数据中心纠删码研究进展

张浩 1)

1) 华中科技大学计算机科学与技术学院 武汉 430074

摘 要 随着全球数据量暴涨，企业部署大规模集群或地理分布式存储系统来管理 PB 级的数据，以应对数据的巨大增长。一个主要的部署挑战是，随着存储系统规模的增长，它们更容易发生暂时性和永久性故障。在提供相同的容错能力的情况下，纠删码技术相对于副本存储所产生的冗余开销明显降低。然而，纠删码的本质仍然是利用数据冗余来提供可靠性保证，更少的冗余意味着能节省更多的成本；此外，校验值的计算会带来额外的延迟。因此，在数据中心中，低冗余、高性能、低延迟的纠删码方案仍然值得探索。本文就数据中心纠删码研究方向的几篇论文进行分析总结，梳理其研究进展。

关键词 纠删码；数据存储；数据中心；容错

Survey on Erasure Codes for Data Center

Hao Zhang[1)]

[1)](Department of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074)

Abstract With the global data volume soaring, enterprises deploy large-scale clusters or geographically distributed storage systems to manage petabytes of data to cope with the huge growth of data. A major deployment challenge is that with the growth of storage systems, they become more prone to temporary and permanent failures. In the case of providing the same fault-tolerance capability, the redundancy overhead of erasure code technology is significantly reduced compared with that of copy storage. However, the essence of erasure code is still to use data redundancy to provide reliability assurance. Less redundancy means that more costs can be saved; In addition, the calculation of the check value will cause additional delay. Therefore, in the data center, the low redundancy, high performance and low delay erasure code scheme is still worth exploring. This paper analyzes and summarizes several papers on the research direction of data center erasure codes, and sorts out their research progress.

Key words Erasure code; Data Storage; Data Center; Fault-tolerance

1 引言

随着全球数据的爆炸式增长，数据中心不得不大规模扩充存储能力以满足需求。随着存储系统规模的增长，它们更容易发生暂时性和永久性故障。因此，为了保持数据的可靠性和可用性，高效且易于实现的存储容错技术是必要的。目前的容错技术都依赖于冗余存储数据。

通常的冗余技术分为数据复制和纠删码。复制方法又称为多副本技术，其原理就是把数据复制多个副本并分别备份存储起来，并不涉及任何的编码和重构算法。由于其简单性，传统存储系统采用复制来提供容错。然而，复制的存储开销非常大，并带来了可扩展性问题。

纠删码是一种广泛采用的冗余技术，作为复制的替代方案，用于在大规模存储系统中实现低成本的可靠性保证。在许多纠删码结构中，Reed-Solomon（RS）code 在生产部署中应用最广泛。RS 码可以由两个可配置的整数参数 k 和 m 来构造。A（k，m）RS 码将 k 个原始固定大小块（称为数据块）编码为 m 个相同大小的编码块（称之为校验

块），使得 k+m 个数据/校验块中的任何 k 个都可以重建 k 个数据块。k+m 个数据/校验块的集合 n 称为条带，其中 k+m 个块分布在 k+m 个存储节点上，以容忍任何 m 个节点故障。大规模存储系统存储具有多个条带的数据，每个条带都是独立编码的。显而易见的是，当系统可以容忍任何 m 个节点故障时，纠删码的冗余开销 (k+m) / k×比复制 m+1×低得多。例如，为了容忍任何四个节点的故障，Facebook 使用冗余度为 1.4 倍的（10，4）RS 代码，而复制会导致冗余度为 5 倍，以容忍相同数量的节点故障。一般而言，从平均数据丢失次数角度衡量，与具有相同耐久性保证的复制相比，纠删码所产生的冗余开销明显更低。

尽管纠删码有效地减少了存储冗余，但存储从业者对纠删码中进一步减少冗余以实现存储的极大节省感兴趣；即使纠删码中冗余减少的一小部分也可以具有显著的成本效益。例如，据报道，Azure 通过将（6，3）RS 代码转换为（14，4）RS 代码，或等效地减少 14%的冗余，节省了数百万美元的生产成本。

为了实现极大的存储节省，宽条带纠删码被广泛研究。宽条带指的是条带的 k 值非常大，而 m 值则与传统纠删码部署一样，仅保留较小的值用于容错。宽条带适用于具有足够节点以支持非常大的 k 的大型存储系统，同时将冗余抑制到几乎等于 1。例如，VAST 考虑（k，m）=（150，4），从而使冗余度仅为 1.027×。考虑到全球不断增长的存储需求，宽条带对于在冷存储系统中实现极端的存储节约特别有吸引力，这些系统更强调低成本的存储可靠性和可用性，而不是高数据访问性能。

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 最先进的纠删码部署中常用参数

2 问题与挑战

2.1 宽条带纠删码生成问题

宽条带方看似美好，但如何生成宽条带仍然是尚待解决的问题。当节点发生故障时，纠删码要求通过从非故障节点检索多个可用块来重建故障节点中的任何丢失块，从而导致数据传输中的大量带宽成本。重建带宽成本随着 k 的增加而增加，尤其是对于宽条纹，重建丢失块的成本将会令人望而却步。在数据中心中，数据块在新写入时访问频率会更高，但随着数据块的老化，访问频率会降低，因此，纠删码可以通过分层方法来组织条带。在分层方法中，首先为了获取高性能，将与传统的纠删码一样，将新写入的数据块编码成具有小 k 的条带（称为窄条带）。而伴随着数据块的老化，窄条带随后被重新编码为宽条带，以实现高存储效率的可靠性。

显然，将窄条带重新编码为宽条带不可避免地会重新定位数据块并重新生成校验块，从而同样会导致数据传输中的大量带宽开销。如何降低这样的合并操作带来的开销是值得研究的问题。

2.2 数据中心存储扩展问题

此外，数据中心的存储系统经常会增加一些新的存储节点以满足不停增长的数据存储需求。在这种情况下，存储系统需要在现有的存储节点中重新分配已经形成构成纠删码的数据，以便在所有现有的和新添加的节点之间维持平衡的数据布局，从而利用所有节点之间最大可能的并行性。这意味着系统需要在新的数据布局下重新计算纠删码，这同样会带来大量的数据传输，形成存储扩展问题。如何降低存储扩展中纠删码重新计算带来的带宽开销值得关注。

2.3 纠删码修复问题

为了满足大数据处理对于内存的需求，数据中心同样需要大规模扩充内存，然而，这些内存并不总是处于工作状态中。Alibaba 和 Google 的研究报告指出，数据中心服务器的平均内存使用率为 60%，也就是说现代数据中心中存在有大量的闲置内存。如何利用这些闲置内存是企业迫切想要解决的问题。远端内存系统允许应用程序透明地访问本地内存以及属于远程机器的内存，可以有效利用闲置内存。与存储面临的情况类似，在包含数十万台

机器的数据中心中， 系统故障是普遍存在的因此，适用于数据中心的远端内存系统在设计时必须充分考虑容错能力。通常的纠删码方案是将一个 span 划分为多个块并分别存放在多个内存节点中， 这意味着当计算节点重建一个 span 时，需要通过网络从多个内存节点处获得数据块和校验块，多次的通信可能会造成网络的拥塞，且会存在尾延迟问题。如何降低纠删码修复所引起的带宽和尾延迟问题亟待解决。

3 研究进展

3.1 大规模纠删码存储的宽条带生成方案

为了在纠删码存储中实现极大的存储节省，纠删码数据被存储在宽条带中，在这种情况下，冗余度 (k+m) / k 随着 k 的增加而接近 1。由于宽条带的修复将产生显著的开销，通常被部署用于很少访问的冷数据，例如备份和存档数据或二进制大对象（BLOB），其访问频率随着其存储时间的增长而降低。这意味着宽条带将在系统中由新数据的窄条带合并生成，如上节所述，从窄条带生成宽条带会导致数据传输中的大量带宽开销。将两个（k，m）窄条带转换为新的（2k，m）宽条带包括以下步骤：

1) 重新分配两个窄条带的 2k 个数据块，以便将它们存储在 2k 个不同的节点中。
2) 将两个窄条带的一些数据和校验块迁移到负责生成宽条带的 m 个新校验块的一些节点（用于计算校验块）；m 个奇偶校验块随后分布在 m 个不同节点上。

数据块的重新分布和从窄条带生成校验块将产生大量的数据传输。为此文章的目标在于最小化宽条带生成带宽[1]。作者观察到，实际的大规模存储系统通常存储分布在多个节点上的大量条带。为了解决将（k，m）窄条带转换为（2k，m）宽条带的问题，文章的主要思想是从当前存储的众多窄条带中选择两个合适的（k，m）窄条带，使得两个选择的条带可以合并为新的（2k、m）宽带，而无需任何宽条带生成带宽，这样的合并文章称之为完美合并。一对符合完美合并的窄条带对如图 1 所示，它们之间具有如下两个属性：（i）它们的所有数据块都位于不同的节点中；以及（ii）它们的校验块都具有相同的编码系数并且驻留在相同的节点中。

完美合并利用了基于 Vandermonde 的 RS 码的关键特性，其中新的校验块可以从现有校验块本地计算得出。

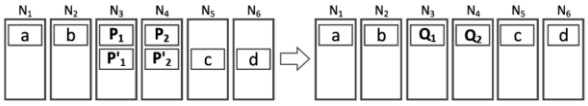

图 1 完美合并的窄条带对

虽然完美合并可以在不进行任何数据传输的情况下有效地生成单个宽条带，但是，选择满足完美合并的窄条带对取决于当前所有窄条带在底层存储系统中的放置方式。搜索大型存储系统的所有窄条带可能是一个耗时的过程。

文章将宽条带生成问题表述为二分图模型并证明了这样的窄条带对的存在性。文章表明，在给定足够多的窄条带的情况下，总是存在一种可以在没有任何宽条带生成带宽的情况下生成所有宽条带的最佳方案。然而，设计这样的最优方案最终归于二分图的最大匹配算法，而这样的算法时间复杂度通常会达到 $O(n^{2.5})$，当存储系统变得巨大时，这样的时间开销将变得不可接受。且在实际运行情况下，每种块放置模式可能具有不同数量的窄条带，因此，很难保证能够形成完整的块放置集合，从而使得最优方案不可行。为此本文提出两种启发式算法。

StripeMerge-G

StripeMerge-G 是一个贪婪启发式算法。假设一系列的窄条带在当前已经被存储，算法首先将会计算所有窄条带对的合并成本。如图 2 所示，算法一开始检查数据块的位置，发现数据块 b 和 c 都存储在 N_2 中，因此选择将块 c 传输到 N_6，如图中的 step1 所示。之后，算法需要确保具有相同编码系数的所有校验块位于相同节点（在 O（m）时间内）。如果不是，相应的校验块 a+b 和 c+d 将被收集到同一节点，如图中的 step2 所示，以便生成新的宽条带校验块 a+b+c+d。最后，数据块 d 将从 N_3 移动到可用节点 N_5，如 step3 所示。因此，图中的合并成本为 3。合并成本的计算可以在 O（k+m）时间内完成，而与条带总数相比，k+m 是一个相对较小的常数。

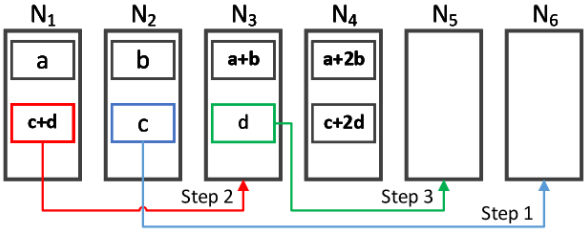

图 2 合并成本计算过程

之后，以计算得到的合并成本的升序合并窄条

带。即首先合并满足完美合并（即，合并成本为零）的窄条带对（如果存在），然后合并具有最小合并成本的剩余窄条带对。可以看出主要的时间复杂度在于对每一个窄条带对的合并成本的计算，时间复杂度是 O（(k+m) n^2），可以看出该算法的时间复杂度并没有比原来的完美合并算法低很多。

StripeMerge-P

从之前完美合并的窄条带对的要求中，我们会发现，对于满足完美合并的一对窄条带，一个必要条件是它们的所有校验块具有相同的编码系数并驻留在相同的节点中。因此，可以通过识别校验块完全对齐的窄条带对，以便快速获得满足完美合并的窄条带对。为此文章提出了另一种复杂度更低的校验对齐启发式算法 StripeMerge-P。

该算法定义了 i-部分校验块对齐窄条带对，这样的窄条带对具有对齐的 i 个校验块。在我们传输少量校验块后，这种窄条带对将会满足完美合并。

如图 3 所示，StripeMerge-P 在生成校验块时将校验块放置的元数据存储在哈希表中。哈希表存储键值项，其中每个键都指向任何 i 个校验块（1≤ i ≤ m）的某种分布，并且其值是具有相应校验块放置的条带的索引的列表。通过这种方式，我们可以使用哈希表来查找具有相同校验块放置的条带，以在 O（1）时间内形成 i-部分校验块对齐对。

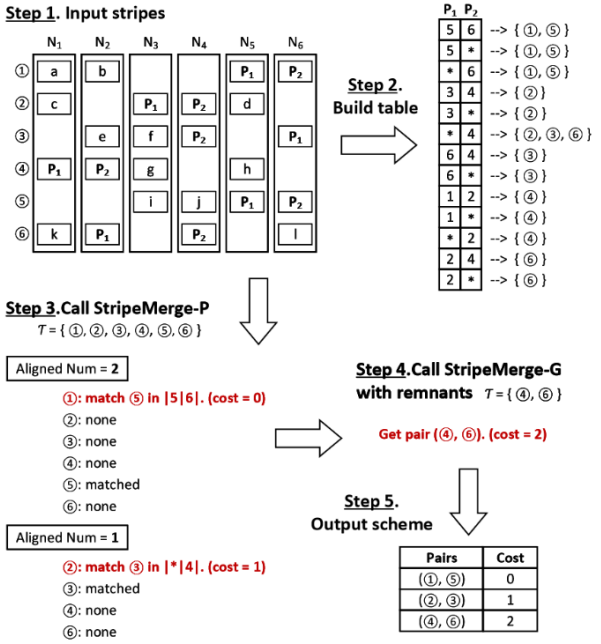

图 3 StripeMerge-P 算法流程示意

之后，算法将从 m 到 1 的降序为 i 赋值，并找出这样的 i-部分校验块对齐对进行合并，这一步之后，剩下的窄条带的校验块已经完全不对齐了，此时再使用 StripeMerge-G 算法对剩下的窄条带进行合并。

完美合并是零合并带宽开销的宽条带生成方案。然而适配所有的完美合并窄条带对将产生大量的时间开销，特别是对于大规模的数据中心，其开销将变得不可接受。文章提出的两种算法，特别是校验对齐算法 StripeMerge-P，可以在低带宽开销下在大型数据中心中快速实现窄条带合并操作。

3.2 宽条带纠删码的组合局部性修复

宽条带纠删码不仅需要解决生成所引起的带宽开销问题，其修复和更新的代价也因为过大的 k 值而变得不可接受，因此，如上节所述，宽条带通常被应用于存储冷数据。然而，其极低的冗余度也吸引研究人员设计其用于热数据存储的方案，虽然对于块大小较小的热存储工作负载，尽管其单块修复带宽远小于冷存储，但是在频繁访问下，大 k 会导致显著的尾延迟。而当 k 值过大时，编码过程更难以将宽条带的输入数据适应到 CPU 缓存中，从而导致显著的编码性能下降。图 4 显示了使用 Intel ISA-L 编码 API 的三个 Intel CPU 系列与 k 的编码吞吐量。在这里，我们将块大小修正为 64MiB，n-k = 4。我们看到编码吞吐量在 k = 4 到 k = 16 之间仍然很高，但是随着 k 从 k = 32 开始进一步增加，吞吐量急剧下降；例如，从 k = 4 到 k = 128，吞吐量下降了 43-70% 。

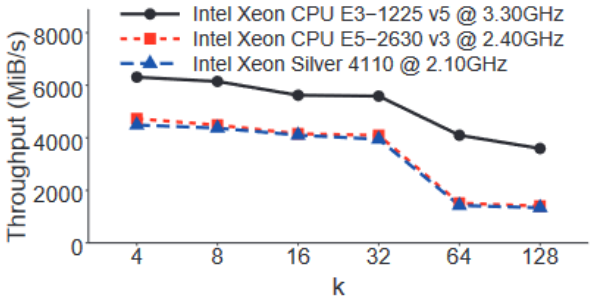

图 4 不同 k 值下的编码吞吐量

目前减少修复带宽主要有两种利用局部化的方法：(i)校验局部化，它引入额外的本地校验块以减少可用块的数量来检索以修复丢失的块；(ii)拓扑局部化，它考虑到系统拓扑的层次性质并执行本地修复操作以减轻跨机架(或跨集群)修复带宽。

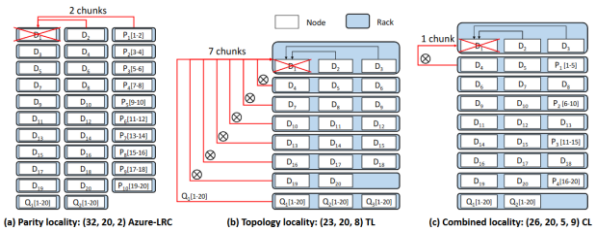

图 5 三种局部化方案示例

校验局部化添加了局部校验块，以减少修复需要的存活块的数量(从而减少修复带宽和 I/O)，从而修复丢失的块。其代表性的纠删码构造是局部可修代码(LRC)。以 Azure 的局部重建代码 Azure-LRC 为例，如图 5(a) 所示的(32,20,2) Azure-LRC，它有 20 个数据块(由 D_1、D_2、…、D_{20} 表示)。它有 10 个局部校验块，其中局部校验块 P`[i-j]是数据块 D_i，D_{i+1}，…，D_j 的线性组合。它还有两个全局奇偶块 Q_1[1-20]和 Q_2[1-20]，每个都是所有 20 个数据块的线性组合。所有以上 32 个块被放置在 32 个节点中，以容忍任何三个节点的故障。因此，(32,20,2) Azure-LRC 具有两个块的单块修复带宽(例如，修复 D1 需要访问 D2 和 P_1[1-2])，同时产生 1.6 × 的冗余。相比之下，(23,20) RS 代码也有 20 个数据块，可以容忍任何三个节点的故障。其单块修复带宽为 20 块，但冗余度仅为 1.15 × 。也就是说校验局部化减少了修复带宽，但是产生了高冗余。

现有的纠删码存储系统（包括 Azure LRC）将条带的每个块放置在位于不同机架中的不同节点中。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。如图 5(b) 所示的（23，20，8）TL，它将 20 个数据块和 3 个 RS 编码的校验块放置在位于 8 个机架中的 23 个节点中，以允许任何三个节点故障和一个机架故障。（23，20，8）TL 具有 1.15×的最小冗余，但传输七个跨机架块以修复丢失的块。拓扑局部化单个块跨机架修复带宽高于（32，20，2）Azure LRC，但实现了最小的冗余。

为此，文章提出了利用组合局部化的方案来减少修复过程的跨机架带宽，同时尽量降低冗余度[2]。组合局部化机制定义为（n，k，r，z）CL，它在 z 个机架上组合（n，k，r）Azure-LRC 和（n，j，z）TL，其中 r 为修复一个块所需的块数，z 为存储一个条带所需的机架数。组合局部化的主要目标即为确定最小化跨机架修复带宽的参数。

文章提出了一个宽条带纠删码存储系统 ECWide，并实现了两种修复操作的组合局部化：单块修复和全节点修复。

单块修复

ECWide 实现了两步组合局部性修复。考虑一个存储系统，它以给定 k、f 和 γ 的固定大小块组织数据。在步骤 1 中，ECWide 确定参数 n 和 r，然后将 k 个数据块编码为 n-k 个局部/全局校验块。在步骤 2 中，ECWide 为每个局部组选择(r + 1)/f 机架，并将每个局部组的所有 r + 1 块均匀地放入横跨这些机架的 r + 1 个不同节点中(即，每个机架的 f 块)。由于上述两个步骤确保单块修复的跨机架修复带宽最小化为(r + 1)/f-1 块，ECWide 只需要为修复操作提供以下细节。图 6 描述了 R1 机架中丢失的块 D_1 的修复。具体来说，ECWide 在 R_1 中选择一个节点 N_1(称为请求者)来负责重建丢失的块。它还选择机架 R_2 中的一个节点 N_4(称为本地修复器)来执行本地修复。然后，N_4 收集 R_2 中的所有块 D_5 和 P_1[1-5]，计算编码块 P_1[1-5]-D_4-D_5(假设 P_1[1-5]是 D_1，D_2，... ... ，D_5 的异或和)，并将编码块发送给请求者 N_1。最后，N_1 在 R_1 中收集数据块 D_2 和 D_3，并通过从接收到的编码块 P_1[1-5]-D_4-D_5 中减去 D_2 和 D_3 来求解 D_1。

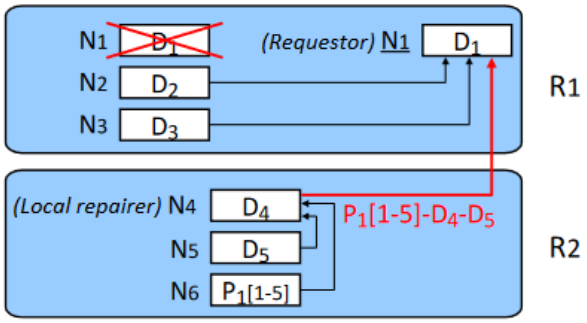

图 6　ECWide 单块修复

全节点修复

全节点修复可以被视为多条带的多个单块修复(例如，每条带一个丢失的块)，它可以并行化。然而，每个单块修复涉及一个请求者和多个本地修复者，因此多个单块修复可能选择与请求者或本地修复者相同的节点，从而使选择的节点超载并降低整体全节点修复性能。因此，我们的目标是选择尽可能多的不同节点作为请求者和本地修复器，以便有效地并行处理多个单块修复。为此，ECWide 设计了一种最近最少选择的方法来选择节点作为请求者或本地修复器，并通过双链表和 hashmap 实现

该方法。双向链表保存所有节点 ID，以跟踪最近选择了哪个节点或以其他方式选择了哪个节点，hashmap 保存节点 ID 和列表的节点地址。然后，我们可以通过简单选择列表的底部之一并在 O (1) 时间内通过 hashmap 更新列表来获得最近选择的节点作为请求者或本地修复器。

多节点编码

如前所述，随着条带的 k 值变大，单节点编码性能将严重下降。文章观察到如英特尔 ISA-L 和 QFS 经常将大型数据块(例如，64 miB)分割成更小的数据片，并使用硬件加速(例如，英特尔 ISA-L)或并行(例如，QFS)执行基于数据片的编码。为了编码一组 k 个数据切片，它们是 k 数据块的一部分，编码节点的 CPU 缓存从 k 数据块预取连续的切片。如果 k 很大，CPU 缓存可能无法保存所有预取的片，从而降低了后续片的编码性能。

由于校验块是数据块的线性组合，因此可以从多个部分编码块中组合奇偶校验块，而且同一机架内的节点之间的带宽通常是充足的。受此影响，文章提出一种多节点编码方案，该方案旨在实现宽条带的高编码吞吐量。其思想是将具有大 k 值的单个节点编码操作划分为跨越不同节点的用于小 k 的多个编码子操作。

图 7 描述了 k＝64 的多节点编码方案，假设要生成两个全局校验块 Q1[1-64]和 Q_2[1-64]。ECWide 首先将所有 64 个数据块均匀分布在同一机架中的四个节点 N_1、N_2、N_3 和 N_4 上。它允许每个节点（例如 N_1）将其 16 个本地数据块（例如 D_1、D_2、…、D_{16}）编码为两个部分编码的块（例如 Q_1[1-16]和 Q_2[1-16]）。第一个节点 N_1 向其下一节点 N_2 发送 Q_1[1-16]和 Q_2[1-16]。N_2 将两个接收的部分编码块与其本地部分编码块 Q_1[17-32]和 Q_2[17-32]组合，以形成两个新的部分编码的块 Q_1[1-32]和 Q_2[1-32]，它们被发送到下一节点 N_3。在 N_3 和 N_4 中执行类似的操作。最后，N_4 生成最终的全局校验块 Q_1[1-64]和 Q_2[1-64]。注意，部分编码的块被并行编码，并经由快速机架内部链路从 N_1 转发到 N_4，以便有效地计算宽条带的全局校验块。ECWide 需要在组合局部化下生成局部校验块，然而，由于 r 通常比 k 小得多，因此可以更有效地从单个节点中每个局部组的 r 个数据块对局部校验块进行编码。

在 Amazon EC2 上的实验表明，与其他基于局部化的前沿编码方法相比，本文方法可以将单块恢复时间减少 90.5%，冗余度仅为 1.063×。实验还验证了编码方法的有效性。在本文中，作者提出了组合局部性的宽条带方法，这是第一个通过结合校验局部化和拓扑局部化来系统地解决宽条带恢复问题的机制。本文进一步利用有效的编码方法和更新方案增强了组合局部性。通过具体实现和实验验证了方法有效。

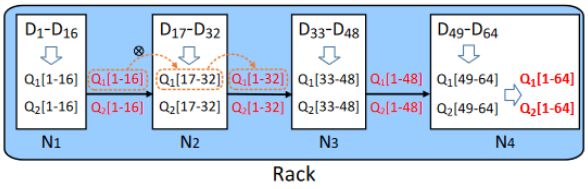

图 7 ECWide 多节点编码方案

3.3 通过网络编码实现最佳存储扩展

存储扩展将导致存储系统将现有存储数据重新定位到不同的节点，并根据新的数据布局重新计算纠删码数据，这将导致大量的数据传输。与纠删码修复问题不同，存储扩展问题会改变编码参数和存储节点的数量。为了最小化扩展带宽，文章提出了利用存储节点的可用计算资源实现扩展的分布式存储系统 NCScale[3]。

NCScale 考虑两种类型的扩展，一种是向外扩展（scale-out），另一种是向内扩展（scale-in），前者是向存储系统中添加新节点，而后者则是从存储系统中删除现有节点。

Scale-out

NCScale 在扩展之前和之后的所有块仍然由 RS 码进行编码。在扩展之前，每个现有节点独立地计算校验增量块，然后将其与现有校验块合并，在扩展之后为新的条带形成新的校验块。最后，NCScale 将一些数据块和新的校验块发送给新节点，同时确保新的条带在节点之间具有统一的数据分布和校验块。

当 n-k = 1 时，NCScale 可以实现最小的扩展带宽(即，每条带有一个校验块)。在这种情况下，每个条带的新校验块可以从同一节点生成的增量块在本地计算。换句话说，通过 NCScale 在网络上发送的块只是那些将存储在新节点中的块。图 8(a)显示了 NCScale 中(3,2,1)-scaling 的一个例子。

另一方面，对于 n−k>1（即，每个条带具有多个奇偶校验块），NCScale 无法达到最佳点。每个现有节点现在不仅生成用于本地计算新的校验块的奇偶校验增量块，而且还发送用于计算不同节点中相同条带的新校验块。然而，发送到其他节点的校验增量块的数量仍然有限，因为我们只使用一个校

验增量来更新每个新的校验块图 8(b)显示了 NCScale 中(4,2,2)-scaling 的示例。

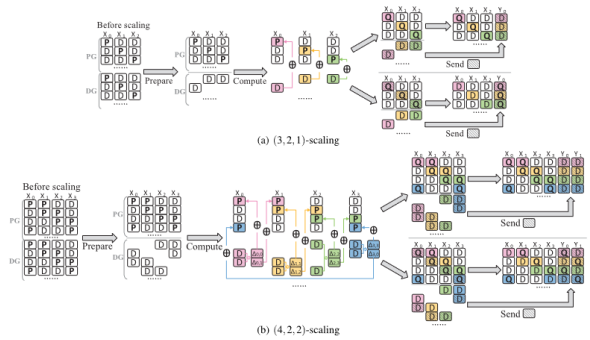

图 8　NCScale 的 Scale-out 方案

具体而言，NCScale 在准备扩展过程中要处理的数据集和校验块。它首先识别组 PG 和 DG。然后，它将 DG 中的数据块分成不同的集合 D_w，其中 $0 \leqslant w \leqslant nk$（n+s）−1，通过以循环方式收集并将 s 个数据块从 X_0 到 X_{n-1} 添加到 D_w 中。具体而言，DG 总共具有 ns（n+s）个条带，因此具有 nsk（n+s）个数据块。我们将 DG 中每个现有节点 X_i（$0 \leqslant i \leqslant n-1$）的数据块划分为 k（n+s）组 s 数据块集，并将每个现有节点 $X_{w \bmod n}$ 的第 w 组 s 数据块集添加到 D_w 中。

准备完成后，NCScale 计算新条带的新校验块，将块发送到新节点，并删除现有节点中的过时块。NCScale 在 PG 中的所有 nk（n+s）条带上运行。为了计算新的校验块，每个现有节点 X_i（$0 \leqslant i \leqslant n-1$）在 $i=w \bmod n$ 的第 w 条带上操作。回想一下，校验块存储在 X_i、⋯、$X_{(i+n-k-1) \bmod n}$ 中。对于 $0 \leqslant j \leqslant n-k-1$ 和 $j' = (j+w) \bmod n$，X_i 计算校验增量块 $\Delta_{i,j'}$ 并将其发送到 $X_{j'}$，这将 $\Delta_{i,j'}$ 添加到 PG 中第 w 条带的第 j 个校验块。当 j=0 时，X_i 在本地更新校验块。

在计算新的奇偶校验块之后，NCScale 将块发送到新节点。如果 $w \leqslant nk$（$n-s$（$n-k-1$））−1，X_i 将 D_w 中的所有 s 个数据块发送到 s 个新节点；否则，X_i 会将本地更新的校验块和 D_w 中的任何 s−1 数据块发送到 s 个新节点（我们假设校验块在 s 个节点上跨不同条带循环，以均匀放置校验块）。例如，图 8 分两种情况显示了扩展的最后一步。最后，X_i 删除所有过时的块，包括发送到新节点的块和 DG 中的校验块。通过这样做，可以保证扩展后数据和校验块的均匀分布。

Scale-in

当 $n-k=1$ 时，每个条带都有一个校验块，因此，通过将条带中移除的数据块传输到校验块所在的节点，同时保留均匀数据和校验分布，很容易在本地更新条带的现有奇偶校验块。图 9 描述了(4,3,-1)-scaling 的步骤。特别是，NCScale 将扩展过程中涉及的条带分为两种情况：

- 移除节点中只有数据块的条带（图 9(a)）：NCScale 首先从移除节点中的数据块为新条带生成新的奇偶校验块（step 1）。然后，它将这些数据块及其生成的校验块作为新条带发送到幸存节点，同时将这些数据块传输到在同一现有条带内具有校验块的节点（step 2）。它使用这些数据块来本地更新现有条带中的校验块（step 3）。
- 移除节点中具有校验块的条带（图 9(b)）：NCScale 首先将现有校验块发送到幸存节点，其中现有条带中的数据块在本地更新这些校验块（step 1）。然后，它通过收集这些用于更新现有校验块的数据块，为新条带生成新的校验块（step 2）。

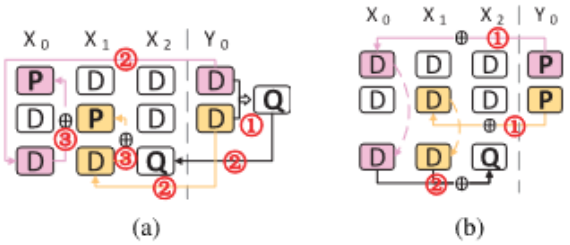

图 9　NCScale 的 Scale-in 方案

当 $n-k=1$ 时，NCScale 可以显著降低扩展带宽。对于第一种情况中的条带，将每个新条带的新奇偶校验块作为编码块发送，该编码块由来自相同移除节点的数据块生成，并且可以本地更新每个现有条带的现有校验块。也就是说 NCScale 通过网络发送的块仅是将添加到幸存节点的块。对于第二种情况中的条带，每个现有条带的现有校验块也可以本地更新，但是每个新条带的新校验块是从驻留在其他幸存节点中的数据块生成的。然而，为生成新的奇偶校验块而传送的数据块的数量仍然是有限的，因为在第二种情况下，校验块仅占据条带的所有块的 1/n。

当 $n-k>1$ 时，每个条带都有一个以上的校验块，因此在保留 P3 的同时，很难在本地更新每个现有的校验。NCScale 产生了额外的扩展带宽来维持均匀数据和校验分布，使得一些现有的校验块可以在本地更新，而剩余的现有校验块需要通过从其他幸存节点生成的校验增量块来更新。

3.4 更改纠删码粒度降低尾延迟

远端内存系统允许应用程序透明地访问本地内存以及属于远程机器的内存，可有效利用数据中心的闲置内存。然而远端内存系统同样需要容错能力来保证数据的可靠性。目前已有一些远端内存系统使用纠删码来进行容错。 Hydra[4]使用纠删码，其编码方案将一个 span 划分为多个块并分别存放在多个内存节点中， 这意味着当计算节点重建一个 span 时，需要通过网络从多个内存节点处获得数据块和校验块，多次的通信可能会造成网络的拥塞，且会存在尾延迟问题。

文章同样提出一个利用纠删码容错的远端内存系统 Carbink[5]，其系统设计如图 10 所示。计算节点执行希望使用远端内存的单进程（但可能是多线程）应用。内存节点提供给计算节点远端内存，用来存储本地 RAM 无法容纳的应用数据。一个逻辑上集中的内存管理器跟踪计算节点和内存节点的有效性。该管理器还协调远端内存区域对计算节点的分配。当一个内存节点想把一个本地内存区域提供给计算节点时，该内存节点会向内存管理器注册该区域。之后，当一个计算节点需要远端内存时，计算节点会向内存管理器发送分配请求，然后管理器会分配一个注册的、未分配的区域。在收到来自一个计算节点的取消分配消息后，内存管理器将相关区域标记为可供其他计算节点使用。一个内存节点可以要求内存管理器取消对先前注册的（但目前尚未分配的）区域的注册，将该区域从全局的远端内存池中撤出。

Carbink 的运行时使用 span 和 spanset 管理对象。 如图 11 所示， 一个 span 是连续内存页的集合；一个计算节点分配的单一区域包含一个或多个 span。 Carbink 将每个对象分配四舍五入到相关 span 的 bin 大小，并将每个 span 与计算节点和内存节点使用的页面大小一致。 Carbink 以 span 的粒度将远端内存换入本地内存；然而，当本地内存压力很大时， Carbink 以 spanset（即相同大小的 span 的集合）的粒度将本地内存换出到远端内存。

Carbink 对一组同等大小的 span 中进行纠删编码，并将这样的组为 spanset。在这种方法中，spanset 中的每个 span 都被视为一个片段，并在该集合的所有 span 中计算校验数据。为了重建一个 span，一个计算节点只需要联系存储该 span 的单个内存节点。Carbink 使用这种方法来最小化尾延迟。

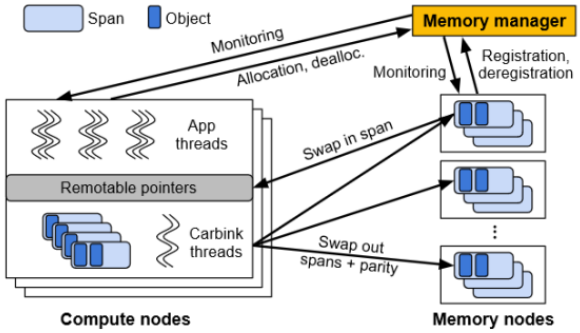

图 10 Carbink 系统结构

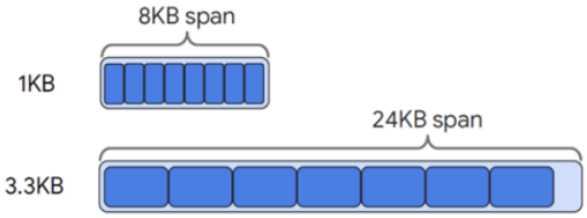

图 11 span 粒度组织

但是， 在 spanset 粒度上进行纠删码却在 span 粒度上进行交换，会带来涉及校验码更新的复杂情况。原因是，换入一个 span s 会在后备 spanset 中留下一个无效的、span 大小的洞；这个洞必须被标记为无效，因为当 s 后来被换出时， s 将被换出作为一个新 span 的一部分。换入 s 所产生的洞会导致后备 spanset 的碎片化。确定如何垃圾回收这个洞并更新相关的校验信息是不容易的。理想情况下，垃圾回收和奇偶校验更新的方案不会在换入或换出的关键路径上产生开销。一个理想的方案还允许校验重计算发生在计算节点或内存节点上，以使这两类节点上的空闲 CPU 资源得到机会性的利用。为此 Carbink 提出两种纠删码方案， EC-Batch Local 和 EC-Batch Remote。这两种方案的纠删码都是以 spanset 为粒度，使用 RMA 进行换入和换出。换入发生在 span 的粒度上，而换出发生在 spanset 的粒度上；因此，两种 EC-Batch 方法在将 span 换入计算节点的本地 RAM 时，都会在远端内存中取消分配 span 的支持区域。其结果是，换入会在远程内存节点上产生死区。两种 EC-Batch 方案都使用异步垃圾回收来回收死区并重新计算校验数据。EC-Batch Local 总是在计算节点上重新计算校验值，而 EC-Batch Remote 可以在计算节点或内存节点上重新计算校验值。当 EC-Batch Remote 将校验计算卸载到远程节点时，它采用了一个并行提交方案，避免了传统的两阶段提交的延迟。

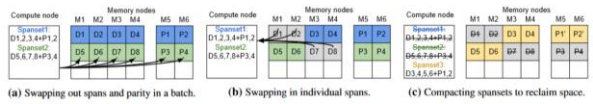

图 12　EC-Batch 方案

EC-Batch：交换

如图 12(a)所示， 在换出时，一个计算节点将一个 batch（即一个 spanset 和它的校验片段）写入多个内存节点。在换入时， Carbink 运行时检查应用程序的指针并提取 span ID。运行时查询远端页面映射以发现哪个远程节点持有该 span。最后，运行时启动适当的 RMA 操作来交换 span。

然而，以 span 为单位的交换会产生远程碎片。在图 12(b)中，运行实例中的计算节点将四个 span（D1、 D2、 D7 和 D8）拉入本地内存。任何特定的 span 都只存在于本地内存或远端内存中；因此，这四个 span 的互换会在相关的远程内存节点上产生死区。如果 Carbink 想用一个新的 span D9 来填补（比如） D1 的死区， Carbink 必须更新奇偶校验片段 P1 和 P2。对于 Reed-Solomon 码，这些奇偶校验片段将取决于 D1 和 D9。

通常来说，有两种简单的方法来更新 P1 和 P2。第一种方法是由计算节点来计算校验值， 计算节点可以将 D1 读入本地内存，生成校验信息，然后向 P1 和 P2 发出写操作。 第二种方法则是由内存节点来计算， 计算节点可以将 D9 发送给内存节点 M1，并要求 M1 计算新的校验数据并更新 P1 和 P2。这两种方法都需要大量的网络带宽来填补远端内存的漏洞。为了回收一个空闲的 span，第一种方法需要三个 span 大小的传输，计算节点必须读取 D1，然后写入 P1 和 P2。第二种方法需要两次 span 大小的传输来更新 P1 和 P2。为了减少这些网络开销，Carbink 进行了远程压缩。

EC-Batch：远程压缩

Carbink 采用远程压缩的方式对远端内存进行碎片整理，使用的网络资源比上面两个朴素的方法少。在一个计算节点上， Carbink 执行了几个压缩线程。这些线程寻找 "匹配 "的 spanset 对；在每一对中，一个 spanset 中包含死区的位置在另一个 spanset 中被占用，反之亦然。例如，图 12(b)中的两个 spanset 是一个匹配的对。一旦压缩线程找到匹配的一对，它们就会创建一个新的 spanset，其数据由匹配的一对中的活的 span 组成（例如，图 12(b)中的< D3, D4, D5, D6 >）。压缩线程使用我们在下一段讨论的技术重新计算并更新校验片段 P1′ 和 P2′ 。最后，压缩线程将匹配对中的死区去掉（例如，图 12(b)中的< D1, D2, D7, D9, P3, P4 >），从而形成图 12(c)中所示的情况。

在计算节点上， Carbink 的运行时可以监测远程内存节点的 CPU 负载和网络利用率。运行时可以默认通过 EC-Batch Local 进行远程压缩，但如果内存节点上出现空闲资源，则会适时切换到 EC-Batch Remote。在切换到不同的压缩模式时，Carbink 允许所有还在进行的压缩完成，然后再发布使用新压缩模式的新压缩。

朴素碎片整理方案需要两个或三个 span 大小的网络传输来恢复一个死 span。在图 12 中，EC-Batch Local 使用四个 span 的网络传输来恢复四个死 span，而 EC-BatchRemote 需要四个 span 的网络传输来恢复四个死 span，效率大大提高。

3.5　改善降级读取延迟的自适应纠删码方案

纠删码相较于复制方案减少了冗余，然而，当数据变得不可用时，读取将会降级，纠删码存储系统必须先修复它们才能进行读取，纠删码在修复数据期间额外读取和传输的数据量将会导致两个问题：降级读的高读取延迟和数据的长重建时间。

不同的存储系统使用不同的纠删码来实现容错。例如，Google ColossusFS 使用（6，3）-RS 代码，而 Facebook HDFS 使用（10，4）-RS 码，Azure 使用（12，2，2）-LRC。如表 2 所示，由于编码族和参数的差异会在各个方面（包括可靠性、性能甚至结构）极大地影响存储系统，因此可以做出多种选择。LRC 基于 RS 码，并且它们添加了一些本地校验，以在恢复数据的同时降低传输成本，但 LRC 不是 MDS 编码。HH 代码可以保持 MDS 属性并减少恢复延迟，但编码时间可能会增加。

Properties	RS codes	LRCs	HH codes	Array codes	Regenerating codes
Storage cost	Low (MDS)	High	Low (MDS)	Low	Low
Constraints in choosing parameters	No	No	No	Yes	Yes
Recovery latency	High	Low	Not high	High	Lowest
Other disadvantages			Long encoding time	Limited fault tolerance ability	Hard to implement

表 2　编码族之间的对比

在大规模分布式存储系统中，数据访问中存在明显的偏差很常见。热数据占据了大多数数据访问，而冷数据占据了大部分存储空间。当考虑降级读取时，大多数读取将应用于热数据，因为热数据占用了大多数数据访问。换句话说，如果可以减少热数据的降级读取延迟，整个系统的降级读取等待时间可以降低到类似的水平。同时，由于热数据在整个系统中占用的存储空间很小，如果我们适当提高热数据的存储开销，总体存储开销几乎不会增

加。

采用单一纠删码方案的存储系统无法在保持低存储成本的同时最大限度降低修复成本。如果使用 MDS 代码来存储冷数据，而使用另一个代码来存储热数据，以达到低恢复延迟，那么整个系统可以从低存储成本和低降级读取延迟中获益。从存储成本的角度来看，冷数据占据了整个系统的主导地位，因此整个系统可以被视为“近 MDS”。就降级的读取延迟而言，热数据主导了大多数访问，因此它们主导了大多数降级的读取。因此，整个系统的降级读取延迟将接近热数据的降级读取等待时间。

文章由此提出一种自适应纠删码方案[6]。该方案使用来自不同编码族的两种纠删码，一个是快速码，用于低修复成本存储热数据，另一个是紧凑码，可以保证在存储冷数据时存储成本最低。系统中应用数据分析器来分析数据冷热。当系统的实际存储成本高于预期存储开销时，最近访问次数最少的热数据将被设置为冷数据。如果在阈值时间内未访问热数据，则将其设置为冷数据。当系统的实际存储成本低于预期存储开销时，最近访问的冷数据将被设置为热数据。如果在短时间内重复访问冷数据，则将其视为热数据。

编码选择

假定降级读取成本为修复单个丢失数据块所需读取的块数。在（k，m）-HH 码中，降级的读取成本为 mk/（m−1），而在（k，m−1，m）-LRC 或（k，m−1，m−1）-LRC 中，降级读取成本为 k/（m−1）。图 13 显示了降级读取成本与存储开销之间的关系，如果将存储开销设置在曲线的粗体区域，整个系统可以从 HH 代码的 MDS 特性和 LRC 的快速恢复特性中获益。根据此图，文章将存储开销设置为 1.4×，从而得到 LRC 码的数据与 HH 码之间的比率为 4:11（方案 I）或 2:3（方案 II），此时方案 I 的降级读取成本为 4.44，方案 II 的降级读取开销为 4.3。而相同冗余度的单一编码的降级读开销如表 3 所示，均高于文中所给方案。

编码切换算法

为了缓解编码切换过程引起的带宽消耗问题，文章选择一对合适的紧凑码和快速码来减少重新编码。（k，m−1，m）-LRC 或（k，m−1，m−1）-LRC 被选择作为快速码，（k，m）-HH 码被选择作为紧凑码。LRC 可以在恢复数据时减少网络带宽和磁盘 I/O 的使用，这意味着它符合快速码的要求。HH 代码保持 MDS 属性，并减少恢复延迟（不如 LRC，但仍比传统 RS 代码有改进）。更重要的是，LRC 和 HH 码都基于 RS 码，这意味着它们在某些方面是相关的。通过利用这两个编码之间的相似性，文章在快速码和紧凑码之间提出了一种编码切换算法。

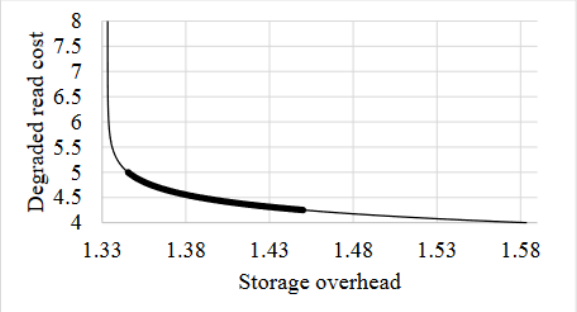

图 13 降级读取开销-冗余度曲线

Scheme	Degraded read cost	MTTF (years)
Scheme I	4.44	1.0×10^{14}
Scheme II	4.3	1.0×10^{14}
(10, 4)-RS code	10	6.7×10^{13}
(10, 4)-HH code	6.7	9.3×10^{13}
(12, 2, 3)-LRC	6	6.3×10^{13}
3-replication		3.5×10^{9}
4-replication		7.7×10^{13}

表 3 不同方案的降级读开销对比

当（k，m−1，m）-LRC 编码数据转换为（k，m）-HH 码形式时，LRC 中的两个条带（a 和 b）被视为 HH 码中条带的两个子条带（分别为 s_a 和 s_b），因此 LRC 中所有块都可以视为符号。可以保留所有数据符号，并且可以保留 a 中的所有全局校验符号作为 s_a 中的校验符号。在 b 中，第一个全局校验符号可以保留为 s_b 中的第一个校验符号，然而，s_b 中的其余校验符号应该是 b 中的一个全局符号及其在 a 中的对应数据符号组的异或和。为了计算和，不必读取 a 中的所有数据符号。相反，a 中的原始局部校验符号正好是应该计算的部分和。因此，要计算 s_b 中的校验符号，唯一需要做的就是读取 a 中的相应局部校验符号并将其与 b 中的原始全局校验符号进行异或运算。图 14 显示了当 k＝6 和 m＝3 时代码切换过程的示例。

当（k，m）-HH 编码数据转换为（k，m-1，m）LRC 形式时，HH 码中的条带的两个子条带（s_a 和 sb）被视为 LRC 中的两个条带（分别为 a 和 b）。类似地，所有数据符号可以作为数据块保留，s_a 中的所有校验符号可以作为 a 中的全局校验块保留。

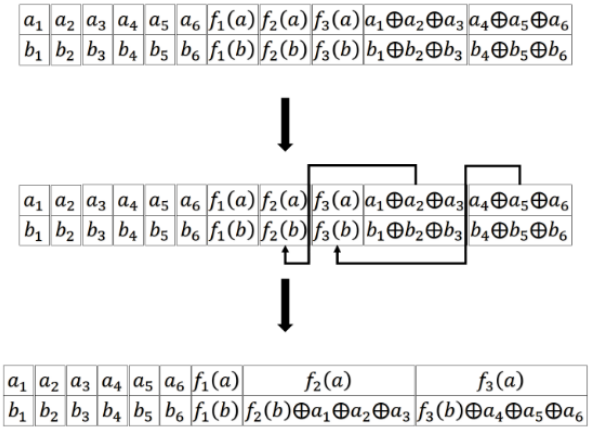

图 14　(6, 2, 3)-LRC 码到 (6, 3)-HH 码的转换

所有本地校验块只能由数据块计算，这部分计算很难避免。sb 中的第一个校验符号可以保留为 b 中的第一全局校验块。在此阶段，可以在没有任何数据块参与的情况下计算 b 中的其余全局块。要计算 b 中的全局符号，只需将 a 中的相应本地校验块（刚刚计算）与 b 中的原始校验符号进行异或。图 15 显示了当 k＝6 和 m＝3 时代码切换过程的示例。

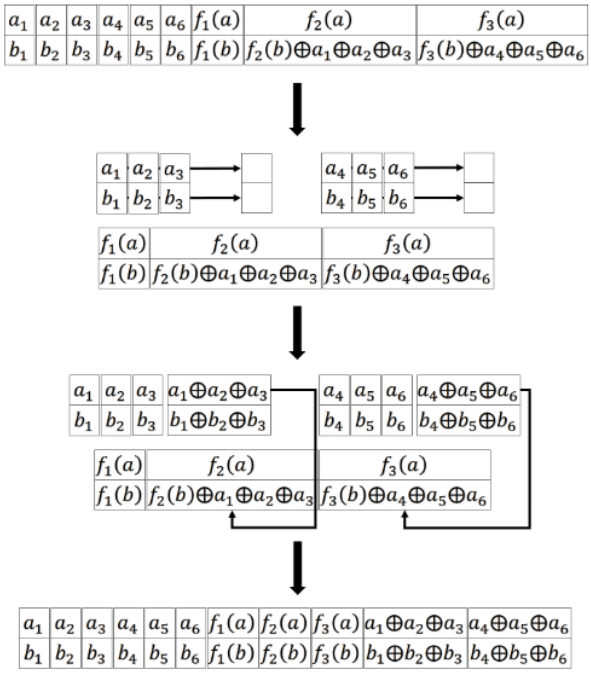

图 15　(6, 3)-HH 码到 (6, 2, 3)-LRC 码的转换

当热数据变冷时，系统需要将 LRC 编码数据转换为（k，m）-HH 码形式。在原始的重新编码算法中，在 HH 码的编码过程之前，所有数据符号都应收集在一个节点中，该节点应在重新编码后发送新的奇偶校验符号。因此，系统需要传输 2（k+m−1）个符号。而通过本文的编码切换算法，只需要传输（m−1）个符号。当 k＝12 且 m＝4 时，与重新编码算法相比，该算法只需要传输 1/10，且该算法可以分布式并发执行，这有助于减少延迟。

当冷数据变热时，系统需要将（k，m）HH 编码数据转换为（k，m-1，m）-LRC。在原始的重新编码算法中，在 LRC 编码过程之前，所有数据符号都应收集在一个节点中，该节点应在重新编码后发送新的校验符号。因此，系统需要传输 2（k+2m−2）个符号。通过文章的编码切换算法，仍然需要所有数据块，但传输的数据总量可以减少到（2k+m−1）个符号。这些操作也可以分布式并发执行，这意味着延迟可以减少到 1/（m−1）。当 k＝12 且 m＝4 时，编码切换算法减少重新编码算法的大约 2/3 的延迟，并且还具有大约 1/4 的带宽消耗。

4　总结

随着全球数据量的急速增长，数据中心规模也不断扩大，随之而来的是节点故障带来的可靠性问题。纠删码作为保障数据可靠性和可用性的低冗余容错技术备受青睐。然而，在现代数据中心中纠删码系统的应用仍然存在一些问题和挑战，吸引大量的研究人员进行优化。本文对现有的一些问题进行梳理和总结，分析评述了最近一些数据中心中纠删码系统优化方案，可以看到降低冗余和延迟仍然是数据中心纠删码优化的重要目标。

宽条带纠删码可以降低系统冗余，但生成宽条带会产生大量的带宽消耗，研究人员采用近似完美合并的算法对此进行优化，在低冗余开销的同时大幅降低带宽的消耗；宽条带纠删码的修复同样需要大量的数据传输，研究人员结合现有的两种局部化方法在减少修复过程的跨机架带宽消耗的同时尽量降低冗余度；数据中心由于规模扩大而引进新的节点，数据需要重新编码来适配布局，研究人员利用存储节点的可用计算资源实现扩展，从而有效降低存储扩展引入的数据传输量；采用纠删码容错的远端内存系统中修复数据需要从多个节点获取数据，由此产生的尾延迟对于内存系统来说不可接受，研究人员更改纠删码编码粒度来优化尾延迟；纠删码低冗余的代价就是修复时的高延迟，研究人员利用不同纠删码族之间的特性以及数据中心数据访问特点设计了自适应的纠删码方案，在几乎不牺牲冗余度的同时有效改善了降级读取延迟。

参 考 文 献

[1] Yao Q, Hu Y, Cheng L, et al. Stripemerge: Efficient wide-stripe generation for large-scale erasure-coded storage[C]//2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS). IEEE, 2021: 483-493.

[2] 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.

[3] Hu Y, Zhang X, Lee P P C, et al. NCScale: Toward Optimal Storage Scaling via Network Coding[J]. IEEE/ACM Transactions on Networking, 2021, 30(1): 271-284.

[4] Lee Y, Maruf H A, Chowdhury M, et al. Mitigating the performance-efficiency tradeoff in resilient memory disaggregation[J]. arXiv preprint arXiv:1910.09727, 2019.

[5] Zhou Y, Wassel H M G, Liu S, et al. Carbink:{Fault-Tolerant} Far Memory[C]//16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). 2022: 55-71.

[6] Wang Z, Wang H, Shao A, et al. An adaptive erasure-coded storage scheme with an efficient code-switching algorithm[C]//49th International Conference on Parallel Processing-ICPP. 2020: 1-11.