length n^I and dimension k^I to n^F and k^F, respectively. Recently, there has been increased interest in *wide codes*, i.e. codes with large k, as they can achieve lower storage overhead given a target level of failure tolerance. One important drawback of wide codes is that even if a single node becomes unavailable, one must incur high resource-costs to repair it. For example, in the case of an MDS code, one must read k different nodes and reconstruct the original data to repair a node. In practice, repair operations are common enough that those costs negatively affect the performance of the cluster [12, 13, 112]. *Locally repairable codes* (LRCs) [70, 113] mitigate this problem by encoding data in a way that allows nodes to be repaired by accessing $r \ll k$ nodes only.

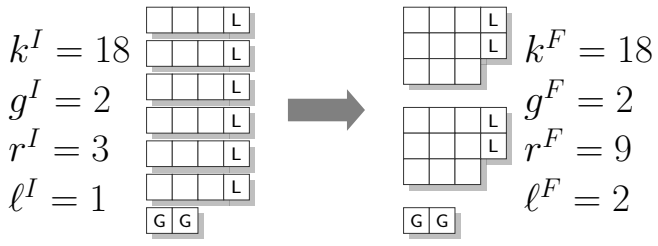


Figure 4.1: Example of LRC conversion. Empty boxes are message symbols. L and G are local and global parities respectively.

To change the optimal repair properties over time, we study the code conversion problem for LRCs (see Figure 4.1). This chapter focuses on codes with (r, ℓ) *data locality*, where k data nodes are divided into groups of size r, each with ℓ *local parities* that are a function of those r data nodes only. In addition, the code has g *global parities* which are a function of all k data nodes. We focus on LRCs with *optimal distance* [65]. As the cost of conversions, we consider *conversion bandwidth*, defined as the total amount of data communicated between nodes during conversion. Our contribution is a new construction technique for LRCs with efficient conversion. This technique can be applied to different types of conversions: in this chapter we focus on *global conversions*, which only change k and g. Even though it is possible to do this type of conversion with existing constructions for MDS codes [101, 102], the constructions presented in this chapter are able to further reduce conversion bandwidth by using both local and global parities. E.g., our construction achieves the conversion of (k, g, r, ℓ) from $(40, 2, 10, 2)$ to $(20, 3, 10, 2)$ with 17.89% less conversion bandwidth than existing constructions [102].

4.1 Background and related work

Let $[i] \coloneqq \{1, \ldots, i\}$. Let $\mathbf{v}_{[i,j]}$ denote entries i through j of a vector $\mathbf{v}$. A linear $[n, k, d, \alpha]$ *vector code* $\mathcal{C}$ over finite field $\mathbb{F}$ is a linear subspace of $\mathbb{F}^{\alpha n}$ of dimension αk. We refer to each coordinate (an element of $\mathbb{F}$) as a *symbol*. A codeword $\mathbf{c} \in \mathcal{C}$ is divided into n *nodes* $\mathbf{c}_i \coloneqq (c_{i,j})_{j=1}^{\alpha}$ $(i \in [n])$. The minimum distance of $\mathcal{C}$ is d, and it is defined as the minimum Hamming distance over $\mathbb{F}^{\alpha}$ between distinct codewords in $\mathcal{C}$. The code $\mathcal{C}$ is said to be MDS if $d = n - k + 1$ (in which case d is omitted). Data $\mathbf{m} \in \mathbb{F}^{\alpha k}$ is encoded via a $\alpha k \times \alpha n$ generator matrix $\mathbf{G}$ as $\mathbf{c} = \mathbf{m}\mathbf{G}$. As an abuse of notation, we denote the encoding of $\mathbf{m}$ under $\mathcal{C}$ as $\mathcal{C}(\mathbf{m})$. Code $\mathcal{C}$ is said to be *systematic* if $\mathbf{c}_i = \mathbf{m}_i \coloneqq (m_{i,j})_{j=1}^{\alpha}$ for $i \in [k]$. The *support* of a code symbol is the set of data symbols corresponding to the non-zero indices in its generator matrix column; the support of a node is the union of the supports of its symbols.

A systematic code $\mathcal{C}$ is said to have (r, ℓ) *data locality* if for each data node $\mathbf{c}_i$ there

exists a set of indices $\Gamma(i)$ containing i such that $|\Gamma(i)| \leq r + \ell$ and the restriction of $\mathcal{C}$ to $\Gamma(i)$ has minimum distance at least $\ell + 1$. Prior work [114] has shown that a code with (r, ℓ) data locality satisfies:

$$d \leq n - k + 1 - \ell\left(\left\lceil \frac{k}{r} \right\rceil - 1\right). \tag{4.1}$$

In this chapter, we consider codes defined by parameters (k, g, r, ℓ), denoting a $[n, k, d, \alpha]$ vector code with $n \coloneqq k + \left\lceil \frac{k\ell}{r} \right\rceil + g$, having (r, ℓ) data locality, and minimum distance d satisfying (4.1) with equality (i.e. optimal distance); we assume $r \mid k$ and treat α as a free variable. The constructions that we present are systematic codes with the following structure: the code has $m \coloneqq \frac{k}{r}$ disjoint *local groups* each with r data nodes and ℓ local parity nodes, and g additional global parity nodes.

4.1.1 Systematic Vandermonde code

A systematic Vandermonde code is an $[n, k, d, \alpha{=}1]$ code defined by a generator matrix that is the concatenation of a $k \times k$ identity matrix and a $k \times (n - k)$ Vandermonde matrix with evaluation points $(\xi_i)_{i=1}^{n-k}$. If the field is large enough, choosing $\xi_i \coloneqq \theta^{i-1}$, where θ is a primitive element, guarantees the MDS property (construction in [7, §V]). Column i of a Vandermonde matrix has the following property: consider a subvector and scale it by a power of ξ_i; this is equivalent to shifting the subvector by i entries. In particular, let $k \coloneqq \lambda t$, and $\mathbf{h}^{(i)} \coloneqq (\xi_i^{j-1})_{j=1}^{k}$ be the i-th encoding vector; then $\mathbf{h}^{(i)}_{[(m-1)t+1,mt]} = \xi_i^{(m-1)t}\mathbf{h}^{(i)}_{[1,t]}$ for all $i \in [n-k]$ and $m \in [\lambda]$.

4.1.2 Basic pyramid code [113]

One method for constructing a code with (r, ℓ) data locality and optimal distance is to start with a $[k + \ell + g, k, \alpha]$ MDS systematic linear code $\mathcal{C}$. Then, the generator matrix column of local parity $j \in [\ell]$ in local group $i \in [\frac{k}{r}]$ is constructed by taking the column of parity j in $\mathcal{C}$, and setting all the entries outside of rows $\{(j-1)r+1, \ldots, jr\}$ to 0.

4.1.3 Piggybacking framework [63]

The *piggybacking framework* constructs an $[n, k, d, \alpha]$ vector code, by using α instances of an $[n, k, d]$ *base code* and adding special functions (called *piggybacks*) to certain symbols. I.e. symbol $c_{i,j}$ is the encoding of $(m_{i,j})_{i=1}^{k}$ under the base code, plus an specially designed piggyback. We refer to the non-piggyback part of a symbol as the *base*. A piggybacked code must have a *decoding order* for the instances of the base code given by a permutation $\sigma : [\alpha] \to [\alpha]$. To satisfy σ, the piggybacking functions used in instance i can only use data from instance j if $\sigma(i) > \sigma(j)$. Thus, when decoding by the order σ, the already-decoded instances are used to remove piggybacks, and the bodies are decoded with the base code. The utility of piggybacks is that they can store useful information which can be retrieved by subtracting the base.

4.1.4 Other related work

Codes designed to have small localities were first proposed in [113, 115], and a bound on the minimum distance of LRCs was proved in [65]. LRCs have been the subject of a wide range of works [66, 68–71, 73–81, 112–114, 116], which has proposed constructions, bounds on field size, and stronger recoverability properties than optimal distance (such as *maximum recoverability*).

The general problem of code conversion was introduced in [7]. Several works [7, 91, 92, 99, 101, 102] have proposed constructions for code conversion. The results in these works consider two types of cost (access cost and conversion bandwidth) and focus on constructions and lower bounds for code conversions in which both the initial and final codes are MDS.

To the best of our knowledge, the idea of converting between different LRCs was first considered in [93] (called *up/downcoding*). Xia et al. [93] propose a conversion procedure for converting between two specific LRCs with different r parameter (and constant $\ell = 1, k, g$). This conversion procedure can be viewed as reducing the number of nodes read during conversion (i.e. *access cost* [7]). In this chapter, we focus on reducing conversion bandwidth instead. Minimization of conversion bandwidth for

MDS codes was studied in [101, 102].

Recently, [90, 117–119] studied LRC conversion (also called *scaling*) in a *clustered* setting, where code symbols are placed in clusters with the goal of reducing inter-cluster communication and satisfying some fault-tolerance constraints. The present chapter is, to the best of our knowledge, the first one to focus on LRC conversion bandwidth (i.e. inter-node communication).

4.2 Conversion of LRCs

We study the LRC conversion from initial parameters (k^I, g^I, r^I, ℓ^I) to final parameters (k^F, g^F, r^F, ℓ^F). Conversion is carried by a *converter* which reads data from nodes, computes new symbols, and writes them. Cost is measured as *conversion bandwidth* [101]: the total amount of data communicated to and from the converter. We focus on reducing *read* conversion bandwidth (i.e. number of symbols read), since the number of symbols written is fixed. We denote read conversion bandwidth as γ, and normalize it as $\tilde{\gamma} \coloneqq \gamma/\alpha$. Codes must satisfy:

P1) the initial $[n^I, k^I, \sigma_{\text{id}}, \alpha]$ code has (r^I, ℓ^I) data locality and optimal distance σ_{id},

P2) the final $[n^F, k^F, d^F, \alpha]$ code has (r^F, ℓ^F) data locality and optimal distance d^F,

P3) there is a conversion procedure from initial code to final code that is efficient in conversion bandwidth.

As in the code conversion literature [7], k is changed by considering $M \coloneqq \text{lcm}(k^I, k^F)$ data nodes evenly divided among $\lambda^I \coloneqq \frac{M}{k^I}$ codewords in the initial code and $\lambda^F \coloneqq \frac{M}{k^F}$ codewords in the final code. Our approach is to construct a code with the piggybacking framework and using the piggybacks to reduce conversion bandwidth. As the base code, we use a basic pyramid code derived from a systematic Vandermonde code, which guarantees optimal distance. Our constructions combine a small number of techniques, which simplifies their description and analysis. We start by presenting two running examples used throughout the chapter to illustrate our techniques. Details will be made clear as we explain our construction approach.

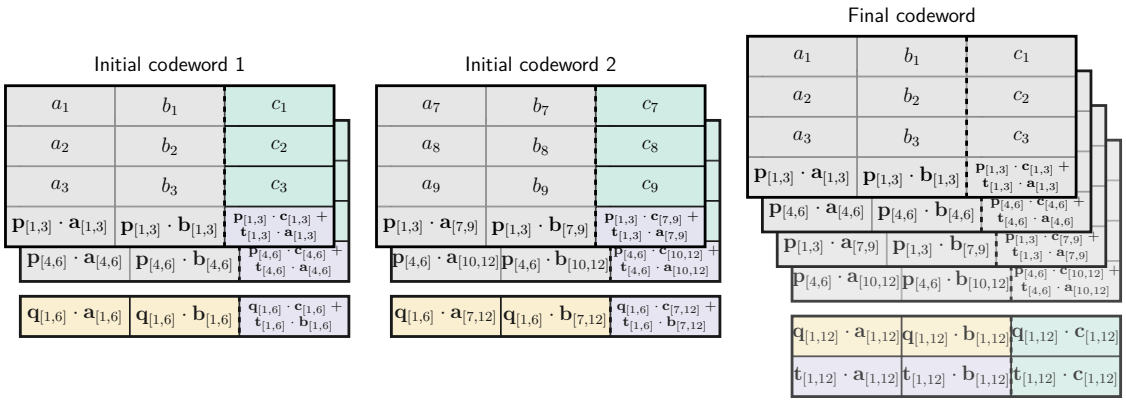


Figure 4.2: (Example 4.1) Example of global merge conversion with parameters $k^I = 6$, $k^F = 12$, $g^I = 1$, $g^F = 2$, $r = 3$, $\ell = 1$.

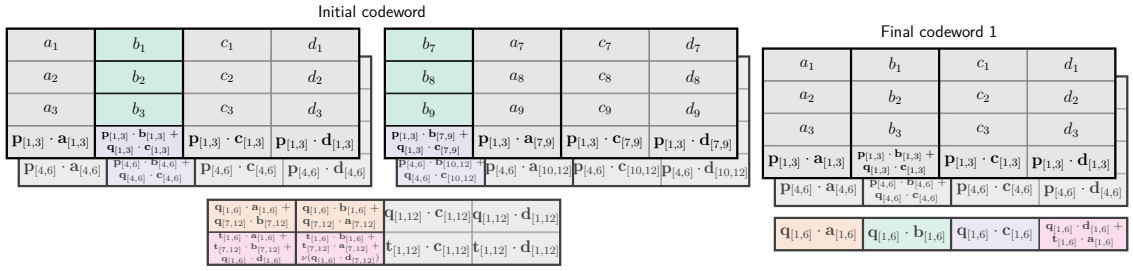


Figure 4.3: (Example 4.2) Example of global split conversion with parameters $k^I = 12$, $k^F = 6$, $g^I = 2$, $g^F = 1$, $r = 3$, $\ell = 1$. In the code, $\nu := \xi_3^6$. For compactness, only one final codeword is shown; the other final codeword has the same encoding.

Example 4.1: Figure 4.2 shows an example of conversion from $(k^I{=}6, g^I{=}1, r^I{=}3, \ell^I{=}1)$ to $(k^F{=}12, g^F{=}2, r^F{=}3, \ell^F{=}1)$. We refer to this type of conversion as a global merge conversion. In the example, data corresponds to $(\mathbf{a}, \mathbf{b}, \mathbf{c})$, and the encoding vectors of the base code are $\mathbf{p}$ (local parity) and $(\mathbf{q}, \mathbf{t})$ (global parities). The non-gray symbols in the initial codewords are read and used in generating the colored symbols in the final codeword (where colors denote techniques that will be described later). Conversion uses the property of Vandermonde codes that, e.g, $\mathbf{p}_{[4,6]} = \xi_1^3 \mathbf{p}_{[1,3]}$. The decoding order in the initial and final codes is $(1, 2, 3)$. By using this construction, conversion requires $\tilde{\gamma} = 7\frac{1}{3}$, compared to 12 (default approach) or 8 (MDS code in [101]). ▸

Example 4.2: Figure 4.3 shows conversion from $(k^I{=}12, g^I{=}2, r^I{=}3, \ell^I{=}1)$ to $(k^F{=}6, g^F{=}1, r^F{=}3, \ell^F{=}1)$. We refer to this type of conversion as a global split conversion. As in the previous example, data corresponds to $(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d})$, encoding vectors are $\mathbf{p}$ (local parity) and $(\mathbf{q}, \mathbf{t})$ (global parities), and non-gray symbols in the initial codeword are read and used in generating the non-gray symbols in the final codewords. The decoding order is $(3, 4, 1, 2)$ in the initial code, and $(3, 1, 4, 2)$ in the final code. Conversion requires $\tilde{\gamma} = 5$, compared to 12 (default approach) or $5\frac{1}{3}$ (MDS code in [102]). ▸

These examples show that it is possible to reduce conversion bandwidth compared to other approaches. Now, we describe our general approach in detail.

4.2.1 Base code

Let $\tilde{k} \coloneqq \max\{k^I, k^F\}$ and $\tilde{g} \coloneqq \max\{\ell^I + g^I, \ell^F + g^F\}$. First, we construct a systematic Vandermonde MDS code $\mathcal{C}$ (Section 4.1.1) of length $\tilde{k} + \tilde{g}$ and dimension $\tilde{k}$. Then, we shorten and puncture $\mathcal{C}$ by removing the last $\tilde{k} - k^I$ rows, the last $\tilde{k} - k^I$ data columns, and the last $\tilde{g} - \ell^I - g^I$ parity columns from the generator matrix to obtain $\mathcal{C}^{\tilde{I}}$. Finally, we derive the initial base code as a basic pyramid code (Section 4.1.2) of $\mathcal{C}^{\tilde{I}}$ (and likewise for the final base code).

4.2.2 Conversion techniques

For ease of exposition, we first present the techniques that will be used in designing conversion-bandwidth efficient codes:

Direct computation (DC). A final parity symbol is computed from the data symbols in its support. E.g., this is used in Example 4.1 to compute $\mathbf{q}_{[1,12]} \cdot \mathbf{c}_{[1,12]}$ from $\mathbf{c}$.

Projection (Pr). A final parity symbol with support S' is computed from an initial parity symbol with support $S \supsetneq S'$ and data symbols in $S \setminus S'$. E.g., used in Example 4.2 to compute $\mathbf{q}_{[1,6]} \cdot \mathbf{a}_{[1,6]}$ from $(\mathbf{q}_{[1,6]} \cdot \mathbf{a}_{[1,6]} + \mathbf{q}_{[7,12]} \cdot \mathbf{b}_{[7,12]})$ and $\mathbf{b}$.

Piggybacks (Pb). A final parity symbol for instance $j \in [\alpha]$ is stored as a piggyback on an initial parity symbol of instance $i \in [\alpha]$ such that $\sigma(i) > \sigma(j)$. The piggyback is recovered by computing and subtracting the base of the initial parity using the data in instance i. E.g., this is used in Example 4.1 to compute $\mathbf{t}_{[1,12]} \cdot \mathbf{b}_{[1,12]}$ from $(\mathbf{q}_{[1,6]} \cdot \mathbf{c}_{[1,6]} + \mathbf{t}_{[1,6]} \cdot \mathbf{b}_{[1,6]})$, $(\mathbf{q}_{[1,6]} \cdot \mathbf{c}_{[7,12]} + \mathbf{t}_{[1,6]} \cdot \mathbf{b}_{[7,12]})$, and $\mathbf{c}$.

Projected piggybacks (PP). A final parity symbol for instance $j \in [\alpha]$ is stored as a piggyback on an initial parity symbol of instance $i \in [\alpha]$ with $\sigma(i) > \sigma(j)$. The base of the initial parity symbol (with support S) is projected using the data in a subset $S' \subsetneq S$; the remaining part (with support $S \setminus S'$) becomes a piggyback in the final parity symbol. In the final code, i and j are swapped in the decoding order. E.g., this is used in Example 4.2 to compute $(\mathbf{q}_{[1,6]} \cdot \mathbf{d}_{[1,6]} + \mathbf{t}_{[1,6]} \cdot \mathbf{a}_{[1,6]})$ from $(\mathbf{t}_{[1,6]} \cdot \mathbf{a}_{[1,6]} + \mathbf{t}_{[7,12]} \cdot \mathbf{b}_{[7,12]} + \mathbf{q}_{[1,6]} \cdot \mathbf{d}_{[1,6]})$ and $\mathbf{b}$.

Linear combination (LC). A final parity symbol with support T is computed as a linear combination of symbols with support S_i such that $T = \bigcup_i S_i$. The linear combination is determined by the base code. E.g., this is used in Example 4.1 to compute $\mathbf{q}_{[1,12]} \cdot \mathbf{a}_{[1,12]}$ from $\mathbf{q}_{[1,6]} \cdot \mathbf{a}_{[1,6]}$ and $\mathbf{q}_{[1,6]} \cdot \mathbf{a}_{[7,12]}$.

Instance reassignment (IR). During conversion, the data symbols associated to data node $i \in [k]$ are reassigned to instances via some permutation $\pi_i : [\alpha] \to [\alpha]$. That is, data in the final code is interpreted as $\mathbf{m}'_i = (m_{i,\pi_i(j)})_{j=1}^{\alpha}$ for $i \in [k]$. This reassignment affects the supports of parities, but it does not modify data nodes. E.g., this is used in Example 4.2 to exchange $\mathbf{a}$ and $\mathbf{b}$ during conversion in some nodes.

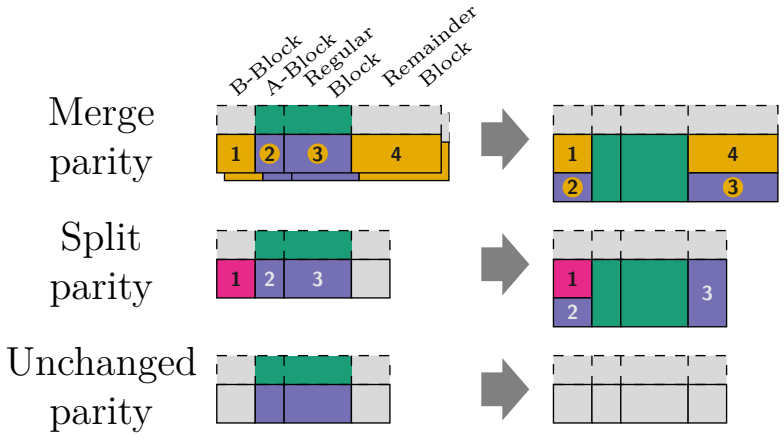


Figure 4.4: Parity designs. Data is shown with a dashed box; parities with a solid box. Parities have one special block (B-block then A-block), a regular block, and a remainder block. Numbers indicate how initial parity symbols are used in the final parities.

We denote linear combination of multiple piggybacks as Pb+LC, e.g., as used in the piggybacks of local parities in both examples. In diagrams, we denote the use of IR with letters, and use the following colors to distinguish the other techniques:

DC Pr Pb PP LC Pb+LC

4.2.3 General strategy

As the output of conversion, the converter constructs new parity nodes, called *target parities*. Target parities are grouped into s sets, such that parity nodes that have the same support are in the same set. Data nodes are divided into s disjoint batches of equal size, corresponding to the supports of the s sets of target parities. In other words, target parities in set i are in the span of batch i ($i \in [s]$). E.g., in Example 4.1, there is single target parity and s=1 set., while in Example 4.2 there are two final parities (one in each final codeword) and thus s=2 sets.

The α instances are divided into s blocks of size B, plus a remainder block of size R (i.e. $\alpha := sB + R$), where s, B, and R are positive integers set depending on the

type of conversion. E.g., in Example 4.1, B=3 and R=0, while in Example 4.2 B=1 and R=2.

We refer to block $i \in [s]$ of nodes in batch i as a *special block*, and to blocks $j \neq i \in [s]$ as *regular blocks*. Special blocks are divided into two sub-blocks: an *accessed sub-block* (A-block) of size E and an *unaccessed sub-block* (B-block) of size $B - E$. In initial parity nodes, block $i \in [s]$ is special if its support and the data in batch i (i.e. data in a special block i) have a non-empty intersection; otherwise, the block is regular. Notice that for each $i \in [s]$, there is a single batch whose nodes have block i as special. In particular, when $s = 1$, all nodes have a single special block, and no regular blocks. E.g., in Example 4.1, each node has a single block (special) and E=1. In Example 4.2, each node has one regular, special, and remainder block; the special block corresponds to $\mathbf{b}$ and E=0.

4.2.4 Design of parities and conversion

We describe three types of parity design: *merge parities*, *split parities*, and *unchanged parities* (see Figure 4.4). In each design, we describe the techniques associated which each symbol.

During conversion, for each batch, the converter downloads all symbols in regular blocks and A-blocks of data nodes (i.e. B-blocks and remainder blocks are not read). In addition, the converter downloads symbols from initial parities and uses them as specified by the parity type. To ensure the final code has optimal distance, each initial parity symbol is used in constructing at most one final parity symbol (which avoids linear dependencies that reduce distance). Thus, we assign at most one technique to each initial parity symbol. In addition, piggybacks in local parities must be a function of data in their local group, and piggybacks in global parities must be a functions of the data in their codeword.

In all parity types, A-blocks and regular blocks are designed the same way: these blocks use Pb or Pb+LC. For symbols in these blocks, all data in their supports is read during conversion, and so piggybacks in them can be recovered. Piggybacks in A-blocks are chosen as parity symbols of instances in the corresponding B-blocks;

piggybacks from regular blocks are chosen as parity symbols of instances in the remainder block. Target parity symbols that are a function of data in A-blocks or regular blocks use DC.

Merge parities: This design is used for parities whose support is a strict subset of the support of a target parity. E.g., in Example 4.1 the initial global parities are merge parities. When $\sigma_{\text{id}} \geq d^F$, the B-block and remainder block of target parities can be fully constructed via LC of initial parity symbols in the respective blocks. Otherwise, we use LC to construct the B-block and remainder block of some target parities, and use Pb or Pb+LC from A-blocks and regular blocks for other target parities.

Split parities: This design is used for parities whose support is a strict superset of the support of a target parity. E.g., in Example 4.2 the initial global parities are split parities. The remainder block of split parities is unused. When $\sigma_{\text{id}} \geq d^F$, then the B-block of target parities can be fully constructed via Pr of split parities in B-blocks. If $\sigma_{\text{id}} > d^F$, the rest of the initial parity symbols in B-blocks use PP to construct final parity symbols in a remainder block. When $\sigma_{\text{id}} < d^F$, the whole B-block of split parities uses Pr. The rest of the final parity symbols in the B-block use Pb from A-blocks.

Unchanged parities: Both B-blocks and remainder blocks are unused. E.g., in both examples local parities are unchanged parities. This type of parity can be kept in the final code.

4.2.5 Instance reassignment

In conversions where the number of codewords increases, we have to ensure that final codewords use the same code. Otherwise, systems would need to keep extra metadata for each codeword, which induces extra complexity and overhead. The template described so far does not meet this requirement:we use IR to correct this.

Let $\text{batch}(i) \coloneqq \left\lfloor \frac{(i-1)s}{k^I} \right\rfloor$. For data node i, we use permutation:

$$\pi_i(j) \coloneqq \begin{cases} ((j - \text{batch}(i)B - 1) \bmod sB) + 1, & \text{if } j \leq sB, \\ j, & \text{otherwise.} \end{cases}$$

Theorem 4.1. *The construction template presented in this section yields codes satisfying properties P1–3.*

Proof. By construction, the initial and final base codes satisfy P1 and P2. Therefore, we must first show that after adding the piggybacks, the codes retain these properties. Then, we must show that the construction template indeed describes a conversion procedure from the initial code to the final code (P3).

For the first part, it suffices to show that there are valid decoding orders for the initial and final code. This, and the restriction that the support of a piggyback must be contained in the support of its node, ensure that P1 and P2 still hold. This is because the decoding order can be used to remove piggybacks both in local decoding and global decoding. Given the design of piggybacks, the following is always a decoding order in the initial code: remainder block, B-blocks, A-blocks. The decoding order in the final code is different only when projected piggybacks are used: in this case, the projected piggybacks (which are part of the remainder block) are decoded after the B-blocks. Notice that IR has no impact the validity of this order, as the relative order of blocks does not matter (as long as the B-blocks are decoded before the corresponding A-blocks).

For the second part, we note that, by construction, the amount of downloaded symbols is sufficient for generating the target parities. Similarly, the design of initial parities is chosen so that generated symbols have the same support as the target parities. Therefore, we must only prove that the different techniques are capable of producing the required final symbols:

DC) Can construct any arbitrary function of the data.

Pb) Given the support of the piggyback, the piggybacking functions are arbitrary, and can thus construct any final symbol.

Pr) When this technique is used, an initial parity symbol is projected onto the space spanned by a single batch. By the properties of systematic Vandermonde codes (Section 4.1.1) and basic pyramid codes (Section 4.1.2), the projected symbol can be scaled to obtain the symbol of the target parity.

LC) Similar to the Pr case, the properties of the systematic Vandermonde base code make it possible to obtain the symbols of the target parity by linear combination.

PP) This case follows from the Pr and Pb cases. Notice that, as a consequence of Pr, a scaling factor might be applied to the piggyback during conversion. However, since the piggyback is arbitrary, the inverse of the scaling factor can be pre-applied to the piggyback, in order to obtain the desired piggyback after conversion.

Pb+LC) Follows directly from case Pb and LC.

□

4.3 Conversion of global parameters

In this section, we describe constructions where both k and g vary, with r and ℓ constant. These conversions are useful to alter the *durability* of the code. In particular, we explore two types: *global merge conversions*, which combine multiple codewords into one; and *global split conversions*, which divide one codeword into multiple. In both types, g changes arbitrarily.

One way to achieve these conversions is to ignore local parities, and use existing constructions for MDS codes [101, 102]. The new constructions also use local parities in conversion, and thus can reduce the conversion bandwidth compared to previous constructions.

4.3.1 Global merge conversion

In global merge conversions, $\lambda^I \geq 2$ codewords are merged into one, i.e. $k^F = \lambda^I k^I$. Local parities are designed as unchanged parities, and global parities as merge parities.

Theorem 4.2. *The construction presented in this section achieves the following conversion bandwidth:*

$$\tilde{\gamma} = \begin{cases} \lambda^I g^F, & \text{if } g^F \leq g^I, \\ \lambda^I \left(\frac{(k^I + m^I \ell)(g^F - g^I)}{g^F + \ell} + g^I \right), & \text{otherwise.} \end{cases}$$

⊡

We present the proof for this theorem after describing the construction. It is worth noting that this construction generalizes the MDS construction ($\ell = 0$).

Case $g^F \leq g^I$: Conversion is carried out using only global parities, as in the MDS case [101].

Case $g^F > g^I$: In this construction (see Figure 4.5), we set:

$$s = 1, \qquad B = g^F + \ell, \qquad R = 0, \qquad E = g^F - g^I.$$

During conversion, LC is used in the global parities to construct symbols in the first g^I final global parities. The rest of the final symbols are constructed via Pb, Pb+LC, or DC.

Proof of Theorem 4.2. In the case where $g^F \leq g^I$, only g^F global parities from each of the λ^I initial codewords need to be read, and the final global parities are computed via LC. Thus, $\gamma = \lambda^I g^F$.

When $g^F > g^I$, we download the full g^I parities from each of the λ^I initial codewords, and E symbols from data nodes and local parities. When normalized, the conversion bandwidth from global parities is 1 from each of the $\lambda^I g^I$ global parities, and $\frac{g^F - g^I}{g^F + \ell}$ from each of the $\lambda^I k^I$ data nodes and $\lambda^I m^I \ell$ local parity nodes. □

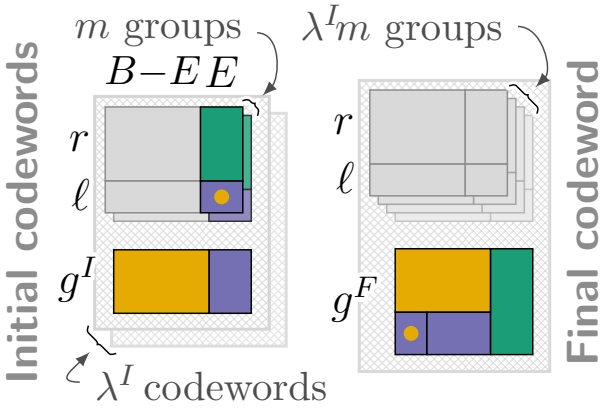


Figure 4.5: Global merge conversion ($r = \frac{k^I}{2}$, $\lambda^I = 2$, and $g^I < g^F$).

4.3.2 Global split conversion

In global split conversions, a single initial codeword is split into $\lambda^F \geq 2$, i.e. $k^I = \lambda^F k^F$. Local parities are designed as unchanged parities, and global parities as split parities.

Theorem 4.3. *The construction presented in this section achieves the following conversion bandwidth:*

$$\tilde{\gamma} = \begin{cases} \lambda^F g^F \frac{(\lambda^F-1)(k^F+m^F\ell)+g^I}{(\lambda^F-1)g^F+g^I+\ell(\lambda^F-1)}, & \text{if } g^F \leq g^I, \\ \frac{\lambda^F g^F\left((k^F+m^F\ell)(\lambda^F g^F-g^I)+g^I g^F\right)}{\lambda^F g^F(g^F+\ell)-g^I\ell}, & \text{otherwise.} \end{cases}$$

⊡

We present the proof for this theorem after describing the construction. It is worth noting that this construction generalizes the MDS construction ($\ell = 0$).

Case $g^I \geq g^F$: Variables are set as follows (see Figure 4.6):

$$s = \lambda^F, \quad B = g^F, \quad R = \ell(\lambda^F - 1) + g^I - g^F, \quad E = 0.$$

The first g^F global parities use Pr to construct symbols in the final global parities; the remaining initial global parities use PP to construct symbols in remainder blocks. Local parities use Pb+LC to construct symbols from remainder blocks.

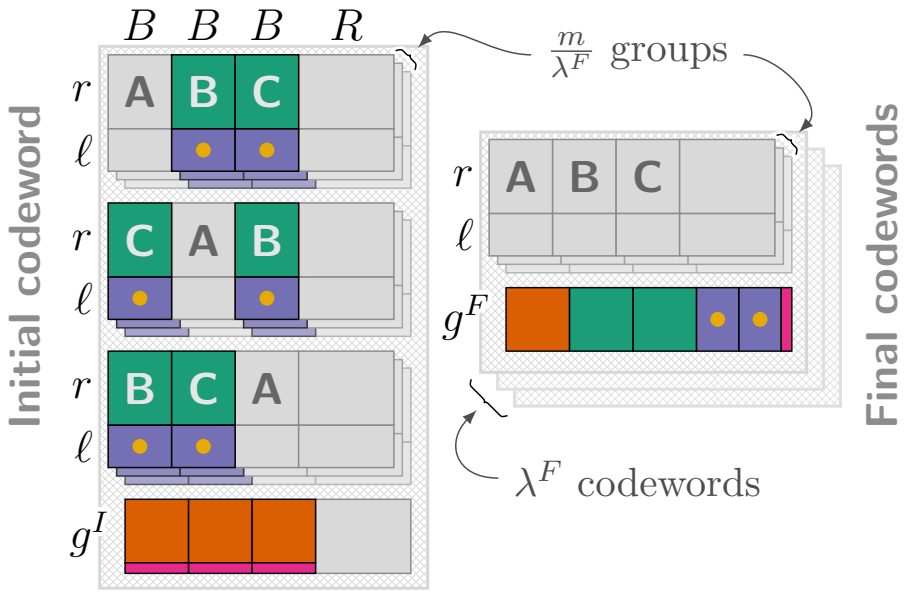


Figure 4.6: Global split conversion ($r = \frac{k^I}{9}$, $\lambda^F = 3$, and $g^I > g^F$).

Case $g^I < g^F$: We set the construction variables as follows:

$$s = \lambda^F, \;\; B = (g^F)^2, \;\; R = \lambda^F \ell g^F - \ell g^I, \;\; E = g^F(g^F - g^I).$$

During conversion, initial global parities use Pr to construct symbols for the B-blocks of the first g^I global parities in each final codeword. The rest of the symbols are constructed via Pb and Pb+LC from the A-blocks. The remainder block of final global parities is constructed via Pb+LC on local parities.

Proof of Theorem 4.3. In the case where $g^F \leq g^I$, we download sB symbols from each global parity node, and $(s-1)B + E$ from each data and local parity. The size of each node is $\alpha = sB + R$. In terms of conversion bandwidth, this is

$$\frac{sB}{sB + R} = \frac{\lambda^F g^F}{(\lambda^F - 1)(\ell + g^F) + g^I}$$

from each of the g^I global parities, and

$$\frac{(s-1)B+E}{sB+R} = \frac{(\lambda^F-1)g^F}{(\lambda^F-1)(\ell+g^F)+g^I}$$

from each of the $\lambda^F k^F$ data nodes, and each of the $\lambda^F m^F \ell$ local parities. With some arithmetic manipulation, this yields the amount in the theorem.

In the case where $g^F > g^I$, we download sB from each global parity node, and $(s-1)B+E$ from data and local parity nodes. The size of each node is $\alpha = sB+R$. In terms of conversion bandwidth, this is

$$\frac{sB}{sB+R} = \frac{\lambda^F (g^F)^2}{\lambda^F g^F (g^F+\ell) - g^I \ell}$$

from each of the g^I global parities, and

$$\frac{(s-1)B+E}{sB+R} = \frac{(\lambda^F-1)\lambda^F (g^F)^2 + g^F(g^F-g^I)}{\lambda^F g^F (g^F+\ell) - g^I \ell}$$

from each of the $\lambda^F k^F$ data nodes, and each of the $\lambda^F m^F \ell$ local parities. With some arithmetic manipulation, this yields the amount in the theorem. □

Chapter 5

Designing distributed storage systems for code conversion

This chapter is based on work from [15], done in collaboration with Saurabh Kadekodi, Suhas Jayaram Subramanya, Juncheng Yang, K. V. Rashmi, and Gregory R. Ganger; and [120], done in collaboration with Saurabh Kadekodi, Sanjith Athlur, Arif Merchant, K. V. Rashmi and Gregory R. Ganger.

So far, our work in this thesis has concentrated on designing erasure codes to make code conversion more efficient. However, the design of a fully-fledged distributed storage system encompasses much more than the erasure codes. Commonly used distributed storage systems (such as HDFS [11]) are not designed with code conversion in mind: it needs to be manually performed by the user, and it can only be performed by reading, re-encoding, and rewriting the data. In this chapter, we propose two novel designs for distributed storage systems which automatically adapt to changes in disk failure rates using code conversion. By adapting in this way, these systems are able to achieve lower storage overhead than a classical distributed storage system without compromising reliability.

The first system of this kind to be proposed was *HeART* (Heterogeneity-Aware Redundancy Tuner) [1]. The main feature of HeART is its ability to automatically detect changes in device failure rates and re-encode data based on that. However, the main disadvantage of HeART is that the work that results from re-encoding the data

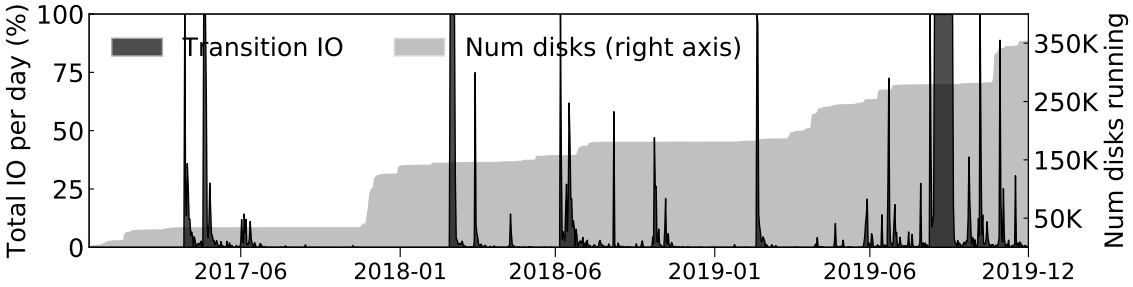
(i.e. reading, re-computing parities, and writing them) can overwhelm the cluster for long periods of time.

The first system that we propose, *Pacemaker*, solves this challenge by dividing disks into groups with similar failure rates, and then placing data strictly within groups. By doing this, Pacemaker can monitor the failure rate trends of disks and proactively re-encode data so as to avoid large bursts of IO usage. However, in real systems, data placement is already very constrained by a lot of factors. Thus, it is not ideal to have additional placement constraints imposed by the system. The second system that we propose, *Tiger*, does not impose any additional constraints on data placement. Instead, it uses more advanced reliability models that take failure rate heterogeneity into account. This allows Tiger to monitor the reliability of data on a finer grain. This fine-grain monitoring, coupled with the diversity that results from unconstrained placement, naturally results into more gradual re-encodings without large bursts of work.

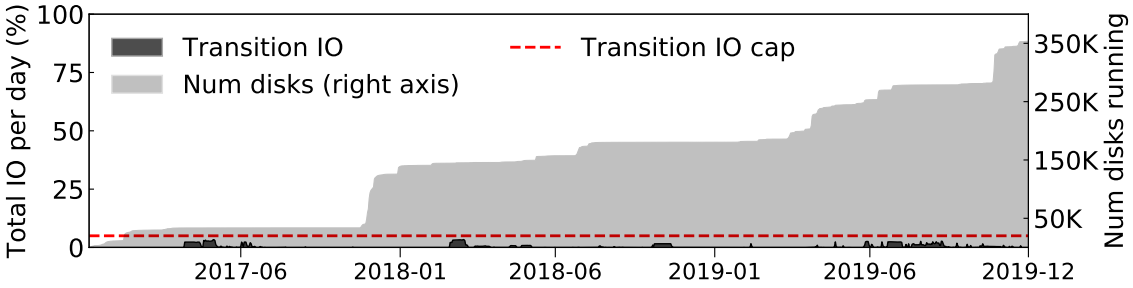
5.1 Pacemaker: avoiding HeART attacks in storage clusters

Distributed storage systems use data redundancy to protect data in the face of disk failures [8, 121, 122]. While it provides resilience, redundancy imposes significant cost overhead. Most large-scale systems today erasure code most of the data stored, instead of replicating, which helps to reduce the space overhead well below 100% [10, 112, 121, 123–125]. Despite this, space overhead remains a key concern in large-scale systems since it directly translates to an increase in the number of disks and the associated increase in capital, operating and energy costs [10, 112, 121, 125].

Storage clusters are made up of disks from a mix of makes/models acquired over time, and different makes/models have highly varying failure rates [1, 126, 127]. Despite that, storage clusters employ a "one-size-fits-all-disks" approach to choosing redundancy levels, without considering failure rate differences among disks. Hence, space overhead is often inflated by overly conservative redundancy levels, chosen to



(a) Transition IO for HeART [1] on Google Cluster1.



(b) Transition IO for PACEMAKER on Google Cluster1.

Figure 5.1: Fraction of total cluster IO bandwidth needed to use disk-adaptive redundancy for a Google storage cluster's first three years. The state-of-the-art proposal [1] shown in (a) would require up to 100% of the cluster bandwidth for extended periods, whereas PACEMAKER shown in (b) always fits its IO under a cap (5%). The light gray region shows the disk count (right Y-axis) over time.

ensure sufficient protection for the most failure-prone disks in the cluster. Although tempting, the overhead cannot be removed by using very "wide" codes (which can provide high reliability with low storage overhead) for all data, due to the prohibitive reconstruction cost induced by the most failure-prone disks (more details in Section 5.2). An exciting alternative is to dynamically adapt redundancy choices to observed failure rates (AFRs)[1] for different disks, which recent proposals suggest could substantially reduce the space overhead [1].

Adapting redundancy involves dynamic transitioning of redundancy schemes, because AFRs must be learned from observation of deployed disks and because AFRs change over time due to disk aging. Changing already encoded data from one redundancy scheme to another, for example from an erasure code with parameters

[1]AFR describes the expected fraction of disks that experience failure in a typical year.

k_1-of-n_1 to k_2-of-n_2 (where k-of-n denotes k data chunks and $n-k$ parity chunks; more details in Section 5.2), can be exorbitantly IO intensive. Existing designs for disk-adaptive redundancy are rendered unusable by overwhelming bursts of urgent transition IO when applied to real-world storage clusters. Indeed, as illustrated in Figure 5.1a, our analyses of production traces show extended periods of needing 100% of the cluster's IO bandwidth for transitions. We refer to this as the *transition overload* problem. At its core, transition overload occurs whenever an observed AFR increase for a subset of disks requires too much urgent transition IO in order to keep data safe. Existing designs for disk-adaptive redundancy perform redundancy transitions as a reaction to AFR changes. Since prior designs are reactive, for an increase in AFR, the data is already under-protected by the time the transition to increase redundancy is issued. And it will continue to be under-protected until that transition completes. For example, around 2019-09 in Figure 5.1a, data was under-protected for over a month, even though the entire cluster's IO bandwidth was used solely for redundancy transitions. Simple rate-limiting to reduce urgent bursts of IO would only exacerbate this problem causing data-reliability goals to be violated for even longer.

To understand the causes of transition overload and inform solutions, we analyse multi-year deployment and failure logs for over 5.3 million disks from Google, NetApp and Backblaze. Two common transition overload patterns are observed. First, sometimes disks are added in tens or hundreds over time, which we call *trickle* deployments. A statistically confident AFR observation requires thousands of disks. Thus, by the time it is known that AFR for a specific make/model and age is too high for the redundancy used, the oldest thousands of that make/model will be past that age. At that point, all of those disks need immediate transition. Second, sometimes disks are added in batches of many thousands, which we call *step* deployments. Steps have sufficient disks for statistically confident AFR estimation. However, when a step reaches an age where the AFR is too high for the redundancy used, *all* disks of the step need immediate transition.

In this chapter we introduce PACEMAKER, a new disk-adaptive redundancy orchestration system that exploits insights from the aforementioned analyses to eliminate

the transition overload problem. PACEMAKER proactively organizes data layouts to enable efficient transitions for each deployment pattern, reducing total transition IO by over 90%. Indeed, by virtue of its reduced total transition IO, PACEMAKER can afford to use extra transitions to reap increased space-savings. PACEMAKER also proactively initiates anticipated transitions sufficiently in advance that the resulting transition IO can be rate-limited without placing data at risk. Figure 5.1b provides a peek into the final result: PACEMAKER achieves disk-adaptive redundancy with substantially less total transition IO and never exceeds a specified transition IO cap (5% in the graph).

We evaluate PACEMAKER using logs containing all disk deployment, failure, and decommissioning events from four production storage clusters: three 160K–450K-disk Google clusters and a ≈110K-disk cluster used for the Backblaze Internet backup service [128]. On all four clusters, PACEMAKER provides disk-adaptive redundancy while using less than 0.4% of cluster IO bandwidth for transitions on average, and never exceeding the specified rate limit (e.g., 5%) on IO bandwidth. Yet, despite its proactive approach, PACEMAKER loses less than 3% of the space-savings as compared to to an idealized system with perfectly-timed and instant transitions. Specifically, PACEMAKER provides 14–20% average space-savings compared to a one-size-fits-all-disks approach, without ever failing to meet the target data reliability and with no transition overload. We note that this is substantial savings for large-scale systems, where even a single-digit space-savings is worth the engineering effort. For example, in aggregate, the four clusters would need ≈200K fewer disks.

We also implement PACEMAKER in HDFS, demonstrating that PACEMAKER's mechanisms fit into an existing cluster storage system with minimal changes. Complementing our longitudinal evaluation using traces from large scale clusters, we report measurements of redundancy transitions in PACEMAKER-enhanced HDFS via small-scale cluster experiments. A prototype of HDFS with Pacemaker is open-sourced and is available at `https://github.com/thesys-lab/pacemaker-hdfs.git`.

The first part of this chapter (Sections 5.1 to 5.9) is dedicated to PACEMAKER, and it makes five primary contributions. First, it demonstrates that transition overload is a roadblock that precludes use of previous disk-adaptive redundancy proposals. Second,

it presents insights into the sources of transition overload from longitudinal analyses of deployment and failure logs for 5.3 million disks from three large organizations. Third, it describes PACEMAKER's novel techniques, designed based on insights drawn from these analyses, for safe disk-adaptive redundancy without transition overload. Fourth, it evaluates PACEMAKER's policies for four large real-world storage clusters, demonstrating their effectiveness for a range of deployment and disk failure patterns. Fifth, it describes integration of and experiments with PACEMAKER's techniques in HDFS, demonstrating their feasibility, functionality, and ease of integration into a cluster storage implementation.

5.2 Whither disk-adaptive redundancy

Cluster storage systems and data reliability. Modern storage clusters scale to huge capacities by combining up to hundreds of thousands of storage devices into a single storage system [8, 122, 129]. In general, there is a metadata service that tracks data locations (and other metadata) and a large number of storage servers that each have up to tens of disks. Data is partitioned into chunks that are spread across storage servers/devices. Although hot/warm data is now often stored on Flash SSDs, cost considerations lead to the majority of data continuing to be stored on mechanical disks (HDDs) for the foreseeable future [130–132]. For the rest of the chapter, any reference to a "device" or "disk" implies HDDs.

Disk failures are common and storage clusters use data redundancy to protect against irrecoverable data loss in the face of disk failures [8, 10, 12, 112, 125, 127, 128]. For hot data, often replication is used for performance benefits. But, for most bulk and colder data, cost considerations have led to the use of erasure coding schemes. Under a k-of-n coding scheme, each set of k data chunks are coupled with n-k "parity chunks" to form a "stripe". A k-of-n scheme provides tolerance for up to $(n - k)$ failures with a space overhead of $\frac{n}{k}$. Thus, erasure coding achieves substantially lower space overhead for tolerating a given number of failures. Schemes like 6-of-9 and 10-of-14 are commonly used in real-world deployments [12, 112, 121, 125]. Under

erasure coding, additional work is involved in recovering from a device failure. To reconstruct a lost chunk, k remaining chunks from the stripe must be read.

The redundancy scheme selection problem. The reliability of data stored redundantly is often quantified as *mean-time-to-data-loss* (MTTDL) [133], which essentially captures the average time until more than the tolerated number of chunks are lost. MTTDL is calculated using the disks' AFR and its *mean-time-to-repair* (MTTR).

Large clusters are built over time, and hence usually consist of a mix of disks belonging to multiple makes/models depending on which options were most cost effective at each time. AFR values vary significantly between makes/models and disks of different ages [1, 126, 127, 134]. Since disks have different AFRs, computing MTTDL of a candidate redundancy scheme for a large-scale storage cluster is often difficult.

The MTTDL equations can still be used to guide decisions, as long as a sufficiently high AFR value is used. For example, if the highest AFR value possible for any deployed make/model at any age is used, the computed MTTDL will be a lower bound. So long as the lower bound on MTTDL meets the target MTTDL, the data is adequately reliable. Unfortunately, the range of possible AFR values in a large storage cluster is generally quite large (over an order of magnitude) [1, 126, 127, 135]. Since the overall average is closer to the lower end of the AFR range, the highest AFR value is a conservative over-estimate for most disks. The resulting MTTDLs are thus loose lower bounds, prompting decision-makers to use a one-size-fits-all scheme with excessive redundancy leading to wasted space.

Using wide schemes with large number of parities (e.g., 30-of-36) can achieve the desired MTTDL while keeping the storage overhead low enough to make disk-adaptive redundancy appear not worth the effort. But, while this might seem like a panacea, wide schemes in high-AFR regimes cause significant increase in failure reconstruction IO traffic. The failure reconstruction IO is derived by multiplying the AFR with the number of data chunks in each stripe. Thus, if either of these quantities are excessively high, or both are moderately high, it can lead to overwhelmingly high failure reconstruction IO. In addition, wide schemes also result in higher tail latencies

for individual disk reconstructions because of having to read from many more disks. Combined, these reasons prevent use of wide schemes for all data all the time from being a viable solution for most systems.

Disk-adaptive redundancy. Since the problem arises from using a single AFR value, a promising alternative is to adapt redundancy for subsets of disks with similar AFRs. A recent proposal, heterogeneity-aware redundancy tuner (HeART) [1], suggests treating subsets of deployed disks with different AFR characteristics differently. Specifically, HeART adapts redundancy of each disk by observing its failure rate on-the-fly[2] depending on its make/model and its current age. It is well known that AFR of disks follow a "bathtub" shape with three distinct phases of life: AFR is high in "infancy" (1-3 months), low and stable during its "useful life" (3-5 years), and high during the "wearout" (a few months before decommissioning). HeART uses a default (one-size-fits-all) redundancy scheme for each new disk's infancy. It then dynamically changes the redundancy to a scheme adapted to the observed useful life AFR for that disk's make/model, and then dynamically changes back to the default scheme at the end of useful life. The per-make/model useful life redundancy schemes typically have much lower space overhead than the default scheme. This suggests the ability to maintain target MTTDL with many fewer disks (i.e., lower cost).

Although exciting, the design of HeART overlooks a crucial element: the IO cost associated with changing the redundancy schemes. Changing already encoded data under one erasure code to another can be exorbitantly IO intensive. Indeed, our evaluation of HeART on real-world storage cluster logs reveal extended periods where data safety is at risk and where 100% cluster IO bandwidth is consumed for scheme changes. We call this problem *transition overload*.

An enticing solution that might appear to mitigate transition overload is to adapt redundancy schemes only by removing parities in low-AFR regimes and adding parities in high-AFR regimes. While this solution eliminates transition IO when reducing the level of redundancy, it does only marginally better when redundancy needs to be increased, because new parity creation cannot avoid reading all data chunks from

[2]Although it may be tempting to use AFR values taken from manufacturer's specifications, several studies have shown that failure rates observed in practice often do not match those [127, 134, 135].

each stripe. What makes this worse is that transitions that increase redundancy are time-critical, since delaying them would miss the MTTDL target and leave the data under-protected. Moreover, addition/removal of a parity chunk massively changes the stripe's MTTDL compared to addition/removal of a data chunk. For example, a 6-of-9 MTTDL is 10000× higher than 6-of-8 MTTDL, but is only 1.5× higher than 7-of-10 MTTDL. AFR changes would almost never be large enough to safely remove a parity, given default schemes like 6-of-9, eliminating almost all potential benefits of disk-adaptive redundancy.

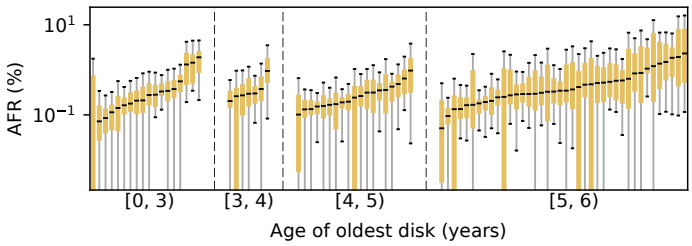
This chapter analyzes disk deployment and failure data from large-scale production clusters to discover sources of *transition overload* and informs the design of a solution. It then describes and evaluates PACEMAKER, which realizes the dream of safe disk-adaptive redundancy without transition overload.

5.3 Longitudinal production trace analyses

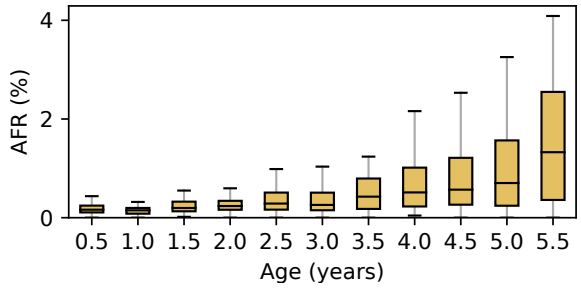
This section presents an analysis of multi-year disk reliability logs and deployment characteristics of 5.3 million HDDs, covering over 60 makes/models from real-world environments. Key insights presented here shed light on the sources of transition overload and challenges/opportunities for a robust disk-adaptive redundancy solution.

The data. Our largest dataset comes from NetApp and contains information about disks deployed in filers (file servers). Each filer reports the health of each disk periodically (typically once a fortnight) using their AutoSupport [136] system. We analyzed the data for a subset of their deployed disks, which included over 50 makes/models and over 4.3 million disks total. As observed in previous studies [1, 127, 134], we observe well over an order of magnitude difference between the highest and lowest useful-life AFRs (see Figure 5.2a).

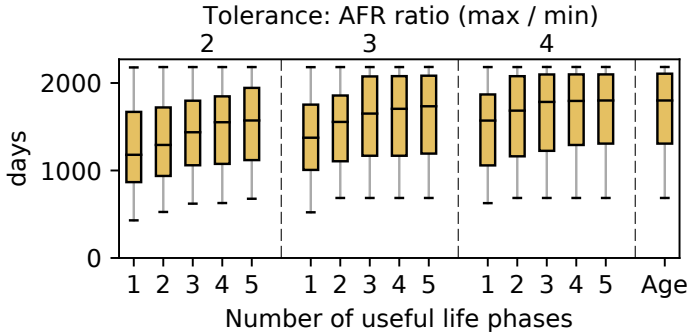
Our other datasets come from large storage clusters deployed at Google and the Backblaze Internet backup service. Although the basic disk characteristics (e.g., AFR heterogeneity and its behavior discussed below) are similar to the NetApp dataset, these datasets also capture the evolution and behavior in our target context



(a) Spread of make/model AFRs



(b) AFR distribution over disk life



(c) Approximate useful-life length

Figure 5.2: (a) AFR spread for over 50 makes/models from NetApp binned by the age of the oldest disk. Each box corresponds to a unique make/model, and at least 10000 disks of each make/model were observed (outlier AFR values omitted). (b) Distribution of AFR calculated over consecutive non-overlapping six-month periods for NetApp disks, showing the gradual rise of AFR with age (outliers omitted). (c) Approximation of useful life length for NetApp disks for 1-5 consecutive phases of useful life and three different tolerance levels.

(large-scale storage clusters), and thus are also used in the evaluation detailed in (Section 5.7). The particular Google clusters were selected based on their longitudinal data availability, but were not otherwise screened for favorability.

For each cluster, the multi-year log records (daily) all disk deployment, failure, and decommissioning events from birth of the cluster until the date of the log snapshot. Google Cluster1's disk population over three years included ≈350K disks of 7 makes/models. Google Cluster2's population over 2.5 years included ≈450K disks of 4 makes/models. Google Cluster3's population over 3 years included ≈160K disks of 3 makes/models. The Backblaze cluster's population since 2013 included ≈110K disks of 7 makes/models.

5.3.1 Causes of transition overload

Disk deployment patterns. We observe disk deployments occurring in two distinct patterns, which we label ***trickle*** and ***step***. Trickle-deployed disks are added to a cluster frequently (weekly or even daily) over time by the tens and hundreds. For example, the slow rise in disk count seen between 2018-01 and 2018-07 in Figure 5.1 represents a series of trickle-deployments. In contrast, a step-deployment introduces many thousands of disks into the cluster "at once" (over a span of a few days), followed by potentially months of no new step-deployments. The sharp rises in disk count around 2017-12 and 2019-11 in Figure 5.1 represent step-deployments.

A given cluster may be entirely trickle-deployed (like the Backblaze cluster), entirely step-deployed (like Google Cluster2), or a mix of the two (like Google Cluster1 and Cluster3). Disks of a step are typically of the same make/model.

Learning AFR curves online. Disk-adaptive redundancy involves learning the AFR curve for each make/model by observing failures among deployed disks of that make/model. Because AFR is a statistical measure, the larger the population of disks observed at a given age, the lower is the uncertainty in the calculated AFR at that age. We have found that a few thousand disks need to be observed to obtain sufficiently accurate AFR measurements.

Transition overload for trickle-deployed disks. Since trickle-deployed disks

are deployed in tiny batches over time, several months can pass before the required number of disks of a new make/model are past any given age. Thus, by the time the required number of disks can be observed at the age that is eventually identified as having too-high an AFR and requiring increased redundancy, data on the older disks will have been left under-protected for months. And, the thousands of already-older disks need to be immediately transitioned to a stronger redundancy scheme, together with the newest disks to reach that age. This results in transition overload.

Transition overload for step-deployed disks. Assuming that they are of the same make/model, a batch of step-deployed disks will have the same age and AFR, and indeed represent a large enough population for confident learning of the AFR curve as they age. But, this means that all of those disks will reach AFR values together, as they age. So, when their AFR rises to the point where the redundancy must be increased to keep data safe, all of the disks must transition together to the new safer redundancy scheme. Worse, if they are the first disks of the given make/model deployed in the cluster, which is often true in the clusters studied, then the system adapting the redundancy will learn of the need only when the age in question is reached. At that point, all data stored on the entire batch of disks is unsafe and needs immediate transitioning. This results in transition overload.

5.3.2 Informing a solution

Analyzing the disk logs has exposed a number of observations that provide hope and guide the design of PACEMAKER. The AFR curves we observed deviate substantially from the canonical representation where infancy and wearout periods are identically looking and have high AFR values, and AFR in useful life is flat and low throughout.

AFRs rise gradually over time with no clear wearout. AFR curves generally exhibit neither a flat useful life phase nor a sudden transition to so-called wearout. Rather, in general, it was observed that AFR curves rise gradually as a function of disk age. Figure 5.2b shows the gradual rise in AFR over six month periods of disk lifetimes. Each box represents the AFR of disks whose age corresponds to the six-month period denoted along the X-axis. AFR curves for individual makes/models (e.g., Figures 5.5b

and 5.5c) are consistent with this aggregate illustration. Importantly, none of the over 60 makes/models from Google, Backblaze and NetApp displayed sudden onset of wearout.

Gradual increases in AFR, rather than sudden onset of wearout, suggests that one could anticipate a step-deployed batch of disks approaching an AFR threshold. This is one foundation on which PACEMAKER's proactive transitioning approach rests.

Useful life could have multiple phases. Given the gradual rise of AFRs, useful life can be decomposed into multiple, piece-wise constant phases. Figure 5.2c shows an approximation of the length of useful life when multiple phases are considered. Each box in the figure represents the distribution over different make/models of the approximate length of useful life. Useful life is approximated by considering the longest period of time which can be decomposed into multiple consecutive phases (number of phases indicated by the bottom X-axis) such that the ratio between the maximum and minimum AFR in each phase is under a given tolerance level (indicated by the top X-axis). The last box indicates the distribution over make/models of the age of the oldest disk, which is an upper bound to the length of useful life. As shown by Figure 5.2c, the length of useful life can be significantly extended (for all tolerance levels) by considering more than one phase. Furthermore, the data show that a small number of phases suffice in practice, as the approximate length of useful life changes by little when considering four or more phases.

Infancy often short-lived. Disks may go through (potentially) multiple rounds of so-called "burn-in" testing. The first tests may happen at the manufacturer's site. There may be additional burn-in tests done at the deployment site allowing most of the infant mortality to be captured before the disk is deployed in production. For the NetApp and Google disks, we see the AFR drop sharply and plateau by 20 days for most of the makes/models. In contrast, the Backblaze disks display a slightly longer and higher AFR during infancy, which can be directly attributed to their less aggressive on-site burn-in.

PACEMAKER's design is heavily influenced from these learnings, as will be explained in the next section.

5.4 Design goals of PACEMAKER

PACEMAKER is an IO efficient redundancy orchestrator for storage clusters that support disk-adaptive redundancy. Before going into the design goals for PACEMAKER, we first chronicle a disk's lifecycle, introducing the terminology that will be used in the rest of the chapter (defined in Table 5.1).

Disk lifecycle under PACEMAKER. Throughout its life, each disk under PACEMAKER simultaneously belongs to a *Dgroup* and an *Rgroup*. There are as many Dgroups in a cluster as there are unique disk makes/models. Rgroups on the other hand are a function of redundancy schemes and placement restrictions. Each Rgroup has an associated redundancy scheme, and its data (encoded stripes) must reside completely within that Rgroup's disks. Multiple Rgroups can use the same redundancy scheme, but no stripe may span across Rgroups. The Dgroup of a disk never changes, but a disk may transition through multiple Rgroups during its lifetime. At the time of deployment (or "birth"), the disk belongs to *Rgroup0*, and is termed as an *unspecialized disk*. Disks in Rgroup0 use the default redundancy scheme, i.e. the conservative one-scheme-fits-all scheme used in storage clusters that do not have disk-adaptive redundancy. The redundancy scheme employed for a disk (and hence its Rgroup) changes via *transitions*. The first transition any disk undergoes is an *RDn transition*. A RDn transition changes the disk's Rgroup to one with lower redundancy, i.e. more optimized for space. Whenever the disk departs from Rgroup0, it is termed as a *specialized disk*. Disks depart from Rgroup0 at the end of their infancy. Since infancy is short-lived (Section 5.3.2), PACEMAKER only considers one RDn transition for each disk.

The first RDn transition occurs at the start of the disk's useful life, and marks the start of its specialization period. As explained in Section 5.3.2, a disk may experience multiple useful life phases. PACEMAKER performs a transition at the start of each useful life phase. After the first (and only) RDn transition, each subsequent transition is an *RUp transition*. An RUp transition changes the disk's Rgroup to one with higher redundancy, i.e. less optimized for space, but the disk is still considered a specialized disk unless the Rgroup that the disk is being RUp transitioned to is Rgroup0. The

Term	*Definition*
Dgroup	Group of disks of the same make/model.
Transition	The act of changing the redundancy scheme.
RDn transition	Transition to a lower level of redundancy.
RUp transition	Transition to a higher level of redundancy.
peak-IO-cap	IO bandwidth cap for transitions.
Rgroup	Group of disks using the same redundancy with placement restricted to the group of disks.
Rgroup0	Rgroup using the default one-scheme-fits-all redundancy used in storage clusters today.
Unspecialized disks	Disks that are a part of Rgroup0.
Specialized disks	Disks that are not part of Rgroup0.
Canary disks	First few thousand disks of a trickle-deployed Dgroup used to learn AFR curve.
Tolerated-AFR	Max AFR for which redundancy scheme meets reliability constraint.
Threshold-AFR	The AFR threshold crossing which triggers an RUp transition for step-deployed disks.

Table 5.1: Definitions of PACEMAKER's terms.

space-savings (and thus cost-savings) associated with disk-adaptive redundancy are proportional to the fraction of life the disks remain specialized for.

Key decisions. To adapt redundancy throughout a disk's lifecycle as chronicled above, three key decisions related to transitions must be made

1. ***When should the disks transition?***

2. ***Which Rgroup should the disks transition to?***

3. ***How should the disks transition?***

Constraints. The above decisions need to be taken such that a set of constraints are met. An obvious constraint, central to any storage system, is that of data reliability. The *reliability constraint* mandates that all data must always meet a predefined target MTTDL. Another important constraint is the *failure reconstruction IO constraint*. This constraint bounds the IO spent on data reconstruction of failed disks, which as explained in Section 5.2 is proportional to AFR and scheme width.

This is why wide schemes cannot be used for all disks all the time, but they can be used for low-AFR regimes of disk lifetimes (as discussed in Section 5.2).

Existing approaches to disk-adaptive redundancy make their decisions on the basis of only these constraints [1], but fail to consider the equally important *IO caused by redundancy transitions*. Ignoring this causes the transition overload problem, which proves to be a show-stopper for disk-adaptive redundancy systems. PACEMAKER treats transition IO as a first class citizen by taking it into account for each of its three key decisions. As such, PACEMAKER enforces carefully designed constraints on transition IO as well.

Designing IO constraints on transitions. Apart from serving foreground IO requests, a storage cluster performs numerous background tasks like scrubbing and load balancing [137–139]. Redundancy management is also a background task. In current storage clusters, redundancy management tasks predominantly consist of performing data redundancy (e.g. replicating or encoding data) and reconstructing data of failed or otherwise unavailable disks. Disk-adaptive redundancy systems add redundancy transitions to the list of IO-intensive background tasks.

There are two goals for background tasks: Goal 1: they are not too much work, and Goal 2: they interfere as little as possible with foreground IO. PACEMAKER applies two IO constraints on background transition tasks to achieve these goals: (1) *average-IO constraint* and (2) *peak-IO constraint*. The average-IO constraint achieves Goal 1 by allowing storage administrators to specify a cap on the fraction of the IO bandwidth of a disk that can be used for transitions over its lifetime. For example, if a disk can transition in 1 day using 100% of its IO bandwidth, then an average-IO constraint of 1% would mean that the disk will transition at most once every 100 days. The peak-IO constraint achieves Goal 2 by allowing storage administrators to specify the peak rate (defined as the *peak-IO-cap*) at which transitions can occur so as to limit their interference with foreground traffic. Continuing the previous example, if the peak-IO-cap is set at 5%, the disk that would have taken 1 day to transition at 100% IO bandwidth would now take at least 20 days. The average-IO constraint and the peak-IO-cap can be configured based on how busy the cluster is. For example, a cluster designed for data archival would have a lower foreground traffic, compared

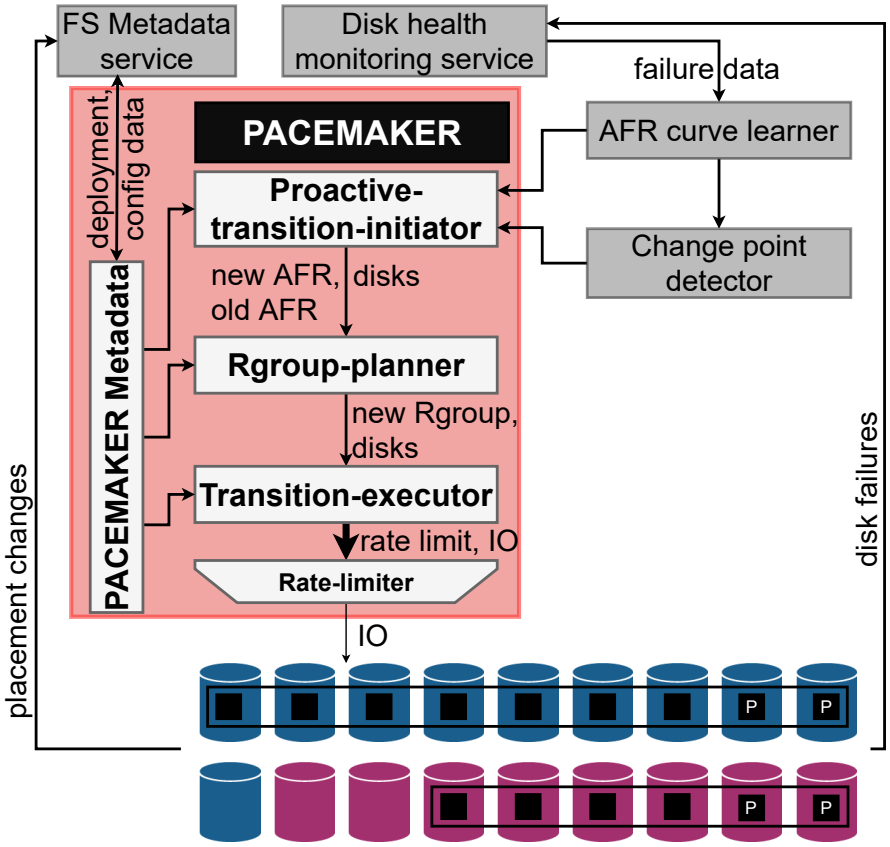


Figure 5.3: PACEMAKER architecture.

to a cluster designed for serving ads or recommendations. Thus, low-traffic clusters can set a higher peak-IO-cap resulting in faster transitions and potentially increased space-savings.

Design goals. The key design goals are to answer the three questions related to transitions such that the space-savings are maximized and the following constraints are met: (1) reliability constraint on all data all the time, (2) failure reconstruction IO constraint on all disks all the time, (3) peak-IO constraint on all disks all the time, and (4) average-IO constraint on all disks over time.

5.5 Design of PACEMAKER

Figure 5.3 shows the high level architecture of PACEMAKER and how it interacts with some other components of a storage cluster. The three main components of PACEMAKER correspond to the three key decisions that the system makes as discussed in Section 5.4. The first main component of PACEMAKER is the ***proactive-transition-initiator*** (Section 5.5.1), which determines when to transition disks using the AFR curves and the disk deployment information. The information of the transitioning disks and their observed AFR is passed to the ***Rgroup-planner*** (Section 5.5.2), which chooses the Rgroup to which the disks should transition. The Rgroup-planner passes the information of the transitioning disks and the target Rgroup to the ***transition-executor*** (Section 5.5.3). The transition-executor addresses how to transition the disks to the planned Rgroup in the most IO-efficient way.

Additionally, PACEMAKER also maintains its own ***metadata*** and a simple ***rate-limiter***. PACEMAKER metadata interacts with all of PACEMAKER's components and also the storage cluster's metadata service. It maintains various configuration settings of a PACEMAKER installation along with the disk deployment information that guides transition decisions. The rate-limiter rate-limits the IO load generated by any transition as per administrator specified limits. Other cluster components external-to-PACEMAKER that inform it are the *AFR curve learner* and the *change point detector*. As is evident from their names, these components learn the AFR curve[3] of each Dgroup and identify change points for redundancy transitions. The AFR curve learner receives failure data from the *disk health monitoring service*, which monitors the disk fleet and maintains their vitals.

5.5.1 Proactive-transition-initiator

Proactive-transition-initiator's role is to determine *when to transition the disks*. Below we explain PACEMAKER's methodology for making this decision for the two types of transitions (RDn and RUp) and the two types of deployments (step and trickle).

[3]The AFR estimation methodology employed is detailed in Section 5.8.

Deciding when to RDn a disk

Recall that a disk's first transition is an RDn transition. As soon as proactive-transition-initiator observes (in a statistically accurate manner) that the AFR has decreased sufficiently, and is stable, it performs an RDn transition from the default scheme (i.e., from Rgroup0) employed in infancy to a more space-efficient scheme. This is the only RDn transition in a disk's lifetime.

Deciding when to RUp a disk

RUp transitions are performed either when there are too few disks in any Rgroup such that data placement is heavily restricted (which we term *purging an Rgroup*), or when there is a rise in AFR such that the reliability constraint is (going to be) violated. Purging an Rgroup involves RUp transitioning all of its disks to an Rgroup with higher redundancy. This transition isn't an imminent threat to reliability, and therefore can be done in a relaxed manner without violating the reliability constraint as explained in Section 5.5.3.

However, most RUp transitions in a storage cluster are done in response to a rise in AFR. These are challenging with respect to meeting IO constraints due to the associated risk of violating the reliability constraints whenever the AFR rises beyond the AFR tolerated by the redundancy scheme (termed *tolerated-AFR*).

In order to be able to safely rate-limit the IO load due to RUp transitions, PACEMAKER takes a *proactive* approach. The key is in determining when to initiate a proactive RUp transition such that the transition can be completed before the AFR crosses the tolerated-AFR, while adhering to the IO and the reliability constraints without compromising much on space-savings. To do so, the proactive-transition-initiator assumes that its transitions will proceed as per the peak-IO constraint, which is ensured by the transition-executor. PACEMAKER's methodology for determining when to initiate a proactive RUp transition is tailored differently for trickle versus for step deployments, since they raise different challenges.

Trickle deployments. For trickle-deployed disks, PACEMAKER considers two category of disks: (1) first disks to be deployed from any particular trickle-deployed

Dgroup, and (2) disks from that Dgroup that are deployed later.

PACEMAKER labels the first C deployed disks of a Dgroup as *canary* disks, where C is a configurable, high enough number of disks to yield statistically significant AFR observations. For example, based on our disk analyses, we observe that C in low thousands (e.g., 3000) is sufficient. The canary disks of any Dgroup are the first to undergo the various phases of life for that Dgroup, and these observations are used to learn the AFR curve for that Dgroup. The AFR value for the Dgroup at any particular age is not known (with statistical confidence) until all canary disks go past that age. Furthermore, due to the trickle nature of the deployment, the canary disks would themselves have been deployed over weeks if not months. Thus, AFR for the canary disks can be ascertained only in retrospect. PACEMAKER never changes the redundancy of the canary disks to avoid them from ever violating the reliability constraint. This does not significantly reduce space-savings, since C is expected to be small relative to the total number of disks of a Dgroup (usually in the tens of thousands).

The disks that are deployed later in any particular Dgroup are easier to handle, since the Dgroup's AFR curve would have been learned by observing the canaries. Thus, the date at which a disk among the later-deployed disks needs to RUp to meet the reliability constraints is known in advance by the proactive-transition-initiator, which it uses to issue proactive RUp transitions.

Step deployments. Recall that in a step deployment, most disks of a Dgroup may be deployed within a few days. So, canaries are not a good solution, as they would provide little-to-no advance warning about how the AFR curve's rises would affect most disks.

PACEMAKER's approach to handling step-deployments is based on two properties: (1) Step-deployments have a large number of disks deployed together, leading to a statistically accurate AFR estimation; (2) AFR curves based on a large set of disks tend to exhibit gradual, rather than sudden, AFR increases as the disk ages (Section 5.3.2). PACEMAKER leverages these two properties to employ a simple *early warning* methodology to predict a forthcoming need to RUp transition a step well in advance. Specifically, PACEMAKER sets a threshold, termed *threshold-AFR*, which

is a (configurable) fraction of the tolerated-AFR of the current redundancy scheme employed. For step-deployments, when the observed AFR crosses the threshold-AFR, the proactive-transition-initiator initiates a proactive RUp transition.

5.5.2 Rgroup-planner

The Rgroup-planner's role is to determine *which Rgroup should disks transition to.* This involves making two *interdependent* choices: (1) the redundancy scheme to transition into, (2) whether or not to create a new Rgroup.

Choice of the redundancy scheme. At a high level, the Rgroup-planner first uses a set of selection criteria to arrive at a set of viable schemes. It further narrows down the choices by filtering out the schemes that are not worth transitioning to when the transition IO and IO constraints are accounted for.

Selection criteria for viable schemes. Each viable redundancy scheme has to satisfy the following criteria in addition to the reliability constraint: each scheme (1) must satisfy the minimum number of simultaneous failures per stripe (i.e., $n - k$); (2) must not exceed the maximum allowed stripe dimension (k); (3) must have its expected failure reconstruction IO (AFR $\times$ k $\times$ disk-capacity) be no higher than was assumed possible for Rgroup0 (since disks in Rgroup0 are expected to have the highest AFR); (4) must have a recovery time in case of failure (MTTR) that does not exceed the maximum MTTR (set by the administrator when selecting the default redundancy scheme for Rgroup0).

Determining if a scheme is worth transitioning to. Whether the IO cost of transitioning to a scheme is worth it or not and what space-savings can be achieved by that transition is a function of the number of days disks will remain in that scheme (also known as *disk-days*). This, in turn, depends on (1) when the disks enter the new scheme, and (2) how soon disks will require another transition out of that scheme.

The time it takes for the disks to enter the new scheme is determined by the transition IO and the rate-limit. When the disks will transition out of the target Rgroup is dependent on the future and can only be estimated. For this estimation, the Rgroup-planner needs to estimate the number of days the AFR curve will remain

below the threshold that forces a transition out. This needs different strategies for the two deployment patterns (trickle and step).

Recall that PACEMAKER knows the AFR curve for trickle-deployed disks (from the canaries) in advance. Recall that step-deployed disks have the property that the AFR curve learned from them is statistically robust and tends to exhibit gradual, as opposed to sudden AFR increases. The Rgroup-planner leverages these properties to estimate the future AFR behavior based on the recent past. Specifically, it takes the slope of the AFR curve in the recent past[4] and uses that to project the AFR curve rise in the future.

The number of disk-days in a scheme for it to be worth transitioning to is dictated by the IO constraints. For example, let us consider a disk running under PACEMAKER that requires a transition, and PACEMAKER is configured with an average-IO constraint of 1% and a peak-IO-cap of 5%. Suppose the disk requires 1 day to complete its transition at 100% IO bandwidth. With the current settings, PACEMAKER will only consider an Rgroup worthy of transitioning to (assuming it is allowed to use all 5% of its IO bandwidth) if at least 80 disk-days are spent after the disk entirely transitions to it (since transitioning to it would take up to 20 days at the allowed 5% IO bandwidth).

From among the viable schemes that are worth transitioning to based on the IO constraints, the Rgroup-planner chooses the one that provides the highest space-savings.

Decision on Rgroup creation. Rgroups cannot be created arbitrarily. This is because every Rgroup adds *placement restrictions*, since all chunks of a stripe have to be stored on disks belonging to the same Rgroup. Therefore, Rgroup-planner creates a new Rgroup only when (1) the resulting placement pool created by the new Rgroup is large enough to overcome traditional placement restrictions such as "no two chunks on the same rack[5]", and (2) the space-savings achievable by the chosen redundancy scheme is sufficiently greater than using an existing (less-space-efficient) Rgroup.

[4]PACEMAKER uses a 60 day (configurable) sliding window with an Epanechnikov kernel, which gives more weight to AFR changes in the recent past [140].

[5]Inter-cluster fault tolerance remains orthogonal to and unaffected by PACEMAKER.

The disk deployment pattern also affects Rgroup formation. While the rules for whether to form an Rgroup remain the same for trickle and step-deployed disks, mixing disks deployed differently impacts the transitioning techniques that can be used for eventually transitioning disks out of that Rgroup. This in turn affects how the IO constraints are enforced. Specifically, for trickle deployments, creating an Rgroup for each set of transitioning disks would lead to too many small-sized Rgroups. So, for trickle-deployments, the Rgroup-planner creates a new Rgroup for a redundancy scheme if and only if one does not exist already. Creating Rgroups this way will also ensure that enough disks (thousands) will go into it to satisfy placement restrictions. Mixing disks from different trickle-deployments in the same Rgroup does not impact the IO constraints, because PACEMAKER optimizes the transition mechanism for few disks transitioning at a time, as is explained in Section 5.5.3. For step-deployments, due to the large fraction of disks that undergo transition together, having disks from multiple steps, or mixing trickle-deployed disks within the same Rgroup, creates adverse interactions (discussed in Section 5.5.3). Hence, the Rgroup-planner creates a new Rgroup for each step-deployment, even if there already exists one or more Rgroups that employ the chosen scheme. Each such Rgroup will contain many thousands of disks to overcome traditional placement restrictions. Per-step Rgroups also extend to the Rgroup with default redundancy schemes, implying a per-step Rgroup0. Despite having clusters with disk populations as high as 450K disks, PACEMAKER's restrained Rgroup creation led to no cluster ever having more than 10 Rgroups.

Rules for purging an Rgroup. An Rgroup may be purged for having too few disks. This can happen when too many of its constituent disks transition to other Rgroups, or they fail, or they are decommissioned leading to difficulty in fulfilling placement restrictions. If the Rgroup to be purged is made up of trickle-deployed disks, the Rgroup-planner will choose to RUp transition disks to an existing Rgroup with higher redundancy while meeting the IO constraints. For step-deployments, purging implies RUp transitioning disks into the more-failure-tolerant RGroup (RGroup0) that may include trickle-deployed disks.

5.5.3 Transition-executor

The transition-executor's role is to determine *how to transition the disks*. This involves choosing (1) the most IO-efficient technique to execute that transition, and (2) how to rate-limit the transition at hand. Once the transition technique is chosen, the transition-executor executes the transition via the rate-limiter as shown in Figure 5.3.

Selecting the transition technique. Suppose the data needs to be conventionally re-encoded from a k_{cur}-of-n_{cur} scheme to a k_{new}-of-n_{new} scheme. The IO cost of conventional re-encoding involves reading–re-encoding–writing all the stripes whose chunks reside on each transitioning disk. This amounts to a read IO of $k_{cur}\times$disk-capacity (assuming almost-full disks), and a write IO of $k_{cur}\times$disk-capacity$\times\frac{n_{new}}{k_{new}}$ for a total IO $> 2 \times k_{cur}\times$disk-capacity for each disk.

In addition to conventional re-encoding, PACEMAKER supports two new approaches to changing the redundancy scheme for disks and selects the most efficient option for any given transition. The best option depends on the fraction of the Rgroup being transitioned at once.

Type 1 (Transition by emptying disks). If a small percentage of an Rgroup's disks are being transitioned, it is more efficient to retain the contents of the transitioning disks in that Rgroup rather than re-encoding. Under this technique, the data stored on transitioning disks are simply moved (copied) to other disks within the current Rgroup. This involves reading and writing (elsewhere) the contents of the transitioning disks. Thus, the IO of transitioning via Type 1 is at most 2×disk-capacity, independent of scheme parameters, and therefore at least $k_{cur}\times$ cheaper than conventional re-encoding.

Type 1 can be employed whenever there is sufficient free space available to move the contents of the transitioning disks into other disks in the current Rgroup. Once the transitioning disks are empty, they can be removed from the current Rgroup and added to the new Rgroup as "new" (empty) disks.

Type 2 (Bulk transition by recalculating parities). If a large fraction of disks in an Rgroup need to transition together, it is more efficient to transition the entire Rgroup rather than only the disks that need a transition at that time. Most cluster

storage systems use systematic codes[6] [121, 141–143], wherein transitioning an entire Rgroup involves only calculating and storing new parities and deleting the old parities. Specifically, the data chunks have to be only read for computing the new parities, but they do not have to be re-written. In contrast, if only a part of the disks are transitioned, some fraction of the data chunks also need to be re-written. Thus, the IO cost for transitioning via Type 2 involves a read IO of $\frac{k_{cur}}{n_{cur}}\times$disk-capacity, and a write IO of only the new parities, which amounts to a total IO of $\frac{n_{new}-k_{new}}{k_{new}} \times \frac{k_{cur}}{n_{cur}}\times$disk-capacity for each disk in the Rgroup. This is at most $2 \times \frac{k_{cur}}{n_{cur}}\times$disk-capacity, which makes it at least $n_{cur}\times$ cheaper than conventional re-encoding.

Selecting the most efficient approach for a transition. For any given transition, the transition-executor selects the most IO-efficient of all the viable approaches. Almost always, trickle-deployed disks use Type 1 because they transition a-few-at-a-time, and step-deployed disks use Type 2 because Rgroup-planner maintains each step in a separate Rgroup.

Choosing how to rate limit a transition. Irrespective of the transitioning techniques, the transition-executor has to resolve the competing concerns of maximizing space-savings and minimizing risk of data loss via fast transitions, and minimizing foreground work interference by slowing down transitions so as to not overwhelm the foreground IO. Arbitrarily slowing down a transition to minimize interference is only possible when the transition is not in response to a rise in AFR. This is because a rising AFR hints at the data being under-protected if not transitioned to a higher redundancy soon. In PACEMAKER, a transition without an AFR rise occurs either when disks are being RDn transitioned at the end of infancy, or when they are being RUp transitioned because the Rgroup they belong to is being purged. For all the other RUp transitions, PACEMAKER carefully chooses how to rate limit the transition.

Determining how much bandwidth to allow for a given transition could be difficult, given that other transitions may be in-progress already or may be initiated at any time (we do observe concurrent transitions in our evaluations). So, to ensure that the aggregate IO of all ongoing transitions conforms to the peak-IO-cap cluster-wide,

[6]In systematic codes, the data chunks are stored in unencoded form. This helps to avoid having to decode for normal (i.e., non-degraded-mode) reads.

PACEMAKER limits each transition to the peak-IO-cap within its Rgroup. For trickle-deployed disks, which share Rgroups, the rate of transition initiations is consistently a small percentage of the shared Rgroup, allowing disk emptying to proceed at well below the peak-IO-cap. For step-deployed disks, this is easy for PACEMAKER, since a step only makes one transition at a time and its IO is fully contained in its separate Rgroup. The transition-executor's approach to managing peak-IO on a per-Rgroup basis is also why the proactive-transition-initiator can safely assume a rate-limit of the peak-IO-cap without consulting the transition-executor. If there is a sudden AFR increase that puts data at risk, PACEMAKER is designed to ignore its IO constraints to continue meeting the reliability constraint—this safety valve was never needed for any cluster evaluated.

After finalizing the transitioning technique, the transition-executor performs the necessary IO for transitioning disks (read, writes, parity recalculation, etc.). We find that the components required for the transition-executor are already present and adequately modular in existing distributed storage systems. In Section 5.6, we show how we implement PACEMAKER in HDFS with minimal effort.

Note that this design is for the common case where storage clusters are designed for a single dedicated storage service. Multiple distinct distributed storage services independently using the same underlying devices would need to coordinate their use of bandwidth (for their non-transition related load as well) in some way, which is outside the scope of this work.

5.6 Implementation of PACEMAKER in HDFS

We have implemented a prototype of PACEMAKER for the Hadoop distributed file system (HDFS) [122]. HDFS is a popular open source distributed file system, widely employed in the industry for storing large volumes of data. We use HDFS v3.2.0, which natively supports erasure coding. The prototype of HDFS with Pacemaker is open-sourced and is available at https://github.com/thesys-lab/pacemaker-hdfs.git.

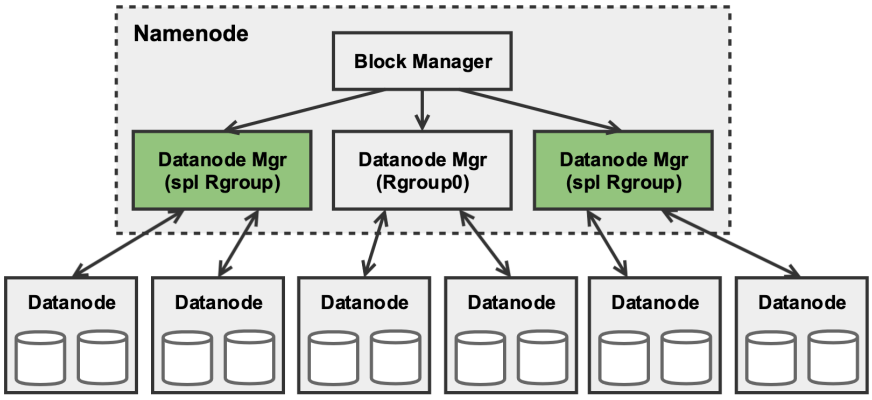


Figure 5.4: PACEMAKER-enhanced HDFS architecture.

Background on HDFS architecture. HDFS has a central metadata server called Namenode (NN, akin to the master node) and a collection of servers containing the data stored in the file system, called Datanodes (DN, akin to worker nodes). Clients interact with the NN only to perform operations on file metadata (containing a collection of the DNs that store the file data). Clients directly request the data from the DNs. Each DN stores data on its local drives using a local file system.

Realizing Rgroups in HDFS. This design makes a simplifying assumption that all disks belonging to a DN are of the same Dgroup and are deployed together (this could be relaxed easily). Under this simplifying assumption, conceptually, an Rgroup would consist of a set of DNs that need to be managed independent of other such sets of DNs as shown in Figure 5.4.

The NN maintains a DatanodeManager (DNMgr), which is a gateway for the NN to interact with the DNs. The DNMgr maintains a list of the DNs, along with their usage statistics. The DNMgr also contains a HeartBeatManager (HrtBtMgr) which handles the periodic keepalive heartbeats from DNs. A natural mechanism to realize Rgroups in HDFS is to have one DNMgr per Rgroup. Note that the sets of DNs belonging to the different DNMgrs are mutually exclusive. Implementing Rgroups with multiple DNMgrs has several advantages.

Right level of control and view of the system. Since the DNMgr resides below the

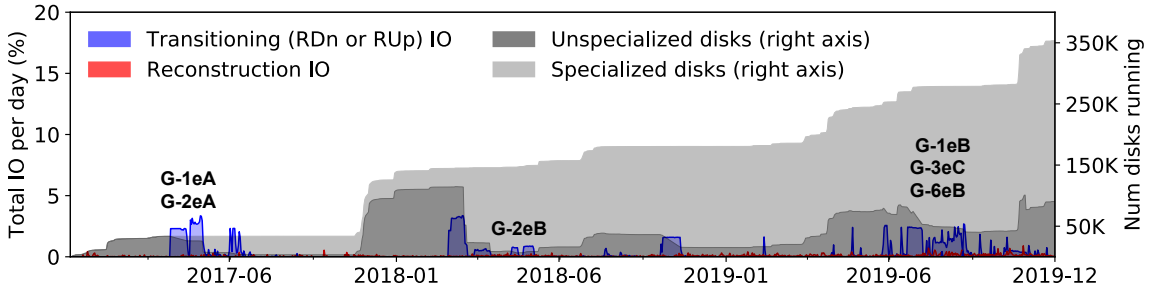
block layer, when the data needs to be moved for redundancy adaptations, the logical view of the file remains unaffected. Only the mapping from HDFS blocks to DNs gets updated in the inode. The statistics maintained by the DNMgr can be used to balance load across Rgroups.

Minimizing changes to the HDFS architecture and maximizing re-purposing of existing HDFS mechanisms. This design obviates the need to change HDFS's block placement policy, since it is implemented at the DNMgr level. Block placement policies are notoriously hard to get right. Moreover, block placement decisions are affected by fault domains and network topologies, both of which are orthogonal to PACEMAKER's goals, and thus best left untouched. Likewise, the code for reconstruction of data from a failed DN need not be touched, since all of the reads (to reconstruct each lost chunk) and writes (to store it somewhere else) will occur within the set of nodes managed by its DNMgr. Existing mechanisms for adding / decommissioning nodes managed by the DNMgr can be re-purposed to implement PACEMAKER's Type 1 transitions (described below).

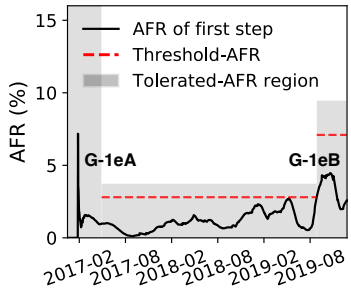
Cost of maintaining multiple DNMgrs is small. Each DNMgr maintains two threads: a HrtBtMgr and a DNAdminMgr. The former tracks and handles heartbeats from each DN, and the latter monitors the DNs for performing decommissioning and maintenance. The number of DNMgr threads in the NN will increase from two to 2× the number of Rgroups. Fortunately, even for large clusters, we observe that the number of Rgroups would not exceed the low tens (Section 5.7.4). The NN is usually a high-end server compared to the DNs, and an additional tens of threads shouldn't affect performance.

Rgroup transitions in HDFS. An important part of PACEMAKER functionality is transitioning DNs between Rgroups. Recall from Section 5.5.3 that one of PACEMAKER's preferred way of transitioning disks across Rgroups is by emptying the disks. In HDFS, the planned removal of a DN from a HDFS cluster is called decommissioning. PACEMAKER re-uses decommissioning to remove a DN from the set of DNs managed by one DNMgr and then adds it to the set managed by another, effectively transitioning a DN from one Rgroup to another.

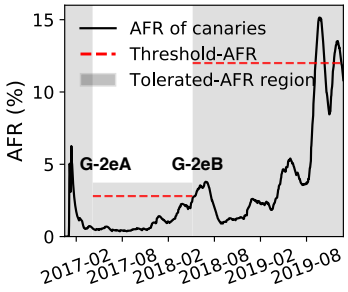
PACEMAKER does not change the file manipulation API or client access paths. But,



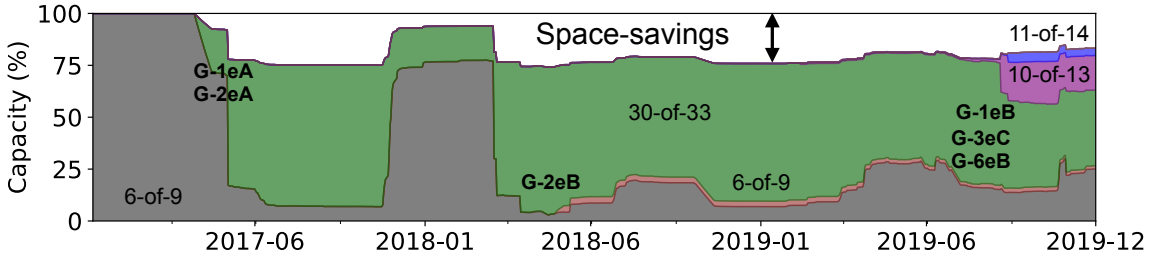
(a) Redundancy management IO due to PACEMAKER over its 2.5+ year lifetime broken down by IO type. This identical to Figure 5.1b with the left Y axis only going to 20% to show the detailed IO activity happening in the cluster.



(b) G-1 (step) AFR curve.



(c) G-2 (trickle) AFR curve.



(d) Space-savings due to PACEMAKER. Each colored region represents the fraction of cluster capacity that is using a particular redundancy scheme. 6-of-9 is the default redundancy scheme (Rgroup0's).

Figure 5.5: Detailed IO analysis and space savings achieved by PACEMAKER-enabled adaptive redundancy on Google Cluster1.

there is one corner-case related to transitions when file reads can be affected internally. To read a file, a client queries the NN for the inode and caches it. Subsequently, the reads are performed directly from the client to the DN. If the DN transitions to another Rgroup while the file is still being read, the HDFS client may find that that DN no longer has the requested data. But, because this design uses existing HDFS decommissioning for transitions, the client software knows to react by re-requesting the updated inode from the NN and resuming the read.

5.7 Evaluation of PACEMAKER

PACEMAKER-enabled disk-adaptive redundancy using is evaluated on production logs from four large-scale real-world storage clusters, each with hundreds of thousands of disks. We also experiment with a proof-of-concept HDFS implementation on a smaller sized cluster. This evaluation has four primary takeaways: (1) PACEMAKER eliminates transition overload, never using more than 5% of cluster IO bandwidth (0.2–0.4% on average) and always meets target MTTDL, in stark contrast to prior work approaches that do not account for transition IO load; (2) PACEMAKER provides more than 97% of idealized-potential space-savings, despite being proactive, reducing disk capacity needed by 14–20% compared to one-size-fits-all; (3) PACEMAKER's behavior is not overly sensitive across a range of values for its configurable parameters; (4) PACEMAKER copes well with the real-world AFR characteristics explained in Section 5.3.2. For example, it successfully combines the "multiple useful life phases" observation with efficient transitioning schemes. This evaluation also shows PACEMAKER in action by measuring disk-adaptive redundancy in PACEMAKER-enhanced HDFS.

Evaluation methodology. PACEMAKER is simulated chronologically for each of the four cluster logs described in Section 5.3: three clusters from Google and one from Backblaze. For each simulated date, the simulator changes the cluster composition according to the disk additions, failures and decommissioning events in the log. PACEMAKER is provided the log information, as though it were being captured live in the cluster. IO bandwidth needed for each day's redundancy management is

computed as the sum of IO for failure reconstruction and transition IO requested by PACEMAKER, and is reported as a fraction of the configured cluster IO bandwidth (100MB/sec per disk, by default).

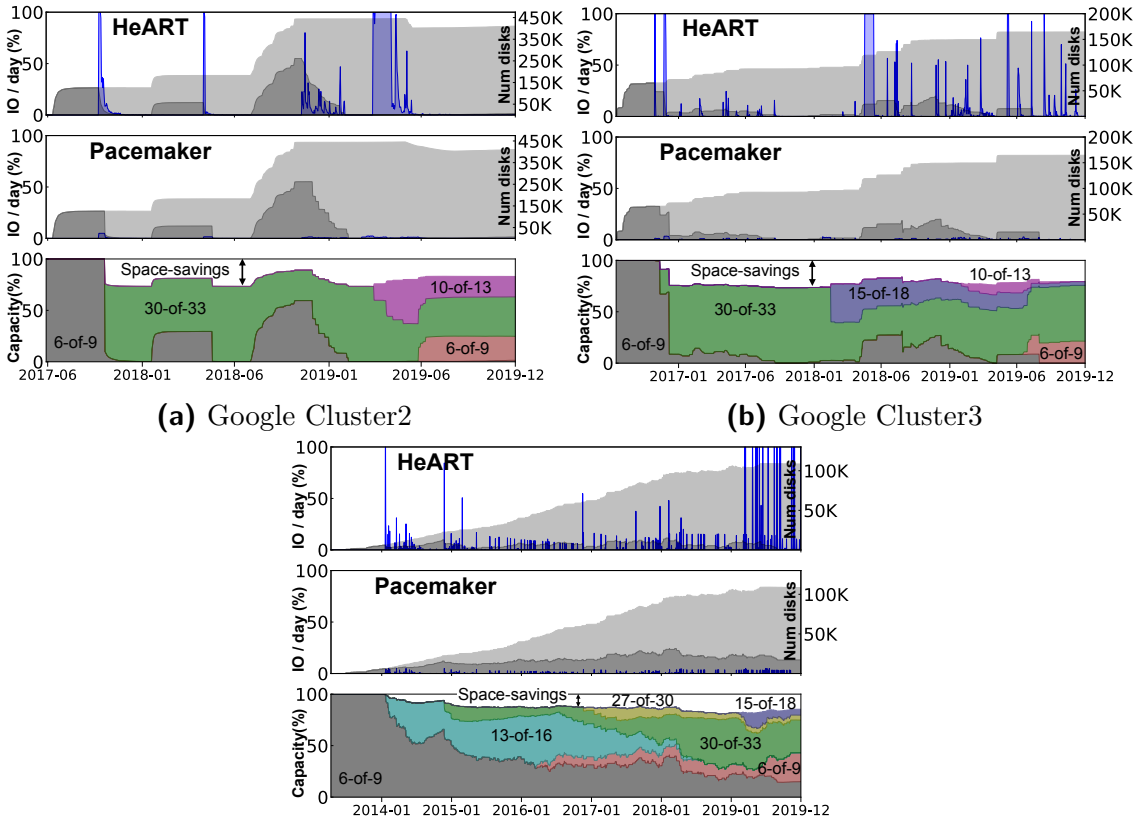
PACEMAKER was configured to use a peak-IO-cap of 5%, an average-IO constraint of 1% and a threshold-AFR of 75% of the tolerated-AFR, except for the sensitivity studies in Section 5.7.3. For comparison, we also simulate (1) an idealized disk-adaptive redundancy system in which transitions are instantaneous (requiring no IO) and (2) the prior state-of-the-art approach (HeART) for disk-adaptive redundancy. For all cases, Rgroup0 uses 6-of-9, representing a one-size-fits-all scheme reported in prior literature [121]. The required target MTTDL is then back-calculated using the 6-of-9 default and an assumed tolerated-AFR of 16% for Rgroup0. These configuration defaults were set by consulting storage administrators of clusters we evaluated.

5.7.1 PACEMAKER on Google Cluster1 in-depth

Figure 5.5a shows the IO generated by PACEMAKER (and disk count) over the ≈3-year lifetime of Google Cluster1. Over time, the cluster grew to over 350K disks comprising of disks from 7 makes/models (Dgroups) via a mix of trickle and step deployments. Figures 5.5b and 5.5c show AFR curves of 2 of the 7 Dgroups[7] (obfuscated as G-1 and G-2 for confidentiality) along with how PACEMAKER adapted to them at each age. G-1 disks are trickle-deployed whereas G-2 disks are step-deployed. The other 5 Dgroups are omitted due to lack of space. Figure 5.5d shows the corresponding space-savings (the white space above the colors).

All disks enter the cluster as unspecialized disks, i.e. Rgroup0 (dark gray region in the Figure 5.5a and left gray region of Figures 5.5b and 5.5c). Once a Dgroup's AFR reduces sufficiently, PACEMAKER RDn transitions them to a specialized Rgroup (light gray area in Figure 5.5a). Over their lifetime, disks may transition through multiple RUp transitions over the multiple useful life phases. Each transition requires IO, which is captured in blue in Figure 5.5a. For example, the sudden drop in the unspecialized disks, and the blue area around 2018-04 captures the Type 2 transitions

[7]The rest of the Dgroups' AFR curves are shown in Figure 5.9 in Section 5.9.



(a) Google Cluster2

(b) Google Cluster3

(c) Backblaze

Figure 5.6: Top two rows show the IO overhead comparison between prior adaptive redundancy system (HeART) and PACEMAKER on two Google clusters and one Backblaze cluster. PACEMAKER successfully bounds all IO under 5% (visible as tiny blue regions in middle graphs, for e.g. around 2017 in (a)). The bottom row shows the 14–20% average space-savings achieved by PACEMAKER across the three clusters. The AFR curves of all three clusters are shown in Figures 5.10 to 5.12 in Section 5.9.

caused when over 100K disks RDn transition from Rgroup0 to a specialized Rgroup. The light gray region in Figure 5.5a corresponds to the time over which space-savings are obtained, which can be seen in Figure 5.5d.

Many transitions with no transition overload. PACEMAKER successfully bounds all redundancy management IO comfortably under the configured peak-IO-cap throughout the cluster's lifetime. This can be seen via an imaginary horizontal line at 5% (the configured peak-IO-cap) that none of the blue regions goes above. Recall that PACEMAKER rate-limits the IO within each Rgroup to ensure simultaneous transitions do not violate the cluster's IO cap. Events ***G-1eA*** and ***G-2eA*** are examples of events where both G-1 and G-2 disks (making up almost 100% of the cluster at that time) request transitions at the same time. Despite that, the IO remains bounded below 5%. ***G-3eC*** and ***G-6eB*** also show huge disk populations of G-3 and G-6 Dgroups (AFRs not shown) requesting almost simultaneous RUp transitions, but PACEMAKER's design ensures that the peak-IO constraint is never violated. This is in sharp contrast with HeART's frequent transition overload, shown in Figure 5.1a.

Disks experience multiple useful life phases. G-1, G-3, G-6 and G-7 disks experience two phases of useful life each. In Figure 5.5a, events ***G-1eA*** and ***G-1eB*** mark the two transitions of G-1 disks through its multiple useful lives as shown in Figure 5.5b. In the absence of multiple useful life phases, PACEMAKER would have RUp transitioned G-1 disks to Rgroup0 in 2019-05, eliminating space-savings for the remainder of their time in the cluster. Section 5.7.3 quantifies the benefit of multiple useful life phases for all four clusters.

MTTDL always at or above target. Along with the AFR curves, Figures 5.5b and 5.5c also show the upper bound on the AFR for which the reliability constraint is met (top of the gray region). PACEMAKER sufficiently protects all disks throughout their life for all Dgroups across evaluated clusters.

Substantial space-savings. PACEMAKER provides 14% average space-savings (Figure 5.5d) over the cluster lifetime to date. Except for 2017-01 to 2017-05 and 2017-11 to 2018-03, which correspond to infancy periods for large batches of new empty disks added to the cluster, the entire cluster achieves ≈20% space-savings. Note that the apparent reduction in space-savings from 2017-11 to 2018-03 isn't actually

reduced space in absolute terms. Since Figure 5.5d shows relative space-savings, the over 100K disks deployed around 2017-11, and their infancy period makes the space-savings appear reduced relative to the size of the cluster.

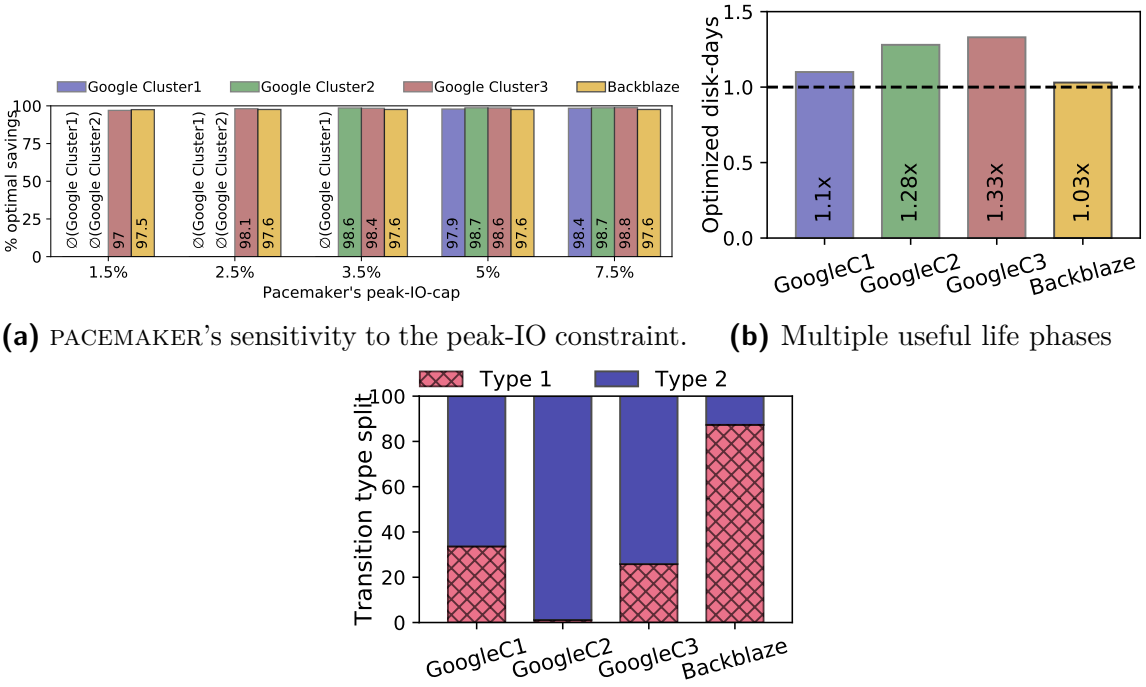
5.7.2 PACEMAKER on the other three clusters

Figure 5.6 compares the transition IO incurred by PACEMAKER to that for HeART [1] for Google Cluster2, Google Cluster3 and Backblaze, along with the corresponding space-savings achieved by PACEMAKER. While clusters using HeART would suffer transition overload, the same clusters under PACEMAKER always had all their transition IO under the peak-IO-cap of 5%. In fact, on average, only 0.21–0.32% percent of the cluster IO bandwidth was used for transitions. The average space-savings for the three clusters are 14–20%.

Google Cluster2. Figure 5.6a shows the transition overload and space-savings in Google Cluster2 and the corresponding space-savings. All Dgroups in Google Cluster2 are step-deployed. Thus, it is not surprising that Figure 5.7c shows that over 98% of the transitions in Cluster2 were Type 2 transitions (bulk parity recalculation). Cluster2's disk population exceeds 450K disks. Even at such large scales, PACEMAKER obtains average space-savings of almost 17% and peak space-savings of over 25%. This translates to needing 100K fewer disks.

Google Cluster3. Google Cluster3 (Figure 5.6b) is not as large as Cluster1 or Cluster2. At its peak, Cluster3 has a disk population of approximately 200K disks. But, it achieves the highest average space-savings (20%) among clusters evaluated. Like Cluster2, Cluster3 is also mostly step-deployed.

Backblaze Cluster. Backblaze (Figure 5.6c) is a completely trickle-deployed cluster. The dark grey region across the bottom of Figure 5.6c's PACEMAKER plot shows the persistent presence of canary disks throughout the cluster's lifetime. Unlike the Google clusters, the transition IO of Backblaze does not produce bursts of transition IO that lasts for weeks. Instead, since trickle-deployed disks transition a-few-at-a-time, we see transition work appearing continuously throughout the cluster lifetime of over 6 years. The rise in the transition IO spikes in 2019, for HeART, is



(a) PACEMAKER's sensitivity to the peak-IO constraint. **(b)** Multiple useful life phases

(c) Transition type distribution

Figure 5.7: (a) shows PACEMAKER's sensitivity to the peak IO bandwidth constraint. (b) shows the advantage of multiple useful life phases and (c) shows the contribution of the two transitioning techniques when PACEMAKER was simulated on the four production clusters.

because of large capacity 12TB disks replacing 4TB disks. Unsurprisingly, under PACEMAKER, most of the transitions are done using Type 1 (transitioning by emptying disks) as shown in Figure 5.7c. The average space-savings obtained on Backblaze are 14%.

5.7.3 Sensitivity analyses and ablation studies

Sensitivity to IO constraints. The peak-IO constraint governs Figure 5.7a, which shows the percentages of optimal space-savings achieved with PACEMAKER for peak-IO-cap settings between 1.5% and 7.5%. A peak-IO-cap of up to 7.5% is used in order to compare with the IO percentage spent for existing background IO activity, such as scrubbing. By scrubbing all data once every 15 days [137], the scrubber uses around 7% IO bandwidth, and is a background work IO level tolerated by today's clusters.

The Y-axis captures how close the space-savings are for the different peak-IO-caps compared to "Optimal savings", i.e. an idealized system with infinitely fast transitions. PACEMAKER's default peak-IO-cap (5%) achieves over 97% of the optimal space-savings for each of the four clusters. For peak-IO constraint set to $\leq$2.5%, some RUp transitions in Google Cluster1 and Cluster2 become too aggressively rate-limited causing a subsequent AFR rise to violate the peak-IO constraints. We indicate this as a failure, and show it as "$\varnothing$". The same situation happens for Google Cluster1 at 3.5%.

Sensitivity to threshold-AFR. The threshold-AFR determines when proactive RUp transitions of step-deployed disks are initiated. Conceptually, the threshold-AFR governs how risk-averse the admin wants to be. Lowering the threshold would trigger an RUp transition when disks are farther away from the tolerated-AFR (more risk-averse), and vice-versa. We evaluated PACEMAKER for threshold-AFRs of 60%, 75% and 90% of the respective Rgroups' tolerated-AFRs. We found that PACEMAKER's space-savings is not very sensitive to threshold-AFR, with space-savings only 2% lower at 60% than at 90%. Data remained safe at each of these settings, but would become unsafe with higher values.

Contribution of multiple useful life phases. Figure 5.7b compares the

increased number of disk-days spent in specialized Rgroups because of considering multiple useful life phases. In the best case, Google Cluster2 spent 33% more disk-days in specialized redundancy, increasing overall space-savings from 16% to 19%. Note that in large-scale storage clusters, even 1% space-savings are considered substantial as it represents thousands of disks.

Contribution of transition types. By proactively keeping step-deployed disks in distinct Rgroups and using specialized transitioning schemes whenever possible, instead of using simple re-encoding for all transitions, PACEMAKER reduces total transition IO by 92–96% for the four clusters. Figure 5.7c shows what percentage of transitions were done via Type 1 (disk emptying) vs. Type 2 (bulk parity recalculation). As expected, Google clusters rely more on Type 2 transitions, because most disks are step-deployed. In contrast, the Backblaze cluster is entirely trickle-deployed and hence mostly uses Type 1 transitions. The small percentage of Type 2 transitions in Backblaze occur when Rgroups are purged.

5.7.4 Evaluating HDFS + PACEMAKER

This section describes basic experiments with the PACEMAKER-enabled HDFS, focusing on its functioning and operation. Note that PACEMAKER is designed for longitudinal disk deployments over several years, a scenario that cannot be reproduced identically in laboratory settings. Hence, these HDFS experiments are aimed to display that integrating PACEMAKER with an existing storage system is straightforward, rather than on the long-term aspects like overall space-savings or transition IO behavior over cluster lifetime as evaluated via simulation above.

The HDFS experiments run on a PRObE Emulab cluster [144]. Each machine has a Dual-Core AMD Opteron Processor, 16GB RAM, and Gigabit Ethernet. We use a 21-node cluster running HDFS 3.2.0 with one NN and 20 DNs. Each DN has a 10GB partition on a 10000 RPM HDD for a total cluster size of 200GB. We statically define the cluster to be made up of two Rgroups of ten DNs each, one using the 6-of-9 erasure coding scheme and the other using a 7-of-10 scheme. DFS-perf [145], a popular open-source HDFS benchmark is used, after populating the cluster to 60% full. Each

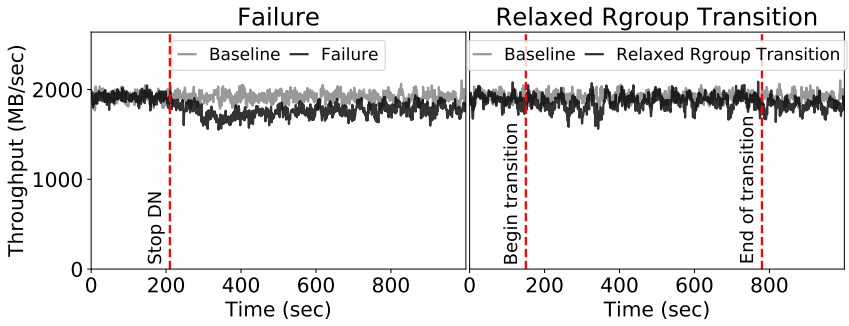


Figure 5.8: DFS-perf reported throughput for baseline, with one DN failure and one Rgroup transition.

DFS-perf client sequentially reads one file over and over again (size=768MB), for a total read size of about 1.75TB over 40 iterations. We use 60 DFS-perf clients, running on 20 nodes separate from the HDFS cluster.

We focus on the behavior of a DN as it transitions between Rgroups, compared with baseline HDFS performance (where all DNs are healthy) and its behavior while recovering from a failed DN. Figure 5.8 shows the client throughput after the setup phase, followed by a noticeable drop in client throughput when a DN fails (emuated by stopping the DN). This is caused by the reconstruction IO that recreates the data from the failed node. Read latency exhibits similar behavior (not shown due to space). Eventually, throughput settles at about 5% lower than prior to failure, since now there are 19 DNs.

Figure 5.8 also shows client throughput when a node is RDn transitioned from 6-of-9 to 7-of-10. There is minor interference during the transition, which can be attributed to the data movement that HDFS performs as a part of decommissioning. The transition requires less work than failed node reconstruction, yet takes longer to complete because PACEMAKER limits the transition IO. Eventually, even though 20 DNs are running, the throughput is lower by ≈5% (one DN's throughput). This happens because PACEMAKER empties the DN before it moves into the new Rgroup, and load-balancing data to newly added DNs happens over a longer time-frame. Experiments with RUp transition showed similar results.

5.8 Failure rate estimation in PACEMAKER

This section describes how we calculate failure rates for each Dgroup based on the disks' age using empirical data. In the storage device reliability literature, the failure rate over a period of time is typically expressed in terms of Annualized Failure Rate (AFR), and calculated as:

$$AFR\ (\%) = \frac{d}{E} \times 100, \tag{5.1}$$

where d is the number of observed disk failures, and E is the sum of the exposure time of each disk, measured in years. The exposure time of a disk is the amount of time it was in operation (i.e., deployed and had not failed nor been retired) during the period in consideration, and it is typically measured at the granularity of days.

If the time to failure is exponentially distributed, then Equation (5.1) corresponds to the maximum likelihood estimate for the rate parameter of the exponential distribution. Due to the memoryless property of this distribution, such a formula would be appropriate only if we assume that failure rate is constant with respect to time or device age. Thus, Equation (5.1) may be useful for estimating AFR over long and stable periods of time, but makes it hard to reason about changes in AFR over time. Therefore, in this work, we estimate AFR using the following approach.

Assume that the lifetime (time from deployment to failure) of each disk is an i.i.d. discrete random variable T with cumulative density function F and probability mass function f. The failure rate (also known as *hazard rate*) [146] of this distribution is given by:

$$h(t) = f(t)/(1 - F(t)). \tag{5.2}$$

The *cumulative hazard* defined as $H(t) = \sum_{i=0}^{t} h(i)$ is commonly estimated using the Nelson-Aalen estimator:

$$\hat{H}(t) = \sum_{i=0}^{t} \frac{d_i}{a_i} \qquad \text{for } t \in \{0, \ldots, m\}, \tag{5.3}$$

where d_i is the number of disks that failed during their i-th day, a_i is the number of disks that were in operation at the start of their i-th day, and m is the age in days of

the oldest observed disk drive. An estimate for the failure rate can be obtained by applying the so-called *kernel method* [147]:

$$\hat{h}(t) = \sum_{i=0}^{m} \frac{d_i}{a_i} K(t-i), \qquad \text{for } t \in \{0, \ldots, m\}, \tag{5.4}$$

where $K(\cdot)$ is a kernel function. In practice, Equation (5.4) can be considered as a smoothing over the increments of Equation (5.3). For our calculations, we utilized an Epanechnikov kernel [140] with a bandwidth of 30 days (the Epanechnikov kernel is frequently used in practice due to its good theoretical properties).

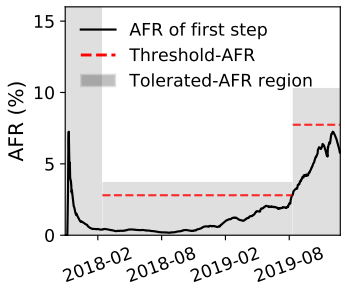
A big advantage of this approach is that it is *nonparametric*, meaning that it does not assume that the lifetime T follows any particular distribution. This allows PACEMAKER to adapt and work effectively with a wide arrange of storage devices with vastly different failure rate behaviors.

5.9 Detailed cluster evaluations of PACEMAKER

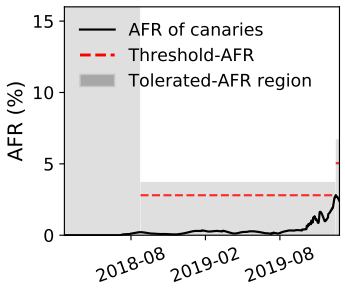
This section shows the remaining Dgroups of Google Cluster1 (Figure 5.9) and provides a similar deep-dive of PACEMAKER on Google Cluster2, Google Cluster3 and the Backblaze cluster along with the AFR curves of all Dgroups of those clusters.

Dgroups of Google Cluster1. Recall from Section 5.7 that Google Cluster1 is made up of seven Dgroups. G-1 and G-2 AFR curves are shown in Figures 5.5b and 5.5c respectively. Here we show the four of the file remaining Dgroups, viz. G-3, G-5, G-6 and G-7 in Figures 5.9a to 5.9e. G-3 and G-7 disks are trickle-deployed similar to G-2 disks, whereas the other disks are step-deployed.

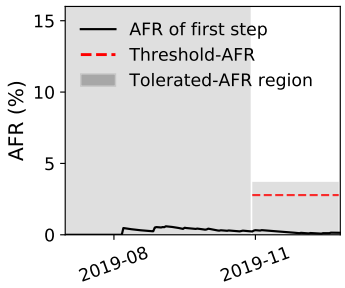
Google Cluster2. Figure 5.10a shows the PACEMAKER-generated IO for redundancy management. Figure 5.10b shows the corresponding space savings. Finally Figures 5.10c to 5.10f shows the AFRs of the four Dgroups that make up Cluster2. All Dgroups in Google Cluster2 are step-deployed. Thus, it is not surprising that Figure 5.7c shows that over 98% of the transitions in Cluster2 were performed by bulk parity recalculation. This is the largest cluster PACEMAKER was simulated on.



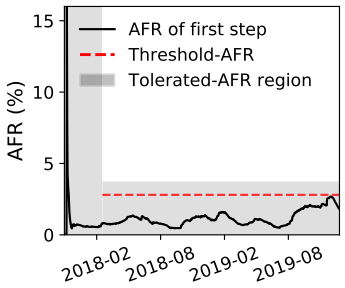
(a) G-6 (step) AFR curve



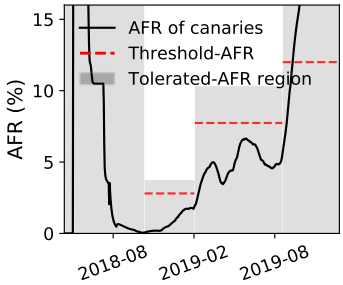
(b) G-4 (trickle) AFR curve



(c) G-8 (step) AFR curve



(d) G-5 (step) AFR curve



(e) G-3 (trickle) AFR curve

Figure 5.9: Detailed IO analysis and space savings achieved by PACEMAKER-enabled adaptive redundancy on Google Cluster1.

Cluster2's disk population exceeds 450K disks. Even at such large scales, PACEMAKER is able to obtain average space savings of almost 17% and peak space savings of over 25%. This translates to needing 100K fewer disks, essentially saving millions of dollars.

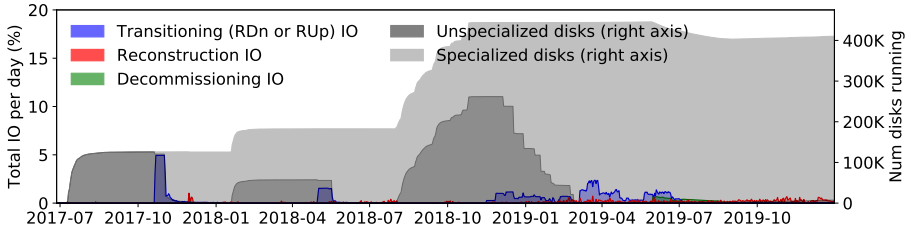
Google Cluster3. Google Cluster3 is not as large as Cluster1 or Cluster2. At its peak, Cluster3 has a disk population of approximately 200K disks. But, it achieves the highest average space savings (20%) compared to all other clusters. Figure 5.11a shows the PACEMAKER-generated IO, Figure 5.11b shows the space savings and Figures 5.11c to 5.11e shows the AFR curves of its three Dgroups. Like Cluster2, Cluster3 is also mostly step-deployed.

Backblaze Cluster. Backblaze is a completely trickle-deployed cluster. Figure 5.12a shows the PACEMAKER-generated IO. Unlike Google clusters, the transition IO of Backblaze does not produce large regions of transition workload. Instead, since trickle-deployed disks transition a-few-at-a-time, we see transition work appearing continuously throughout the cluster lifetime of over 6 years. Unsurprisingly, most of the transitions are done by emptying disks (Type 1; refer to Figure 5.7c). In terms of sensitivity, the Backblaze cluster is the most insensitive to the peak-IO constraint since always requires much lower transition bandwidth per day.

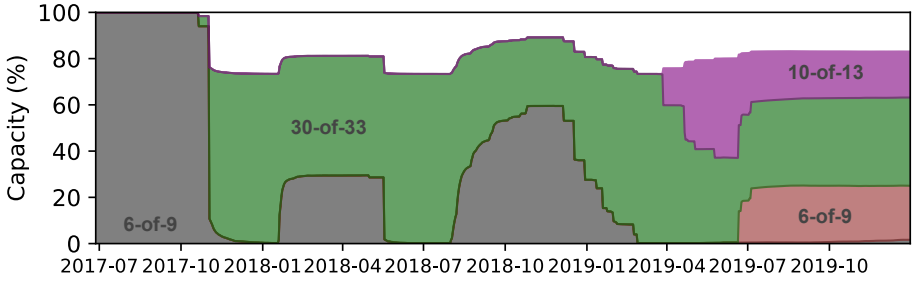
5.10 Tiger: disk-adaptive redundancy without additional placement restrictions

As we have shown in this chapter, the conventional approach of using a single erasure code across the cluster uses excessive redundancy (wasting capacity, and thus money and energy) to guarantee data safety, given that different disks have different failure rates. Instead, adapting the redundancy scheme selection to the observed failure (as done by Pacemaker) reduces the space overhead of redundancy by up to 20%.

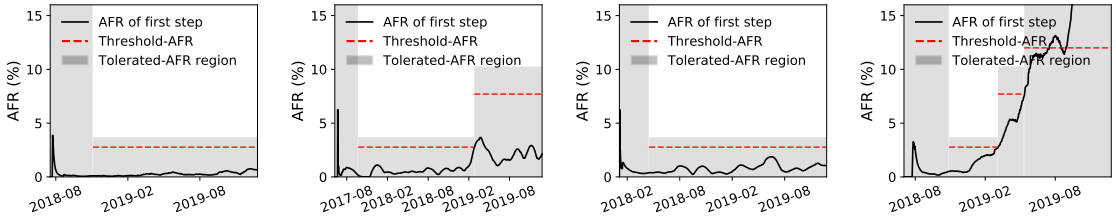
The design of Pacemaker, however, faces several significant adoption hurdles. At its core, this design is based on rigidly partitioning a storage cluster into subclusters of disks (called redundancy groups or Rgroups) that have similar failure rates, so



(a) Google Cluster2 redundancy management IO due to PACEMAKER over its 2+ year lifetime broken down by IO type.

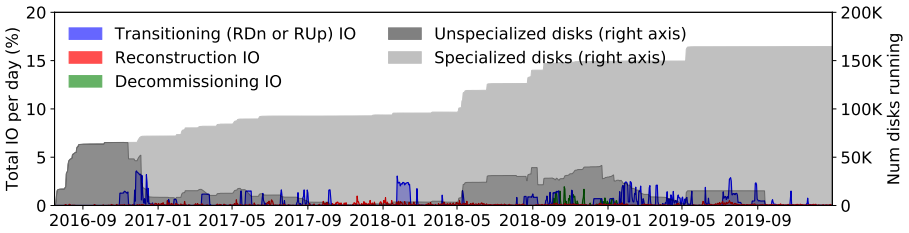


(b) Google Cluster2 space savings achieved by PACEMAKER.

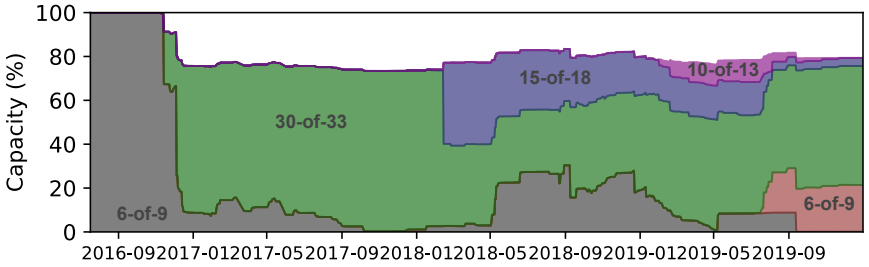


(c) G-6 (step) AFR curve **(d)** G-1 (step) AFR curve **(e)** G-5 (step) AFR curve **(f)** G-3 (step) AFR curve

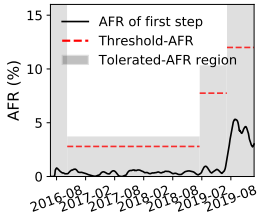
Figure 5.10: Detailed IO analysis and space savings achieved by PACEMAKER-enabled adaptive redundancy on Google Cluster2.



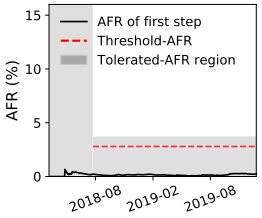
(a) Google Cluster3 redundancy management IO due to PACEMAKER over its 3 year lifetime broken down by IO type.



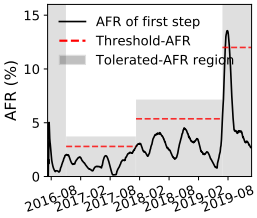
(b) Google Cluster3 space savings achieved by PACEMAKER.



(c) G-2 (step) AFR curve

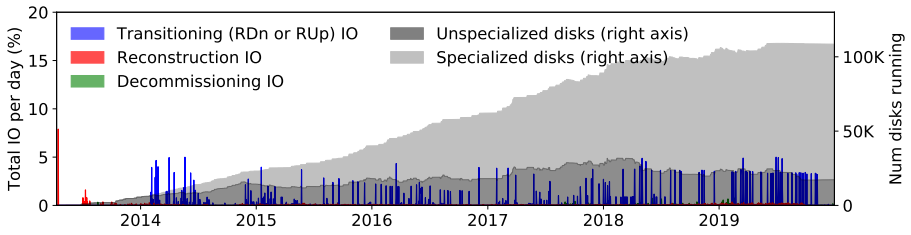


(d) G-4 (step) AFR curve

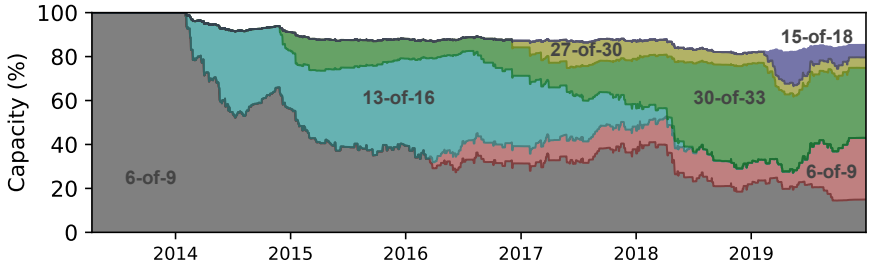


(e) G-1 (step) AFR curve

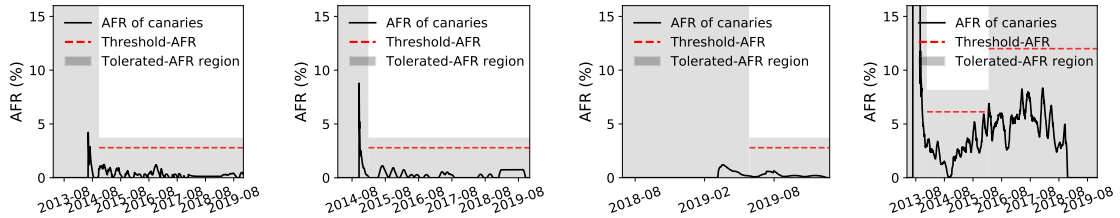
Figure 5.11: Detailed IO analysis and space savings achieved by PACEMAKER-enabled adaptive redundancy on Google Cluster3.



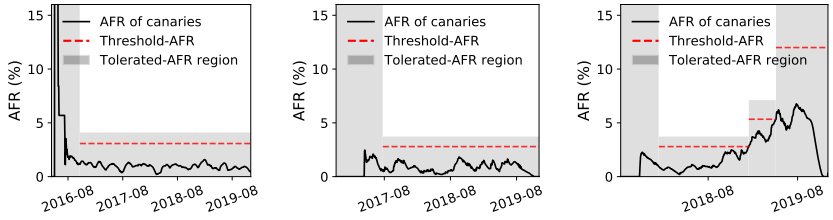
(a) Backblaze redundancy management IO due to PACEMAKER over its 6+ year lifetime broken down by IO type.



(b) Backblaze space savings achieved by PACEMAKER.

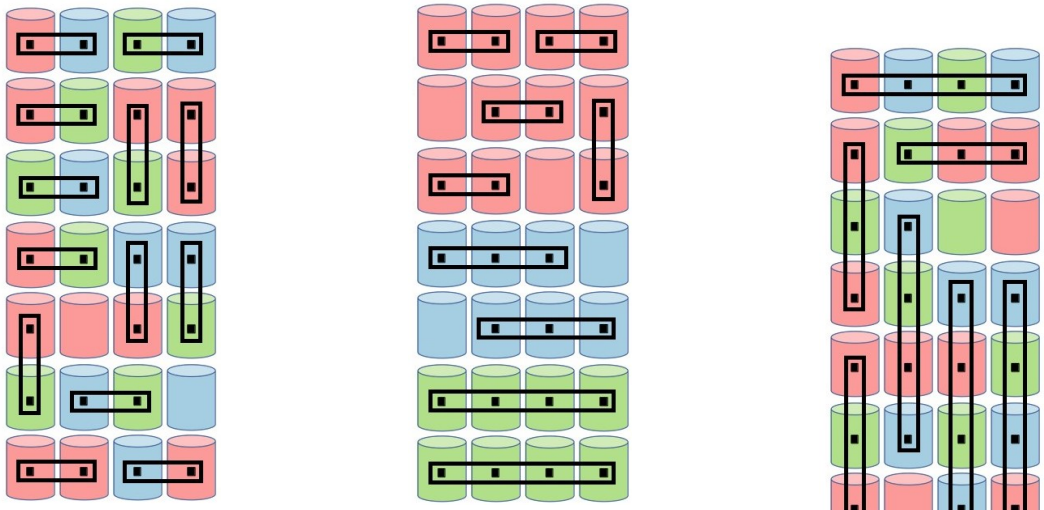


(c) H-4A (trickle) AFR curve **(d)** H-4B (trickle) AFR curve **(e)** H-12E (trickle) AFR curve **(f)** S-4 (trickle) AFR curve



(g) S-8C (trickle) AFR curve **(h)** S-8E (trickle) AFR curve **(i)** S-12E (trickle) AFR curve

Figure 5.12: Detailed IO analysis and space savings achieved by PACEMAKER-enabled adaptive redundancy on the Backblaze cluster.



(a) Conventional cluster storage

(b) Pacemaker (subcluster-based)

(c) Tiger

Figure 5.13: Stripe placements and configurations in different erasure coding systems: Disks of same color have similar annualized failure rates (AFRs), with red being least reliable (highest AFR), then blue, then green. Rectangles represent stripes with shorter stripes having higher redundancy. Conventional one-scheme-fits all designs (Figure 5.13a) impose no placement restrictions, but make no distinction of disk AFRs and therefore overprotect much of the data—all stripes use the widest redundancy scheme, shown as 2-wide for illustration. Pacemaker (Figure 5.13b) and Tiger (Figure 5.13c) tailor redundancy based on disk AFRs, resulting in different stripe widths in the illustration, and thereby reduce storage overhead. Pacemaker does this with rigid AFR-based subcluster boundaries, whereas Tiger requires no such boundaries.

they can use a subcluster-wide redundancy scheme tailored to meet the required data reliability target (e.g., Figure 5.13b). Key adoption hurdles include: (1) Since each stripe must be entirely within a single Rgroup, this subcluster-based design can interfere with other data placement considerations, such as enhancing risk-diversity by spreading data across fault domains and different makes/models/batches of disks. Indeed, many of the Rgroups consist of a single make/model. (2) To provide reasonable degrees of performance and reconstruction speed scalability, subclusters must be sizable, making these designs only suitable for very large storage clusters. (3) When failure rates rise for a given make/model, as it ages, the redundancy scheme for an entire Rgroup (potentially 100s of PBs) may need to change to maintain target data reliability levels—all at once. The Pacemaker design [15] proposes to predict such changes and start them early, but they need to predict a month or more in advance to avoid reliability problems given the huge amount of data being transitioned, which is inherently a risky proposition. (4) The subcluster-based designs assume full adoption of disk-adaptive redundancy, not allowing for selective adoption for some data corpuses but not for others.

In the remainder of this chapter, we present Tiger, a disk-adaptive redundancy system that eliminates the placement constraints posed by subcluster-based disk-adaptive redundancy designs while providing equal or greater benefits. Tiger's core new abstraction is the *eclectic stripe*, in which disks of different AFRs can be used to store a stripe that has redundancy tailored to the set of AFRs for those disks. In terms of placement flexibility, eclectic stripes are identical to stripes in conventional (non-disk-adaptive redundancy) designs. But, unlike conventional stripes, eclectic stripes do not conservatively assume the worst-case AFR for all disks. Instead, with eclectic stripes, the redundancy scheme is dynamically set for each stripe based on the AFRs of the chosen disks (e.g., Figure 5.13c). Tiger's eclectic stripe approach avoids all the adoption hurdles discussed above, while simultaneously increasing the effectiveness (higher space-savings) and robustness (lower burstiness of urgent transition IO) of disk-adaptive redundancy.

Efficiently incorporating the proposed new abstraction of eclectic stripes is challenging due to multiple reasons. Tiger introduces several new design elements

to overcome these challenges. First, calculating the exact reliability in terms of mean-time-to-data-loss (MTTDL) of a stripe can be prohibitively expensive, since accounting for different failure rates can lead to an exponential number of states in the traditional Markov chain reliability model. To address this, we provide a novel approximation technique that speeds up MTTDL calculation by 2-4 orders of magnitude while always preserving accuracy of over 95%, and on average over 99.5%. Second, while disks for a stripe can be chosen based on pre-existing placement policies, the chosen disks may not form an adequately-reliable stripe for a planned redundancy scheme, since the reliability is dependent on the chosen disks' AFRs. Tiger uses an AFR-aware stripe-width-reduction policy to quickly achieve sufficient reliability. Third, disk AFRs change over time [1], which can require changing the redundancy schemes of some eclectic stripes. Keeping track of AFRs for each stripe and triggering the redundancy schemes can significantly increase the overhead for metadata and background operations. Tiger introduces an *eclectic volume* abstraction to reduce metadata overhead and make identification of required changes efficient. It also introduces policies to reduce transition IO: the IO involved with enacting changes to stripe redundancy schemes.

Evaluating the feasibility and efficacy of eclectic stripes requires analysis of long-term effects on huge storage clusters. We evaluate Tiger using the same logs as used to evaluate Pacemaker, enabling an apples-to-apples comparison. These logs contain all disk-deployment, failure, and decommissioning events from four production storage clusters: three 160K–450K-disk Google clusters and a ≈110K-disk cluster used for the Backblaze Internet backup service [128]. Simulation driven by production logs allows us to analyze reliability, space usage, and redundancy maintenance traffic for multiple clusters each with over 100K disks and over multiple years, which would be infeasible otherwise as part of a research setup. For all four clusters, Tiger provides equal or better space-savings than Pacemaker, while requiring at most 0.5% of daily IO bandwidth for transition IO. More importantly, the transition IO is both less bursty, in terms of when it is needed, and less urgent, in terms of how unsafe an unsafe stripe might be if the scheme transition were delayed. For instance, in response to a tiny rise in AFR ($< 0.25\%$) for disks of a given make/model, Pacemaker would

need 196% of the total IO bandwidth from each of those disks in order to make the data safe—to avoid stealing more than 5% of IO bandwidth for transition IO, Pacemaker would have to know to start 40 days in advance—but Tiger would need <1.6% even for a 1% AFR increase because of the diversity of its eclectic stripes. And, most importantly, Tiger exhibits significantly better risk-diversity, stemming from removing placement constraints and allowing differently-reliable disks (and hence disks of different makes/models) to belong to the same stripe. For example, even with random selection of disks for each stripe, most of Tiger's eclectic stripes span most of a cluster's make/models; Pacemaker's strict Rgroup boundaries disallow use of more than one make/model for most stripes.

Contributions. In the rest of this chapter, we make four main contributions. First, we introduce eclectic stripes as a tool for realizing disk-adaptive redundancy without the placement restrictions posed by prior designs. Second, we present a reliability model and its approximation to efficiently calculate the MTTDL of eclectic stripes. A surprising outcome is that a homogeneous stripe with the same scheme and average disk AFR as an eclectic stripe is less reliable! Third, we present the design and architecture of Tiger, the first disk-adaptive redundancy system for supporting and efficiently managing eclectic stripes. Fourth, we evaluate Tiger and compare it to the state-of-the-art, using logs from four large real-world storage clusters, demonstrating its effectiveness in realizing disk-adaptive redundancy without prior designs' adoption challenges and with greater space-savings and lower risk.

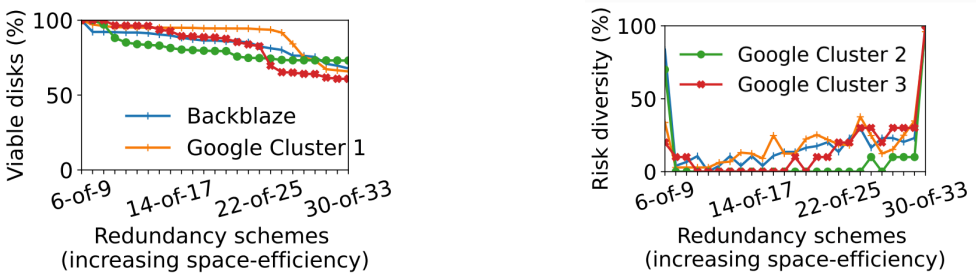
5.11 Motivation of Tiger

In this section, we describe the problems with existing disk-adaptive redundancy systems, which is the motivation for this system.

As shown by prior work [126, 127], disk AFRs are highly correlated with their vintage, and can vary dramatically over their life. Disk-adaptive redundancy capitalizes on differences in disk AFRs and dynamically tailors data redundancy to observed disk failure rates [148]. Disk-adaptive redundancy systems take into account

various constraints including the reconstruction costs when making the decision of a target stripe width to adapt to. Specifically, wide schemes are used only when a stripe's average AFR is low enough to keep the reconstruction cost contained below a configured limit. More generally, wide stripes provide cost savings in terms of smaller storage overhead at the cost of higher reconstruction costs and higher degraded mode reads. We know from conversing with architects of large-scale storage clusters that the cost of the excess byte footprint matters more than the cost of excess IO required in the context of redundancy, given existing workloads. This is especially so since, in general, large-scale capacity-tier storage cluster workloads tend to be cold (have low IO/s per byte). Additionally, cold data experiences fewer reads, and therefore has very few costly degraded mode reads. Backblaze is an example where, for archival data that has low IO access rates, administrators have publicly confirmed use of wide redundancy schemes such as 17-of-20 [149]. By using more space-efficient redundancy schemes during low AFR regimes, disk-adaptive redundancy can provide substantial space-savings ($> 20\%$) in clusters with over 100K disks.

There are two disk-adaptive redundancy systems that have been proposed prior to Tiger: HeART [1] and Pacemaker, which we presented earlier in this chapter. In HeART, the authors propose a tool to statistically learn the AFRs of different disk groups and identify change-points for safe redundancy transitions. By transitioning to an encoding scheme with minimum storage overhead that still meets the target MTTDL, HeART was able to obtain $\approx 20\%$ space-savings when tailoring erasure codes, and $\approx 33\%$ space-savings when tailoring replication. Although lucrative, HeART overlooked an important practical hurdle in performing disk-adaptive redundancy: *transition overload*, i.e. the IO overhead of performing redundancy transitions. Crippling transition overload when thousands of disks require simultaneous redundancy transitions forms the basis for Pacemaker. The gist of Pacemaker is to convert urgent redundancy transitions into schedulable ones by making conservative predictions of the rise in AFR and proactively issuing redundancy transitions. This allows the transition overload to be spread out over time, such that it can be completed within tolerable IO limits without compromising data safety.



(a) Placement constraints

(b) Risk-diversity

Figure 5.14: Figure 5.14a shows Pacemaker's placement constraints by highlighting the fraction of the disk fleet that is viable for different schemes exercised on four production clusters. Figure 5.14b shows the risk-diversity obtained by the same clusters on particular dates in their lifetime. A risk-diversity of 100% implies at least one chunk stored on every possible make/model, whereas a 0% risk-diversity implies that the particular scheme was not feasible in the cluster. Pacemaker performs poorly in both placement constraints and risk-diversity.

5.11.1 Existing designs are impractical

Despite remarkable space-savings and low IO costs, existing disk-adaptive redundancy systems remain impractical in real-world settings.

Placement restrictions. The primary hurdle stems from the placement restrictions posed by reliance on redundancy groups (*Rgroups*). Recall that an Rgroup is a set of disks with similar AFRs, such that they can use the same redundancy scheme. Prior systems redundancy management techniques rigidly partition the cluster's disks into Rgroups, and every stripe must be stored entirely within a single Rgroup. Figure 5.14a shows the percentage of disks that are rendered infeasible for various redundancy schemes Pacemaker can employ on a particular day in four large storage clusters. More than 30% of the disks are deemed infeasible for space-efficient schemes beyond 22-of-25, because their AFRs are not low enough for those disks to participate in an Rgroup for which schemes beyond 22-of-25 can meet the target MTTDL. Furthermore, in order to maintain proper redundancy, stripes are typically constrained to span across different racks, servers, power lines, etc. Adding another placement constraint may be close to impossible.

Lower risk-diversity. Due to high correlation of AFRs and makes/models/batches [126, 127], and in order to enable efficient transitioning mechanisms, many Rgroups contain disks from just one make/model. This is undesirable from a risk-diversity perspective. Figure 5.14b shows the fraction of makes/models that are covered for the same stripe configurations in the same four clusters described above. Higher risk-diversity is valuable for mitigating consequences of bulk failure situations (e.g., from rapid degradation due to manufacturing defects), especially in a disk-adaptive redundancy system where redundancy is tuned rather than regularly excessive.

Reliance on AFR prediction. With lower risk-diversity, Pacemaker's Rgroups are already susceptible to data loss due to bulk failures in a single make/model (uncommon, but not impossible). Furthermore, Pacemaker's IO cost reduction is highly dependent on being able to accurately predict an AFR rise well in advance. Currently AFR is calculated only on the basis of age. Prior work has highlighted that it is dependent on various factors such as vintage, temperature, vibration, etc. [126, 127, 150, 151]. This makes an already difficult task of accurate AFR prediction even harder.

All-or-nothing. Current disk-adaptive redundancy designs depend on forming Rgroups, and work efficiently if entire Rgroups perform redundancy transitions together (for step-deployed disks). This implies that the entire cluster must commit to performing disk-adaptive redundancy for all of their data stored on all disks. Such a restriction makes disk-adaptive redundancy unusable without a major overhaul of the architecture of the existing storage cluster.

The key takeaway is that additional data placement restrictions create adoption-blocking limitations and risks. In order have have both placement flexibility and disk-adaptivity, we need a new approach that includes the ability to reason about and tune the reliability of stripes that span disks with different AFRs. We achieve this via ***eclectic stripes***.

5.12 *Eclectic Stripes* and their challenges

Eclectic stripes are central to Tiger's approach of providing disk-adaptive redundancy without placement restrictions. An eclectic stripe is an EC stripe placed on a collection of disks that can have different failure rates. The reliability model of conventional EC stripes forces them to be allocated on disks having (or worse, assumed to be having) the same failure rate. In terms of composition an eclectic stripe is no different than what a conventional EC stripe would be. Specifically, the same disks that make up a conventional stripe can also make up an eclectic stripe, just that eclectic stripes are cognizant of the AFR differences of the underlying disks and can accurately reason about the resulting reliability. A disk-adaptive redundancy system that supports eclectic stripes has to overcome several challenges.

1. Ensure efficient creation of sufficiently reliable eclectic stripes. Taking AFR differences of all disks in a stripe into account makes exact MTTDL calculation of eclectic stripes prohibitively expensive (see Section 5.13.1). Since stripe creation is a critical-path operation, it is imperative that a disk-adaptive redundancy system supporting eclectic stripes reasons about its reliability in an efficient and accurate manner.

2. Ensure efficient management of eclectic stripes. All underlying disks of an eclectic stripe will not experience an AFR rise or fall together. A system supporting eclectic stripes must efficiently identify which stripes need to change their redundancy in response to changing AFRs.

3. Support unchanged placement policies. While tweaking the placement policies might provide additional optimizations, a system that supports eclectic stripes must support existing placement policies without any change.

4. Retain key benefits of disk-adaptive redundancy. Dynamic redundancy adaptation at a low transition IO cost; continuously providing adequate reliability; providing space-savings by using more space-efficient redundancy schemes in low-AFR regimes are the key benefits of disk-adaptive redundancy. Any proposed disk-adaptive redundancy system should strive to maintain these benefits.

5. Ensure an adoption-friendly design. Apart from placement restrictions,

existing disk-adaptive redundancy system designs require that the entire cluster commits entirely to perform disk-adaptive redundancy, or it cannot gain any of its benefits. Moreover, only the very large-scale storage clusters can use existing disk-adaptive redundancy designs, whereas the small and medium sized clusters are outside their scope. High emphasis on usability and showcasing a way for easy adoption of disk-adaptive redundancy in existing storage clusters of all shapes and sizes is an important design challenge.

5.13 Mechanisms to enable eclectic stripes

In this section, we address the two main challenges of eclectic stripes: their reliability and their management.

5.13.1 Interpreting reliability of eclectic stripes

We first shed light on key takeaways from our study of the reliability of eclectic stripes and then provide the detailed theory and the associated analysis.

Calculating MTTDL of eclectic stripes is efficient and accurate. The exact calculation of the MTTDL of an eclectic stripe is computationally expensive. We provide a novel approximation that provides the MTTDL with over 99.5% accuracy (on average), and always provides over 95% accuracy in our tests. In practice, a difference of 5% in MTTDL typically translates into a difference of around 0.1% AFR for a homogeneous stripe, which is negligible. The exact MTTDL calculation and the approximation are detailed in Section 5.13.1.

Eclectic stripes are more reliable than homogeneous stripes. When comparing the MTTDL of an eclectic stripe with a homogeneous stripe having the same EC scheme and same avg. AFR, the MTTDL of the eclectic stripe is always higher than the MTTDL of the corresponding homogeneous stripe for typical system parameters (Section 5.13.2, Figure 5.16).

Eclectic stripes are robust to AFR changes of individual disks. The MTTDL of the eclectic stripes does not react abruptly to the increase in AFR of a

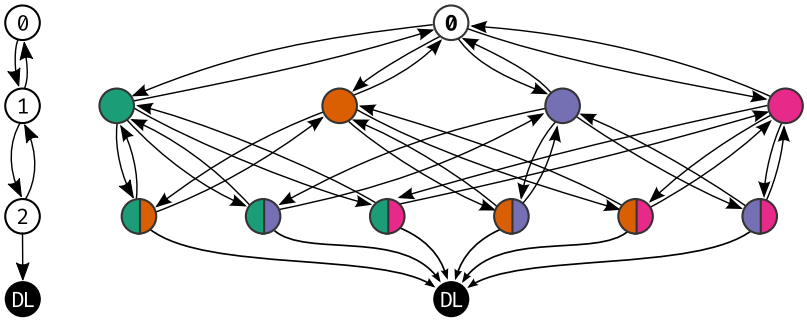


Figure 5.15: Left: Classic Markov chain model for the MTTDL of a 2-of-4 homogeneous stripe. Right: Markov chain model for the MTTDL of a 2-of-4 eclectic stripe.

few disks. Compared to the conventional approach of treating stripes as homogeneous with AFR equal to the maximum AFR in the stripe, MTTDL of eclectic stripes react very gradually to AFR changes.

Eclectic stripes are more robust to AFR misestimations. Due to the nature of empirical data, any system that measures AFR has to estimate it. Since the AFRs of different disk make/models are estimated independently, it is unlikely that there will be simultaneous underestimation of the AFR of every disk in an eclectic stripe, and hence the impact of estimation errors is smaller (Figure 5.17) and may even cancel each other out. Furthermore, disk-adaptive redundancy systems are made even more robust against misprediction by the use of confidence intervals. Thus, eclectic stripes are more robust to AFR misestimations compared to homogeneous stripes.

Exact MTTDL calculation is costly

Using a Markov chain model to calculate the MTTDL of storage systems is a classic approach [133]. A generalization of this approach helps us take into account disks with different failure rates. Consider an EC stripe of a k-of-n scheme, placed over n disks with failure rates $\lambda_i (i \in [n])$ and a disk repair rate of μ. The state of the system is given by an n-length vector $\mathbf{s} = (s_0, \ldots, s_n)$ with $s_i = 1$ if disk i has failed, and $s_i = 0$ otherwise $(i \in [n])$. The state space is given by states $(s_i)_{i=1}^{n}$ such that the total number of failure $\sum_{i=0}^{n} s_i$ is at most the number of parities $n - k$, and a data

loss state labeled DL. Therefore, the total number of states is $1 + \sum_{i=0}^{n-k} \binom{n}{i}$. The rate of transition from state $\mathbf{s}$ to $\mathbf{s}'$ is defined as:

- λ_i if $s_i = 0, s'_i = 1$, and $s_j = s'_j$ for $i \neq j$ (i^{th} disk fails),
- μ if $s_i = 1, s'_i = 0$, and $s_j = s'_j$ for $i \neq j$ (i^{th} disk repaired),
- $\sum_{i=1}^{n}(1 - s_i)\lambda_i$ if $\sum_{i=1}^{n} s_i = n - k$ and $\mathbf{s}' = DL$ (any disk fails when $n - k$ disks have failed and are not repaired).

The MTTDL is defined as the mean time to state DL from the initial state $\mathbf{0} = (0, \ldots, 0)$.

Given the values of $n, k, (\lambda_i)_{i=1}^{n}$, and μ, one can compute the MTTDL by using the standard approach of solving a system of equations. However, this approach is not tractable, due to the exponential explosion on the number of states with respect to $n - k$ (see Figure 5.15 to compare conventional Markov chain with that of an eclectic stripe). For example, the Markov chain of a 10-of-14 eclectic stripe has 1472 states, compared to 6 states in the case of a 10-of-14 homogeneous stripe. Reasoning about this model can be hard too, since it is not directly clear how disk AFRs affect MTTDL. Furthermore, this approach tends to be numerically unstable, which makes obtaining precise MTTDLs hard. We find that computing a single MTTDL using this approach with realistic parameters can take up to several seconds using the Mathematica 12 software [152] on a desktop PC. This is too slow in practice, because not only do we need to compute the MTTDL when creating new stripes, but we also need to periodically compute the MTTDL of every stripe in the system (typically billions) as device AFRs change. The next section describes an efficient approximation that makes the MTTDL calculation of eclectic stripes computationally tractable and highly accurate.

Efficient and accurate MTTDL approximation

In order to compute and better understand the MTTDL of eclectic stripes, we propose an approximation formula, building on the approach presented in [153] for

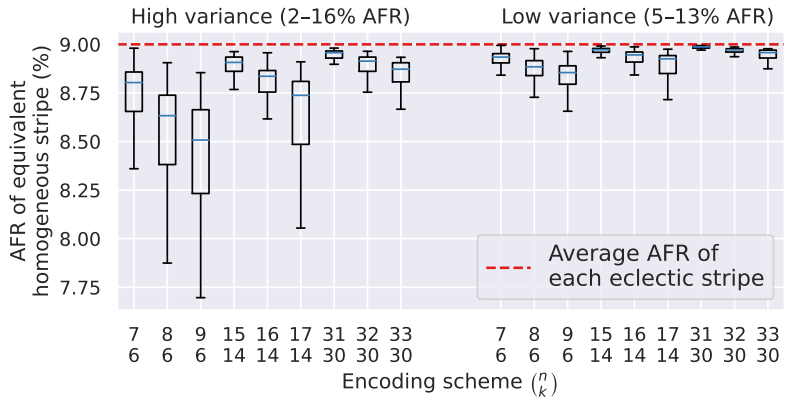


Figure 5.16: Reliability of eclectic stripes compared to homogeneous stripes. For each scheme, we sample 1000 eclectic stripes and for each stripe we compute its MTTDL ρ and then compute the AFR λ of a homogeneous stripe with the same scheme and MTTDL equal to ρ. The boxes show the distribution of λ over the 1000 stripes. The AFR of the first $n-1$ disks in a eclectic stripe are sampled uniformly at random from the range 2–16% (high variance) or 5–13% (low variance), and the AFR of the last disk in a stripe is chosen to ensure that the average AFR of the disks *in each stripe* is fixed at 9%. E.g. the median 6-of-9 eclectic stripe from the high-variance group is as reliable as a 6-of-9 homogeneous stripe with AFR 8.5%, despite having an average AFR of 9%.

homogeneous stripes. This approximation is extremely good when $\mu \gg \max_i \lambda_i$, which is true for modern cluster storage systems.

The main idea behind this approximation is to note that (in the steady state) disk i will be available a fraction $A_i = \mu/(\mu + \lambda_i)$ of the time, and that the system will reach the DL state when exactly $k-1$ of the disks are available. Therefore, the MTTDL can be approximated with the following formula (see Section 5.16 for the full derivation):

$$\text{MTTDL} \approx \left(\mu(n-k+1)\,\text{PBin}(k-1; n, (A_i)_{i=1}^n)\right)^{-1}, \tag{5.5}$$

where $\text{PBin}(k; n, (p_i)_{i=1}^n)$ is the probability of obtaining exactly k heads when flipping n biased coins with probability of heads p_i for coin i. PBin is known as the Poisson-binomial distribution, and it can be efficiently evaluated [154, 155].

We tested this approximation against the Markov chain approach over all values of $6 \leq k \leq 30$, $1 \leq n-k \leq 3$, and AFRs of 1–16%. The relative difference between the two output MTTDLs never exceeded 5% and was less than 0.5% on average[8]. As a benefit, the approximation is 2–4 orders of magnitude faster to evaluate (in the order of milliseconds), more numerically stable, significantly simpler to implement, and gives direct insight into how the parameters affect MTTDL.

5.13.2 Understanding MTTDL of eclectic stripes

The main difference between the reliability of an eclectic stripe and a homogeneous stripe is given by the Poisson-binomial factor in Equation (5.5), which becomes Binomial when all probabilities are equal. Notice that the difference between A_i in Equation (5.5) will be small because $\mu \gg \max_i \lambda_i$, and therefore the corresponding Poisson-Binomial distribution will not deviate too much from a Binomial distribution

[8]The median relative difference between the exact and approximated eclectic stripe MTTDL was 0.1%, the 90^{th} percentile error was 0.5%, and the 95^{th} percentile error was 0.7%.

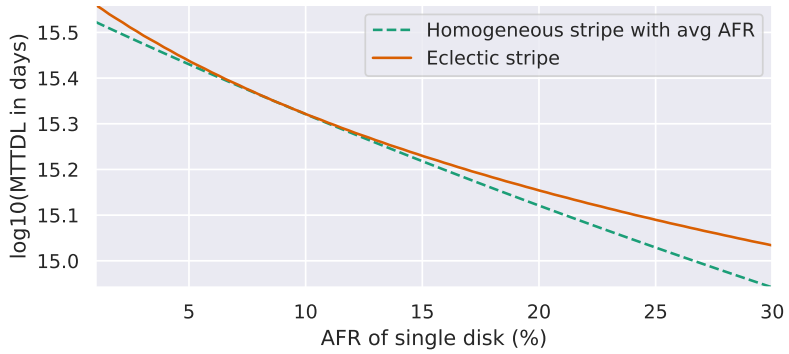


Figure 5.17: Reliability of a 6-of-9 eclectic stripe when the AFR of a single disk varies. The eclectic stripe is composed of 8 devices with AFR 9%, and one device whose AFR varies from 1% to 30% (x axis). The dashed line denotes the MTTDL of a 6-of-9 homogeneous stripe with the same average AFR as the eclectic stripe. The solid line denotes the MTTDL of the eclectic stripe. Reliability of the eclectic stripe is always above the corresponding homogeneous stripe.

with trial success probability $A = \sum_{i=1}^{n} A_i/n$ [156]. Furthermore, we have:

$$\sum_{i=1}^{n} \frac{A_i}{n} = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{1+\lambda_i/\mu} \approx \frac{1}{n} \sum_{i=1}^{n} \left(1 - \frac{\lambda_i}{\mu}\right) = 1 - \frac{\sum_{i=1}^{n} \lambda_i/n}{\mu},$$

where we use the approximation $1/(1+x) \approx 1-x$ for small x. This means that the reliability of an eclectic stripe will tend to be close to the reliability of a homogeneous stripe with AFR equal to the average AFR of the eclectic stripe.

To measure how close the MTTDL of an eclectic stripe will be to that of a homogeneous stripe with the same scheme and average AFR, we conduct two numerical experiments. Figure 5.16 compares eclectic stripes against homogeneous stripes that have the same MTTDL, across different schemes and AFR ranges. In this experiment, instead of directly showing an MTTDL ρ (which is hard to interpret) in the y-axis, we show the AFR λ of a homogeneous stripe that has MTTDL equal to ρ (under the relevant scheme). The results show that eclectic stripes are *more* reliable than homogeneous stripes with the same scheme and average AFR. In other words, for a homogeneous stripe composed of disks with AFR λ to match the reliability of an

eclectic stripe with AFRs $(\lambda_i)_{i=1}^n$, the disks in the homogeneous stripe have to be more reliable on average, i.e., $\lambda < \sum_{i=1}^n \lambda_i/n$. The difference, however, becomes small when the ratio n/k is small, or the range of AFRs is small. Figure 5.17 shows the reliability of an eclectic stripe when the AFR of a single disk in the eclectic stripe varies in the range 1–30%. This experiment shows that eclectic stripes provide a dampening effect against AFR rises of a small number of devices in two ways: (1) a small number of devices have a smaller impact on the average AFR of the stripe (slope of the dashed line), and (2) the convex shape of the curve shows that the eclectic stripe is even more reliable than a homogeneous stripe with the same scheme and average AFR.

Checking if a stripe is safe: Typically, a minimum level of reliability is set in the cluster by setting a *MTTDL threshold* that all stripes must satisfy in order to be deemed safe. Given the results presented in this section, we now describe a simple method to determine whether a stripe is safe. We define the *critical AFR* of a k-of-n scheme and MTTDL threshold θ as the highest AFR that disks in a homogeneous k-of-n stripe can attain while still having an MTTDL of at least θ. The critical AFRs for the different schemes that are used in a system can be precomputed and stored. Then, a simple andx efficient way of checking whether an eclectic stripe under some scheme is safe is to check whether the average AFR in the stripe is less than the critical AFR for that scheme. Since an eclectic stripe is at least as reliable as a homogeneous stripe with the same scheme and average AFR, if the stripe passes this check, then we can be certain that the stripe is safe. If the stripe does not pass the check, then it *may* be unsafe, which can determined by computing its MTTDL. This test can help greatly reduce the amount of work needed in checking whether stripes are still safe, and it also provides a simple way of understanding the reliability of eclectic stripes.

5.13.3 Eclectic Volumes

Disk AFR changes may trigger redundancy transitions. Prior designs performed disk-adaptive redundancy at the disk level. Thus, if a disk's AFR changed, either all

or none of the stripes on that disk required a redundancy transition. With eclectic stripes, each disk may store chunks of stripes with different reliabilities. An AFR change might only require redundancy transitions for a subset of those stripes. With millions of eclectic stripe chunks being stored on each disk, a linear search through all of them for each AFR change is impractical.

An eclectic volume is a collection of eclectic stripes that use the same EC scheme and are stored on the same set of disks. A disk can contain multiple volume fragments identified by their globally unique volume ID. Each disk maintains a map of stripe ID to eclectic volume ID. Since each eclectic volume spans the exact same disks, whenever a disk's AFR changes, Tiger only needs to check whether the EC scheme used for each of the disk's constituent volumes still meets the required MTTDL target. There is no need to check the reliability of each of the individual eclectic stripes within a volume since they are all identically reliable. The details of how Tiger manages eclectic volumes is described in Section 5.14.3.

Eclectic volumes prove to be efficient only if they represent a large number of eclectic stripes. Therefore, in Tiger the default size of an eclectic volume is set to 1 TeraByte (TB). This way, even though Tiger performs reliability monitoring at the volume granularity it ensures that each eclectic stripe is always sufficiently reliable.

5.14 Design and working of Tiger

Tiger is a practical disk-adaptive redundancy system designed to overcome the challenges described in Section 5.12. Figure 5.18 shows the architectural components of Tiger (colored boxes) and how they interact with existing cluster storage system components and common disk-adaptive redundancy components.

5.14.1 Data flow in Tiger

We overview Tiger by explaining the lifecycle of eclectic stripes. An eclectic stripe is created via the *Eclectic Stripe Allocator* (ESAllocator), which identifies a set of disks and the corresponding scheme on which this data is to be stored. The

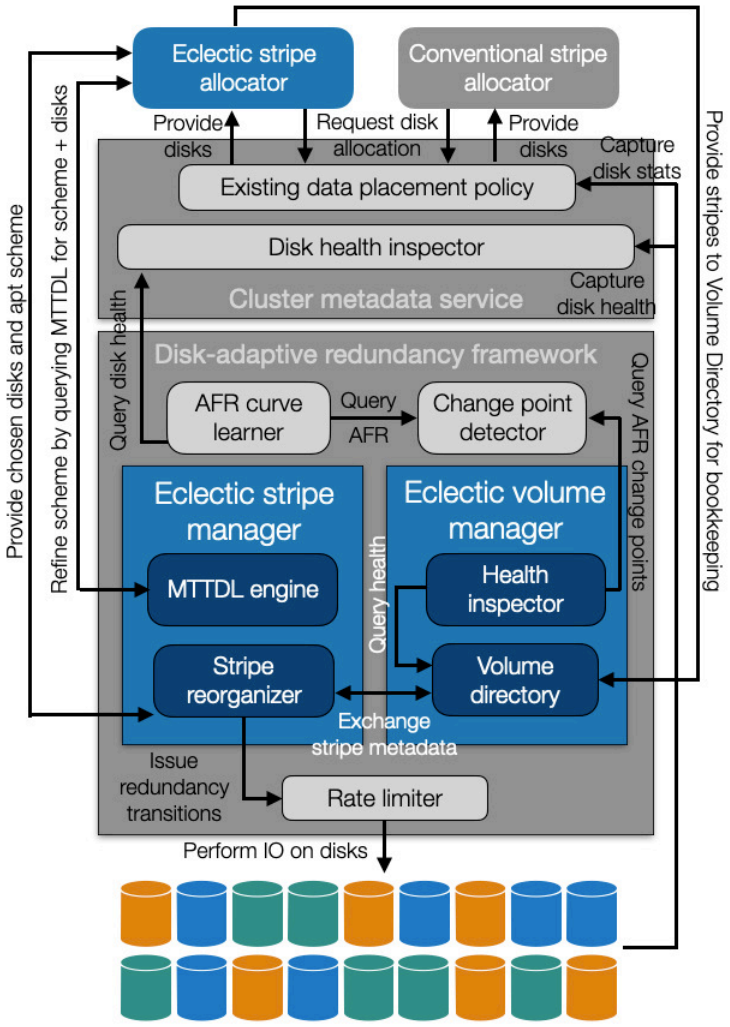


Figure 5.18: Architecture of Tiger. The blue boxes correspond to Tiger's components. The gray boxes correspond to existing components in cluster storage system architecture and components present in existing disk-adaptive redundancy systems.

ESAllocator uses the existing and unmodified data placement policy to obtain a set of disks. That placement policy uses whatever knowledge designers choose (e.g., available freespace, load balance, and fault domain constraints) in selecting the set of disks. The ESAllocator then queries the *Eclectic Stripe Manager's MTTDL Engine* (ESMTTDLEngine) with the AFRs of the chosen disks, and a stripe configuration, to verify that the planned stripe's MTTDL meets the required target MTTDL. If it does not, the ESAllocator boosts the MTTDL by changing the stripe configuration until an appropriately safe redundancy scheme is found. Section 5.14.2 details this process.

Once created, the ESAllocator passes the stripe to the Eclectic Volume Manager (EVManager, see Section 5.14.3) to either add the stripe to an existing volume, or create a new volume which will contain the new stripe. The Eclectic Volume Health Inspector (EVHInspector) continuously monitors the reliability of the eclectic volume by querying the change point detector, which identifies significant AFR changes in the data from the AFR curve learner. The AFR curve learner, change point detector and the rate limiter can be reused without change from any existing disk-adaptive redundancy system[9]. In reaction to a significant AFR change (rise or fall), the EVHInspector alerts the EVManager, which fetches the eclectic stripe metadata from the EVDirectory and provides both the AFR change and the metadata to the Eclectic Stripe Reorganizer (ESReorganizer; see Section 5.14.2). The ESReorganizer includes techniques to efficiently perform redundancy transitions. If eclectic stripes must change, the ESReorganizer consults the ESAllocator in forming them. Non-urgent redundancy transitions (when the target MTTDL is not at risk of being violated) are throttled by the rate limiter in order to not overwhelm the storage cluster.

Tiger's stripe-by-stripe disk-adaptive redundancy approach enables incremental adoption by allowing data to be stored either as an eclectic stripe or a homogeneous stripe. This is in contrast to subcluster-based designs that are all-or-nothing.

[9]Tiger reuses the Ruptures change-point detection library [157, 158], the AFR curve-learner and the rate-limiter from HeART [1] and Pacemaker [15].

5.14.2 The Eclectic Stripe Manager

The Eclectic Stripe Manager (ESManager) handles construction, maintenance and reorganization of eclectic stripes.

Constructing eclectic stripes. In the absence of an existing eclectic volume that has space (described later in Section 5.13.3), the ESAllocator asks the existing data placement policy for disks to store each new eclectic stripe. Since that placement policy is unaware of disk-adaptive redundancy, it may return a set of disks whose AFRs produce an MTTDL that either fails to meet or far exceeds the target MTTDL. Algorithm 1 describes the process to build a space-efficient, yet adequately reliable eclectic stripe.

Algorithm 1

$\theta_{\text{MTTDL}} \leftarrow$ target MTTDL
$n_{\max} \leftarrow max\{n \mid (n,k) \in \text{schemes}\}$
$(d_1, \ldots, d_{n_{\max}}) \leftarrow n_{\max}$ randomly sampled devices
for $(n,k) \in$ schemes in order of increasing n/k **do**
  if $\text{MTTDL}(n,k,(d_1,\ldots,d_n)) \geq \theta_{\text{MTTDL}}$ **then return** (n,k)

To give itself flexibility, ESAllocator asks the placement policy to provide a set of disks for the maximum-width-allowed stripe (e.g., 33 for 30-of-33). The ESAllocator then queries the ESMTTDLEngine with the provided disks and its planned scheme to get the MTTDL value. If the MTTDL does not meet the target MTTDL, ESAllocator discards a disk from the set and increases the redundancy of the corresponding scheme (e.g., 29-of-32 instead of 30-of-33) to boost the stripe's MTTDL, repeating this process until sufficient MTTDL is achieved. This process is guaranteed to terminate, since the least space-efficient scheme in a storage cluster must meet the target MTTDL. Moreover, by iterating from the most space-efficient scheme allowed, the algorithm terminates at the most space-efficient scheme for the provided disks.

Ensuring reliability amid disk failures. The reliability of each eclectic stripe is a function of the AFRs on the disks on which it is stored. So, when a disk fails, the reconstructed data cannot simply be placed on a randomly chosen disk, since its AFR might be high enough to cause the eclectic stripe's MTTDL to exceed the target.

Recall, from Section 5.13.2, that the critical AFR of an EC scheme is the highest AFR that a homogeneous stripe of that scheme can reliably support, and a simple way to test that an eclectic stripe is safe is to check that its average AFR is below the critical AFR for its EC scheme. Therefore, we can ensure that reliability will be preserved if we choose a disk that keeps the average AFR of the affected stripes under their respective critical AFRs.

When a disk in Tiger fails, the EVManager is notified. This triggers a lookup in the EVDirectory for eclectic stripes whose chunks need to be reconstructed. The EVManager forwards the list of chunks to the ESReorganizer. For each stripe, the ESReorganizer asks the ESAllocator for disks to replace the failed disks, providing the critical AFR for the stripe. The ESAllocator returns suitable disks, if they are found, otherwise, it allocates (one or more) new eclectic stripes and moves the prior stripe's data (including any reconstructed data) to the new stripes. Finding sufficiently reliable disks to store the reconstructed data results in lower transition IO than allocating new eclectic stripes, since the latter involves moving data of disks that did not fail. After the reconstruction process (whether or not new eclectic stripes are formed), ESReorganizer informs the EVManager of the changes, which then updates the EVDirectory accordingly.

Dealing with AFR changes over time. A disk's AFR is not constant throughout its lifetime [148, 159–161]. In addition to building and maintaining eclectic stripes, ESManager must also ensure that data is kept safe when a disk's AFR changes.

Ensuring data reliability with increasing AFRs. The EVManager monitors AFR by querying the change point detector. Whenever the AFR rises, the EVManager identifies any eclectic volumes whose data is at a risk of becoming under-reliable. It alerts the ESReorganizer, with the necessary stripe metadata of such stripes, which calls the ESAllocator with the current and previous disk AFR values and the number of chunks that need reallocation onto safer disks.

As with failed data reconstruction, ESAllocator prefers finding suitable disk alternates whose AFRs are less than or equal to previous AFRs values of the disks whose AFRs rose. If ESAllocator cannot find suitable disks, new eclectic stripes are

formed and data is moved, as described previously.

Reducing data over-protection with reducing AFRs. When a disk's AFR decreases, there is no reliability threat to the data stored on that disk, but there may be an opportunity to reduce redundancy and obtain space-savings.

The simplest way (that also entails no transition IO cost) of reducing a stripe's redundancy is by deleting excess parities[10]. However, deleting parities is rarely an option for two reasons. First, most storage clusters have a minimum requirement on the number of parities per stripe, set by the system administrator. Second, adding/deleting a parity has a much higher impact on the MTTDL value of a stripe than adding/deleting a data chunk—deleting even a single parity usually makes the stripe miss the target MTTDL. When ESReorganizer receives metadata of possibly over-redundant stripes from the EVManager, it queries the ESMTTDLEngine whether reducing parities is feasible and, if so, enacts the change.

When deleting parities is not an option, there are two additional ways redundancy can be reduced. First, the ESAllocator could find candidate disks with AFR higher than the current disk's AFR, but low enough that the mean AFR is below the stripe's critical AFR. This method is cost-effective, since it involves only reading and writing those chunks that are on over-protected disks. Second, if the ESAllocator cannot find suitable disks, it performs new stripe allocations if it can find a new eclectic stripe with lower storage overhead. Although re-allocation has a high IO overhead (since it involves copying data over to the new stripe), it is not urgent when lowering redundancy and can be throttled by the rate limiter without putting any data at risk.

The eclectic stripe reorganizer (ESReorganizer). The ESReorganizer uses several techniques to ensure adequate reliability and provide maximum space-savings.

At any given time, the ESReorganizer might be dealing with multiple eclectic stripes seeking possible changes. ESReorganizer processes requests in priority of maintaining reliability: failed data reconstruction, then near-risk stripes that need to increase their redundancy, then requests of decommissioning disks to move data off of them, and then stripes seeking a redundancy reduction. It processes eclectic stripes

[10]Deleting parities may not work reducing redundancy of non-MDS codes.

that are requesting reduction in redundancy in descending order of their storage overhead.

5.14.3 The Eclectic Volume Manager

The EVManager is responsible for creating, maintaining and monitoring the health of eclectic volumes. Recall (from Section 5.13.3) that an eclectic volume (typically in TBs) contains hundreds-of-thousands of eclectic stripes (typically in MBs). Along with health, the EVManager maintains usage statistics (e.g., freespace and load) for each eclectic volume.

Constructing and populating eclectic volumes. Similar to how ESManager manages eclectic stripes, EVManager dynamically creates and destroys eclectic volumes. The construction of the first eclectic stripe forces the creation of the first eclectic volume on the same set of disks that are chosen by the ESAllocator. When creating subsequent eclectic stripes, the ESAllocator first queries the EVManager to check if there are eclectic volumes that are conducive for storing new stripes. The EVManager does this by maintaining capacity and load-balancing metrics for each eclectic volume. Thus, the EVManager also avoids hot-spotting within eclectic volumes by spreading hot data evenly across multiple eclectic volumes. Once the target eclectic volume is identified, the set of disks comprising the eclectic volume are returned to the ESAllocator. If there is no space available, the ESAllocator gets a new set of disks from the placement policy which causes EVManager to create a new eclectic volume atop those disks. Tiger's eclectic volumes operate similar to Ceph's placement groups [129].

The Eclectic Volume Directory. Recall from Section 5.13.3 that eclectic volumes are simply a logical grouping of all the eclectic stripes with the same redundancy scheme on the same set of disks. Each eclectic volume has a unique entry in the EVDirectory and stored against the eclectic volume ID are the disks on which the eclectic volume is stored. In addition, the EVDirectory also contains a mapping from disk serial number to list of volume IDs whose fragments are stored on that disk. Note that the size of this metadata is very small. With TB-sized volume fragments,

even a 100K disk storage cluster with 20TB disks will have an EVDirectory less than 100MB.

The tiny size of the EVDirectory also implies that it is unlikely to be a bottleneck. The EVDirectory will typically be queried and updated whenever disks fail, or their AFR increases significantly (in order to fetch the eclectic volumes IDs stored on the affected disks). It might also be queried to fulfill an allocation request in order to get the disks on which an eclectic volume is stored, if the eclectic-volume-to-disks mapping is not cached. Even a cluster with 500K disks has at most a few hundred disk failures in a day and typically not more than 10 makes/models, thus limiting the EVDirectory updates to less than 1000 per day. Although allocations are more frequent, caching can filter most queries for them, and their rate is also much lower than the rate of file metadata lookups in a cluster with billions of files. And, if necessary, traditional metadata scaling techniques can be employed to prevent EVDirectory from becoming a bottleneck.

Reacting to failures and AFR changes. The EVHInspector continuously polls the change point detector and the cluster metadata service to gather information about disk failures and significant AFR changes. For all significant changes, the EVHInspector reconfirms the MTTDL of the affected volumes by querying the ESMTTDLEngine with the changed AFRs. Even though it is technically not a stripe, a EVDirectory has all information required to calculate the reliability of an eclectic volume, viz. the AFRs of the disks on which the volume resides, and the redundancy scheme configuration. Due to its small metadata footprint, EVHInspector can check the health of billions of stripes by checking the reliability of only thousands of eclectic volumes.

Whenever a disk fails, or a disk's AFR increases, the EVHInspector looks up the EVDirectory to find the volumes affected due to this failure/AFR rise. If the disk in question is alive, the volume manager queries the disk to obtain the stripe IDs belonging to that volume ID. If the disk has failed, the EVHInspector queries other disks of that particular eclectic volume and gathers the stripe IDs from them. Note that all disks storing a particular eclectic volume have the same list of eclectic stripe IDs in common (but they also each may have other stripes as well from

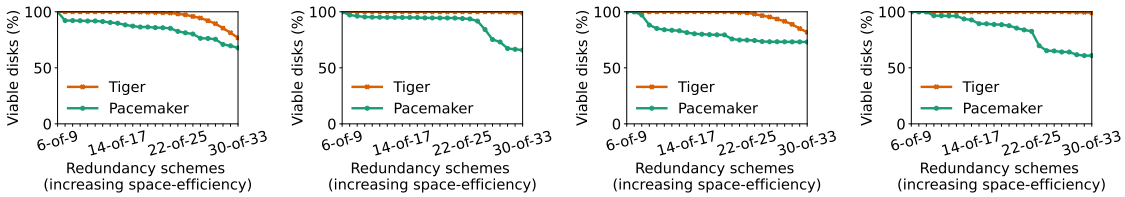
non-overlapping eclectic volumes).

The EVHInspector then forwards the list of stripe IDs to the ESReorganizer along with the updated and previous AFR information and the action to be taken (reconstruct data, increase redundancy or reduce redundancy). On performing the appropriate task, the ESReorganizer communicates the metadata changes back to the EVManager, and the EVManager subsequently reflects it in the EVDirectory. For reconstruction and increase in redundancy, if a replacement disk is found, and has enough capacity to accommodate all chunks of the failed disk/disks whose AFR has increased, the eclectic volume of all constituting eclectic stripes after the operation remains the same. For redundancy reductions, or in case of not finding a replacement disk, or not finding one with enough capacity, the eclectic stripes depart from their original eclectic volume (unlike Ceph's placement groups) since they will now be stored on potentially different subset of disks.

5.15 Evaluation of Tiger

We now evaluate how Tiger performs on real-world data, and show how it fulfills the challenges laid out in Section 5.12. Tiger is evaluated using real-world deployment and failure logs from four production clusters at two different organizations (Google and Backblaze). Each cluster has a multi-year lifetime and disks from multiple makes/models/batches. Backblaze uses *trickle*-deployed disks. These disks are added to the cluster every few days in the tens or hundreds. Google Cluster 2 and Cluster 3 have *step*-deployed makes/models where disks are introduced into the cluster in large batches of tens-of-thousands of disks within a very short span of time. Google Cluster 1 is a mix of step- and trickle-deployed disks.

The highlights of our evaluation are (1) Tiger significantly lowers placement restrictions posed by Pacemaker (existing state-of-the-art disk-adaptive redundancy system); (2) Tiger's eclectic stripes provide much higher risk-diversity compared to Pacemaker; (3) Tiger is closer to the target MTTDL, and thus more efficient than existing disk-adaptive redundancy approaches; (4) Tiger outperforms Pacemaker in



(a) Backblaze **(b)** Google Cluster 1 **(c)** Google Cluster 2 **(d)** Google Cluster 3

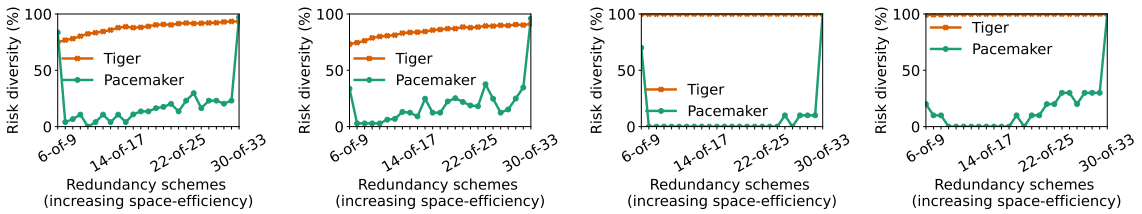
Figure 5.19: Placement constraints posed by Tiger compared to Pacemaker by observing the percentage of the disk fleet that is viable for the different redundancy schemes. Tiger has lower placement constraints than Pacemaker. Tiger has over >75% disks being viable for all four clusters for all scheme configurations. Pacemaker's placement constraints are more pronounced in Google clusters since they are mostly step-deployed. This results in strict Rgroup boundaries disallowing disks from different makes/models being a part of the same Rgroup.

space-savings while keeping the average transition IO $\leq 0.5\%$ and peak transition IO $< 5\%$ of cluster IO bandwidth and (5) Tiger's eclectic stripes are less sensitive to rising AFR and provide better data safety.

5.15.1 Tiger enables flexible data placement

We capture the flexibility in data placement by measuring the percentage of the disk fleet that is considered viable for storing data using a particular redundancy scheme. The viability is decided by whether the data stored on those disks will meet the target MTTDL. The X-axis in Figure 5.19a shows the various schemes that can be supported in each storage cluster[11]. For estimating Tiger's viable disk candidates, we perform a Monte-Carlo simulation on specific days in each of the cluster's lifetime. We allocate 1000 eclectic stripes by picking disks uniformly at random and check how many of the possible schemes can use the chosen disks. For Pacemaker, we bin the disks by AFRs to mimic Rgroups and measure the ratio of the population of the Rgroups to the entire disk fleet.

[11]The narrowest scheme is set to 6-of-9 and widest is set to 30-of-33. Schemes with higher width have lower redundancy since the number of parities are kept the same. This is based on reference to prior work [1, 15], and also on the basis of communication with storage administrators of large-scale cluster storage systems at various organizations.



(a) Backblaze risk-diversity **(b)** Google Cluster 1 risk-diversity **(c)** Google Cluster 2 risk-diversity **(d)** Google Cluster 3 risk-diversity

Figure 5.20: Risk-diversity achieved by Tiger over three large-scale cluster storage systems. All three plots are average risk-diversity measurements taken over 5 days spread equally over the lifetime of the clusters. Pacemaker due its Rgroup based design has much lower risk-diversity compared to Tiger, more evident in Figures 5.20c and 5.20d which are entirely step-deployed clusters.

Tiger has almost all disks available for allocation for any scheme in Google Clusters 1 and 3 (Figures 5.19b and 5.19d), whereas in Backblaze and Google Cluster 2 (Figures 5.19a and 5.19c) at most 25% disks are deemed not viable for the widest schemes (beyond 22-of-25). When a large fraction of disks of the cluster have a high AFR (as is the case with Backblaze and Google Cluster 2 for the chosen dates), formation of eclectic stripes ends up with mostly high AFR disks. In such situations, Tiger cannot employ a very space-efficient redundancy scheme. Pacemaker's strict Rgroup boundaries, on the other hand, limit all disks in an Rgroup to a single scheme that may not be very wide. Therefore, for Pacemaker, all clusters see a significant drop in viable disks as the width increases.

5.15.2 Tiger achieves high risk-diversity

Risk-diversity of a stripe is directly proportional to the number of unique makes/models participating in that stripe. If all makes/models in the storage cluster have representation in the stripe, its risk-diversity is defined to be 100%. A 0% risk-diversity implies that there were no disks in the cluster that could be used for the particular scheme. The setup used for evaluating risk-diversity is a Monte-Carlo simulation, where 100 stripes were allocated for each scheme configuration by choosing disks

uniformly at random. For Tiger, we measure risk-diversity by capturing the average number of unique disk makes/models on which the chunks of an eclectic stripe are stored for each stripe configuration. For Pacemaker, we again bin the disks by AFR to form Rgroups, and count the unique number of makes/models within each Rgroup. We take the average of this simulation performed on five equally spaced days in the cluster lifetime to get an overall sense of risk-diversity of both systems.

Tiger significantly outperforms Pacemaker in providing high risk-diversity. Figure 5.20 captures the risk-diversity achieved by Tiger vs Pacemaker. Since Tiger has no partitioning of disks, all disks of any make/model are viable for allocating any scheme. The minimum risk-diversity achieved by Tiger is 60% across all four clusters, that too for the narrowest scheme (6-of-9) for Backblaze (Figure 5.20a) and Google Cluster 1 (Figure 5.20b) clusters. Both these clusters have seven makes/models, and it is unlikely that seven out of nine chunks will be across different makes/models. As the stripe width increases, Tiger's risk-diversity also improves. Entirely step-deployed clusters, Google Cluster 2 (Figure 5.20c) and Google Cluster 3 (Figure 5.20d) have four and three makes/models respectively. Tiger achieves perfect risk-diversity for all possible schemes in those clusters. For Pacemaker, it is more likely that clusters where all makes/models are trickle-deployed will have a better risk-diversity because multiple makes/models can be a part of the same Rgroup so long as their AFRs are in the same range, for e.g. Backblaze (Figure 5.20a). Nevertheless, even clusters with all trickle-deployed disks do not see perfect (or even good) risk-diversity since different makes/models are deployed at different times, and they go through different phases of life at different dates. Risk-diversity is poorer for Pacemaker in clusters with step-deployed makes /models as seen in Figures 5.20c and 5.20d. This is because Rgroups and steps have a 1:1 mapping and each step only contains disks of a single make/model. The reason Pacemaker has 100% risk-diversity for 30-of-33 is because when averaging over multiple days (5 for this experiment), all makes/models on some date belonged to an Rgroup with the 30-of-33 redundancy scheme.

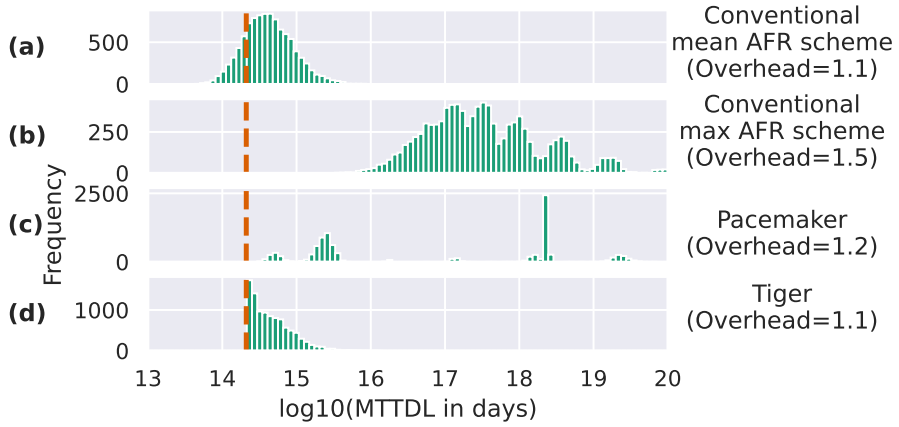
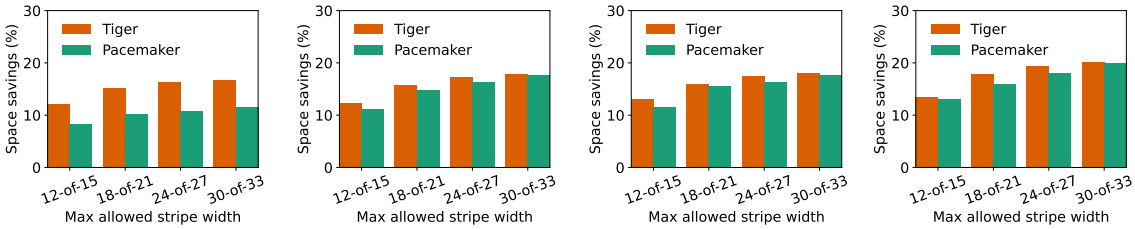


Figure 5.21: Comparison of MTTDL distributions for different approaches. We form 10000 random stripes for each approach using the AFRs from Google Cluster 1 (notice the different scales in the Y-axis). In a conventional system, a single scheme is chosen for all stripes based on the average AFR (a) or maximum AFR (b). (c) In Pacemaker, stripes must reside within an Rgroup, and the scheme depends on the Rgroup. (d) In Tiger, the scheme for each stripe is chosen based on the AFRs in the stripe. The dashed vertical line denotes the target MTTDL.



(a) Backblaze space-savings **(b)** Google Cluster 1 space-savings **(c)** Google Cluster 2 space-savings **(d)** Google Cluster 3 space-savings

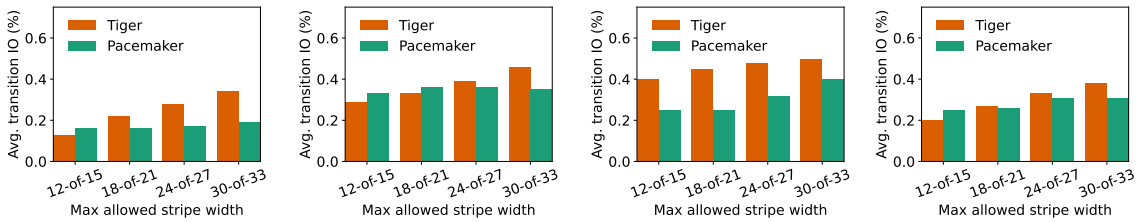
Figure 5.22: Space-savings achieved by Tiger for disk-adaptive redundancy simulated on four production clusters compared to Pacemaker over conventional one-scheme-fits-all redundancy approaches. Figures 5.22a to 5.22d show that across all clusters with different maximum stripe width configurations, Tiger provides up to 5% higher average space-savings compared to Pacemaker.

5.15.3 Tiger adapts redundancy efficiently

The efficacy of disk-adaptive redundancy performed by Tiger is evaluated using three metrics. First, we discuss the MTTDL distribution of data stored using Tiger. Subsequently, using the same four clusters used by Pacemaker we evaluate the resulting space-savings obtained by Tiger because of disk-adaptive redundancy, and finally we measure the IO overhead needed to perform necessary redundancy transitions. For fair comparison, when evaluating Tiger, we employ the same configurations (such as the IO constraints and permitted redundancy schemes) and tools (such as the AFR curve learner and the change-point detector) that are used in Pacemaker.

Tiger's achieves tight reliability. Storage clusters have to ensure that all data in the cluster always meets a specified target level of reliability typically specified as a MTTDL value. Tiger's target MTTDL is set as the lowest acceptable MTTDL in the system. This is calculated using the MTTDL of the most conservative homogeneous stripe possible (6-of-9) having the maximum possible AFR (16%). These settings are borrowed from Pacemaker's evaluation for a fair comparison with Tiger.

Figure 5.21 shows a comparison in the distribution of stripe MTTDL with different approaches to redundancy selection for a specific day in Google Cluster 1. Figure 5.21a shows conventional systems choosing the redundancy scheme based on the avg. AFR,



(a) Backblaze IO overhead **(b)** Google Cluster 1 IO overhead **(c)** Google Cluster 2 IO overhead **(d)** Google Cluster 3 IO overhead

Figure 5.23: IO overhead of redundancy scheme transitions of Tiger versus Pacemaker. In most configurations, Tiger has a higher IO overhead compared to Pacemaker due to Pacemaker leveraging its IO-efficient transitioning mechanisms. Despite being higher, the average IO overhead of Tiger is still at most 0.5% of the overall cluster's IO bandwidth; much lower than existing background tasks such as scrubbing, that require approximately 7% IO bandwidth [137]

which results in small storage overhead, but puts a big fraction of the stripes at risk. Figure 5.21b shows conventional systems that choose the redundancy scheme on the basis of max AFR. Although all stripes are sufficiently protected, the storage overhead is the highest among all four alternatives. Figure 5.21c shows Pacemaker where the different MTTDL clusters represent different Rgroups with different redundancy schemes. Pacemaker achieves good reduction in storage overhead, and keeps all stripes above the target MTTDL. In fact, some Rgroups (with higher MTTDL values) are too over-protected and denote lost opportunities for space-savings. Finally, Figure 5.21d shows Tiger's MTTDL distribution. Despite all its eclectic stripes being above the MTTDL threshold, Tiger has least storage overhead.

Tiger achieves attractive space-savings. Akin to Pacemaker, by dynamically tailoring redundancy to disk AFRs, Tiger's eclectic stripes can use more space-efficient redundancy schemes to meet the required MTTDL target. Figure 5.22 shows that Tiger achieves equal or better average space-savings compared to Pacemaker in all four clusters. For Google Clusters 1, 2 and 3 (Figures 5.22b to 5.22d), the highly cost-efficient redundancy transitions of Pacemaker allows a large step-deployed make/model to spend more time in lower redundancy. This boosts Pacemaker's overall space-savings for these clusters and prevents Tiger from surpassing it easily.

In the Backblaze cluster (Figure 5.22a), the reason for Tiger achieving better space-savings is because eclectic stripes allow high AFR disks to be mixed with low AFR disks and yet use an optimized redundancy scheme. In Pacemaker, high AFR disks cannot be mixed with other disks, resulting in lower space-savings. In the Backblaze cluster, all the seven makes/models are trickle-deployed. This results in a non-trivial fraction of disks constantly being in high-AFR regimes of infancy or wearout. While Pacemaker is forced to use the default, most conservative redundancy scheme on these disks, Tiger can use these disks for more space-efficient redundancy schemes by combining them with other, more robust disks. As a result, Tiger is able to achieve up to 5% higher space-savings compared to Pacemaker.

Tiger has very low IO overhead. Figure 5.23 shows the IO overhead comparison between Pacemaker and Tiger. Although both systems are capped at 5% and in general require very low IO (compared to background tasks such as scrubbing that requires $\approx 7\%$ [137]), our evaluation shows that Tiger can achieve all its benefits with an average IO bandwidth required for redundancy transitions of at most 0.5%. In an absolute sense, Tiger's low IO overhead is mainly attributed to Tiger's efficient redundancy transitions for an AFR rise (detailed in Section 5.14.2), where Tiger moves the potentially risky chunk from an unsafe disk to a safe disk rather than re-encoding it or reallocating it; both having a significantly higher IO cost.

Compared to Pacemaker, Tiger still incurs slightly higher IO overhead. This is due to Tiger's mechanism of coalescing space-inefficient (high-redundancy) eclectic stripes into new space-efficient (low-redundancy) eclectic stripes in response to AFR reduction by moving all chunks. It leads to more data movement compared to moving just the chunks of the high-AFR disks (as is the case when AFR rises). This is a conscious design choice made in Tiger in order to maximize space-savings for non-urgent redundancy transitions at the expense of a minor increase in the IO overhead. Moreover, Pacemaker's IO-efficient redundancy transitioning mechanisms (that are more suitable for its Rgroup-based design) further help in reducing its IO overhead.

Tiger does not experience urgent IO bursts. In order to understand the burstiness of the IO that can be experienced by Tiger compared to Pacemaker, we artificially increase the AFR of a make/model and measure the resulting transition

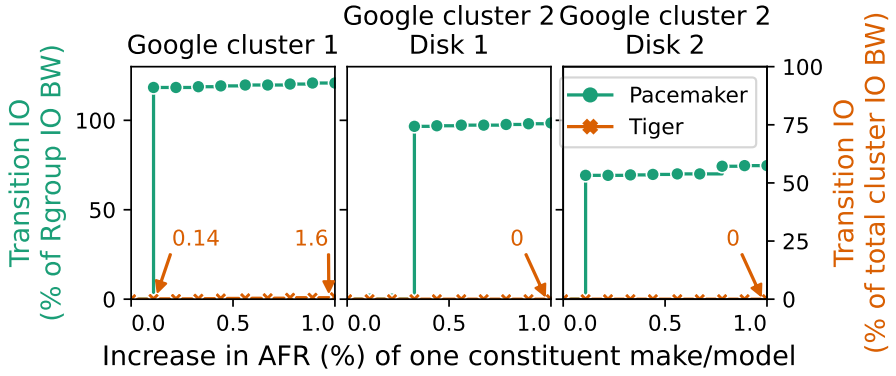


Figure 5.24: IO cost of redundancy transitions associated with the increase of AFR for one constituent make/model. IO cost is measured as a percentage of the total IO bandwidth of the Rgroup for Pacemaker, whereas it is the total cluster IO bandwidth for Tiger. It is calculated by scaling up a simulation of 1000 random stripes in each system and measuring the number of stripes that become unsafe after the given increase in AFR.

IO load for maintaining data reliability. Figure 5.24 shows the comparison of IO loads experienced by Pacemaker vs Tiger for three instances of increasing AFR of a single step-deployed make/model. Performed on three different dates in two Google clusters (Cluster 1 and Cluster 2), we observe that Pacemaker needs orders of magnitude higher IO bandwidth than Tiger to achieve the required transitions. In fact for Google Cluster 2, in both instances none of Tiger's stripes needed transitioning despite observing a 1% rise in AFR.

We explain Pacemaker's high IO requirement with an example. Suppose a 20TB disk, which can perform 100MB/s needs to transition away from using a 30-of-33 scheme. Despite using Pacemaker's optimized Type 2 transitions[12], simply reading the data to recalculate new parities would require 196% of the disk's possible IO bandwidth in a day (assuming 90% fullness to match Pacemaker's setup). In a step-deployed make/model all disks of an Rgroup transition together. In order to spread out the resulting IO burst over time, Pacemaker relies on predicting the AFR rise well in advance. To maintain a 5% IO cap, Pacemaker would need to know the

[12]In Type 2 transitions, Pacemaker re-encodes data from one scheme to another without re-writing any data. It simply recalculates new parities, writes them and deletes the old ones.

AFR rise at least 40 days in advance. Long-term AFR predictions are both non-robust and non-trivial.

In contrast, Tiger for the same transition does not suffer from any IO bursts. Firstly, because of eclectic stripes, even if the disk AFR increases, only a limited fraction of data stored on it will need a redundancy transition, since other stripes might be residing on more robust disks and might continue to meet the target MTTDL. Secondly, other disks over which the eclectic stripes needing an increase in redundancy are spread need not (and probably will not) belong to the same make/model/batch. Therefore, they will not require a simultaneous increase in redundancy and can assist in transitioning data from the affected stripes. Thus disks in Tiger are spared from any sudden IO bursts.

5.15.4 Challenging situations for Tiger

There are certain situations that create fundamental challenges for Tiger and other disk-adaptive redundancy systems.

Sudden rise in AFRs mimicking bulk failures. Although Figure 5.24 shows that Tiger is robust to AFR rises in any make/model in a cluster, there could be bulk failure scenarios where large fraction of the disks in the cluster fail together. On such occasions, any system (including Tiger) that depends on redundancy will suffer from potential data loss unless the system includes cross-cluster redundancy.

A cluster with a single step-deployed make/model. Suppose a cluster had only one make/model, deployed in a step-deployed manner (note: we have not come across such an example for the large clusters targeted): there would be no diversity to exploit and all disks of the cluster would undergo redundancy transitions together. Not only would this produce bursty IO, but also will potentially result in a capacity crunch (when increasing redundancy). Such clusters would either need to keep some space unutilized to account for the bulk redundancy-increasing transitions, or will need to make provisions to add more disks to the cluster before the redundancy-increasing transitions are issued.

5.16 Derivation of approximation of MTTDL of eclectic stripes

In order to approximate the MTTDL of an eclectic stripe, we will assume that the stripe can be repaired in the data loss state and we will approximate the MTTDL as the mean time between visits to the data loss state. In particular, we will analyze the stripe as an alternating renewal process. Let A_s be the stripe availability (i.e., the fraction of the time that the stripe is not in the data loss state), μ_s be the repair rate in the data loss state, and λ_s the stripe data loss rate. As described above, the MTTDL is approximately λ_s^{-1}. For an alternating renewal process, we have that:

$$A_s = \frac{\mu_s}{\mu_s + \lambda_s} \iff \frac{1}{\lambda_s} = \frac{A_s}{\mu_s(1 - A_s)} \tag{5.6}$$

The repair rate in the data loss state is simply the number of failed disks in that state:

$$\mu_s = (n - k + 1)\mu. \tag{5.7}$$

We assume that each disk in the stripe fails independently from the rest, and that it is repaired with rate μ if it fails. Then, in steady state, disk i is available with probability:

$$A_i = \frac{\mu}{\mu + \lambda_i}. \tag{5.8}$$

Let $P(j)$ be the probability that we find the stripe in a state where exactly j disks are available in the stripe. Since there are no states with more than $n - k + 1$ failed disks, we have that:

$$P(j) = \frac{Q(j)}{Q(k-1) + \cdots + Q(n)}, \text{for } k - 1 \leq j \leq n, \tag{5.9}$$

where $Q(j)$ is the probability that exactly j disks are available. Since disks are independent, $Q(j)$ is equal to a Poisson-binomial distribution, with probabilities

$(A_i)_{i=1}^n$. Given this, the availability of stripe is given by:

$$A_s = P(k) + \cdots + P(n). \tag{5.10}$$

Thus, we have:

$$\frac{1}{\lambda_s} = \frac{Q(k) + \cdots + Q(n)}{\mu(n-k+1)Q(k-1)} \approx \frac{1}{\mu(n-k+1)Q(k-1)}. \tag{5.11}$$

Where the approximation comes from the fact that $Q(n) \approx 1$ because $\mu \gg \max_i \lambda_i$ and thus all A_i are close to 1.

In summary, we have that:

$$\text{MTTDL} \approx \frac{1}{\mu(n-k+1)Q(k-1)}. \tag{5.12}$$

5.17 Related Work

The closest related work is HeART, which we have discussed several times throughout this chapter. Additional related work can be divided into works that study the reliability of disks and distributed storage, and systems that manage multiple EC schemes and transitions between them. One essential part of disk-adaptive redundancy is the monitoring of disk AFRs, which are used by Tiger to assess the reliability of stripes. Many works have studied the behavior of disk AFRs and their impact on distributed storage reliability [21, 126, 127, 134, 137, 138, 162–166]. Also, multiple works have studied the prediction of disk AFRs based on different features [151, 167–172].

Many existing distributed storage systems allow for multiple EC schemes to coexist in the same cluster [141, 173]. There are systems that propose choosing different EC schemes for different data [93, 174]. The problem of transitioning data from one EC scheme to another has been widely studied in the Coding Theory literature, with many works studying its cost, as well as proposing special code designs that reduce the cost of transitions [19, 63, 83, 86, 87, 89, 91–93, 95, 96, 99, 175]. Such designs

could be used with Tiger, though our evaluations indicate that transition IO is not a significant problem.

Part II

Dynamic storage codes for change across space

The second part of this thesis deals with *changes in storage codes across space.* In particular, we focus on the challenge of designing storage codes and systems that can adapt to differences in density across geographic regions. In Chapter 6, we design *minimum-update cost* codes, a new type of storage code designed for minimizing storage overhead and WAN bandwidth usage in geo-distributed storage systems. In Chapter 7, we propose Pudu, a strongly-consistent geo-distributed storage system that leverages minimum-update cost codes to reduce the resource-cost of the system.

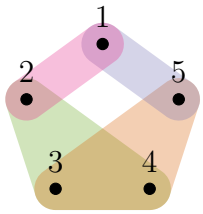
Chapter 6

Irregular Array Codes with Arbitrary Access Sets for Geo-Distributed Storage

This chapter is based on work from [176], done in collaboration with K. V. Rashmi.

Erasure codes are commonly used in distributed storage systems to provide resiliency against failures. In such applications, an $[n, k]$ block code C is used to encode a message $\mathbf{m}$ consisting of k symbols into a codeword $\mathbf{c}$ consisting of n symbols, where symbols are taken from a finite field $\mathbb{F}_q$ of size q. In a *scalar* code, each codeword symbol is then placed in a different node in the system. *Maximum distance separable* (MDS) codes are widely used in practice, because they require the least amount of storage for a given level of failure tolerance.

Scalar MDS codes have the property that the message $\mathbf{m}$ can be decoded from *any* subset of k out of all n nodes. Some applications, however, require the ability to decode the message from *only a few* of those subsets. In some cases, it may even be desirable to decode the message from certain subsets of size smaller than k. An example of such an application is *geo-distributed storage systems* [6, 177–180]. In these systems, data is encoded and distributed across different servers around the globe. Clients in diverse geographical locations then decode and update the data by communicating with these servers. One important constraint is given by the



(a)

node 1	a	b
node 2	c	
node 3	a	
node 4	b	
node 5	c	

(b)

Figure 6.1: (a) Example of arbitrary access sets over five nodes and (b) an irregular array code that satisfies them.

maximum latency facing clients when decoding data (i.e. read latency). Due to the geo-distributed nature of this application, network latency across pairs of clients and servers varies significantly. Thus each client is only able to communicate with a small subset of nearby servers under a given read latency threshold. On top of this, one wants to provide certain failure tolerance guarantees, such as ensuring that each client can decode the data even if one of the nearby servers fails. This leads to the requirement that clients must be able to decode data from specific subsets of nodes.

We formalize such constraints on decodability through the notion of access sets. An *access set* is a subset of nodes $S \subseteq \{1, \ldots, n\}$ expressing the requirement that the message $\mathbf{m}$ must be decodable using only symbols stored in the nodes in S. We say that a collection of access sets $\mathbb{S}$ is satisfied by a code if the message $\mathbf{m}$ is decodable from each of the access sets in $\mathbb{S}$. Note that there can be other subsets not in $\mathbb{S}$ which are also sufficient for decoding $\mathbf{m}$. Our goal is to design a code satisfying the given access sets while minimizing other cost metrics.

Existing work on codes with arbitrary access sets [181, 182] has focused on minimizing *storage overhead*, i.e. the ratio between the total number of symbols in a codeword and the number of symbols in the message it encodes. In this chapter, we focus on an additional metric of practical importance: update cost. The *update cost* of a code is the average number of symbols communicated when a single symbol of the message is updated. In the setting of geo-distributed storage systems, these transmissions would consume the *wide-area network (WAN) bandwidth* which is a

scarce and expensive resource in distributed systems [183]. Thus, update cost is an important metric to minimize in geo-distributed storage systems.

When the access sets correspond to all $\binom{n}{k}$ subsets of size k, it can be shown from basic results about MDS codes that the minimum storage overhead is n/k and the minimum update cost is $(n-k+1)$, both of which are simultaneously achieved by systematic $[n,k]$ Reed-Solomon codes (or other MDS codes).

Towards the goal of optimizing these two metrics in geo-distributed storage systems, we ask the following question: When the required access sets are more relaxed than all subsets of size k, *is it possible to reduce update cost and storage overhead simultaneously by tailoring the code construction to the access sets?* We answer this question in the affirmative.

Consider the following toy example.

Example 6.1: Consider $n = 5$ nodes and access sets $\mathbb{S} = \{\{1,2\},\{1,5\},\{2,3,4\},\{3,4,5\}\}$ (see Figure 6.1a). A code that satisfies $\mathbb{S}$ is a systematic $[5,2]$ Reed-Solomon code, with update cost 4 and storage overhead $5/2 = 2.5$. Another code that satisfies $\mathbb{S}$ is the irregular array code shown on Figure 6.1b, which encodes the message $\mathbf{m} = (a,b,c)$. Note that $\mathbf{m}$ can be fully decoded from any of the access sets in $\mathbb{S}$, and that each message symbol is present in only two different nodes. Thus, this code has update cost 2 and storage overhead $6/3 = 2$. ▶

In Example 6.1, it was possible to reduce storage overhead by placing data unevenly on nodes. This kind of code is called an *irregular array code*, which allows for the storage of a variable number of symbols on each node, in contrast to scalar codes which only store a single symbol per node. Note also that in Example 6.1 the sparsity of the access sets makes it possible to reduce update cost.

We study the problem of designing codes that satisfy the given access sets while minimizing update cost and storage overhead. We start by presenting the problem formally and exploring its fundamental limits (Section 6.2). We do this by first deriving the minimum update cost achievable by an irregular array code satisfying the given access sets (Section 6.2.1). We then focus on analyzing update cost and storage overhead in conjunction, and demonstrate that *employing irregular array*

codes is necessary for jointly minimizing both of these metrics (Section 6.2.2). We also show that, unlike the case where all subsets of size k are access sets, it is not always possible to simultaneously achieve the minimum update cost and minimum storage overhead (Section 6.2.3). Since the cost of WAN bandwidth tends to be higher than the cost of storage [183, 184], we focus on codes with minimum update cost (termed MUC) and minimize storage overhead subject to this constraint. We model MUC codes using information flow graphs, and use this model to derive a lower bound on their storage overhead (Section 6.3). Finally, we show the existence of MUC codes meeting this lower bound through a randomized construction (Section 6.3). Overall, the results show that it is possible to obtain significant savings in update cost and storage overhead compared to traditional MDS codes when one adapts the design of a code to the given access sets. This chapter also exposes a new trade-off space between update cost and storage overhead under arbitrary access sets, which is of significance in geo-distributed storage systems.

6.1 Related work and existing results

In this section, we summarize the related work and review existing results which will be used in this chapter.

6.1.1 Related work

The concept of codes with arbitrary access sets has been previously studied in the information dispersal and secret sharing literatures. *Information dispersal* [181, 182, 185] studies the problem of encoding and distributing a given file f among nodes in a way that satisfies prespecified access sets. While [181, 182] study the minimum storage overhead required by arbitrary access sets, to the best of our knowledge, work in the information dispersal literature has not focused on update cost. *Secret sharing with general access structures* [186] considers the same scenario as information dispersal but adds a security requirement: any subset of nodes that is not an access set leaks no information about f. Security is not among the objectives of this chapter.

Gonen et al. [187] consider collections of access sets which are restricted to be of the same size k and study the field size requirement of scalar codes satisfying them.

Irregular array codes have also been used by [188–190] in a line of work called *irregular MDS array codes*. In this setting, the following parameters are given as input: the number of nodes n, the number of message symbols m_i stored in node $i \in \{1, \ldots, n\}$, and a number k such that *all* message symbols can be decoded from any k nodes. The goal in these works is: 1) to determine the necessary number of parity symbols p_i stored in each node i while minimizing the total number of parity symbols $\sum_{i=1}^{n} p_i$, and 2) to design a code that stores m_i message symbols and p_i parity symbols in node i such that the message can be decoded from any k nodes.

Several works have studied the cost of updates in storage codes via different metrics, such as update complexity, update efficiency, and update bandwidth. *Update complexity* [23, 24, 35, 191–202] is defined as the average number of codeword symbols updated when a single message symbol is updated. In linear codes, this is related to the fraction of non-zero entries (i.e. density) of the generator matrix. *Update efficiency* [203–205] refers to the asymptotic behavior of update complexity. *Update bandwidth* [190] assumes a systematic irregular array code, and is the average amount of symbols communicated among nodes when *all* message symbols stored in a single node are updated. All these metrics differ from the update cost considered in our work, which is defined as the average number of symbols communicated when a single message symbol is updated. Update complexity and update efficiency focus on the number of symbols *updated*, whereas update cost focuses on the number of symbols *communicated* (when nodes store multiple symbols, a single communicated symbol can be used to update several symbols at a node). We focus on the number of symbols communicated rather than updated in order to capture the usage of WAN bandwidth in geo-distributed storage systems. Update bandwidth also focuses on the number of symbols communicated, but is defined for systematic codes only and is motivated by a setting where data is generated *at* the nodes, which is not a good fit for our target application of geo-distributed storage systems. Several other works have studied handling updates in storage systems in different settings, such as multiversion coding [206] and oblivious updates [207].

The design of erasure codes for distributed storage systems has also been studied by other lines of research with the goal of optimizing other metrics. For example, regenerating codes (e.g. [27–30, 35, 37, 40, 43]) minimize the bandwidth cost of node repair, locally repairable codes (e.g. [65, 66, 70, 71, 114]) reduce the number of nodes that must participate in the repair of a single node, and Piggyback codes [47, 63, 208] construct codes to reduce the amount of data read and downloaded during repair while having a low number of symbols per node (i.e., substripes).

6.1.2 Existing results on storage overhead for arbitrary access sets

In this subsection, we summarize results from previous works that are used in this chapter. Naor and Roth [181] proved a lower bound on the minimum storage overhead of a code satisfying given access sets $\mathbb{S}$. Let $[n] = \{1, \ldots, n\}$. Each node $v \in [n]$ is modeled as a variable $w_v \in [0, 1]$ denoting the size of node v as a fraction of the size of the message $\mathbf{m}$. One clear restriction is that the combined size of the nodes in an access set must be at least that of the message. Therefore, a lower bound on storage overhead is given by the solution to the following linear program (LP).

$$\begin{array}{rll} \text{minimize} & \sum_{v\in[n]} w_v & \\ \text{subject to} & \sum_{v\in S} w_v \geq 1, & \forall S \in \mathbb{S} \\ & w_v \in [0,1], & \forall v \in [n] \end{array} \tag{LP1}$$

Lemma 6.1 ([181]). *LP1 gives a lower bound on the storage overhead of irregular array codes satisfying access sets* $\mathbb{S}$.

Proof. Let M be a uniform random variable representing message $\mathbf{m}$, and let $\{W_v\}_{v\in[n]}$ be random variables representing the contents of nodes. A necessary condition for decoding to be possible is that the message is completely determined by the nodes in an access set, i.e., $H(M|\{W_v\}_{v\in S}) = 0$ for all $S \in \mathbb{S}$. Thus, for all $S \in \mathbb{S}$

it holds that:

$$\begin{aligned} H(M) &\leq H(M, \{W_v\}_{v\in S}) \\ &= H(\{W_v\}_{v\in S}) + H(M|\{W_v\}_{v\in S}) \\ &= H(\{W_v\}_{v\in S}) \leq \textstyle\sum_{v\in S} H(W_v). \end{aligned}$$

Moreover, $H(M) = k$, and clearly $H(W_v) \leq \ell_v$ for all $v \in [n]$. This is captured by LP1 when we introduce variables $w_v := {}^{\ell_v}/_{k}$. □

Lemma 6.2 ([182]). *The bound of Lemma 6.1 is achievable by using a sufficiently long MDS code and distributing symbols according to the weights* $\{w_v\}_{v\in[n]}$.

Proof. Let $\{w_v^*\}_{v\in[n]}$ be a rational solution to LP1. The message size k is chosen as the least common denominator of the w_v^* variables, and the number of symbols in node v is set to $\ell_v = kw_v^*$. Let $N = k\sum_{v\in[n]} w_v^*$ be the total number of symbols. Then, one can utilize an $[N, k]$ MDS block code and distribute its symbols according to the node sizes $(\ell_v)_{v\in[n]}$ to construct an irregular array code which has minimum storage size and satisfies the access sets $\mathbb{S}$. □

6.2 Fundamental limits on codes with arbitrary access sets

An irregular array code over finite field $\mathbb{F}_q$ with n nodes, message of length k, and node sizes $(\ell_i)_{i\in[n]}$ is defined by a mapping $C : \mathbb{F}_q^k \to \mathbb{F}_q^N$, where $N = \sum_{i\in[n]} \ell_i$ is the length of a codeword. An irregular array code is said to be linear if C is linear. Each codeword $C(\mathbf{m}) = (c_1, \ldots, c_N)$ is interpreted as an n-node array, with node i denoted as $C_i(\mathbf{m}) = (c_{(\ell_i^<+1)}, \ldots, c_{(\ell_i^<+\ell_i)})$ where $\ell_i^< = \sum_{j=1}^{i-1} \ell_j$. Let the symbol $*$ denote an erasure and, for a subset $S \subseteq [n]$, let $C_S(\mathbf{m})$ denote the result of erasing every symbol in $C_i(\mathbf{m})$ for $i \in [n] \setminus S$. We say that an irregular array code satisfies access sets $\mathbb{S} \subseteq 2^{[n]}$ if $\mathbf{m}$ can be decoded from $C_S(\mathbf{m})$, i.e., there exists a decoding function $D : (\mathbb{F}_q \cup \{*\})^N \to \mathbb{F}_q^k$ such that $D(C_S(\mathbf{m})) = \mathbf{m}$ for all $\mathbf{m} \in \mathbb{F}_q^k$ and $S \in \mathbb{S}$. Note that

if it is possible to decode $\mathbf{m}$ from an access set S, then it is possible to decode $\mathbf{m}$ from any access set $S' \supseteq S$. Thus, we assume without loss of generality that all subsets in $\mathbb{S}$ are minimal, and thus $|\mathbb{S}| \leq \binom{n}{\lfloor n/2 \rfloor}$ (by Sperner's theorem [209]). The *storage overhead* of an irregular array code is defined as the ratio N/k. We define the *update cost* of an irregular array code as the average number of symbols communicated to nodes when a *single* symbol of the message is updated. In general, update cost is at least the average number of nodes that are updated when a single message symbol is updated, since at least one symbol must be communicated to each updated node. In linear codes, both the number of symbols communicated and the number of nodes updated are equivalent because if message symbol m_i is updated to m'_i it suffices to send symbol $\Delta m_i = (m'_i - m_i)$ to every updated node. Each node then scales Δm_i by the appropriate factor and updates its symbols. Thus, for linear codes update cost is:

$$\text{update-cost}(C) := \frac{\sum_{i=1}^{k} |\{j \in [n] : C_j(\mathbf{e}_i) \neq \mathbf{0}\}|}{k},$$

where $\mathbf{e}_i \in \mathbb{F}_q^k$ is the i-th standard basis vector. When $\ell_i = \ell$ for all $i \in [n]$ we say that the code is a regular array code, and when $\ell = 1$ we say that it is a scalar code.

Our ultimate goal is to construct codes that minimize update cost and storage overhead while satisfying the given access sets $\mathbb{S}$. We begin by studying update cost in isolation (Section 6.2.1). Then, we study both update cost and storage overhead in conjunction (Section 6.2.3). Along the way, we show that using irregular array codes is necessary for minimizing these two metrics (Section 6.2.2).

6.2.1 Minimum update cost

Now, we derive a lower bound on the update cost of a code satisfying the given access sets $\mathbb{S}$. To achieve this, we model each node $v \in [n]$ with a binary variable $\dot{w}_v \in \{0, 1\}$ indicating whether the node is updated or not when updating any single arbitrarily chosen message symbol. Observe that if a symbol of the message is updated, then at least one node in each access set $S \in \mathbb{S}$ has to be updated, since otherwise the output of the decoding function on this access set would remain unchanged. Thus,

one can compute a lower bound on the number of nodes updated by a single symbol update and, by extension, a lower bound on update cost, through the following integer program (IP).

$$\begin{array}{rll} \text{minimize} & \sum_{v\in[n]} \dot{w}_v & \\ \text{subject to} & \sum_{v\in S} \dot{w}_v \geq 1, & \forall S \in \mathbb{S} \\ & \dot{w}_v \in \{0,1\}, & \forall v \in [n] \end{array} \tag{IP1}$$

Note that this formulation corresponds to computing a set cover of the access sets by nodes, where a node v is said to cover access set S if $v \in S$.

Lemma 6.3. *IP1 gives a lower bound on the update cost of irregular array codes satisfying access sets* $\mathbb{S}$.

Proof. Let u^* be the optimal value of IP1, and suppose there exists a message symbol that updates fewer nodes when updated. Let U' be the subset of nodes updated. Since $|U'| < u^*$, there must exist at least one access set $S \in \mathbb{S}$ such that $(U' \cap S) = \emptyset$. This is a contradiction, since decoding from S would yield the same output as before the update.

Since the above holds for every symbol in the message, u^* is also a lower bound on the average number of nodes changed by a single symbol update, which is in turn a lower bound on the average number of symbols communicated, i.e., the update cost. □

We show that this bound is achievable via strategic replication.

Lemma 6.4. *The bound of Lemma 6.3 is achievable.*

Proof. Let $\{\dot{w}_v^*\}_{v\in[n]}$ be the optimal solution to IP1. Consider a code that places a full copy of the message $\mathbf{m}$ on each node v where $\dot{w}_v^* = 1$. Clearly, this code achieves the minimum update cost and satisfies the access sets $\mathbb{S}$. □

The next theorem follows from Lemmas 6.3 and 6.4.

Theorem 6.5. *The minimum update cost of an irregular array code satisfying access sets* $\mathbb{S}$ *is given by IP1 and is achieved by strategic replication.*

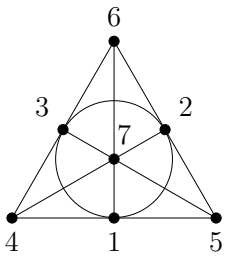
Minimum update sets or μ-sets: IP1 may have multiple optimal solutions for the given access sets $\mathbb{S}$. Each optimal solution can be interpreted as a subset of nodes U where $v \in U$ iff $\dot{w}_v = 1$. We call such subsets of nodes a *minimum update set* or *μ-set*, and denote the collection of all μ-sets for the given access sets as $\mathcal{U}$. Note that a code can have minimum update cost only if every update to a message symbol updates a number of nodes equal to the minimum update cost. Because of this, μ-sets are important for studying codes with minimum update cost, and each message symbol must be associated to a specific μ-set that is updated when that message symbol is updated. As a consequence, in codes with minimum update cost, a node v depends on a certain message symbol iff it belongs to its corresponding μ-set. For example, in Example 6.1, one can verify that the minimum update cost is 2, and that each message symbol updates a μ-set: a updates $\{1, 3\}$, b updates $\{1, 4\}$, and c updates $\{2, 5\}$. Note that the size of $\mathcal{U}$ can be exponential in n and, like $\mathbb{S}$, it is upper bounded by $\binom{n}{\lfloor n/2 \rfloor}$.

6.2.2 The necessity of irregular array codes

In this subsection, we show that considering irregular array codes (instead of traditional scalar codes) is not only important for the sake of generality, but also a necessity for reducing both update cost and storage overhead.

Lemma 6.6. *Irregular array codes are necessary for achieving the minimum storage overhead of arbitrary access sets.*

Proof. The proof proceeds by contradiction using an example. Consider Example 6.1. Notice that any coding scheme that places the same number of symbols in each node must place at least $k/2$ symbols on each node, due to decoding set $\{1, 2\}$ and Lemma 6.1. This results in an storage overhead of at least 2.5. On the other hand, the code proposed in Example 6.1 achieves the minimum storage overhead, which is 2 by Lemma 6.1 (consider the solution $w_1 = 2/3$ and $w_v = 1/3$ for $v \in \{2, \ldots, 5\}$). This means that storing a different amount of symbols in each node is necessary for minimizing storage overhead. □



(a)

node 1	a	b
node 2	c	d
node 3	e	f
node 4	$a+e$	$d+e$
node 5	$a+f$	$c+f$
node 6	$b+e$	$c+e$
node 7	$b+f$	$d+f$

(b)

Figure 6.2: (a) Access sets over seven nodes defined by the Fano plane and (b) an array code that satisfies it.

In general, irregular array codes tend to achieve better storage overhead than scalar codes when the access sets are of different sizes, and when nodes belong to different number of access sets. These two situations arise naturally in geo-distributed storage systems because of the difference in density of servers in distinct regions. Note that existing codes for arbitrary access sets with reduced storage overhead compared to MDS codes (from the literature on information dispersal [181, 182, 185] and secret sharing [186]) are indeed irregular array codes.

Lemma 6.7. *Even for access sets where scalar codes can achieve the minimum storage overhead, array codes are necessary for additionally minimizing the update cost.*

We use the next example in the proof to this lemma.

Example 6.2: Consider $n = 7$ and the access sets $\mathbb{S}$ defined by the Fano plane, where each subset of three nodes is in $\mathbb{S}$ iff they lie on the same line (see Figure 6.2a). The minimum storage overhead for $\mathbb{S}$ is $7/3$, by Lemma 6.1. The minimum update cost is 3, by Theorem 6.5 (every line is a μ-set). The access sets $\mathbb{S}$ are satisfied by a systematic $[7, 3]$ Reed-Solomon code, which has the minimum storage overhead and its update cost is 5 (higher than the minimum update cost). The access sets $\mathbb{S}$ are also satisfied by the irregular array code shown in Figure 6.2b, which encodes the message $\mathbf{m} = (a, b, c, d, e, f)$, and has the minimum storage overhead and the minimum update cost. ▸

Proof of Lemma 6.7. The proof proceeds by contradiction using an example. Consider Example 6.2. For these access sets, no code which places a single symbol per node and has minimum storage overhead can achieve the minimum update cost, as explained below. Clearly, the μ-sets associated with the message symbols must cover all nodes, as otherwise uncovered nodes would never be updated. Thus, the code must have $k = 3$, since at least three μ-sets are needed to cover every node, and there are exactly three nodes in each access set. However, for these access sets, any triple of μ-sets that covers all nodes must intersect at exactly one node. No code that places a single symbol in each node can satisfy the access sets in such a triple, since two of the nodes in it would be a function of the same message symbol, and the remaining node would be a function of all three message symbols. □

6.2.3 Tradeoff of update cost vs. storage overhead

So far, we looked at update cost and storage overhead separately, and saw how to construct codes that achieve the minimum cost possible on each metric separately. However, it is easy to see that while the constructions in Lemmas 6.1 and 6.4 achieve the minimum cost with respect to one metric, they do not perform well with respect to the other. Therefore, it is a natural question to ask whether it is possible to construct codes that minimize both of these metrics at the same time. For some collections of access sets, it is possible to simultaneously achieve both the minimum update cost and minimum storage overhead. For instance, the collections of access sets discussed in Examples 6.1 and 6.2 both have this property. As another example, the access sets consisting of all size k subsets of $[n]$ are satisfied by a systematic $[n, k]$ MDS code, which achieves the minimum update cost $(n - k + 1)$ and minimum storage overhead (n/k). However, as the next example shows, this is impossible for some collections of access sets and there is a tradeoff between update cost and storage overhead.

Example 6.3: Consider $n = 5$ and access sets $\mathbb{S} = \{\{i, j\} : i \neq j \in [5]\} \setminus \{\{4, 5\}\}$ (see Figure 6.3). From Lemma 6.1, it follows that the minimum storage overhead for $\mathbb{S}$ is $5/2$. Here, the only μ-set is $U = \{1, 2, 3\}$, since any other subset of at most three nodes leaves at least one access set uncovered. Thus the minimum update cost is 3.

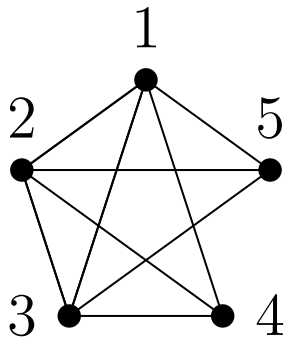


Figure 6.3: Example of access sets for which minimum update cost and storage overhead cannot be simultaneously achieved.

Any MUC code cannot place any symbols in node 4 or 5, since any update to those nodes would require updating more nodes than the minimum. Since $\{1, 4\}$, $\{2, 4\}$, and $\{3, 4\}$ are access sets and 4 is empty, 1, 2, and 3 each must have at least a full copy of message $\mathbf{m}$. This requires storage overhead of at least 3, which is higher than the minimum. ▸

Since, in general, it is impossible to achieve the minimum cost of both metrics simultaneously, and due to the premium in WAN bandwidth cost over storage cost, we focus on codes with minimum update cost and then minimize the storage overhead subject to that constraint. This approach attains one of the Pareto-optimal points in the tradeoff.

Definition 6.1 (MUC code)**:** An irregular array code satisfying access sets $\mathbb{S}$ is said to be a minimum update cost (MUC) code if it achieves the minimum update cost (Theorem 6.5) corresponding to $\mathbb{S}$. ▸

6.3 Storage overhead of MUC codes: lower bound and achievability

In this section, we focus on deriving a lower bound on the storage overhead of MUC codes. In order to derive a lower bound on storage overhead, we model the decoding

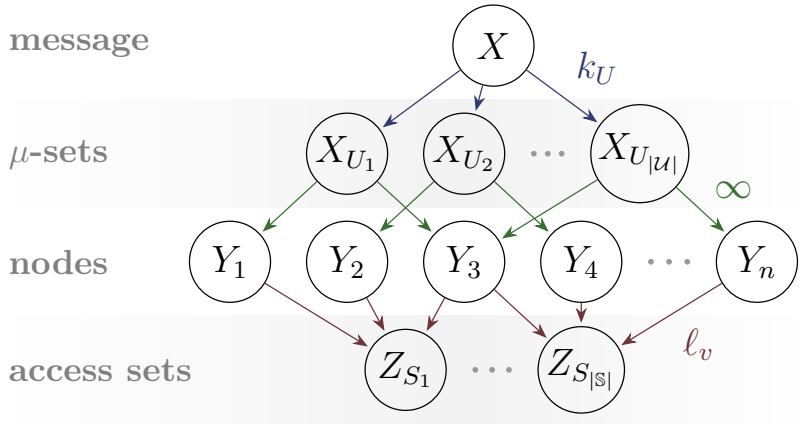


Figure 6.4: Information flow graph for given access sets.

process as a network information flow graph [109].

Recall from Section 6.2.1 that in MUC codes every message symbol is associated to a μ-set $U \in \mathcal{U}$ that is updated whenever that message symbol is updated. For given access sets $\mathbb{S}$, we build an information flow graph with the following vertices:

- X, for the message $\mathbf{m}$;
- $\{X_U\}_{U\in\mathcal{U}}$, for the fraction of $\mathbf{m}$ encoded in μ-set U;
- $\{Y_v\}_{v\in[n]}$, for the contents of node v;
- $\{Z_S\}_{S\in\mathbb{S}}$, for the decoding of access set S.

The graph also contains the following directed edges:

- $\{(X, X_U)\}_{U\in\mathcal{U}}$, where (X, X_U) has capacity k_U;
- $\{(X_U, Y_v) : v \in U\}_{U\in\mathcal{U}}$, each with unlimited capacity;
- $\{(Y_v, Z_S) : v \in S\}_{S\in\mathbb{S}}$, where (Y_v, Z_S) has capacity ℓ_v.

A necessary condition for a MUC code with parameters n, $k = \sum_{U\in\mathcal{U}} k_U$, and $(\ell_i)_{i\in[n]}$ to exist is that in its information flow graph the maximum flow from the source to each sink is at least k. Therefore, the values of $\{k_U\}_{U\in\mathcal{U}}$ and $\{\ell_v\}_{v\in[n]}$ must

be set in order to allow the flows while minimizing the ratio $\sum_{v\in[n]} \ell_v / \sum_{U\in\mathcal{U}} k_U$, which corresponds to the storage overhead.

In order to model the information flow graph, we introduce the following variables:

- $x_U \in [0,1]$ for $U \in \mathcal{U}$ representing the fraction of message $\mathbf{m}$ associated with μ-set U, i.e., $x_U := k_U/k$;
- $y_v \in [0,1]$ for $v \in [n]$ representing the size of node v as a fraction of the size of the message $\mathbf{m}$, i.e., $y_v := \ell_v/k$;
- $z_{U,v,S} \in [0,1]$ representing the flow from μ-set U through node v when access set S is used as the sink.

The following LP captures the information flow graph:

$$
\begin{array}{ll}
\text{minimize} \quad \sum_{v\in[n]} y_v & \\
\text{subject to:} & \\
\sum_{U\in\mathcal{U}} x_U = 1 & \\
z_{U,v,S} \le \mathbb{1}\{v \in (U \cap S)\}, & \forall U \in \mathcal{U},\ v \in [n],\ S \in \mathbb{S} \\
x_U = \sum_{v\in[n]} z_{U,v,S}, & \forall U \in \mathcal{U},\ S \in \mathbb{S} \\
\sum_{U\in\mathcal{U}} z_{U,v,S} \le y_v, & \forall v \in [n],\ S \in \mathbb{S} \\
x_U \in [0,1], & \forall U \in \mathcal{U} \\
y_v \in [0,1], & \forall v \in [n] \\
z_{U,v,S} \in [0,1], & \forall U \in \mathcal{U},\ v \in [n],\ S \in \mathbb{S}
\end{array}
\tag{LP2}
$$

where $\mathbb{1}\{\cdot\}$ is equal to 1 if the condition inside the braces is true, and 0 otherwise.

Theorem 6.8. *For a MUC code satisfying access sets $\mathbb{S}$, LP2 gives a lower bound on the storage overhead.*

Proof. Let f_S be the flow of size k from source X to sink Z_S, where $f_S(u,v)$ denotes the flow from vertex u to v. Clearly, because $k = \sum_{U\in\mathcal{U}} k_U$, it must be the case that $f_S(X, X_U) = k_U$ for all $U \in \mathcal{U}$. Similarly, due to the conservation of flow on the X_U vertices, it must hold that $\sum_{v\in S} f_S(X_U, Y_v) = k_U$. Finally, due to the conservation

of flow on the Y_v vertices, it must hold that $\sum_{U \in \mathcal{U}} f_S(X_U, Y_v) = f_S(Y_v, Z_S) \leq \ell_v$. By defining $z_{U,v,S} := {}^{f_S(X_U, Y_v)}/_k$ and substituting these inequalities with the relevant variables, we obtain the constraints of LP2. A storage overhead lower bound can thus be obtained by specifying storage overhead as the minimization objective. □

Now, we show that MUC codes achieving the lower bound of Theorem 6.8 exist over finite fields of size $q > |\mathbb{S}|$. The key idea is to construct a generator matrix with carefully chosen entries set to zero in order to ensure that the code has minimum update cost, and random entries elsewhere. When the field size is large enough, the probability that the constructed code satisfies $\mathbb{S}$ is strictly greater than zero, thus showing that such codes exist.

Theorem 6.9. *MUC codes over $\mathbb{F}_q$ satisfying given access sets $\mathbb{S}$ with storage overhead matching the lower bound of Theorem 6.8 exist for $q > |\mathbb{S}|$.*

Proof. Consider an optimal rational solution to LP2:

$$\{x^*_U\}_{U \in \mathcal{U}},\ \{y^*_v\}_{v \in [n]},\ \{z^*_{U,v,S}\}_{U \in \mathcal{U},\, v \in [n],\, S \in \mathbb{S}}.$$

The message size k is chosen as the least common denominator of the solution values. The node sizes are set to $\ell_v = ky^*_v$ for $v \in [n]$, and thus $N = \sum_{v \in [n]} ky^*_v$. We specify the code by constructing a generator matrix $\mathbf{G} \in \mathbb{F}_q^{k \times N}$. Let $\mathcal{U}^* = \{U \in \mathcal{U} : x^*_U \neq 0\}$ and $V^* = \{v \in [n] : y^*_v \neq 0\}$. We index the rows of $\mathbf{G}$ with pairs (U, i) where $U \in \mathcal{U}^*$ and $i \in [kx^*_U]$, and the columns with pairs (v, j) where $v \in V^*$ and $j \in [ky^*_v]$. The entries of $\mathbf{G}$ are set as follows: for every $U \in \mathcal{U}^*$ and $v \in V^*$, the block of entries $\{((U, i), (v, j)) : i \in [kx^*_U],\ j \in [ky^*_v]\}$ is set to independently drawn elements of $\mathbb{F}_q$ if $v \in U$, and 0 otherwise. This ensures that when a message symbol associated to U is modified, only symbols in nodes $v \in U$ need to be updated, and thus the code has minimum access cost.

Let $\mathbf{G}_S$ $(S \in \mathbb{S})$ be the submatrix of $\mathbf{G}$ obtained by selecting the columns of $\mathbf{G}$ indexed by $\{(v, j) : v \in (S \cap V^*),\ j \in [ky^*_v]\}$. The message $\mathbf{m}$ can be decoded from S if and only if $\mathbf{G}_S$ contains a $k \times k$ invertible submatrix $\mathbf{G}'_S$. In other words, the

determinant of $\mathbf{G}'_S$ must be nonzero. The Leibniz formula for the determinant states that:

$$\det(\mathbf{G}'_S) = \sum_{\sigma \in \text{perm}(k)} \text{sgn}(\sigma) \prod_{i=1}^{M} (\mathbf{G}'_S)_{i,\sigma(i)}, \tag{6.1}$$

where $\text{perm}(k)$ is the group of permutations over k elements and $\text{sgn}(\sigma)$ is the sign of permutation σ. Given this formula, and the fact that all nonzero entries in $\mathbf{G}'_S$ are chosen independently at random, the determinant in Equation (6.1) will be either a uniformly random element of $\mathbb{F}_q$, or trivially zero (that is, every term in the summation will be always equal to zero). If the determinant is not trivially zero, then it is equal to zero with probability q^{-1}. By taking the union bound over all access sets, it holds that the probability that at least one of the matrices $\mathbf{G}_S$ is not invertible is at most $|\mathbb{S}|q^{-1}$. Thus, as long as $q > |\mathbb{S}|$, there exists a satisfactory generator matrix $\mathbf{G}$.

It only remains to show that for each $\mathbf{G}_S$ there exists $\mathbf{G}'_S$ such that $\det(\mathbf{G}'_S)$ is not trivially zero. This is equivalent to showing that for all $S \in \mathbb{S}$ there exists a sequence of k distinct column indices $(j_1, j_2, \ldots, j_k)$ such that $(\mathbf{G}_S)_{i,j_i}$ is random for all $i \in [k]$. To construct this sequence, the $z^*_{U,v,S}$ values can be used. Consider the set of rows corresponding to update set U. By LP2, it must hold that:

$$kx^*_U = \sum_{v \in S} kz^*_{U,v,S}.$$

Thus, for each U, we pick kx^*_U distinct columns by picking $kz^*_{U,v,S}$ columns from each node v that have not been picked already. Note that LP2 ensures that there always are enough columns to pick from each node via the following constraint:

$$\sum_{U \in \mathcal{U}} kz^*_{U,v,S} \leq ky^*_v.$$

Since $z^*_{U,v,S} > 0$ only if $v \in (S \cap U)$, the entries corresponding to the chosen sequence are guaranteed to be random elements of $\mathbb{F}_q$. □

6.4 Conclusion

Geo-distributed storage systems operate in highly heterogeneous environments, yet most existing systems use erasure codes designed with homogeneity in mind, which results in high costs. This chapter shows that it is possible to use density-aware coding to automatically tailor the design of the erasure code to the density of datasites and, as a consequence, significantly reduce operating costs.

Chapter 7

Density-aware redundancy for efficient geo-distributed storage

This chapter is based on unpublished work (at the time of writing), done in collaboration with Muhammed Uluyol, Mosharaf Chowdhury, Harsha V. Madhyastha, and K. V. Rashmi.

To ensure fault-tolerance and to serve users with low latency, web services employ geo-distributed storage systems which redundantly store data across datacenters in different geographical locations [6, 210, 211]. Requests from users are directed to designated nearby servers called *frontends*, which in turn interact with processes and data located in other servers (called *datasites*) across different regions (see Figure 7.1). Consensus protocols, such as Paxos or its variants, are used to guarantee strong consistency among the datasites [6, 212–220].

In such a setting, it is critical to optimize resource consumption. In particular, storage and wide-area network (WAN) bandwidth are two costly resources that add to the high capital and operating expenses of these systems. Therefore, though geo-distributed storage systems have traditionally replicated every data item in its entirety across one or more datasites, due to cost considerations, erasure coding has been gaining popularity more recently [5, 6, 219, 221]. For example, with Reed–Solomon (RS) codes [20], every object is split into k data chunks, encoded into n code chunks, and distributed across n datasites. Thanks to the so-called *MDS property* that RS

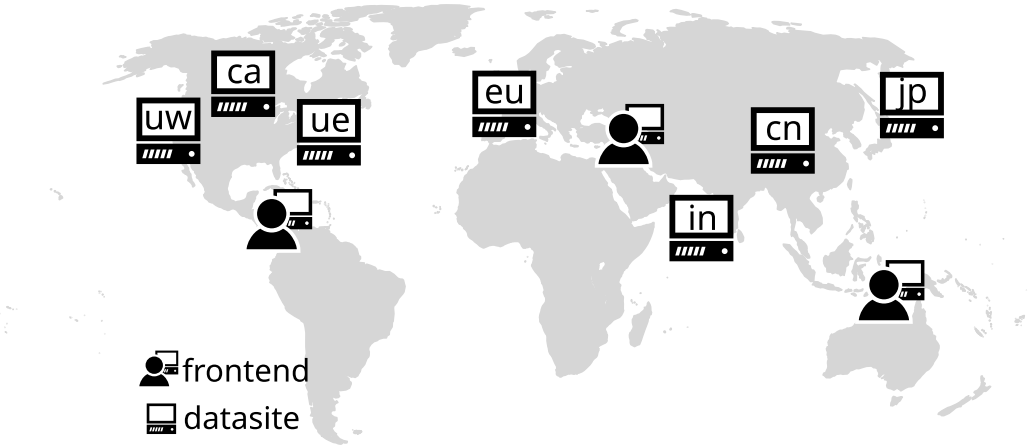


Figure 7.1: Frontends, which are responsible for serving user requests, interact with datasites across different regions.

codes offer, any k out of n code chunks suffice to recover the original object. The higher the k, the lower the storage overhead (for fixed $(n-k)$), and the lower the WAN bandwidth consumption. Furthermore, recent work [6] has proposed the use of complex optimization-based techniques to reduce storage overhead by picking the appropriate RS code parameters given fault-tolerance and latency constraints.

However, existing approaches to erasure coding in geo-distributed storage systems [5, 6, 219] use codes in a similar way as they would be used within data centers. We observe that this makes them severely restrictive and leaves significant headroom. Specifically, in a global deployment, network latencies between frontends and different datasites can vary significantly [222]. For example, for the set of servers we use in our evaluation, we observe that the median latency between servers can vary by more than 300ms or 40 times (Section 7.4). This leads to two deficiencies in existing approaches.

First, the heterogeneity does not mesh well with the homogeneity of RS codes (any frontend needing to read an object needs to connect to k datasites and any k datasites are sufficient). A particular choice of the parameter k might be too high for sparse regions: users connecting to frontends which have fewer datasites nearby will incur high read/write latency due to the need to connect to k data sites that are

far apart (e.g., users in India or Australia in Figure 7.1). Yet, the same parameter k can be too small for dense areas: the system could have reduced storage overhead and WAN bandwidth consumption by choosing a higher value for k (e.g., users in the North America region in Figure 7.1). We note that this problem cannot be solved by just tuning the parameter k since an object can have accesses from both dense and sparse regions. For example, consider the frontends and datasites as shown in Figure 7.1. Suppose the objective is to store the object such that frontends can access it by communicating with nearby datasites (i.e. a read quorum); the frontend should be able to decode the object even if one of the datasites is unavailable. Figure 7.2 shows how different approaches would encode and distribute the object when using read quorums with similar latencies. Notice how unevenly distributing data across datasites can result in lower storage overhead (under the solution title PUDU). That is, having datasites in dense regions store less data reduces the amount of damage that any one failure can cause, and thus permits lower storage overhead. Moreover, notice that code symbols can be designed to be a function of fewer object symbols (compared to RS codes); we refer to them as *sparse parities*. Having sparse parities turns out to be essential for reducing WAN bandwidth, because it reduces the amount of data that must be communicated when the object is modified.

Second, given that it is advantageous to adapt the erasure code to the access pattern (as discussed above), encoding needs to be different for objects with different access patterns. For example, in Figure 7.1, an object that is accessed only from North America will see drastically different distribution of latency to datasites as compared to an object which is accessed both from North America and India; thus it would need a code with entirely different data layout and parity functions. This also means that replication can be beneficial over erasure coding for certain access patterns.

In this chapter, we make a case for tailoring the redundancy scheme—*not just the tuning the parameters but using entirely new parity functions and data layout*—to the geographical distribution of the datasites and the access pattern profiles. We show that one can leverage the spatial heterogeneity to significantly optimize resource consumption and costs. We refer to this new approach to designing redundancy

in geo-distributed storage systems as *density-aware redundancy*. We demonstrate the utility of density-aware redundancy by showcasing significant reduction in WAN bandwidth usage, while continuing to meet latency SLOs and being efficient in storage overhead.

There are several challenges to realizing density-aware redundancy. First, handcrafting redundancy schemes for various access pattern profiles is unrealistic. Codes that are employed in current storage systems are all handcrafted by theoreticians using combinatorics and algebra. However, density-aware redundancy requires designing entirely new parity functions and data layout for new access profiles, and cannot rely on handcrafted designs. Second, having to prove correctness of consensus for every new design of parity functions and data layout is a showstopper. The design of the redundancy scheme (parity functions and the data layout) and the consensus protocol (read and write quorums) interact in complex ways. Existing works that use erasure coding with consensus protocols are all solely based on RS codes, and hence, their correctness has been investigated only for RS codes. Third, to leverage the benefits of sparse parities in reducing WAN bandwidth, we must design a consensus protocol with explicit support for operations that only modify part of an object. Existing approaches to using erasure coding along with consensus protocols are all for RS codes which have homogeneous parity functions and thus are not applicable when some parity functions are dense and some are sparse. Specifically, the log management used by consensus protocols needs to be redesigned.

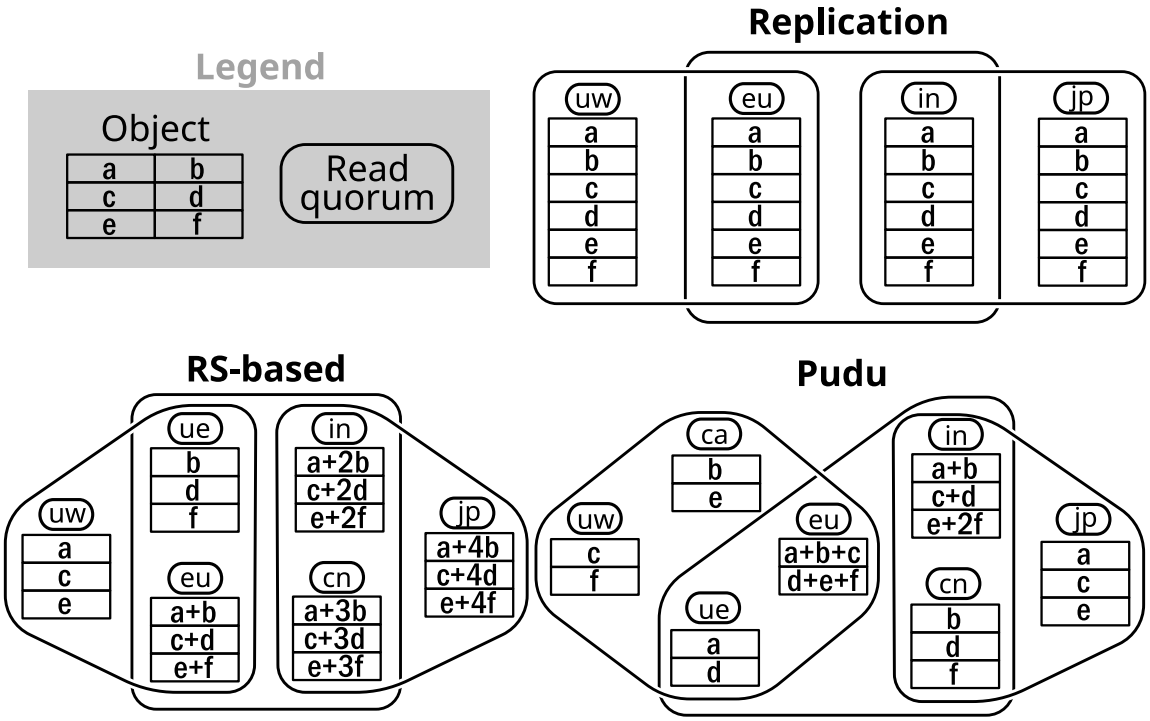
We present Pudu, a geo-distributed storage system that is more resource efficient than the state-of-the-art by employing density-aware redundancy and overcoming the above-mentioned challenges. Pudu leverages an optimization-based theoretical framework for designing redundancy schemes [176]. Since the outputs of that framework are not guaranteed to be practical (as explained later), Pudu overcomes challenges in arriving at practical redundancy schemes. Moreover, the design of Pudu is such that this optimization can be performed jointly with optimizations performed for designing the consensus protocol (i.e., read and write quorums), while keeping the computations tractable; we refer to this module as "`ECOptimizer`". This overcomes the first challenge by automating the process of designing redundancy schemes and

thus enabling tailoring to different access profiles.

Pudu introduces a new abstraction between the optimization for consensus design and the optimization for redundancy design. This abstraction enables Pudu to guarantee correctness of consensus for any redundancy scheme (i.e., any parity functions and data layout) output by `ECOptimizer`. This overcomes the second challenge by negating the need to prove the correctness of consensus individually for different access profiles. Finally, Pudu overcomes the third challenge by a new design of log management at datasites that efficiently reconciles the update operations at datasites both with dense and sparse parities.

In our extensive evaluation, we see that Pudu significantly improves over the state-of-the-art. We evaluate Pudu by running it over a wide variety of application requirements, and using several different optimization goals. In 42.37% of the inputs we test, Pudu improves over the state-of-the-art in *all* of the dimensions we optimize for. In the cases where Pudu achieves an improvement, on average, Pudu reduces ready latency by 6.47ms (up to 28ms or 17.9%), write latency by 6.77ms (up to 36ms or 21.7%), storage overhead by 0.14 (up to 1 or 40%), and update cost by 35.4% (up to 72.7%).

This chapter makes four contributions. First, we identify that heterogeneity in latencies in a global deployment presents the need for designing entirely new parity functions and data layout for every access profiles (density-aware redundancy). Second, we design Pudu that realizes density-aware redundancy. Pudu co-optimizes the consensus protocol and the redundancy scheme design while keeping the computations tractable and ensuring that the erasure code outputs are practical, uses a new abstraction layer to ensure correctness of consensus for any design of the parity functions and data layout, and uses a sophisticated log management to ensure correct updates in both dense and sparse parities. Third, we implement Pudu using a combination of Go, Python, and C. Fourth, we perform extensive evaluation showing that Pudu is able to unlock better resource tradeoffs and achieve significant reduction in WAN bandwidth consumption with negligible overheads. Prior works on redundancy in storage systems are restricted to choosing between replication and erasure coding [8, 11, 123], and choosing parameters of a particular code [1, 15, 120].



Method	storage overhead	update cost
Replication	4	4
RS-based	3	5
Pudu (this chapter)	**2.83**	**4**

Figure 7.2: Example of benefits of density-aware redundancy. The rounded boxes represent datasites (labeled as in Figure 7.1) and the boxes below represent the data they store. The object is split into 6 symbols (labeled a–f). The blobs represent read quorums, each corresponding to a different frontend in Figure 7.1. Each approach must satisfy the following fault-tolerance requirement: the object must be recoverable from any read quorum even if 1 arbitrary datasite fails. The bottom table shows the costs of different approaches in this example.

To the best of our knowledge, PUDU is the first storage system design that tailors redundancy by entirely changing the erasure code itself (i.e. the parity functions and the data layout) across objects.

7.1 Geo-distributed storage systems: Opportunity and challenges

We begin by describing the state-of-the-art tools for designing strongly-consistent geo-distributed storage systems. In this process, we highlight some of the limitations of these tools when it comes to accommodating spatial heterogeneity.

7.1.1 Optimizing resources in geo-distributed storage

In order to be robust and useful in practice, geo-distributed storage systems must satisfy several criteria:

- **Consistency:** many applications expect reads and writes to an object to be *linearizable*, i.e., all successful writes can be totally-ordered and every read obtains the most recently successful write.
- **Reliability:** the system must be able to tolerate at least f arbitrary failures, without losing data, consistency, or the ability to make progress.
- **Low latency:** the system must provide low read- and write-latency. These are specified in the form of service-level objectives (SLOs), which have to be respected in most cases, but might be violated in exceptional circumstances.

Geo-distributed storage systems require a lot of resources in order to operate and fulfil the above criteria: two of the most important and costly ones are storage space needed for storing objects and the WAN bandwidth used by the communication between datasites. To make geo-distributed storage systems affordable, it is critical to design the system so as to minimize resource consumption.

Recent work [6] has proposed using optimization to design certain aspects of the system (such as quorums) to minimize storage costs while meeting the given latency SLOs. In many Paxos-based consensus protocols, instead of using majority quorums, read and write quorums can be specifically chosen, with the restriction that every read quorum intersect every write quorum (i.e. Flexible Paxos [220]). In addition to this, some systems use RS codes to encode and distribute objects instead of using replication, because RS codes require lower storage overhead [6, 219].

7.1.2 Reed-Solomon codes and their shortcomings

To encode an object of size B using a k-of-n Reed–Solomon (RS) code, the object is first split into k data chunks of size $\left\lceil \frac{B}{k} \right\rceil$ and then the RS code is used to encode these k data chunks into n code chunks of the same size, which are then distributed to servers. These code chunks have the property that any k of them is enough to decode the original object [223]. The *storage overhead* of a code is the ratio between the total size of the encoded object, and the original size of the object. The storage overhead of a k-of-n RS code is $\frac{n}{k}$ (ignoring the rounding in chunk size). RS codes are widely used in practice because they can tolerate erasures with significantly less storage overhead than replication.

Next, we describe some reasons why RS codes are not well-suited for geo-distributed storage.

Every code chunk is treated the same

In an RS code, every code chunk is of exactly the same size ($\lceil \frac{B}{k} \rceil$), and every subset of $k-1$ code chunks is insufficient to decode the original object. However, geo-distributed systems typically have a lot of spatial heterogeneity: some datasites are in dense areas close to many other datasites, while others are distant and isolated. Treating both types of datasites the same way gives us unfavorable trade-offs between latency and storage-overhead.

For example, consider Figure 7.2. One of the fault-tolerance requirements in this example is that the object is still recoverable from any quorum even after any one

failure. When using RS codes, all datasites store the same amount of data. As a consequence, it may not be convenient to use all of the datasites in dense areas (e.g. "ca" is not used). This is because the parameter k of the RS code (and thus the size of each code chunk) is limited by the sparse regions, where contacting k different datasites is slower. It would be ideal to to put a smaller code chunk on datasites in dense areas. This reduces the amount of data lost with any given datasite failure, and thus makes it possible to achieve the same level of fault-tolerance with lower storage-overhead.

Given the above, it is clear that the only two parameters available to RS codes, n and k, are not enough for adapting to heterogeneous global deployments.

RS codes have uniformly dense parities

A necessary property of RS codes is that they have *dense parities* [223]: this means that every parity is a function of all data chunks. This implies that any update, no matter how small, will affect at least $n - k + 1$ different code chunks (i.e., all parity chunks leaving only other data chunks). As we shall see next, this leads to high WAN bandwidth usage.

7.1.3 Impact of redundancy on WAN bandwidth

The cost of transferring data over the WAN is relatively high compared to other operating costs. The main way in which we can reduce WAN bandwidth usage is by reducing the amount of object data that needs to be transferred when executing different operations. Whenever we perform a full-write of an object, we need to transmit an amount of data that is roughly equal to the storage-overhead (plus some protocol overhead). For this reason, reducing storage-overhead by using RS codes instead of replication, for example, can help reduce WAN bandwidth.

However, in low-latency collaborative applications users normally only update a relatively small part of the object in each operation. In such cases, encoding and transmitting the whole object is wasteful; instead, users can just send the difference with the previous version. That is, if the user changed the object from x to x', then it

can send $\Delta x = (x' - x)$ (which will be mostly zeros except in the small part that was modified). Let $\text{Enc}(x)$ be the encoding of x. Almost all erasure codes used in practice (including RS codes) are linear codes. Linear erasure codes satisfy the property that $\text{Enc}(x) + \text{Enc}(\Delta x) = \text{Enc}(x + \Delta x) = \text{Enc}(x')$. Therefore, datasites can update their local code chunks with just Δx.

When updates are taken into account, RS codes can actually increase WAN bandwidth usage compared to replication, because RS codes require updating more datasites.

Opportunity

Geo-distributed systems are highly heterogenous, due to the natural differences in population-density and geography around the world. Density-aware coding is the perfect tool for exploiting this heterogeneity. Consider the example in Figure 7.2: one of the users finds themselves in a region that is more dense with potential datasites. However, replication and RS codes are not flexible enough to exploit those differences, because they place the same amount of data in each datasite, and thus have to adapt to the regions with fewer datasites. It would be ideal if one could use a larger read quorum in datasite-dense regions along with placing less data on each datasite, and vice-versa in sparse regions. By doing this, density-aware redundancy reduces the amount of storage overhead compared to other alternatives, because it reduces the amount of data that is lost should any one datasite fail.

Another way in which heterogeneity can be exploited is in the design of parities. As explained above, RS codes have uniformly dense parities which result in high WAN bandwidth usage. If some of the parity functions are sparse, i.e., if they are independent of some of the data chunks, those code chunks need not be updated when the corresponding data chunks are updated. We term the number of code chunks that need to be updated when a single data chunk is updated as the update cost of the code. As seen above, the update cost of the code is a function of how dense/sparse the parity functions of the code are. Thus, heterogeneity creates an opportunity for reducing update cost by making parities sparser. E.g., in Figure 7.2

the parities (under PUDU) are designed to be as sparse as possible, yet still provide fault-tolerance. Because of the reasons above, using density-aware coding to adapt to the heterogeneity in geo-distributed systems is a promising opportunity for reducing cost.

Challenges

While density-aware redundancy is appealing due to its potential in reducing consumption of costly resources, it comes with several challenges.

Automating code design. Erasure codes currently used in practice are handcrafted by theoreticians, and thus are specifically designed for a narrow setting, or are very general and thus do not take advantage of the structure of the setting. Since we cannot design an erasure code for each specific case, we must find a way to automatically generate new erasure codes which satisfy the necessary requirements, minimize cost, and can be used in practice.

Enabling consensus on arbitrary codes. Quorum-based consensus protocols are well-suited to a geo-distributed setting, where each frontend accesses data from nearby datasites. However, existing approaches across the WAN [6, 215, 224, 225] use quorums that are identical in structure across frontends. E.g., when they are of fixed size, all frontends use read quorums containing the same number of sites.

As we discuss in Section 7.1.3, recovering data from a variable number of data sites can yield benefits. To realize these benefits, we need two things. First, a consensus protocol that is agnostic to the underlying code, unlike the existing which rely on redundancy scheme being either replication or the k-of-n RS codes. Second, we need a mechanism to define quorums that can *vary in size* across frontends while maintaining consistency and durability.

Complexity in co-optimizing. Optimizing the consensus protocol configuration or the erasure code design are challenging tasks by themselves, but optimizing both simultaneously proves to be even more complicated. Capturing all the variables, requirements, and objectives into a single integer problem is conceptually and computationally intractable.

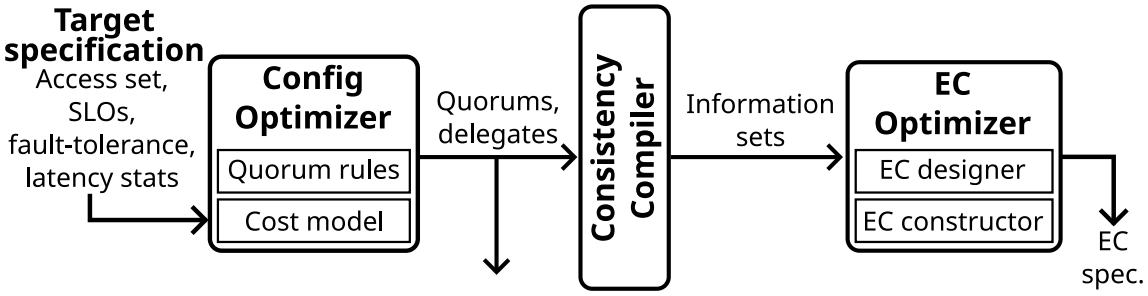


Figure 7.3: High-level architecture of PUDU's optimizer.

Generating practical erasure codes. The optimization framework that can be employed to automate the design of the code [176], is primarily theoretical in nature, and the construction proposed therein are not guaranteed to be practical. In particular, the theoretical construction might require dividing the object into an unbounded number of parts, or it might require using a finite field of size larger than the standard GF(2^8). Another limitation is that prior work can *only* produce codes which have the exact minimum update-cost.

Handling incremental updates efficiently. Typical write operations do not modify the whole object. Thus, to reduce WAN bandwidth we must add explicit support for operations that modify part of an object, and implement in a WAN-efficient manner. To guarantee correctness, we must give special attention to the way these operations are managed in the Paxos log.

7.2 Pudu design

PUDU is a geo-distributed key-value store that employs *density-aware redundancy*, designed to overcome the challenges in Section 7.1.3. The two main components of PUDU are (1) an optimizer, and (2) a geo-distributed key-value store. In this section, we present a high-level overview of the design (Section 7.2.1) and how read/write/update operations work in PUDU (Section 7.2.2). A more detailed description of the components and density-aware redundancy will be provided in Section 7.3.

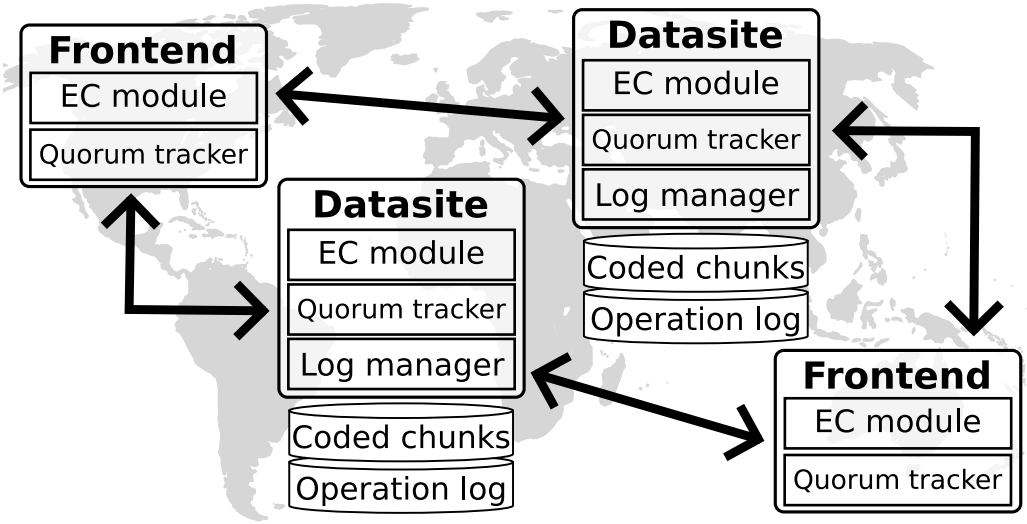


Figure 7.4: High-level architecture of PUDU's distributed storage system.

7.2.1 Overview

The optimizer (shown in Figure 7.3) is run offline to generate the configuration for consensus protocol (that is, the read and write quorums) and the redundancy scheme (parity functions including replication and the data layout) for a given access profile, latency SLOs, and fault tolerance targets. The geo-distributed key-value store (shown in Figure 7.4) uses the generated configuration and erasure code to execute the consensus protocol and operate on objects. In the following, we give a high-level overview of these two parts and their interaction.

Pudu's optimizer

Recall that density-aware redundancy requires designing an entirely new redundancy schemes (parity functions and data layout) for different access profiles. Hand designing access-pattern-specific redundancy schemes is unrealistic. PUDU overcomes this challenge by employing an optimization-based framework for designing codes [176] to automate the task of designing access-pattern-specific redundancy schemes. A key novelty behind PUDU's optimizer design is that it is capable of *co-optimizing* both

the consensus protocol configuration and the design of the erasure code with the goal of reducing operating costs while meeting latency targets.

The input to the optimizer consists of: (1) the network topology: the location of potential datasites, and statistics on the latencies between any pair of sites; (2) the application requirements: the number of tolerable failures f, the application SLOs (such as the maximum acceptable read/write latencies or the maximum storage-overhead), and the optimization goals (along with their priorities); and (3) an *access set*: defined as the subset of sites that will serve as frontends (applications determine the access set based on the geographic locations of the users that operate on the object). We refer to the combination of these inputs as a *target-specification*. The optimizer considers the tradeoffs between four dimensions of interest: read latency, write latency, storage-overhead, and update-cost. Given a target-specification, PUDU minimizes the specified dimensions subject to the specified restrictions. We call the dimensions that are minimized as `Optimized Dimensions`. For example, the target-specification may include SLOs for the read and write latency, and may designate update-cost and storage-overhead (in that order) as the `Optimized Dimensions`.

The output of the optimizer consists of: (1) the configuration of the consensus protocol: the locations of the chosen datasites, the quorum selection, and other protocol details; and (2) the specification of the redundancy scheme: how objects are split, encoded, and distributed to datasites.

Optimizing the consensus protocol configuration or the redundancy scheme separately are already complex tasks on their own. Optimizing both of them simultaneously proves to be even more complex. In order to reduce the complexity and to keep the optimizer tractable, PUDU's optimizer is divided into three stages (see Figure 7.3): the `ConfigOptimizer`, the `ConsistencyCompiler`, and the `ECOptimizer`.

The **`ConfigOptimizer`** chooses the configuration of the consensus protocol: the most important of which is the quorum selection. This selection is made according to a set of rules (specified in Section 7.3.2). We design this stage to be as flexible as possible so as to not restrict the options for the `ECOptimizer`: each quorum is allowed to be of a different size, and the rules are as general as possible while still guaranteeing the necessary properties. The `ConfigOptimizer` is equipped with a

`Cost Model` which allows it to guarantee that the SLOs will be satisfied, and to predict the final costs of the `Optimized Dimensions`.

The **`ConsistencyCompiler`** is the interface between the `ConfigOptimizer` and the `ECOptimizer`. It takes the quorums produced by the `ConfigOptimizer` and, by applying a set of rules that ensure correctness and consistency, translates it into a set of requirements that must be satisfied by the erasure code. These requirements are expressed as a collection of *information sets*. Each information set is a subset of code chunk indexes, expressing the requirement that a decoder must be able to recover the object when given the code chunks indexed by the information set.

The **`ECOptimizer`** takes the information sets from the `ConsistencyCompiler` an input and outputs an erasure code specification that satisfies the decoding requirement enforced by the information sets. `ECOptimizer` is composed of an *erasure code designer* which determines the structure of the erasure code (that is, the parity functions and the layout of the chunks) in a way that satisfies the information sets and minimizes the update-cost and storage-overhead. We note that replication is a trivial form of erasure code and thus replication of chunks is also a valid output. As discussed in Section 7.1.3, the outputs from the theoretical optimization framework for code design [176] are not guaranteed to be practical. The framework might only provide guarantees on existence of a code. Hence, `ECOptimizer` also comprises of an *erasure code constructor* which is responsible for searching for an explicit, practical representation of the erasure code that can be used efficiently in practice (explained in detail in Section 7.3.1).

This three-stage design ensures that the consensus protocol configuration and the erasure code can be jointly optimized in a way that is easier to analyze, produces good results, and is computationally tractable.

Pudu's key-value store

Pudu implements a geo-distributed key-value store. We distinguish two types of entities: *frontends*, and *datasites* (Figure 7.4). Users interact with frontends, which serve their requests and execute operations by communicating with datasites. Fron-

tends have an EC module and a quorum tracker. The EC module contains metadata about the erasure code and information sets, and is responsible for encoding/decoding objects. The quorum tracker contains all the metadata about quorums, and keeps track of datasites responses to ensure correctness. Datasites manage data and execute the consensus protocol. They store the coded chunks assigned to them and keep the Paxos log of operations. In addition to the EC module and quorum tracker, datasites have a log manager. The log manager is in charge of garbage collecting old entries and filling missing entries that might occur during execution due to lost messages or unavailabilities. It is possible for both a frontend and datasite to be in the same location.

7.2.2 Operations in Pudu

Now, we describe the operations supported by PUDU, and give a brief description of how they are executed.

Writes

A write operation sets the object associated with a key (replacing it if it already exists). Writes occur over two phases: in phase 1, the writer asks datasites to *prepare* and waits for a read quorum to *promise* to accept their write. In phase 2, the writer encodes the object and sends the corresponding code chunks to each datasite; then it waits for a write quorum to *accept* the write. After phase 2 completes successfully, the write is considered committed, and the writer notifies datasites about this. The time it takes to complete these two phases determines the *write latency*.

Reads

A read operation retrieves the latest committed object associated with a key. Reads occur on a single phase: the reader contacts the datasites specified by the quorums, which reply with the state of their log. Once a read quorum replies, the reader uses the data in the logs to reconstruct (i.e. decode) the latest version of the object.

Updates

An update operation modifies an existing object associated with a key. Updates follow the same 2-phase procedure of writes. The only difference is that instead of encoding and sending the object, the writer sends the "delta" with respect to the previous version. That is, if the value of the object is changed from x to x', the writer sends $\Delta x = (x' - x)$. Let $\text{Enc}(x)$ be the encoding of x; linear erasure codes satisfy the property that $\text{Enc}(x) + \text{Enc}(\Delta x) = \text{Enc}(x + \Delta x) = \text{Enc}(x')$. Datasites keep Δx in their log; to apply it, they simply encode it as $\text{Enc}(\Delta x)$ and add it (xor) to their code chunk. This is valid since the output of `ECOptimizer` is always a linear code.

Notice that if the code chunks in a datasite are not a function of the nonzero entries in Δx, then those code chunks will be unaffected by the update. Thus, a datasite only needs Δx if it affects its code chunks. Thus, the writer uses the code metadata to determine which datasites are affected by the update, and sends Δx to those datasites only; to other datasites it only sends the consensus protocol information.

7.3 Density-aware redundancy

In this section, we explain the design and implementation of the different parts of Pudu in greater detail.

7.3.1 Dividing the problem into stages

We start by explaining the rationale behind our three-stage optimization design. There are several challenges that we considered when designing Pudu's optimizer (Section 7.1.3).

The first challenge is guaranteeing the correctness of the system, even when the configuration of the consensus protocol and the erasure code are unknown a priori. To solve this challenge, we identify a very general set of rules to be followed by the optimizer: this set of rules is strict enough to ensure correctness, but also loose enough to leave enough room for optimization. In the erasure code, we enforce these

rules through the abstraction of *information sets*.

A second challenge is in performing optimization in a way that is conceptually and computationally tractable. To solve this challenge, we divide the optimizer into three sequential stages. This makes the procedure easier to understand and analyze, but it also makes the problem easier to solve by dividing it into smaller parts with a sequential dependency.

A third challenge is ensuring that the produced erasure code is practical. This is the disadvantage of our sequential approach: the first stage does not know if its solution will lead to a practical erasure code. To ameliorate this problem, we allow the final stage of the optimizer to output erasure codes with slightly higher costs when an optimal solution is not found or, in the worst case, to default to RS codes.

7.3.2 Flexible quorums for optimization

In Section 7.1, we showed that recovering data from a variable number of datasites can yield benefits due to highly heterogeneous nature of geo-distributed systems. Previous works depend on quorums with some regular structure. Majority (and other fixed-size) quorums are the most common type. Other types of quorums with regular structure exist, such as grid [220] and hierarchical quorums [226]. However, the distribution of sites across the globe does not follow a regular structure, and thus imposing additional requirements on quorums leads to sub-optimal solutions.

For this reason, we use an optimizer to directly construct each quorum based on the network structure, the access set, the SLOs, and the objective we want to achieve. We impose minimal constraints on quorums to allow the optimizer flexibility in accomplishing the objective. As a fallback, we ensure that any subset of all but f datasites contains a read and write quorum; this allows optimizing common scenarios more aggressively without giving up failure-tolerance.

Design of the `ConfigOptimizer`

As input, the `ConfigOptimizer` receives a model of the network (datasites available and latencies between each pair of datasites), the application inputs (access set, failure

tolerance, and latency or storage SLOs), and dimensions to minimize. As output, the `ConfigOptimizer` produces the consensus protocol configuration. In the protocol, we use the *fast-read quorum* and *write delegation* techniques used in prior work [6]. Thus, the output of the `ConfigOptimizer` will include the set of quorums (read, write, and fast-read) and the selection of write delegates for each frontend. We formulate the problem as an integer program, and solve it using CPLEX [227]. For a potential configuration, the integer program models the read and write latencies, as well as the minimum achievable update-cost and storage-overhead. Using this, it searches for the consensus protocol configuration that minimizes the given dimensions subject to the given SLOs.

The program additionally ensures that the choice of quorums satisfies the following correctness requirements (assuming we desire to tolerate at most f failures).

Q1. Every read or write quorum has size greater than f.

Q2. The intersection of any read/fast-read quorum and any write quorum is non-empty.

Q3 (Backup quorums). Every subset of all but f datasites must contain a write quorum and a read quorum.

7.3.3 Design of parity functions via optimization

In this section, we explain how we optimize the design of erasure codes to minimize update-cost and storage-overhead. Our strategy is to adopt an erasure-coding framework that tailors the design of the erasure code to the characteristics of each target-specification. This type of erasure code is called *Minimum-Update-Cost* (MUC) codes [176].

The MUC codes framework

The *update-cost* of an erasure code is defined as the expected number code chunks that change when a randomly-chosen symbol of the data is modified. E.g., the update-cost of t-replication is t and the update-cost of a k-of-n RS code is at least $n - k + 1$

(number of parities plus one data chunk). At a high-level, the MUC codes framework takes the decodability requirements of the code (represented as information sets) as input, and searches for an erasure code that satisfies them and minimizes update-cost and storage-overhead.

The optimization procedure first determines the minimum update-cost achievable for the given information sets. Then, it determines minimum storage-overhead achievable by an erasure code that has minimum update-cost and satisfies the information sets. After that, an erasure code with the calculated costs can be constructed via a randomized construction (although it is not guaranteed to be a practical erasure code). Unlike RS codes, erasure codes produced via this approach can have code chunks of different sizes, and whose contents are arbitrary linear combinations designed in a way that minimizes update-cost.

Design of the `ECOptimizer`

Our `ECOptimizer` takes the information sets produced by the `ConsistencyCompiler` as input, and produces an erasure code that satisfies them as output. One important abstraction used by the `ECOptimizer` is that of an *update-set*. The update-set of a data symbol x corresponds to the subset of datasites whose code chunk depends on x. Thus, if x were to change its value, all the code chunks in the update-set of x would also change their value. Each update-set of a code should intersect every information-set; otherwise, it would be possible to make an update which would not be seen by one of the information-sets [176].

The `ECOptimizer` executes the following steps: (1) determine all the update-sets below a certain size, set as the minimum update-cost plus some slack (solved as a constraint-satisfaction problem); (2) decide how much data to assign to each update-set, and the size of each code chunk based on a theoretical model that casts the problem as a flow-network problem (solved as a linear program); (3) find a generic generator matrix (where non-zero entries are indeterminates) for the erasure code that divides the object into the least number of pieces (solved as an integer program); and (4) construct the code by randomly instantiating the non-zero entries of the

matrix, and checking that the resulting matrix satisfies all of the information sets. In the rare case that the `ECOptimizer` fails or takes too long to find a solution, we default a solution that uses RS codes or replication.

7.3.4 Interaction between density-aware redundancy and consensus

Now, we describe the stage that serves as the interface between the `ConfigOptimizer` and the `ECOptimizer`. To achieve this, we use the information set abstraction as the interface between the consensus protocol and the erasure code. This generalization allows the system to accommodate any arbitrary erasure code, and it simplifies the interaction between consensus and erasure coding.

Prior work [6, 219] shows how to achieve consensus using RS codes. We do not think it is practical to modify consensus protocols and prove their correctness each time a new erasure code is developed, especially since they are prone to having subtle bugs [228]. Moreover, because our approach is to construct a specialized code for each combination of access set and requirements, we require a mechanism for achieving consensus using an *arbitrary erasure code*.

Instead of relying on the erasure code having a particular structure, PUDU's consensus protocol relies on information sets for correctness. This allows the consensus protocol to ignore the inner workings of the erasure code, and to only interact with it through information sets. This is the most general interface that can be defined, as it perfectly captures the needs of the system (to recover the whole object) without imposing any additional onerous constraints on the structure of the erasure code (such as the MDS property).

Design of `ConsistencyCompiler`

The `ConsistencyCompiler` acts as the interface that couples the optimization of the consensus protocol configuration and the optimization of the erasure code. The objective of this stage is to derive the most generic set of decodability requirements

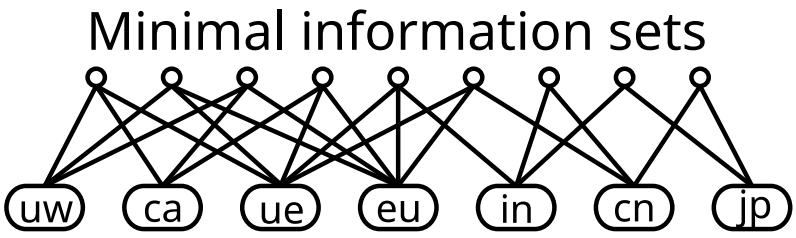


Figure 7.5: Information sets for PUDU configuration in Figure 7.2, depicted as a bipartite graph where the nodes at the top are information sets, and edges represent set inclusion.

that must be fulfilled by an erasure code that is compatible with the configuration calculated by the `ConfigOptimizer`. This is achieved by transforming the chosen quorums into a collection of information sets via the following rules.

R1 (Availability). Removing any f datasites from a read quorum yields an information set.

R2 (Durability). Removing any f datasites from a write quorum yields an information set.

R3 (Readable writes). The intersection between any read quorum and write quorum yields an information set.

R4 (Fast). Every fast-read quorum is an information set.

For example, Figure 7.5 shows the information sets for PUDU in Figure 7.2 (only shows the minimal information sets).

Correctness

Our consensus protocol is a generalization of the protocol presented in [6]. The consensus protocol of [6] crucially depends on the property of RS codes that any k code chunks are sufficient to decode an object. This property is used to ensure several basic properties of the protocol, such as availability, durability, progress, stability, etc. At its core, the generalization we perform is conceptually simple: wherever [6] would require k code chunks we instead require an information set. This guarantees that, as long as the erasure code satisfies the required information sets, the consensus

protocol will have the desired properties.

Theorem 7.1. *When instantiated with an erasure code and configuration produced by its optimizer,* Pudu *provides the same basic corectness guarantees as Paxos.*

Proof. This theorem follows from rules Q1–4, R1–4, and the fact that the produced erasure code will satisfy the generated information sets. Essentially, the proof of correctness presented in [6] remains valid in Pudu when replacing when replacing every requirement of "at least k datasites" with "an information set", because it does not use any other properties of RS codes other than the fact that any k code chunk always define an information set.

Consider the proof of correctness of the consensus protocol in [6, appendix A]. This consensus protocol considers the same correctness guarantees and the same features that we consider in this work. We now describe each of the places in that proof which depend on the MDS property of the underlying RS code, and describe why that requirement can be changed to an information set, without loss of generality.

- "The intersection of any fast-read (phase 1a) and write (phase 2) quorums contains at least one datasite": this does not depend on the erasure code, but is directly enforced by rule Q2.
- "The intersection of any read (phase 1b) and write (phase 2) quorums contains at least k datasites": this constraint is captured by rule Q2 and rule R3, which requires this intersection to just be an information set instead. The only reason this property is required is to ensure that a reader can always decode the most recently committed value of the object upon reading from a read quorum, and thus correctness is preserved.
- "A fast-read (phase 1a) quorum must contain at least k datasites": this is replaced by rule R4, which specifies that the fast-read quorum must be an information set instead. This preserves the correctness, because readers are still able to decode the object from a fast-read quorum when there are no failures.
- "After f nodes fail, at least one read (phase 1b) and write (phase 3) quorum must consist of nodes that are available.": this is directly enforced by rule Q3.

In addition, the properties of availability and durability, which in [6] are inferred to hold because read and write quorums are chosen to be of size $k + f$, are directly enforced by rules R1 and R2 in our work. □

7.3.5 Log management with new parity functions

Consensus protocols manage some state (i.e. some version of the object) and a log of operations on that state. To avoid running out of memory, consensus protocols also have some mechanism by which to trim this log without violating correctness. With full-object writes, trimming is relatively simple: once a full-write is committed, we can update the state and trim all prior operations safely, because previous operations do not directly affect the current state of the object.

However, we have to be careful when trimming updates from the log. During normal operation, "holes" might be introduced in the operation log of datasites. This occurs when an operation is accepted by a write quorum but, for some reason, a datasite outside of the quorum does not learn about the operation. Correctness issues could emerge when there are holes in the log, because a node needs all the updates in order to update its state to the newest version. To address this, datasites include information about their log entries in other protocol messages, and keep track of the highest version other datasites can produce. Datasites only trim an update from the log once they are sure that there is at least one write quorum that can reproduce the corresponding version. If at any point a datasite falls behind and cannot fill its holes, it can execute a read operation to get up-to-date.

7.4 Evaluation

We implemented Pudu's optimizer using Python along with the CPLEX solver [227] and Google's OR-tools [229]. For Pudu's key-value store and consensus protocol, we implement everything in Go except for the EC module which we implement in C. To evaluate Pudu, we test it over a variety of settings (access sets, optimization objectives, SLOs). With our evaluation, we attempt to answer the following questions

about PUDU relative to previous systems: (1) Can PUDU achieve better tradeoffs? (2) In which situations does PUDU yield the best improvements? (3) What are the overheads of PUDU? The main takeaways are:

- In 42.37% of the target-specifications we evaluate, the solution generated by PUDU's optimizer is a Pareto-improvement over the state-of-the-art.
- On the target-specifications where PUDU produces a Pareto-improvement, on average, PUDU reduces read latency by 6.47ms, write latency by 6.77ms, storage overhead by 0.14, and update cost by 35.4%.
- When optimized to reduce WAN bandwidth, PUDU reduces WAN bandwidth usage by 11.5%, with savings of up to 82.4% in some cases.

7.4.1 Evaluation Setup

Recall that PUDU has two parts: an optimizer that generates a configuration based on an target-specification, and the distributed key-value store that runs based on the generated configuration. We consider both parts in our evaluation. We consider 25 datasites from Azure's global network, and form access sets by picking frontends from them: frontends are picked at random such that no two frontends are closer than 50ms. The `Cost Model` component within the `ConfigOptimizer` (Section 7.2) uses collected statistics on the latency of communication between every pair of Azure sites for predicting the latency of configurations during optimization.

Baselines. As a main point of comparison, we use the state-of-the-art geo-distributed storage system, which also uses optimization-based consensus configuration, Pando [6]. To compare fairly, we enhance Pando with the ability to control update-cost via the parameters n and k of the RS code (recall that for systematic RS codes update-cost is $n - k + 1$), and add support for UPDATE operations. With the enhancements, this baseline represents a strong class of baselines that employ optimization to design consensus configurations and use parameter tuning to tailor the RS code for given input settings. We call this class as "enhanced-SOTA".

For `Optimized Dimension`, we consider four dimensions: *read latency, write latency, storage-overhead, and update-cost*. When optimizing, we constrain two of these dimensions using a set of SLOs, and minimize the remaining two dimensions in a particular chosen order: i.e. the second dimension is minimized subject to the first dimension attaining the minimum. We evaluate by constraining different dimensions and using different optimization orders. E.g., we might constrain read and write latency via SLOs, and then minimize update-cost first, and storage-overhead second. This means that the optimizer will first minimize update-cost subject to the read and write latency constraints and then, out of all the solutions that achieve minimum update-cost, it will choose one that minimizes storage-overhead.

We measure WAN bandwidth consumption by running the generated solutions on synthetic workloads with varying parameters. Each frontend and datasite is run as a different process, and latency of between processes is simulated using the inter-site latency statistics collected from Azure. WAN bandwidth corresponds to the number of bytes sent over-the-wire between processes. The synthetic workload consists of a PUT operation followed by a sequence of UPDATEs at random offsets, executed by frontends in a round-robin fashion. The workload has three parameters: the size of the object, the number of bytes modified in UPDATEs, and the ratio of UPDATEs to PUTs. We sweep through these parameters to cover a broad range of workloads.

7.4.2 Pudu achieves better tradeoffs

In this section, we compare the solutions produced by Pudu and enhanced-SOTA in terms of the four dimensions considered: read latency, write latency, storage-overhead, and update-cost. We show that Pudu is able to produce solutions which achieve better tradeoffs across the considered dimensions. We run two sets of experiments: in the first set we minimize update-cost first, then one of the other three dimensions; in the second set we first minimize one of the other three dimensions, then update-cost. The dimension is optimized first will be referred to as the "primary objective" and the dimension that is optimized second will be referred to as the "secondary objective". Below we will see that while enhanced-SOTA can come close to Pudu in update-cost

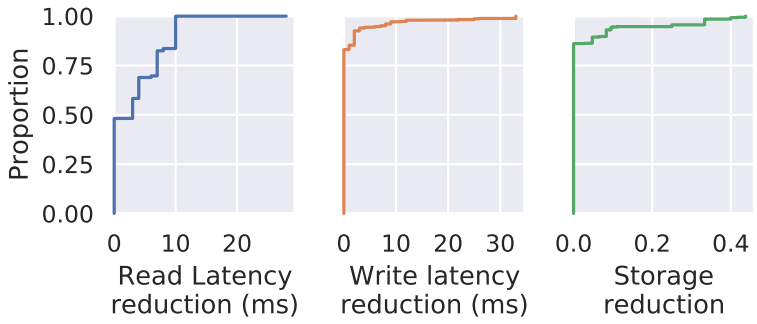


Figure 7.6: Reductions achieved when update-cost is minimized first.

when it is the primary objective, the choices it makes to achieve this make enhanced-SOTA significantly worse than PUDU in the secondary objective. Nonetheless, even when read/write latency or storage overhead is the primary objective, PUDU performs significantly better in both the primary and secondary objectives. PUDU, due to the flexibility offered by density-aware redundancy, can achieve the best of both worlds, that is perform well on both the primary and the secondary objective.

Minimizing update-cost first

We find that, in nearly all cases, both systems are able to achieve the same minimum update-cost; however, PUDU is able to perform significantly better in the other optimization dimension. We find that PUDU achieves very small improvements in update-cost compared to enhanced-SOTA (1.56% reduction on average across all experiments). However, PUDU is able to perform significantly better in the secondary objective. The reason for this is that in most cases enhanced-SOTA can adjust n and k so that $n - k$ is relatively low, but this hurts its ability to optimize for other objectives. Figure 7.6 shows the cumulative distribution function (CDF) of the reductions. We observe that PUDU is able to improve over enhanced-SOTA in many instances: 51.9% of them when optimizing read latency (average reduction of 6.47ms up to 28ms), 16.8% for write latency (average reduction of 6.77ms up to 33ms), and 16.7% for storage-overhead (average reduction of 0.14 up to 0.44). The range of the

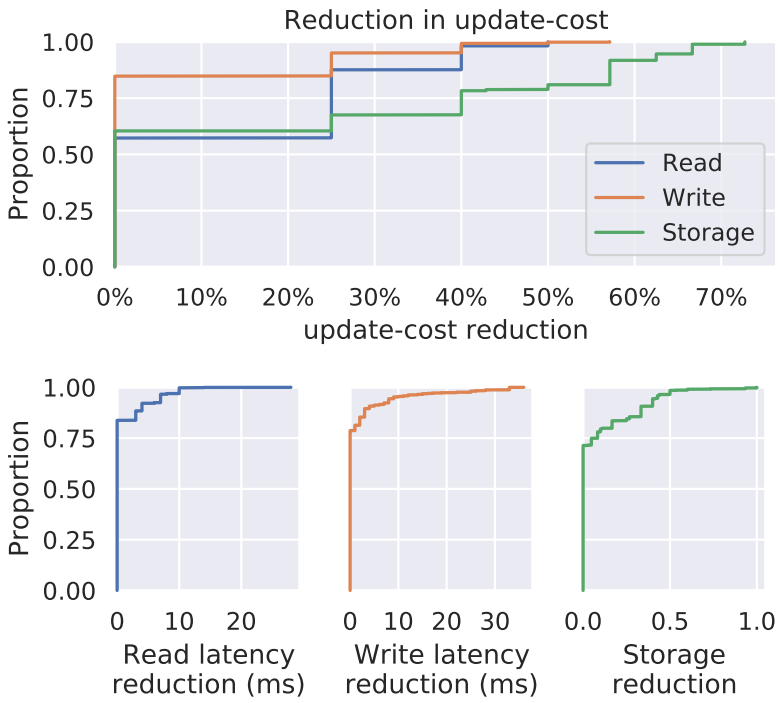


Figure 7.7: Reductions achieved when update-cost is minimized second.

Figure 7.8: WAN bandwidth savings of PUDU compared to enhanced-SOTA (notice y-axis starts at 0.5). The plot shows the CDF of the distribution of WAN bandwidth savings over all target-specifications, divided according to the size of updates (left) and the ratio of update operations (right).

improvements, however, is very broad: these go up to 3 in update-cost, 28ms in read latency, 33ms in write latency, and 0.44 in storage-overhead.

Minimizing update-cost second

In this case, we find that PUDU is able to achieve improvements in both of the optimization dimensions: Figure 7.7 shows the CDFs for these improvements. In particular, PUDU achieves significant reduction in update-cost (Figure 7.7, top): 34.9% of the target-specifications see an improvement, with an average reduction of 35.4%. This is because enhanced-SOTA must add many additional parities to minimize whichever dimension is minimized first (here either read latency, write latency, or storage overhead). This results in a high update-cost.

When it comes to the dimension that is optimized first (Figure 7.7, bottom), the improvements in read/write latency are moderate (0.95ms and 1.68ms in average, respectively), while the improvement in storage-overhead is more significant (0.08 in average). This is because the flexibility of PUDU to place different amounts of data in each datasite allows it to minimize storage-overhead much more effectively. As in the previous set of experiments, the range is very broad, with reductions of up to 4 in update cost, 28ms is read latency, 36ms in write latency, and 1 in storage-overhead.

7.4.3 Pudu reduces WAN bandwidth

In this subsection, we use the configurations generated in the experiments in Section 7.4.2. In particular, we focus on the experiments that minimize storage-overhead and update-cost (in that order), as they show the largest improvements in the dimensions that affect WAN bandwidth. Figure 7.8 shows the savings in WAN bandwidth for the generated solutions. We observe that in average WAN bandwidth is reduced by 11.5%, with ten percent experiencing a reduction of over 42.3%. When the size of updates is a small fraction of the object, improvements in WAN bandwidth are smaller (Figure 7.8, top): when 10% of the object is updated in each operation, the average reduction is 15.7%, but when 0.01% of the object is updated in each operation, the average reduction is 5.2%. This is because messages are composed of two parts: protocol information related to consensus, and data payloads. When updates are small, the payload will be relatively small compared to the rest of the message. This is further amplified by the fact that Pudu tends to generate solutions which use more datasites, which increases the amount of protocol messages sent. We also observe (Figure 7.8, bottom) that improvements in WAN bandwidth are bigger when the ratio of UPDATE operations is larger: when there is 10 UPDATEs for every PUT the average savings are 10.1%; when there is 50 UPDATEs for every PUT the average savings are 12.7%. This is due to the fact that the improvement is mainly driven by a reduction in update-cost.

7.4.4 Effect of constraints on savings

We further analyze the results of Section 7.4.2 to identify the cases in which Pudu is able to provide best improvements. To measure this we analyse the avarage reduction in the optimization dimension against different values of the constraints. Figure 7.9 shows the results when update-cost is minimized first. We find that write latency constraints do not have a large effect, except when the constraint is very restrictive. This is because write quorums are usually the largest, and thus will not have a very large impact on the minimal information sets. For read latency, we observe the biggest improvements in the middle: when the constraints are not too strict or too

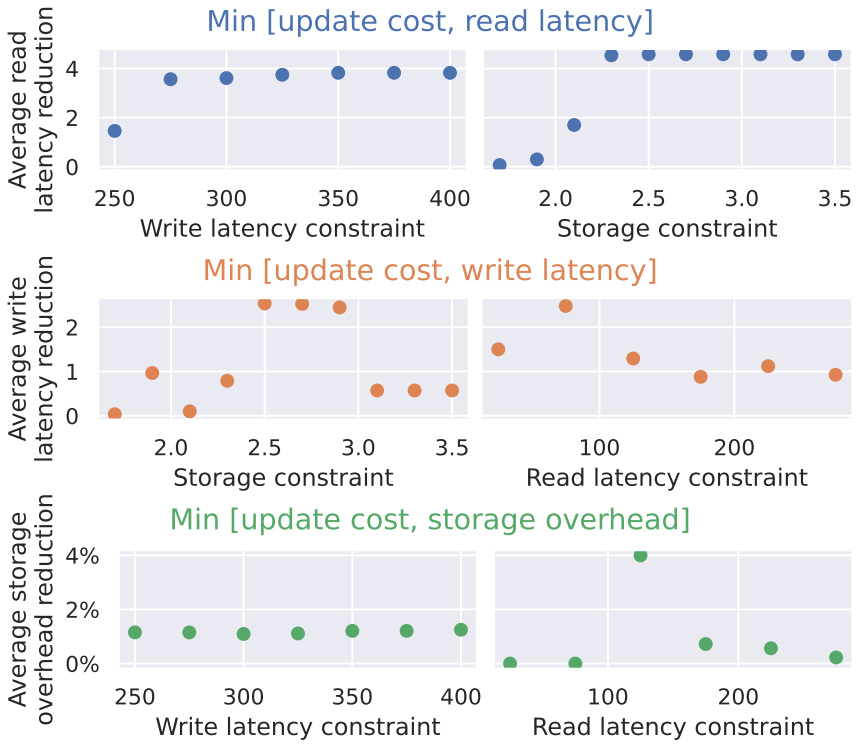


Figure 7.9: Effect of constraints on reductions achieved when update-cost is optimized first. Average reduction in read/write latency, and storage overhead is respectively shown in the first, second, and third rows.

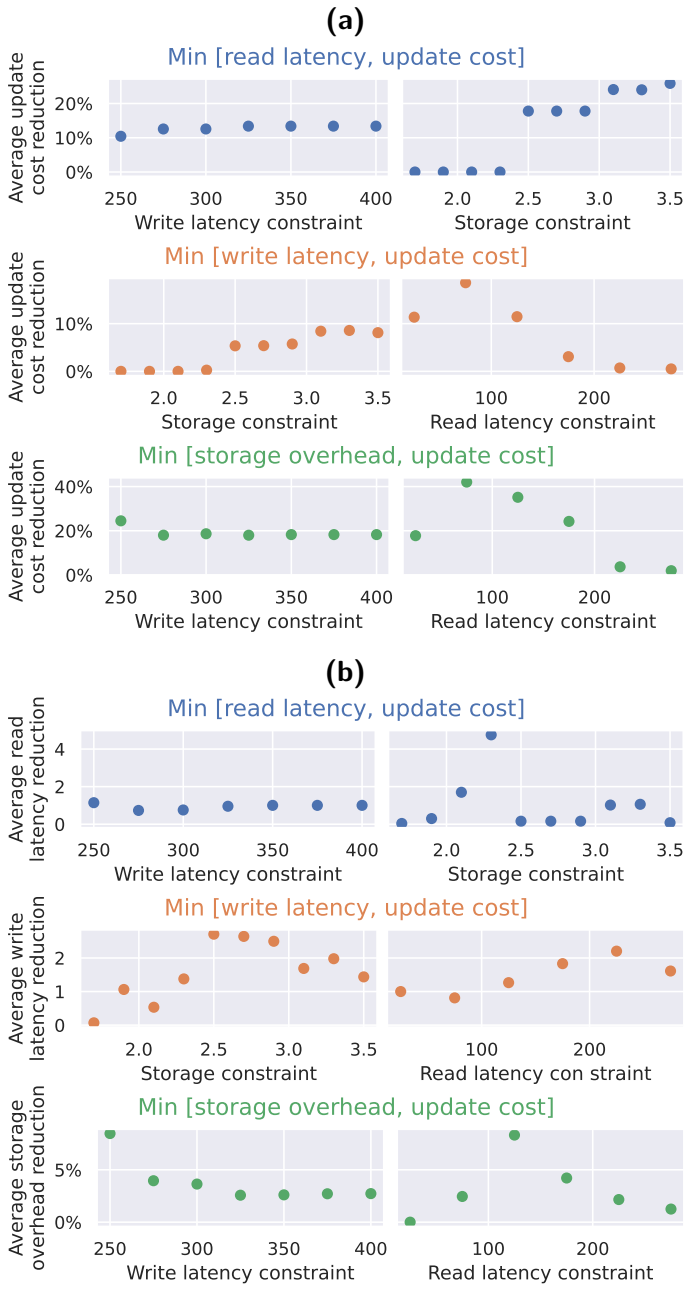


Figure 7.10: Effect of constraints on average reduction of update-cost ((a)) and the other three dimensions (read/write latency, storage) ((b)) when update-cost is minimized second.

lax. When the constraint is too strict, the placement of data is very constrained and thus will always result in a solution similar to replication. When the constraint is too lax, quorums can be chosen to be very large, which results in very homogeneous information sets, and thus there is not much room for density-aware redundancy to provide benefits. Finally, the relationship with the storage constraint is more complex: because the storage overhead of RS codes is $\frac{n}{k}$ and n and k cannot take very large values, RS codes are not always able to use the full storage budget. This leads to bad solutions when the storage overhead SLO is not close to a simple ratio, in which cases PUDU can improve the solution by taking advantage of the whole storage budget.

Figure 7.10 shows the results when update-cost is minimized second, with Figure 7.10a showing the effect on update-cost and Figure 7.10b showing the effect on the dimension that is optimized first (read/write latency or storage overhead). In Figure 7.10a, we see that write latency does not have a large impact on the reduction, and that the biggest improvements are for read latency constraints in the mid-range. We also see that when the storage overhead constraint is large, PUDU sees the largest improvements in update-cost; this is because it is hard for RS codes to reduce update-cost when they have a large number of parities. In Figure 7.10b, we observe a similar behaviour as with the set of experiments in Figure 7.9.

7.4.5 Effect of heterogeneity on results

As discussed through the chapter, the main advantage of PUDU is that it can adapt to heterogeneity in latencies to reduce cost. To test this, we artificially modify the latencies between datasites to make them more homogeneous and see how it affects the results. Figure 7.11 shows the average reductions obtained for three trials: one where latencies are unmodified, one where the standard deviation of latencies is scaled by one half, and one where it is scaled by one fourth. We observe that in almost all cases the magnitude of the improvements of PUDU reduces as the network becomes more homogeneous.

7.4.6 Overheads of Pudu

In this section, we measure the overheads of Pudu compared to the state-of-the-art.

Metadata

Given that Pudu generates a configuration and erasure code that are more closely tailored to each target-specification, there is an extra overhead in terms of metadata associated to each access profile. Compared to enhanced-SOTA, Pudu needs to explicitly specify quorums, erasure code, and information sets as metadata. On average, Pudu requires 72.9 extra bytes, and in the worst case it requires 1649 per access profile. This overhead is negligible since it will be amortized across all the multiple objects that use the same access profile.

Optimizer time

Pudu's optimizer is more complex than the optimizer used by enhanced-SOTA, and thus it still stands to reason that it will be slower in finding solutions. We measure the total amount of time spent solving each input in the experiments above. The median, p90, p99 times (in seconds) for Pudu are $(38, 542, 1575)$. On average, Pudu takes 1.68 times longer than enhanced-SOTA. These times are acceptable in practice, because (1) the optimizer is run offline and is not in the critical path, (2) the solutions of the optimizer remain valid for long periods of time (and need to be re-run only in the rare cases where latencies between sites change significantly).

Erasure coding overhead

The erasure code that is produced by the `ECOptimizer` of Pudu is always a linear code, and it is implemented using the ISA-L library [230]. Given that Pudu's EC module is more complex than a typical RS code, we might expect worse performance. To test this, we benchmark the EC module with the different codes generated in previous experiments, by encoding and decoding objects of sizes from 100B to 100kB. We observe that Pudu does take more time to encode objects than an RS

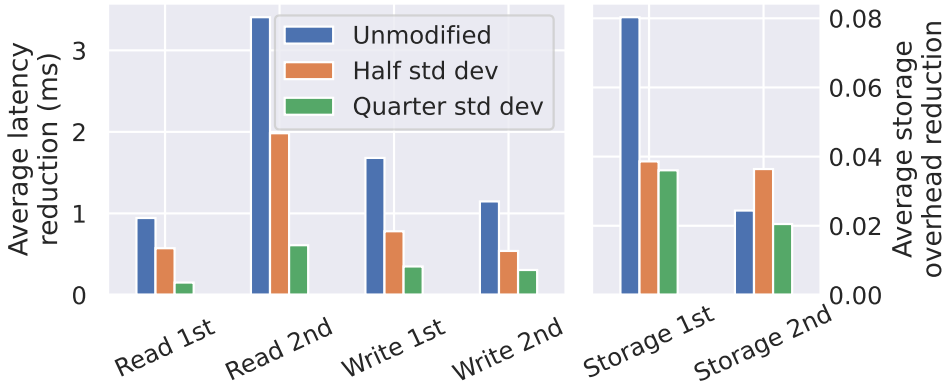


Figure 7.11: Effect of heterogeneity in density: improvements of PUDU reduce as the network of datasites becomes more homogeneous.

code, especially smaller objects, where the overhead of our implementation is more noticeable. However, the time to encode/decode an object never exceeds 1ms, and is typically much smaller than that. Given that latencies between datasites are in tens of ms, this overhead to not significant. Moreover, the implementation has not been optimized to reduce encoding/decoding times, and such an engineering effort can further reduce the encoding/decoding times.

7.5 Related work

Multiple systems for geo-distributed storage have been proposed: some with weak consistency guarantees [210, 231–233], and some with strong consistency guarantees [177, 234–236]. Several systems have been specially designed to allow consensus with low latency [212, 215, 237]. Some systems use replication along with special data placement to keep cost low [211, 238].

Pando [6] optimizes the configuration of a geo-distributed storage system to reduce cost. However, Pando only uses RS codes, and does not directly consider the impact the WAN bandwidth in cost. Other works that have also explored the use of erasure codes in consensus protocols are [5, 219].

Erasure codes have been applied in several other storage systems to provide

reliability at a low cost. After the influential system of RAID [21], several other systems have used erasure coding for storing data [178, 180, 239–241]. More recently, several works have explored the idea of adapting the parameters of an RS code to adapt to changes in failure rates [1, 15, 120].

Part III

Future directions

Chapter 8

Future directions for Part I

In the first part of this thesis, we have focused on studying the change of storage codes and distributed storage systems through time. In the theoretical front, we have proposed the convertible codes framework and the conversion problem as ways of formalizing and studying storage code changes. On the practical front, we have proposed new distributed storage systems which are designed with storage code changes in mind, and which are able to perform these changes in ways that are efficient and robust.

Our work shows that the conversion problem has a lot of depth from both a theoretical and practical point of view, and that there is a lot of potential for improving the efficiency of production distributed storage systems and reducing their operations costs. However, there are still many related open problems which are yet to be explored.

8.1 Future directions for convertible codes

8.1.1 Conversion bandwidth in the general regime

The main unsolved problem in our study of linear MDS convertible codes is in deriving lower bounds for the conversion bandwidth of the general regime, as well as optimal constructions that match said lower bounds. For example, in the case where $r^I > r^F$

in the split regime (Section 3.6), the best known lower bound is not tight (and the best known construction is only conjectured to be optimal). Compared to access cost, conversion bandwidth lower bounds are more difficult to analyze because conversion bandwidth is more fine-grained than access cost. This is because to minimize access cost it suffices to consider scalar codes, whereas to minimize conversion bandwidth, we must consider vector codes: in a scalar code, either the whole symbol is read or not read, but in a vector code, we can read only part of a symbol. Thus, a more detailed analysis is required in the case of conversion bandwidth. Similarly, on the construction side, it is harder to reuse the constructions from the merge and split regimes in the general regime, because they both require the use of vector codes, which are more complex than scalar codes. Composing these vector code constructions, while possible, is less straightforward than their scalar counterparts.

8.1.2 Practical general constructions

In the research presented in this thesis, we have made a special effort to devise constructions and techniques that can be implemented in practice, considering the technical limitations of real distributed storage systems in use today. Despite our best efforts, however, there are still cases where our constructions are not as flexible and easy-to-deploy as popular general-purpose storage codes (like Reed-Solomon codes).

For example, while in many cases our access-optimal constructions have a low field-size requirement, there are some cases (chiefly, $r^I = r^F$) where the best known construction has a very high field size (see Chapter 2). Discovering constructions that have low-field size requirement in all cases is still an open problem and would help convertible codes be usable in a wider array of situations.

On a similar vein, subpacketization (i.e. the length of vectors in a vector code), is of great practical importance. Subpacketization is required for bandwidth-optimal convertible codes (see Chapter 3), however, it is not clear if our constructions have minimum subpacketization, or what is the tradeoff when subpacketization is limited.

8.1.3 Multiple/branching conversions

Although, most of our work in this thesis focuses on single code conversions, we have discussed at several points how to support multiple subsequent conversions, or branching conversions where the set of final parameters is unknown ahead of time. However, our current solution to these problems in the case of conversion-bandwidth minimization relies on composing the codes for different single conversions. This solution results in codes with very high subpacketization, which is not practical because high subpacketization can hurt performance. Therefore, to solve this problem in practice, a better approach is needed (one that might not necessarily be bandwidth-optimal, for example).

In the above, we considered the problem of multiple/branching conversions from a coding-theoretical point of view. However, there are also many open questions when we consider this problem from a "systems" perspective. For example: How should a distributed storage system choose the sequence of erasure code parameters that will be used during these multiple conversions? When and how should the system perform the conversions? These questions are non-trivial, and would need to be solved before implementing this into a real system.

8.1.4 Code conversions beyond MDS codes

In Chapter 4, we considered conversion between locally-recoverable codes (with information locality). However, there is a vast array of storage codes proposed in the Coding Theory literature, such as LRCs with all-symbol locality [65], partial-MDS or maximally-recoverable codes [68], regenerating codes [27], and many other non-MDS codes. Investigating conversions within or across these different classes of codes is an important avenue for future research.

8.2 Future directions for disk-adaptive redundancy

8.2.1 Disk-adaptive redundancy with convertible codes

On the system front, there are also vast opportunities for future work. One important opportunity for future work is to design a fully-featured distributed storage system that utilizes convertible codes. The typical design of distributed storage systems used in practice, makes the integration of convertible codes challenging. This is because these systems typically assume a strict data layout which may not be preserved when combining data from multiple codewords (as the convertible codes framework does). To obtain the full benefits of convertible codes, it is necessary to modify the way these systems deal with data layouts and the management of codewords.

8.2.2 Enhancing the reliability of disk-adaptive redundancy systems

Another avenue for future work on distributed storage systems is in improving the monitoring of disk failure rates. For example, the disk-adaptive redundancy techniques that we describe could be further improved by introducing machine learning algorithms capable of predicting changes in AFRs, and incorporating those predictions into the redundancy management. Similarly, the reliability models used for assessing the health of the system (such as the calculation of MTTDLs) could be made more robust to variations in AFR over time, or correlated failures, as well as more efficient to compute.

Chapter 9

Future directions for Part II

In Chapter 6, we introduced MUC codes which, given an arbitrary collection of sets that need to decode an object (access set), it minimizes update cost and then storage overhead. Our results show that in this setting, Reed-Solomon are not optimal and we can improve by designing a code that is tailored to the required access sets. Then, in Chapter 7, we integrated the MUC codes framework into a strongly-consistent geo-distributed storage system. By using this framework, we co-optimized the design of the erasure code along with the configuration of the protocol in order to minimize the operating cost of the system. Our work shows that the MUC codes framework is useful in practice, and it can be used to achieve significant savings in operating cost.

9.1 Future directions for MUC codes

As future work, there are many opportunities around the design of codes with arbitrary access sets. Compared to the classical setting where every subset of a certain size is an access set, this setting with arbitrary access sets is relatively uncharted. Our work focuses on only two metrics (storage overhead and update cost) and proposes a randomized construction. There are, however, multiple other aspects that can be considered in the design of a code (such as repair, or convertibility).

However, even for the metrics that we consider, there are still opportunities for

future work. The construction that we give in Chapter 6, is randomized. It would be of interest to have a deterministic construction: such a construction could give a better understanding of the problem and open up the opportunity to improve the field size of the code or other practical metrics.

9.2 Future directions for geo-distributed storage systems

Our work in Chapter 7 considers a specific setting where the system is optimized to a particular network topology and arrangement of users in different regions. However, these aspects might change throughout time because of users being added or departing, or because of changes in the network (which may or may not be transient). In these cases, the optimal configuration of the system and storage code might change. Clearly, as in the case of convertible codes, re-encoding data and redistributing data is always possible, though it might be expensive. Therefore, a interesting future direction could be studying how to design the system and storage code to accommodate these changes more efficiently.

Bibliography

[1] Saurabh Kadekodi, K. V. Rashmi, and Gregory R. Ganger. Cluster storage systems gotta have HeART: improving storage efficiency by exploiting disk-reliability heterogeneity. In Arif Merchant and Hakim Weatherspoon, editors, *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*, pages 345–358. USENIX Association, 2019. URL https://www.usenix.org/conference/fast19/presentation/kadekodi.

[2] Google Docs. Google Docs: Online document editor. https://www.google.com/docs/about/, 2022. Accessed: 2022-10-03.

[3] Overleaf. Overleaf, Online LaTeX Editor. https://www.overleaf.com/, 2022. Accessed: 2022-10-03.

[4] Shuai Mu, Kang Chen, Yongwei Wu, and Weimin Zheng. When paxos meets erasure code: Reduce network and storage cost in state machine replication. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '14, page 61–72, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327497. doi: 10.1145/2600212.2600218. URL https://doi.org/10.1145/2600212.2600218.

[5] Zizhong Wang, Tongliang Li, Haixia Wang, Airan Shao, Yunren Bai, Shangming Cai, Zihan Xu, and Dongsheng Wang. CRaft: An erasure-coding-supported version of raft for reducing storage cost and network cost. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 297–308, Santa Clara,

CA, February 2020. USENIX Association. ISBN 978-1-939133-12-0. URL https://www.usenix.org/conference/fast20/presentation/wang-zizhong.

[6] Muhammed Uluyol, Anthony Huang, Ayush Goel, Mosharaf Chowdhury, and Harsha V. Madhyastha. Near-optimal latency versus cost tradeoffs in geo-distributed storage. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 157–180. USENIX Association, 2020.

[7] Francisco Maturana and K. V. Rashmi. Convertible codes: enabling efficient conversion of coded data in distributed storage. *IEEE Transactions on Information Theory*, 68:4392–4407, 2022. ISSN 1557-9654. doi: 10.1109/TIT.2022.3155972.

[8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In Michael L. Scott and Larry L. Peterson, editors, *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 29–43. ACM, 2003. doi: 10.1145/945445.945450.

[9] Dhruba Borthakur, Rodrigo Schmidt, Ramkumar Vadali, Scott Chen, and Patrick Kling. HDFS RAID - Facebook (presentation), 2010. URL http://www.slideshare.net/ydn/hdfs-raid-facebook.

[10] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in Windows Azure storage. In Gernot Heiser and Wilson C. Hsieh, editors, *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, pages 15–26. USENIX Association, 2012. URL https://www.usenix.org/conference/atc12/technical-sessions/presentation/huang.

[11] Apache Software Foundation. Apache Hadoop documentation: HDFS erasure coding, 2019. URL https://hadoop.apache.org/docs/r3.0.0/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html. Accessed: 2019-07-23.

[12] K. V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster. In Ajay Gulati, editor, *5th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage'13, San Jose, CA, USA, June 27-28, 2013*. USENIX Association, 2013. URL https://www.usenix.org/conference/hotstorage13/workshop-program/presentation/rashmi.

[13] K. V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A "hitchhiker's" guide to fast and efficient data reconstruction in erasure-coded data centers. In Fabián E. Bustamante, Y. Charlie Hu, Arvind Krishnamurthy, and Sylvia Ratnasamy, editors, *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, pages 331–342. ACM, 2014. doi: 10.1145/2619239.2626325.

[14] Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. Xoring elephants: Novel erasure codes for big data. *Proceedings of the VLDB Endowment*, 6(5), 2013.

[15] Saurabh Kadekodi, Francisco Maturana, Suhas Jayaram Subramanya, Juncheng Yang, K. V. Rashmi, and Gregory R. Ganger. PACEMAKER: Avoiding HeART attacks in storage clusters with disk-adaptive redundancy. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 369–385. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/kadekodi.

[16] Jianzhong Huang, Xianhai Liang, Xiao Qin, Ping Xie, and Changsheng Xie. Scale-RS: an efficient scaling scheme for RS-coded storage clusters. *IEEE Transactions on Parallel and Distributed Systems*, 26(6):1704–1717, 2015. doi: 10.1109/TPDS.2014.2326156.

[17] Brijesh Kumar Rai. On adaptive (functional MSR code based) distributed storage systems. In *2015 International Symposium on Network Coding, NetCod*

2015, Sydney, Australia, June 22-24, 2015, pages 46–50. IEEE, 2015. doi: 10.1109/NETCOD.2015.7176787.

[18] M. Sonowal and B. K. Rai. On adaptive distributed storage systems based on functional MSR code. In *Proceedings of the Signal Processing and Networking (WiSPNET) 2017 International Conference on Wireless Communications*, pages 338–343, 2017. doi: 10.1109/WiSPNET.2017.8299774.

[19] Yuchong Hu, Xiaoyang Zhang, Patrick P. C. Lee, and Pan Zhou. Generalized optimal storage scaling via network coding. In *2018 IEEE International Symposium on Information Theory, ISIT 2018, Vail, CO, USA, June 17-22, 2018*, pages 956–960. IEEE, 2018. doi: 10.1109/ISIT.2018.8437684.

[20] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960. doi: 10.1137/0108018. URL https://doi.org/10.1137/0108018.

[21] David A. Patterson, Garth A. Gibson, and Randy H. Katz. A case for Redundant Arrays of Inexpensive Disks (RAID). In Haran Boral and Per-Åke Larson, editors, *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 1-3, 1988*, pages 109–116. ACM Press, 1988. doi: 10.1145/50202.50214.

[22] J.S. Plank. T1: Erasure codes for storage applications. *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, pages 1–74, 01 2005.

[23] Mario Blaum, Jim Brady, Jehoshua Bruck, and Jai Menon. EVENODD: an efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Transactions on Computers*, 44(2):192–202, 1995. doi: 10.1109/12.364531.

[24] Lihao Xu and Jehoshua Bruck. X-code: MDS array codes with optimal encoding. *IEEE Transactions on Information Theory*, 45(1):272–276, 1999. doi: 10.1109/18.746809.

[25] Cheng Huang and Lihao Xu. STAR : an efficient coding scheme for correcting triple storage node failures. *IEEE Transactions on Computers*, 57(7):889–901, 2008. doi: 10.1109/TC.2007.70830.

[26] James Lee Hafner. WEAVER codes: highly fault tolerant erasure codes for storage systems. In Garth Gibson, editor, *Proceedings of the FAST '05 Conference on File and Storage Technologies, December 13-16, 2005, San Francisco, California, USA*. USENIX, 2005. URL http://www.usenix.org/events/fast05/tech/hafner_weaver.html.

[27] Alexandros G. Dimakis, Brighten Godfrey, Yunnan Wu, Martin J. Wainwright, and Kannan Ramchandran. Network coding for distributed storage systems. *IEEE Transactions on Information Theory*, 56(9):4539–4551, 2010. doi: 10.1109/TIT.2010.2054295.

[28] K. V. Rashmi, Nihar B. Shah, and P. Vijay Kumar. Optimal exact-regenerating codes for distributed storage at the MSR and MBR points via a product-matrix construction. *IEEE Transactions on Information Theory*, 57(8):5227–5239, 2011. doi: 10.1109/TIT.2011.2159049.

[29] Nihar B Shah, K. V. Rashmi, P Vijay Kumar, and Kannan Ramchandran. Interference alignment in regenerating codes for distributed storage: Necessity and code constructions. *IEEE Transactions on Information Theory*, 58(4): 2134–2158, 2011.

[30] Changho Suh and Kannan Ramchandran. Exact-repair MDS code construction using interference alignment. *IEEE Transactions on Information Theory*, 57(3): 1425–1442, 2011. doi: 10.1109/TIT.2011.2105003.

[31] Zhiying Wang, Itzhak Tamo, and Jehoshua Bruck. On codes for optimal rebuilding access. In *49th Annual Allerton Conference on Communication, Control, and Computing, Allerton 2011, Allerton Park & Retreat Center, Monticello, IL, USA, 28-30 September, 2011*, pages 1374–1381. IEEE, 2011. doi: 10.1109/Allerton.2011.6120327.

[32] Viveck R. Cadambe, Cheng Huang, Jin Li, and Sanjeev Mehrotra. Polynomial length MDS codes with optimal repair in distributed storage. In Michael B. Matthews, editor, *Conference Record of the Forty Fifth Asilomar Conference on Signals, Systems and Computers, ACSCC 2011, Pacific Grove, CA, USA, November 6-9, 2011*, pages 1850–1854. IEEE, 2011. doi: 10.1109/ACSSC.2011.6190343.

[33] Nihar B Shah, K Vinayak Rashmi, P Vijay Kumar, and Kannan Ramchandran. Distributed storage codes with repair-by-transfer and nonachievability of interior points on the storage-bandwidth tradeoff. *IEEE Transactions on Information Theory*, 58(3):1837–1852, 2011.

[34] Zhiying Wang, Itzhak Tamo, and Jehoshua Bruck. Long MDS codes for optimal repair bandwidth. In *2012 IEEE International Symposium on Information Theory, ISIT 2012, Cambridge, MA, USA, July 1-6, 2012*, pages 1182–1186. IEEE, 2012. doi: 10.1109/ISIT.2012.6283041.

[35] Itzhak Tamo, Zhiying Wang, and Jehoshua Bruck. Zigzag codes: MDS array codes with optimal rebuilding. *IEEE Transactions on Information Theory*, 59 (3):1597–1616, 2013. doi: 10.1109/TIT.2012.2227110.

[36] Viveck R. Cadambe, Syed Ali Jafar, Hamed Maleki, Kannan Ramchandran, and Changho Suh. Asymptotic interference alignment for optimal repair of MDS codes in distributed storage. *IEEE Transactions on Information Theory*, 59(5):2974–2987, 2013. doi: 10.1109/TIT.2013.2237752.

[37] Dimitris S. Papailiopoulos, Alexandros G. Dimakis, and Viveck R. Cadambe. Repair optimal erasure codes through Hadamard designs. *IEEE Transactions on Information Theory*, 59(5):3021–3037, May 2013. doi: 10.1109/TIT.2013.2241819.

[38] Birenjith Sasidharan, Gaurav Kumar Agarwal, and P. Vijay Kumar. A high-rate MSR code with polynomial sub-packetization level. In *IEEE International*

Symposium on Information Theory, ISIT 2015, Hong Kong, China, June 14-19, 2015, pages 2051–2055. IEEE, 2015. doi: 10.1109/ISIT.2015.7282816.

[39] Min Ye and Alexander Barg. Explicit constructions of MDS array codes and RS codes with optimal repair bandwidth. In *IEEE International Symposium on Information Theory, ISIT 2016, Barcelona, Spain, July 10-15, 2016*, pages 1202–1206. IEEE, 2016. doi: 10.1109/ISIT.2016.7541489.

[40] Min Ye and Alexander Barg. Explicit constructions of optimal-access MDS codes with nearly optimal sub-packetization. *IEEE Transactions on Information Theory*, 63(10):6307–6317, 2017. doi: 10.1109/TIT.2017.2730863.

[41] Ankit Singh Rawat, Onur Ozan Koyluoglu, and Sriram Vishwanath. Progress on high-rate MSR codes: enabling arbitrary number of helper nodes. In *2016 Information Theory and Applications Workshop, ITA 2016, La Jolla, CA, USA, January 31 - February 5, 2016*, pages 1–6. IEEE, 2016. doi: 10.1109/ITA.2016.7888191.

[42] Birenjith Sasidharan, Myna Vajha, and P. Vijay Kumar. An explicit, coupled-layer construction of a high-rate MSR code with low sub-packetization level, small field size and $d < (n-1)$. In *2017 IEEE International Symposium on Information Theory, ISIT 2017, Aachen, Germany, June 25-30, 2017*, pages 2048–2052. IEEE, 2017. doi: 10.1109/ISIT.2017.8006889.

[43] Sreechakra Goparaju, Arman Fazeli, and Alexander Vardy. Minimum storage regenerating codes for all parameters. *IEEE Transactions on Information Theory*, 63(10):6318–6328, 2017. doi: 10.1109/TIT.2017.2690662.

[44] Ameera Chowdhury and Alexander Vardy. New constructions of MDS codes with asymptotically optimal repair. In *2018 IEEE International Symposium on Information Theory, ISIT 2018, Vail, CO, USA, June 17-22, 2018*, pages 1944–1948. IEEE, 2018. doi: 10.1109/ISIT.2018.8437590.

[45] Kaveh Mahdaviani, Ashish Khisti, and Soheil Mohajer. Bandwidth adaptive & error resilient MBR exact repair regenerating codes. *IEEE Transactions on Information Theory*, 65(5):2736–2759, 2019. doi: 10.1109/TIT.2018.2878223.

[46] Kaveh Mahdaviani, Soheil Mohajer, and Ashish Khisti. Product matrix MSR codes with bandwidth adaptive exact repair. *IEEE Transactions on Information Theory*, 64(4):3121–3135, 2018. doi: 10.1109/TIT.2018.2796599.

[47] Ankit Singh Rawat, Itzhak Tamo, Venkatesan Guruswami, and Klim Efremenko. MDS code constructions with small sub-packetization and near-optimal repair bandwidth. *IEEE Transactions on Information Theory*, 64(10):6506–6525, 2018. doi: 10.1109/TIT.2018.2810095.

[48] Nihar B. Shah, K. V. Rashmi, and P. Vijay Kumar. A flexible class of regenerating codes for distributed storage. In *IEEE International Symposium on Information Theory, ISIT 2010, June 13-18, 2010, Austin, Texas, USA, Proceedings*, pages 1943–1947. IEEE, 2010. doi: 10.1109/ISIT.2010.5513353.

[49] Kenneth W. Shum. Cooperative regenerating codes for distributed storage systems. In *Proceedings of IEEE International Conference on Communications, ICC 2011, Kyoto, Japan, 5-9 June, 2011*, pages 1–5. IEEE, 2011. doi: 10.1109/icc.2011.5962548.

[50] Vitaly Abdrashitov, N. Prakash, and Muriel Médard. The storage vs repair bandwidth trade-off for multiple failures in clustered storage networks. In *2017 IEEE Information Theory Workshop, ITW 2017, Kaohsiung, Taiwan, November 6-10, 2017*, pages 46–50. IEEE, 2017. doi: 10.1109/ITW.2017.8277979.

[51] Karthikeyan Shanmugam, Dimitris S. Papailiopoulos, Alexandros G. Dimakis, and Giuseppe Caire. A repair framework for scalar MDS codes. *IEEE Journal on Selected Areas in Communications*, 32(5):998–1007, 2014. doi: 10.1109/JSAC.2014.140519.

[52] Venkatesan Guruswami and Mary Wootters. Repairing Reed-Solomon codes. *IEEE Transactions on Information Theory*, 63(9):5684–5698, 2017. doi: 10.1109/TIT.2017.2702660.

[53] Hoang Dau and Olgica Milenkovic. Optimal repair schemes for some families of full-length Reed-Solomon codes. In *2017 IEEE International Symposium on Information Theory, ISIT 2017, Aachen, Germany, June 25-30, 2017*, pages 346–350. IEEE, 2017. doi: 10.1109/ISIT.2017.8006547.

[54] Itzhak Tamo, Min Ye, and Alexander Barg. Optimal repair of Reed-Solomon codes: achieving the cut-set bound. In Chris Umans, editor, *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*, pages 216–227. IEEE Computer Society, 2017. doi: 10.1109/FOCS.2017.28.

[55] Jay Mardia, Burak Bartan, and Mary Wootters. Repairing multiple failures for scalar MDS codes. *IEEE Transactions on Information Theory*, 65(5):2661–2672, 2018.

[56] Hoang Dau, Iwan M. Duursma, Han Mao Kiah, and Olgica Milenkovic. Repairing Reed-Solomon codes with multiple erasures. *IEEE Transactions on Information Theory*, 64(10):6567–6582, 2018. doi: 10.1109/TIT.2018.2827942.

[57] Itzhak Tamo, Min Ye, and Alexander Barg. The repair problem for Reed–Solomon codes: Optimal repair of single and multiple erasures with almost optimal node size. *IEEE Transactions on Information Theory*, 65(5):2673–2695, 2018.

[58] S. B. Balaji and P. Vijay Kumar. A tight lower bound on the sub-packetization level of optimal-access MSR and MDS codes. In *2018 IEEE International Symposium on Information Theory, ISIT 2018, Vail, CO, USA, June 17-22, 2018*, pages 2381–2385. IEEE, 2018. doi: 10.1109/ISIT.2018.8437486.

[59] K. V. Rashmi, Preetum Nakkiran, Jingyan Wang, Nihar B Shah, and Kannan Ramchandran. 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)*, pages 81–94, 2015.

[60] Itzhak Tamo, Zhiying Wang, and Jehoshua Bruck. Access versus bandwidth in codes for storage. *IEEE Transactions on Information Theory*, 60(4):2028–2037, 2014. doi: 10.1109/TIT.2014.2305698.

[61] Sreechakra Goparaju, Itzhak Tamo, and A. Robert Calderbank. An improved sub-packetization bound for minimum storage regenerating codes. *IEEE Transactions on Information Theory*, 60(5):2770–2779, 2014. doi: 10.1109/TIT.2014.2309000.

[62] Omar Alrabiah and Venkatesan Guruswami. An exponential lower bound on the sub-packetization of MSR codes. In Moses Charikar and Edith Cohen, editors, *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 979–985. ACM, 2019. doi: 10.1145/3313276.3316387.

[63] K. V. Rashmi, Nihar B. Shah, and Kannan Ramchandran. A piggybacking design framework for read-and download-efficient distributed storage codes. *IEEE Transactions on Information Theory*, 63(9):5802–5820, 2017. doi: 10.1109/TIT.2017.2715043.

[64] Venkatesan Guruswami and Ankit Singh Rawat. MDS code constructions with small sub-packetization and near-optimal repair bandwidth. In *ACM-SIAM Symposium on Discrete Algorithms*, 2017.

[65] Parikshit Gopalan, Cheng Huang, Huseyin Simitci, and Sergey Yekhanin. On the locality of codeword symbols. *IEEE Transactions on Information Theory*, 58(11):6925–6934, 2012. doi: 10.1109/TIT.2012.2208937.

[66] Ankit Singh Rawat, Onur Ozan Koyluoglu, Natalia Silberstein, and Sriram Vishwanath. Optimal locally repairable and secure codes for distributed storage systems. *IEEE Transactions on Information Theory*, 60(1):212–236, 2013. doi: 10.1109/TIT.2013.2288784.

[67] Jonathan Katz and Luca Trevisan. On the efficiency of local decoding procedures for error-correcting codes. In F. Frances Yao and Eugene M. Luks, editors, *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, pages 80–86. ACM, 2000. doi: 10.1145/335305.335315.

[68] Mario Blaum, James Lee Hafner, and Steven Hetzler. Partial-MDS codes and their application to RAID type of architectures. *IEEE Transactions on Information Theory*, 59(7):4510–4519, 2013. doi: 10.1109/TIT.2013.2252395.

[69] Parikshit Gopalan, Cheng Huang, Bob Jenkins, and Sergey Yekhanin. Explicit maximally recoverable codes with locality. *IEEE Transactions on Information Theory*, 60(9):5245–5256, 2014. doi: 10.1109/TIT.2014.2332338.

[70] Dimitris S. Papailiopoulos and Alexandros G. Dimakis. Locally repairable codes. *IEEE Transactions on Information Theory*, 60(10):5843–5855, 2014. doi: 10.1109/TIT.2014.2325570.

[71] Itzhak Tamo and Alexander Barg. A family of optimal locally recoverable codes. *IEEE Transactions on Information Theory*, 60(8):4661–4676, 2014. doi: 10.1109/TIT.2014.2321280.

[72] Govinda M. Kamath, N. Prakash, V. Lalitha, and P. Vijay Kumar. Codes with local regeneration and erasure correction. *IEEE Transactions on Information Theory*, 60(8):4637–4660, 2014. doi: 10.1109/TIT.2014.2329872.

[73] Viveck R. Cadambe and Arya Mazumdar. Bounds on the size of locally recoverable codes. *IEEE Transactions on Information Theory*, 61(11):5787–5794, 2015. doi: 10.1109/TIT.2015.2477406.

[74] Itzhak Tamo, Dimitris S. Papailiopoulos, and Alexandros G. Dimakis. Optimal locally repairable codes and connections to matroid theory. *IEEE Transactions on Information Theory*, 62(12):6661–6671, 2016. doi: 10.1109/TIT.2016.2555813.

[75] Itzhak Tamo, Alexander Barg, and Alexey A. Frolov. Bounds on the parameters of locally recoverable codes. *IEEE Transactions on Information Theory*, 62(6): 3070–3083, 2016. doi: 10.1109/TIT.2016.2518663.

[76] Alexander Barg, Kathryn Haymaker, Everett W. Howe, Gretchen L. Matthews, and Anthony Várilly-Alvarado. Locally recoverable codes from algebraic curves and surfaces. In Everett W. Howe, Kristin E. Lauter, and Judy L. Walker, editors, *Algebraic Geometry for Coding Theory and Cryptography*, pages 95–127, Cham, 2017. Springer International Publishing. ISBN 978-3-319-63931-4. doi: 10.1007/978-3-319-63931-4_4.

[77] S. Luna Frank-Fischer, Venkatesan Guruswami, and Mary Wootters. Locality via partially lifted codes. In Klaus Jansen, José D. P. Rolim, David Williamson, and Santosh S. Vempala, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2017, August 16-18, 2017, Berkeley, CA, USA*, volume 81(43) of *LIPIcs*, pages 1–17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. doi: 10.4230/LIPIcs.APPROX-RANDOM.2017.43.

[78] Abhishek Agarwal, Alexander Barg, Sihuang Hu, Arya Mazumdar, and Itzhak Tamo. Combinatorial alphabet-dependent bounds for locally recoverable codes. *IEEE Transactions on Information Theory*, 64(5):3481–3492, 2018. doi: 10.1109/TIT.2018.2800042.

[79] Arya Mazumdar. Capacity of locally recoverable codes. In *IEEE Information Theory Workshop, ITW 2018, Guangzhou, China, November 25-29, 2018*, pages 1–5. IEEE, 2018. doi: 10.1109/ITW.2018.8613529.

[80] Venkatesan Guruswami, Chaoping Xing, and Chen Yuan. How long can optimal locally repairable codes be? In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[81] Sivakanth Gopi, Venkatesan Guruswami, and Sergey Yekhanin. Maximally recoverable LRCs: A field size lower bound and constructions for few heavy parities. In Timothy M. Chan, editor, *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 2154–2170. SIAM, 2019. doi: 10.1137/1.9781611975482.130.

[82] Guangyan Zhang, Weimin Zheng, and Jiwu Shu. ALV: A new data redistribution approach to RAID-5 scaling. *IEEE Transactions on Computers*, 59(3):345–357, 2010. doi: 10.1109/TC.2009.150.

[83] Weimin Zheng and Guangyan Zhang. Fastscale: accelerate RAID scaling by minimizing data migration. In Gregory R. Ganger and John Wilkes, editors, *9th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 15-17, 2011*, pages 149–161. USENIX, 2011. URL http://www.usenix.org/events/fast11/tech/techAbstracts.html#Zheng.

[84] Chentao Wu and Xubin He. GSR: A global stripe-based redistribution approach to accelerate RAID-5 scaling. In *41st International Conference on Parallel Processing, ICPP 2012, Pittsburgh, PA, USA, September 10-13, 2012*, pages 460–469. IEEE Computer Society, 2012. doi: 10.1109/ICPP.2012.32.

[85] Guangyan Zhang, Weimin Zheng, and Keqin Li. Rethinking RAID-5 data layout for better scalability. *IEEE Transactions on Computers*, 63(11):2816–2828, 2014. doi: 10.1109/TC.2013.143.

[86] Si Wu, Yinlong Xu, Yongkun Li, and Zhijia Yang. I/O-efficient scaling schemes for distributed storage systems with CRS codes. *IEEE Transactions on Parallel*

and Distributed Systems, 27(9):2639–2652, 2016. doi: 10.1109/TPDS.2015.2505722.

[87] Xiaoyang Zhang, Yuchong Hu, Patrick P. C. Lee, and Pan Zhou. Toward optimal storage scaling via network coding: from theory to practice. In *2018 IEEE Conference on Computer Communications, INFOCOM 2018, Honolulu, HI, USA, April 16-19, 2018*, pages 1808–1816. IEEE, 2018. doi: 10.1109/INFOCOM.2018.8485961.

[88] Xiaoyang Zhang and Yuchong Hu. Efficient storage scaling for MBR and MSR codes. *IEEE Access*, 8:78992–79002, 2020. doi: 10.1109/ACCESS.2020.2989822.

[89] Brijesh Kumar Rai, Vommi Dhoorjati, Lokesh Saini, and Amit K. Jha. On adaptive distributed storage systems. In *IEEE International Symposium on Information Theory, ISIT 2015, Hong Kong, China, June 14-19, 2015*, pages 1482–1486. IEEE, 2015. doi: 10.1109/ISIT.2015.7282702.

[90] Si Wu, Zhirong Shen, and Patrick P. C. Lee. On the optimal repair-scaling trade-off in locally repairable codes. In *2020 IEEE Conference on Computer Communications, INFOCOM 2020, Virtual Conference, July 6-9, 2020*. IEEE, 2020.

[91] K. V. Rashmi, Nihar B. Shah, and P. Vijay Kumar. Enabling node repair in any erasure code for distributed storage. In Alexander Kuleshov, Vladimir M. Blinovsky, and Anthony Ephremides, editors, *2011 IEEE International Symposium on Information Theory Proceedings, ISIT 2011, St. Petersburg, Russia, July 31 - August 5, 2011*, pages 1235–1239. IEEE, 2011. doi: 10.1109/ISIT.2011.6033732.

[92] Sara Mousavi, Tianli Zhou, and Chao Tian. Delayed parity generation in MDS storage codes. In *2018 IEEE International Symposium on Information Theory, ISIT 2018, Vail, CO, USA, June 17-22, 2018*, pages 1889–1893. IEEE, 2018. doi: 10.1109/ISIT.2018.8437700.

[93] Mingyuan Xia, Mohit Saxena, Mario Blaum, and David Pease. A tale of two erasure codes in HDFS. In Jiri Schindler and Erez Zadok, editors, *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015*, pages 213–226. USENIX Association, 2015. URL https://www.usenix.org/conference/fast15/technical-sessions/presentation/xia.

[94] Xian Su, Xiaomei Zhong, Xiaodi Fan, and Jun Li. Local re-encoding for coded matrix multiplication. In *IEEE International Symposium on Information Theory, ISIT 2020, Los Angeles, California, USA, June 21-26, 2020*, 2020.

[95] Si Wu, Zhirong Shen, and Patrick P. C. Lee. Enabling I/O-efficient redundancy transitioning in erasure-coded KV stores via elastic Reed-Solomon codes. In *39th Symposium on Reliable Distributed Systems, SRDS 2020, Shanghai, China, September 21-24, 2020*, 2020.

[96] Francisco Maturana and K. V. Rashmi. Convertible codes: new class of codes for efficient conversion of coded data in distributed storage. In Thomas Vidick, editor, *11th Innovations in Theoretical Computer Science Conference, ITCS 2020, January 12-14, 2020, Seattle, Washington, USA*, volume 151 of *LIPIcs*, pages 66:1–66:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi: 10.4230/LIPIcs.ITCS.2020.66.

[97] Florence Jessie MacWilliams and Neil James Alexander Sloane. *The theory of error-correcting codes*, volume 16. Elsevier, 1977.

[98] Heide Gluesing-Luerssen, Joachim Rosenthal, and Roxana Smarandache. Strongly-MDS convolutional codes. *IEEE Transactions on Information Theory*, 52(2):584–598, 2006. doi: 10.1109/TIT.2005.862100.

[99] Francisco Maturana, V. S. Chaitanya Mukka, and K. V. Rashmi. Access-optimal linear MDS convertible codes for all parameters. In *IEEE International Symposium on Information Theory, ISIT 2020, Los Angeles, California, USA, June 21-26, 2020*, 2020.

[100] Ron M. Roth and Gadiel Seroussi. On generator matrices of MDS codes. *IEEE Transactions on Information Theory*, 31(6):826–830, 1985. doi: 10.1109/TIT.1985.1057113.

[101] Francisco Maturana and K. V. Rashmi. Bandwidth cost of code conversions in distributed storage: Fundamental limits and optimal constructions. In *2021 IEEE International Symposium on Information Theory (ISIT)*, pages 2334–2339, 2021. doi: 10.1109/ISIT45174.2021.9518121.

[102] Francisco Maturana and K. V. Rashmi. Bandwidth cost of code conversions in the split regime. In *2022 IEEE International Symposium on Information Theory (ISIT)*, pages 3262–3267, 2022. doi: 10.1109/ISIT50566.2022.9834604.

[103] Rudolf Ahlswede, Ning Cai, Shuo-Yen Robert Li, and Raymond W. Yeung. Network information flow. *IEEE Transactions on Information Theory*, 46(4):1204–1216, 2000. doi: 10.1109/18.850663.

[104] Shuo-Yen Robert Li, Raymond W. Yeung, and Ning Cai. Linear network coding. *IEEE Transactions on Information Theory*, 49(2):371–381, 2003. doi: 10.1109/TIT.2002.807285.

[105] Ralf Koetter and Muriel Médard. An algebraic approach to network coding. *IEEE/ACM Transactions on Networking*, 11(5):782–795, 2003. doi: 10.1109/TNET.2003.818197.

[106] Tracey Ho, Muriel Médard, Ralf Koetter, David R. Karger, Michelle Effros, Jun Shi, and Ben Leong. A random linear network coding approach to multicast. *IEEE Transactions on Information Theory*, 52(10):4413–4430, 2006. doi: 10.1109/TIT.2006.881746.

[107] Peter Sanders, Sebastian Egner, and Ludo M. G. M. Tolhuizen. Polynomial time algorithms for network information flow. In Arnold L. Rosenberg and Friedhelm Meyer auf der Heide, editors, *SPAA 2003: Proceedings of the Fifteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, June*

7-9, 2003, San Diego, California, USA (part of FCRC 2003), pages 286–294. ACM, 2003. doi: 10.1145/777412.777464.

[108] Sidharth Jaggi, Peter Sanders, Philip A. Chou, Michelle Effros, Sebastian Egner, Kamal Jain, and Ludo M. G. M. Tolhuizen. Polynomial time algorithms for multicast network code construction. *IEEE Transactions on Information Theory*, 51(6):1973–1982, 2005. doi: 10.1109/TIT.2005.847712.

[109] Raymond W. Yeung. *A First Course in Information Theory*. Springer US, Boston, MA, 2002. ISBN 978-1-4419-8608-5. doi: 10.1007/978-1-4419-8608-5_15.

[110] K. V. Rashmi, Nihar B. Shah, and Kannan Ramchandran. A piggybacking design framework for read-and download-efficient distributed storage codes. In *2013 IEEE International Symposium on Information Theory, ISIT 2013, Istanbul, Turkey, July 7-12, 2013*, pages 331–335. IEEE, 2013. doi: 10.1109/ISIT.2013.6620242.

[111] Francisco Maturana and K. V. Rashmi. Locally repairable convertible codes: Erasure codes for efficient repair and conversion. In *2023 IEEE International Symposium on Information Theory, ISIT 2023, Taipei, Taiwan, June 25-30, 2023*, 2023.

[112] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris S. Papailiopoulos, Alexandros G. Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. XORing elephants: novel erasure codes for big data. *Proceedings of the VLDB Endowment*, 6(5):325–336, 2013. doi: 10.14778/2535573.2488339. URL http://www.vldb.org/pvldb/vol6/p325-sathiamoorthy.pdf.

[113] Cheng Huang, Minghua Chen, and Jin Li. Pyramid codes: flexible schemes to trade space for access efficiency in reliable data storage systems. *ACM Transactions on Storage*, 9(1):3:1–3:28, March 2013. doi: 10.1145/2435204.2435207.

[114] N. Prakash, Govinda M. Kamath, V. Lalitha, and P. Vijay Kumar. Optimal linear codes with a local-error-correction property. In *Proceedings of the 2012 IEEE International Symposium on Information Theory, ISIT 2012, Cambridge, MA, USA, July 1-6, 2012*, pages 2776–2780. IEEE, 2012. doi: 10.1109/ISIT.2012.6284028.

[115] Junsheng Han and Luis Alfonso Lastras-Montaño. Reliable memories with subline accesses. In *IEEE International Symposium on Information Theory, ISIT 2007, Nice, France, June 24-29, 2007*, pages 2531–2535. IEEE, 2007. doi: 10.1109/ISIT.2007.4557599.

[116] Natalia Silberstein, Ankit Singh Rawat, O. Ozan Koyluoglu, and Sriram Vishwanath. Optimal locally repairable codes via rank-metric codes. In *2013 IEEE International Symposium on Information Theory*, pages 1819–1823, Istanbul, Turkey, 2013. IEEE. ISBN 978-1-4799-0446-4. doi: 10.1109/ISIT.2013.6620541.

[117] Yuchong Hu, Liangfeng Cheng, Qiaori Yao, Patrick P. C. Lee, Weichun Wang, and Wei Chen. Exploiting combined locality for wide-stripe erasure coding in distributed storage. In Marcos K. Aguilera and Gala Yadgar, editors, *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 233–248. USENIX Association, 2021. URL https://www.usenix.org/conference/fast21/presentation/hu.

[118] Si Wu, Zhirong Shen, Patrick P. C. Lee, and Yinlong Xu. Optimal repair-scaling trade-off in locally repairable codes: analysis and evaluation. *IEEE Transactions on Parallel and Distributed Systems*, 33:56–69, 2022. ISSN 2161-9883. doi: 10.1109/TPDS.2021.3087352.

[119] Si Wu, Qingpeng Du, Patrick P. C. Lee, Yongkun Li, and Yinlong Xu. Optimal data placement for stripe merging in locally repairable codes. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*, pages 1669–1678, London, United Kingdom, 2022. IEEE. ISBN 978-1-6654-5823-8. doi: 10.1109/INFOCOM48880.2022.9796704.

[120] Saurabh Kadekodi, Francisco Maturana, Sanjith Athlur, Arif Merchant, K. V. Rashmi, and Gregory R. Ganger. Tiger: Disk-Adaptive redundancy without placement restrictions. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 413–429, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/kadekodi.

[121] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. In Remzi H. Arpaci-Dusseau and Brad Chen, editors, *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 61–74. USENIX Association, 2010. URL http://www.usenix.org/events/osdi10/tech/full_papers/Ford.pdf.

[122] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, et al. The hadoop distributed file system. In *IEEE/NASA Goddard Conference on Mass Storage Systems and Technologies (MSST)*, 2010.

[123] Hakim Weatherspoon and John Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In Peter Druschel, M. Frans Kaashoek, and Antony I. T. Rowstron, editors, *Peer-to-Peer Systems, First International Workshop, IPTPS 2002, Cambridge, MA, USA, March 7-8, 2002, Revised Papers*, volume 2429 of *Lecture Notes in Computer Science*, pages 328–338. Springer, 2002. doi: 10.1007/3-540-45748-8_{3}{1}. URL https://doi.org/10.1007/3-540-45748-8_31.

[124] Zhe Zhang, Amey Deshpande, Xiaosong Ma, Eno Thereska, and Dushyanth Narayanan. Does erasure coding have a role to play in my data center. *Microsoft research MSR-TR-2010*, 52, 2010.

[125] K V Rashmi, Nihar B Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A hitchhiker's guide to fast and efficient data

reconstruction in erasure-coded data centers. *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2014.

[126] Ao Ma, Rachel Traylor, Fred Douglis, Mark Chamness, Guanlin Lu, Darren Sawyer, Surendar Chandra, and Windsor Hsu. RAIDShield: characterizing, monitoring, and proactively protecting against disk failures. *ACM Transactions on Storage (TOS)*, 2015.

[127] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure Trends in a Large Disk Drive Population. In *USENIX File and Storage Technologies (FAST)*, 2007.

[128] Backblaze. Disk Reliability Dataset. https://www.backblaze.com/b2/hard-drive-test-data.html, 2013-2018.

[129] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[130] Eric Brewer. Spinning Disks and Their Cloudy Future. https://www.usenix.org/node/194391, 2018.

[131] Eric Brewer, Lawrence Ying, Lawrence Greenfield, Robert Cypher, and Theodore T'so. Disks for data centers. Technical report, Google, 2016.

[132] Seagate. The Digitization of the World From Edge to Core. https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf, 2018.

[133] Garth A. Gibson. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*. PhD thesis, EECS Department, University of California, Berkeley, Dec 1990. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/1990/6373.html.

[134] Bianca Schroeder and Garth A Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *USENIX File and Storage Technologies (FAST)*, 2007.

[135] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash Reliability in Production: The Expected and the Unexpected. In *USENIX File and Storage Technologies (FAST)*, 2016.

[136] Larry Lancaster and Alan Rowe. Measuring real-world data availability. In *15th Systems Administration Conference (LISA 2001)*, 2001.

[137] Lakshmi N Bairavasundaram, Garth R Goodson, Shankar Pasupathy, and Jiri Schindler. An analysis of latent sector errors in disk drives. In *ACM SIGMETRICS Performance Evaluation Review*, 2007.

[138] Bianca Schroeder, Sotirios Damouras, and Phillipa Gill. Understanding latent sector errors and how to protect against them. *ACM Transactions on Storage (TOS)*, 2010.

[139] Alina Oprea and Ari Juels. A Clean-Slate Look at Disk Scrubbing. In *USENIX File and Storage Technologies (FAST)*, 2010.

[140] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. Kernel smoothing methods. In *The elements of statistical learning*. Springer, 2009.

[141] Apache Software Foundation. HDFS Erasure Coding. https://hadoop.apache.org/docs/r3.0.0/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html, 2017 (accessed November 5, 2020).

[142] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[143] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, et al. f4: Facebook's warm BLOB storage system. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[144] Garth Gibson, Gary Grider, Andree Jacobson, and Wyatt Lloyd. Probe: A thousand-node experimental cluster for computer systems research. *USENIX; login*, 2013.

[145] Rong Gu, Qianhao Dong, Haoyuan Li, Joseph Gonzalez, Zhao Zhang, Shuai Wang, Yihua Huang, Scott Shenker, Ion Stoica, and Patrick PC Lee. DFS-PERF: A scalable and unified benchmarking framework for distributed file systems. *EECS Dept., Univ. California, Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2016-133*, 2016.

[146] Kishor Trivedi. *Probability and Statistics with Reliability, Queueing, and Computer Science Applications*. Wiley, 2001.

[147] Martin A. Tanner and Wing Hung Wong. The estimation of the hazard function from randomly censored data by the kernel method. *Annals of Statistics*, 1983.

[148] Saurabh Kadekodi. *DISK-ADAPTIVE REDUNDANCY: tailoring data redundancy to disk-reliability heterogeneity in cluster storage systems*. PhD thesis, Carnegie Mellon University, 2020.

[149] Backblaze. Erasure coding used by Backblaze. https://www.backblaze.com/blog/reed-solomon/, 2013-2018.

[150] Nosayba El-Sayed, Ioan A Stefanovici, George Amvrosiadis, Andy A Hwang, and Bianca Schroeder. Temperature management in data centers: Why some (might) like it hot. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 163–174, 2012.

[151] Farzaneh Mahdisoltani, Ioan Stefanovici, and Bianca Schroeder. Proactive error prediction to improve storage system reliability. In *USENIX Annual Technical Conference (ATC)*, 2017.

[152] Wolfram. Wolfram Mathematica. https://www.wolfram.com/mathematica, 2023. Accessed on: 2023-06-30.

[153] John E Angus. On computing MTBF for a k-out-of-n: G repairable system. *IEEE Transactions on Reliability*, 37(3):312–313, 1988.

[154] Manuel Fernández and Stuart Williams. Closed-form expression for the Poisson-binomial probability density function. *IEEE Transactions on Aerospace and Electronic Systems*, 46(2):803–817, 2010.

[155] Yili Hong. On computing the distribution function for the Poisson binomial distribution. *Computational Statistics & Data Analysis*, 59:41–51, 2013.

[156] Werner Ehm. Binomial approximation to the Poisson binomial distribution. *Statistics & Probability Letters*, 11(1):7–16, 1991.

[157] Charles Truong, Laurent Oudre, and Nicolas Vayatis. A review of change point detection methods. In *arXiv:1801.00718v1 [cs.CE]*, 2018.

[158] Charles Truong, Laurent Oudre, and Nicolas Vayatis. ruptures: change point detection in python. In *arXiv:1801.00826v1 [cs.CE]*, 2018.

[159] Jon G Elerath. AFR: problems of definition, calculation and measurement in a commercial environment. In *IEEE Reliability and Maintenance Symposium (RAMS)*, 2000.

[160] Jon G Elerath. Specifying reliability in the disk drive industry: No more MTBF's. In *IEEE Reliability and Maintenance Symposium (RAMS)*, 2000.

[161] J. Yang and Feng-Bin Sun. A comprehensive review of hard-disk drive reliability. In *Annual Reliability and Maintainability Symposium. 1999 Proceedings (Cat.*

No.99CH36283), pages 403–409, January 1999. doi: 10.1109/RAMS.1999.744151.

[162] Jon Elerath. Hard-disk drives: The good, the bad, and the ugly. *Communication of ACM*, 2009.

[163] Eric Heien, Derrick Kondo, Ana Gainaru, Dan LaPine, Bill Kramer, and Franck Cappello. Modeling and tolerating heterogeneous failures in large parallel systems. In *ACM / IEEE High Performance Computing Networking, Storage and Analysis (SC)*, 2011.

[164] Weihang Jiang, Chongfeng Hu, Yuanyuan Zhou, and Arkady Kanevsky. Are disks the dominant contributor for storage failures?: A comprehensive study of storage subsystem failure characteristics. *ACM Transactions on Storage (TOS)*, 2008.

[165] Bianca Schroeder and Garth A Gibson. Understanding failures in petascale computers. In *Journal of Physics: Conference Series*. IOP Publishing, 2007.

[166] Sandeep Shah and Jon G Elerath. Disk drive vintage and its effect on reliability. In *IEEE Reliability and Maintenance Symposium (RAMS)*, 2004.

[167] Greg Hamerly, Charles Elkan, et al. Bayesian approaches to failure prediction for disk drives. In *International Conference on Machine Learning (ICML)*, 2001.

[168] Joseph F Murray, Gordon F Hughes, and Kenneth Kreutz-Delgado. Hard drive failure prediction using non-parametric statistical methods. In *Springer Artificial Neural Networks and Neural Information Processing (ICANN/CONIP*, 2003.

[169] Brian D Strom, SungChang Lee, George W Tyndall, and Andrei Khurshudov. Hard disk drive reliability modeling and failure prediction. *IEEE Transactions on Magnetics*, 2007.

[170] Yu Wang, Eden WM Ma, Tommy WS Chow, and Kwok-Leung Tsui. A two-step parametric method for failure prediction in hard disk drives. *IEEE Transactions on industrial informatics*, 2014.

[171] Ying Zhao, Xiang Liu, Siqing Gan, and Weimin Zheng. Predicting disk failures with HMM-and HSMM-based approaches. In *Springer Industrial Conference on Data Mining (ICDM)*, 2010.

[172] Preethi Anantharaman, Mu Qiao, and Divyesh Jadav. Large Scale Predictive Analytics for Hard Disk Remaining Useful Life Estimation. In *IEEE International Conference on Big Data*, 2018.

[173] erasure code ceph documentation. Erasure code Ceph Documentation. https://docs.ceph.com/docs/master/rados/operations/erasure-code/, (accessed September 25, 2019).

[174] Eno Thereska, Michael Abd-El-Malek, Jay J Wylie, Dushyanth Narayanan, and Gregory R Ganger. Informed data distribution selection in a self-predicting storage system. In *IEEE International Conference on Autonomic Computing (ICAC)*, 2006.

[175] Francisco Maturana and K. V. Rashmi. Bandwidth cost of code conversions in distributed storage: Fundamental limits and optimal constructions. *arXiv preprint arXiv:2008.12707*, 2020.

[176] Francisco Maturana and K. V. Rashmi. Irregular array codes with arbitrary access sets for geo-distributed storage. In *2021 IEEE International Symposium on Information Theory (ISIT)*, pages 3002–3007, 2021. doi: 10.1109/ISIT45174.2021.9517809.

[177] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan,

Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems*, 31(3):8:1–8:22, 2013.

[178] Muralidhar Subramanian, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Sivakumar Viswanathan, Linpeng Tang, and Sanjeev Kumar. f4: Facebook's warm BLOB storage system. In Jason Flinn and Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 383–398. USENIX Association, 2014. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/muralidhar.

[179] Henry C. H. Chen, Yuchong Hu, Patrick P. C. Lee, and Yang Tang. NCCloud: a network-coding-based storage system in a cloud-of-clouds. *IEEE Transactions on Computers*, 63(1):31–44, 2014. doi: 10.1109/TC.2013.167.

[180] Yu Lin Chen, Shuai Mu, Jinyang Li, Cheng Huang, Jin Li, Aaron Ogus, and Douglas Phillips. Giza: erasure coding objects across global data centers. In Dilma Da Silva and Bryan Ford, editors, *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, pages 539–551. USENIX Association, 2017. URL https://www.usenix.org/conference/atc17/technical-sessions/presentation/chen-yu-lin.

[181] Moni Naor and Ron M. Roth. Optimal file sharing in distributed networks. *SIAM Journal on Computing (SICOMP)*, 24(1):158–183, 1995. doi: 10.1137/S0097539792237462.

[182] Philippe Béguin and Antonella Cresti. General information dispersal algorithms. *Theoretical Computer Science*, 209(1-2):87–105, 1998. doi: 10.1016/S0304-3975(97)00098-4.

[183] Microsoft. Bandwidth pricing details. [Online] https://web.archive.org/web/20210503232035/https://azure.microsoft.com/en-us/pricing/details/bandwidth/, 2021. Accessed: 2021-05-03.

[184] Microsoft. Azure storage overview pricing. [Online] https://web.archive.org/web/20210503231640/https://azure.microsoft.com/en-us/pricing/details/storage/, 2021. Accessed: 2021-05-03.

[185] A. De Santis and B. Masucci. On information dispersal algorithms. In *Proceedings IEEE International Symposium on Information Theory,*, page 410, Lausanne, Switzerland, 2002. IEEE. ISBN 0-7803-7501-7. doi: 10.1109/ISIT.2002.1023682.

[186] Amos Beimel. Secret-sharing schemes: A survey. In *Coding and Cryptology - Third International Workshop, IWCC 2011, Qingdao, China, May 30-June 3, 2011. Proceedings*, volume 6639 of *Lecture Notes in Computer Science*, pages 11–46. Springer, 2011. doi: 10.1007/978-3-642-20901-7_2.

[187] Mira Gonen, Ishay Haviv, Michael Langberg, and Alex Sprintson. Minimizing the alphabet size of erasure codes with restricted decoding sets. In *IEEE International Symposium on Information Theory, ISIT 2020, Los Angeles, CA, USA, June 21-26, 2020*, pages 144–149. IEEE, 2020. doi: 10.1109/ISIT44484.2020.9174012.

[188] Cheng Chen, Sian-Jheng Lin, and Nenghai Yu. Irregular MDS array codes with fewer parity symbols. *IEEE Communications Letters*, 23(11):1909–1912, 2019. doi: 10.1109/LCOMM.2019.2937778.

[189] Filippo Tosato and Magnus Sandell. Irregular MDS array codes. *IEEE Transactions on Information Theory*, 60(9):5304–5314, 2014. doi: 10.1109/TIT.2014.2336656.

[190] Zhengrui Li and Sian-Jheng Lin. Update bandwidth for distributed storage. In *IEEE International Symposium on Information Theory, ISIT 2019, Paris, France, July 7-12, 2019*, pages 1577–1581. IEEE, 2019. doi: 10.1109/ISIT.2019.8849225.

[191] Mario Blaum, Jehoshua Bruck, and Alexander Vardy. MDS array codes with independent parity symbols. *IEEE Transactions on Information Theory*, 42(2): 529–542, 1996. doi: 10.1109/18.485722.

[192] Mario Blaum and Ron M. Roth. On lowest density MDS codes. *IEEE Transactions on Information Theory*, 45(1):46–59, 1999. doi: 10.1109/18.746771.

[193] Jeff Hartline, Tapas Kanungo, and James Hafner. R5X0: an efficient high distance parity-based code with optimal update complexity. *IBM Research Report*, RJ 10322(A0408-005), 01 2004.

[194] Chao Jin, Hong Jiang, Dan Feng, and Lei Tian. P-Code: a new RAID-6 code with optimal properties. In *Proceedings of the 23rd international conference on Supercomputing, 2009, Yorktown Heights, NY, USA, June 8-12, 2009*, pages 360–369. ACM, 2009. doi: 10.1145/1542275.1542326.

[195] Zhijie Huang, Hong Jiang, Ke Zhou, Chong Wang, and Yuhong Zhao. XI-Code: a family of practical lowest density MDS array codes of distance 4. *IEEE Transactions on Communications*, 64(7):2707–2718, 2016. doi: 10.1109/TCOMM.2016.2568205.

[196] Sheng Lin, Gang Wang, Douglas S. Stones, Xiaoguang Liu, and Jing Liu. T-Code: 3-erasure longest lowest-density MDS codes. *IEEE Journal on Selected Areas in Communication*, 28(2):289–296, 2010. doi: 10.1109/JSAC.2010.100218.

[197] Mingqiang Li and Jiwu Shu. On cyclic lowest density MDS array codes constructed using starters. In *IEEE International Symposium on Information Theory, ISIT 2010, June 13-18, 2010, Austin, Texas, USA, Proceedings*, pages 1315–1319. IEEE, 2010. doi: 10.1109/ISIT.2010.5513740.

[198] Yuval Cassuto and Jehoshua Bruck. Cyclic lowest density MDS array codes. *IEEE Transactions on Information Theory*, 55(4):1721–1729, 2009. doi: 10.1109/TIT.2009.2013024.

[199] Erez Louidor and Ron M. Roth. Lowest density MDS codes over extension alphabets. *IEEE Transactions on Information Theory*, 52(7):3186–3197, 2006. doi: 10.1109/TIT.2006.876235.

[200] Zhijie Huang, Hong Jiang, Ke Zhou, Yuhong Zhao, and Chong Wang. Lowest density MDS array codes of distance 3. *IEEE Communications Letters*, 19(10): 1670–1673, 2015. doi: 10.1109/LCOMM.2015.2464379.

[201] Zhijie Huang, Hong Jiang, and Nong Xiao. Efficient lowest density MDS array codes of column distance 4. In *IEEE International Symposium on Information Theory, ISIT 2017, Aachen, Germany, June 25-30, 2017*, pages 834–838. IEEE, 2017. doi: 10.1109/ISIT.2017.8006645.

[202] Min Ye and Alexander Barg. Explicit constructions of high-rate MDS array codes with optimal repair bandwidth. *IEEE Transactions on Information Theory*, 63(4):2001–2014, 2017. doi: 10.1109/TIT.2017.2661313.

[203] N. P. Anthapadmanabhan, E. Soljanin, and S. Vishwanath. Update-efficient codes for erasure correction. In *2010 48th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 376–382, 2010. doi: 10.1109/ALLERTON.2010.5706931.

[204] Arya Mazumdar, Gregory W. Wornell, and Venkat Chandar. Update efficient codes for error correction. In *Proceedings of the 2012 IEEE International Symposium on Information Theory, ISIT 2012, Cambridge, MA, USA, July 1-6, 2012*, pages 1558–1562. IEEE, 2012. doi: 10.1109/ISIT.2012.6283534.

[205] Arya Mazumdar, Venkat Chandar, and Gregory W. Wornell. Update-efficiency and local repairability limits for capacity approaching codes. *IEEE Journal on Selected Areas in Communications*, 32(5):976–988, 2014. doi: 10.1109/JSAC.2014.140517.

[206] Zhiying Wang and Viveck R. Cadambe. Multi-version coding - an information-

theoretic perspective of consistent distributed storage. *IEEE Transactions on Information Theory*, 64(6):4540–4561, 2018. doi: 10.1109/TIT.2017.2725273.

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

[208] Reyna Hulett and Mary Wootters. Limitations of piggybacking codes with low substriping. In *2017 55th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 1131–1138. IEEE, 2017.

[209] Stasys Jukna. *Extremal Combinatorics - With Applications in Computer Science*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2011. ISBN 978-3-642-17363-9. doi: 10.1007/978-3-642-17364-6.

[210] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for low-latency geo-replicated storage. In Nick Feamster and Jeffrey C. Mogul, editors, *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 313–328. USENIX Association, 2013. URL https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/lloyd.

[211] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. SPANStore: cost-effective geo-replicated storage spanning multiple cloud services. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 292–308. ACM, 2013. doi: 10.1145/2517349.2522730.

[212] Leslie Lamport. Fast paxos. *Distributed Comput.*, 19(2):79–103, 2006. doi: 10.1007/s00446-006-0005-x.

[213] Leslie Lamport and Mike Massa. Cheap paxos. In *2004 International Conference on Dependable Systems and Networks (DSN 2004), 28 June - 1 July 2004, Florence, Italy, Proceedings*, pages 307–314. IEEE Computer Society, 2004. doi: 10.1109/DSN.2004.1311900.

[214] Butler W. Lampson. The abcd's of paxos. In Ajay D. Kshemkalyani and Nir Shavit, editors, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing, PODC 2001, Newport, Rhode Island, USA, August 26-29, 2001*, page 13. ACM, 2001. doi: 10.1145/383962.383969.

[215] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 358–372. ACM, 2013. doi: 10.1145/2517349.2517350.

[216] Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. Dpaxos: Managing data closer to users for low-latency and mobile applications. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1221–1236. ACM, 2018. doi: 10.1145/3183713.3196928.

[217] Leslie Lamport. Paxos made simple, fast, and byzantine. In Alain Bui and Hacène Fouchal, editors, *Procedings of the 6th International Conference on Principles of Distributed Systems. OPODIS 2002, Reims, France, December 11-13, 2002*, volume 3 of *Studia Informatica Universalis*, pages 7–9. Suger, Saint-Denis, rue Catulienne, France, 2002.

[218] Robbert van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Comput. Surv.*, 47(3):42:1–42:36, 2015. doi: 10.1145/2673577.

[219] Shuai Mu, Kang Chen, Yongwei Wu, and Weimin Zheng. When paxos meets erasure code: reduce network and storage cost in state machine replication. In

Beth Plale, Matei Ripeanu, Franck Cappello, and Dongyan Xu, editors, *The 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'14, Vancouver, BC, Canada - June 23 - 27, 2014*, pages 61–72. ACM, 2014. doi: 10.1145/2600212.2600218.

[220] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible paxos: Quorum intersection revisited. In Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone, editors, *20th International Conference on Principles of Distributed Systems, OPODIS 2016, December 13-16, 2016, Madrid, Spain*, volume 70 of *LIPIcs*, pages 25:1–25:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi: 10.4230/LIPIcs.OPODIS.2016.25.

[221] Nicolas C. Nicolaou, Viveck R. Cadambe, N. Prakash, Kishori M. Konwar, Muriel Médard, and Nancy A. Lynch. ARES: Adaptive, Reconfigurable, Erasure coded, atomic Storage. In *39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019, Dallas, TX, USA, July 7-10, 2019*, pages 2195–2205. IEEE, 2019. doi: 10.1109/ICDCS.2019.00216.

[222] Fan Lai, Mosharaf Chowdhury, and Harsha V. Madhyastha. To relay or not to relay for inter-cloud transfers? In *USENIX HotCloud*, 2018.

[223] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley, 2001. ISBN 9780471062592. doi: 10.1002/0471200611.

[224] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. WPaxos: Wide area network flexible consensus. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):211–223, 2019.

[225] Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. State-machine replication for planet-scale systems. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020.

[226] Akhil Kumar. Hierarchical quorum consensus: A new algorithm for managing replicated data. *IEEE Trans. Computers*, 40(9):996–1004, 1991. doi: 10.1109/12.83661.

[227] IBM. IBM ILOG CPLEX optimizer. https://www.ibm.com/products/ilog-cplex-optimization-studio/cplex-optimizer, 2023. Accessed on: 2023-06-30.

[228] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. Ironfleet: proving safety and liveness of practical distributed systems. *Commun. ACM*, 60(7): 83–92, 2017. doi: 10.1145/3068608.

[229] Google AI. OR-tools: Google Optimization Tools. https://github.com/google/or-tools, 2023. Accessed: 2023-06-30.

[230] Intel. Intel(R) Intelligent Storage Acceleration Library. https://github.com/intel/isa-l, 2023. Accessed: 2023-06-30.

[231] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In Ted Wobber and Peter Druschel, editors, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 401–416. ACM, 2011. doi: 10.1145/2043556.2043593.

[232] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno M. Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In Chandu Thekkath and Amin Vahdat, editors, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 265–278. USENIX Association, 2012. URL https://www.usenix.org/conference/osdi12/technical-sessions/presentation/li.

[233] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 309–324. ACM, 2013. doi: 10.1145/2517349.2522731.

[234] Jason Baker, Chris Bond, James C. Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Fifth Biennial Conference on Innovative Data Systems Research, CIDR 2011, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 223–234. www.cidrdb.org, 2011. URL http://cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf.

[235] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In Ted Wobber and Peter Druschel, editors, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 385–400. ACM, 2011. doi: 10.1145/2043556.2043592.

[236] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 276–291. ACM, 2013. doi: 10.1145/2517349.2522729.

[237] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machine for wans. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California,*

USA, Proceedings, pages 369–384. USENIX Association, 2008. URL http://www.usenix.org/events/osdi08/tech/full_papers/mao/mao.pdf.

[238] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, and Alec Wolman. Volley: automated data placement for geo-distributed cloud services. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2010, April 28-30, 2010, San Jose, CA, USA*, pages 17–32. USENIX Association, 2010. URL http://www.usenix.org/events/nsdi10/tech/full_papers/agarwal.pdf.

[239] John Kubiatowicz, David Bindel, Yan Chen, Steven E. Czerwinski, Patrick R. Eaton, Dennis Geels, Ramakrishna Gummadi, Sean C. Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Y. Zhao. OceanStore: an architecture for global-scale persistent storage. In Larry Rudolph and Anoop Gupta, editors, *ASPLOS-IX Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, USA, November 12-15, 2000*, pages 190–201. ACM Press, 2000. doi: 10.1145/356989.357007.

[240] Andreas Haeberlen, Alan Mislove, and Peter Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In Amin Vahdat and David Wetherall, editors, *2nd Symposium on Networked Systems Design and Implementation (NSDI 2005), May 2-4, 2005, Boston, Massachusetts, USA, Proceedings*. USENIX, 2005. URL http://www.usenix.org/events/nsdi05/tech/haeberlen.html.

[241] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 401–417. USENIX Association, 2016. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/rashmi.