纠删编码原理及性能优化

田浩然[1)]

[1)](华中科技大学 计算机科学与技术学院，湖北省武汉市 435400)

摘 要 纠删编码是一种广泛应用于分布式存储系统的编码机制。纠删编码种类很多，其中最常用的一种是 Reed-Solomon 编码。纠删编码最主要的优势在于，相比于直接备份存储，其冗余度大大降低。同时，纠删编码又拥有较高的容错能力，能够保证数据的可靠性。而纠删编码的主要问题在于编码、解码的时间开销过大，同时会引入大量的跨机架传输流量。当一个数据块发生故障，需要读取多个其他数据块，并通过解码计算进行修复。如何提升纠删码的性能，减小跨机架传输带宽对于需要存储海量数据的数据中心来说至关重要，因此研究人员提出了多种方案，来改善纠删编码的性能。文中主要介绍了纠删码的基本原理及其优势和不足，并根据最新的研究成果，针对纠删做出的性能改进进行详尽阐述。

关键词 纠删码；数据中心；分布式存储

Principles and performance optimization of erasure code

Haoran Tian[1)]

[1)](School of Computer Science and Technology,Huazhong University Of Science and Technology,Wuhan,Hubei)

Abstract Corrective coding is a widely used coding mechanism for distributed storage systems. There are many types of corrective coding, and one of the most commonly used is Reed-Solomon coding. The main advantage of corrective coding is that its redundancy is greatly reduced compared to direct backup storage. At the same time, the corrective coding has a high fault tolerance, which can ensure the reliability of the data. The main problem of corrective coding is that the time overhead of encoding and decoding is too large, and it introduces a large amount of cross-rack transfer traffic. When a data block fails, multiple other data blocks need to be read and repaired by decoding computation. How to improve the performance of corrective coding and reduce the cross-rack transfer bandwidth is crucial for data centers that need to store huge amounts of data, so researchers have proposed various schemes to improve the performance of corrective coding. In this paper, we introduce the basic principles of corrective coding, its advantages and shortcomings, and elaborate on the performance.

Key words Erasure Code; Data Center; distributed storage

1 引言

在大数据时代，随着数据存储量的需求急速增长，大规模数据中心和分布式的存储系统逐渐普及。在一个大型的分布式存储系统中存在海量的存储节点，因而会频繁发生软硬件故障。通常情况下，数据中心存储着大量的重要数据，对于数据可靠性有较高要求。因此，在大型存储系统中需要存储冗余数据来

表 1 当今最先进的基于纠删编码的存储系统

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

保证数据的可靠性。简单的多副本备份存储稳定性良好，且修复数据效率较高，但其缺点在于需要存储大量的冗余数据，代价高昂。而随着数据量的爆炸时增长，多副本的方法造成的成本变得无法接受。基于此种情况，以 Reed-Solomon 编码为代表的纠删码应运而生，逐渐成为了分布式存储系统中主流的冗余策略。以数据节点总数除以存储非冗余数据节点数作为冗余度，传统大规模存储系统中采用的双副本、三副本方式具有 2 或 3 的高冗余度。

如表 1 所示，当今大多数互联网公司和数据存储公司可以通过纠删码方案将冗余度控制在 1.5 以下，这大大节约了存储成本。

2 纠删码基本原理

2.1 纠删码基本概念

在应用了纠删码的存储系统中，每个存储节点都包含大量的存储块，存储块主要由用于存储数据的数据块，和用来修复丢失数据的校验块组成。在系统读取数据时，会依据数量和编号来选择多个数据块来进行编码，将编码后的数据称为编码块。一组独立编码的编码块构成一个条带。条带大小的选取，通常与数据内容的划分，以及对于容错能力和冗余度的要求有关。其中，容错能力指的是，一个条带上最多可丢失块的数量，当出现故障的块数大于容错能力时，将无法修复丢失数据，本质上讲，纠删编码是对于数据存储过程中面对的可能的数据丢失和节约存储成本两个问题的一种折衷和妥协。

纠删码有多种类型，其中最常见的毫无疑问是 Reed-Solomon 编码，通常用 n 来表示一个条带中总的块数，即包括原始数据块和校验块，用 k 表示数据块数，用 r=n-k 表示校验块数，也即该纠删码系统的容错能力，表明此 k 个数据块可以容忍 r 个数据块同时出错。通常，会用 RS(n, k)来表示一个纠删码系统，其冗余度为 n/k。如 RS(6, 4)则表示一个使用 RS 编码，条带总大小为 6，数据块数为 4，校验块数为 2，冗余度为 6/4=1.5。

2.2 纠删码编码原理

纠删码的编码原理为通过数据块左乘一个生成矩阵得到校验块，如公式(1-1)所示

通常情况下，用 d 表示数据块，p 表示校验块。在公式(1-1)中，下标 1～k 表示共有 k 个数据块，1～r 表示 r 个校验块。第一个矩阵是校验块的生成矩阵 A，矩阵前 k 行单位矩阵用于复制数据块，后 r 行用于生成校验块。公式即表示用 k 个数据块左乘生成矩阵，得到 k 个数据块和 r 个校验块。

$$\begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \\ \alpha_{1,1}\alpha_{1,2}\cdots\alpha_{1,k} \\ \alpha_{2,1}\alpha_{2,2}\cdots\alpha_{2,k} \\ \vdots \quad \vdots \quad \ddots \quad \vdots \\ \alpha_{r,1}\alpha_{r,2}\cdots\alpha_{r,k} \end{bmatrix} \times \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_k \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_k \\ p_1 \\ p_2 \\ \vdots \\ p_r \end{bmatrix} \quad (1-1)$$

则对于校验块p_i，其计算公式为(1-2)所示

$$p_i = \alpha_{i,1}d_1 + \alpha_{i,2}d_2 + \cdots + \alpha_{i,k}d_k, \quad i = 1, 2, \cdots, r \quad (1-2)$$

由此可知，在生成校验块p_i时，需要读取 k 个数据块，并进行 k 次矩阵乘和(k-1)次矩阵加。

2.3 纠删码解码原理

纠删码解码主要用于修复丢失的数据块，通过读取其他未发生故障的数据块和在编码过程中得到的校验块与解码矩阵相乘，便可恢复原始数据。

$$\begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \\ \alpha_{2,1}\alpha_{2,2}\alpha_{2,3}\alpha_{2,4}\cdots\alpha_{2,k} \end{bmatrix} \times \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ \vdots \\ d_k \end{bmatrix} = \begin{bmatrix} d_1 \\ d_3 \\ d_4 \\ \vdots \\ d_k \\ p_1 \end{bmatrix} \quad (1-3)$$

$$\begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_k \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & 0 \\ \beta_{2,1}\beta_{2,2}\beta_{2,3}\cdots\beta_{2,k-1}\beta_{2,k} \\ 0 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 0 \end{bmatrix} \times \begin{bmatrix} d_1 \\ d_3 \\ d_4 \\ \vdots \\ d_k \\ p_1 \end{bmatrix} \quad (1-4)$$

$$d_2 = \beta_{2,1}d_1 + \beta_{2,2}d_2 + \cdots + \beta_{2,k}d_k \quad (1-5)$$

现以数据块d_2发生故障为例，原始数据的数据块和修复数据时需要使用的数据块、校验块的关系如公式(1-3)所示。计算数据块左乘矩阵的逆矩阵后，得到公(1-4)，即为解码原理。其中，需要恢复的数据块的计算方法如公式(1-5)所示。

3 组合式定位优化

3.1 使用大条带纠删码降低冗余度

如表 1 所示，当今最先进的基于纠删编码的存储系统具有 1.18～1.5 的冗余度，存储了 18%～50%冗余数据。虽然其相比于双副本和三副本模式，已经将冗余度大大降低，但仍然具有较大的优化空间。同时，降低纠删编码存储系统的冗余度能够进一步降低数据中心的存储成本。为此，Hu[1]等人首

次提出了大条带纠删编码，并提出配套方案，解决了大条带纠删编码带来的问题。

传统观点认为，纠删码的条带大小 n 通常不应超过 20，现存最先进的大型数据中心也普遍采用中等大小的条带，其条带大小一般在 9～20 左右。相比之下，大条带虽然极大的节约了存储成本，但也进一步加重了纠删码修复开销大的缺点。由于每当一个数据块出现故障时，均需要读取 k 个存储块。随着条带的增大，修复时间和跨机架传输带宽增大，这将导致网路拥塞、吞吐量下降，从而降低了存储的可靠性。如图 1 所示，当块大小为 64Mib 且 n−k = 4 时，不同 Intel CPU 系列上的编码吞吐量随 k 的变化，但 k 从 32 增加到 64 时，吞吐量急剧下降。

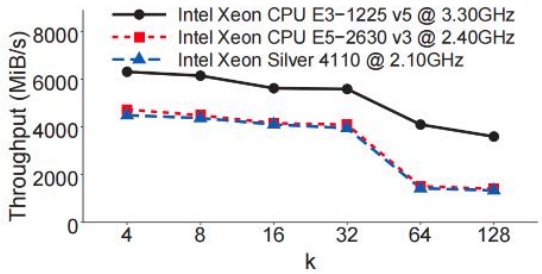


图 1　不同 Intel 芯片吞吐量随 k 的变化

而 Hu 等人提出了一种组合定位方式（将在§3.2 中介绍），该方式可以大大降低跨机架传输带宽，将单块的修复时间减小了 90.5%。

3.2　使用组合定位方式降低跨机架传输流量

首先说明，本节中使用 f 表示一个条带上的容错能力，z 表示用于存储同一条带的机架数，LRC 表示基于 Parity Locality 的 Azure-LRC，TL 表示 Topology Locality，CL 表示 Combined Locality。

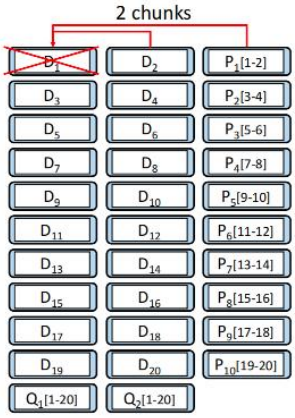


图 2-a Parity Locality: (32,20,2) Azure-LRC

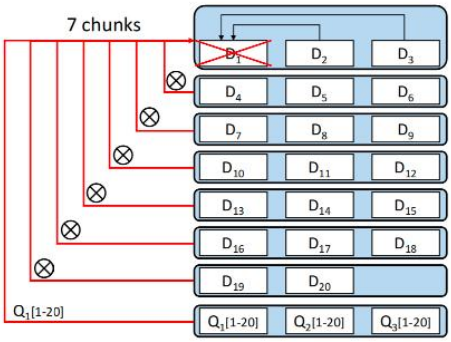


图 2-b Topology Locality: (23,20,8) TL

如图 2-a 所示，Azure-LRC 方式中，共有 20 个数据块和 12 个校验块。每 2 个数据块会配合一个本地校验块，最后还有两个全局校验块 Q1 和 Q2。

全部的 32 的存储块分别放置在 32 个节点中，其在单块修复时，需要读取 2 个存储块的信息，如 D3 发生故障，则需要读取 D4 和P_2(3-4)。相比于一般的 Topology Locality（如图 2-b 所示），RS(23,20) 同样具有 20 个数据块，且可容忍 3 块故障，但其修复时需要读取 20 个数据块的信息，因此修复带宽比 Parity Locality 高很多。但另一方面，Azure-LRC 的冗余度高达 1.6，相比之下，Topology Locality 的冗余度只有 1.15。

综上所述，Parity Locality 和 Topology Locality 各具优势，但都不能同时解决高冗余度和高修复带宽的问题，且应用于大条带时，两者的缺点都会更加明显。

如图 3 所示，Hu 等人设计了一种(26,20,5,9)的 Combined Locality。即一个条带包含 26 个存储块，其中包含 20 个数据块，和 6 个校验块，假设每个机架包含 3 个存储块，则每两个机架作为一组，配有 1 个校验块，该校验块负责这两个机架上的 5 个

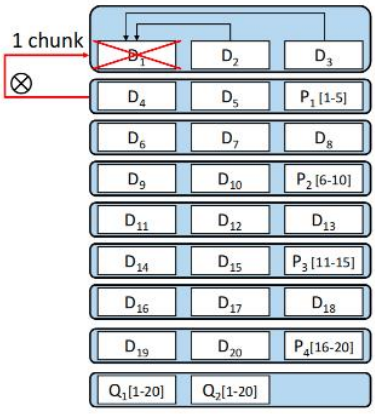


图 3 Combined Locality: (26,20,5,9) CL

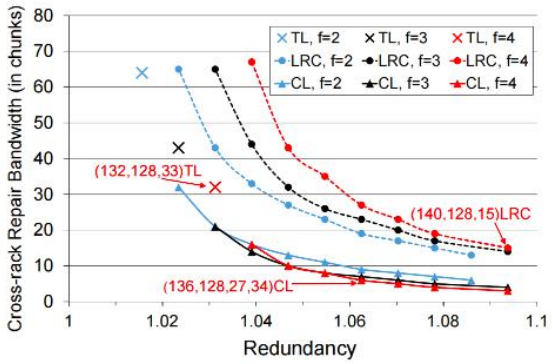


图 4 三种方式的对比

数据块。则该条带的 20 个数据块分别分布于 4 个小组，每组两个机架。该方式权衡了冗余度和跨机架传输带宽，是 Parity Locality 和 Topology Locality 的一种结合。

如图 4 所示，在 k=128 和 f=2,3,4 的情况下，Azure-LRC（LRC）、拓扑定位（TL）和组合定位（CL）的冗余度和跨机架修复带宽之间的权衡，其允许的最大冗余度为 1.1。通过结合奇偶性定位和拓扑性定位，在冗余度和跨机架修复带宽之间的权衡方面，组合定位优于 Azure-LRC 和拓扑定位。

表 2 三种方式的冗余度和跨机架修复带宽对比

	Redundancy	Cross-rack repair bandwidth
(n,k,r) Azure-LRC	$\frac{k+\lceil k/r\rceil+f-1}{k}$	r
(n,k,z) TL	$\frac{k+f}{k}$	$\lceil (k+f)/f\rceil-1$
(n,k,r,z) CL	$\frac{k+\lceil k/r\rceil+f-1}{k}$	$(r+1)/f-1$

以 f=4 为例。对于拓扑定位，(132,128,33)TL 的最小冗余度为 1.031，然而它的跨机架修复带宽达到 32 块，即使许多机架执行本地修复操作。(140,128,15)Azure-LRC 通过奇偶性定位将跨机架修复带宽大大降低到 r=15 块，但其冗余度（1.094）并不接近最小冗余度。而组合定位，(136,128,27,34)CL 不仅具有更接近最小冗余（即 1.063）的冗余，而且还进一步大大减少了跨架修复带宽，最多为(r +1)/ f -1 = 6 块，与 Azure-LRC 相比减少 60%。原因是组合定位的跨架修复带宽是μ（r/f）（表 2），所以它在有限冗余下的跨架修复带宽更低。

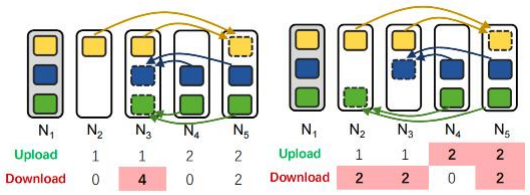
4 通过 RepairBoost 优化

4.1 RepairBoost 简介

全节点故障，即某一个存储节点的所有数据块同时出现故障。当前，大多数纠删编码的改进工作主要用于提高单块修复的性能。然而，全节点故障需要同时进行多个数据块的修复工作。Lin[2]等人提出 RepairBoost 调动框架，在 Amazon EC2 上的进行大量的实验表明，该框架可以将纠删码的修复速度提高 35-90.1%。

现有的修复方法难以满足全节点修复的要求，主要是受到了以下 4 点限制。

第一，它们未能利用全双工传输，大多数现有的研究忽略了修复中的全双工传输[3,4]，它使节点能够同时独立地发送和接收数据；因此，他们不能正确地平衡全节点修复中的上传和下载流量。如图 5 所示 CR 下的两个修复例子（其中 Ni 是第 i 个节点），第一个例子需要下载四个块（在 N3）来完成修复，而后者最多只上传和下载两个块进行修复。这个例子表明，在全双工传输中平衡上传和下载的修复流量有可能减少整个修复时间，这是由拥有最多上传或下载修复流量的节点决定的。



(a) Unbalanced repair solutions (b) Balanced repair solutions

图 5 限制一：未能利用全双工传输

具体来说，修复效率高的代码减少了单块的修复流量，而不涉及流量平衡。一些修复算法主要关注于加速单个块的修复，而对全节点修复中的上传和下载修复流量的平衡关注有限。虽然 ClusterSR 平衡了集群间的上传和下载修复流量，但它仍然没有平衡一般存储系统中的上传和下载带宽。

第二，他们未能利用每个时隙的带宽（如图 6 所示）。虽然现有的修复算法可以缓解单块修复中的下载瓶颈，但它们只是将多个分块的修复方案结合起来，以应对全节点修复。这可能会无意中再次导致链路拥堵，使全节点修复中的带宽得不到充分利用。

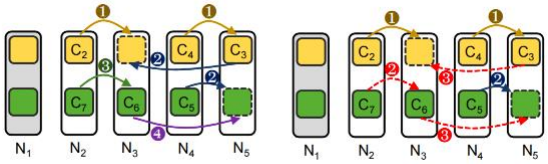


图 6 限制二：未能充分利用每个时间片的带宽

第三，修复过程不够灵活。许多修复算法按照同样的方法处理各个存储块，并使用相同的修复算法修复所有丢失的数据块。这种方式简单但不够灵活，因为它不能有效结合不同的修复算法，并使它们在现实世界的存储系统中针对不同的可靠性要求和倾斜的访问流行性进行合作。Lin[2]等人尝试将 ECPipe 和 CR 结合起来，比如我们用 ECPipe 来修复需要更高的可靠性保证的块，而用 CR 来恢复其余的块，这样就可以调节修复流量和前台流量。

第四，他们缺少全节点修复的一般框架。现有的分布式存储系统为单块修复部署了不同的具体修复算法。例如，许多商品存储系统由于其简单性，在单块修复中仍然依赖于 CR，而 PPR 和 CAR 可用于分级存储系统，以减少机架间的修复流量。最好能有一个通用框架，同时支持不同部署场景的不同类型的修复算法。

4.2 RDAG

4.2.1　RDAG 简介

为解决这些问题，Lin 等人通过一个 RDAG（即通过一个有向无环图 DAG 来正式确定一个单块修复方案）制定了一个单块修复。然后，它通过仔细地将 RDAG 的修复任务分配给节点来平衡上传和下载的修复流量。RepairBoost 还制定了最大流量问题，以安排数据传输，使未占用的带宽饱和。我们还表明，RepairBoost 可以扩展到解决多节点故障，并在异构环境中提升修复能力。之后 Lin 使用 C++实现了 RepairBoost 的原型，它可以是一个独立的中间件，部署在现有的存储系统上进行修复调度，并证明了 RepairBoost 的可移植性。最终，经评估，RepairBoost 在 Amazon EC2 上的性能，表明它可以支持各种擦除代码和修复算法，并将修复吞吐量提高 35.0-97.1%。

4.2.2　RDAG 的构建

首先需要构建 RDAG。在构建 RDAG 时，对于 RS(k,m)，一个丢失的块的 RDAG 可以在 k+1 个顶点上初始化，其中 k 个顶点{v1, v2,---, vk}代表存储要求修复的存活块的 k 个节点，而 vk+1 表示目标节点。另外，采用顶点之间的有向边来代表修复算法中规定的数据路由方向。

构建时，Lin 等人根据以下规则来构建边。给定两个顶点v_i和v_j（$1 \le i \ne j \le k+1$），用一条有向边$e_{i,j}$，来表示 vi 被指定发送一个幸存的块到v_j。因此，如果 ei, j 存在，我们说v_i是v_j的孩子，反之，v_j是v_i的父母。对于v_j其中 $j \ne k+1$），它必须从它的子代收集所有请求的幸存块，将它们与它使用解码系数存储的本地数据相加，并将结果发送到它的父代。因此，在这个 RDAG 中，一旦收到来自其子代的所有要求的块，v_j就可以发送一个块进行修复。由于全节点修复必须恢复多个块，一个节点可能在不同的 RDAG 中有多个父母和子女。

4.2.2　基于 RDAG 的节点修复

在 RDAG 的修复过程中，修复从叶顶点（即没有任何子节点的顶点）开始，在v_{k+1}结束：对于每条边 $e_{i,j}$ ($1 \le i \ne j \le k+1$)，如果 vi 已经将请求的块发送到v_j进行修复，将 $e_{i,j}$从 RDAG 中移除；此外，对于每个叶顶点v_i，如果没有连接到它的边（表明v_i已经将所有请求的块传送到其父母），我们也将v_i从 RDAG 中移除。因此，随着修复的进行，RDAG 中的顶点数量将逐渐减少，一旦顶点v_{k+1}最终成为叶顶点，丢失的块就被成功修复。

4.2.3　RDAG 的优点

首先，RDAG 是单块修复方案的一般形式化。也就是说，一旦确定了修复所需的存活块以及在它们之间执行的数据路由策略，就可以为任何给定的修复算法构建一个 RDAG。因此，RepairBoost 适用于各种擦除码（如 RS 码、LRC 和再生码）和单条纹修复算法（如 CR、PPR 和 ECPipe）。这种设计也解决了部署特定修复算法的矛盾（即通用性），并促进了它们的共存（即灵活性）。

其次，RDAG 描述了数据如何在网络上传输，也描述了参与修复的 k 个存活块之间的依赖关系。对于一个给定的 RDAG，其具有不同父本的叶子顶点有可能被平行传输，以占用可用的带宽而不会出现网络拥堵，如图 7 中的v_1和v_3节点。

第三，RDAG 还指出了每个顶点的修复任务，如图 7 中的v_4在修复中需要下载两个块并上传一个块，可通过修复任务的分配来探索流量平衡。

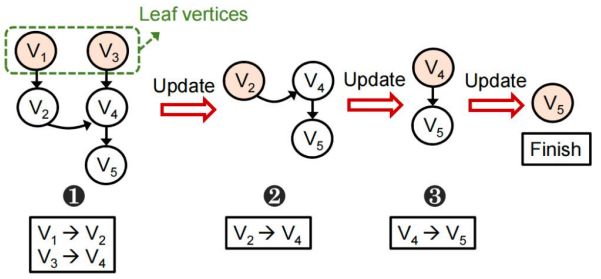


图 7 当 k=4 时，PPR 的 RDAG 实例

4.3 流量调度

4.3.1 平衡修复流量

在为故障节点中丢失的块构建 RDAG 后，RepairBoost 通过将顶点映射到节点来分配修复任务，这样在修复后必须保留擦除编码提供的容错度（即故障节点的可容忍数量），且整个系统的上传和下载修复流量应尽可能平衡。

4.3.2 传输调度

在建立了从 RDAG 到节点的映射，以平衡整体的上传和下载修复流量后，它不一定能实现修复时间的下限，因为带宽可能不会在修复期间的每个时隙都被利用。在修复过程中，带宽可能没有被利用。

为了进一步使带宽利用率饱和，RepairBoost 将传输调度制定为一个最大流量问题。具体来说，假设有 n 个节点参与全节点修复。根据丢失块的 RDAG,构建一个网络，该网络建立在 2n+2 个顶点上，有一个源 s，一个汇 t，n 个发送者顶点{S1,S2,---,Sn}代表可能发送数据进行修复的 n 个节点，另有 n 个接收者顶点{R1,R2,---,Rn}代表可能同时接收数据的 n 个节点。在港湾罗中建立连接如下：对于任何两个顶点 Si 和 Rj（1≤i 6 = j≤n），一旦 Si 可以根据 RDAGs 向 Rj 发送一个块，就建立 Si 和 Rj 之间的连接。Si 和 Rj 之间的每个连接都被分配了一个容量，这意味着每次可以从 Si 和 Rj 发送一个幸存的块。因此，需要在网络上找到一个最大的流量，其容量表示在这个时间段可以同时传输最多的块，以便使可用的上传和下载带宽达到饱和。

在建立最大流之后，可根据最大流的选定边来分配块。如果 Si 有很多块需要发送，则可先发送一个块，这样发送这个块可以使 Si 的一个父节点在下一个时隙成为 RDAG 的一个叶顶点。目标在于建立最大流量，同时在下一个调度中帮助增加网络的边的数量。这可以在下一次传输中潜在地增加最大流量的容量。

一旦与最大流量的边相关的块都被传输，就可以删除 RDAG 中的相应边，并根据剩余的 RDAG 更新网络。之后可以重复此调度过程，直到故障节点的所有丢失的数据块都被修复。

5 利用程序优化加速 XOR 纠删码

5.1 直线程序SLP

在改善纠删码性能时，常常会使用 XOR（异或）思想，但目前最先进的基于 XOR 的 EC 方法吞吐量较低，只能达到 4.9GB/S，相比之下，Intel 高性能的 EC 库则的编码吞吐量高达 6.7GB/S，优势明显。而 Yuya[5]等人通过 SLP 程序优化技术优化后的基于 XOR 的 EC 方法，可以将吞吐量提高到 8.92GB/S 。

SLP（straight-line programs）直线程序广泛应用于程序优化中，是一种具有单一二进制运算符的程序，没有分支、循环和其他复杂函数。后文中，将一个 SLP 通过元组(V, C, s, g, ⊕）来表示。其中 V 表示一组变量，C 表示一组常量，s 表示一串指令，是程序的主体，g 表示一串变量返回，是程序的变量返回序列，⊕表示其中的二进制运算符。

5.2 XOR性能优化

SLP 优化主要分为了三个部分，分别是压缩、融合、调度。压缩的重要思路是减少 XOR 运算的数量，使用带有 XOR 算子的 SLP，采用语法压缩理论中的 RePair 作为启发式算法，最终可以将原先的 8 个 XOR 运算减少到 5 个，并通过容纳 XOR-cancellativity 来扩展 RePair，引入了辅助过程 Rebuild(v), 从而增强了 Repair 并得到了 XorRepair。在融合方面，使用 Deforestation，即一种函数式的程序优化方法，来减少 SLP 内存访问。由于 SLP 简单清晰，使用 Deforestation 可以较为简单的转换和优化 SLP，从而减少了 SLP 访存。在调度优化方面，使用 red-blue pebble game 来进行缓存优化，可以减少 CCap 和 IO 代价，并通过将缓存和内存识别为常用寄存器来进行寄存器分配，寄存器在分配时，首先要重命名给定程序的变量，使用最小变量名来节约开销，之后判断寄存器是否足够存储变量，如果不够，则插入指令将寄存器的内容存放到存储器中，最后进行凝聚，将语法或语义相同的两边进行合并。总体而言，寄存器分配优化对于 SLP 的性能改进效果有限，因此便提出了卵石博弈来优化 SLP。在此种方法中，采用 DAG 来表示 SLP 的价值依赖性，此种方法不仅有助于将缓存优化问题

具体化，而且有助于正确参考编译器构建和程序分析的既定结果。

6 在网计算

6.1 在网计算的优势

由于数据密集型的应用对现代 HPC 集群上的高性能和可靠的存储系统的要求越来越高，导致了高昂的存储成本。

由于现存纠删编码存在较高的性能开销，Shi[6] 等人提出了一套 coherent in-network EC primitives（INEC）。

其中，主要的原因在于传统数据中心基于冯诺依曼架构，即以中央处理器为核心的架构，因此所有的数据都需要送到 CPU 进行处理。但随着数据中心的高速发展，摩尔定律逐渐失效，数据的告诉增长远远超过了 CPU 运算速度的增长，使得 CPU 的运算能力无法满足处理数据的需求。因此，如何分担 CPU 的计算工作成为了一大难题。当前，计算架构从以 CPU 为中心的 Onload 模式，向以数据为中心的 Offload 模式转变，因此，网卡逐渐成为了分担CPU计算任务的角色。Remote direct memory access(RDMA)，即远程直接访存技术是一种 host-offload, host-bypass 技术,凭借其优势，实现了低延迟、高带宽的内存之间的直接通信。Shi 等人提出的 INEC 已经在 RDMA 的商品网卡上得以实现，并且已经整合到了 5 个当下最先进的纠删编码方案中，分别是 conventional RS Code, Local Reconstruction Code (LRC), Partial Parallel Repair(PPR) Code, Repair Pipelining (ECPipe) Code, Tripartite Graph based EC (TriEC)，如图 9。

在上述的这一方案中，为了能够完全支持所有五个不同的最先进的 EC 协议，只需要三种类型的基本连贯的网络内 EC 原语（即 ec/xor-send、recv-ec/xor-send 和 recv-ec/xor）；为了将这些 EC 原语和协议完全卸载到网络上，提出了带有 RDMA WAIT 的高效设计，以提供快速的 EC 和网络能力，并在 Mellanox OFED 驱动中实现了 INEC 原语。据我们所知，这是第一个在 RNIC 上用 RDMA WAIT 提出一整套连贯的网络内 EC 原语的设计，以支持多种类型的 EC 协议；提出了一个α-β性能模型来分析 INEC 基元对五个最先进的 EC 方案的性能提升；通过基准测试和与键值存储系统的共同设计，所提出的 INEC 基元与五个最先进的 EC 协议自下而上地得到验证。经过微观基准测试表明，INEC 基元加速的 EC 方案的编码、解码带宽分别达到了 5.87 倍和 2.94 倍。

在对 YCSB 工作负载的评估表明，与 INEC 共同设计的键值存储在端到端吞吐量方面获得了高达 99.57%的改进，并在写和降级读延迟方面分别减少了 47.30%和 49.55%。在论文中,进行了原语的设计和分析，并进行了评估测试。在原语的设计方面，主要设计了 ec/xor-send、recv-ec/xor-send 和 recv-ec/xor 三种原语。其中对于 ec/xor-send 原语的设计思路为，由于当前最先进的 EC 方案中，几乎都会有一些节点负责执行 EC/XOR 计算，文章中将相应的 EC/XOR 计算抽象为 ec/xor-send，这使得上层应用可以将这一过程卸载到 SmartNIC，并从网络内连贯的 EC 中受益。 inec_ec_send 是 ec/xor-send 原语的功能。参数 calc 代表一个数据结构，它描述

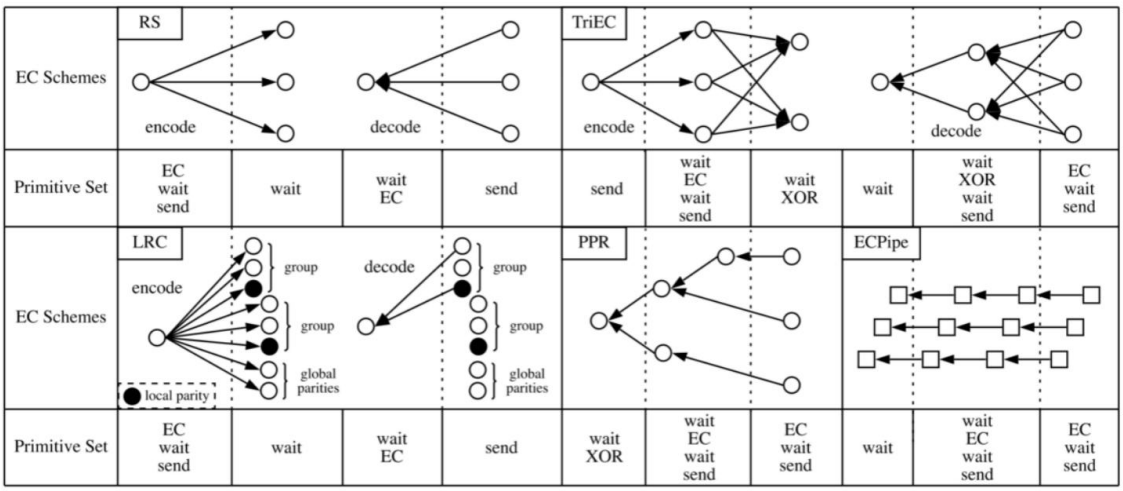


图 8 五种 EC 方案：RS、LRC、PPR、ECPipe、TriEC

了计算器类型（即 EC 或 XOR 计算器），用于 EC 计算的发生器矩阵等。内存布局由 mem 参数描述，而 stripe 表示用于发送 EC 条带的通信通道；对于原语 recv-ec/xor-send，社区中的许多 EC 方案将编码、解码计算分解成子问题，并将它们安排在参与计算的不同节点上，以获得更好的并行性、重叠性和对可用资源的充分使用，得到的 EC 方案较为复杂，但内容清晰明确。而对于 recv-ec/xor 原语，其功能是抽象出 RS 解码过程中，第一层接收远端对等实体发送的数据块，并对收到的数据块重建的过程。在文中之后的评估和分析过程汇总，主要从 operations、communication 和 gains 三个方面进行度量。之后又进行了 microbennchmark（微基准测试）和 YCSB（雅虎云服务基准测试），最终发现 INEC 在大多数情况下，都可以体现良好的性能。

7 利用 CRaft 降低存储和网络成本

7.1 Raft

Raft 是共识协议之一，它为构建实用系统提供了良好的基础。Raft 有三种服务器状态，如图 1 所示。当一个候选人收到大多数服务器的投票时，就会选出一个 leader。只有当候选人的日志至少与服务器的日志一样是最新的，服务器才能为候选人投票。每台服务器在每个任期内最多可以投票一次，因此 Raft 保证在一个任期内最多有一个 leader。

leader 接受来自客户的日志条目，并试图将其复制到其他服务器上，迫使其他人的日志与自己的日志一致。当 leader 发现在其任期内接受的一个日志条目已经被复制到大多数服务器上时，这个条目和之前的条目就可以安全地应用到其状态机中。leader 将提交并应用这些条目，然后通知 follower 应用它们。实用系统的共识协议通常具有安全性和有效性。安全性是指在所有非拜占庭条件下，它们不会返回错误的结果。有效性是指只要大多数的服务器是活的，并且能够相互通信和与客户通信，它们就是完全有效的。

7.2 CRaft性能优化

共识协议可以提供高度可靠和可用的分布式服务。在这些协议中，日志条目被完全复制到所有服务器。这种完全的条目复制会导致高的存储和网络成本，从而损害了性能。如果共识协议中的完整条目复制能够被擦除编码复制所取代，存储和网络

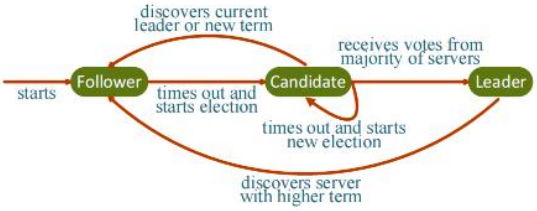


图 9:Raft 的三种状态

成本就可以大大降低。RS-Paxos 是第一个支持擦除编码数据的共识协议，但与常用的共识协议，如 Paxos 和 Raft 相比，它的可用性要差很多。Wang[7]等人指出了 RSPaxos 的可用性问题并试图解决它。在 Raft 的基础上，提出了一个新的协议，CRaft。它是一种支持擦除编码的改革版 Raft，并提供了两种不同的复制方法，它可以像 RS-Paxos 一样使用擦除编码来节省存储和网络成本，同时它也保持了与 Raft 一样的有效性。Wang 等人的实验评估证明，CRaft 可以节省 66%的存储空间，与原始 Raft 相比，写吞吐量提高了 250%，写延迟降低了 60.8%。

7.3 CRaft算法实现

接下来我们将讨论这两种复制方法的细节，然后我们尝试将它们整合到一个完整的协议--CRaft。为了解释 CRaft 的细节，我们首先定义一些参数。我们假设协议中有 N= (2F+1) 台服务器。由于 CRaft 应该具有与 Raft 相同的可用性，它的有效性级别应该是 F，这意味着 CRaft 在至少(F+1)台服务器是健康的时候仍然可以工作。我们为 CRaft 选择了一个(k,m)-RS 编码。k 和 m 应该满足 k+m=N，所以协议中的每个服务器可以对应于每个日志条目的一个编码-片段。如表 1 所示，CRaft 支持擦除编码，因此它可以节省存储和网络成本，同时它拥有一个 F 级的活泼度。

7.3.1 编码片段复制

当 CRaft 中的 leader 试图通过编码碎片复制方法复制一个条目时，它首先对该条目进行编码。在 Raft 中，每个日志条目都应包含其来自客户端的原始内容，以及其在协议中的术语和索引。当 CRaftleader 试图对一个条目进行编码时，内容可以通过协议选择的(k,m)-RS 代码被编码成 N = (k + m) 个片段。术语和索引不应该被编码，因为它们在协议中起着重要作用，编码过程如图 10 所示。

编码后，leader 将有 N 个条目的编码片段。然后，它将向每个 follower 发送相应的编码片段。在收到其相应的编码片段后，每个追随者都会回复给

是完整的条目复制到其余的 follower。

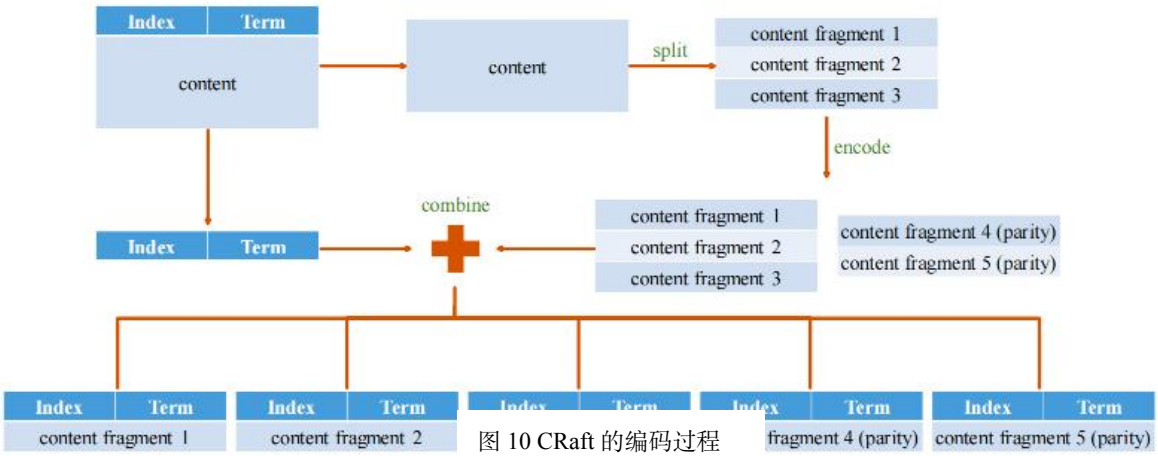


图 10 CRaft 的编码过程

leader。当 leader 确认至少有(F+k)个服务器存储了一个编码片段时，该条目及其之前的条目就可以被安全应用。领导将承诺并应用这些条目，然后通知 follower 应用它们。编码碎片复制的承诺条件比 Raft 的更严格。这个承诺条件也意味着，当没有(F+k)健康的服务器时，leader 不能使用编码-碎片复制来复制一个条目，然后提交它。

当一个 leader 出现故障时，将选出一个新的 leader。如果一个条目已经被承诺，Raft 的选举规则保证新的 leader 至少拥有该条目的一个片段。这意味着安全属性可以得到保证。由于至少有 至少有(F+k)个服务器存储了一个已承诺条目的片段。在任何(F+1)个服务器中都应该有至少 k 个编码片段。2 因此，新的 leader 可以收集 k 个编码片段，然后在有完整条目时恢复。所以新的 leader 可以收集 k 个编码片段，然后在至少有(F+1)个健康的服务器时恢复完整的条目。所以新的 leader 可以收集 k 个编码碎片，然后在至少有(F+1)个健康的服务器时恢复完整的条目，这意味着可以保证有效性。

7.3.2 全条目的复制

为了减少存储和网络成本，我们鼓励 leader 使用编码碎片复制。然而，当没有(F +k)健康的服务器时，编码-碎片复制将不起作用。当健康服务器的数量大于 F 且小于（F+k）时，leader 应该使用完整条目复制方法来复制条目。

在完整条目复制中，leader 必须在提交条目之前将完整条目复制到至少（F+1）台服务器上，就像 Raft 一样。由于提交规则与 Raft 相同，安全性和有效性都不是问题。然而，由于 CRaft 支持编码片段，leader 可以在提交条目后通过编码片段而不在实际的实现中，有许多策略可以通过完整条目复制来复制一个条目。定义一个整数参数 0≤p≤F，leader 可以首先将一个条目的完整副本发送给(F+p)follower，然后将编码片段发送给剩余的(F-p)follower。较小的 p 意味着较少的存储和网络成本，但也意味着较高的概率会有较长的承诺延迟（如果(F + p)答案中没有 F 及时返回，在承诺之前可能需要更多轮通信）。当 p=F 时，该策略变得与 Raft 的复制方法相同。图 11 显示了 p=0,1,F 时的不同策略。

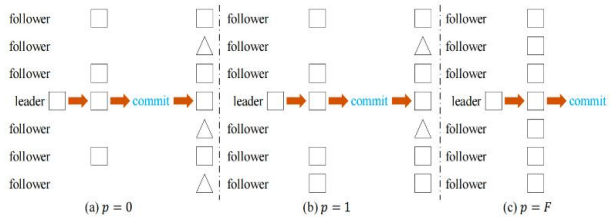


图 11 完整条目复制（图中正方形代表完整条目，三角形代表条目片段）

7.3.3 预测与新选 leader

在两种复制方法均可用时，使用编码碎片复制的性能优于完整条目复制。一个贪婪策略是，leader 总会试图通过编码碎片复制来复制条目。如果它发现没有(F+k)健康的服务器，它就转而通过完整条目复制来复制该条目。然而，如果 leader 已经知道健康服务器的数量少于(F + k)，那么通过编码片段的第一次复制尝试是没有意义的。

在 leader 试图复制一个条目时，应使用这种预测方法来决定复制方式。如果最近的心跳回答的数量不低于(F+k)，leader 应该首先使用编码碎片复制，如果编码碎片复制不起作用，它再尝试完整条目复制。否则，leader 直接使用完整入口复制，如

图 12：日志条目的复制流程

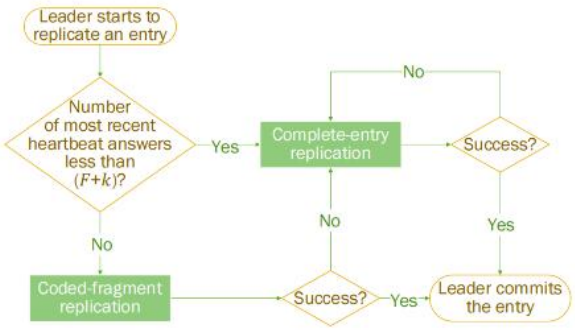


图 12 所示。

当 leader 拥有所有完整的条目时，这两种复制方法都能保证安全性和有效性。然而，当一个 leader 新当选时，新当选的 leader 的日志很可能没有完整的副本，而只有一些条目的编码碎片。当只有(F+1)个健康的服务器时，这些不完整的条目不能保证可以恢复。如果这些无法恢复的条目中有一些没有被新当选的 leader 应用，leader 就没有办法处理这些条目。因此，需要一些额外的操作来保证有效性。

新当选的 leader 的日志中的编码片段可以被 leader 应用或不应用。如果一个编码片段被应用，该条目必须是由前一个 leader 承诺的。根据两种复制方法的承诺条件，至少有 k 个编码片段或一个完整的 条目的完整副本被存储在任何（F+1）个服务器中。所以 当有(F+1)个健康的服务器时，leader 总是可以恢复这个条目。健康的服务器。然而，如果一个编码片段没有被应用。没有任何规则可以保证在有(F+1)个健康服务器时，这个条目可以被恢复。有(F+1)个健康的服务器。

为了处理未应用的编码片段，在 CRaft 中新当选的领导人应该在进行 LeaderPre 操作之前进行 在 CRaft 中，新当选的领导人应该进行 LeaderPre 操作。他们可以成为功能齐全的领导人。

新选出一个 leader 时，首先检查它日志，找出其未应用的编码片段。然后，它向 follower 询问日志，重点是未应用的编码片段的索引。如没能收集到（F+1）个以上的答案，则应等待。新的 leader 应该尝试依次恢复其未应用的编码片段。对于每个条目，如果在（F+1）答案中至少有 k 个编码片段或一个完整的副本，那么它可以被恢复，但不允许被立即提交或应用。否则，新的 leader 应该在其日志中删除这个条目和所有后面的条目。在恢复或删除所有未应用的条目之后，就可以进行整个 LeaderPre 操作了。在 LeaderPre 期间，新当选的 leader 应该不断向其他服务器发送心跳，防止它们超时并开始新的选举。

8 结语

以 Reed-Solomon 编码为代表的纠删码是大数据时代收到广泛使用的存储策略。基于纠删编码，本文首先介绍了其背景和用途，之后介绍了其优势、不足以及编码解码的基本原理，之后针对纠删编码的不足分别阐述了 5 种不同不同的优化策略。当下，纠删编码仍然具有很大的改进空间，进一步的优化仍然能够字很大程度上提升数据中心的性能，节约其成本，具有很大的发展空间。

参 考 文 献

[1] Hu, Yuchong, Liangfeng Cheng, Qiaori Yao, Patrick PC Lee, Weichun Wang, and Wei Chen. Exploiting Combined Locality for Wide-Stripe Erasure Coding in Distributed Storage. Proceedings of 19th USENIX Conference on File and Storage Technologies (FAST'21), 2021:233-248.

[2] Shiyao Lin, Guowen Gong, and Zhirong Shen, Patrick P. C. Lee, Jiwu Shu.Boosting Full-Node Repair in Erasure-Coded Storage.Proceedings of 2021 USENIX Annual Technical Conference.2021:641-656.

[3] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica. HUG: Multi-Resource Fairness for Correlated and Elastic Demands. In Proceedings of USENIX NSDI, 2016.

[4] H. Liu, M. Mukerjee, C. Li, et al. Scheduling Techniques for Hybrid Circuit/Packet Networks. In Proceedings of ACM CoNEXT, 2015.

[5] Yuya Uezato. Accelerating XOR-based Erasure Coding using Program OptimizationTechniques.arXiv:2108.02692.https://doi.org/10.48550/arXiv.2108.02692.

[6] Haiyang, Xiaoyi Lu.ShiProceedings of SC20: International Conference for High Performance Computing, Networking, Storage and Analysis.

[7] Zizhong Wang, Tongliang Li, Haixia Wang, Airan Shao, Yunren Bai, Shangming Cai, Zihan Xu, and Dongsheng Wang. CRaft: An Erasure-coding-supported Version of Raft for Reducing Storage Cost and Network Cost. Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20). 2021:297-308.