

# 硕士学位论文

## (专业学位)

纠删码存储系统中的数据修复技术研究

Research on Data Repair Techniques in

Erasure-Coded Storage Systems

申请人学号 20195227076

专业名称 计算机技术

研究方向 分布式存储

论文关键词 分布式存储； 预先修复； 混合纠删码；

数据修复

# 纠删码存储系统中的数据修复技术研究

## 摘要

分布式存储系统作为一种可以存储海量数据并提供便捷数据访问的服务在大数据时代被广泛应用。由于系统庞大而导致数据丢失的情况时有发生，系统一般采用纠删码的方式来进行数据存储并冗余。但是纠删码存在数据修复计算开销大，网络负载高，修复速度低等问题。本文针对以上问题，对纠删码的预先修复和混合纠删码修复技术进行了深入研究，主要研究与贡献如下：

(1) 针对纠删码修复速度慢而导致系统可靠性降低问题，本文在预先重建与迁移修复的基础上，改进预先修复算法使其适应快速变化的网络环境，提出了划分迁移集和重建集算法 SMSRS，并引入多级中继传输概念提出了改进的多级传输调度算法 ISA 加速修复任务数据的传输。实验结果表明，本文提出的划分算法和调度技术较传统方法修复时间降低了 7.2%~13.2%。

(2) 针对传统单一纠删码修复技术的退化读高延迟问题，本文在传统混合纠删码 LRC&HH 码的基础上，提出了一种可感知数据热度的负载动态自适应的混合纠删码数据修复技术。该技术根据实际存储系统 I/O 特点，定义了读写负载模式和修复负载模式，通过建模分析编码分配规则和动态队列划分冷热数据算法提出负载自适应策略，并结合编码切换算法进而提升系统数据修复速度。实验结果表明，本文提出的负载自适应策略，相比于现有的 LRC&HH 码随机切换策略，修复时间降低了 6.8%~18.7%。

(3) 基于上述理论研究成果，设计实现了一个基于混合纠删码的容错存储原型系统。该系统含有多种混合纠删码方案，包括 HACFS、EC-Fusion 以及 LRC&HH，支持相应的扩展接口，支持修复调度中的重建与迁移的融合。系统包含以下模块：存储中心架构，节点故障模型，混合纠删码策略，节点放置策略，可靠性度量指标，事件处理模式。系统可接受含存储中心架构、混合纠删码策略和冗余设置等参数，并且可以定量地分析各种纠删码方案之间的可靠性量化指标。实验结果表明，基于预先修复的混合纠删码方案与传统单一纠删码相比可以提升系统可靠性，其中 PDL 参数降低了 30%~35%，NOMDL 降低了 40%~42.5%，BR 降低了 35.6%~37.8%。

**关键词：**分布式存储，预先修复，混合纠删码，数据修复

作 者：\*\*\*

指导老师：\*\*\*

# Research on Data Repair Techniques in Erasure-Coded Storage Systems

## Abstract

Distributed storage systems are widely used in the era of big data as a service that can store large amounts of data and provide convenient data access, but data loss occurs due to the large system, and the system generally uses the erasure code for data storage and redundancy. However, there are problems such as high computational overhead of data repair, high network load and low repair speed. In this paper, we address the above problems and conduct an in-depth study on the pre-repair and hybrid erasure code repair techniques for erasure codes. The main research and contributions are as follows:

(1) In order to address the problem of slow repair speed of erasure codes which leads to the reduction of system reliability, this paper improves the pre-repair algorithm to adapt to the fast changing network environment based on the pre-repair and migration repair, proposes the division of migration set and reconstruction set algorithm SMSRS, and introduces the concept of multi-stage relay transmission and proposes an improved multi-stage transmission scheduling algorithm ISA to speed up the transmission of repair task data. The experimental results show that the proposed partitioning algorithm and scheduling technique reduce the repair time by 7.2%~13.2% compared with the traditional method.

(2) To address the degraded read-high latency problem of traditional single-code repair technology, this paper proposes a load dynamic adaptive hybrid censored data repair technology that can sense data heat based on the traditional hybrid censored LRC&HH codes. The technique defines the read/write load mode and repair load mode according to the I/O characteristics of the actual storage system, proposes a load adaptive strategy by modeling and analyzing the code allocation rules and dynamic queueing algorithm to divide hot and cold data, and combines the code switching algorithm to improve the system data repair speed. The experimental results show that the proposed load adaptive strategy reduces the repair time by 6.8%~18.7% compared with the existing LRC&HH code random switching

strategy.

(3) Based on the above theoretical research results, a prototype system of fault-tolerant storage based on hybrid censoring codes is designed and implemented. The system contains multiple hybrid censoring schemes, including HACFS, EC-Fusion and LRC&HH, and supports the corresponding extension interfaces and the fusion of reconstruction and migration in repair scheduling. The system includes the following modules: storage center architecture, node failure model, hybrid code correction policy, node placement policy, reliability metrics, and event processing model. The system can accept parameters containing storage center architecture, hybrid coding strategy and redundancy settings, and can quantitatively analyze the reliability quantifiers among various coding schemes. The experimental results show that the hybrid pre-repair-based censoring scheme can improve the system reliability compared with the traditional single censoring scheme, in which the PDL parameters are reduced by 30%~35%, NOMDL by 40%~42.5%, and BR by 35.6%~37.8%.

**Key words:** Distributed Storage, Predictive repair, Mixed erasure Code, Data repair

Written by \*\*\*

Supervised by \*\*\*

# 目 录

<b>第一章 绪论</b> .....	1
1.1 分布式存储系统概述 .....	1
1.2 数据容错技术概述 .....	2
1.2.1 结构容错机制 .....	3
1.2.2 数据容错机制 .....	3
1.3 纠删码概述 .....	5
1.3.1 Reed-Solomon 码 .....	6
1.3.2 基于阵列结构的纠删码 .....	7
1.3.3 基于网络编码的纠删码 .....	9
1.3.4 基于分组结构的纠删码 .....	10
1.4 基于纠删码的数据修复技术 .....	11
1.4.1 基于星型拓扑的数据修复 .....	12
1.4.2 基于树型修复拓扑的数据修复 .....	13
1.4.3 基于 XOR 的纠删码数据修复 .....	14
1.5 论文主要研究内容 .....	14
1.6 论文组织结构 .....	16
<b>第二章 相关研究</b> .....	18
2.1 基于编码结构优化的数据修复技术 .....	18
2.2 基于调度优化的数据修复技术 .....	20
2.3 基于带宽优化的数据修复技术 .....	22
2.4 基于混合纠删码的数据修复技术 .....	26
2.5 基于预先修复的数据修复技术 .....	30
2.6 本章小结 .....	32
<b>第三章 基于预先重建迁移的数据修复技术研究</b> .....	33
3.1 引言 .....	33
3.2 预先修复问题建模 .....	34
3.2.1 问题描述 .....	34
3.2.2 模型假设 .....	35
3.2.3 理论修复时间 .....	36
3.2.4 迁移与重建时间建模 .....	36

3.3 预先修复集合划分算法 .....	38
3.4 预先修复调度算法 .....	39
3.5 实验结果与分析 .....	45
3.5.1 实验环境 .....	45
3.5.2 修复与调度实验 .....	46
3.5.3 条带数和块大小的影响 .....	48
3.5.4 空闲节点数的影响 .....	48
3.5.5 带宽变化影响 .....	49
3.6 本章小结 .....	50
<b>第四章 基于混合纠删码的数据修复技术研究 .....</b>	<b>52</b>
4.1 引言 .....	52
4.2 编码策略选择 .....	53
4.2.1 局部修复码（LRC） .....	53
4.2.2 Hitchhiker（HH）码 .....	54
4.3 负载自适应策略 .....	54
4.3.1 编码分配规则 .....	55
4.3.2 自适应选择算法 .....	59
4.4 编码切换算法 .....	61
4.5 实验结果与分析 .....	63
4.5.1 实验参数与数据集 .....	63
4.5.2 读写负载性能实验结果 .....	64
4.5.3 修复负载性能结果 .....	65
4.5.4 整体负载性能实验结果 .....	66
4.5.5 临界比实验结果 .....	67
4.6 本章小结 .....	67
<b>第五章 基于混合纠删码的容错存储原型系统设计与实现 .....</b>	<b>68</b>
5.1 系统设计 .....	68
5.1.1 存储中心架构 .....	69
5.1.2 故障模式建模 .....	70
5.1.3 纠删码混合策略 .....	71
5.1.4 纠删码接口 .....	71
5.1.5 可靠性分析指标 .....	72
5.1.6 事件处理 .....	73

5.2 实验结果与分析 .....	75
5.2.1 实验设置 .....	75
5.2.2 PDL 参数实验结果 .....	76
5.2.3 NOMDL 参数实验结果 .....	76
5.2.4 BR 参数实验结果 .....	77
5.3 本章小结 .....	78
<b>第六章 总结与展望 .....</b>	<b>79</b>
6.1 论文工作总结 .....	79
6.2 未来工作展望 .....	80
<b>参考文献 .....</b>	<b>81</b>
<b>攻读学位期间的成果 .....</b>	<b>92</b>
<b>攻读学位期间的主要科研工作 .....</b>	<b>93</b>



# 第一章 绪论

## 1.1 分布式存储系统概述

信息技术以及数据存储技术的发展引起了重大且深刻的社会变革，并推进了科技的进步与发展，而随着信息技术与数据存储技术在社会生活中的不断渗入，电子数据的产生数量与速度都呈现着爆炸式的增长。根据 IDC (International Data Corporation, 国际数据公司) 预测，全球数据量将从 2020 年的 44ZB 增长到 2025 年的 175ZB<sup>[1]</sup>。在如此急剧增长的数据体量面前，传统的数据存储方式已然无法满足时代对于数据存储的需求，所以如何保证数据的安全存储以及便捷读取成为当前时代的一个非常重要的课题。

伴随着用户电子数据的不断增长和存储需求的不断增加，分布式存储系统凭借便捷性、统一性、安全性、可靠性得到了广泛的应用，并且逐渐成为了当今信息社会的网络基础设施。分布式存储系统结合了负载均衡技术、集群技术、虚拟化技术、分布式技术、CDN 加速技术，为用户提供了数据存取的方便以及快捷且低成本的文件存储服务。

但是，目前流行的云存储系统普遍依赖中心服务器来提供云存储和数据处理服务<sup>[2]</sup>。存储节点本身硬件故障或者技术原因导致的风险等原因会导致存储在服务器上的数据丢失或者永久不可访问。从而导致存储的数据永久性的丢失。例如：2018 年腾讯云因磁盘故障导致文件元数据的丢失，大量的个人用户资料数据丢失；Sathiamoorthy 等<sup>[3]</sup> 曾指出“Facebook 数据中心的一个大集群在一个月内的节点故障数目超过 20 个而一天内失效节点的数量峰值可高达 100 个”。因此如何提高分布式存储系统的数据可靠性亦即数据修复技术是业界和学术界广泛关注的问题。对于可靠性低且经常发生故障的海量存储节点，如何采用性能优越的技术和措施才能保证数据的可靠性和故障发生时如何以相对低廉的成本快速修复故障节点的数据是目前研究的重要方向。

目前，分布式存储系统通常采取超过原本 1 倍的冗余数据来保证数据可靠性以对抗节点故障而导致的数据丢失，进而通过冗余数据进行相关的计算还原出丢失的数据。分布式存储系统中冗余数据的产生方法大致可以分为两类：

(1) 多副本方式，即将数据对象复制多份（通常为三份），如  $r$  份，然后将这些副本均匀地分散到  $r$  个不同的存储节点上，当某个节点故障时，只需要从剩下任意节

点中下载原文件  $r - 1$  次，并将其重新存放到其他正常节点即可。但是在数据存储急剧增长的情况下，备份冗余会引发大量存储开销，早期的分布式集群一般采用这种方式。其中典型的代表有 Google File System<sup>[4]</sup>，Hadoop Distributed File System<sup>[5]</sup> 等。

(2) 纠删码方式，即使用纠删码（erasure codes）技术生产冗余数据块。纠删码是通过编解码矩阵的计算创建了远小于多副本技术的冗余数据，同时提供同级别的容错性<sup>[6]</sup>。如今的大规模存储群越来越多地采用纠删码 (Ford 等<sup>[7]</sup>, Huang 等<sup>[8]</sup>, Muralidhar 等<sup>[9]</sup>, Ovsianikov 等<sup>[10]</sup>)，进而取得存储空间和可靠性二者之间的均衡。具体的编码方式则是将原始文件计算分割成若干个数据块，再进行编码生成多个冗余的编码块。与多副本方式相比，纠删码的优点在于能够极大地节省了存储空间，进而减少了存储空间上的开销。目前学术界有大量的纠删码的编码方式，如 Reed-Solomon 码（简称 RS 码）<sup>[11]</sup>，Regenerating Code（简称 RGC 码）<sup>[12]</sup>，Local Regeneration Code（简称 LRC 码）<sup>[13]</sup> 等。

无论是多副本技术还是纠删码方式都会产生冗余数据，并且都有其固有的缺陷。多副本技术消耗了更多的存储空间，消耗  $(m - 1)$  倍于原始数据大小的存储空间，例如三副本机制在大数据时代，当数据规模达到 PB、EB 甚至 ZB 级别时，多副本技术极高的存储空间成本以及维护开销将使得数据中心无法承担。除此之外，纠删码自身也存在着相应地无法忽视的缺陷，例如修复成本过高、消耗大量的 CPU 资源进行编解码操作、实现过程复杂等，从而使得将该技术应用到分布式存储系统中时面临着诸多挑战。

## 1.2 数据容错技术概述

通常采用数据容错技术来对系统的可靠性和访问性进行提升。数据容错技术是指通过对存储数据进行冗余，并将冗余数据传输到其他节点上，使得即使一部分数据失效也可以通过其他节点将数据进行恢复。

为了保证系统中数据的快速访问，在部分数据因节点故障而丢失时必须利用健康节点上的剩余数据快速地将数据重新恢复出来，此过程被称为“**数据修复 (Data Repair)**”<sup>[14]</sup>。维持数据原有的冗余度是数据修复技术想要达到的目的，同时也要确保数据的可靠性。因为对于数据容错技术而言，能够容忍的数据丢失是极其有限的，一旦丢失的数据超过了容错技术的容忍度，原本的数据对象将无法恢复，从而导致数据的永久性丢失。一般的数据修复过程主要遵循以下过程：当系统检测到节点故障事

件产生时，根据系统设定的容错方案开始进行启用对应的修复方案，利用剩余的健康数据节点对丢失的数据进行恢复，通过一定的算法将数据分散存储到存活节点中去，在这个过程中修复时提供数据的节点称为提供节点，修复出的数据存在的节点称为新生节点。分布式存储系统冗余机制可以分为结构容错机制和数据容错机制。

### 1.2.1 结构容错机制

结构容错机制实现系统容错的方式是通过提供冗余的物理设备的方法来实现的。具体的分类方式是根据冗余物理设备的不同可以分为以下两种：基于节点冗余的冗余机制和基于链路冗余的容错机制。

基于节点冗余的容错机制实现文件冗余的方式是使用增加冗余节点的方式实现的。当系统中的某个数据节点失效以后，系统将使用冗余节点替代失效节点<sup>[4,15,16]</sup>。

基于链路冗余的容错机制则是通过增加冗余的网络链路的方式来实现文件冗余。主要通过两种方式进行实现，其中之一为通过在树结构的上层链路加入冗余链路来提高系统容错性<sup>[17,18]</sup>，或者通过在服务节点上添加网卡的方式实现多链路冗余<sup>[19,20]</sup>。

### 1.2.2 数据容错机制

在传统的分布式存储系统中，用户节点通过将文件切块后再将数据块分发到存储节点中。通过创建冗余的数据编码块来对分布式存储系统的容错能力进行提高的方式被称为数据容错机制。根据冗余数据编码块创建方式的不同，数据容错机制可以分为基于复制的容错机制和基于编码的容错机制。

基于复制的容错机制原理如图 1-1 所示，将原始数据复制多份，创建多个数据副本并分发到不同的存储节点上从而实现数据冗余<sup>[4,21,22]</sup>。当一个或者多个存储着原始文件副本数据的存储节点故障时，系统通过访问其他存活节点的原始文件的副本数据，进行数据的恢复，进而提高系统的可靠性。基于复制的冗余机制原理较为简单，易于部署和实现，并且在确保数据可靠性的基础上还能提高数据的并发访问能力。然而，缺点显而易见，在文件储存过程中，需要分发多个多个数据副本，因此提高了数据的存储开销和通信数据的开销，存储效率较低。对于  $n$  副本的复制容错机制，系统存储空间的总体利用率只有  $\frac{1}{n}$ 。在此基础之上，为了提高存储效率，降低存储空间的开销，可以采用动态复制机制，依据系统的网络状况、用户需求等因素进行综合考量并进行动态创建与删除副本<sup>[23-25]</sup>。但是，动态复制技术也存在其固有的缺陷，因为

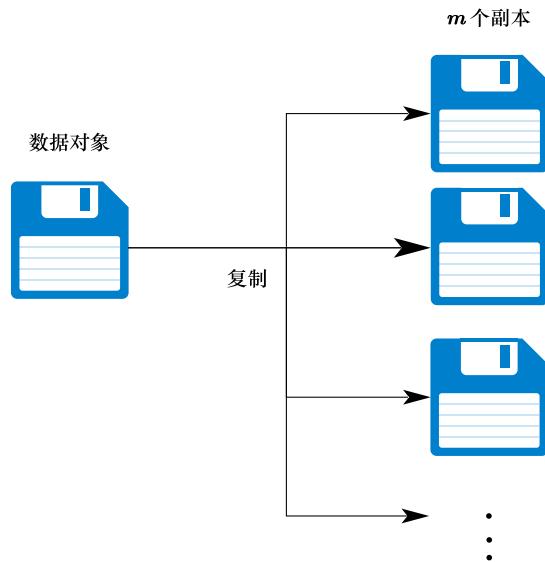


图 1-1 基于复制的容错机制原理

在动态创建与删除副本的过程中，需要 CPU 执行大量的计算，并且需要进行频繁的数据分发，从而增加了系统的计算开销和通信开销。

相对应的，基于编码的容错机制首先对原始数据文件进行切块，再对切成的块文件进行编码，生成体积较小的编码块从而提高数据可用性。基于编码的容错机制生成的编码块体积小、数量少，因此，相对于基于复制的容错机制，具有较小的存储开销。基于编码的容错机制可以分为两种：RAID 技术、纠删码容错。

RAID 技术<sup>[26,27]</sup>是一种传统的基于编码的容错技术，RAID 技术实现容错的方式是通过将数据进行条带化，然后将数据条带分发到不同的存储节点上，并生成一个编码块进而实现数据的容错与冗余。RAID 技术一般只能容忍一到两个数据块的失效，因此对于大规模的分布式存储系统而言，RAID 技术无法满足其数据可靠性的要求。

更加主流的基于编码的容错方式便是纠删码技术，其本身是一种编码容错技术，最早被应用到通信领域，解决数据在传输中的纠错问题。传统的纠删码技术实现容错机制的步骤一般如下，首先将原始数据文件分割成  $k$  个数据块，然后将这些数据块通过相应的矩阵计算进行编码生成  $n - k$  个编码块，最后将生成的  $k$  个数据块和  $n - k$  个编码块分发到不同的存储节点中。当用户访问存储系统下载数据时，只需要从  $n$  个数据块中任意下载  $k$  个可用的块即可恢复出完整的原始数据文件。

相较于基于复制的容错技术即多副本技术，纠删码技术的优势主要体现在更加低的存储成本同时提供更加稳定的数据修复方式<sup>[6,28]</sup>。例如，RS(12, 10) 码可以凭借

0.2 倍的额外存储空间开销获取两个节点的容错能力，这比具有同等容错能力的三副本技术节省了约 60% 的存储空间。在同等的存储空间利用的情况下，以纠删码技术构建的分布式系统的数据可靠性比采用多副本技术构建的分布式存储系统中的数据可靠性要大几个数量级<sup>[6,28]</sup>。尽管有很多代表性的工作 (Reed 等<sup>[11]</sup>, Huang 等<sup>[29]</sup>, Xu 等<sup>[30]</sup>, Roth 等<sup>[31]</sup>) 对于纠删码的计算性能进行了优化，但是纠删码方案由于其固有的缺陷如大量的矩阵计算编解码操作，依然会产生巨大的通信开销。因此如何提高纠删码容错的效率是一个亟需解决的重要问题。

### 1.3 纠删码概述

近年来，越来越多的大规模存储系统采用了具有较强容错能力和空间利用率较高的纠删码实现系统冗余 (Xia 等<sup>[32]</sup>, Kubiatowicz 等<sup>[33]</sup>)。随着时代发展，数据规模越来越大，如何提升纠删码的容错能力用以满足系统的存储和访问需求已然成为学界与业界广泛关注的问题。

纠删码的主要思想是把一个数据对象  $D$  分为  $k$  个相同大小的数据块，运用异或计算和矩阵计算从而生成  $n$  个块 ( $n > k$ )，使得通过其中任意  $k'$  个块即可重新还原出原始数据，具体过程如图 1-2 所示。

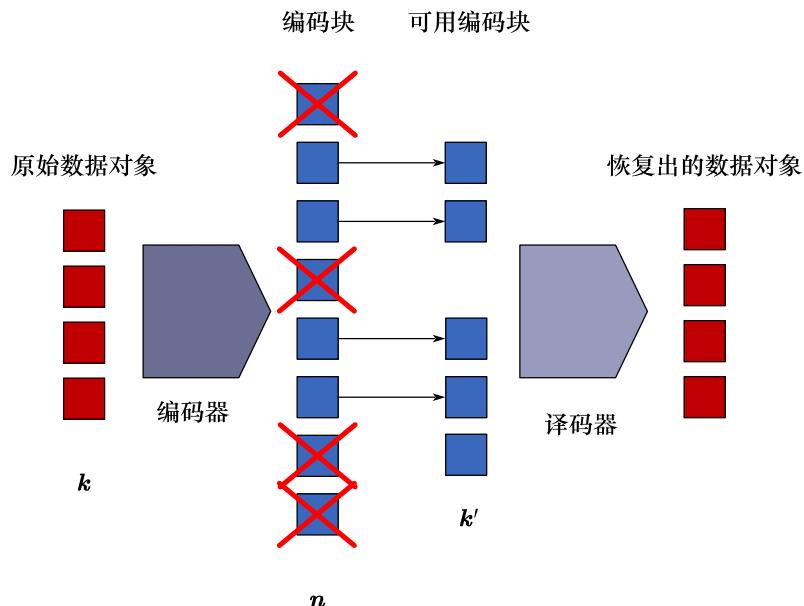


图 1-2 纠删码编码和解码示意图

一般来说，纠删码的通用表示方法为  $(n, k, k')$ 。其中  $k$  是对象切分成的块数， $k'$

是一个不小于  $k$  的数,  $n$  是编码后的块数。首先, 将  $D$  分割成  $k$  个大小 (平均) 相等的数据块  $D_1, D_2, \dots, D_k$ , 然后通过数学算法和矩阵运算将这  $k$  个数据块进行编码, 生成  $n$  个同等大小的编码块  $I_1, I_2, \dots, I_n, n > k$ , 使得通过其中任意  $k'$  个块都能通过矩阵运算还原出  $D$  的对应的  $k$  个数据块  $D_1, D_2, \dots, D_k$ , 从而组合出原始的数据。纠删码可以容忍的最多失效块的个数称为容错度, 当  $k = k'$  时, 纠删码达到理论上的最高容错力, 称之为最大距离可分码 (Maximum Distance Separable, 简称 MDS 码)<sup>[34]</sup>。这样的纠删码通常可以表示为  $(n, k)$ , 如果纠删码生成的  $n$  个块集合中包含全部的  $k$  个原数据块, 这样的纠删码便是系统码 (System Code)<sup>[35]</sup>。对于系统码而言, 编码后生成的额外  $n - k$  个块称为编码块, 并且记为  $P_1, P_2, \dots, P_m$ , 其中  $m = n - k$ , 系统码可提升系统性能, 这也是现有的纠删码大多是系统码的原因。

在存储系统实际的运行过程中, 先将文件分割成固定大小的数据块, 进而将这些块每  $k$  个作为一组, 每组独立进行编码操作生成  $n$  个块, 其中  $n$  个块的集合称为一个条带 (Stripe)<sup>[36]</sup>,  $k$  称为条带长度 (Stripe Length)。当数据对象的块数不能被  $k$  的整除时, 则对相应的数据域以 0 填充, 被 0 填充的部分只是在逻辑上存储在磁盘中, 在数据修复时也不需要读取。根据计算方式和几何构造的不同, 纠删码可分为 Reed-Solomon 码、阵列结构型纠删码、网络编码型纠删码和分组结构型纠删码。

### 1.3.1 Reed-Solomon 码

Reed-Solomon 码<sup>[11]</sup>是最为经典的 MDS 码, 是唯一可以用任意的数据磁盘数目和任意冗余磁盘数目的 MDS 码。RS 码的编码原理是通过矩阵的运算来完成的, 如果分布式存储集群采用的是结构为  $(k + r, r)$  的 RS 编码, 则整个分布式存储集群应具有  $k$  个数据块和  $r$  个编码块。在基于 RS 编码的分布式存储集群中, 如图 1-3 所示, 编码块是通过将  $k$  个数据块与  $k \times (k + r)$  个生成矩阵相乘来构造的, 该生成矩阵主要包含了两个变量, 一个  $k \times k$  校验矩阵和一个  $k \times r$  冗余矩阵<sup>[37]</sup>。一个由  $(k + r, r)$  组成的 RS 编码存储集群, 含有  $k + r$  个存储节点的阵列, 其中  $k$  个数据块和  $r$  个编码块分别存储在存储集群中的  $k$  数据节点服务器和  $r$  个奇偶校验节点服务器上。

RS 码实际上是利用生成矩阵和数据列向量进行矩阵乘法得到编码列向量。RS 码使用的生成矩阵  $G$  中的任意  $k$  行组成的矩阵需要在 Galois 域上可逆, 故获得生成矩阵的计算量并不低, 且可由范德蒙德 (Vandermonde) 矩阵或柯西 (Cauchy) 矩阵<sup>[31]</sup> 通过变换得到。通过相应的方法计算得到的 RS 码分别称为范德蒙德 RS 码和柯西 RS

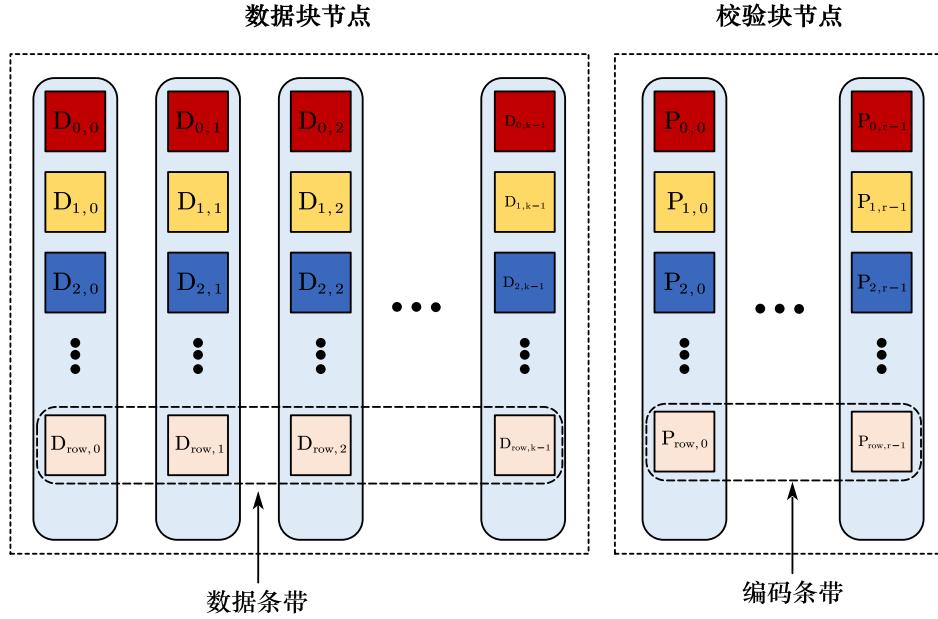


图 1-3 传统基于 RS 码结构的存储集群布局

码。在 Galois 域上，异或运算被定义为加法，使用离散对数运算和查表来进行乘法运算，因此计算开销很大，而应用柯西矩阵可将乘法运算转化为二进制的乘法运算，从而减轻运算量。

然而，纠删码的其中一个弊端在于会引起大量的带宽以及计算资源的消耗，进而导致存储效率的降低。大量的矩阵计算所产生的计算开销增加了系统的修复开销，使其在分布式存储系统中的应用有很多亟待解决的问题。与此同时，RS 码可以提供优秀的容错能力以及更加低的空间消耗，从而对提高分布式存储系统的数据可靠性和降低其经济成本具有重大意义。为了解决这一问题，涌现出了众多修复成本较低的纠删码。这些低修复成本的纠删码根据其采用的基本技术可以分为三类：基于阵列结构的纠删码、基于网络编码的纠删码和基于分组结构的纠删码。

### 1.3.2 基于阵列结构的纠删码

阵列码是一种采用阵列结构编码且完全基于异或运算的纠删码，从冗余磁盘阵列 (RAID)<sup>[26]</sup> 技术中不断发展出来。其主要思想就是将数据块和校验块存储到二维矩阵内，根据其放置方式的不同，可以分为横向阵列码和纵向阵列码。

横向阵列码 (horizontal parity array codes) 的特点是将数据块和校验块分别传输到不同的存储节点上。在这样的系统中专设有冗余磁盘来存储冗余数据，正常的数据块

存储在正常的磁盘中，从而提高阵列码的可扩展性。

EVENODD 编码<sup>[38]</sup>是最早被发表的阵列码，EVENODD 码的编码块和数据块之间的计算关系如图 1-4 所示。其中第一列上的编码块分别由各自同一行上的其他数据块进行异或得到。此外，RDP<sup>[39]</sup> 编码也是一种应用较为广泛的横向阵列码，这些编码体系均对磁盘数目提出了严格的要求（要求磁盘数目必须是素数或者素数减 1），同时还要求在单个条带中的数据元素的个数必须与磁盘的数目相匹配。

除了 EVENODD 和 RDP，其他的横向阵列码有 EEO<sup>[40]</sup>，STAR<sup>[41]</sup>，RAID-6 Liberation<sup>[35]</sup> 和 Feng's Code<sup>[42]</sup> 等，它们具有性能比较优秀的编码效率和极高的存储空间利用率。由于横向阵列码本身固有的缺陷，如写入数据总会用到冗余磁盘，故其中的 I/O 流量成为了横向阵列码的性能瓶颈。

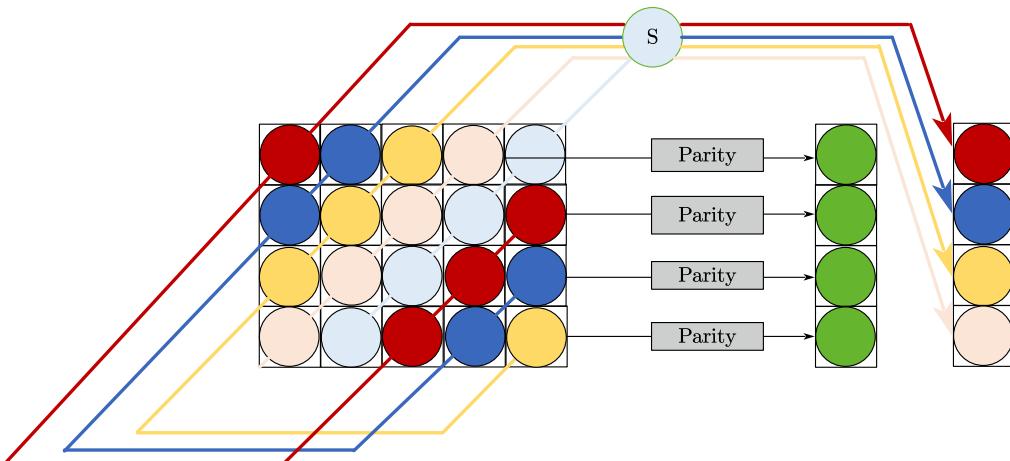


图 1-4 EVENODD 码的编码结构示意图

纵式阵列码 (vertical parity array codes) 是指冗余存储在数据条带中的阵列编码方式，将数据块和校验块不加区分地存放节点的磁盘中。这样的方式会将计算操作和写入操作平均地分摊到各个磁盘上，实现了负载均衡。X-code<sup>[30]</sup> 是具有 MDS 性质的纵式纠删码，其在编码操作，数据更新以及数据重构等步骤上具有理论上的最有效率，缺点就是可扩展性比较差。

X-code 的编码方法如图 1-5 所示，每一列的元素存储在同一个数据节点中，每个节点上存储着  $k$  个编码块。其中红色块代表数据块，蓝色块代表编码块， $n$  需要是一个大于 2 的素数，数据个数必须为  $n - 2$  个，编码块的个数也必须是 2 个。此外，第一个编码块由主对角线上的数据块进行计算得到，另外一个编码块由副对角线的数据块计算得到，这些计算需要用到  $k$  个磁盘上的数据块，计算成本较高，且修复时需

要按照一定的顺序进行计算，失去了并行修复的可能，这些均是 X-code 的不足之处。

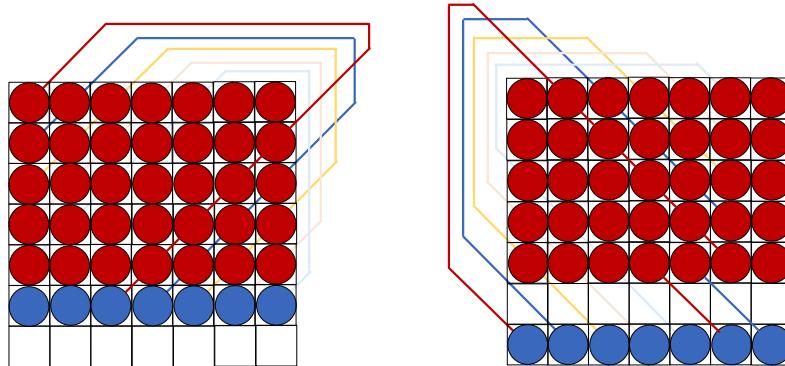


图 1-5 X-code 的编码结构示意图

阵列码在计算上的主要特点就是所有的运算都是异或运算，相较于传统的 RS 编码在计算性能上有优化，但需要的数据量却并没有丝毫减少，传输的数据量较大。

### 1.3.3 基于网络编码的纠删码

网络编码最初由 Ahlswede 等<sup>[43]</sup>于 2000 年首次提出。网络编码优化了网络中节点的功能，使得其不仅可以存储转发数据，并且还可以在转发数据之前对数据进行编码和矩阵计算，从而提高了网络吞吐率。

Dimakis 等<sup>[44]</sup>将网络编码的思想应用到纠删码上并称之为再生码 (Regenerating Codes)。再生码允许新生节点访问多于  $k$  个节点，从而显著降低修复过程需要传输的总数据量。

图 1-6 显示了再生码修复失效数据节点过程，总共有四个用来存储数据的节点，并且每个存储节点上存放着两个数据块，并且是通过相关的矩阵运算生成的。再生码修复数据的方法是首先在存储节点中进行计算后再进行数据分发，传输数据块的总数目为 3，相较于传统的修复方法，再生码修复数据传输量降低了 25%。

Dimakis 等<sup>[44]</sup>提出的再生码可以对丢失的数据进行精确修复，但对编码块只能进行功能性修复。这里修复出的数据并不等同于原来的数据但是可以与原来的存储系统维持相同的容错度。针对此问题，Wu 等<sup>[45]</sup>提出了确定性再生码 (Deterministic Regenerating Codes)，并从概率学角度证明了其存在性。

根据存储效率的不同，再生码可以分为  $\frac{k}{n} \leq \frac{1}{2}$  的低比特率码和  $\frac{k}{n} > \frac{1}{2}$  的高比特率码。一般来说，这样的方式可以减少数据的传输量，但是由于编码系数所在域的要求

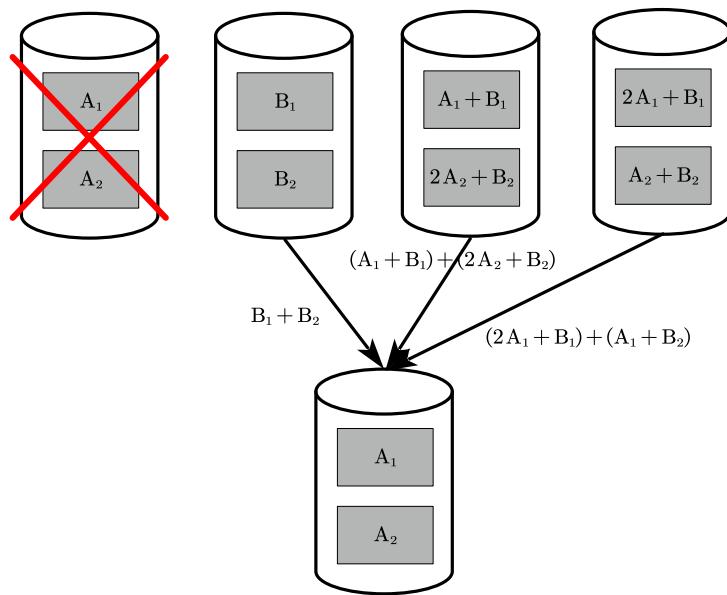


图 1-6 再生码修复失效数据节点示意图

限制、选择方法的不规则和难以实现等原因，目前应用并不广泛。

#### 1.3.4 基于分组结构的纠删码

基于分组结构的纠删码可以减少数据修复时需要下载的数据量，因传统的 RS 码在修复一个丢失的块时需要下载的数据条带长度为  $k$ ，而分组结构的纠删码的方法则是将  $k$  减小从而达到降低数据量的目标。分组编码的纠删码首先将数据条带分成若干组，各个组各自计算相应的编码块，实现组内修复丢失块，需要的块数则取决于每个组内的数据块数。RS 码其实是一种特殊的分组编码，可以看成将整个数据条带作为一组，此外根据分组方法的不同，此类纠删码可以分为两类：水平分组结构的纠删码和交叉分组结构的纠删码。

比较经典的水平分组纠删码一般是将数据条带分为两组。每个小组生成一个局部的冗余块，然后整个条带再生产一个全局的冗余块。这样数据修复就可以依靠小组内的编码块完成，降低了修复成本。图 1-7 显示的是微软的 Azure 系统<sup>[46]</sup> 采用 LRC (Local Reconstruction Codes)<sup>[8]</sup> 的一个实际的例子 LRC(8,2,2)。可以看出，它将数据块平均分为了两组，每组各自内部计算出一块局部冗余编码块，相当于一个 RS(5,4) 码。对于两个条带再生成 2 个全局的校验块，则相当于一个 RS(10,8) 码。当丢失块数不多时，例如丢失 1 个块，则只需要 4 个块即可进行数据修复，这样就有效降低了修复成本。

由此可以发现，组数越多，那么可以容忍的失效的块数也就越多，故 Pyramid 码<sup>[47]</sup>便诞生了，它的特点就是可以对条带内的数据块进行任意层次的分组，层次越多平均的修复成本也就越低，但是相对的存储空间消耗也就越大，因为产生了更多的全局校验块，对于 Pyramid 码需要根据实际的存储系统环境选择更加适中的方案。除此以外水平结构的编码还有运行在 Facebook 旗下的分布式存储系统中的 XORing Elephants<sup>[3]</sup>、EXPYramid<sup>[48]</sup>、LRCs<sup>[3]</sup>等。EXPYramid 码是对两层 Pyramid 码的一种改进，其分为了三组。

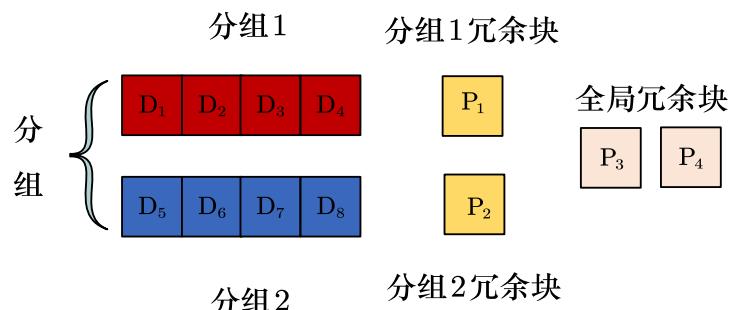


图 1-7 LRC(8, 2, 2) 的编码示意图

交叉分组结构的纠删码与水平分组不同之处大致有两点。第一，交叉分组纠删码的分组方式不同，交叉分组的各组之间相互重叠但却互不包含，相比之下水平分组各组之间一般不重叠且会有一个全局的校验块包含着各个分组的校验块。第二，交叉分组纠删码编码计算较为简单，基于异或计算。不难看出，交叉分组有着相对来说较低的计算性能消耗的优点。常见的交叉结构的纠删码有 Woitaszek 等<sup>[49]</sup>提出的 Tornado 码，如图 1-8 所示，其任何组别的冗余块均为组内的数据块的校验块，并且修复任何一个数据块或者冗余块只需 4 个或 3 个块，避免了大量下载编码块的特殊情况。此外还有 LDPC<sup>[50]</sup>、WEAVER 码<sup>[51]</sup>和 Tanner 码、Hover 码<sup>[52]</sup>等。

## 1.4 基于纠删码的数据修复技术

在基于纠删码的分布式存储系统中，数据传输并不是简单的线性路线，每当修复一个数据块时，提供者节点要向新生节点传输若干数据块从而进行计算，数据的传输路径往往比较多样化，根据传输路径的不同一般可分为两种修复技术：基于星型拓扑的数据修复、基于树型拓扑的数据修复。除此之外，还有一种基于 XOR 的纠删码数据修复技术。

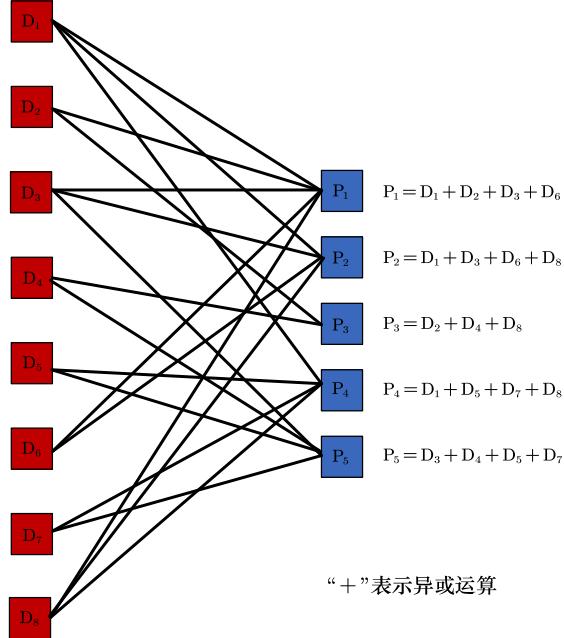


图 1-8 Tornado 码的构造示意图

#### 1.4.1 基于星型拓扑的数据修复

基于星型拓扑的串行修复技术 (Star Structure Based, SSR) 是按照顺序修复最近失效的数据块，修复过程遵循星型结构。Dimakis 等<sup>[44]</sup>, Wu 等<sup>[45]</sup>提出的再生码数据修复过程要求每个存活节点向新生节点传输均等的数据量进行修复，数据收集节点 (DC 节点) 仅连接  $k$  个节点，通常是  $k - 1$  个数据节点和第一个校验数据节点，并从每个节点下载  $\frac{M}{k}$  的数据量 ( $M$  为原文件大小)。这里存在一个非常明显的缺陷，节点可以选择的链路比较单一，带宽较高的链路很有可能会被忽视，导致修复效率降低。Shah 等<sup>[53]</sup>提出了弹性修复的方法，即允许存活节点和 DC 节点充分利用所有可用的链路，从中选择带宽较大的链路进行数据传输，传输的数据不均等。DC 节点驱动任务从节点  $i$  ( $1 \leq i \leq n$ ) 获取  $\mu_i$  ( $0 \leq \mu_i \leq \alpha$ )，其中需要满足总的获取量不小于  $M$  即可。同理，新生节点从存活节点  $i$  ( $1 \leq i \leq n$ ) 接收  $\beta_i$  ( $0 \leq \beta_i \leq \beta_{max}$ ) 数据量，满足总接收量大于或等于一个设定参数  $\gamma$  即可，用不等式表示则如式 1.1 和 1.2：

$$\sum_{i=1}^n \mu_i \geq M, 0 \leq \mu_i \leq \alpha \quad (1.1)$$

$$\sum_{i=1}^n \beta_i \geq \gamma, 0 \leq \beta_i \leq \alpha \quad (1.2)$$

节点发生故障时，新生节点可以确定从任意  $k$  个节点下载  $\frac{M}{k}$  的数据量，修复产生的流量相等于原文件大小。对于弹性修复方法而言，新生节点，可以根据可用带宽的大小从不同节点上下载不均等的数据来降低修复时间。当然星型拓扑的缺点也很明显，DC 节点需要收集多个节点的数据，修复速度会受到磁盘和网络 I/O 的限制，编解码的计算消耗也很大。

#### 1.4.2 基于树型修复拓扑的数据修复

上节所描述的星型拓扑的技术其实是树型拓扑修复技术的特殊情况，然而修复速度受到了诸多的制约，为了提升修复的速度，树型拓扑修复方法因此被提出。

Li 等<sup>[14,54]</sup> 独辟蹊径地首次设计了一个新的基于网络带宽的树形修复方法（Tree-structured Data Regeneration），利用木桶原理并通过提高最低传输速度来加速数据的修复。修复路径与树结构类似，组成的元素包括提供者节点和新生节点。提供节点主要是接收数据和对数据进行存储和计算然后再进行转发，在这过程中生成树结构，其中最大生成树是最优的修复树。

树型修复方法是对星型修复方法的改进，传统的方法在修复时每个节点承担的修复任务繁重，导致节点在 I/O 性能和计算上很难取得一个良好的平衡，有大量节点处于闲置状态，这样整个系统的资源就没有完全利用起来。树型修复方法利用树结构的优势可以将计算任务和传输任务在网络中并行地进行，同时可以利用流水线优化技术，加速修复任务的进行。当然缺点也比较明显，在网络带宽变化较快的环境中，树型修复的带宽判断往往不是当前网络的最优，容易陷入局部最优，而在全局来看也同样浪费了部分资源。

另外两类比较经典的树型修复修复技术分别是根据网络拓扑优化的 NTar<sup>[55]</sup> 和面向多节点修复的并行修复技术<sup>[56]</sup>。NTar 提出了网络距离的概念，综合考量传输流量和网络距离来加速修复。并行修复技术<sup>[56]</sup> 构建了分离边策略和共享边策略，目标是充分地利用网络资源，其存在的问题也比较明显，即会出现无法建立树型路径的可能以及空闲带宽没有完全得到充分利用。

### 1.4.3 基于 XOR 的纠删码数据修复

Xiang 等<sup>[57]</sup>提出的基于异或运算 (XOR) 的纠删码低网络负载数据修复技术，专门用于优化 RDP 码<sup>[39]</sup>修复失效数据所传输的数据量。Khan 等<sup>[58]</sup>将其理论进行一般化并拓展到异或运算上，任何通过异或运算构建的纠删码都可以采用这种修复方式。目前有三种比较主流的基于异或运算的纠删码数据修复方法，根据修复目标的不同，可以分为三种方法，分别面向优化降级读性能的方法、降低修复时间的方法以及降低修复成本的方法。

面向优化降级读性能的方法。降级读主要针对的是数据的暂时失效，即在请求读取系统数据时，先将数据构建出来进行读取，然后再进行修复操作。对于带宽不对等问题，Zhu 等<sup>[59]</sup>提出了枚举贪心算法（Enumerated Greedy Algorithm），简称 EG 算法，EG 算法通过排列组合计算出  $d$  个可用节点可能的情况，确定在每一种组合下有  $l$  个 CDREs，并且为每个 CDREs 计算其降级读时间，更新降级读时间，使每个块的降级读时间最小。

降低修复时间的方法。Niu 等<sup>[60]</sup>针对 RAID-6 编码系统提出了多条带修复的策略，其将修复过程划分为三个部分，同时还考虑了系统 I/O 和节点的带宽不对等性，提出了并行不对等修复算法（parallel heterogeneous recovery），简称 PHR 算法。算法对三个部分的修复过程分别进行了优化，并且是按顺序执行的，各个条带修复呈现相对独立，通过线程技术和流水线修复技术可以加速修复任务，从而降低修复时间。

降低修复成本的方法。针对同一条传输路径上由于带宽不对等造成成本差异，Zhu 等<sup>[61]</sup>结合传输成本和带宽性能，定义修复总成本为  $C = \sum_{i=0, i \neq k}^p w_i y_i$ ，其中假设节点  $k$  失效， $w_i$  表示从节点  $i$  读取一个对象的成本， $y_i$  表示从节点  $i$  读取的对象数目，进而他们提出了基于传输成本的异构恢复（cost-based heterogeneous recovery）算法，简称 CHR 算法。CHR 算法所有的最小读取量恢复序列进行枚举，并且计算这些序列的修复成本的总和，返回其中最小的总修复成本对应的修复序列。

## 1.5 论文主要研究内容

本文面向纠删码存储系统中的数据预先修复问题和混合纠删码问题，以纠删码技术为基础，以高可靠性、低存储成本和高修复速度为目标，对纠删码数据修复速度低，修复代价高等问题开展研究。

### (1) 预先数据修复技术研究

纠删码面临的最基本问题就是过量的修复开销：修复流量随着存储冗余度的降低而增加。因此，前人对改善纠删码的修复性能进行了广泛的研究，使修复过程中的修复流量或 I/O 最小化，或设计适用于所有实践中的纠删码包括 RS 码的提高修复效率的技术。目前，大部分的传统修复方法都是被动修复，只有在检测到节点故障后才会触发修复操作。如果可以提前预测即将发生的故障，就可以在任何实际故障发生之前主动修复任何即将发生的节点故障，以提高系统可靠性。

针对反应式修复降低系统可靠性问题，通过结合迁移与重建来设计实现预先修复的机制，用以修复 STF(Soon To Fail) 节点。对于重建集与迁移集问题，设计了划分迁移集与重建集算法 SMSRS (Split Migration Set And Reconstruction Set) 算法，根据中继节点的特性，提出了 ISA (Improved SMFRepair Algorithm) 算法来进行节点修复的调度，使得反应式修复慢的问题得到了缓解和优化。

### (2) 混合纠删码数据修复技术研究

在传统的存储系统中大多只采用一种纠删码进行数据的处理和存储，这种方式一般先将文件分割成固定大小的数据块，再将这些数据块每  $k$  个作为一组，每组独立进行编码操作生成  $n$  个块，其中  $n$  个块的集合称为一个条带 (Stripe)，然而这种方式难以在保持低存储空间消耗的情况下降低退化读的延迟时间。

针对单一纠删码的延迟问题，在前人工作的基础上，提出了一种可感知数据热度的负载动态自适应的混合纠删码 (LRC&HH) 数据修复方案。旨在根据真实储存系统数据修复场景中的数据 I/O 的特点，以及数据访问和故障事件的时空局部性，提出更加符合相应情况的混合纠删码修复方案，从而对于计算开销型修复任务降低其开销和存储成本，对读取密集型和频繁重建型修复任务降低其工作负载，并加速系统的修复速度和减少重建时间。

### (3) 基于混合纠删码的容错存储原型系统设计与实现

为了进一步验证本文的理论研究成果，本文设计实现了基于混合纠删码的容错存储原型系统。该系统有三个特点：1) 基于开源分布式存储中心的离散事件模拟器 SimEDC 进行设计实现；2) 含有丰富的混合纠删码方案，包括已有的 HACFS、EC-Fusion 以及 LRC&HH，支持相应的扩展接口；3) 支持修复调度中的重建与迁移的融合。容错存储原型系统设计主要包含以下模块：存储中心架构，节点故障模型，混合纠删码策略，节点放置策略，可靠性度量指标，事件处理模式。实验结果表明，基于

混合纠删码结构的容错存储原型系统具有较高的可靠性，PDL、NOMDL 和 BR 都有一定程度的降低，系统可靠性得到提升。

## 1.6 论文组织结构

本文共分为六章，研究方向技术路线如图 1-9所示，各章节组织结构如下：

**第一章：绪论。**论述了课题的研究背景和意义。介绍了分布式存储系统和分布式存储系统中的数据容错问题；概述了纠删码技术及基于纠删码的数据修复技术，简述了本文的主要研究内容。

**第二章：相关研究。**综述了与本文主要工作相关的研究内容，主要包括编码结构优化的数据修复技术、调度优化的数据修复技术、带宽优化的数据修复技术、基于混合纠删码的数据修复技术和基于预先修复的数据修复技术等并总结了各种技术的优点与不足。

**第三章：基于预先重建迁移的数据修复技术研究。**首先介绍了预先修复的重建技术和迁移技术的修复细节；然后提出了重建和迁移集的划分算法，结合中继节点的概念提出了预先修复的调度算法；最后通过对比仿真实验评估了预先修复算法的性能。

**第四章：基于混合纠删码的数据修复技术研究。**首先分析了混合纠删码的编码策略选择方案 LRC 和 HH 码；对自适应策略的编码分配进行了数学分析与验证并提出自适应选择算法；然后分析了与自适应策略相对应的编码切换算法，并进行了算法分析；最后通过数据集的仿真实验，对自适应策略与传统方法进行了对比分析比较。

**第五章：基于混合纠删码的容错存储原型系统设计与实现。**介绍了本文设计实现的混合纠删码容错存储系统，包括总体架构、数据修复的基本流程以及混合纠删码的实现方式，并通过仿真实验对系统的可靠性进行了分析。

**第六章：总结与展望。**总结本文的主要研究工作并指出未来的研究方向。

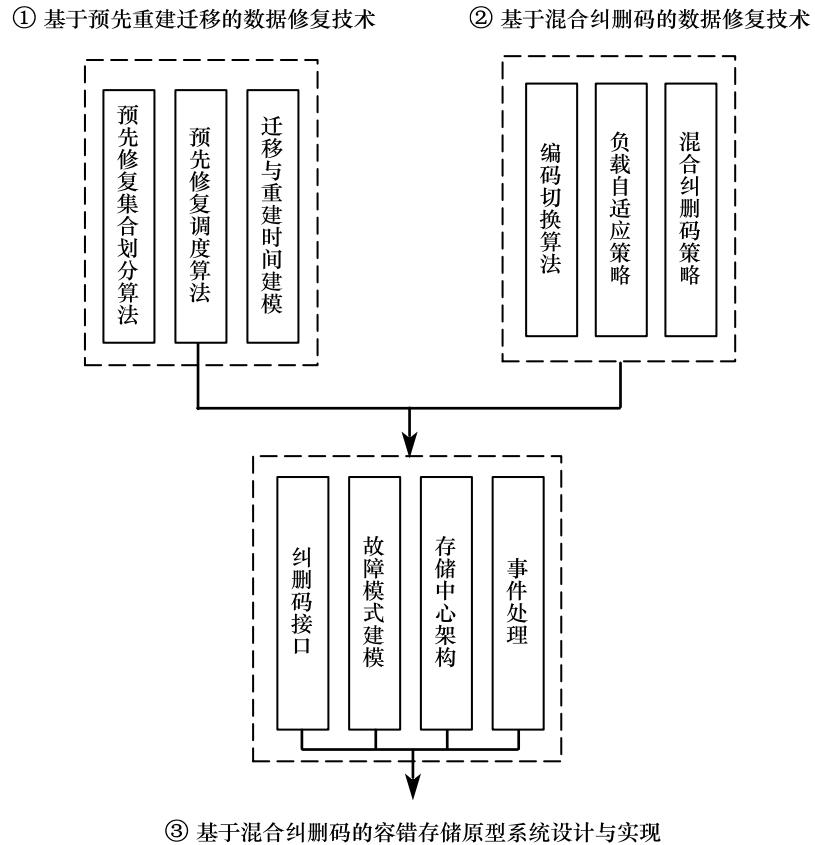


图 1-9 本文研究点技术路线

## 第二章 相关研究

目前针对纠删码数据的修复技术大多集中在修复调度和构建新的存储策略并配备相应的适应这种策略的修复技术。除此之外，还有改进编码结构，感知带宽环境进而优化的，以及动态切换纠删码和融合多种技术进行构建数据修复流程等技术，本章对上述内容进行简要介绍。

### 2.1 基于编码结构优化的数据修复技术

在多个同时发生的故障事件中，单个节点故障的恢复率可达到 98%<sup>[62,63]</sup>。因此，减少单一故障修复所产生的延迟对于提高系统性能是至关重要的。Dimakis 等<sup>[44,64]</sup>提出了再生码 RGC，它可以在存储开销和修复故障节点所需的数据量之间取得良好的均衡。另一方面，处于最小存储的再生码均衡点的 MSR (Minimum Storage Regenerating) 对于大部分纠删码来说是最优的替代选择，因为 MSR 可以在最小存储开销下实现最佳修复成本和最大距离可分 MDS 的特性。MDS 属性意味着只要故障块数小于或等于校验块数，那么数据就可以被恢复。然而，MSR 码有一些缺点，诸如：(1) 大多数现有的 MSR 码不是系统码，只在编码后存储奇偶校验块信息，因此造成了很高的计算开销；(2) 许多纠删码的存储开销超过系统存储量的 2 倍，因此在实际存储系统中没有用处不大；(3) 存储开销低于系统存储量 2 倍的纠删码不仅需要复杂且容易出错的代码策略，而且还需要很高资源的开销。对于这些问题，Butterfly<sup>[65]</sup> 是可以在保持低存储开销（低于 2 倍）的情况下并同时具有实际应用价值的 MSR 码，是一种系统化的码，读请求可以直接得到数据而不需要任何解码操作。然而，Butterfly 最多只能容忍两个块的故障，因此不能达到高可靠性的标准。此外，它还存在一个成簇的子包，它将块分成小的片段（最小的编码元素），导致编解码操作中会损失块的位置数据，还会产生数据读取和传输的高通信成本。

Ye 等<sup>[66]</sup> 提出了 Hybrid-RC 利用 Butterfly 码来对 MSR 进行优化改进，理论上它是可以在低存储开销和低修复带宽之间取得最优均衡的纠删码。Hybrid-RC 使用 Butterfly 作为分组校验编码的编码策略，使用伽罗瓦运算进行全局校验码的计算从而保持存储系统的可靠性。与 LRC 相比，在相同的存储开销下，Hybrid-RC 在处理单块故障恢复时有着更多的数据块参与。每个块的 I/O 消耗相对减少且数据传输是高度

并行的，从而达到了良好的平衡。

Hybrid-RC 由三个标准化参数组成， $k, l, g$ ，记为  $A(k, l, g)$ 。一个条带中有  $k$  个数据块和  $l + g$  个校验块，其中  $g$  个全局校验块通过对所有数据块进行伽罗瓦运算求出。Hybrid-RC 将  $k$  块数据分成  $\frac{l}{2}$  个组，其中  $l$  必须为偶数。当  $k$  不能被  $\frac{l}{2}$  整除时且余数为  $r$  时，那么第 0 到第  $r - 1$  组应该包含  $\lceil \frac{2k}{l} \rceil$  个块并且每个块被分成了  $2^{\lceil \frac{2k}{l} \rceil - 1}$  个片段。如果不同的组的切片数不同，那么从第  $r - 1$  个到第  $\frac{l}{2} - 1$  个组应该加倍以保证每个块的存储消耗保持一样。例如，考虑  $A(7, 4, 1)$  的 Hybrid-RC 码，节点中的每个块大小为 1GB。第一个组包含 4 个数据块和 2 个校验块，所以每个块有 8 个切片且每个切片大小为 0.125GB。但是，第二个组只有 3 个数据块，因此每个块有 4 个切片。为了两个组能保持同样的块大小，第二组的每个块大小需要为 0.25GB。总的来说，不同组的奇偶块构建和块修复是相互独立的。对于全局编码或修复，最小的操作元素是相同的。因此，不同数量或大小的切片对全局奇偶校验计算是没有影响的。在一个  $(k, l, g)$  的 Hybrid-RC 码中， $D_{i,j}$  表示  $D_i (0 \leq i < k)$  数据块的第  $j$  个数据切片。设  $P_h$  为垂直码， $P_b$  为 butterfly 码， $P_{h,x}$  和  $P_{b,x}$  分别为第  $x$  组的分组校验码，其中  $0 \leq x < l/2$ 。类似地，第  $y$  个全局校验码为  $P_{g,y}$ ，其中  $0 \leq y < g$ 。如图 2-1 所示的是，第二组和全局校验码的构建过程。不同颜色的高亮表示不同元素所属相同的运算集群。

全局							$P_{g,0}$
$D_0$	$D_1$	$D_2$	$D_3$	$D_4$	$D_5$	$D_6$	$D_0 + 2D_1 + 4D_2 + 8D_3 + 16D_4 + 32D_5 + 64D_6$

第 2 个组			$P_{h,1}$	$P_{b,1}$		
$D_4$	$D_5$	$D_6$				
$D_{4,0}$	$D_{5,0}$	$D_{6,0}$	$D_{4,0} + D_{5,0} + D_{6,0}$	$D_{4,3} +$	$D_{5,1} +$	$D_{6,0}$
$D_{4,1}$	$D_{5,1}$	$D_{6,1}$	$D_{4,1} + D_{5,1} + D_{6,1}$	$D_{4,2} +$	$D_{5,0} + D_{6,0}$	$+ D_{6,1}$
$D_{4,2}$	$D_{5,2}$	$D_{6,2}$	$D_{4,2} + D_{5,2} + D_{6,2}$	$D_{4,1} + D_{5,1} + D_{6,1} + D_{5,3} + D_{6,3} + D_{6,2}$		
$D_{4,3}$	$D_{5,3}$	$D_{6,3}$	$D_{4,3} + D_{5,3} + D_{6,3}$	$D_{4,0} + D_{5,0} + D_{6,0} + D_{5,2} +$	$D_{6,3}$	

图 2-1 (7,4,1)Hybrid-RC

Xie 等<sup>[67]</sup>，提出了可用性区域 AZ (Availability Zone) 的概念，所谓 AZ 就是数据中心中的一个物理隔离区域，它由几个服务器机架或数据中心的一个机柜，甚至是整个数据中心组成。通过使用多个 AZ，云存储系统可以保有高可用性，高可靠性以及低网络延时。当前的 AZ 有以下三个缺点，首先，对于基于 XOR 的编码，大多

数基于 XOR 的纠删码 (Blaum 等<sup>[38]</sup>, Corbett 等<sup>[39]</sup>) 只能容忍两个或三个节点的故障, 这不能满足容忍 AZ 级节点容错能力的要求。一些基于 XOR 的编码可以容忍三个以上的节点, 但可扩展性却相当差<sup>[41,68]</sup>。第二, 对于基于 RS 的纠删码, 虽然可以提供较高的可扩展性, 但修复成本 (如重建 I/O, 修复带宽) 非常高。第三, 虽然最小存储再生码<sup>[69,70]</sup>可以有效地减少恢复开销, 但计算复杂度比基于 RS 和基于 XOR 的代码要高得多。基于此, 他们提出了 AZ-Code 为 AZ 提供低修复开销且高可靠性的保证。AZ-Code 是基于 MSR 码和 LRC 码的改进, 其利用一个特定的 MSR 码作为 LRC 码的局部校验块, 用 RS 码生成全局校验块, 通过数学分析和 Hadoop 系统实验分析, AZ-Code 节省了高达 78.24% 的修复带宽。

## 2.2 基于调度优化的数据修复技术

在加速节点传输的技术上, PPR<sup>[71]</sup>, RP<sup>[72]</sup>, TDR<sup>[14]</sup> 等技术充分利用各节点的带宽资源, 加速修复的速度。由于磁盘 I/O 和编码计算的时间开销比网络传输的要小得多<sup>[71,72]</sup>, 所以优化修复数据的传输路径对修复性能的提升有着很大的意义。对于单节点恢复, PPR 和 RP 使用的是一种固定的树结构, 而 TDR 技术计算的是最大生成树作为传输结构。在 RP 和 TDR 中, 节点每次只执行一种任务, 即将数据输送到下一个节点, 而在 PPR 中, 部分节点并行地将数据传输到它们的专属节点。PR 和 TDR 没有对多节点修复进行优化, 而 PPR 需要额外的中继节点来参与多节点的修复, 同时也造成了多余的带宽消耗。PPR 和 RP 针对的是流量均匀的网络, 而对于 TDR 来说, 基于的假设是: 每个链路的带宽是一个常数, 不受到连接到同一节点的节点数的影响。

但是在带宽不对等的环境中, 当一个节点同时接收两个或多个节点的数据时, 其与每个对应的节点的实时带宽会减半或者相应比例的减少, 若这样进行并行传输可能会比通过中继节点传输消耗的时间更短。基于此, Bai 等<sup>[73]</sup> 提出了 PPT (Parallel Pipeline Tree) 通用传输技术, 根据各节点之间的带宽值构建了树状修复路径, 进一步利用带宽不对等来加速单节点的修复。若 PPT 找不到进一步的优化, 则退化到 RP 进行修复。当多节点发生故障时, 每个节点的数据请求需要接收所有中继节点的数据, 每个中继节点将向所有请求节点发送数据。在大多数情况下, 中继节点的数量多于请求节点的数量, 则整个网络的修复瓶颈则是请求节点的带宽瓶颈。为了优化此问题, 他们同时提出了一种通用解决方案, 称为 PPCT (Parallel Pipeline Cross-Tree), 从而

构建了一种交叉修复路径，并在交叉树上并行地输送数据，进一步加快多节点修复速度，且 PPT 和 PPCT 支持基于 Reed-Solomon (RS) 码的实用纠删码结构。

然而，Zhou 等<sup>[74]</sup>发现 PPT 的并行传输会引发网络拥堵和带宽争夺消耗，同时会产生新的低带宽链路。基于此他们提出了 SMFRepair，一种单节点多级传输的修复技术。如图 2-3b 所示，在分布式存储系统中，图 2-2d 中空闲节点是非常普遍的，并且通常有足够的可用带宽。在图 2-2a 中，*helper1* 和 *helper2* 会异步地向 *requestor* 传输数据块。而在图 2-2b 中，PPR 的链接 *helper2* → *helper2* 的带宽很低，只有 2MB/s。通过利用链路之间的带宽差距，PPT 使用并行传输来绕过低带宽的链路。在图 2-2c 中，*helper1* 和 *helper2* 同时向 *requestor* 发送数据块。因为 *helper1* → *requestor* 的带宽为  $8/2=4$ MB/s，且 *helper1* → *requestor* 的带宽为  $5/2=2.5$ MB/s，都大于 *helper2* → *helper1* 的 2MB/s 的低带宽，所以通过新的低带宽链路可以避免带宽瓶颈问题。而在 Zhou 等<sup>[74]</sup>的研究中，可以通过空闲节点来绕过低带宽的链接。在图 2-2d 中，他们使用空闲节点来构建两个链接 (*helper2* → 空闲节点和空闲节点 → *helper1*)，它满足两个链路带宽的倒数之和小于第三个链路带宽的倒数的条件 ( $1/10 + 1/10 < 1/2$ )。他们用这个转发链路来代替低带宽链路，且对于一个 20MB 的数据块，PPR 需要  $20/2+20/8=12.5$ s，PPT 需要  $\max(20/4, 20/2.5)=8.0$ s，而 SMFRepair 只需要  $20/10+20/10+20/8=6.5$ s。通过仿真实验表明，与最先进的修复技术相比，SMFRepair 可以加速单节点的恢复，最高可达 47.69%，且他们提出的多节点调度算法 MSRepair 可以减少多节点恢复时间 33.78%~67.53%。

针对快速修复技术，还有的研究聚焦于改善数据放置技术以提高修复速度。Liu 等<sup>[75]</sup>提出了一种数据放置算法 ESet 来改善恢复性能，他们通过理论分析说明 ESet 可以使用相应的配置实现 I/O 聚集情况下的快速单一故障修复，ESet 是对现有修复技术的增强，最后他们通过在 HDFS<sup>[5]</sup> 和 Ceph<sup>[76]</sup> 中的实验证明 ESetStore 的有效性，并将 ESetStore 进行了开源<sup>1</sup>。此外，Xu 等<sup>[77]</sup>针对不同节点上工作负载的不均衡问题，定义了修复平行度 DRP 用来反映一个批次的重建任务中的负载平衡，同时提出了一个平衡调度框架 SelectiveEC，通过动态选择重建条带和平衡节点的上传下载带宽来改善单一修复任务的负载均衡。Lin 等<sup>[78]</sup>提出了一种快速修复调度框架 RepairBoost，可以协助现有的线性纠删码和修复算法提高全节点的修复性能。RepairBoost 建立在三个主要技术上：（1）总体修复，采用有向无环图来进行单块数据的修复；（2）平衡修复流量，同时平衡上传和下载的修复流量；（3）传输调度，精细地分配请求的数

<sup>1</sup>开源地址：<https://github.com/stevenlcj/ESetStore>

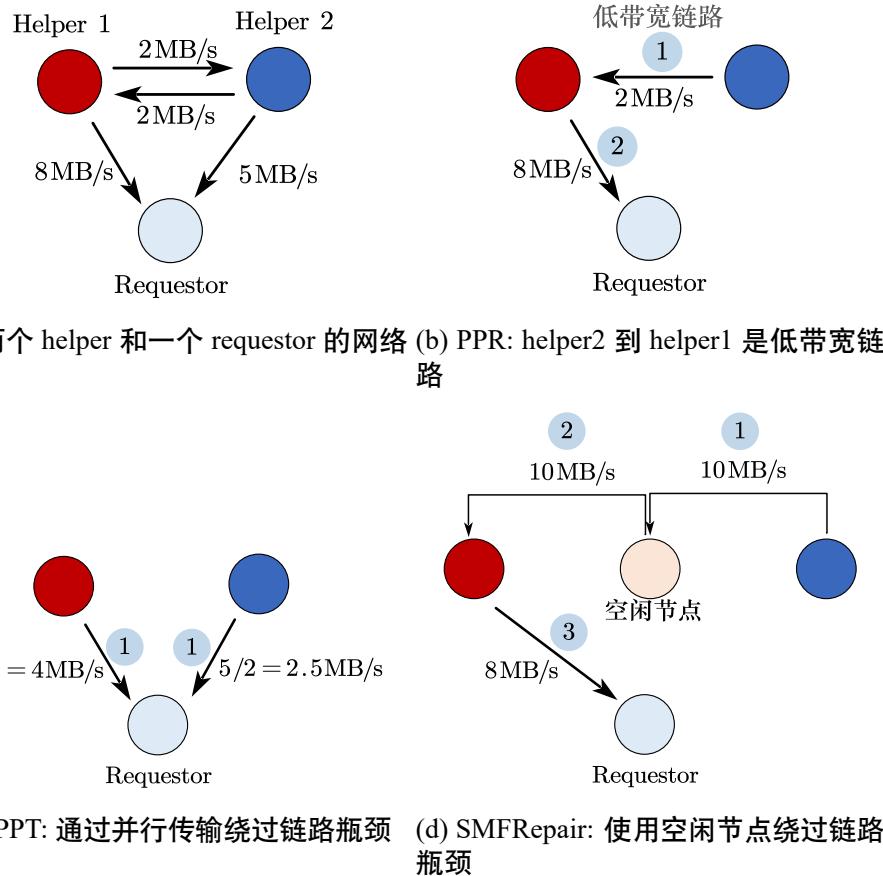


图 2-2 空闲节点可以绕过链路瓶颈，对于一个 20MB 的数据块而言，PPR、PPT 和 SMFRepair 的修复时间为 12.5s, 8.0s, 6.5s

据块，使最未被占用的带宽达到饱和状态。RepairBoost 对于各种纠删码和修复算法，可以将修复速度提高 35.0%~97.1%。

### 2.3 基于带宽优化的数据修复技术

目前实际的分布式存储系统经常独立地安排每个修复任务，鲜有考虑到任务彼此的影响。这些分布式的修复任务共享着底层基础设施的网络、计算和存储资源，这就造成了数据流之间的干扰，导致系统资源利用率低下。为了缓解这一问题，Popa 等<sup>[79]</sup>提出了 Elasticswitch，一种可以在各个服务器之间分配带宽的技术。通过这一技术，将服务器的总带宽资源进行整合，可以为每个修复任务分配必要的网络带宽。此外，系统中主要有三种任务需要优化执行，其中包括修复任务调度、调度数据的选择以及不同网段的后台修复任务的带宽分配。因此，Li 等<sup>[80]</sup>提出了 LSPT (Linear Programming for Selected Tasks) 在满足数据放置、网络拓扑结构以及带宽限制的条件

下，最大限度地增加满足时间限制的修复任务数。为了最佳地安排每个任务，LSPT 共同解决了（1）确定用于产生修复流量的纠删码块的选择问题，（2）在修复任务之间分配 TOR 交换机和聚合交换机带宽的带宽分配问题，以及（3）根据最后修复时间限制安排任务的调度问题。

假设一个数据存储中心的一个聚合交换机连接着  $u$  个机架交换器（Top-of-Rack）。 $r$  个存储节点 ( $\mathcal{R} = \{1, 2, \dots, r\}$ ) 放置在  $u$  个机架内 ( $\mathcal{U} = \{1, 2, \dots, u\}$ ) 并且每个 TOR 交换器。同一个机架内所有的节点的修复流量不需要经过聚合交换机，而不同机架上的存储节点之间的流量则不需要经过两个 TOR 交换机和聚合交换机。每份文件  $i$  用  $(n_i, k_i)$  纠删码进行存储，保证为 MDS (Maximum-Distance-Separable) 码，确保  $n_i$  中的任意  $k_i$  块可以重建文件  $i$ 。此问题可以描述为，令  $\mathcal{A} = \{A_1, A_2, \dots, A_m\}$  表示为系统后台任务的集合，诸如备份，修复和数据再分配。对于每个任务  $A_i$ ，相应地有关联参数包括数据块分发源  $n_i$  (表示为  $o_{i,1} \in \mathcal{U}, o_{i,2} \in \mathcal{U}, \dots, o_{i,n_i} \in \mathcal{U}$ )，分发接收节点 (表示为  $p_i \in \mathcal{U}$ )，需要获取的块个数  $k_i$ ，每个块的大小  $v_i$ ，任务开始时间  $s_i$ ，以及任务的终止时间  $d_i$ 。任务的开始时间和结束时间用秒作为单位，满足  $0 \leq s_i \leq d_i$ 。不失一般性，考虑带有时间段的系统，假设  $y_{i,j}$  为编码块的选择变量，例如  $y_{i,j} = 1$  表示块  $j$  被选择去执行任务  $A_i$ ，而  $y_{i,j} = 0$  则未被选中。既然块  $k_i$  必然被选择，则有

$$\sum_j y_{i,j} = k_i, \forall i \quad (2.1)$$

为了确定成功完成的任务数，使用二元变量  $z_i$  来进行表示，当  $z_i = 1$  时任务  $A_i$  在时间限制内完成，否则未完成。设  $x_{t,i,j}$  是在时间段  $t$  中分配给传输任务  $A_i$  的  $j$  块的数据流量的带宽，如果任务在最后期限  $d_i$  之前完成，所有的  $k$  个块传输应该在  $d_i$  之间完成，即时间限制约束条件为：

$$\sum_{t=s_i}^{d_i} x_{t,i,j} y_{i,j} \geq v_i, \text{ if } z_i = 1, \forall i, \forall j \quad (2.2)$$

由于每个传输头尾对都有一个预先确定的路线，对于一个给定的任务集，用  $RC_g$  来表示跨越一个 TOR 或者聚合交换机  $g$  的任务或者块的集合，即  $(i, j) \in RC_g$  如果任务  $i$  的块  $j$  的流量经过了交换机  $g$ 。同样， $SC_h$  是经过节点  $h$  的任务或者块集合。每个 TOR 交换机有着容量的限制  $CTA$  且每个节点有着容量限制  $CST$ ，因此有以下容量限制：

$$\sum_{(i,j) \in RC_g} x_{t,i,j} y_{i,j} \leq CTA, \forall g, t \quad (2.3)$$

$$\sum_{(i,j) \in SC_h} x_{t,i,j} y_{i,j} \leq CST, \forall h, t \quad (2.4)$$

任务的目标就是要在基于纠删码的存储系统中最大化在时间截止内成功完成的修复任务数。可以看做是一个联合调度和优化问题：

$$\begin{aligned} \max \quad & \sum_i z_i \\ \text{s.t.} \quad & \sum_j y_{i,j} = k_i, \forall i \\ & \sum_{t=s_i}^{d_i} x_{t,i,j} y_{i,j} \geq v_i z_i, \forall i \\ & \sum_{(i,j) \in RC_g} x_{t,i,j} y_{i,j} \leq CTA, \forall g, t \\ & \sum_{(i,j) \in SC_h} x_{t,i,j} y_{i,j} \leq CST, \forall h, t \\ \text{var.} \quad & x_{t,i,j} \geq 0, y_{i,j} \in \{0, 1\}, z_i \in \{0, 1\} \end{aligned} \quad (2.5)$$

其中式 2.5 的第三行正好对应式 2.2 任务成功时，即  $z_i = 1$ 。

此外，还有部分研究集中在提升存储系统的适应动态网络变化的能力。Li 等<sup>[54]</sup>提出了一个基于 Prim 算法的树状拓扑感知修复方案，并构建了一个基于启发式方法进行修复的修复方案，考虑了修复过程中第二个存储节点丢失的情况。然而，这种方案所造成的延迟通常情况下不满足系统的可靠性要求。其他相关研究集中在设计特定的网络拓扑结构，Akhlaghi 等<sup>[81]</sup>根据读取节点的带宽开销，将节点分为了两个集合，即“低廉”和“昂贵”两种集合。他们在这里引入了广义的再生码，并表示通过从“低廉”集合的节点中下载更多的块，可以减少修复故障节点的所造成的开销。Gastón 等<sup>[82]</sup>为一个采用再生码的含有 2 个机架（rack）的系统提出了类似的模型，考虑了访问机架间和机架内数据的不同访问成本和新加入节点的存储位置。Hu 等<sup>[83]</sup>引入了 DRC (Double Regenerating Codes)，最大限度地减少多机架数据中心的机架间流量。他们提出了一个两阶段的再生修复程序，在修复之前，将存储在机架内的数据进行重新组合，实验结果显示，与传统的再生码相比他们的方案有着 45.5% 的优化。

Sipos 等<sup>[84]</sup> 通过引入一个通用的计算框架，提前计算不同的可能的修复方案的开销，从而使得纠删码的修复方案能够及时感知实时的网络环境，做出相应的改变优化。

跨机架带宽往往会成为修复性能的瓶颈<sup>[85,86]</sup>，针对这个问题，Liu 等<sup>[87]</sup> 提出了 RPR (Rack-Aware Pipeline Repair Scheme) 技术，一种机架感知的管道修复方案，当存储系统发生单一故障或多个故障时，可以显著减少跨机架数据传输和总修复时间。从负载均衡方面看，RPR 对数据中心的有相应积极意义的。RPR 技术由三个部分组成：(1) 校验数据块的放置，当校验块产生时，按照提高解码速度的规则放置在相应机架内；(2) 机架内部部分解码，在数据块传输到修复节点或机架之前，首先进行部分解码，这就减少了跨机架的数据传输，提高了负载平衡；(3) 跨机架管道修复，在部分解码之后，由修复管道算法决定内部机架和跨机架传输的顺序，以实现最佳的修复性能。实验结果表明，与传统修复方法相比，单节点修复 RPR 减少了 81.5% 的修复时间，与 CAR<sup>[88]</sup> 相比修复时间减少了 50.2%，多节点修复总时间减少了 64.5% 且机架间传输流量比传统 RS 码修复减少了 50%。

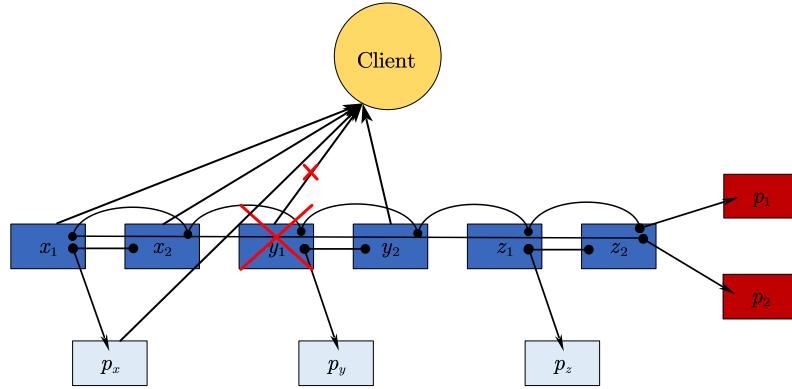
Shen 等<sup>[89]</sup> 将问题的维度从机架感知扩大到集群感知，他们发现在大规模集群存储系统中，修复变得非常复杂，同一集群内的节点之间因各种工作负载（如复制写入和 MapReduce 作业）而竞争的跨集群带宽往往被超额占用，而且带宽消耗比集群内带宽消耗更加严重。因此，产生大量跨集群修复流量的修复任务（即跨集群搜索数据进行修复）将大大延长修复过程，并消耗更多的修复时间。因此他们提出了 ClusterSR<sup>2</sup>，一种智能感知集群带宽变化的分散修复技术，旨在最小化和平衡跨集群的修复流量。ClusterSR 首先验证系统的数据分布，并给出每个故障块的修复方案（指定读取同一条带的存活数据和存储修复数据的节点），其主要目的是最小化跨集群修复流量。然后，ClusterSR 将执行多个数据块的修复，从而使产生的跨集群上传和下载流量在各集群之间保持平衡。ClusterSR 是第一个在分散修复中考虑最小化和平衡跨集群上传和下载流量的工作。他们在 Alibaba 的 ECS 服务器上进行了大规模验证实验，ClusterSR 可以减少 8.6-52.7% 的跨集群修复流量以及 14.4-68.8% 的集群内修复流量。

<sup>2</sup>开源地址：<https://github.com/shenzr/clustersr>

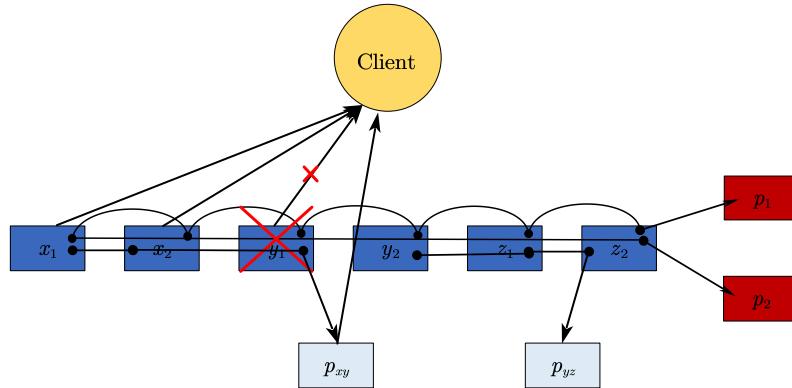
## 2.4 基于混合纠删码的数据修复技术

大多数分布式存储系统倾向于采用恢复单一的纠删码来存储数据，但事实上存储系统中数据的访问频率并不相同，存在着冷数据与热数据之分。改进方法便是采用恢复性能优秀的纠删码来存储热数据，存储开销低廉的纠删码来存储冷数据。例如，考虑 LRC(6,3,2) 和 LRC(6,2,2)，如果只使用其中任一单一编码都会造成更高的存储开销和恢复成本。定义 I/O 开销为被读取的数据量和用户请求的数据量之间的差值，差值越小，则开销越小。如图 2-3a 所示，假设 Client 请求  $x_1, x_2$  和  $y_1$ ，但是  $y_1$  不可用，有很多种方法可以重建  $y_1$ ，其中最高效的方法就是读取  $p_x$  和  $y_2$  来重建  $y_1$ 。在这个例子中，读取的数据块为 4，且 Client 请求的块数为 3，所以 I/O 开销为  $(4 - 3) = 1$  块。如图 2-3b 所示，假设 Client 在 LRC(6,2,2) 编码策略下请求  $x_1, x_2$  和  $y_1$ ，但是  $y_1$  不可用，所以读取  $p_{xy}$  来重建  $y_1$ 。在这个例子中，读取的数据块为 3，且 Client 请求的块数为 3，所以 I/O 开销为  $(3 - 3) = 0$  块。

LRC(6,3,2) 和 LRC(6,2,2) 的存储开销分别为  $(6+3+2)/6=1.83$  和  $(6+2+2)/6=1.67$ 。在相同的故障模式下，LRC(6,2,2) 的重建成本和存储开销比 LRC(6,3,2) 低。因此，在这种情况下，应该选择 LRC(6,2,2) 作为可靠性方案。然而，上面描述的场景是一个特例，存储系统的存储要求比这个场景更复杂。例如，一个应用程序可能一次读取一个或两个数据块，甚至不是一个完整的数据块。在图 2-3a 和图 2-3b 中，他们只考虑了块  $y_1$  不可用的情况，但所有数据块都有相同的不可用概率。受此启发，Wei 等<sup>[90]</sup> 决定为分布式存储系统设计一种新的自适应编码选择方法，并用 HDFS 进行了验证，实验结果显示该方法将存储消耗减少了 5%，修复带宽降低了 22%。这个方法根据数据的访问特性来选择合适的 LRC 方案。假设：用户请求的数据块是不可用的，则可以计算出最小修复成本，并选择具有最小修复成本的 LRC 作为暖数据的存储方案。若某份数据在过去一段时间内的被访问量极低，则冷数据编码模块就会启动 MapReduce 作业，用针对存储开销进行优化的 RS 码将 LRC 块转化为 RS 块，具体方法就是对于文件的每个编码条带的局部校验块  $p_x, p_y, p_z$  进行 XOR 运算，从而生成一个新的全局校验块  $p_0$ ，与原来的  $p_1, p_2$  组合并将  $p_x, p_y, p_z$  删除就将 LRC 转换为了 RS 码存储。但是，对于这个自适应方案仍有几个问题需要解决。首先，冷数据随着时间的推移可能会转换成暖数据或者热数据，应该针对这种情况配备对应的自适应的转换机制。第二，没有考虑缺失的奇偶校验块，尽管缺失的奇偶校验块并不直接影响修复性能，但会导致存储系统中 I/O 开销和网络带宽开销的增加，便间接地影响了修复性能。第



(a) LRC(6,3,2), Client 请求  $x_1$ ,  $x_2$  和  $y_1$  且  $y_1$  处于丢失状态, 所以 Client 访问  $p_x$  和  $y_2$  来重建  $y_1$



(b) LRC(6,2,2), Client 请求  $x_1$ ,  $x_2$  和  $y_1$  且  $y_1$  处于丢失状态, 所以 Client 访问  $p_{xy}$  来重建  $y_1$

图 2-3 LRC(6,3,2) 与 LRC(6,2,2) 不同重建方法

三, 分布式存储系统有不同的故障模式, 如块故障、磁盘故障和节点故障, 这里只针对了块故障。

目前比较成熟和前沿的混合纠删码技术一般是含有两种纠删码, 并对冷数据和热数据进行优化存储。其中的技术包含了在两种纠删码进行高效切换的算法, 在数据读取和写入过程中进行两种数据存储方式的转换, 此外还有采取优先级队列的方式来区别冷热数据并且配以相应的纠删码方案。Xia 等<sup>[63]</sup>采用两种纠删码来实现系统冗余, 两种编码属于同一种编码族, 用 Product Code 作为快速码 (Fast code) 为热数据来优化退化读和重建性能, 用 LRC 作为比较码 (Compact code) 针对冷数据来提供较为低廉的存储消耗。具体实现方式如 2-4所示。

HACFS 的自适应编码模块管理着系统状态。系统状态跟踪模块记录着每个纠删码构建的数据文件的以下文件状态: 文件大小、最后修改时间、读取次数和编码状态。文件大小和最后修改时间是由 HDFS 维护的属性, 并由 HACFS 用来计算文件的

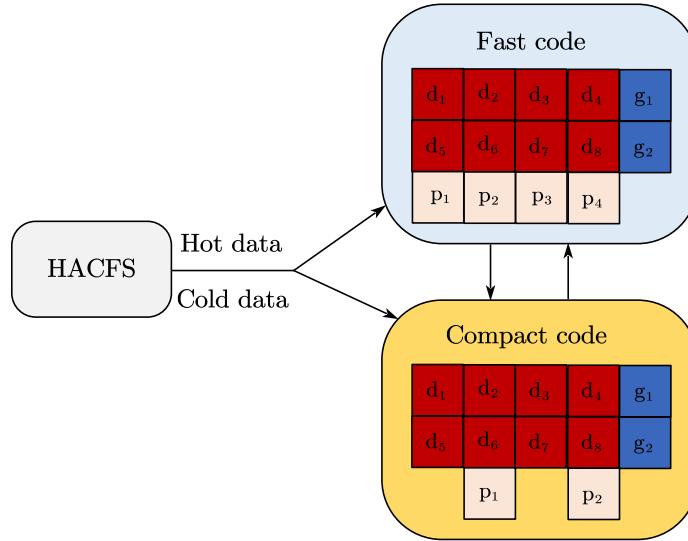


图 2-4 HACFS 混合纠删码的基本框架

总存储量和写入年龄。自适应编码模块还记录文件的读取次数，此外一个文件的编码状态代表了它是否被复制了三份或是否已经被编码。文件状态可以在 HDFS 客户端对该文件进行创建、读取或写入操作时被更新。

EC-Fusion 由 Qiu 等<sup>[91]</sup> 提出，他们将 RS 码和 MSR (Minimum Storage Regeneration) 码结合到一起，其中 RS 码具有较低的计算成本，为应用程序中的写请求提供了较低的开销。而 MSR 码的修复过程中，单盘故障恢复的数据量和复杂度较低，MSR 在降低网络传输成本进行恢复方面具有很大的优势，该技术将修复应用的速度提升了 0.77 倍，且修复时间降低了 69.10%。该方案包含三个主要特征：

(1) Code 选择的调整，EC-Fusion 将热数据划分为读取密集和写入密集型数据，然后又对数据的高低丢失风险做了相应的划分。冷数据或低风险数据采用 RS 进行编码，这保持了较高的存储效率，而且保证了良好的应用性能，特别是对于写入密集型 I/O 任务而言。对于具有高失效风险的读取密集数据，采取 MSR 编码可以节省大量的修复流量带宽，对应用程序的 I/O 产生很少的负面影响。

(2) 适应性规则，EC-Fusion 采取现有的缓存算法（如 LRU、LFU 等）来识别热数据或冷数据。在 EC-Fusion 中，两个队列分别用于访问模式和故障特征，并且现有的缓存算法可以应用于这些队列中。其采用两个队列，第一个队列中的每个块包含了块 ID、缓存命中次数，第二个队列的每个块包含块 ID、缓存命中次数和编码方案的标志。

(3) Code 的切换，在数据的属性保持动态变化的时候，如从热数据转换到了冷数据，高丢失风险数据变成了低风险数据。因为其本身数据保存时选择的编码方式不同，故相应的编码方式也要随着数据属性的变化而变化。即 RS 和 MRS 可以互相转换，其中 MSR 转 RS 如图 2-5 所示。

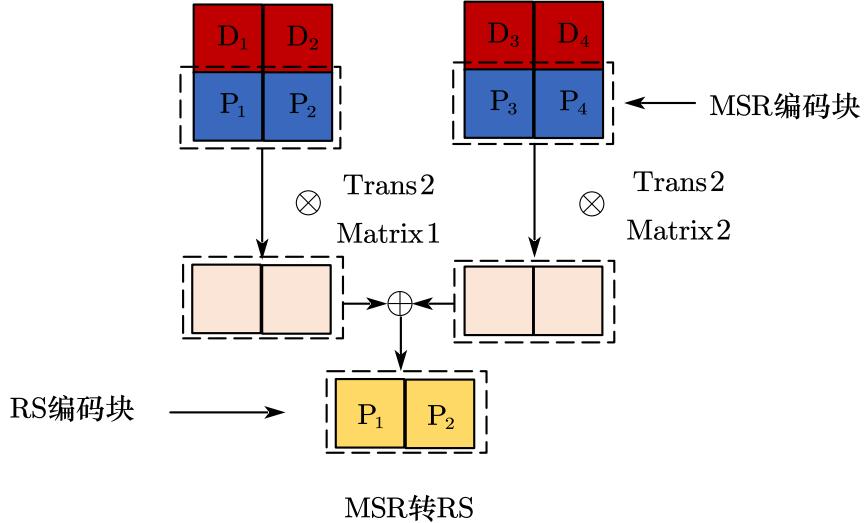


图 2-5 EC-Fusion 中 MSR 转 RS 过程

Wang 等<sup>[92]</sup>提出的这种策略使用 LRC (Local Reconstruction Code) 来存储热数据，用 HH 码 (Hitchhiker) 来存储冷数据，并且提出了相应的高效转换算法平衡两者之间的网络带宽与计算性能的消耗。HH 码和 LRC 均为 RS 码的一种改进，在使用  $(k, m)$ HH 码进行编码时，首先需要进行  $(k, m)$ RS 码的编码，再进行一系列 Piggybacking 框架中关于  $g_i(a)$  的异或运算；而在使用  $(k, l, g)$ LRC 进行编码时， $g$  个全局冗余块需要通过执行  $(k, g)$ RS 码的编码函数得到，而  $l$  个局部冗余块则只需要进行异或运算得到。另外  $(k, m)$ HH 码中  $g_i(a)$  的计算，需要将  $k$  个数据块分为  $(m - 1)$  组，而  $(k, l, g)$ LRC 中局部冗余块的计算，则需要将  $k$  个数据块分为  $l$  组。

从  $(k, t - 1, t)$ LRC 转换为  $(k, t)$ HH 码时，首先将子条带  $a$  中的各局部冗余子块读取并传输到对应的全局冗余块所在节点，并与该节点中子条带  $b$  的全局冗余子块进行异或运算，之后再将各局部冗余块删除即可。从  $(k, t)$ HH 码转换为  $(k, t - 1, t)$ LRC 时，首先计算出所有局部冗余块，之后再将其中子条带  $a$  中的局部冗余子块传到对应的全局冗余块所在节点，并与该节点中子条带  $b$  的全局冗余子块进行异或运算，使得这些子条带  $b$  的全局冗余子块不再包含子条带  $a$  中的额外冗余信息  $g_i(a)$ 。

## 2.5 基于预先修复的数据修复技术

由于磁盘故障预测技术的发展，磁盘故障的预测精度一般可以达到 95% (Botezatu 等<sup>[93]</sup>, Li 等<sup>[94]</sup>, Mahdisoltani 等<sup>[95]</sup>, Zhu 等<sup>[96]</sup>)。在高度准确的磁盘故障预测潜力的加持下，Shen 等<sup>[97]</sup> 在通过假设拥有完备的预测技术精准预测除出 STF (soon-to-fail) 节点前提下，耦合两种修复方法，即迁移与重建并提出 FastPR 算法，来对 STF 节点进行预先修复，其中（1）迁移，将 STF 节点上当前存储的数据块迁移到其他健康的节点上；（2）重建，通过检索存储集群中所有健康节点的数据块来重建（或解码）STF 节点的数据块，类似传统的反应式修复方法。解决了纠删码中固有的带宽和 I/O 放大问题，而重建则利用了所有健康节点的总带宽资源。如图 2-6 所示，假设系统使用的编码为 RS(5,3)，即  $n = 5, k = 3$ ，则一个条带中有 5 个块包含了数据块和校验块，当前有 7 个节点 ( $M = 7$ )，其中 6 个健康节点和 1 个 STF 节点。设  $N_i$  为第  $i$  个可用节点， $S_i$  是要修复的 STF 节点的第  $i$  个块的条带。假设现在得到了一个 STF 节点的  $c_m$  和  $c_r$  的集合，其中  $c_m$  为迁移的块的个数， $c_r$  为重建的块的个数。对于这样的通常情况需要解决两个基本问题，其一是在给定的通过重建修复的 STF 节点的  $c_r$  块中，需要确定从  $k - c_r$  个可用节点中检索出  $k - c_r$  个块，可将  $k - c_r$  的搜索问题表述为一个二分图最大匹配问题。其二是需要确定存储  $c_m + c_r$  个修复后的块。一般而言，将修复的块存储在  $c_m + c_r$  个现有的可用节点中，这样就可以保持节点的容错性，即任何  $n - k$  个节点的故障都是可以容忍的，再次将选择  $c_m + c_r$  个可用节点表述为二分图最大匹配问题。

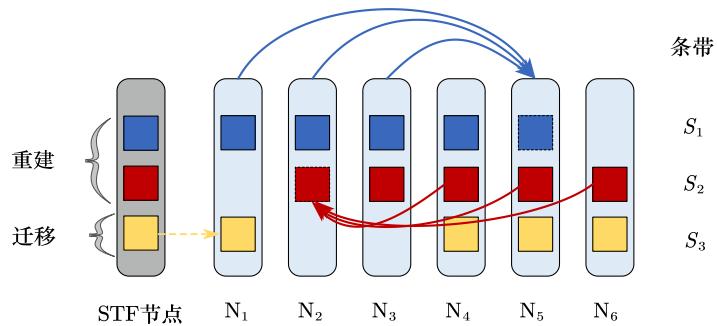


图 2-6 RS(5,3) 编码系统的 FastPR 修复流程

根据访问频率，数据一般可以被区分为热、暖和冷数据。一般来说，与热数据中的内容相比，暖数据的请求率较低，且可以通过历史访问数据和用户偏好从而预测数据访问热度，为了从用户角度减少访问时间，Cao 等<sup>[98]</sup> 融合了两种机器学习算法

——预测和推荐，提出了新的一种预先修复技术 POST。针对暖数据，他们用一种独特的方式进行处理，以优化系统性能和存储空间的利用。如图 2-7 所示，为了使得故障的数据节点在响应 I/O 请求瞬时被重建，他们采用了两种机器学习算法，其中包括用 K-prototype 聚类算法和基于贝叶斯的文本分析算法将文件进行聚类，在数据重建过程中应用推荐算法优先修复访问量更高的文件。他们提出的系统依赖于一个聚类机制，将文件分为多个集群，在每个集群中，文件都有类似的特征。对于预测模块，通过跟踪历史数据的 I/O 访问，预测未来的访问热度较高的文件集，这个访问热度较高的文件集反过来又提供了对可能接下来被访问的文件进行预测。预测模块负责计算用户之间的相似性，从而设定要重建的数据块的优先级别。实验结果表明，POST 系统加快了并行存储系统的数据恢复，同时为在线用户保持了较高的数据访问性能。

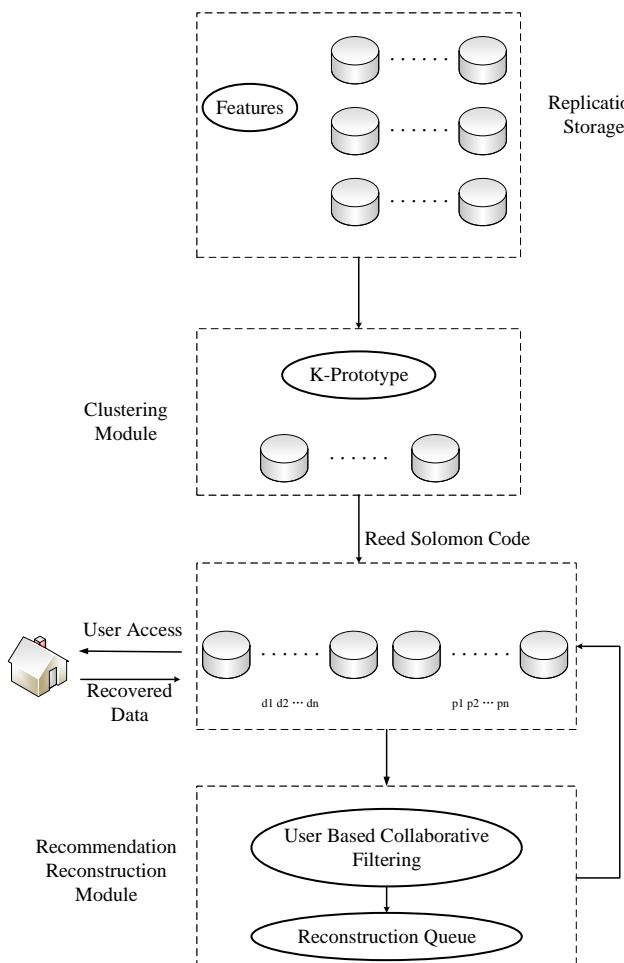


图 2-7 在 POST 的架构设计中，当带有特征的暖数据来自 3 倍复制的存储系统时，集群会通过 K-Prototype 算法将数据存储到基于纠删码的存储系统中，当用户在访问数据出现节点故障时，推荐系统会根据用户设置提供推荐列表，并为存储系统中的每个条带提供重建列表

此外，还有一些其他修复技术。针对纠删码数据块的随机分布问题，Xu 等<sup>[99]</sup>提

出了  $D^3$  (Deterministic Data Distribution)，一种数据分布策略，其可以在节点之间均匀地分发数据块和校验块，以及相应的基于  $D^3$  的高效数据修复算法。以往的研究很多忽略了数据块、校验块和文件之间的复杂关系，例如同时有 5 个块在进行重建，且 5 个块属于 5 个不同的文件，那么此时 5 个文件都将无法访问。针对此问题，Zeng 等<sup>[100]</sup> 提出了一种带有文件感知的图修复技术 FAGR (File-aware Graph Recovery Scheme)，从而提高重建过程中的文件修复速度。FAGR 的思想建立一个文件、数据块、条带、校验块、节点和文件访问频率之间的映射图，并从文件的角度引导数据修复过程。为了证明 FAGR 的有效性，他们在集群中进行了一些数值分析和实验。结果表明，与典型的快速修复方法相比，FAGR 将文件的平均响应时间减少了 81.63%，吞吐量提高了 4.44 倍。

## 2.6 本章小结

本章主要介绍了基于纠删码构建的存储系统的数据修复技术研究现状，研究者在包括但不限于编码结构、调度优化、带宽优化、混合纠删码、预先修复等领域探索提升数据修复技术的研究。

## 第三章 基于预先重建迁移的数据修复技术研究

### 3.1 引言

分布式存储系统通过对数据进行冗余来保证数据的可靠性和可用性，基于纠删码的冗余技术能减少存储空间。在存储集群中利用多个节点来提供足够的数据冗余以降低数据丢失的可能性。多备份冗余是创建了至少三份数据副本，但是在数据存储急剧增长的情况下，备份冗余会引发大量存储开销，早期的分布式集群一般采用这种方式。纠删码则是通过编解码矩阵的计算创建了远小于多备份技术的冗余数据，同时提供同级别的容错性<sup>[6]</sup>。如今的大规模存储群越来越多地采用纠删码<sup>[7-10]</sup>，以取得存储空间和可靠性的均衡。

纠删码面临的最基本问题就是过量的修复开销：修复流量随着存储冗余度的降低而增加<sup>[44]</sup>。因此，前人对改善纠删码的修复性能进行了广泛的研究，使修复过程中的修复流量或 I/O 最小化 (Sathiamoorthy 等<sup>[3]</sup>, Huang 等<sup>[8]</sup>, Dimakis 等<sup>[44]</sup>)，或设计适用于所有实践中的纠删码包括 RS 码的提高修复效率的技术 (Mitra 等<sup>[71]</sup>, Li 等<sup>[72]</sup>, Shen 等<sup>[88]</sup>, Bhagwan 等<sup>[101]</sup>, Li 等<sup>[102]</sup>, Silberstein 等<sup>[103]</sup>)。目前，大部分的传统修复方法都是被动修复，只有在检测到节点故障后才会触发修复操作。如果可以提前预测即将发生的故障，就可以在任何实际故障发生之前主动修复任何即将发生的节点故障，以提高系统可靠性。

随着机器学习技术的不断发展，很多研究也将其与磁盘预测技术结合在一起进而提高预测精准度。在某些情况下，预测精度甚至可以达到至少 95%<sup>[93-96]</sup>，并且误报率非常小。故 Shen 等<sup>[97]</sup> 定义了 STF 节点为根据预测技术得到的精确的即将故障的节点，在这前提下通过耦合两种修复方法——迁移和重建提出了 FastPR 算法，准确定位 STF 节点并加速修复操作。然而在修复调度上，FastPR 在带宽快速变化的网络环境中，尤其是在带宽不对等的环境下修复性能会受到限制。

本文在 FastPR 对于 STF 节点修复任务的前提下，进一步将其扩展到网络环境快速变化的情形中。利用空闲节点加速数据的传输并感知当前网络环境的变化，改进预先修复算法使其适应快速变化的网络环境，提出了划分迁移集和重建集算法 SMSRS，并引入多级中继传输概念提出了改进的多级传输调度算法 ISA 加速修复任务数据的传输。

## 3.2 预先修复问题建模

本文通过结合迁移与重建来设计实现预先修复的机制，用以修复 STF(Soon To Fail)<sup>[97]</sup> 节点。这两种修复方式各有其优缺点，迁移是直接从一个 STF 节点上读取存储的编码与数据块，将其重新传输到一个或多个正常的节点中，与常规读取相比，它并不会引发额外的带宽消耗。然而，迁移的性能却受到 STF 节点可用带宽的限制，另一方面，重建修复遵循着传统的反应式修复，亦即只有当节点数据不可访问时才会触发修复机制，通过从可用节点中检索多个条带来重建 STF 节点的数据条带。由于这些数据与编码条带通常分布在存储集群中，可以利用存储集群的可用带宽资源，让所有可用节点以并行的方式参与 STF 节点的多个条带的修复，但是会引发额外的带宽消耗。

### 3.2.1 问题描述

以 RS(5,3) 为例，其中  $n = 5$  和  $k = 3$ ，设  $N_i$  为第  $i$  个可用节点， $S_i$  是要修复的 STF 节点的第  $i$  个块的条带。假设现在得到了一个 STF 节点的  $c_m$  和  $c_r$  的集合，其中  $c_m$  为迁移的块的个数， $c_r$  为重建的块的个数。如图 3-1 所示的存储集群，总共有  $M = 7$  个节点，其中  $M - 1 = 6$  个可用节点， $N_1, N_2, \dots, N_6$ ，而 STF 节点的块对应于条带  $S_1, S_2, S_3$ 。

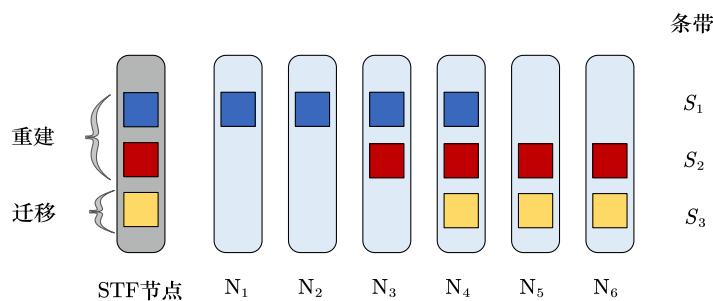


图 3-1 RS(5,3) 的条带分布

Shen 等<sup>[97]</sup> 提出，对于这样的情况通常需要解决两个基本问题，其一是在给定的通过重建修复的 STF 节点的  $c_r$  块中，需要确定从  $k - c_r$  个可用节点中检索出  $k - c_r$  个块，将  $k - c_r$  的检索选择表述为一个二分图最大匹配问题。其二是需要确定存储  $c_m + c_r$  个修复后的块。一般而言，将修复的块存储在  $c_m + c_r$  个现有的可用节点中，这样就可以保持节点的容错性，即任何  $n - k$  个节点的故障都是可以容忍的，再次将选

择  $c_m + c_r$  个可用节点表述为二分图最大匹配问题。

然而在实际分布式网络修复中，不仅仅只关注需要用到哪些块进行修复与存储，其中并行的性能还受到节点之间的快速变化的带宽和文件访问频次的影响。首先按照在一个存储集群中对于不同种类的数据访问频次的不同，包含相应文件的节点宕机造成的损失要远高于存放那些不常用的数据的节点宕机所造成的损失，这种情况在预先修复中就需要纳入考虑，提高对热数据的修复顺序的权重可以极大地增加存储集群的可靠性与持久性；其次，对于一个 STF 节点而言，存储着来自不同数据的数据块与校验块，如何确定当前所需要的块的修复方法也是至关重要的。在快速变化的网络环境中，可用节点并行地处理修复任务与迁移任务，不同节点的带宽瓶颈不同，如果在修复过程中随机选择迁移或者重建方法，势必会造成整个网络修复性能得不到充分利用，或者某个节点超出负荷的运行而剩下的节点却处于空闲状态，需要设计一个算法充分利用整个网络的带宽进行重建与迁移，其中至关重要的就是对于一个 STF 节点确认其重建集  $c_m$  与迁移集  $c_r$ 。

问题转化为计算出  $c_m$  和  $c_r$  的集合，亦即确定重建集和迁移集的具体块。有如下具体问题，若一个 STF 节点拥有  $m+r$  个数据块，每个块对应的文件有着相应的访问热度  $T_i (1 \leq n \leq m+r)$ ，且在可用节点中， $w_{nj}$  表示  $n$  节点到  $j$  节点实时带宽速度，每个节点拥有  $t$  个数据块，分别为  $[N_s, \dots, N_k, \dots, N_t] (s < k < t)$ ，以及每个节点是否处于参与修复任务状态位  $State_n$ ，取值为  $\{T, F\}$ ，现在需要确定当 STF 节点产生时，其中的数据块通过耦合重建和修复达到修复时间最短的方案。

### 3.2.2 模型假设

修复技术的主要目标就是减少修复时间，并且最大化地利用实时带宽。为此我们做出如下假设：

- (1) 存储集群每次最多只有一个 STF 节点，这是因此单节点修复是最主要的修复事件，且占据修复事件总数的 98%<sup>[104]</sup>；
- (2) 随着系统运行，配备的预测算法预测出的 STF 节点的成功率是 100%；
- (3) STF 节点中的分块在修复过程中仍然可以访问，直到 STF 节点实际发生故障；
- (4) 在多次修复后，块分布可能会变得不平衡，假设集群拥有自动再平衡的能力。

### 3.2.3 理论修复时间

设  $M$  为存储系统中的节点总数,  $U$  为 STF 节点需要被修复的块的总量。设  $x$  是被迁移修复的块的数量, 则  $U - x$  是重建修复的块的数量。对于迁移修复,  $t_m$  为将一个块从 STF 节点迁移到另一个正常节点的时间, 因此总的迁移时间为  $x \cdot t_m$ 。对于重建修复, 令  $t_r$  为修复 STF 节点的一个块的时间。在  $RS(n, k)$  的设置下,  $k$  个正常节点需要搜索  $k$  个块来重建 STF 节点的每个块, 若  $M - 1$  个正常节点数远大于  $k$ , 则可以同时重建 STF 节点的多个块。假设重建修复可分为多轮, 那么在每一轮中, 可以找到  $G \leq \frac{M-1}{k}$  个不重叠的组, 这些组中的  $k$  个节点属于不同的条带, 并可以从这些组并行地搜索出  $k$  个块。因此, 可以在  $t_r$  时间内通过重建修复 STF 节点的  $G$  个块, 重建消耗的总时间为  $\frac{U-x}{G} \leq t_r$ 。

令  $T(x)$  为预先修复的总时间, 又重建和迁移是并行执行的, 则:

$$T(x) = \max\left(x \cdot t_m, \frac{U-x}{G} \cdot t_r\right) \quad (3.1)$$

当  $x \cdot t_m = \frac{U-x}{G} \cdot t_r$  时,  $T(x)$  最小, 也就是  $x = \frac{U \cdot t_r}{G \cdot t_m + t_r}$ 。因此, 最小的预先修复时间  $T_P$  为:

$$T_P = \frac{U \cdot t_r \cdot t_m}{G \cdot t_m + t_r} \quad (3.2)$$

对于反应式修复只会进行重建, 则总的修复时间为  $T_R$  则为式 3.1 中, 当  $x = 0$  时的  $T(0)$ :

$$T_R = \frac{U \cdot t_r}{G} \quad (3.3)$$

### 3.2.4 迁移与重建时间建模

本文将修复过程分解为三个部分, 并按照顺序方式进行。(1) 读取, 从本地存储中读取数据块; (2) 传输, 通过网络进行数据传输; (3) 写入, 将修复后的数据写入正常节点。设  $b_d$  和  $b_n$  分别为磁盘和网络带宽,  $c$  是块大小。首先对于迁移任务, 对于每个块的读取时间为  $\frac{c}{b_d}$ , 传输时间为  $\frac{c}{b_n}$ , 写入时间为  $\frac{c}{b_d}$ , 则  $t_m$  有:

$$t_m = \frac{2c}{b_d} + \frac{c}{b_n} \quad (3.4)$$

对于重建任务， $k$ 个正常节点可以并行地读取同一条带的块，故对每一个待读取块的读取时间为 $\frac{c}{b_d}$ 。因为接收 $k$ 个块的节点一个时间步只能接收一个块，故所有 $k$ 个块的传输时间为 $k \cdot \frac{c}{b_n}$ ，正常节点接收了所有 $k$ 个块后进行矩阵计算（这里忽略计算时间）得出一个数据块写入正常节点，故写入时间为 $\frac{c}{b_d}$ ，因此 $t_r$ 有：

$$t_r = \frac{2c}{b_d} + \frac{k \cdot c}{b_n} \quad (3.5)$$

设 $M = 100$ ,  $U = 1,000$ , 块大小 $c = 64MB$ ,  $b_d = 100MB/s$ , 且 $b_n = 1Gb/s$ 。考虑RS(9,6), RS(14,10), RS(16,12)三种分布式存储系统中的常见RS码配置。如图3-2所示，当节点数较少、 $k$ 较大、 $b_d$ 较大、 $b_n$ 较小时，分别对应了3-2a, 3-2b, 3-2c, 3-2d四张子图，预先修复的性能要反应式修复更加高效。总的来说，预先修复在所有情况下都减少了反应式修复的修复时间，例如，在RS(16,12)中减少了33.1%（图3-2b）。

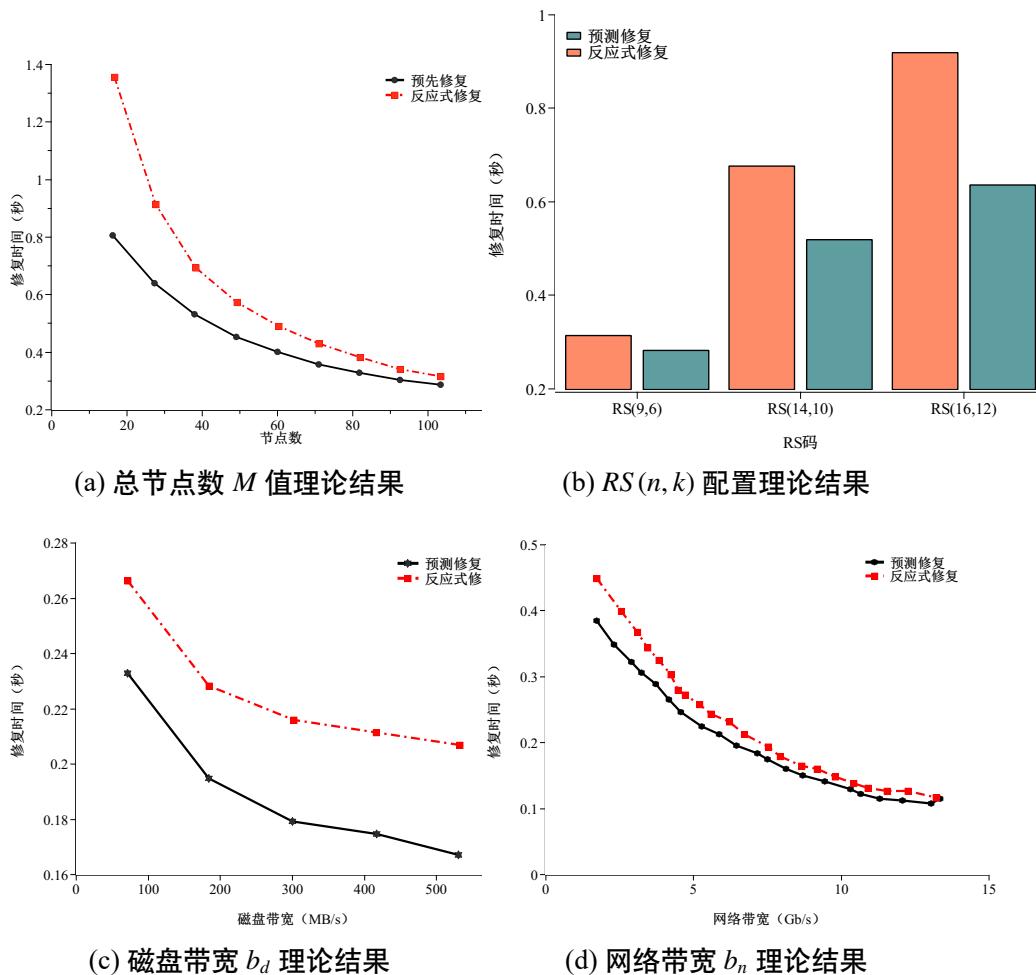


图3-2 预测修复与反应式修复建模对比

### 3.3 预先修复集合划分算法

在 FastPR<sup>[97]</sup> 中，算法的目标是最大限度地在每轮修复中增加 STF 节点的块的数量。假设 STF 节点中的所有块都要通过重建方法进行修复，即从  $k$  个正常节点中获取  $k$  个块，这样最多可以从  $M - 1$  个正常节点每个节点获取一个块，这样就可以在一个修复轮次中完成修复任务。所以为了提高并行性，应该最大化重建集的块数量，如章节 3.2.3 所说，最大为  $\frac{M-1}{k}$ 。

如图 3-3 所示，本文针对重建集与迁移集问题，设计了划分迁移集与重建集算法 SMSRS（Split Migration Set And Reconstruction Set）算法，如算法 1 所示。该算法主要使用数据热度队列与最小堆技术，根据数据访问热度高低处理待修复的块的出队修复顺序，利用最小堆不断优化当前出堆的网络带宽性能最优的可用节点进行迁移修复，然后通过贪心策略确定最佳的重建节点以及对应的重建块接收节点。

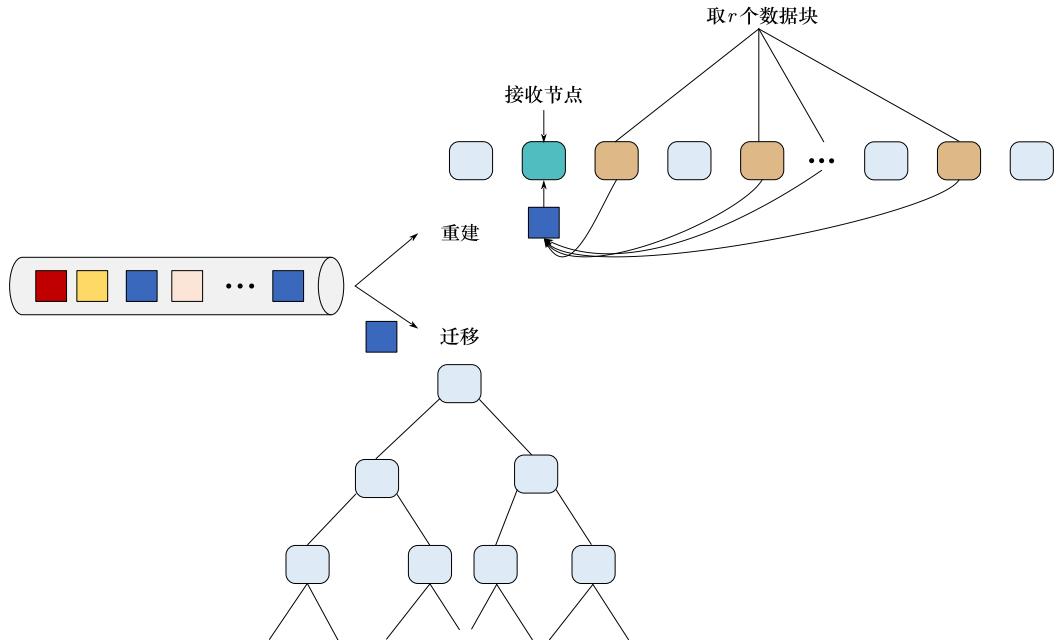


图 3-3 重建与迁移修复流程

在获得迁移集  $c_m$  和重建集  $c_r$  中，需要先根据文件访问热度对相应文件的数据块进行排序送入队列  $Q$ （第 1~2 行），按照热度从高到低的顺序从队列  $Q$  中弹出 STF 节点中待修复的块，将其对应的节点编号按照与可用节点当前连接带宽的大小进行小堆化，这样可以保证每次在堆顶取出当前与 STF 节点连接带宽速度最快的节点进行修复（第 4~5 行）。在可用节点中找出当前不含有该 STF 节点待修复块的节点集

$S_1$ , 并且不断从堆顶取出节点编号, 判断该节点是否处于执行修复任务状态以及是否包含于节点集  $S_1$ , 若是则跳出, 然后将不满足条件但弹出堆的数据再放回堆中, 并计算通过迁移块修复所耗费的时间  $t_m$  (第 6~16 行)。在可用节点中找出当前含有该 STF 节点待修复块的节点集  $S_2$ , 并且定义带宽集  $\mathbf{D}$ , 表示的是节点集  $S_2$  与可用节点集  $\mathbf{N} - S_2$  中带宽各个节点传输的实时带宽值, 在节点集  $S_2$  确定  $r$  个节点, 并在可用节点集中找出与这  $r$  个节点连接带宽值最大的节点  $RNode$ , 然后得出  $r$  个节点中与  $RNode$  连接最慢的节点除以块大小即得出对应的通过重建块修复所耗费的时间  $t_r$  (第 17~30 行)。最后通过判断  $t_r$  和  $t_m$  的大小来确定重建集和迁移集, 当队列  $Q$  为空时, 则结束整个算法, 返回相应的迁移集  $c_m$  和重建集  $c_r$  (第 31 行 ~39 行)。

### 3.4 预先修复调度算法

常用的调度算法为单节点传输算法, 亦即从源头节点直接传输数据到目标节点。但是这种方式往往会忽略通过高链路带宽传递数据带来的更快的修复速度的优势。进而引出了多级传输的技术概念, 亦即源节点数据在传递到源节点之前经过中继节点的转发, 可以在更好地利用中间节点的高带宽优势的同时提高传输的并行性。如 PPR<sup>[71]</sup>, RP<sup>[72]</sup>, PPT<sup>[73]</sup> 等技术都利用了这种技术特性提高了调度速度。

受 Zhou 等<sup>[74]</sup> 启发, 本文提出了 ISA (Improved SMFRepair Algorithm) 算法来进行节点修复的调度。如图 3-3 所示, 考虑 RS(5, 3), 修复一个失效的节点需要从  $k$  个 helper 节点钟获取  $k$  个数据块, 其中 helper 节点由平均带宽进行确定。对于  $n$  个节点  $N_1, N_2, \dots, N_n$ ,  $N_i$  的平均带宽就是  $Ave_B W_i = \frac{\sum_{j=1, j \neq i}^n \text{Bandwidth}_{Ni-Nj}}{n-1}$ 。例如, 根据表 3-1<sup>[74]</sup> 图 3-3 中的  $D_2$  的所在节点的平均带宽为  $(2 + 5 + 3 + 2)/4 = 3MB/s$ 。可以计算出所有  $M - 1$  个节点的平均带宽, 并选择平均带宽最大的  $k$  个节点作为 helper 节点。平均带宽可以更直观地显示节点的传输数据的能力, 平均带宽越大, 那么低带宽的情况就会相对少见。

在这个单节点修复的多级传输问题中, 需要确定三个基本要素, 即 helper 节点、链路选择和传输方向的确定。在图 3-3 中, 故障节点  $N_f$  中有块  $D_1$ , 其中块的大小为  $20MB$ 。当  $D_1$  丢失时, 平均带宽最大的  $k(k = 3)$  个正常节点为  $D_3, P_1, D_2$ , 故 helper 节点确定。在时间步 1 中, 通过排列组合有三种修复方案, 其中第三个方案的最低的链路在三个方案的最低中最大, 因此时间步 1 的链路选择方案确定, 即  $P_1 \rightarrow D_1, D_2 \rightarrow D_3$ , 其中方向分别是因为  $D_1$  为丢失的块和  $D_3$  的平均带宽大于  $D_2$ 。若选择了  $D_3 \rightarrow D_2$ ,

---

**算法 1 SMSRS 算法**

---

**Input:** ChunkSize,  $m$ ,  $r$ ,  $\mathbf{w} = \{w_{i,j}\}$ ,  $\mathbf{N} = \{N_i\}$ ,  $\mathbf{State} = \{State_i\}$ ,  $\mathbf{T} = \{T_i\}$ ;

**Output:**  $c_m, c_r$ ;

```

1:  $c_m, c_r, TempSet = \emptyset$ ;
2:  $Q \leftarrow \text{SortedByAccessHeat}(m + r, \mathbf{T})$ ;
3: while true do
4:   STFNodeChunk =  $Q.pop()$ ;
5:   WHeap  $\leftarrow \text{HeapIFY}(j, \text{key} = w_{STFNodeChunk,j} (j \in \mathbf{N}))$ ;
6:    $S_1 = \text{FindSTFNodeChunkNotIn}(\text{STFNodeChunk}, \mathbf{N})$ ;
7:   while true do
8:     AvailableNode =  $WHeap.pop()$ ;
9:     if  $STATE_{AvailableNode} == T$  and  $AvailableNode \in S_1$  then
10:      break;
11:    else
12:      TempSet  $\leftarrow TempSet \cup AvailableNode$ ;
13:    end if
14:   end while
15:    $WHeap.push(TempSet)$ ;
16:    $t_m = \text{ChunkSize} / w_{STFNodeChunk, AvailableNode}$ ;
17:    $S_2 = \text{FindChunkIn}(\text{STFNodeChunk}, \mathbf{N})$ ;
18:    $\mathbf{D} = \emptyset$ ;
19:   for  $n_1 \in S_2$  do
20:     DTemp =  $\emptyset$ ;
21:     for  $n_2 \in \mathbf{N} - S_2$  do
22:       DTemp  $\leftarrow DTemp \cup w_{n_1, n_2}$ ;
23:     end for
24:      $\mathbf{D} \leftarrow \mathbf{D} \cup DTemp$ ;
25:   end for
26:    $TNode, RNode \leftarrow \text{FindMaximizedBandwidthTransferAndReceiveNode}(\mathbf{D})$ ;
27:   for  $n \in TNode$  do
28:     Bdh =  $\text{MIN}(w_{n, RNode})$ ;
29:   end for
30:    $t_r = \text{ChunkSize} / Bdh$ ;
31:   if  $t_m < t_r$  then
32:      $c_m \leftarrow c_m \cup \text{STFNodeChunk}$ ;
33:   else
34:      $c_r \leftarrow c_r \cup \text{STFNodeChunk}$ ;
35:   end if
36:   if Length( $Q$ ) == 0 then
37:     break;
38:   end if
39: end while
40: return  $c_m, c_r$ 

```

---

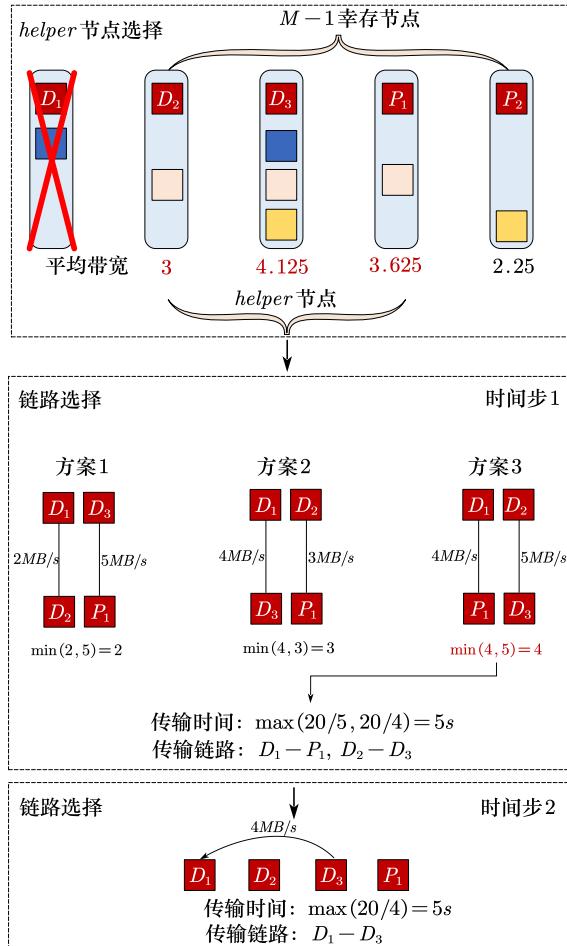


图 3-4 helper 节点选择和链路选择过程图

那么在低链路带宽就会留到时间步 2，相较而言造成的影响更大。当故障节点接收到块之后，就可以通过解码参数来计算丢失的  $D_1$ ，也就是  $D_1 = xD_2 + yD_3 + zP_1$ 。虽然在传递过程中，已经通过绕过低链路和提高并行速度的方式来进行，但是带宽瓶颈的问题依然存在，并没有达到实时带宽的最优。

通过加入空闲节点 (idle nodes) 进行数据中继的方式可以对传输问题进一步优化。如图 3-5 所示，空闲节点中不包含同一条带中的数据块或者包含其他条带的数据块，在修复过程中处于空闲状态，没有进行其他修复任务的传输。可以看出通过空闲节点的中继，有效减少了传输时间，原本的  $P_1 \rightarrow D_1$  需要 5s，然而通过  $I_1$  进行中继后的  $P_1 \rightarrow I_1 \rightarrow D_1$  只需要 4s。为了简化问题，假设每个空闲节点同时只能执行一种转发任务，在多级转发过程中，需要源头节点和目标节点以及中继节点集合，问题的目标是找到一条消耗时间最短的链路路径，例如图 3-5 中的  $P_1, D_1$  和  $I_1, I_2$  组成的集合，且这样的组合路径可以用树来进行表示。 $P_1$  节点则为树根， $D_1$  为叶子节点，空闲节

表 3-1 7 个节点测量带宽 (MB/s)

带宽	$D_1$	$D_2$	$D_3$	$P_1$	$P_2$	$I_1$	$I_2$
$D_1$	*	2	4	4	2	8	4
$D_2$	2	*	5	3	2	10	5
$D_3$	4	5	*	5	2.5	10	5
$P_1$	4	3	5	*	2.5	10	5
$P_2$	2	2	2.5	2.5	*	5	8
$I_1$	8	10	10	10	5	*	5
$I_2$	4	5	5	5	8	5	*

点的集合则可以表示为非叶节点，其中任何一条从树根到叶子节点的路径则为一条转发链路，假设共有  $x$  个非叶节点，则一共有  $C_x^1 + C_x^2 + \dots + C_x^x$  种转发链路的可能。当  $x$  变得很大的时候，寻找路径的算法复杂度将会非常高。针对此问题 Zhou 等<sup>[74]</sup> 提出了可以绕过低带宽的链路，更快地将数据传输到目标节点。故而提出了带宽感知的多级调度算法 SMFRepair (single-node multi-level forwarding repair algorithm)。

SMFRepair 不仅可以运用在重建技术上，本文在该算法的基础上将其运用到了迁移技术上并进行了改善。该算法基于 PPR 技术，改善了 PPR 容易受到异构环境中低链路带宽的影响，提高每轮修复的效率，从而在整个修复过程中达到最优。图 3-6 显示了具体的优化过程，假设每个节点只能协助转发一次（避免性能瓶颈和逆向转发），整个转发过程包括两个节点和一个集合。在重建任务中两个节点是源节点  $N_1$  和目标节点  $N_2$ ，闲置节点则是含有相同条带的其他块的节点的集合，称之为转发中间节点集。在迁移任务中，两个节点的源节点为 STF 节点，目标节点为不含有相同条带的其他块的节点，例如  $N_2$ ，闲置节点则为所有可用节点，定义为迁移任务的中间集。优化的目标就是找到一条从源节点出发，经过中间集，到达目标节点的最短路径。其中，从源节点到中间集以及中间集到目标节点的链接是单向的，中间集中的每个节点都是双向互通的，但每个节点只能通过一次。

对于纠删码结构为  $RS(n, k)$  的分布式存储集群来说，当某个节点失效，相应的数据块不可访问时，对于重建任务而言，一般情况下需要在剩下的  $M - 1$  个节点中检索  $k$  个从属于同一个条带的数据进行数据修复，通常会有两种情况（结合图 3-6 进行阐述）：

- (1) 从节点  $N_1$  检索到一个块，从其传输到节点  $N_3$ ，同时节点  $N_3$  也含有相应的

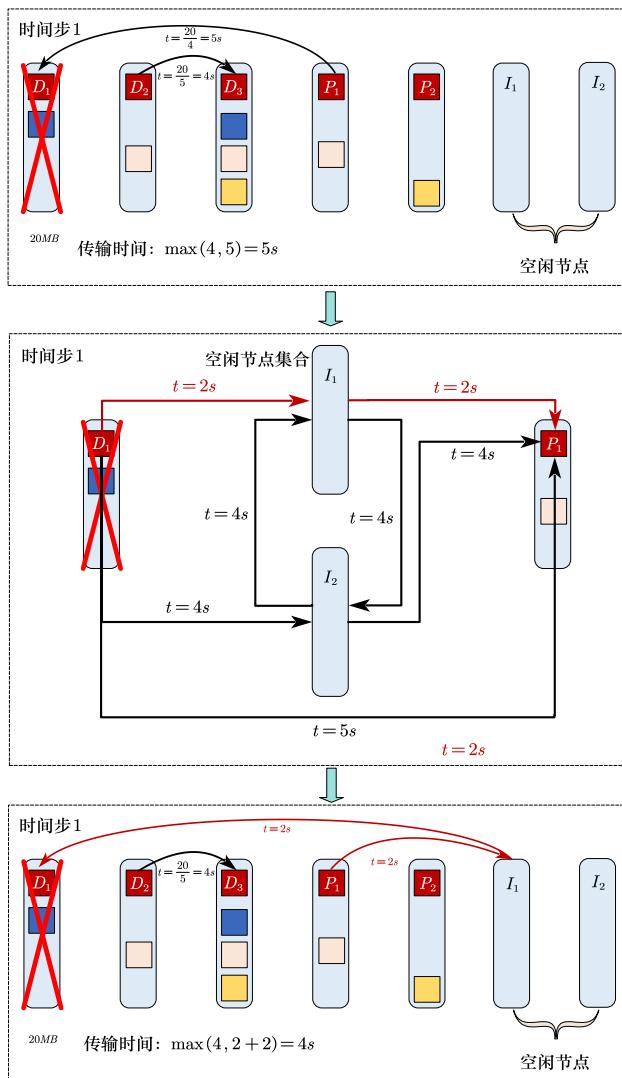


图 3-5 结合空闲节点的链路传输优化过程图

从属于同一个条带的数据块，此时节点  $N_3$  含有两个可以传输的同一个条带的数据块，将其传输到节点  $N_2$  进行修复，总共花费的时间为  $t_1 + t_2$ ，小于直接从节点  $N_1$  传输数据到节点  $N_2$  所花费的时间  $t_3$ ；

(2) 从节点  $N_1$  检索到一个块，从其传输到节点  $N_4$ ，同时节点  $N_4$  并不含有相应的从属于同一个条带的数据块，可能含有其他数据块或者没有，然后从节点  $N_4$  将数据块转发到节点  $N_3$ ，之后的流程与情况 (1) 相同，而花费的总时间为  $t_4 + t_5 + t_2 < t_1 + t_2$ ，即  $t_4 + t_5 < t_1$ ，在这种情况下，在两个均含有相同条带的数据块的节点传输数据的速度慢于先将数据块传输到空节点（或者含有其他条带的数据块）再转发到另一个含有相同条带的数据块的节点的速度。对于迁移任务而言，一般只有一种情况需要考虑：

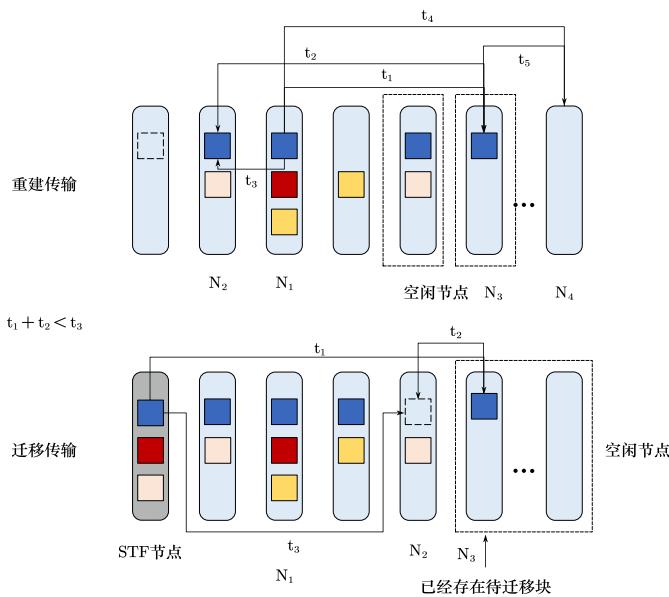


图 3-6 重建与迁移的中继传输

当有 STF 节点产生时，则需要将数据块迁移到不存放该条带下数据块的节点  $N_2$ ，首先将其传输到已经含有相同条带下数据块的节点  $N_3$ ，然后再从节点  $N_3$  将数据块转发设计的目标节点  $N_2$ ，所消耗的总时间为  $t_1 + t_2$ ，且小于直接将数据块从 STF 节点传输到目标节点  $N_2$  的时间  $t_3$ 。

针对上述情况的特点，节点传输过程主要通过类似最短路的机制来减少整个修复所消耗的时间。可以通过计算最短路径的 Dijkstra 算法 ( $O(N \log N)$ ) 求得最优的转发修复方案。如图 3-7 所示，主要是通过建立一个图结构来描述问题，图的源节点为  $N_1$  或  $N_2$ ，对于重建任务需要从含有该块的节点出发，则源节点为  $N_1$ ，对于迁移任务需要从 STF 节点出发，经过若干次跳跃，最终节点一定是未包含该条带的节点，所以源节点是  $N_2$ 。从源节点开始，逐个计算每条路径消耗的权重时间，可以通过剪枝来加快算法的运行，即对于某些情况下，路径消耗一定是大于当前已经遍历的情况，则在算法运行中不纳入考虑。

因此本文针对迁移与重建问题的调度特性，设计了改进的 SMFRepair 算法 (IBA, Improved SMFRepair Algorithm) 算法。原算法在修复过程中总是构建当前带宽的全局最优路径，使用的是树的搜索算法，由于本文需要解决的是重建与迁移交融的修复任务，所以在寻找最优路径的过程中使用的是 Dijkstra 算法。IBA 在每一轮修复中首先判断该任务是迁移还是重建，根据输入参数 type 进行确认，紧接着采用局部最优修复（第 4~11 行），在此过程中链路带宽不断发生变化，从而实现最优化。详细算法

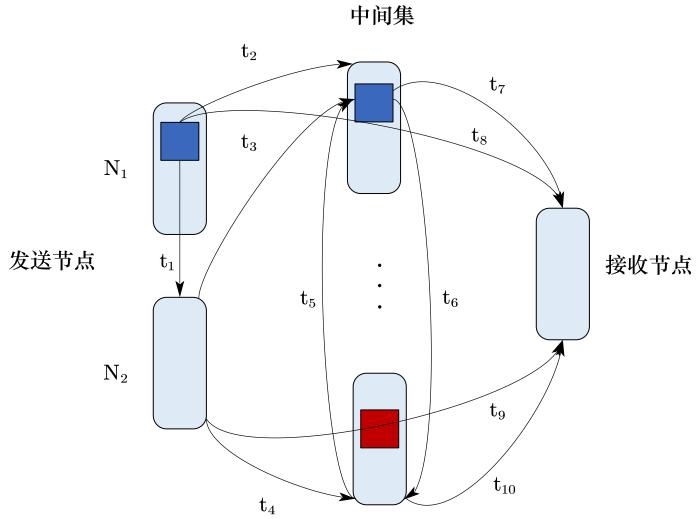


图 3-7 节点中继的流程

如算法 2 所示。

### 3.5 实验结果与分析

本文对 SMSRS 和 ISA 算法进行了仿真实验，以评估其在大规模存储集群中的性能。基于 SimEDC<sup>[105]</sup> 本文为算法原型为设计了一个仿真平台。SimEDC 是一个用于模拟离散事件程序集，其主要功能是用来特征量化基于纠删码构建的数据中心的可靠性。它旨在通过接收多种影响因子的输入，包括数据中心拓扑，纠删码（例如，经典的 Reed-Solomon 码，以及最近提出的 LRC 和 DRC），数据的冗余放置（即单节点单块数据和单节点多块数据），以及数学统计模型或系统历史运行数据的不同子系统的故障或者修复模式，同时可以量化基于纠删码构建的数据中心的持久性和可靠性的量化指标。此外，它可以采用重要性采样来加速整个仿真模拟的过程，从而能够对基于纠删码构建的数据中心的可靠性进行全面的仿真模拟分析。在我们的仿真平台中，没有实现编码操作，因为主要是将本文算法原型与 FastPR 和 SMFRepair 进行性能比较。

#### 3.5.1 实验环境

实验环境是在虚拟机中配置的 Ubuntu 16.04 LTS 系统，配置为 3.70GHz Intel Core i5-9600K 处理器，16GB 内存，1 Gbit/s 网络接口。在该系统中通过仿真平台搭建了分布式存储系统集群，集群包含 100 个节点，磁盘带宽上限  $b_d = 100MB/s$ ，网络带宽上

**算法 2 Improved SMFRepair Algorithm****Input:**  $w = (w_1, w_2, \dots)$ , type**Output:** NewW

```

1: NewW = w;
2: while true do
3:   R = FindMaxTimeRoad(NewW);
4:   if type == 1 then
5:     S = FindNodeWithoutChunk(NewW);
6:     D = R.Destination;
7:   else
8:     S, D = R.Source, R.Destination;
9:   end if
10:  RIdle = GetIdleNode(R);
11:  NewR = DijkstraMinTimeRoad(S, D, RIdle);
12:  Update(NewW, NewR);
13:  if FindMaxTimeRoad(NewW) == R then
14:    break;
15:  end if
16: end while
17: return NewW;

```

限  $b_n = 1Gb/s$ , 0~10 个空闲节点（只用于转发）。我们用 RS(6, 4), RS(9, 6)<sup>[10]</sup> (QFS 采用的编码策略), RS(14, 10)<sup>[9]</sup> (Facebook 采用的编码策略), RS(16, 12)<sup>[8]</sup> (Azure 采用的编码策略) 四种 RS 码进行对比实验。其中将块大小固定为 64MB 和 128MB 两种情况，并在存储集群中随机分布 1,000 个条带的块，测试结果采用运行 50 次以上的平均结果。

### 3.5.2 修复与调度实验

图 3-8 展示的是每个数据块的修复时间的仿真实验结果，其中针对不同的总节点数  $M$ , 不同的 RS 码, 不同的磁盘带宽  $b_d$  以及不同的网络带宽  $b_n$  分别进行了实验。当总结点数  $M$  比较低时, SMFRepair 表现出了较差的性能, 这是由于节点数少时, 中继节点产生的转发带宽性能优势发挥不出明显的作用。对于 FastPR 在节点数在 50 左右时, 与本文算法表现出了相同的修复性能, 而在节点较少的系统中, 本文的算法表现出了相对较优的修复性能。这是由于 SMSRS 算法在低节点数的环境中可以快速低根据最小堆技术和队列技术筛选出重建集, 之后与 IBA 算法结合可以在图搜索中快速确定一条相对修复路径更短的通路, 提高修复速度, 如图 3-8a 所示, 平均性能而言, 本文算法将修复时间降低了 9.2%。在不同 RS 码配置的实验中, 当  $k$  值较小时, 各个

算法的性能差别不大，这是由于系统中的冗余度不高，并且修复的任务所需要的块数不会产生过大的波动影响。相应地， $k$  值较大时 ( $k = 16$ ) 算法之间的差别的修复性能也较为相似，因此时的系统冗余度相当大，同一修复任务产生的调度集比较大，从而导致系统的修复性能稍有下降。而在中间状态的参数配置中，本文算法相较于传统的 FastPR 有着一定程度的改善，修复时间降低了 14.7%，整体平均降低了 10.2%。

如图 3-8c，在磁盘带宽大范围的变动下，算法之间的差别并不是特别明显，因为三种算法对于磁盘的读取速度并没有进行相应的优化，故差别不大。如图 3-8d，在网络带宽的实验中，FastPR 在带宽大于  $10Gb/s$  时表现出了较为优越的修复性能，而本文算法在较低磁盘带宽时也依然能保持着较低的修复时间，究其原因主要是由于高带宽时，预先修复的优势得到了充分的发挥，进而发挥出 FastPR 算法的优势，而本文算法在低带宽下考虑了预先修复策略，且加入了中继节点的方案从而缓解了低磁盘带宽环境下的修复劣势，在平均性能上，降低了 7.2% 的修复时间。

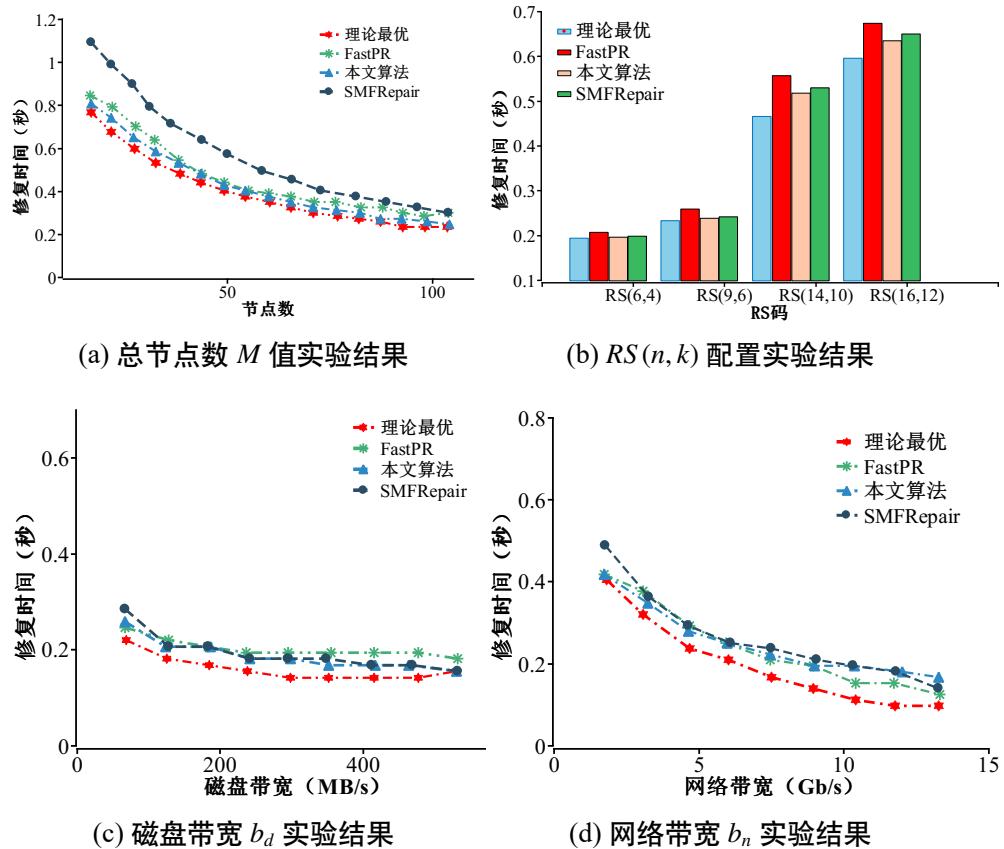


图 3-8 修复与调度对比实验

### 3.5.3 条带数和块大小的影响

如图 3-9a 所示，增加条带的数量为各种算法提供了更多的灵活性，进而帮助本文算法和 FastPR 确定最大化并行性的重建集。可以观察到，当条带的数量超过为 400 条时，FastPR、SMFRepair 和本文算法之间的差异非常小（在 5% 以内）。意味着可以将重建集的选择限制在较小的块组中，以减小运行时间的开销，同时实现接近最优的修复性能。此外，本文算法在条带数较小时，也发挥着相应的修复时间更低的优势，这是由于对于重建集和迁移集的划分改进中减少了相应的划分开销，在平均性能上，降低了 8.6% 的修复时间开销。

如图 3-9b 所示，实验评估了块大小的影响，范围从 32MB 到 128MB 不等。总体而言，每个数据块的修复时间随着数据块大小的增加而增加，但是本文算法仍然在所有数据块大小的修复时间上有所优化。可以看到，块大小越大优化的情况就越不明显，在块大小中等偏小的情况下，本算法能达到最优，平均优化 13.2% 的修复时间。

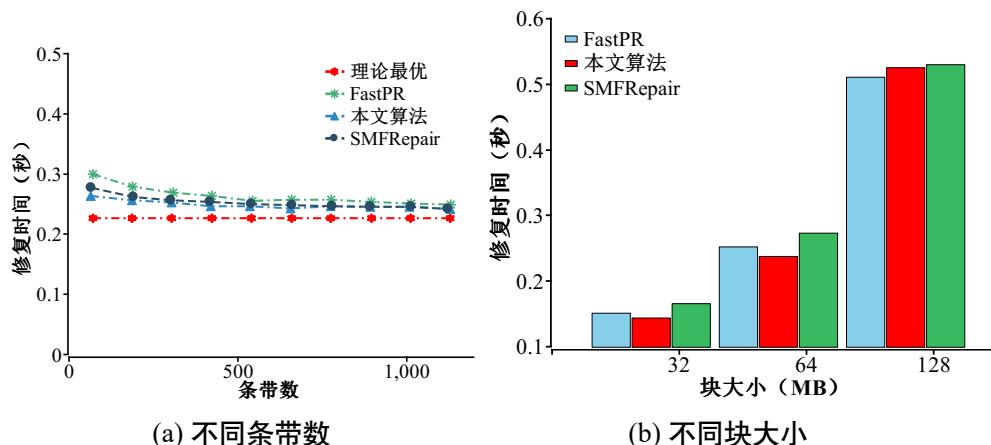


图 3-9 不同条带数和块大小的实验结果

### 3.5.4 空闲节点数的影响

系统中空闲节点数的多寡很大程度上可以影响中继转发的修复性能。实验设置空闲节点的数量从 0 到 10 不等，由于算法的优化在很大程度上取决于空闲节点，所以找出最佳的空闲节点数在实验中尤为重要。实验中设置四个典型的真实系统纠删码进行测试，结果显示在图 3-10 中。可以发现，SMFRepair 和本文算法的总体修复时间随着空闲节点数量的增加而减少，当数量从 0 到 2 时，修复时间下降最多。例如，SMFRepair 的修复时间是从 0.89s 减少到 0.7s 左右，在 RS(9, 6) 的配置下。当空闲节

点的数量从 0 开始增加时，两种技术的修复时间都会慢慢减少，直到空闲节点的数量达到 3 到 5 个左右。与没有空闲节点相比，本文算法实现了 1.58 倍修复效率的性能。。此外，可以观察到，当空闲节点的数量从 7 个逐渐增加到 10 个时，修复的效果性能改善不大，这是由于单节点的修复实验中，转发性能在高数量的空闲节点下已经饱和，增加下去已经无益。实验表明，空闲节点可以显著提高修复性能，且本文算法可以对原本的 FastPR 进行相应的改善，且空闲节点在 3 个左右比较有利于修复任务的加速。

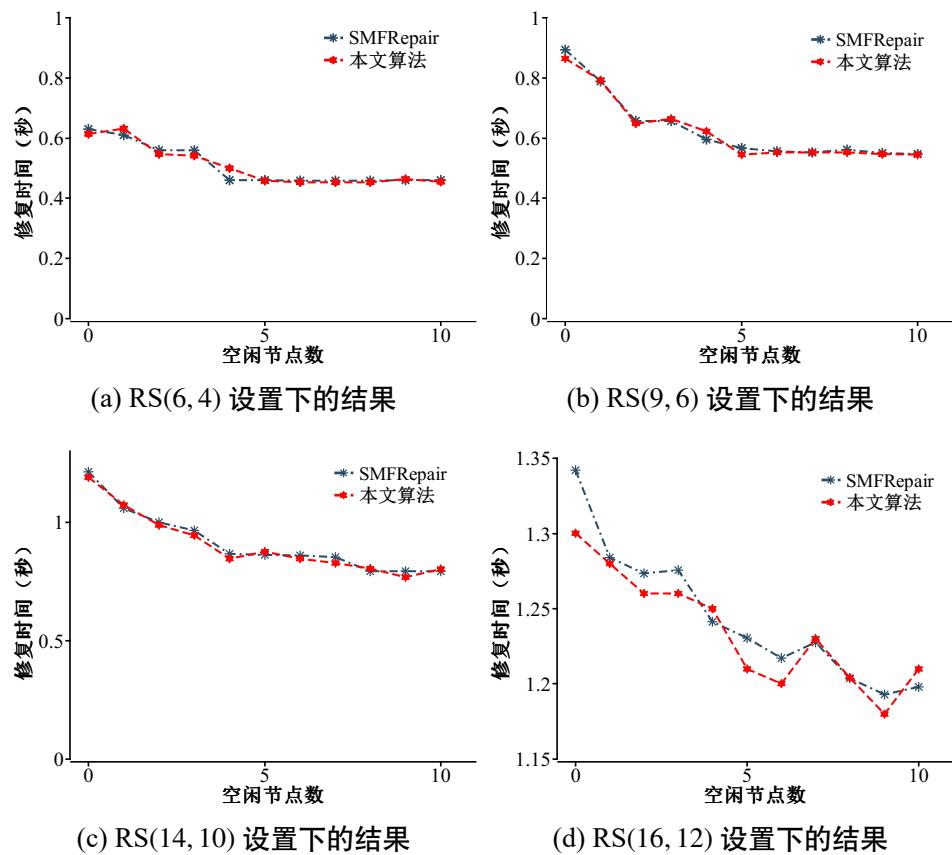


图 3-10 不同空闲节点数实验结果

### 3.5.5 带宽变化影响

本文的算法优化的是局部修复任务时的最优，所以对于全局情况下并非是最优的，由于系统修复事件是随机的，所以一般情况下也无法达到全局最优。本实验测试了在带宽不对等且快速变化的带宽环境中，不同纠删码配置下的各个算法的修复时间。使用不同的带宽配置来产生带宽值以模拟变化的带宽环境的变化。具体来说就是每当一个修复任务完成时，带宽将立刻发生变化，保证两个修复任务之间的带宽值是

不一样的。图 3-11 显示了相应的实验结果，可以看出，FastPR 在这样的带宽变化环境中表现不佳，主要是由于其本身并未针对快速变化的带宽的系统环境进行优化。例如。配置 RS(14, 10) 中，FastPR 的修复时间为 1.3s，相较于 SMFRepair 增加了不少。在  $k = 9, k = 14$  中，本文算法性能受到了牵制，主要也是由于图搜索的时间复杂度上的提升。但是本文算法在 FastPR 的基础上，加入了 SMFRepair 的优势，可以灵活地根据修复开始时的带宽构建一个最优的图搜索路径，根据每个时隙的当前带宽构建最佳修复方案，以更好地适应不断变化的时隙，灵活地构建最佳修复方案，以更好地适应带宽变化的网络环境。这对于当前的存储系统的快速修复来说具有重要意义，在平均水平上，本文算法降低了 11.2% 的修复时间开销。

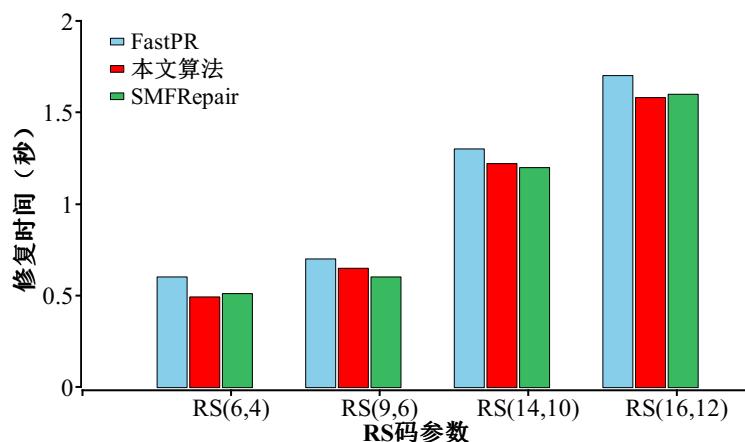


图 3-11 带宽变化实验结果

### 3.6 本章小结

针对大部分的传统修复方法都是被动修复，只有在检测到节点故障后才会触发修复操作。如果可以提前预测即将发生的故障，就可以在任何实际故障发生之前主动修复任何即将发生的节点故障，以提高系统可靠性。本文在 FastPR 对于 STF 节点修复任务的前提下，进一步将其扩展到网络环境快速变化的情形中。利用空闲节点加速数据的传输并感知当前网络环境的变化。针对重建集与迁移集问题，设计了划分迁移集与重建集算法 SMSRS (Split Migration Set And Reconstruction Set) 算法。利用多级传输的技术概念，亦即源节点数据在传递到源节点之前经过中继节点的转发，可以在更好地利用中间节点的高带宽优势的同时提高传输的并行性，本文提出了 ISA (Improved SMFRepair Algorithm) 算法来进行节点修复的调度。通过调整系统总节点数，RS 码配置，磁盘带宽，网络带宽，条带数，块大小，空闲节点数等实验因素，实

验结果表明，本文算法相较于传统的 FastPR 有着一定程度的改善，修复时间降低了 7.2%~13.2%。

## 第四章 基于混合纠删码的数据修复技术研究

### 4.1 引言

数据恢复一般有两种技术：重构（reconstruction）和退化读（degraded read）。重构是指数据已经在存储设备中丢失，在实际的数据修复后将其存储到新的设备中。退化读指的是在应对暂时性故障时，恢复操作读取的数据不必写入到设备中。研究一般集中在降低数据修复时所需要读取与传输的数据量上，缓解高退化读的延迟问题。

在传统的存储系统中大多只采用一种纠删码进行数据的处理和存储，这种方式一般先将文件分割成固定大小的数据块，再将这些数据块每  $k$  个作为一组，每组独立进行编码操作生成  $n$  个块，其中  $n$  个块的集合称为一个条带（Stripe），然而这种方式难以在保持低存储空间消耗的情况下降低退化读的延迟时间。最近，一些系统开始采用多副本技术与纠删码混合或者混合纠删码的方式来进行数据的存储<sup>[63,106,107]</sup>。Xia 等<sup>[63]</sup>, Friedman 等<sup>[107]</sup>设计了一种融合了多副本技术和纠删码技术存储系统。而在 Xia 等<sup>[63]</sup>提出的 HACFS 中，将从属于同一码种（product code 和 LRC）但不同参数的纠删码部署到系统中，分别处理冷数据和热数据。然而，由于选取了同一码族的编码，该码族的既定缺陷将难以避免。如采用 product code (PC) 时，其编码特性使得这一策略无论如何最多允许丢失三个数据块；而无论是采用 PC 还是 LRC，其存储冷数据时所采用编码均不是 MDS 的，在存储开销方面存在缺陷。为了改进此问题，Wang 等<sup>[92]</sup>尝试选取来自不同码族的两种编码，即 LRC 和 HH (Hitchhiker) 码来组成混合存储策略。目的是为了降低系统总体的退化读延迟，且对于热数据采用低恢复延迟的编码，冷数据采取保证 MDS 特性的编码。然后问题在于，虽然 LRC&HH 码的方案可以通过 downcode 和 upcode 的方式将数据存储从两种编码之间进行切换，但却没有良好的自适应负载，亦即没有针对数据的 I/O 特点针对性地提出各种负载场景下数据的动态切换方案。

本文在前文预先修复的单一纠删码的基础上，进一步推进了纠删码修复技术的应用。在 Wang 等<sup>[92]</sup>工作的基础上，提出了一种可感知数据热度的负载动态自适应的混合纠删码（LRC&HH）数据修复方案。旨在根据真实储存系统数据修复场景中的数据 I/O 的特点，以及数据访问和故障事件的时空局部性，提出更加符合相应情况的混合纠删码修复方案，从而对于计算开销型修复任务降低其开销和存储成本，对读

取密集型和频繁重建型修复任务降低其工作负载，并加速系统的修复速度和减少重建时间。

## 4.2 编码策略选择

为了选择出一对合适的编码策略来适应性地调整数据存储和恢复性能，本文总结了以下要求。

- (1) 两种编码策略应该有类似的布局，更加方便且低成本地建立它们之间的转换；
- (2) 两种编码策略应具有较高的灵活性，以支持任意数量的存储节点，并且可以适用于负载平衡的任务要求；
- (3) 两种编码策略在数据存储和数据修复任务中分别发挥不同的作用，实现功能上的互补。

### 4.2.1 局部修复码 (LRC)

LRC 在传统 RS 编码的基础上，引入局部分组的思想，将编码块划分成多个小分组，分组内部生成局部校验块进行容错保护。数据修复不再完全依赖于全局校验组，优先使用局部分组内的数据进行重建，从而通过减少参与节点的数量，降低了修复过程中的网络传输开销和磁盘读取开销。在 LRC 中， $m$  个冗余块被细分为全局冗余块及局部冗余块，两者的块数是 LRC 中的重要参数。记一个 LRC 编码中全局冗余块的块数为  $g$ ，局部冗余块的块数为  $l$ ，就有  $m = g + l$ ；这样的一个 LRC 被记为  $\text{LRC}(k, l, g)$ 。在进行  $\text{LRC}(k, l, g)$  的编码操作时，首先将  $k$  个数据块通过一个  $\text{RS}(k, g)$  码编码得到  $g$  个全局冗余块，之后将  $k$  个数据块（相对）平均地分为  $l$  个组，每一组将对应一个局部冗余块。在每一组中，局部冗余块将通过这一组中的数据块直接进行异或运算得到。

局部冗余块的设置减少了恢复数据时需要访问的节点数。当一个数据块需要恢复时，只需要获取它所在分组对应的局部冗余块及分组中其余的数据块，便可以对它们进行异或运算得出恢复结果。一个  $\text{LRC}(k, l, g)$  恢复一个数据块需要另外  $k/l$  块，而对应的一个  $\text{RS}(k, g)$  码则需要  $k$  块，是前者的  $l$  倍。如图 4-1 所示，是  $\text{LRC}(6, 2, 2)$  的具体组合方式，其中  $p_x$  和  $p_y$  是两个全局冗余块，而两个数据块被平均地分为了三组  $(x_1, x_2), (y_1, y_2), (z_1, z_2)$ ，它们分别对应的局部冗余块为  $p_{xy}$  和  $p_{yz}$ 。当  $x_1$  需要被修复

时，只需要获取  $x_2, y_1$  和  $p_{xy}$  即可。

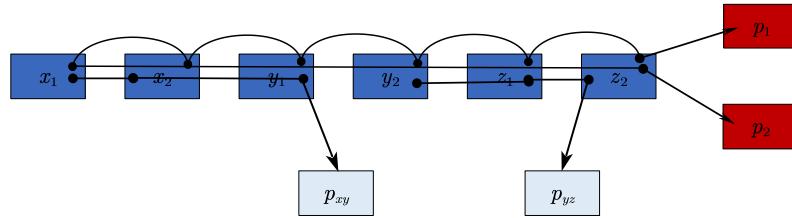


图 4-1 LRC(6,2,2) 的组成

#### 4.2.2 Hitchhiker (HH) 码

在一个  $\text{HH}(k, m)$  码中，每个条带被分为两个子条带，分别为  $a = a_i$  和  $b = b_i$ ，且其中有  $k$  个数据块和  $m$  个校验块。在编码计算中，同一个子条带中的数据首先会被编码成  $\text{RS}(k, m)$ ，从而得到校验块  $f_i(a)$  和  $f_i(b)$  ( $1 \leq i \leq m$ )。然后，将子条带  $a$  中的  $k$  个数据块平均地分为  $(m - 1)$  组，每个组  $G_i$  对应子条带  $b$  中的校验块  $b_{k+i+1}$ ，最后将  $G_i$  和  $f_{i+1}(a)$  中的数据块进行 XOR 计算。

在丢失一个数据块时，设其在  $a$  中的子数据块在上述分组中对应  $g_i(a)$ 。首先通过子条带  $b$  中可用的另外  $(k - 1)$  个数据块以及  $f_1(b)$ ，可以恢复得到丢失的在  $b$  中的子数据块，进而可以求出  $f_{(t-k)}(b)$ 。而  $a$  中子数据块则可以通过上述分组中同一组内其余的  $a$  中子数据块，子条带  $b$  的第  $t$  个子块，以及刚刚求出的  $f_{(t-k)}(b)$  进行异或运算得到。这样，要恢复一个数据块，需要读取的子数据块数为  $k + \frac{k}{m-1} = \frac{km}{m-1}$ ，折合为  $\frac{km}{2(m-1)}$  块，相比于 RS 码需要  $k$  块，在  $m \geq 3$  时均有较为明显的减少。

如图 4-2 所示，展示的是  $\text{HH}(12, 4)$  组成，如果想修复  $a_1$  和  $b_1$ ，首先需要读取  $b_2$  到  $b_{13}$  总共 12 个数据块。这 12 个数据块足以恢复出  $b_1$ ，然后可以计算出  $b_{14}$  对应的 RS 校验块  $f_2(b)$ 。最后，再读取  $a_2$  到  $a_4$  和  $b_{14}$ ，计算出  $a_1 = a_2 \oplus a_3 \oplus a_4 \oplus b_{14} \oplus f_2(b)$ 。总共需要读取 16 个数据块，相比 RS 码少读了 8 个数据块。

### 4.3 负载自适应策略

受 Qiu 等<sup>[91]</sup> 工作的启发，针对实际存储系统数据 I/O 的特点，本文静态地将数据分为六类（如表 4-1 所示），其中风险指的是因数据丢失而产生对系统风险的影响。1) 低风险的冷数据；2) 高风险的冷数据；3) 低风险的写入密集型数据；4) 高风险的写入密集型数据；5) 低风险的读取密集型数据；6) 高风险的读取密集型数据。其

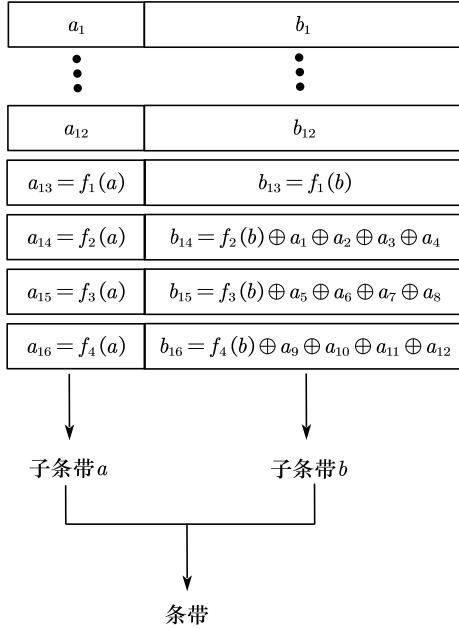


图 4-2 HH(12, 4) 码组成

中，读取密集型和写入密集型数据都属于热数据，而读写平衡的系统应用可以属于写入密集型或读取密集型。一般来说在低风险环境下，这种应用更可能是写入密集型的，因为整体性能主要受写入操作的影响。然后可以确定为其分配何种编码，并为数据的动态 I/O 提供自适应方法。

表 4-1 不同系统负载情况下的编码分配方案

编码策略		修复负载	
		高风险	低风险
系统负载	写入密集型	LRC 或 HH	Hitchhiker
	读取密集型	LRC	Hitchhiker
	冷数据	Hitchhiker	Hitchhiker

### 4.3.1 编码分配规则

为了让不同的数据和读写任务工作负载情况匹配上适当的编码方案，显然可以得到如表 4-1 中总结了编码分配策略。直观地说，冷数据或低风险数据可以选择存储开销较低的 HH 码，这样可以保持较高的存储效率，以及良好的系统可靠性。特别是对于写入密集型 I/O 的任务，而对于具有高故障风险且以读为主的数据，LRC 码可以

节省大量的重建带宽，并对 I/O 速度造成更少的负面影响。然而，对于高风险的写入密集型数据，难以选择一个合适的编码策略，因需要在系统的读取性能和修复 I/O 之间进行平衡，从而确定合适的编码来最大化系统的整体性能。为了解决这个问题，本文进行了如下计算，最终结果呈现在表 4-2。

**读写负载模式。**首先确定 RS( $k, r$ ) 在单个数据块上的计算性能消耗  $RWCost_{compute}^{RS}$ ，磁盘 I/O 消耗  $RWCost_{I/O}^{RS}$ ，传输消耗  $RWCost_{transmission}^{RS}$ 。设系统读写调度的写入速度为  $v_w$ ，读取速度为  $v_r$ ，一个数据块大小为  $\gamma$ ，那么对于 RS( $k, r$ ) 而言，其编码过程中的 GF 运算次数等价于运算复杂度即  $k \times r$ ，而对于读写交叉的任务而言其所占据的比例为  $\frac{v_w}{v_w + v_r}$ ，故在读写负载模式中 RS( $k, r$ ) 的计算性能消耗  $RWCost_{compute}^{RS}$  为：

$$RWCost_{compute}^{RS} = \frac{v_w}{v_w + v_r} \gamma kr \quad (4.1)$$

这里令  $\frac{v_w}{v_r}$  为  $\beta$ ，则式 4.1 可改写为：

$$RWCost_{compute}^{RS} = \frac{\beta}{1 + \beta} \gamma kr \quad (4.2)$$

设磁盘一次 I/O 可传输的字节数为  $\phi$ ，则可得到读写负载模式中 RS( $k, r$ ) 的磁盘 I/O 消耗  $RWCost_{I/O}^{RS}$  为：

$$RWCost_{I/O}^{RS} = \frac{\gamma}{\phi} \quad (4.3)$$

对于传输消耗  $RWCost_{transmission}^{RS}$ ，分为读取传输和写入传输，其中读取任务的传输消耗为  $\frac{v_r}{v_r + v_w}$ ，写入任务的传输消耗为  $\frac{v_w \frac{r+k}{k}}{v_r + v_w}$ 。那么读写负载模式中 RS( $k, r$ ) 的传输消耗  $RWCost_{transmission}^{RS}$  为：

$$\begin{aligned} RWCost_{transmission}^{RS} &= \frac{v_r}{v_r + v_w} + \frac{v_w \frac{r+k}{k}}{v_r + v_w} \\ &= \frac{1}{1 + \beta} + \frac{\beta}{1 + \beta} \cdot \frac{r + k}{k} \\ &= \frac{\beta(r + k) + k}{k(1 + \beta)} \\ &= 1 + \frac{\beta}{1 + \beta} \cdot \frac{r}{k} \end{aligned} \quad (4.4)$$

由于 LRC( $k, l, g$ ) 恢复一个数据块需要另外  $k/l$  块，而对应的一个 RS( $k, g$ ) 码则需要  $k$  块，是前者的  $l$  倍。故将其分别代入式 4.2，式 4.3，式 4.4，则可以得到读写负载

模式中  $LRC(k, l, g)$  的计算性能消耗  $RWCost_{compute}^{LRC}$ , 磁盘 I/O 消耗  $RWCost_{I/O}^{LRC}$ , 传输消耗  $RWCost_{transmission}^{LRC}$  分别为:

$$RWCost_{compute}^{LRC} = \frac{\beta}{l(1+\beta)} \gamma kr \quad (4.5)$$

$$RWCost_{I/O}^{LRC} = \frac{\gamma}{\phi} \quad (4.6)$$

$$RWCost_{transmission}^{LRC} = 1 + \frac{\beta}{k(1+\beta)} \cdot rl \quad (4.7)$$

同样, 对于 HH 码, 其需要读取  $\frac{km}{2(m-1)}$  块数据进行处理, 分别将其代入式 4.2, 式 4.3, 式 4.4, 则可以得到读写负载模式中  $HH(k, m)$  的计算性能消耗  $RWCost_{compute}^{HH}$ , 磁盘 I/O 消耗  $RWCost_{I/O}^{HH}$ , 传输消耗  $RWCost_{transmission}^{HH}$  分别为:

$$RWCost_{compute}^{HH} = \frac{\gamma \beta kmr}{2(1+\beta)(m-1)} \quad (4.8)$$

$$RWCost_{I/O}^{HH} = \frac{\gamma}{\phi} \quad (4.9)$$

$$RWCost_{transmission}^{HH} = 1 + \frac{\beta}{1+\beta} \cdot \frac{2r(m-1)}{km} \quad (4.10)$$

**修复负载模式。**与读写模式相同, 首先确定  $RS(k, r)$  在修复负载模式下单个数据块上的计算性能消耗  $RecoverCost_{compute}^{RS}$ , 磁盘 I/O 消耗  $RecoverCost_{I/O}^{RS}$ , 传输消耗  $RecoverCost_{transmission}^{RS}$ 。因在修复过程中需要传输  $k$  个数据块和  $(r+k) \times r \times r^{[91]}$  次运算, 故修复负载模式下的  $RS(n, k)$  的计算性能消耗  $RecoverCost_{compute}^{RS}$  为:

$$RecoverCost_{compute}^{RS} = (r+k)r^2 + \gamma k \quad (4.11)$$

同样可得, 修复负载模式下的  $RS(n, k)$  的磁盘 I/O 消耗  $RecoverCost_{I/O}^{RS}$  为:

$$RecoverCost_{I/O}^{RS} = \frac{\gamma}{\phi} \quad (4.12)$$

显然, 修复时需要传输  $k$  个数据块, 则传输消耗  $RecoverCost_{transmission}^{RS}$  为:

$$RecoverCost_{transmission}^{RS} = k \quad (4.13)$$

同理, 分别将 LRC 码的  $\frac{k}{l}$  和 HH 码的  $\frac{km}{2(m-1)}$  代入到式 4.11, 4.12, 4.13, 可得修复负载模式中  $LRC(k, l, g)$  的计算性能消耗  $RecoverCost_{compute}^{LRC}$ , 磁盘 I/O 消耗  $RecoverCost_{I/O}^{LRC}$ , 传输消耗  $RecoverCost_{transmission}^{LRC}$  分别为:

$$RecoverCost_{compute}^{LRC} = r^3 + \frac{kr^2}{l} + \frac{k\gamma}{l} \quad (4.14)$$

$$RecoverCost_{I/O}^{LRC} = \frac{\gamma}{\phi} \quad (4.15)$$

$$RecoverCost_{transmission}^{LRC} = \frac{k}{l} \quad (4.16)$$

修复负载模式中  $HH(k, m)$  的计算性能消耗和磁盘 I/O 消耗  $RecoverCost_{compute}^{HH}$ 、 $RecoverCost_{I/O}^{HH}$ , 传输消耗  $RecoverCost_{transmission}^{HH}$  分别为:

$$RecoverCost_{compute}^{HH} = r^3 + \frac{km(r^2 + \gamma)}{2(m-1)} \quad (4.17)$$

$$RecoverCost_{I/O}^{HH} = \frac{\gamma}{\phi} \quad (4.18)$$

$$RecoverCost_{transmission}^{HH} = \frac{km}{2(m-1)} \quad (4.19)$$

最终, 读写负载模式和修复负载模式下的 LRC 和 HH 各种任务的消耗计算如表 4-2 所示。

表 4-2 不同系统负载情况下的编码分配方案

	编码策略	$LRC(k, l, g)$	$HH(k, m)$
读写负载模式	$RWCost_{compute}$	$\frac{\beta}{l(1+\beta)}\gamma kr$	$\frac{\gamma\beta kmr}{2(1+\beta)(m-1)}$
	$RWCost_{I/O}$	$\frac{\gamma}{\phi}$	$\frac{\gamma}{\phi}$
	$RWCost_{transmission}$	$1 + \frac{\beta}{k(1+\beta)} \cdot rl$	$1 + \frac{\beta}{1+\beta} \cdot \frac{2r(m-1)}{km}$
修复负载模式	$RecoverCost_{compute}$	$r^3 + \frac{kr^2}{l} + \frac{k\gamma}{l}$	$r^3 + \frac{km(r^2 + \gamma)}{2(m-1)}$
	$RecoverCost_{I/O}$	$\frac{\gamma}{\phi}$	$\frac{\gamma}{\phi}$
	$RecoverCost_{transmission}$	$\frac{k}{l}$	$\frac{km}{2(m-1)}$

设  $\alpha$  为系统进行 XOR 计算的速度 (次/s),  $\lambda$  为网络中传输的数据量 (bytes/s), 故可得相应编码的写读开销  $W$  和  $R$  分别为:

$$\begin{aligned}
W &= RWCost_{compute} \cdot \frac{1+\beta}{\beta} \cdot \frac{1}{\alpha} \\
&+ RWCost_{I/O} \\
&+ \frac{\gamma}{\lambda} (RWCost_{transmission} - \frac{1}{1+\beta}) \cdot \frac{1+\beta}{\beta}
\end{aligned} \tag{4.20}$$

$$\begin{aligned}
R &= \frac{RecoverCost_{compute}}{\alpha} \\
&+ RecoverCost_{I/O} \\
&+ \frac{\gamma RecoverCost_{transmission}}{\lambda}
\end{aligned} \tag{4.21}$$

基于表 4-2 和式 4.20, 式 4.21 可以为 LRC 码和 HH 码定义如下变量:

$$\begin{aligned}
W_{LRC} &= \gamma \left( \frac{kr}{\alpha l} + \frac{k+rl}{k\lambda} + \frac{\gamma}{\phi} \right) \\
R_{LRC} &= \frac{r^3 + \frac{kr^2}{l} + \frac{ky}{l}}{\alpha} + \gamma \left( \frac{k}{\lambda l} + \frac{1}{\phi} \right) \\
W_{HH} &= \frac{\gamma kmr}{2\alpha(m-1)} + \frac{\gamma}{\lambda} \left( 1 + \frac{2r(m-1)}{km} \right) + \frac{\gamma}{\phi} \\
R_{HH} &= \frac{r^3}{\alpha} + \frac{km(r^2 + \gamma)}{2\alpha(m-1)} + \frac{\gamma}{\phi} + \frac{kmy}{2\lambda(m-1)}
\end{aligned} \tag{4.22}$$

由式 4.22, 可以得到如下不等式来决定编码的选择:

$$\delta = \frac{\text{writes}}{\text{recoveries}} \geq \frac{R_{LRC} - R_{HH}}{W_{HH} - W_{LRC}} = \eta \tag{4.23}$$

当  $\delta \geq \eta$  时, 意味着瓶颈性能主要受到写需求的影响, 所以选用 HH 码更加合适; 否则重建带宽就是最大的影响因素, 那么 LRC 码则是更好的选择。在实践中<sup>[91]</sup>, 为了缓解频繁切换编码策略所产生的开销, 将不等式 4.23 调整为下式:

$$\delta \geq \eta + \Delta \text{ 或 } \delta \leq \eta - \Delta (0 \leq \Delta < \eta) \tag{4.24}$$

### 4.3.2 自适应选择算法

随着系统的运行, 数据的冷热属性会不断地进行切换。为此需要构造相应的冷热数据划分算法, 记录数据访问发生失败时的时间和节点位置, 利用这些信息可以使用 LRU 算法来为冷热数据划分定制自适应选择算法。定义两个队列分别用于正常读取

模式和故障模式，将缓存算法应用于队列，具体如图 4-3 所示，根据 HACFS<sup>[63]</sup> 用 Fast Code 和 Compact Code 指代对应的两种编码方式，相应地分别对应 LRC 码和 HH 码。

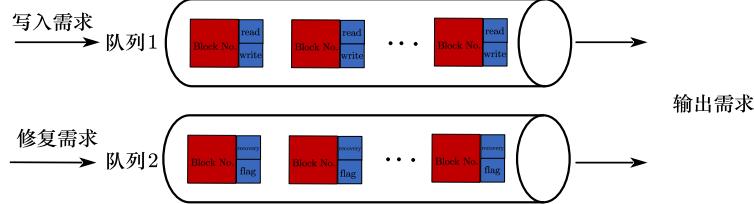


图 4-3 动态自适应划分冷热数据队列

用队列 1 来记录数据写入的信息流，其中每个块记录了块 ID 和缓存命中次数。队列 2 存储着数据块的恢复请求，其中包含了区块 ID、缓存命中次数和表示数据的编码方案。首先将  $LRC(k, l, g)$  设定为整个存储系统的默认编码，两个队列在启动后会有一个默认初始化，在自适应规则中，三个条件来触发自适应过程。首先，对于在队列 2 头部插入的每个修复请求，算法确定相关块是否需要用 LRC 码进行编码。其次，对于在队列 1 头部插入的每个写请求，算法确定相关的块是否需要用 HH 码进行编码。最后，对于在队列 2 尾部删除的每个修复请求，算法会将 LRC 码转换成 HH 码。具体的过程如算法 3 中所示。

---

### 算法 3 动态自适应冷热数据划分算法

---

**Input:** Queue1, Queue2 and  $\eta$

```

1: for Queue2 头的每个插入请求 do
2:   if flag ≠ LRC and (writes/recoveries) <  $\eta$  then
3:     set flag = LRC;
4:     convert to LRC;
5:   end if
6: end for
7: for Queue1 头的每个插入请求 do
8:   if flag ≠ HH and (writes/recoveries) ≥  $\eta$  then
9:     set flag = HH;
10:    convert to HH;
11:   end if
12: end for
13: for Queue2 尾的每个弹出请求 do
14:   if flag == LRC then
15:     set flag = HH;
16:     convert back to HH;
17:   end if
18: end for

```

---

## 4.4 编码切换算法

HH 码和 LRC 码均为 RS 码的一种改进。在使用  $\text{HH}(k, m)$  码进行编码时，首先需要进行  $\text{RS}(k, m)$  码的编码，在进行一系列 Piggybacking 框架中关于  $g_i(a)$  的异或运算；而在使用  $\text{LRC}(k, l, g)$  进行编码时， $g$  个全局冗余块需要通过执行  $\text{RS}(k, g)$  码的编码函数得到，而  $l$  个局部冗余块则需要进行异或运算得到。另外  $\text{HH}(k, m)$  码中的  $g_i(a)$  的计算，需要将  $k$  个数据块分为  $(m - 1)$  组，而  $\text{LRC}(k, l, g)$  中的局部冗余的计算，则需要将  $k$  个数据块分为  $l$  组。由此，当  $k_{\text{HH}} = k_{\text{LRC}}$  且  $m_{\text{HH}} = g_{\text{LRC}}$  且  $m_{\text{HH}} - 1 = l_{\text{LRC}}$  时，相应的 HH 码和 LRC 码可以共用一个 RS 码的冗余块，并且可以进行高效地切换。

记  $k = k_{\text{HH}} = k_{\text{LRC}}$  以及  $t = m_{\text{HH}} = g_{\text{LRC}}$ ，则  $l_{\text{LRC}} = m_{\text{HH}} - 1 = t - 1$ ，系统中采用  $\text{HH}(k, t)$  和  $\text{LRC}(k, t - 1, t)$ 。由于 HH 码会将原 RS 码中每两个条带视为一个条带中的两个子条带，因此此切换算法也对 LRC 进行类似处理，即每一个条带中都包含两个子条带。在 LRC 中，一个条带中的两个子条带分别进行各自的编码；而在 HH 码中，其中一个条带  $a$  的额外冗余信息  $g_i(a)$  会被异或到另一子条带  $b$  的冗余子块中。

从  $\text{LRC}(k, t - 1, t)$  转换为  $\text{HH}(k, t)$  码时，首先将子条带  $a$  中的各局部冗余子块读取并传输到对应的全局冗余块所在节点，并与该节点中子条带  $b$  的全局冗余子块进行异或运算，之后再将各局部冗余块删除即可。详细过程如算法 4 和图 4-4 所示。

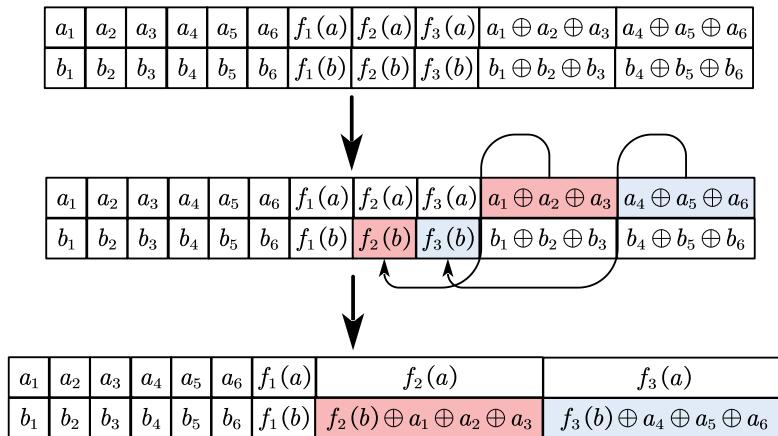
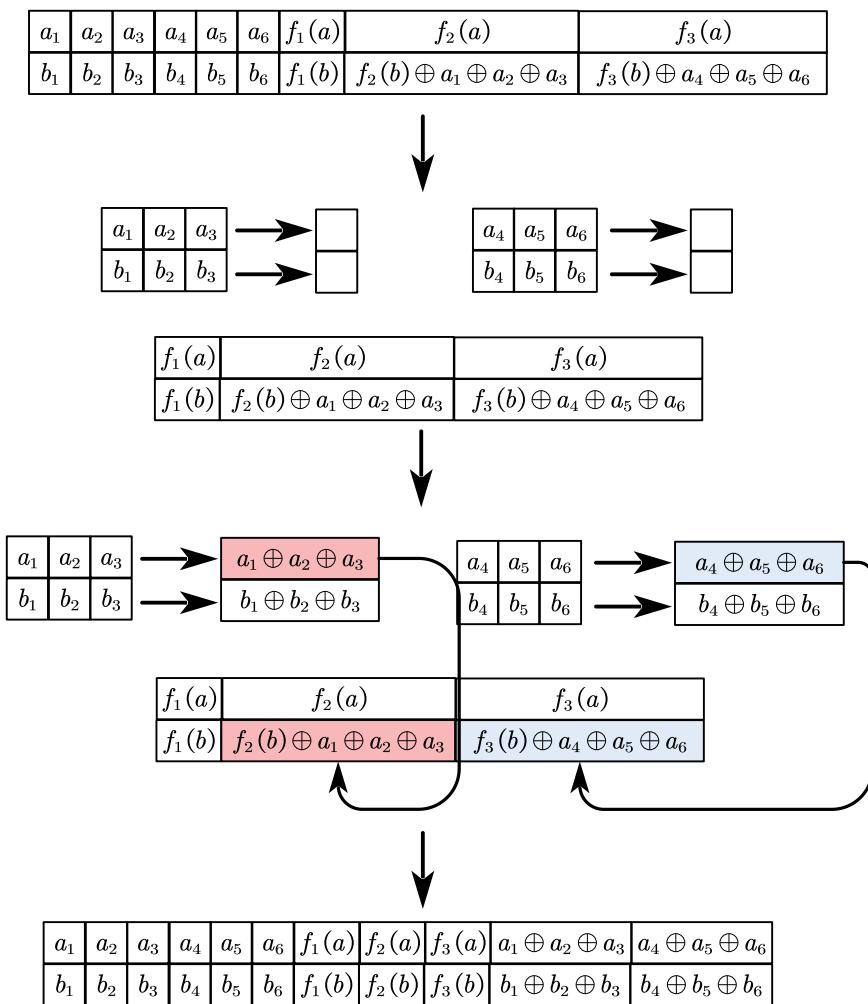


图 4-4  $\text{LRC}(6, 2, 3) \rightarrow \text{HH}(6, 3)$  过程

从  $\text{HH}(k, t)$  码转换为  $\text{LRC}(k, t - 1, t)$  时，首先计算出所有局部冗余块，之后再将其中子条带  $a$  中的局部冗余子块传到对应的全局冗余块所在节点，并与该节点中子条带  $b$  的全局冗余子块进行异或运算，使得这些子条带  $b$  的全局冗余子块不再包含子条带  $a$  中的额外冗余信息  $g_i(a)$ 。详细过程如算法 5 和图 4-5 所示。

**算法 4 LRC( $k, t - 1, t$ )  $\rightarrow$  HH( $k, t$ ) 算法****Input:** LRC 中的两个子条带  $a$  和  $b$ **Output:** HH 码中的两个子条带  $s_a$  和  $s_b$ 

- 1:  $s_a.data := a.data$
- 2:  $s_b.data := b.data$
- 3:  $s_a.parity := a.global\_parity$
- 4:  $s_b.parity[1] := b.global\_parity[1]$
- 5: **for**  $i := 2$  to  $t$  **do**
- 6:      $s_b.parity[i] := b.global\_parity[i] \oplus a.local\_parity[i - 1]$
- 7: **end for**

图 4-5 HH(6,3)  $\rightarrow$  LRC(6,2,3) 过程

**算法 5**  $\text{HH}(k, t) \rightarrow \text{LRC}(k, t - 1, t)$  算法**Input:** HH 码中的两个子条带  $s_a$  和  $s_b$ **Output:** LRC 码中的两个子条带  $a$  和  $b$ 

```

1:  $a.\text{data} := s_a.\text{data}$ 
2:  $b.\text{data} := s_b.\text{data}$ 
3:  $a.\text{global\_parity} := s_a.\text{parity}$ 
4:  $b.\text{global\_parity}[1] := s_b.\text{parity}[1]$ 
5: for  $i := 1$  to  $t - 1$  do
6:    $a.\text{local\_parity}[i] := 0$ 
7:    $b.\text{local\_parity}[i] := 0$ 
8:   for  $G_i$  中的每个块  $j$  do
9:      $a.\text{local\_parity}[i] := a.\text{local\_parity}[i] \oplus a.\text{data}[j]$ 
10:     $b.\text{local\_parity}[i] := b.\text{local\_parity}[i] \oplus b.\text{data}[j]$ 
11:   end for
12: end for
13: for  $i := 2$  to  $t$  do
14:    $b.\text{global\_parity}[i] := s_b.\text{parity}[i] \oplus a.\text{local\_parity}[i - 1]$ 
15: end for

```

## 4.5 实验结果与分析

### 4.5.1 实验参数与数据集

实验选取参数的  $(k, t) = (12, 4)$ , 即在系统中部署 LRC(12, 3, 4) 和 HH(12, 4) 两种编码。根据这一选择, 在仿真平台上需要 19 个以上的节点部署, 其他实验环境与第三章一致。本文选择了四种数据集<sup>[108]</sup> 作为经典的系统运行负载如表 4-3 所示, 关于每个数据集的解释如下:

- MSR-MDS1, 来自于媒体服务器且在四个数据集中拥有最高的读取比例;
- MSR-RSRCH0, MSR-RSRCH2, 两个数据集均通过研究项目产生, 其中 MSR-RSRCH0 的读取量最低, 且 MSR-RSRCH2 含有中等的读取量数据;
- MSR-WEB1, 通过 Web/SQL 服务器产生, 拥有中等的读取数据量。

对于修复负载, 在实验平台中随机初始化故障事件, 然后记录故障节点的位置和时间戳。对于时间间隔的处理, 计算从最后一次发生故障的时间间隔, 并通过输入的时间间隔生成满足正态分布故障概率的故障事件。对于故障位置, 设置目标故障节点和最近故障节点之间的相对距离与故障概率成反比。同样由于存储系统中 98% 的故障时单节点故障<sup>[63]</sup>, 所以实验基于单节点故障进行。对于 Wang 等<sup>[92]</sup> 的 LRC&HH 策略默认存储方式为 HH 码, 因为他们在实验中的数据访问遵循 Zipf 模拟, 并且没

表 4-3 数据集介绍

数据集	请求数	读取比	IOPS	平均请求数据大小
MSR-MDS1	1637711	92.88%	27.29	113.00KB
MSR-RSRCH2	207597	65.69%	3.54	8.17KB
MSR-WEB1	160891	54.11%	2.66	58.14KB
MSR-RSRCH0	1433655	9.32%	23.70	17.86KB

有相应的自适应负载切换算法，所以设置为数据访问时随机切换编码策略，具体来说当数据访问产生时，随机选择是否根据当前编码策略读取数据，否则切换为另一种编码策略从而访问数据。

此外，本文选取了四种指标作为实验结果的衡量尺度，分别为读写负载性能  $\epsilon_1$ ，修复负载性能  $\epsilon_2$ ，整体负载性能  $\epsilon$  和临界比  $\zeta$ 。正如章节4.3.1所分析的三种参数的量化。具体的指标定义如下：

- (1) 读写负载性能  $\epsilon_1$ ，定义为读写任务中的读写操作的平均延迟，包括了计算和数据访问消耗即 transmission+disk I/O；
- (2) 修复负载性能  $\epsilon_2$ ，定义为修复任务中解码操作的平均消耗，包括计算和访问消耗；
- (3) 整体负载性能  $\epsilon$ ，定义为  $\epsilon = \frac{\mu_1\epsilon_1 + \mu_2\epsilon_2}{\mu_1 + \mu_2}$ ，这里  $\mu_1$  和  $\mu_2$  分别代表读写和修复中的请求数；
- (4) 临界比  $\zeta$ ，定义为  $\zeta = \frac{1}{\epsilon \times \rho}$ ，表示性能优先和存储优先之间的比例，其中  $\rho = \frac{k+r}{k}$ 。

#### 4.5.2 读写负载性能实验结果

在读写负载性能实验中，RS(10, 4) 在四个数据集上的修复时间都略高于 HH(10, 4) 和 LRC(12, 2, 3) 的方案。其中 LRC(12, 2, 3) 构建的方案比 HH(10, 4) 的修复速度更快一点。而利用了二者优势的 LRC&HH Random 码，在重建性能上远优于二者，这是由于在系统上冷热数据转换时，会将 LRC( $k, t - 1, t$ ) 切换为 HH( $k, t$ )，原本的切换方式需要传递  $(k + t - 1)$  个块，而混合方案只需要传输  $(t - 1)$  个子块，即  $\frac{t-1}{2}$ ，而当  $k = 12$  且  $t = 4$  时，这种切换算法只需要重新编码算法的  $\frac{1}{10}$ 。反过来，冷数据转换为热数据时，会将 HH( $k, t$ ) 切换为 LRC( $k, t - 1, t$ )，若重新编码则需要传输  $(t + 2t - 2)$  个块，而

切换算法只需要传输  $(2k + t - 1)$  个子块，即  $\frac{2k+t-1}{2}$  块，当  $k = 12$  且  $t = 4$  时，传输的数据量为原来的  $\frac{1}{4}$ 。

在这基础上，本文算法在四个数据集上平均性能比 LRC&HH Random 又有了相应的优化，平均修复时间降低了 18.7%。因为 LRC&HH Random 在存储系统的读取与修复过程中没有对数据的自适应转换做相应的优化，本文策略在读写负载上，利用队列技术以及相应的计算分析出了负载均衡点，从而可以从图 4-6 中的蓝色柱子看出，本文的策略在前三个数据集上的优化较为明显，因为其相应的读取比很高。

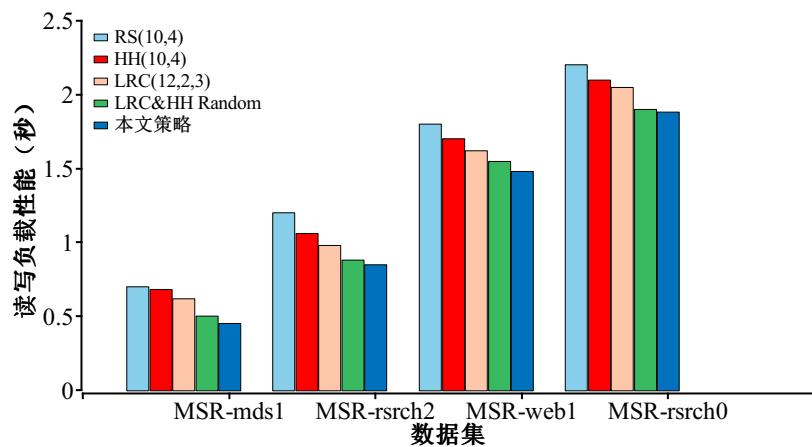


图 4-6 四个数据集的读写负载性能实验

### 4.5.3 修复负载性能结果

与读写负载实验相比，四个数据集的修复负载实验中的结果的差异化更加明显。其中，RS(10,4) 在四个数据集上都表现出了远超于其他方案的修复时间，主要是修复负载下，热数据的冗余更多，而 RS(10,4) 消耗了巨大的计算性能，从而拉高了修复时间。在 MSR-MDS1 中，LRC&HH Random 与本文策略的修复时间差距不大，主要是由于其具有较大的请求数、读取比以及 IOPS，这样的高度集中的数据请求的情况下，算法不能做出及时的反应。而在其他三个数据集 MSR-WEB1、MSR-RSRCH2 以及 MSR-RSRCH0，本文策略都用较为明显的优势，在平均性能上降低了 9.4% 的修复时间。

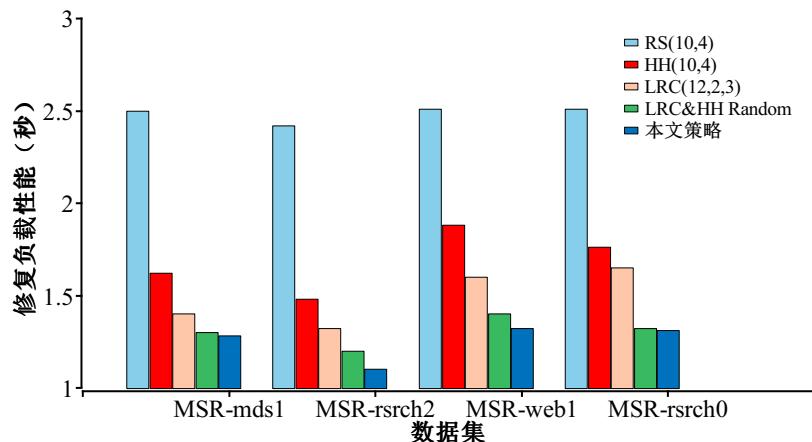


图 4-7 四个数据集的修复负载性能实验

#### 4.5.4 整体负载性能实验结果

在整体性能实验中，即将读写和修复任务均匀地分摊到系统事件中，从而观察各个策略的修复性能。从图 4-8可以看出，在读写和修复交叉的任务中，RS(10,4) 在不同数据集上的修复性能有所上升。此外 HH(10,4) 和 LRC(12,2,3) 在不同数据集上的差异很小，在 MSR-RSRCH0 上，HH(10,4) 的修复时间最高，且 LRC&HH Random 的平均性能还是会优于其他的单一纠删码，因为在整体负载情况下，切换任务时有发生。但是由于切换时机的随机性，本文策略在基于实时的冷热数据优化算法以及自适应策略的帮助下，在大部分情况下依然优于随机切换的方式，平均性能上降低了 6.8% 的修复时间。此外，在计算效率方面，这两种策略都是进行 XOR 运算，且涉及范围相对整一组数据较小，这对比重新编码算法需要整份数据进行的伽罗瓦运算也有明显的优化，大大提升了计算效率。

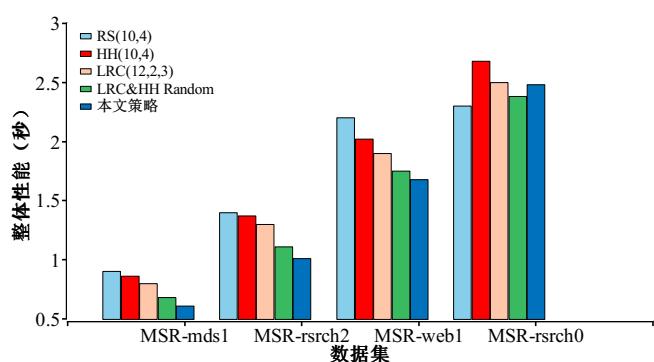


图 4-8 四个数据集的整体负载性能实验

#### 4.5.5 临界比实验结果

临界比是一种衡量修复消耗的参数，数值越低则代表具有相应更低的修复消耗，修复性能更加优越。从图 4-9 可以看出，LRC(12, 2, 3) 在四个数据集上都有着较高的临界比，意味着更加低的修复性能，与之相类似的是 HH(10, 4)，在前两个数据集上的数值基本下基本相同。在平均性能上，本文策略的临界比相较于 LRC&HH Random 低 5.8%。

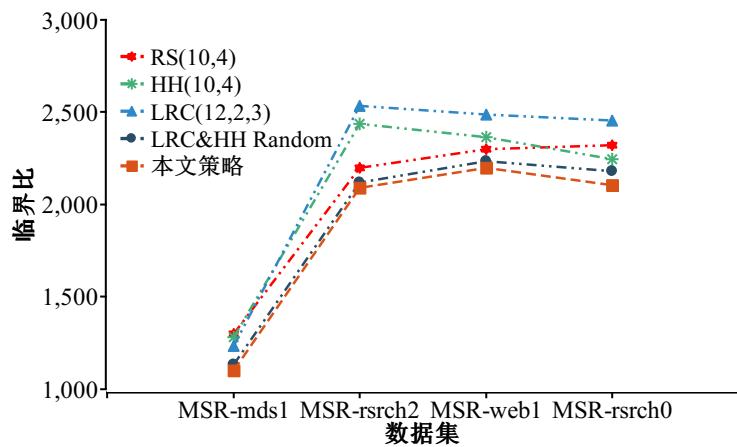


图 4-9 四个数据集的临界比实验

### 4.6 本章小结

针对在传统的存储系统中大多只采用一种纠删码进行数据的处理和存储，难以在保持低存储空间消耗的情况下降低退化读的延迟时间问题。本文在前文预先修复的单一纠删码的基础上，进一步推进了纠删码修复技术的应用，提出了一种可感知数据热度的负载动态自适应的混合纠删码（LRC&HH）数据修复方案。静态地将数据分为六类，分别根据读写负载平衡和修复负载，根据数学计算提出相应的负载均衡策略，使用 LRU 算法来为冷热数据划分定制自适应选择算法。选取四个数据集，分别对读写负载，修复负载，整体负载和临界比进行了对比实验，实验结果表明，本文策略相较于 LRC&HH Random 降低了 6.8%~18.7% 的修复时间。

# 第五章 基于混合纠删码的容错存储原型系统设计与实现

为了验证预先修复技术和混合纠删码修复技术在实际应用中的性能，本章设计并实现了一个容错存储原型系统。该系统有三个特点：（1）基于开源分布式存储中心的离散事件模拟器 SimEDC 进行设计实现；（2）含有丰富的混合纠删码方案，包括已有的 HACFS<sup>[63]</sup>、EC-Fusion<sup>[91]</sup>以及 LRC&HH<sup>[92]</sup>，支持相应的扩展接口；（3）支持修复调度中的重建与迁移的融合。本章主在原型系统中对存储的可靠性进行分析，对不同的混合纠删码方案进行可靠行测试。

## 5.1 系统设计

容错存储原型系统设计主要包含以下模块：存储中心架构，节点故障模型，混合纠删码策略，节点放置策略，可靠性度量指标，事件处理模式。

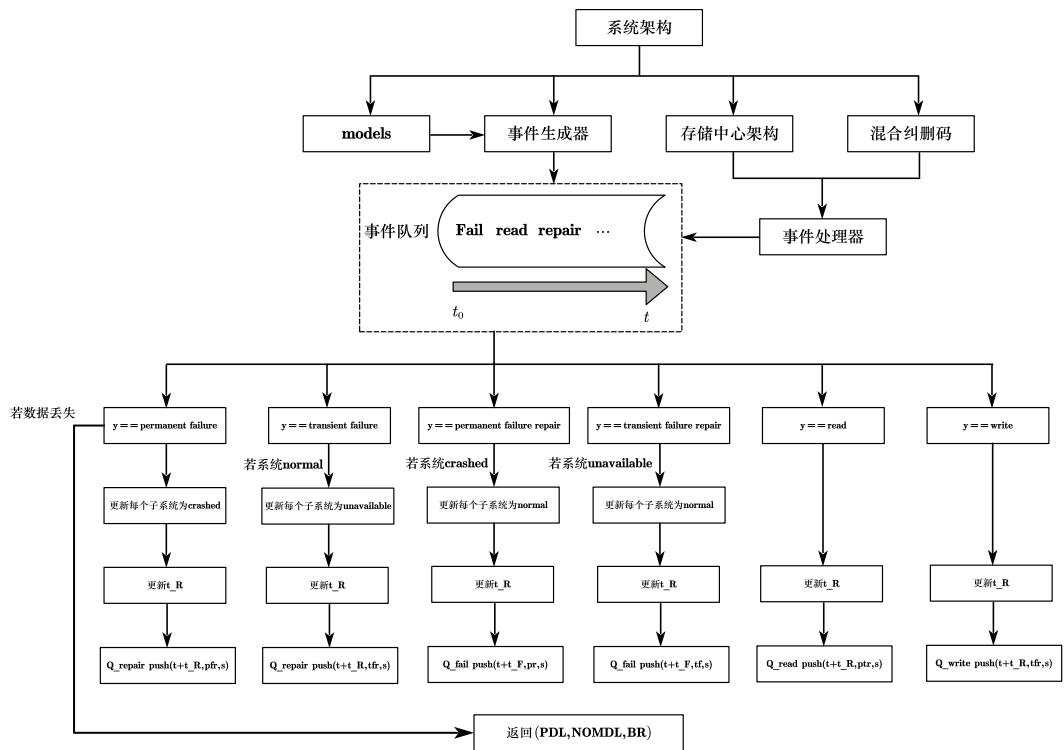


图 5-1 原型系统架构

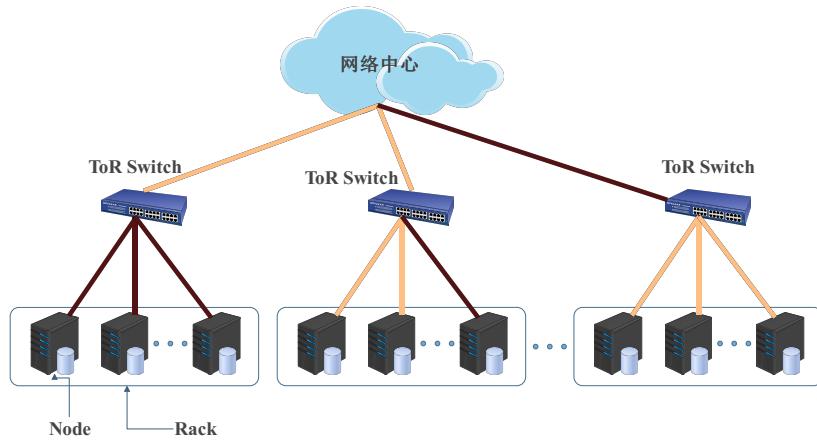


图 5-2 基于纠删码构建的存储系统架构

### 5.1.1 存储中心架构

常见的存储中心采用分层的架构来进行数据的存储和流转，如图 5-2 所示，它由多个数据柜（Rack）组成，每个数据柜上都运行着一定量的数据节点（Node）。每个节点连接着一个或多个磁盘，提供相应的存储空间进行数据服务。同一个数据柜的节点由数据交换机（ToR Switch）互联，而数据柜则由网络中心互联，整个存储中心由各层汇聚以及交换机组成。传统的存储中心的数据传输速度受制于数据柜之间的可用带宽，常常达不到理论上带宽最大值，并且网络中心经常需要同时进行计算与转发操作，故性能也会受到制约。每个节点可以连接多个磁盘，所以虽然磁盘本身有 I/O 限制，依然可以实现比单个带宽更高的并行速度，目前有两种架构形式可以缓解这种跨数据柜传输数据的带宽受限问题，分别为水平放置和层次放置。

为了从多角度模拟一个存储系统的运行可靠性，将纠删码每个条带的块放置在不同的节点和数据柜中，并且为每个有  $n$  块的条带考虑两种放置方案：

(1) 水平放置：一个条带的  $n$  个数据块被存储在  $n$  个不同的节点上，也就是说这些节点位于  $n$  个不同的数据柜上（即每个数据柜上有一个块）。这样的放置方式最大化地利用了节点和数据柜的容错能力。在这种情况下，修复一个丢失的数据块必须从其他数据柜搜索可用的数据块，从而产生了大量的跨机柜的修复传输流量，水平放置通常用于各种存储数据中心。

(2) 分层放置：一个条带的  $n$  个块被存储在  $n$  个不同的节点中，这些节点位于  $r$  ( $r < n$ ) 个数据柜中，每个节点有  $\frac{n}{r}$  个块，这里假设  $n$  能被  $r$  整除。这样的放置方式减少了跨机柜之间的修复数据传输量，因为在修复任意丢失的数据块时，可以利用同

一机柜中的可用块。相比之下，其可以容忍的机柜故障的能力比水平放置的方式要低得多。

以 RS(6,3) 码对水平和分层放置方式的特点进行阐述，即  $n = 6, k = 3$ ，将原数据文件分割成 3 块，并且再进行编码另外的 3 块，只需要其中任意 3 块即可恢复出原始的数据。具体如图 5-3 所示，假设一个节点想要修复其本地存储中的一个故障磁盘中的数据块，在水平放置的方式中，一个条带的六个块都被放置在一个不同的数据柜中，所以修复丢失的数据块将从另外三个数据柜中进行块搜索。另一方面，在分层放置中，可以在一个数据柜中放置两个块，修复丢失的数据块可以从同一数据柜中搜索一个数据块，再从其他数据柜中接收两个数据块，因此跨机柜的修复流量减少到两个数据块。可见分层放置比水平放置产生的跨机柜的修复流量更少。

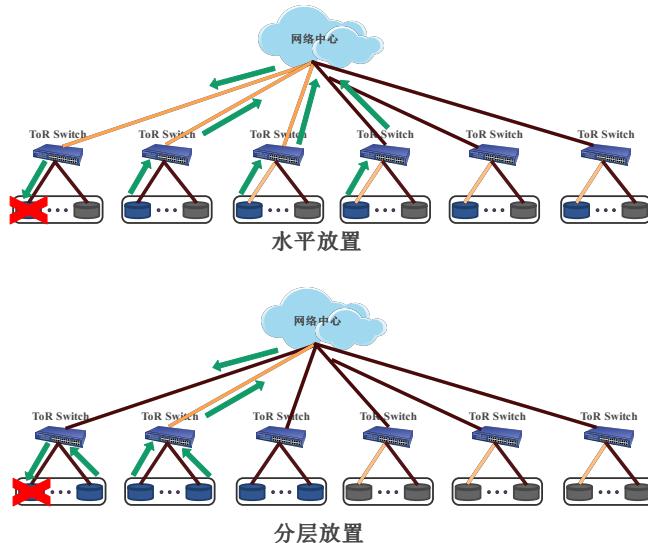


图 5-3 RS(6,3) 的水平放置和层次放置

### 5.1.2 故障模式建模

分布式存储系统在运行中发生故障事件的概率很高，本文专注于发生在三个层面的子系统故障：数据柜、节点和磁盘。故障本身可以是暂时性的，即一个子系统只是暂时不可用或者不可访问而不会造成实际的数据丢失（比如网络断开、系统重启或维护），也可以是永久性的，亦即一个子系统的故障会导致永久性的数据丢失（比如磁盘损坏）。故障可以进一步分类为是否独立，即子系统自身故障，或者关联故障，例如一些子系统由于一个共同的故障问题而发生同时故障。关联故障往往比独立故障要严重得多，例如当一个数据柜的交换机损坏时，那么机柜内的所有节点将失效，

从而造成数据丢失。一般比较常见的故障原因是停电，在这种情况下，相当一部分节点（高达 1%）将在通电重启后损坏，并导致永久性数据丢失。本文采用在分布式存储系统中比较常见的四种故障模式进行模拟：

- (1) 磁盘故障：本文专注于永久性的磁盘故障，在这种情况下，故障磁盘上的所有数据都会丢失。为了简单起见，不考虑只破坏磁盘部分数据的潜伏扇区错误，默认一旦损坏则整个磁盘数据丢失。
- (2) 节点故障：考虑暂时性和永久性的节点故障。对于前者，故障节点下的所有磁盘只是暂时不可用而没有造成数据丢失，对于后者，假设存储在该节点下的所有磁盘上的数据都全部且永久性的丢失。
- (3) 机柜故障：考虑短暂的数据柜故障，在这种情况下，故障机柜内的所有节点的数据变得无法访问，但没有产生数据丢失。
- (4) 关联故障：对于发生故障的数据柜，将存储在其下的所有节点都视为故障节点，也就是关联故障会使一个机柜内的所有节点瘫痪。本文专注于永久性的关联故障，例如故障节点会发生数据不可用，类似断电现象。

### 5.1.3 纠删码混合策略

目前比较成熟和前沿的混合纠删码技术一般是含有两种纠删码，并对冷数据和热数据进行优化存储。其中的技术包含了在两种纠删码进行高效切换的算法，在数据读取和写入过程中进行两种数据存储方式的转换，此外还有采取优先级队列的方式来区别冷热数据并且配以相应的纠删码方案。一般包含三个主要特征：`code` 选择的调整，适应性规则和 `code` 的切换。混合纠删码策略都对于冷热数据进行了相应的划分，并且用不同的纠删码对其进行存储上传和下载。所以在存储原型系统中需要构造相应的冷热数据划分算法，记录数据访问发生失败时的时间和节点位置，利用这些信息可以使用 LRU 算法来为冷热数据划分定制自适应算法。这里根据章节 4.3 对于算法 3 论述，为了将算法适应范围扩大，从而得到了算法 6。

### 5.1.4 纠删码接口

原型系统中的纠删码模块为数据编码提供了四个主要接口：编码（`encode`）、解码（`decode`）、编码上载（`upcode`）和编码下传（`downcode`）。编码操作的输入是一个原数据文件（`data file`）和编码方案（`codec`）作为输入，并为原数据文件中的分割块

---

**算法 6 容错存储原型系统动态自适应划分算法**


---

**Input:** Queue1,Queue2 and  $\eta$

```

1: for Queue2 头的每个插入请求 do
2:   if flag ≠ Fast Code and (writes/recoveries) <  $\eta$  then
3:     set flag = Fast Code;
4:     convert to Fast Code;
5:   end if
6: end for
7: for Queue1 头的每个插入请求 do
8:   if flag ≠ Compact Code and (writes/recoveries) ≥  $\eta$  then
9:     set flag = Compact Code;
10:    convert to Compact Code;
11:   end if
12: end for
13: for Queue2 尾的每个弹出请求 do
14:   if flag == Fast Code then
15:     set flag = Compact Code;
16:     convert back to Compact Code;
17:   end if
18: end for

```

---

生成一个奇偶校验块（parity file）。解码操作是在块丢失后的退化读取中调用的，或者作为磁盘或节点故障时启动重建工作的一部分。它还需要文件中丢失或损坏的块的索引（lost block index），并使用输入的编码方案从条带中剩余的数据和奇偶块中重建丢失的块。

自适应编码模块通过调用编码上载和编码下传的操作，来进行两种编码的切换。这两种转换操作在改变数据文件的编码方案时只更新相关的奇偶校验文件。编码上载操作将数据从 Fast Code 转换为 Compact Code，从而减少了奇偶校验文件的大小，达到降低存储开销的目的，此外它不需要读取数据文件。编码下传操作将数据从 Compact Code 转换为 Fast Code 表示，从而降低修复成本，它需要读取原数据文件和奇偶性校验块，具体如表 5-1 所示。

### 5.1.5 可靠性分析指标

本文采用的是广泛运用于评估系统可靠性的度量指标，包括数据丢失率 PDL (Probability of data loss)，归一化数据损失度 NOMDL (Normalized magnitude of data loss) 以及阻塞率 BR (Blocked ratio)。

PDL 衡量了一个存储中心在一定时间内发生任意数据块不可恢复的失效事件的

表 5-1 容错存储原型系统纠删码操作接口

接口操作	输入	输出
encode	data file, codec;	parity file
decode	data file, parity file codec, lost block index	recovered block
	parity file, original fast codec	parity file encoded with compact codec
upcode	new compact codec;	
downcode	data file, parity file original compact codec, new fast codec	parity file encoded with fast codec

可能性（即纠删码条带中永久失效的块的数量超过了可容忍的限制），一般计算方法为失效块占总数据块的比例。

NOMDL 它是由 Greenan<sup>[109]</sup> 等提出的，用来衡量预估的数据丢失量（以字节为单位）与存储容量进行正则化。它所具有的几个关键属性改进了现有的可靠性指标，具体计算方法如公式 5.1。

$$NOMDL = \frac{\text{avg\_num\_lost\_chunks}}{\text{num\_strips} \times \text{code\_n}} \quad (5.1)$$

BR 衡量的是由于存储一个数据块的子系统的短暂或永久的故障而不能直接访问该块的时间比例。但是这样一个无法访问的块可能仍然可以从其他子系统中同一条带的其他可用块中恢复，但会产生重建该块的额外开销。因此，BR 的值可以代表在正常模式下（即没有故障）无法直接访问一个块的时间，具体计算方法如公式 5.2。

$$BR = \frac{\text{sum\_unavail\_time}}{\text{num\_chunks} \times \text{mission\_time}} \quad (5.2)$$

### 5.1.6 事件处理

原型系统中的每个故障事件或修复事件都用一个由三个字段组成的元组来表示。  
(1) 事件发生时的时间戳 time  $t$  (2) 事件类型 type  $y$  (3) 与事件相关的子系统 subsystem  $s$ 。原型系统会将所有的事件存储在一个事件队列中，事件的发生机制遵从于统计模型概率表（见表 5-2），该队列以优先级队列的方式进行实现，并且返回具有最小时

间戳（亦即最新发生）的事件，供事件处理程序进行相应处理。原型系统将分别处理永久性故障和瞬时性故障，这里考虑四种事件类型：（1）永久性故障，（2）瞬时性故障，（3）永久性故障修复，以及（4）瞬时性故障修复。

故障处理。在模拟过程中，每个子系统（即机柜、节点和磁盘）都与三种状态之一有关。（1）正常（即没有发生故障）（2）不可用（即发生瞬时故障）和（3）崩溃（即发生永久性故障）。就严重程度而言，正常是最不严重的，不可用处于前者中间状态，而崩溃是最严重的。假设如果一个子系统发生故障，只有当状态变得更加严重时，其状态才会被更新。也就是说一个正常或不可用的状态在永久性故障中会变成崩溃状态，或者一个正常的状态在暂时性故障中会变成不可用状态。而崩溃的状态会依旧保持不变，同时在一个分层的分布式存储中心里，它所有的连接上它的子系统将继承同样的状态，如果一个节点崩溃了，那么附着在该节点上的所有磁盘也会崩溃；如果一个机架（即节点）不可用，那么该机架内的所有节点和磁盘（即所有附着的磁盘）如果原本是正常的，也将变成不可用。

原型系统的整个运行流程就是在不断地处理来自事件队列的事件作业。在收到一个永久性的故障事件后，它检查存储在该子系统中的每一个块是否可以被足够数量的同一条带的可用块修复。如果不可以，它就得出结论认为数据丢失，并返回当前迭代的可靠性指标。如果没有数据丢失或收到一个瞬时故障事件，原型系统为故障子系统触发一个相同类型（即永久或瞬时）的修复事件，并将该事件插入到事件队列中，供以后修复处理。

修复处理。在将修复事件插入事件队列之前，原型系统计算出修复永久性或暂时性故障所需的修复时间。对于永久性故障，修复时间的计算方法是将所有故障块的跨机柜修复流量总量除以可用的跨机柜带宽。对于瞬时故障，修复时间由统计模型或相应子系统的事件追踪来决定。

读取和写入。当读取和写入事件发生时，将根据算法 6 的方法进行纠删码的决策并且计算出修复和读取时间。对于读取首先判断节点是否处于故障状态，然后再进行读取时间和纠删码切换的时间计算，对于写入同理，区别在于不需要切换，需要计算跨柜放置节点数据的传输流量。

当从事件队列中收到一个修复事件时，原型系统将相关子系统状态更新为正常状态。此外，如果任何子系统有相同的故障类型，同时也会将其状态更新为正常状态。例如，如果一个崩溃（标记为 unavailable）的节点被修复，则与其相关的任何一个

一个崩溃的磁盘也被修复，其状态变成正常。最后，原型系统为子系统创建下一个相同类型（即永久或短暂）的故障事件，并将该事件插入到事件队列中，以便以后处理故障。

本文采用的默认故障和修复事件统计模型概率表的如表 5-2 所示。其中  $W = (\beta, \eta, \gamma)$  表示 Weibull 分布且带有形状参数  $\beta$ ，尺度参数  $\eta$ ，位置参数  $\gamma$ 。 $EXP(\lambda)$  表示带有参数  $\lambda$  的指数分布。

表 5-2 默认故障和修复事件统计模型表

故障类型	故障时间	修复时间
Permanent disk failures	$W(1.12, 10\text{years}, 0)$	$\frac{\text{cross-rack repair traffic}}{\text{cross-rack bandwidth}}$
Permanent node failures	$EXP\left(\frac{1}{125\text{months}}\right)$	
Transient node failures	$EXP\left(\frac{1}{4\text{months}}\right)$	$EXP\left(\frac{1}{0.25\text{hours}}\right)$
Transient rack failures	$EXP\left(\frac{1}{10\text{years}}\right)$	$W(1, 24\text{hours}, 10)$
Permanent correlated failures	$EXP\left(\frac{1}{1\text{year}}\right)$	$EXP\left(\frac{1}{15\text{hours}}\right)$

## 5.2 实验结果与分析

### 5.2.1 实验设置

系统基于 Python3 实现并进行数据运行测试，测试平台操作系统为 Ubuntu 16.04 LTS，配置为 3.70GHz Intel Core i5-9600K 处理器，16GB 内存，1 Gbit/s 网络接口。

实验中的主要输入参数详情包括，`place_type` 接收两种参数的输入：flat 和 hie，基于节 5.1.1 提到的节点放置策略进行实现；`code_type` 参数接收三种输入，分别为 hacfs (HACFS)，ecf (EC-Fusion)，lrchh (LRC&HH)，基于节 5.1.3 中论述的三种混合纠删码策略进行实现。实验中为了便于方便分析不同纠删码策略的可靠性数据，针对混合策略的参数输入保持不变，其他单一纠删码实验的参数进行改变进行对比实验。具体为 `cross-rack bandwidth` 设置为 1Gb/s，`place_type` 设置为 flat，`chunk_size` 设置为 64MB 等。

### 5.2.2 PDL 参数实验结果

如图 5-4 所示是 PDL 多次实验的结果，误差区间是  $\pm 4\%$ 。误差是通过实验得到的数据结果的范围进行确定的，不同的方案的结果范围不尽相同，同时也可以确定其在真实运行环境中的可靠性变化情况。其中蓝色柱子是混合纠删码策略的 PDL 实验数据，红色柱子是水平放置的 RS 码的水平放置数据，白色柱子是层次放置的 RS 码的实验数据，其中对于 y 轴作了对数化处理，这样结果看起来更为直观。可以看出相较于水平放置的纠删码层次放置的纠删码的 PDL 更低，也就意味着层次放置的失效块占总数据块的比例更低，主要由于集中放置时，跨机柜之间的数据传输更少，从而提高了可靠性。而三种混合纠删码策略因为在利用了重建与迁移技术上的优化，在平均水平拥有更加优秀的冗余能力，PDL 指数显著下降，其中 PDL 平均降低了 30%~35%.

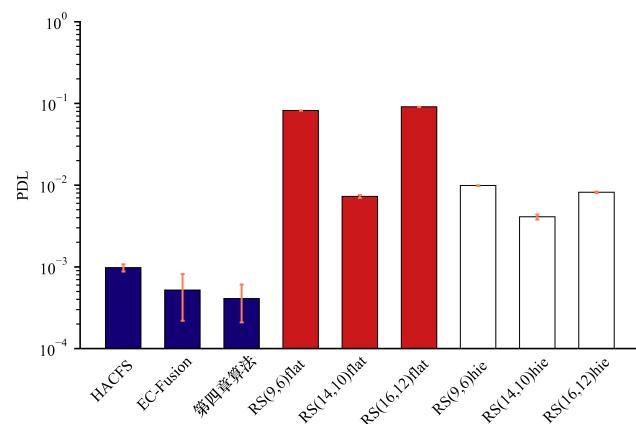


图 5-4 原型系统的 PDL 实验结果

### 5.2.3 NOMDL 参数实验结果

如图 5-5 所示是 NOMDL 多次实验的结果，误差区间是  $\pm 2.5\%$ 。与 PDL 测试结果相同的是，对于 y 轴同样进行了对数化处理，依然可以看出平均水平而言，层次放置的 RS 码依然比水平放置的 RS 码的 NOMDL 指数更低，也就意味着数据丢失量（以字节为单位）与存储容量的比例更低，整个系统的数据存储的可用数据比例更高，此外三种混合纠删码策略也比层次放置的单一纠删码的性能要更好，且 LRC&HH 方案比其他两种方案要更好，因为其本身利用了不同编码族的特性并进行了相应的切换算法的优化，并且得益于重建修复调度的优化，NOMDL 平均降低了 40%~42.5%。

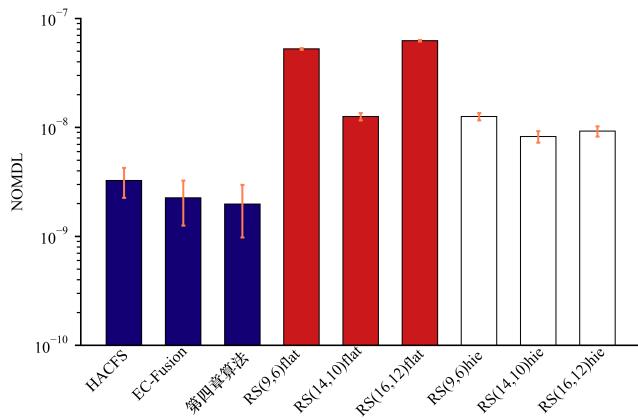


图 5-5 原型系统的 NOMDL 实验结果

#### 5.2.4 BR 参数实验结果

如图 5-6 所示是 BR 多次实验的结果，误差区间是  $\pm 1.7\%$ 。就平均水平而言，层次放置的 RS 码依然比水平放置的 RS 码的 BR 指数更低，也就意味着存储一个数据块的子系统由于短暂的或永久的故障而不能直接访问该块的时间的比例大大减少，整个系统数据存储的可访问时间占据全运行时间的比例更高。RS(16,12) 的冗余性能非常优越，得益于其有着更多的编码块冗余和在层次放置策略下更加低的跨机柜之间的数据传输量，此外三种混合纠删码策略也比层次放置的单一纠删码的性能要更好，且 LRC&HH 方案和 EC-Fusion 在 BR 指数上基本持平，说明两种编码方案的在系统的访问时间上的可靠性是相近的，BR 平均降低了 35.6%~37.8%。

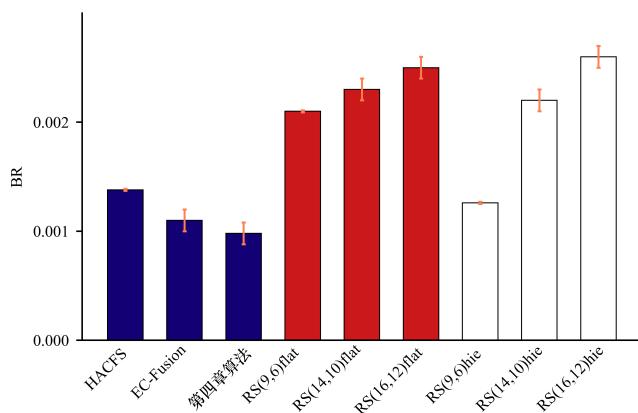


图 5-6 原型系统的 BR 实验结果

### 5.3 本章小结

在前两章的基础上,本章为了验证预先修复技术和混合纠删码修复技术在实际应用中的性能,设计并实现了一个容错存储原型系统。容错存储原型系统设计主要包含以下模块:存储中心架构,节点故障模型,混合纠删码策略,节点放置策略,可靠性度量指标,事件处理模式。实验结果表明,基于混合纠删码结构的容错存储原型系统具有较高的可靠性,PDL参数平均降低了30%~35%,NOMDL平均降低了40%~42.5%,BR平均降低了35.6%~37.8%。

## 第六章 总结与展望

### 6.1 论文工作总结

分布式存储系统通常采取存储超过原本数据 1 倍的冗余数据来保证数据可靠性以对抗节点故障而导致的数据丢失，进而通过冗余数据进行相关的计算还原出丢失的数据。但无论是多副本技术还是纠删码方式都会产生冗余数据，并且都有其固有的缺陷。本文主要对基于纠删码构建的存储系统的数据修复技术进行了研究实验，受到系统带宽性能和计算性能的影响，数据修复技术要综合考虑修复延迟，数据重建，修复块传输，带宽变化等多方面因素。本文在传统反应式修复的基础上改进了重建与迁移修复的传输方式，并且结合混合纠删码的策略以及相应的自适应算法完成对基于纠删码构建的存储系统的数据修复技术研究。现对本文的工作总结如下：

纠删码面临的最基本问题就是过量的修复开销：修复流量随着存储冗余度的降低而增加。目前，大部分的传统修复方法都是被动修复，只有在检测到节点故障后才会触发修复操作。如果可以提前预测即将发生的故障，就可以在任何实际故障发生之前主动修复任何即将发生的节点故障，以提高系统可靠性。针对上述问题，本文通过结合迁移与重建来设计实现预先修复的机制，用以修复 STF(Soon To Fail) 节点。首先，本文针对重建集与迁移集问题，设计了划分迁移集与重建集算法 SMSRS (Split Migration Set And Reconstruction Set) 算法，该算法主要使用数据热度队列与最小堆技术，根据数据访问热度高低处理待修复的块的出队修复顺序，利用最小堆不断优化当前出堆的网络带宽性能最优的可用节点进行迁移修复，然后通过贪心策略确定最佳的重建节点以及对应的重建块接收节点。其次，本文提出了 ISA (Improved SMFR Repair Algorithm) 算法来进行节点修复的调度。通过加入空闲节点 (idle nodes) 进行数据中继的方式可以对传输问题进一步优化。IBA 在每一轮修复中首先判断该任务是迁移还是重建，根据输入参数进行确认，采用局部最优修复，在此过程中链路带宽不断发生变化，从而实现最优化。通过调整系统总节点数，RS 码配置，磁盘带宽，网络带宽，条带数，块大小，空闲节点数等实验因素，实验结果表明，本文算法相较于传统的 FastPR 有着一定程度的改善，修复时间降低了 7.2%~13.2%。

在传统的存储系统中大多只采用一种纠删码进行数据的处理和存储，然而这种方式难以在保持低存储空间消耗的情况下降低退化读的延迟时间。本文在前文预先

修复的单一纠删码的基础上，进一步提出了一种可感知数据热度的负载动态自适应的混合纠删码（LRC&HH）数据修复方案。针对实际存储系统数据 I/O 的特点，将不同的数据进行分类，并且为了让不同的数据和读写任务工作负载情况匹配上适当的编码方案通过理论定量计算区分了读写负载模式和修复负载模式下的纠删码修复方案。根据相应的两种纠删码的切换算法，选取四个数据集，分别对读写负载，修复负载，整体负载和临界比进行了对比实验，实验结果表明，本文策略相较于 LRC&HH Random 降低了 6.8%~18.7% 的修复时间。

为了验证预先修复技术和混合纠删码修复技术在实际应用中的性能，本章设计并实现了一个容错存储原型系统。该系统有三个特点：（1）基于开源分布式存储中心的离散事件模拟器 SimEDC 进行设计实现；（2）含有丰富的混合纠删码方案，包括已有的 HACFS、EC-Fusion 以及 LRC&HH，支持相应的扩展接口；（3）支持修复调度中的重建与迁移的融合。容错存储原型系统设计主要包含以下模块：存储中心架构，节点故障模型，混合纠删码策略，节点放置策略，可靠性度量指标，事件处理模式。实验结果表明，基于混合纠删码结构的容错存储原型系统具有较高的可靠性，PDL 参数平均降低了 30%~35%，NOMDL 平均降低了 40%~42.5%，BR 平均降低了 35.6%~37.8%。

## 6.2 未来工作展望

本文主要针对基于纠删码构建的存储系统中的预先修复技术和混合纠删码技术进行了研究实验，提出了重建划分算法和修复调度算法以及混合纠删码的自适应算法。但是，本文针对的修复任务是单节点的修复任务，并未涉足于多节点的修复任务，在之后的研究中，需要将相应技术扩展到多节点的并行修复任务中，提升系统的可靠性。

## 参考文献

- [1] Rydning D R J G J, Reinsel J, Gantz J. The digitization of the world from edge to core [J]. Framingham: International Data Corporation, 2018: 16.
- [2] Wang S, Zhang Y, Zhang Y. A blockchain-based framework for data sharing with fine-grained access control in decentralized storage systems [J]. Ieee Access, 2018, 6: 38437-38450.
- [3] Sathiamoorthy M, Asteris M, Papailiopoulos D, Dimakis A G, Vadali R, Chen S, Borthakur D. Xoring elephants: Novel erasure codes for big data [J]. arXiv preprint arXiv:1301.3791, 2013.
- [4] Ghemawat S, Gobioff H, Leung S T. The google file system [C]//Proceedings of the nineteenth ACM symposium on Operating systems principles. 2003: 29-43.
- [5] Borthakur D, et al. Hdfs architecture guide [J]. Hadoop apache project, 2008, 53 (1-13): 2.
- [6] Weatherspoon H, Kubiatowicz J D. Erasure coding vs. replication: A quantitative comparison [C]//International Workshop on Peer-to-Peer Systems. Springer, 2002: 328-337.
- [7] Ford D, Labelle F, Popovici F I, Stokely M, Truong V A, Barroso L, Grimes C, Quinlan S. Availability in globally distributed storage systems [C]//9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10). 2010.
- [8] Huang C, Simitci H, Xu Y, Ogus A, Calder B, Gopalan P, Li J, Yekhanin S. Erasure coding in windows azure storage [C]//2012 USENIX Annual Technical Conference (USENIX ATC 12). 2012: 15-26.
- [9] Muralidhar S, Lloyd W, Roy S, Hill C, Lin E, Liu W, Pan S, Shankar S, Sivakumar V, Tang L, et al. f4: Facebook's warm {BLOB} storage system [C]//11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). 2014: 383-398.
- [10] Ovsianikov M, Rus S, Reeves D, Sutter P, Rao S, Kelly J. The quantcast file system [J]. Proceedings of the VLDB Endowment, 2013, 6(11): 1092-1101.

- [11] Reed I S, Solomon G. Polynomial codes over certain finite fields [J]. Journal of the society for industrial and applied mathematics, 1960, 8(2): 300-304.
- [12] Wu Y, Dimakis A G. Reducing repair traffic for erasure coding-based storage via interference alignment [C]//2009 IEEE International Symposium on Information Theory. IEEE, 2009: 2276-2280.
- [13] Kamath G M, Prakash N, Lalitha V, Kumar P V. Codes with local regeneration and erasure correction [J]. IEEE Transactions on information theory, 2014, 60(8): 4637-4660.
- [14] Li Y, He B, Luo Q, Yi K. Tree indexing on flash disks [C]//2009 IEEE 25th International Conference on Data Engineering. IEEE, 2009: 1303-1306.
- [15] Hua Y, Jiang H, Zhu Y, Feng D, Tian L. Smartstore: A new metadata organization paradigm with semantic-awareness for next-generation file systems [C]//Proceedings of the conference on high performance computing networking, storage and analysis. IEEE, 2009: 1-12.
- [16] Weil S A, Pollack K T, Brandt S A, Miller E L. Dynamic metadata management for petabyte-scale file systems [C]//SC'04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing. IEEE, 2004: 4-4.
- [17] Al-Fares M, Loukissas A, Vahdat A. A scalable, commodity data center network architecture [J]. ACM SIGCOMM computer communication review, 2008, 38(4): 63-74.
- [18] Greenberg A, Hamilton J R, Jain N, Kandula S, Kim C, Lahiri P, Maltz D A, Patel P, Sengupta S. VL2: a scalable and flexible data center network [J]. Communications of the ACM, 2011, 54(3): 95-104.
- [19] Guo C, Wu H, Tan K, Shi L, Zhang Y, Lu S. Dcell: a scalable and fault-tolerant network structure for data centers [C]//Proceedings of the ACM SIGCOMM 2008 conference on Data communication. 2008: 75-86.
- [20] Guo C, Lu G, Li D, Wu H, Zhang X, Shi Y, Tian C, Zhang Y, Lu S. Bcube: a high performance, server-centric network architecture for modular data centers [C]//Proceedings of the ACM SIGCOMM 2009 conference on Data communication. 2009:

- 63-74.
- [21] Shvachko K, Kuang H, Radia S, Chansler R. The hadoop distributed file system [C]// 2010 IEEE 26th symposium on mass storage systems and technologies (MSST). Ieee, 2010: 1-10.
- [22] 王意洁, 许方亮, 裴晓强. 分布式存储中的纠删码容错技术研究 [J]. 计算机学报, 2017, 40(1): 236-255.
- [23] Lakshman A, Malik P. Cassandra: a decentralized structured storage system [J]. ACM SIGOPS Operating Systems Review, 2010, 44(2): 35-40.
- [24] Gill N K, Singh S. A dynamic, cost-aware, optimized data replication strategy for heterogeneous cloud data centers [J]. Future Generation Computer Systems, 2016, 65: 10-32.
- [25] Gai-Mei G, Shang-Wang B. Design and simulation of dynamic replication strategy for the data grid [C]//2012 International Conference on Industrial Control and Electronics Engineering. IEEE, 2012: 901-903.
- [26] Patterson D A, Gibson G, Katz R H. A case for redundant arrays of inexpensive disks (raid) [C]//Proceedings of the 1988 ACM SIGMOD international conference on Management of data. 1988: 109-116.
- [27] Chen P M, Lee E K, Gibson G A, Katz R H, Patterson D A. Raid: High-performance, reliable secondary storage [J]. ACM Computing Surveys (CSUR), 1994, 26(2): 145-185.
- [28] Lin W, Chiu D M, Lee Y. Erasure code replication revisited [C]//Proceedings. Fourth International Conference on Peer-to-Peer Computing, 2004. Proceedings. IEEE, 2004: 90-97.
- [29] Huang S, Hou H, Yu X. A lower bound on disk reads for single information disk failure recovery and one recovery scheme for evenodd (p, 3) [C]//2019 Ninth International Workshop on Signal Design and its Applications in Communications (IWSDA). IEEE, 2019: 1-5.
- [30] Xu L, Bruck J. X-code: Mds array codes with optimal encoding [J]. IEEE Transactions on Information Theory, 1999, 45(1): 272-276.

- [31] Roth R M, Lempel A. On mds codes via cauchy matrices [J]. IEEE transactions on information theory, 1989, 35(6): 1314-1319.
- [32] Xia H, Chien A A. Robustore: a distributed storage architecture with robust and high performance [C]//SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing. IEEE, 2007: 1-11.
- [33] Kubiatowicz J, Bindel D, Chen Y, Czerwinski S, Eaton P, Geels D, Gummadi R, Rhea S, Weatherspoon H, Weimer W, et al. Oceanstore: An architecture for global-scale persistent storage [J]. ACM SIGOPS Operating Systems Review, 2000, 34(5): 190-201.
- [34] Blaum M, Bruck J, Vardy A. Mds array codes with independent parity symbols [J]. IEEE Transactions on Information Theory, 1996, 42(2): 529-542.
- [35] Plank J S. The raid-6 liber8tion code [J]. The International Journal of High Performance Computing Applications, 2009, 23(3): 242-251.
- [36] Hafner J L, Deenadhayalan V, Rao K, Tomlin J A. Matrix methods for lost data reconstruction in erasure codes. [C]//FAST: volume 5. 2005: 15-30.
- [37] Li P, Li J, Stones R J, Wang G, Li Z, Liu X. Procode: A proactive erasure coding scheme for cloud storage systems [C]//2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS). IEEE, 2016: 219-228.
- [38] Blaum M, Brady J, Bruck J, Menon J. Evenodd: An efficient scheme for tolerating double disk failures in raid architectures [J]. IEEE Transactions on computers, 1995, 44(2): 192-202.
- [39] Corbett P, English B, Goel A, Grcanac T, Kleiman S, Leong J, Sankar S. Row-diagonal parity for double disk failure correction [C]//Proceedings of the 3rd USENIX Conference on File and Storage Technologies. San Francisco, CA, 2004: 1-14.
- [40] Feng J, Chen Y, Summerville D. Eeo: An efficient mds-like raid-6 code for parallel implementation [C]//2010 IEEE Sarnoff Symposium. IEEE, 2010: 1-5.
- [41] Huang C, Xu L. Star: An efficient coding scheme for correcting triple storage node failures [J]. IEEE Transactions on Computers, 2008, 57(7): 889-901.
- [42] Feng J B, Wu H C, Tsai C S, Chu Y P. A new multi-secret images sharing scheme

- using lagrange's interpolation [J]. Journal of Systems and Software, 2005, 76(3): 327-339.
- [43] Ahlswede R, Cai N, Li S Y, Yeung R W. Network information flow [J]. IEEE Transactions on information theory, 2000, 46(4): 1204-1216.
- [44] Dimakis A G, Godfrey P B, Wu Y, Wainwright M J, Ramchandran K. Network coding for distributed storage systems [J]. IEEE transactions on information theory, 2010, 56(9): 4539-4551.
- [45] Wu Y, Dimakis A G, Ramchandran K. Deterministic regenerating codes for distributed storage [C]//Allerton conference on control, computing, and communication. IEEE Press Washington DC, 2007: 1-5.
- [46] Calder B, Wang J, Ogus A, Nilakantan N, Skjolsvold A, McKelvie S, Xu Y, Srivastav S, Wu J, Simitci H, et al. Windows azure storage: a highly available cloud storage service with strong consistency [C]//Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. 2011: 143-157.
- [47] Huang C, Chen M, Li J. Pyramid codes: Flexible schemes to trade space for access efficiency in reliable data storage systems [J]. ACM Transactions on Storage (TOS), 2013, 9(1): 1-28.
- [48] 周松, 王意洁. EXPyramid: 一种灵活的基于阵列结构的高容错低修复成本编码方案 [J]. 计算机研究与发展, 2011, 1.
- [49] Woitaszek M, Tufo H M. Tornado codes for maid archival storage [C]//24th IEEE Conference on Mass Storage Systems and Technologies (MSST 2007). IEEE, 2007: 221-226.
- [50] Gallager R. Low-density parity-check codes [J]. IRE Transactions on information theory, 1962, 8(1): 21-28.
- [51] Hafner J L. Weaver codes: Highly fault tolerant erasure codes for storage systems. [C]//Fast: volume 5. 2005: 16-16.
- [52] Hafner J L. Hover erasure codes for disk arrays [C]//International Conference on Dependable Systems and Networks (DSN'06). IEEE, 2006: 217-226.
- [53] Shah N B, Rashmi K, Kumar P V. A flexible class of regenerating codes for distributed

- storage [C]//2010 IEEE International Symposium on Information Theory. IEEE, 2010: 1943-1947.
- [54] Li J, Yang S, Wang X, Li B. Tree-structured data regeneration in distributed storage systems with regenerating codes [C]//2010 Proceedings IEEE INFOCOM. IEEE, 2010: 1-9.
- [55] 许方亮, 王意洁, 裴晓强. NTar: 基于网络拓扑的纠删码树型修复方法 [J]. 计算机研究与发展, 2013, 2.
- [56] Weidong S, Yijie W, Xiaoqiang P. Tree-structured parallel regeneration for multiple data losses in distributed storage systems based on erasure codes [J]. China Communications, 2013, 10(4): 113-125.
- [57] Xiang L, Xu Y, Lui J C, Chang Q. Optimal recovery of single disk failure in rdp code storage systems [J]. ACM SIGMETRICS Performance Evaluation Review, 2010, 38 (1): 119-130.
- [58] Khan O, Burns R C, Plank J S, Pierce W, Huang C. Rethinking erasure codes for cloud file systems: minimizing i/o for recovery and degraded reads. [C]//FAST: volume 20. 2012: 2208461-2208481.
- [59] Zhu Y, Lin J, Lee P P, Xu Y. Boosting degraded reads in heterogeneous erasure-coded storage systems [J]. IEEE Transactions on Computers, 2014, 64(8): 2145-2157.
- [60] Niu F, Xu Y, Zhu Y, Zhang Y. Phr: A pipelined heterogeneous recovery for raid6-coded storage systems [C]//2013 International Conference on Parallel and Distributed Computing, Applications and Technologies. IEEE, 2013: 325-331.
- [61] Zhu Y, Lee P P, Xiang L, Xu Y, Gao L. A cost-based heterogeneous recovery scheme for distributed storage systems with raid-6 codes [C]//IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012). IEEE, 2012: 1-12.
- [62] Rashmi K V, Shah N B, Gu D, Kuang H, Borthakur D, Ramchandran K. A "hitch-hiker's" guide to fast and efficient data reconstruction in erasure-coded data centers [C]//Proceedings of the 2014 ACM conference on SIGCOMM. 2014: 331-342.
- [63] Xia M, Saxena M, Blaum M, Pease D A. A tale of two erasure codes in {HDFS} [C]// 13th USENIX conference on file and storage technologies (FAST 15). 2015: 213-226.

- [64] Dimakis A G, Godfrey P B, Wainwright M J, Ramchandran K. The benefits of network coding for peer-to-peer storage systems [C]//Third Workshop on Network Coding, Theory, and Applications. 2007.
- [65] Pamies-Juarez L, Blagojevic F, Mateescu R, Gyuot C, Gad E E, Bandic Z. Opening the chrysalis: On the real repair performance of {MSR} codes [C]//14th USENIX conference on file and storage technologies (FAST 16). 2016: 81-94.
- [66] Ye L, Feng D, Hu Y, Liu Q. Hybrid-rc: Flexible erasure codes with optimized recovery performance and low storage overhead [C]//2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS). IEEE, 2017: 124-133.
- [67] Xie X, Wu C, Gu J, Qiu H, Li J, Guo M, He X, Dong Y, Zhao Y. Az-code: An efficient availability zone level erasure code to provide high fault tolerance in cloud storage systems [C]//2019 35th Symposium on Mass Storage Systems and Technologies (MSST). IEEE, 2019: 230-243.
- [68] Zhang Y, Wu C, Li J, Guo M. Tip-code: A three independent parity code to tolerate triple disk failures with optimal update complexity [C]//2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. IEEE, 2015: 136-147.
- [69] Vajha M, Ramkumar V, Puranik B, Kini G, Lobo E, Sasidharan B, Kumar P V, Barg A, Ye M, Narayananamurthy S, et al. Clay codes: Moulding {MDS} codes to yield an {MSR} code [C]//16th USENIX Conference on File and Storage Technologies (FAST 18). 2018: 139-154.
- [70] Ye M, Barg A. Explicit constructions of optimal-access mds codes with nearly optimal sub-packetization [J]. IEEE Transactions on Information Theory, 2017, 63(10): 6307-6317.
- [71] Mitra S, Panta R, Ra M R, Bagchi S. Partial-parallel-repair (ppr) a distributed technique for repairing erasure coded storage [C]//Proceedings of the eleventh European conference on computer systems. 2016: 1-16.
- [72] Li R, Li X, Lee P P, Huang Q. Repair pipelining for {Erasure-Coded} storage [C]//2017 USENIX Annual Technical Conference (USENIX ATC 17). 2017: 567-579.

- [73] Bai Y, Xu Z, Wang H, Wang D. Fast recovery techniques for erasure-coded clusters in non-uniform traffic network [C]//Proceedings of the 48th International Conference on Parallel Processing. 2019: 1-10.
- [74] Zhou H, Feng D, Hu Y. Bandwidth-aware scheduling repair technique in erasure-coded clusters design and analysis [J]. IEEE Transactions on Parallel & Distributed Systems, 2022(01): 1-1.
- [75] Liu C, Wang Q, Chu X, Leung Y W, Liu H. Esetstore: An erasure-coded storage system with fast data recovery [J]. IEEE Transactions on Parallel and Distributed Systems, 2020, 31(9): 2001-2016.
- [76] Weil S A, Brandt S A, Miller E L, Long D D, Maltzahn C. Ceph: A scalable, high-performance distributed file system [C]//Proceedings of the 7th symposium on Operating systems design and implementation. 2006: 307-320.
- [77] Xu L, Lyu M, Li Q, Xie L, Xu Y. {SelectiveEC}: Selective reconstruction in erasure-coded storage systems [C]//12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20). 2020.
- [78] Lin S, Gong G, Shen Z, Lee P P, Shu J. Boosting {Full-Node} repair in {Erasure-Coded} storage [C]//2021 USENIX Annual Technical Conference (USENIX ATC 21). 2021: 641-655.
- [79] Popa L, Yalagandula P, Banerjee S, Mogul J C, Turner Y, Santos J R. Elastic-switch: Practical work-conserving bandwidth guarantees for cloud computing [C]// Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM. 2013: 351-362.
- [80] Li S, Lan T, Ra M R, Panta R. Joint scheduling and source selection for background traffic in erasure-coded storage [J]. IEEE Transactions on Parallel and Distributed Systems, 2018, 29(12): 2826-2837.
- [81] Akhlaghi S, Kiani A, Ghanavati M R. A fundamental trade-off between the download cost and repair bandwidth in distributed storage systems [C]//2010 IEEE International Symposium on Network Coding (NetCod). IEEE, 2010: 1-6.
- [82] Gastón B, Pujol J, Villanueva M. A realistic distributed storage system that minimizes

- data storage and repair bandwidth [J]. arXiv preprint arXiv:1301.1549, 2013.
- [83] Hu Y, Lee P P, Zhang X. Double regenerating codes for hierarchical data centers [C]// 2016 IEEE International Symposium on Information Theory (ISIT). IEEE, 2016: 245-249.
- [84] Sipos M, Gahm J, Venkat N, Oran D. Network-aware feasible repairs for erasure-coded storage [J]. IEEE/ACM Transactions on Networking, 2018, 26(3): 1404-1417.
- [85] Chowdhury M, Kandula S, Stoica I. Leveraging endpoint flexibility in data-intensive clusters [J]. ACM SIGCOMM Computer Communication Review, 2013, 43(4): 231-242.
- [86] Dean J, Ghemawat S. Mapreduce: Simplified data processing on large clusters [J]. 2004.
- [87] Liu T, Alibhai S, He X. A rack-aware pipeline repair scheme for erasure-coded distributed storage systems [C]//49th International Conference on Parallel Processing-ICPP. 2020: 1-11.
- [88] Shen Z, Shu J, Lee P P. Reconsidering single failure recovery in clustered file systems [C]//2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, 2016: 323-334.
- [89] Shen Z, Shu J, Huang Z, Fu Y. Cluster-aware scattered repair in erasure-coded storage: Design and analysis [J]. IEEE Transactions on Computers, 2020.
- [90] Wei B, Xiao L M, Wei W, Song Y, Zhou B Y. A new adaptive coding selection method for distributed storage systems [J]. IEEE Access, 2018, 6: 13350-13357.
- [91] Qiu H, Wu C, Li J, Guo M, Liu T, He X, Dong Y, Zhao Y. Ec-fusion: An efficient hybrid erasure coding framework to improve both application and recovery performance in cloud storage systems [C]//2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 2020: 191-201.
- [92] Wang Z, Wang H, Shao A, Wang D. An adaptive erasure-coded storage scheme with an efficient code-switching algorithm [C]//49th International Conference on Parallel Processing-ICPP. 2020: 1-11.
- [93] Botezatu M M, Giurgiu I, Bogoleska J, Wiesmann D. Predicting disk replacement to-

- wards reliable data centers [C]//Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. 2016: 39-48.
- [94] Li J, Ji X, Jia Y, Zhu B, Wang G, Li Z, Liu X. Hard drive failure prediction using classification and regression trees [C]//2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. IEEE, 2014: 383-394.
- [95] Mahdisoltani F, Stefanovici I, Schroeder B. Proactive error prediction to improve storage system reliability [C]//2017 USENIX Annual Technical Conference (USENIX ATC 17). 2017: 391-402.
- [96] Zhu B, Wang G, Liu X, Hu D, Lin S, Ma J. Proactive drive failure prediction for large scale storage systems [C]//2013 IEEE 29th symposium on mass storage systems and technologies (MSST). IEEE, 2013: 1-5.
- [97] Shen Z, Li X, Lee P P. Fast predictive repair in erasure-coded storage [C]//2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, 2019: 556-567.
- [98] Cao T, Peng X, Zhang C, Al Tekreeti T K, Mao J, Qin X, Huang J. A popularity-aware reconstruction technique in erasure-coded storage systems [J]. Journal of Parallel and Distributed Computing, 2020, 146: 122-138.
- [99] Xu L, Lyu M, Li Z, Li Y, Xu Y. Deterministic data distribution for efficient recovery in erasure-coded storage systems [J]. IEEE Transactions on Parallel and Distributed Systems, 2020, 31(10): 2248-2262.
- [100] Zeng H, Zhang C, Wu C, Yang G, Li J, Xue G, Guo M. Fagr: an efficient file-aware graph recovery scheme for erasure coded cloud storage systems [C]//2020 IEEE 38th International Conference on Computer Design (ICCD). IEEE, 2020: 105-112.
- [101] Bhagwan R, Tati K, Cheng Y, Savage S, Voelker G M. Total recall: System support for automated availability management. [C]//Nsdi: volume 4. 2004: 25-25.
- [102] Li X, Li R, Lee P P, Hu Y. {OpenEC}: Toward unified and configurable erasure coding management in distributed storage systems [C]//17th USENIX Conference on File and Storage Technologies (FAST 19). 2019: 331-344.
- [103] Silberstein M, Ganesh L, Wang Y, Alvisi L, Dahlin M. Lazy means smart: Reduc-

- ing repair bandwidth costs in erasure-coded distributed storage [C]//Proceedings of International Conference on Systems and Storage. 2014: 1-7.
- [104] Rashmi K V, Shah N B, Gu D, Kuang H, Borthakur D, Ramchandran K. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the facebook warehouse cluster [C]//5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 13). 2013.
- [105] Zhang M, Han S, Lee P P. Simedc: A simulator for the reliability analysis of erasure-coded data centers [J]. IEEE Transactions on Parallel and Distributed Systems, 2019, 30(12): 2836-2848.
- [106] Ma Y, Nandagopal T, Puttaswamy K P, Banerjee S. An ensemble of replication and erasure codes for cloud file systems [C]//2013 Proceedings IEEE INFOCOM. IEEE, 2013: 1276-1284.
- [107] Friedman R, Kantor Y, Kantor A. Replicated erasure codes for storage and repair-traffic efficiency [C]//14-th IEEE International Conference on Peer-to-Peer Computing. IEEE, 2014: 1-10.
- [108] Narayanan D, Donnelly A, Rowstron A. Write off-loading: Practical power management for enterprise storage [J]. ACM Transactions on Storage (TOS), 2008, 4(3): 1-23.
- [109] Greenan K M. Reliability and power-efficiency in erasure-coded storage systems [M]. University of California, Santa Cruz, 2009.

## 攻读学位期间的成果

### · 论文

- (1) 第一作者. \*\*\*, 计算机软件著作权, 未发表, 登记号: \*\*\*.
- (2) 第一作者. \*\*\*, 计算机软件著作权, 未发表, 登记号: \*\*\*.
- (3) 计算机系统应用.2022. 外审中. 第一作者

### · 比赛

- (1) 第十七届中国研究生数学建模竞赛, 二等奖.

## 攻读学位期间的主要科研工作

(1) 国防科技创新特区支持. 江苏省高校优势学科建设资助项目. 江苏省高校自然科学研究项目, 项目号 (\*\*\*\*\*), “\*\*\*\*\*”, 2019-2021.