分布式存储系统中容错技术综述

高源君

摘 要 随着大数据时代的到来,海量数据的存储面临着巨大挑战。大规模分布式存储系统以其海量存储能力、高吞吐量、高可用性和低成本的突出优势取代了集中式存储系统成为主流系统。由于分布式存储系统中节点数量庞大，经常会产生各种类型故障。因此，必须采用容错技术来保证在部分存储节点失效的情况下，数据仍然能够被正常读取和下载，具有容错能力且节约存储资源的分布式存储编码成为大数据时代重点研究的核心技术之一。本文讨论了大数据背景下存储与可靠性的问题，从而引出数据容错对分布式存储的重要性。阐述了传统的 2 种数据存储容错技术，即多副本机制和ＭＤＳ码。重点介绍两种最新的分布式存储编码技术。面向数据的容错存储，针对存储中的节点修复问题，为大数据和移动数据的分布式存储编码提供理论基础，为海量数据的高效、可靠存储提供技术支撑。

关键词 纠删码；分布式存储系统；MDS 码；Piggybacking 纠删码

Overview of fault tolerance techniques in distributed storage systems

Gao Yuanjun

Abstract With the advent of the era of big data, massive data storage is facing great challenges. Large-scale distributed storage system has replaced centralized storage system as the mainstream system because of its outstanding advantages of massive storage capacity, high throughput, high availability and low cost. Due to the large number of nodes in a distributed storage system, various types of faults often occur. Therefore, fault-tolerant technology must be adopted to ensure that data can be read and downloaded normally even when some storage nodes fail. Distributed storage coding with fault-tolerant ability and saving storage resources has become one of the core technologies in the era of big data. This paper discusses the problems of storage and reliability in the context of big data, which leads to the importance of data fault tolerance for distributed storage. Two traditional data storage fault-tolerant technologies, namely multi-copy mechanism and MDS code, are introduced. This paper focuses on two new distributed storage coding technologies. Data-oriented fault-tolerant storage can repair problems of nodes in storage, provide theoretical basis for distributed storage coding of big data and mobile data, and provide technical support for efficient and reliable storage of massive data.

Key words Rt delete code; Distributed storage system; MDS code; Uncertainty remedy delete code

1 引言

在当前云计算时代，全球流量快速增长，互联网、物联网、移动终端、安全监控和金融等领域的数据量呈现出“井喷式”增长态势。存储在云服务器中数据的增长速率甚至超越了摩尔法则，云存储系统成为云计算关键组成部分之一。数据的飞速增长对存储系统的性能和扩展性提出了更苛刻的挑战。传统的存储系统采用集中式的方法存储数据，使得数据的安全性和可靠性均不能保证，不能满足大数据应用的需求。分布式存储系统以其巨大的存储潜力、高可靠性和易扩展性等优点成为大数据存储的关键系统并被推广应用到降低存储负荷、疏通网络拥塞等业务领域上，且其以高吞吐量和高可用性成为云存储中的主要系统。目前 Hadoop 分布式文件系统（HDFS）作为谷歌文件管理系统（DFS）[1]的开源实现已成为最主流的分布式系统，它被应用于许多大型企业，如脸书，亚马逊等。为了应对海量数据的存储需求，该类存储系统的规模往往非常庞大，一般包含几千到几万个存储节点不等。早在2014 年，中国互联网百度公司单个集群的节点数量就超过了 10000[2]。近两年，腾讯云的分布式调度系统 VStition 管理和调度单集群的节点数量可 10 0000。然而数量庞大的节点集群经常会产生如电源损坏、系统维修及网络中断等故障致使节点失效频发。据一些大型分布式存储系统的统计数据表明，平均每天都会有 2 %左右的节点发生故障[3]。在过去一年中，迅速发展的云计算市场不断涌出如谷歌云、亚马逊 AWS、微软 Azure 及阿里云等主流云服务器大型宕机事件。由此可见，全球的云服务器发生故障导致数据丢失的情况时有发生。显然，能够可靠存储数据并有效修复失效数据的容错技术对分布式存储系统的重要性不言而喻。因此如何及时有效地修复失效节点，确保数据能被正常地读取和下载就显得非常重要。据统计，在一个有 3000 个节点的脸书集群中，每天通常至少会触发 20 次节点修复机制。具有节点修复能力的数据容错技术是通过引入冗余的方式来保证分布式存储系统的可靠性，存储系统能够容忍一定数量的失效节点，即提高了系统的数据容错能力。为了保证用户访问数据的可靠性、维持分布式存储系统的容错性，需要及时修复失效节点。良好的容错技术要求存储系统具有低的冗余开销、低的节点修复带宽、低的编译码复杂度等特点。如何降低失效节点的修复带宽、降低磁盘 IO、降低存储系统编译码的复杂度、提高系统的存储效率成为分布式存储中研究的热门方向，因此，能够有效弥补多副本机制和 MDS 码不足的分布式纠删码应运而生。

传统最常见也是最早的存储容错是多副本机制和 MDS 码。通常，多副本机制引入冗余简单,但其存储负载很大,存储效率非常低,如 Google 文件系统和 Hadoop 文件系统等。传统的最大距离可分(Maximum Distance Separable,MDS)码结构可以实现分布式存储编码，比如里德－所罗门(Reed 一 Solomon，RS)码，以及 Google Colossus，HDFSRaid ，Microsoft Azure,Ocean Store 等。其存储成本更低,但是其拥有更高的修复成本和访问延迟，而且没有考虑存储开销、磁盘 I/O 开销等，因此并不适合用于大规模分布式存储系统。

（1）多副本机制是产生冗余最基本的方式。为了弥补数据失效带来的损失，确保存储系统中数据的完整性，将数据的副本存储在多个磁盘中，只要有一个副本可用，就可以容忍数据丢失。如果数据以最常见的 3 副本方式存储，那么原始数据会被复制到 3 个磁盘上，因此任何 1 个磁盘故障都可以通过剩余 2 个磁盘中任意 1 个来修复。显然，该 3 副本机制有效的存储空间理论上不超过总存储空间的 1/3,多副本机制显著降低了存储效率。

（2）如果系统编码设计的目标是在不牺牲磁盘故障容忍度的前提下最大化存储效率，那么存储系统采用 MDS 码来储存数据，因为它可以为相同的存储开销提供更高的可靠性。也就是说 MDS 码是一种能提供最优储存与可靠性权衡的纠删码。在一个分布式存储方案中,原始文件首先被分成 k 个数据块,接着编码生成 n 个编码块并存储在 n 个节点中。访问任意 l(l≥k)个节点,通过纠删码的译码就可恢复出原始文件，如果 l 一 k ，该码就满足 MDS 性质，最多能够容忍(n—k)个节点的失效。原始文件被 MDS 码编码生成 5 个数据块并放置到 5 个磁盘中。任意某个磁盘损坏，都能通过访问其他 4 个幸存磁盘中的 3 个来修复。MDS 码的 3 个主要缺点阻碍了它在分布式存储系统中的推广。首先,为了读取或写入数据 v 系统需要对数据进行编译码，由于 CPU 限制会导致高延迟和低吞吐量。根据调研，即使 Facebook 存储系统的集群中用 MDS 码只编码一半数据,修复流量也会使集群中的网络链路接近饱和。其次,MDS 码修复失效节点时必须访问多个编码块,而访问的这些编码块足以得到所有的原始数据,这

样的修复代价也太大了。最后,托管在云存储中的应用程序对磁盘Ⅰ/O性能很敏感，且由于大多数数据中心引入链路超额配置;因此带宽始终是数据中心内的有限资源,使得 MDS 码的节点修复在带宽开销和磁盘 I/O 开销方面都非常昂贵。

2 原理与优势

2.1 结合局部信息的宽条带纠删码

纠删码是分布式存储系统的一种低成本冗余机制，通过存储数据条带和奇偶校验块达到容错目的。宽条带最近被提出来抑制条带中奇偶校验块的比例，以实现极端的存储节省。然而，宽条带加重了修复成本，而现有能有效修复数据的纠删码方法并不能很好地解决宽条纹问题。Yuchong Hu 等人[5]，提出了组合局部性，这是第一个通过结合奇偶校验局部性和拓扑局部性来系统地解决宽条带修复问题的机制。用有效的编码和更新方案进一步增强了组合局部性。在 AmazonEC2 上的实验表明，与基于局部的技术相比，组合局部减少了 90.5%的单块修复时间，冗余低至 1.063×。[5]结合奇偶校验局部性和拓扑局部性的特点，以获得更好的权衡，从而使宽条纹实际适用。

2.2 Pigggybacking纠删码

相比于 RGC 和 LRC，Piggybacking 编码以其较低的计算复杂度、设计灵活等特点开始在数据容错技术中占有一席之地。编码设计完成后，它不改变原有系统的节点分布结构，不产生额外的存储负载、在修复失效数据时将底层码的译码替代为求解与失效数据相关的 Piggyback 方程。这样不仅显著降低了修复带宽，还有效降低了修复复杂度，为海量数据可靠、高效的存储提供了强有力的保障。Piggybacking 是降低分布式存储系统修复带宽的有效框架，特别是当系统满足 MDS、高码率和少量子条纹时。通过对 Piggybacking 修复率的分析，本文[6]发现 piggybacking 保护条带的比例 pp 是显著降低修复带宽的关键。

2.3 RGC纠删码

为了降低 MDS 码的节点修复带宽，Dimakis 等人受网络编码的启发将编码数据的修复建模为信息流图，从而提出了再生码，其约束条件是维持容忍磁盘故障的能力[7]。基于此模型来表征存储与带宽之间的最优权衡，即给定磁盘上存储的数据量大小,可以得到修复过程中需要传输数据量的最优下界。在信息流图中,所有的服务器可以分为源节点、存储节点和数据收集器,其中源节点表示数据对象产生的服务器。图 1 为 RGC 信息流图。

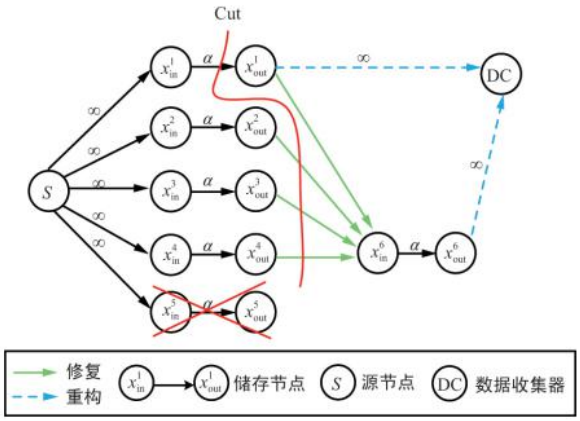

图 1

假设再生码是将大小为 M 的原始文件编码(一般为 RS 编码)存储(多播)到 n(n=6)个节点中,每个节点存储大小为 α 。特别是,在信息流图中源节点由一个顶点 S 表示,存储节点由 2 个节点 x_{in}^{i} ， x_{out}^{i} (1 ≤i≤n)表示，边缘的权重对应存储节点中存储的数据量。经过多播后，数据收集器 DC 连接剩余(n一1)个存储节点中任意 k 个足以恢复原来的数据对象,这表明 kα ≥M。当某个节点失效时,重构新的节点是访问幸存(n－1)个存储节点中的 d(d ≥k)个,即从每个节点下载大小为 β（β ≤ α）的数据进行修复。因此修复该故障节点所需读取和下载的数据量,也称修复带宽 γ=dβ 。综上所述,再生码的平均修复带宽 γ 小于文件大小 M。理论上,当连接节点数 d=n-1 时,再生码的修复带宽达到最小值[7]。

MDS 码节点的数据恢复是要新生节点接收来自服务器(数据提供者)的 k 个数据块后，在新生节点上进行编码产生节点新数据。但再生码是在数据提供者和新节点上都能进行编码,即数据提供者(存储节点)有不止一个编码段。当需要修复节点时，提供者先发送自己编码段的线性组合,新节点将接收到的编码段再次编码重构新的节点。假定任意 2 个磁盘都足以恢复原始数据的 MDS 码,即 k=2,每个磁盘存储一个包含 2 个编码段的编码块。对 MDS 码而言，恢复故障磁盘至少需要给新磁盘发送 2.个编码块(4 个编码段)。但数据经提供者编码后再发送给新磁盘的数据量可减少至 1.5 个编码块(3 个编码段),这样就能减小数据的传输量,降低 25%的带宽消耗。

3 研究进展

3.1 结合局部信息的宽条带纠删码

考虑一个分布式存储系统，它以跨多个存储节点的固定大小的块组织数据，这样擦除编码就以块为单位操作。根据存储工作负载的类型，用于擦除编码的块大小可能变化很大，从内存键值存储（即热存储）[8]中小至 4KB，到持久文件存储（即小型 I/O 请求成本）中高达 256MB[9]。擦除编码能以不同的形式构建，其中 RS 码是最流行的擦除码，并被广泛部署。RS 代码的工作原理是将 k 个固定大小未编码的数据块编码为 n-k 个相同大小的奇偶校验块。RS 代码是存储最优的，即 n 个块中的任意 k 个块都足以重构故障的 k 个数据块，而冗余是所有可能的擦除代码构造中最小的。我们把每一组 n 个块称为条带。分布式存储系统包含多个独立编码的条带，每个条带的 n 个块存储在 n 个不同的节点中，以提供对任何 n 个节点故障的容错能力。宽条带带来了三个问题。昂贵的修复成本，修复单个丢失的块的传统方法是从其他非故障节点中检索 k 个可用的块，这意味着带宽和 I/O 成本被放大了 k 倍。复杂的编码，随着 k 的增加，擦除编码的（每条带）编码开销变得越加严重。昂贵的更新代价，擦除编码的（每条带）更新开销非常大：如果同一条带的任意数据块已经被更新，那么所有 n-k 奇偶校验块都需要更新。

现有的关于擦除编码修复问题的研究已经产生了大量的文献，其中许多文献都集中于利用局部性来减少修复带宽，包括奇偶校验局部性和拓扑局部性。一个（n，k）RS 代码需要检索 k 个块来修复丢失的块。奇偶校验局部性增加了局部奇偶校验块，以减少幸存块的数量（以及修复带宽和 I/O）。其具有代表性的擦除代码方法是本地可修复代码(LRCs)。以 Azure-LRC[10]为例。给定三个可配置参数 n，k 和 r(r<k<n)，一个(n，k，r)Azure-LRC 把每个本地组 r 数据块编码(除了最后一组，可能少于 r 数据块)到一个本地奇偶校验块，所以修复丢失的块现在只需要访问 r 个幸存块(r<k)。它还包含从所有数据块编码的 n-k-$\left\lceil \frac{k}{r} \right\rceil$ 全局奇偶校验块。Azure-LRC 满足最大可恢复属性，并且可以容忍任意 n-k-$\left\lceil \frac{k}{r} \right\rceil$+1 节点故障。现有的擦除编码存储系统(包括 Azure-LRC)将条带的每个块放置在不同机架中的不同节点中。这提供了对相同数量的节点故障和机架故障的容忍度，但修复会导致大量的跨机架带宽，这通常比内机架带宽更有限。[11]利用拓扑局部性，通过在机架内定位修复操作来减少跨机架修复带宽，从而降低了机架级的容错能力。它们将条带块存储在机架内的多个节点中，并将修复操作分为机架内部和跨机架修复子操作。跨机架修复带宽根据最小冗余[11]可以证明是最小化的。一些类似的研究集中于通过集群内修复子操作来最小化跨集群修复带宽。我们将一个拓扑局部性方案定义为（n、k、z）TL，其中（n、k）RS 编码的块被放置在 z 机架（或集群）中。对于 k 较大的宽条带，无论是奇偶校验局部性（高冗余度）还是拓扑局部性（高修复惩罚），都不能有效地平衡冗余度和修复惩罚之间的权衡。

这促使我们结合这两种类型的特点，以获得更好的权衡，从而使宽条纹实际适用。我们将组合局部性机制定义为(n、k、r、z)CL，它在 z 机架上组合了(n、k、r)Azure-LRC 和(n、k、z)T。我们的组合局部性机制的主要目标是确定最小化跨机架修复带宽的参数（n、k、r、z），前提是：(1)可容忍的节点故障次数（用 f 表示）和(2)允许的最大冗余（用γ表示）。对于宽条带擦除编码，我们考虑表 1 中所示的典型容错级别的 k(例如，k=128)。在此，我们确保每个机架中条带的最大块数（用 c 表示）不能大于条带的可容忍节点故障数 f；否则，机架故障可能会导致数据丢失。因此，我们要求 c≤f。每个第一个 z-1 机架存储一条条带的 c 个块，最后一个机架存储 n-c(z-1)(≤c)剩余块。本方法专注于优化两种类型的修复操作：单块修复和全节点修复。这两种修复操作都假设每个失败的条带都有一个失败的块，就像之前的大多数研究 一样，包括那些在奇偶校验局部性[10]和拓扑局部性[11]上的块。对于有多个失败块的失败条带，我们采用传统的修复方法，检索 k 个可用的块来重构所有 RS 编码中的失败块。

Storage systems	(n,k)	Redundancy
Google Colossus [25]	(9,6)	1.50
Quantcast File System [49]	(9,6)	1.50
Hadoop Distributed File System [3]	(9,6)	1.50
Baidu Atlas [36]	(12,8)	1.50
Facebook f4 [47]	(14,10)	1.40
Yahoo Cloud Object Store [48]	(11,8)	1.38
Windows Azure Storage [34]	(16,12)	1.33
Tencent Ultra-Cold Storage [8]	(12,10)	1.20
Pelican [12]	(18,15)	1.20
Backblaze Vaults [13]	(20,17)	1.18

表 1

为了实现组合局部性的目标，我们从图 2 中观察到，组合局部性通过下载 r-1 数据块加上一个本

地奇偶校验块(即修复带宽为 r 块)来修复一个数据块。由于组合的局部性将一些 r 块放置在相同的机架中，因此它可以对每个机架中的块进行局部修复，从而减少跨机架修复带宽。直观地说，如果 c 增加（即，更多的条带块可以驻留在一个机架中），本地修复可以包含更多的块，从而进一步降低跨机架修复带宽。因此，本文的目标是找到最大可能的 c。因此，(n、k、r、z)CL 的构造是为了确保 c=f。然而，(n、k、r)LRC 有不同的结构，它们提供了不同水平的容错能力。因此，我们的想法是选择具有最高容错性的合适的 LRCs 结构。

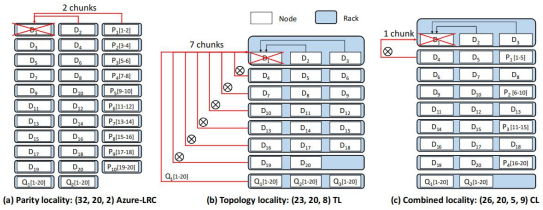

图 2

本文考虑了四个有代表性的 LRCs。

- Azure-LRC[10]：它计算出每个本地组的 r 个数据块的线性组合，并通过 RS 码计算出全局奇偶校验块。请注意，修复一个全局奇偶校验块需要检索 k 个块。
- Xorbas[12]：它与 Azure-LRC 的不同之处在于，它允许每个全局奇偶校验块最多可由 r 个块进行修复，这可能包括其他全局奇偶校验块和本地奇偶校验块。
- Optimal-LRC[13]：它将所有数据块和全局奇偶块分成大小为 r 的本地组，并向每个本地组添加一个本地奇偶块，以允许最多用 r 个块修复任何丢失的块。
- Azure-LRC+1[14]：它构建在 Azure-LRC 之上，通过为所有全局奇偶校验块添加一个新的本地奇偶校验块，允许本地修复任何丢失的全局奇偶校验块。

	(n,k,r)	$(16,10,5)$
Azure-LRC [34]	$f=n-k-\lceil k/r\rceil+1$	$f=5$
Xorbas [63]	$f\leq n-k-\lceil k/r\rceil+1$	$f=4$
Optimal-LRC [69]	$f\leq n-k-\lceil k/r\rceil+1$	$f=4$
Azure-LRC+1 [35]	$f=n-k-\lceil k/r\rceil$	$f=4$

表 2

表 2 显示了实际设置(n，k，r)=（16,10,5）[14]的可容忍节点故障数 f。注意，Xorbas 和 Optimal-LRC 给出了它们的 f 的上界，从表 2 可以看出，Azure-LRC 在相同(n，k，r)下的 f 最大，因此可以作为组合局部 LRC 的适当选择。

我们提供了（n、k、r、z）CL 的构造如下。在这里，我们关注一个条带，它有 k 个数据块的条带，具有固定数量的可容忍节点故障，但服从最大允许冗余γ(即$\frac{n}{k}\leq\gamma$)。构造包括两个步骤：(1)查找 Azure-LRC 的参数(n、k、r)(2)将所有 n 个块横跨在 z 个机架上进行局部修复操作。

组合局部性的每一组参数(n、k、r、z)产生相应的冗余和跨机架修复带宽值。我们还可以根据 k 和 f 得到 Azure-LRC 和拓扑局部性的值。对于 Azure-LRC，我们通过$n=k+\left\lceil\frac{k}{r}\right\rceil+f-1$和跨机架修复带宽作为 r 块（假设每个块存储在一个不同的机架中）。对于拓扑局部性，我们得到了其冗余 f=n-k 和跨机架修复带宽的影响，即机架数量减去 1。表 4 列出了 Azure-LRC、拓扑局域性和组合局域性的冗余性和跨机架修复带宽，分别表示为(n、k、r) Azure-LRC、(n、k、z)TL 和(n、k、r、z)CL。

表 2 显示通过结合奇偶校验局部性和拓扑局部性，在冗余和跨机架修复带宽之间的权衡方面优于 Azure-LRC 和拓扑局部性。以 f=4 为例。对于拓扑局部性，（132,128,33）TL 具有最小冗余 1.031×，但它的跨机架修复带宽达到 32 块，尽管许多机架执行本地修复操作。（140,128,15）Azure-LRC 通过奇偶校验局部性将跨机架修复带宽大大减少到 r=15 块，但其冗余度（1.094×）并不接近最小值。原因是 Azure-LRC 的冗余是$\propto(\frac{1}{r})$，而它的跨机架修复带宽是∝r，所以对于较小的跨机架修复带宽，r 应该很小，代价是产生更高的冗余。相比之下，对于组合局部性，（136,128,27,34）CL 不仅与最小块有更紧密的冗余（即 1.063×），而且进一步显著降低了跨机架修复带宽，与 Azure-LRC 相比减少了 60%，因为组合区域的跨机架修复带宽为$\propto(\frac{r}{f})$。

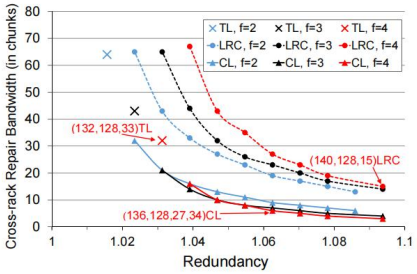

图 3

宽条纹是使用擦除编码的分布式存储的一个新概念，以实现极端的存储节省。本文提出了组合局部性，一种新的结合奇偶局部性和拓扑局部性的修复机制，以有效地解决了宽条纹的修复问题。

3.2 Pigggybacking纠删码

本文提出了一种基于piggybacking框架(REPB)的高效编码方法。REPB的修复比趋于0，接近理论边界，因为比例pp不再固定在1/2。此外，[6]所提出的REPB码在修复操作中具有较低的计算复杂度。[6]选择一个（n，k）的MDS码C_2作为REPB的基础码。r(=n−k)是奇偶校验号。引入两个参数s和p分别表示PP和MoP条纹的数量。图4描述了（s+p)个C_2初始的存储结构，如图三所示，所有表示被分为如下4个区域：

- 区域A包含PP条纹中的所有PPS表示。
- 区域B包含MoP条带中节点(k+1)的所有MPS符号和奇偶校验符号。
- 区域C包含PP条带中的所有奇偶校验符号。
- 区域D包含所有MoP条带的最后一个(r−1)奇偶校验符号。

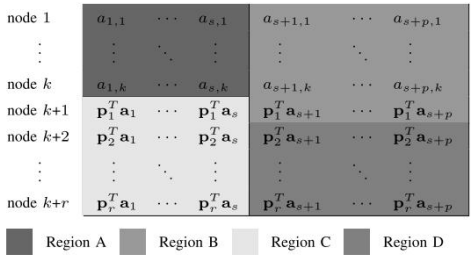

图 4

B区域中的所有符号都是可以自我维持的，对于A区中的PPS符号，构造了piggyback函数保护它们。Piggybacking约束允许添加在条带i上的piggyback函数只能包含来自条带{1、・・・、i−1}的原始消息。根据这一原则，我们将这些设计的piggyback函数嵌入到区域D的符号中。因此，piggyback函数的最大数量等于区域D为(r−1)p的大小。因此piggyback函数的最大数量等于区域D:(r−1)p的大小。当系统节点失效时，B区域缺失的p-mps符号可以通过B区域其他k行幸存的符号直接恢复，而丢失的PPS符号可以通过求解线性piggyback组合来恢复。根据线性代数理论，至少需要一种不相关的线性组合。因此，应满足以下必要条件，以同时恢复s缺失的PPS符号。

REPB的(r−1)p piggyback函数的显式构造如下：

（1）构造一个大小为$\left\lceil \frac{\mathrm{ks}}{(r-1)p} \right\rceil \times (r-1)p$的piggyback函数组:每一列对应一个piggyback函数；

（2）将区域A的所有PPS符号填充到piggyback数组中：区域A中的PPS符号形成一个k×s的数组，如图4所示。步骤2从k×s数组中获取这些符号,并将它们填充到piggybacking数组中。显然，如果ks不能被(r−1)p整除，那么这个附加数组的最后一行就不是满的；

（3）获取(r−1)p piggyback函数，并将其添加到区域D中：将所有PPS符号分配到piggyback数组后，将(r−1)列相加。结果是(r−1)p得到的piggyback函数，它们可以按任意顺序添加到第D区域中。假设(r−1)p叠加函数分别包含n1、n2、⋯、n(r−1)ppps符号。

假设(r−1)ppiggyback函数分别包含n1、n2、⋯、n(r−1)pps符号。显然，$\sum_{i=1}^{(\mathrm{r-1})p} n_i = \mathrm{ks}$值得注意的是，piggyback函数只是一些pps符号的总和。因此，恢复缺失的pps符号的计算可能非常简单。本文举例说明了piggyback系统功能的构造和一个失败的系统节点的修复程序。

例子：将（8,4）系统MDS代码作为基本代码。设置s=3和p=2。表示a、b、c、d、e为长度为4的5个输入消息向量。因此，初始存储结构为:

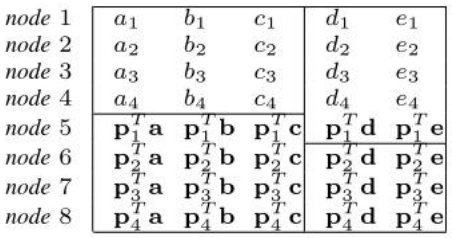

图 5

区域A中的PPS符号为{a1、a2、a3、a4}、{b1、b2、b3、b4}和{c1、c2、c3、c4}。将它们填充到一个2×6 piggyback阵列如下:

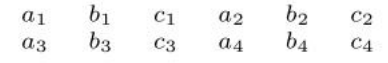

图 6

将每列的符号相加，得到6个叠加函数F1=a1+a3、

F2=b1+b3 、 F3=c1+c3 、 F4=a2+a4 、 F5=b2+b4 和 F6=c2+c4。因此，对 REPB 代码的说明如下:

node 1	a_1	b_1	c_1	d_1	e_1
node 2	a_2	b_2	c_2	d_2	e_2
node 3	a_3	b_3	c_3	d_3	e_3
node 4	a_4	b_4	c_4	d_4	e_4
node 5	$\mathbf{p}_1^T\mathbf{a}$	$\mathbf{p}_1^T\mathbf{b}$	$\mathbf{p}_1^T\mathbf{c}$	$\mathbf{p}_1^T\mathbf{d}$	$\mathbf{p}_1^T\mathbf{e}$
node 6	$\mathbf{p}_2^T\mathbf{a}$	$\mathbf{p}_2^T\mathbf{b}$	$\mathbf{p}_2^T\mathbf{c}$	$\mathbf{p}_2^T\mathbf{d}+F_1$	$\mathbf{p}_2^T\mathbf{e}+F_2$
node 7	$\mathbf{p}_3^T\mathbf{a}$	$\mathbf{p}_3^T\mathbf{b}$	$\mathbf{p}_3^T\mathbf{c}$	$\mathbf{p}_3^T\mathbf{d}+F_3$	$\mathbf{p}_3^T\mathbf{e}+F_4$
node 8	$\mathbf{p}_4^T\mathbf{a}$	$\mathbf{p}_4^T\mathbf{b}$	$\mathbf{p}_4^T\mathbf{c}$	$\mathbf{p}_4^T\mathbf{d}+F_5$	$\mathbf{p}_4^T\mathbf{e}+F_6$

图 7

假设第二个节点故障。缺失的 MPS 表示 d2、e2 可以通过下载 d1、d3、d4、$\mathrm{P}_1^T d$ 、e1、e3、d4、$\mathrm{P}_1^T e$ 来修复。缺失的 PPS 符号 a2、b2 和 c2 包含在 F4、F5 和 F6 中。因此，需要 $\mathrm{P}_3^T e$+F4、$\mathrm{P}_4^T d$ +F5、$\mathrm{P}_4^T e$+F6 和 a4、b4、c4 来恢复这些缺失的 PPS 符号。假设第 1 个系统节点失败。节点 1 的修复程序可以作为算法 1 进行操作。

Algorithm 1 The Repair Algorithm of REPB Codes

1 **Recover the missing MPS-symbols** $\{a_{s+i,l}\}_{i=1}^{p}$ **in Region B:** These missing MPS-symbols can be directly recovered with the kp surviving symbols in Region B;

2 **Download the piggyback functions containing the missing PPS-systems** $\{a_{i,l}\}_{i=1}^{s}$ **in Region A:** The missing PPS-symbols $\{a_{i,l}\}_{i=1}^{s}$ are included in s different piggyback functions denoted as $F_{i_1}, F_{i_2}, \cdots, F_{i_s}$, and $F_{i_1}, F_{i_2}, \cdots, F_{i_s}$ are embedded in s parity check symbols in Region D. Download these parity check symbols, and subtract the parity check components of $\{\mathbf{a}_{s+i}\}_{i=1}^{p}$. Then, $F_{i_1}, F_{i_2}, \cdots, F_{i_s}$ are left;

3 **Recover the missing PPS-symbols** $\{a_{i,l}\}_{i=1}^{s}$**:** Assume $F_{i_1}, F_{i_2}, \cdots, F_{i_s}$ involve $n_{i_1}, n_{i_2}, \cdots, n_{i_s}$ PPS-symbols, respectively. For F_{i_t} $(t \in \{1, \cdots, s\})$, other $(n_{i_t} - 1)$ surviving symbols in Region A are also involved in F_{i_t}. Download these surviving symbols from Region A, and subtract them from the corresponding piggyback functions. Thus, the missing MPS-symbols $\{a_{i,l}\}_{i=1}^{s}$ are obtained.

图 8

EFFICIENCY COMPARISON FOR DIFFERENT EXPLICIT CODES

parameters			RSR-II codes		REPB codes		
n, k	code rate	r	stripes	γ_1^{sys}	s, p	stripes	γ_2^{sys}
8, 4	0.5	5	5	0.6250	1, 1	2	0.6875
16, 12	0.75		5	0.6000	1, 1	2	0.6667
20, 16	0.8		5	0.6016	1, 1	2	0.6680
40, 36	0.9		5	0.6000	1, 1	2	0.6667
18, 9	0.5	9	15	0.5391	2, 1	3	0.5062
36, 27	0.75		15	0.5347	2, 1	3	0.5007
45, 36	0.8		15	0.5342	2, 1	3	0.5000
90, 81	0.9		15	0.5334	2, 1	3	0.5001
40, 20	0.5	20	37	0.5147	7, 2	9	0.3678
80, 60	0.75		37	0.5139	7, 2	9	0.3656
100, 80	0.8		37	0.5138	7, 2	9	0.3656
200, 180	0.9		37	0.5136	7, 2	9	0.3655
98, 49	0.5	49	95	0.5055	6, 1	7	0.2503
196, 147	0.75		95	0.5053	6, 1	7	0.2501
245, 196	0.8		95	0.5053	6, 1	7	0.2500
490, 441	0.9		95	0.5053	6, 1	7	0.2500
200, 100	0.5	100	197	0.5026	9, 1	10	0.1819
400, 300	0.75		197	0.5026	9, 1	10	0.1818
500, 400	0.8		197	0.5026	9, 1	10	0.1818
1000, 900	0.9		197	0.5025	9, 1	10	0.1818

图 9

与以往的设计相比，[6]提出的 REPB 码可以通过优化 PP 条带的比例来显著降低修复带宽。当奇偶校验节点数趋于无穷大时，一个失败的系统节点的修复比接近于零。对于修复失败的奇偶校验节点，与 RSR-II 码相比，REPB 码能够节省高达 40%的修复带宽。此外，REPB 码在数据重建和节点修复方面的复杂度较低。

3.3 RGC纠删码

分布式存储网络中一个重要的性能指标是它的安全性。RGC 除了数据存储容错外,还可以应用在信息安全上。它是一种用于提供数据可靠性和可用性的分布式存储网络编码,能实现高效的节点修复和信息安全。其中窃听者可能获得存储节点的数据，也可能访问在修复某些节点期间下载的数据(如果窃听者发现添加到系统中的新节点替换了失效的节点,那么它就可以访问修复期间下载的所有数据，这严重威胁了数据的安全和隐私)。在这种不利环境下,Dimakis 提出了 RGC 的准确构造从而实现了信息保密,即在不向任何窃听者透露任何信息的情况下,可以安全存储并向合法用户提供最大的数据量,同时得到了一般分布式存储系统保密容量的上限，并证明了这个界对于带宽受限的系统来说是紧密的,这在点对点分布式存储系统中是有用的[15]。考虑用分散数据保护一个分布式存储系统的问题，研究了节点间交互在减少总带宽方面的作用。当节点相互独立,交互是没有用的,并且总是存在一个最优的非交互方案。

纵然 RGC 达到了存储与带宽开销之间的最优权衡，但因为其需要繁琐的数学参数和复杂的编码理论基础,实现过程很难。目前 RGC 大多在有限域 GF(2')上进行多项式运算。加法在计算机上处理较为简单,然而乘法和除法却相对复杂,更有甚者需要用到离散对数和查表才能处理。这使得 RGC 的编译码复杂度很高,因而很难满足分布式存储系统对计算复杂度的要求。RGC 作为 MDS 码的改进版，拥有良好的理论基础,但是现有的绝大部分 RGC 编译码复杂度很高且码率低,所以迫切地提出高码率且编译码复杂度低的RGC,这具有积极的现实意义。若同时结合磁盘 I/O、数据存储安全和网络结构等方面构造，能进一步推广 RGC 的应用范围。

4 总结与展望

本文简单介绍了目前分布式存储系统中常见

的容错技术，主要分为两部分：①传统的容错技术即最早的多副本机制和 MDS 码；②分布式存储纠删码。分布式纠删码介绍了三种最新进展的纠删码，分别是结合局部信息的宽条带纠删码、Piggybacking 纠删码、RGC 纠删码。传统的容错技术与分布式存储编码的容错相比，其基本性能劣势比较明显，慢慢会被取代。不同的分布式存储编码都有自己优缺点，具体选择哪种容错方式要 根据分布式存储系统的需求来决定。

参 考 文 献

[1] Ghemawat S, Gobioff H, Leung S T. The Google file system [C]. Proceedings of the 19th ACM Symposium on Operating Systems Principles. 2003: 29-43.

[2] Shvachko K , Kuang H , Radia S , et al. The Hadoop Distributed File System[C]// IEEE Symposium on Mass Storage Systems & Technologies. IEEE, 2010.

[3] Schroeder B , Gibson G A . Disk Failures in the Real World: WhatDoes an MTTF of 1,000,000 Hours Mean to You? (CMU-PDL-06-111)[J]. 2006.

[4] Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2/, Retrieved in Jan 2021.

[5] Hu Y, Cheng L, Yao Q, et al. Exploiting Combined Locality for Wide-Stripe Erasure Coding in Distributed Storage[C]//19th {USENIX} Conference on File and Storage Technologies ({FAST} 21). 2021: 233-248.

[6] Yuan S, Huang Q, Wang Z. A repair-efficient coding for distributed storage systems under piggybacking framework[J]. IEEE Transactions on Communications, 2018, 66(8): 3245-3254.

[7] Dimakis A G, Godfrey P B, Wu Y, et al. Network coding for distributed storage systems[J]. IEEE transactions on information theory, 2010, 56(9): 4539-4551.L.

[8] Cheng, Y. Hu, and P. P. C. Lee. Coupling decentralized key-value stores with erasure coding. In Proc. Of ACM SoCC, pages 377–389, 2019.

[9] Rashmi K V, Shah N B, Gu D, et al. 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.

[10] Huang C, Simitci H, Xu Y, et al. Erasure coding in windows azure storage[C]//2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12). 2012: 15-26.

[11] Hou H, Lee P P C, Shum K W, et al. Rack-aware regenerating codes for data centers[J]. IEEE Transactions on Information Theory, 2019, 65(8): 4730-4745.

[12] Sathiamoorthy M, Asteris M, Papailiopoulos D, et al. Xoring elephants: Novel erasure codes for big data[J]. arXiv preprint arXiv:1301.3791, 2013.

[13] Tamo I, Barg A. A family of optimal locally recoverable codes [J]. IEEE Transactions on Information Theory, 2014, 60(8): 4661-4676.

[14] Kolosov O, Yadgar G, Liram M, et al. On fault tolerance, locality, and optimality in locally repairable codes[J]. ACM Transactions on Storage (TOS), 2020, 16(2): 1-32.

[15] Pawar S, El Rouayheb S, Ramchandran K. On secure distributed data storage under repair dynamics[C]//2010 IEEE International Symposium on Information Theory. IEEE, 2010: 2543-2547.