分　数：	
评卷人：	

華中科技大學

研究生（数据中心技术）课程论文（报告）

题　目：纠删码存储系统性能优化

学　　号　M202273927

姓　　名　欧　鹏

专　　业　计算机技术

课程指导教师　施展　童薇

院（系、所）　计算机科学与技术学院

2022 年 1 月 4 日

目录

纠删码存储系统性能优化

欧鹏[1)]

[1)](华中科技大学计算机科学与技术学院 湖北 武汉 430074)

摘 要 随着数据的爆炸式增长，以及人们对于数据存储的可靠性和低成本需求，使得纠删码技术变得火热起来，并被广泛应用到分布式存储系统中，尤其是数据中心。然而，纠删码在数据更新和修复失效的块时会生成大量网络流量，进而影响了整个存储系统的性能，由此引起广大学者的深入研究。目前，学术界主要从两个方向进行研究：1）构建具有低流量交互的纠删码体系；2）设计有效的更新或修复算法以减少数据传输。本文将分别讨论以下五种算法：1）RackCUE：最佳机架协同更新；2）PDL：纠删码存储系统的数据布局；3）CRaft:支持纠删码的 Raft 版本；4）纠删码存储中全节点修复的提升；5）LogECMem：内存键值存储与校验日志的耦合。并讨论针对于最小化数据流量的发展趋势。

Performance Optimization of Erasure Coded Storage Systems

Ou Peng[1)]

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

Abstract With the explosive growth of data ,and the demand for reliability and low cost of data storage, erasure code technology has become popular and has been widely used in distributed storage systems, especially in data centers. However, erasure codes generate a large amount of network traffic when the data is updated and the invalid blocks are repaired, which affects the performance of the entire storage system. At present, the academic research mainly focuses on two directions: 1) constructing erasure codes with low traffic interaction; 2) Design effective update or repair algorithms to reduce data transmission. In this article, we will discuss the following five algorithms: 1) RackCU: Optimal rack-coordinated updates; 2) PDL: the data layout of erasure coded storage systems; 3) CRaft: A version of Raft that supports erasure codes; 4) Improvement of full node repair in erasure coded storage; 5) LogECMem: Coupling erasure-coded in-memory key-value stores with parity logging. We also discuss the development trends aimed at minimizing data traffic.

1.引言

在分布式存储系统中，节点故障已成为一种常态，为了保证数据的高可靠性，系统通常采用数据冗余的方式.目前主要有 2 种冗余机制：一种是多副本，另一种是纠删码.伴随着数据量的与日俱增，多副本机制带来的效益越来越低，人们逐渐将目光转向存储效率更高的纠删码.但是纠删码本身的复杂规则导致使用纠删码的分布式存储系统的读、写、更新操作的开销相比于多副本较大，所以纠删码在实际使用中存在着一定的限制。本文专注于纠删码存储系统的性能优化，如何在纠删码更新规则和修复规则的系统架构角度优化纠删码存储系统，尤其是最小化交互的数据流量，这也是未来值得继续深入研究的问题。

本文主要调研了四篇 3 年内的顶会文献，第一篇《PDL: A Data Layout towards Fast Failure Recovery for Erasure-coded Distributed Storage Systems 》 (INFOCOM' 2020)，第二篇《 CRaft: An Erasure-coding-supported Version of Raft for Reducing Storage Cost and Network Cost 》(FAST'20) ， 第 三 篇 《 Boosting Full-Node Repair in Erasure-Coded Storage》(USENIX'21)，第四篇《LogECMem: Coupling Erasure-Coded In-Memory Key-Value Stores with Parity Logging》(SC'21)。

2 最佳机架协同更新

2.1 介绍

本文出自 infocom 21，研究方向是机架间检验更新流量的优化。在本文中，文中提出了一种新的机架协调更新机制来抑制跨机架更新流量，该机制包括两个连续的阶段:收集数据增量块的增量收集阶段，以及基于更新模式和奇偶校验布局更新奇偶校验块的另一个选择性奇偶校验更新阶段。文中进一步设计了 RackCU，这是一种最佳的机架协调更新解决方案,可以实现跨机架更新流量的理论下限。最后，在大规模模拟和真实数据中心实验的基础上，文中进行了广泛的评估，表明 RackCU 可以减少 22.1%-75.l%的跨机架更新流量，从而提高 34.2%-292.6%的更新吞吐量。

2.2 研究背景

现有的纠删码更新研究主要集中在减少磁盘寻道、减少正在更新的校验块的数量以及减少更新流量。虽然 CAU 可以减少跨机架更新流量，但它降低了系统可靠性(通过推迟奇偶校验更新)，并且无法实现理论上最小的跨机架更新流量。不幸的是，如何在不损害系统可靠性的情况下最小化跨机架更新流量，在很大程度上被现有的研究所忽略。文中提出了机架协调更新，一种新的奇偶校验更新机制，包括增量收集阶段和另一个选择性校验更新阶段，以在数据更新后立即更新校验块，目标是在保证系统可靠性的情况下最小化跨机架更新流量。

机架协调更新的主要思想是在一些专用机架(称为收集器机架)中收集数据增量(即新旧数据块之间的差异)，并通过选择适当的更新方法来更新校验块。通过根据更新模式和校验块布局选择收集器机架，文中进一步设计了 RackCU，这是一种最佳的机架协调更新解决方案，能够以线性计算复杂性达到跨机架更新流量的下限。总而言之，文中的贡献包括:

•提出了一种新的机架协调更新机制，旨在显著减少跨机架更新流量。

•设计了 RackCU，这是一种最佳的机架协调更新解决方案，可以达到跨机架更新流量的下限。文中还表明，RackGU 是不同的代表性纠删码的一般设计。

•实现了一个 RackCU 原型，并通过大规模模拟和阿里云弹性计算服务(ECS) 实验进行了广泛的评估，表明 RackCU 减少了 22.l%-75.l%的跨机架更新流量，从而增加了 34.2%-292.69%的更新吞吐量。

2.3 方法介绍及结构设计

1）方法介绍

RackCU 方法聚焦于具有两层分层架构的数据中心，首先将一群节点组织到一个机架中，多个机架通过网络核心(即聚合和核心交换机)进一步互连。以 RS 纠删码为例进行示范，但可以应用在其他纠删码的存储系统上。使用参数 k 和 m，在编码阶段，纠删码对 k 个数据组块进行编码，以通过伽罗瓦有限域上的算法生成额外的 m 个校验组块。这些 k + m 块编码在一起共同构成一个条带，保证一个条带内 k + m 个块中的任何 k 个足以再现原始 k 个数据块。

A.纠删码中的校验更新

本文主要考虑纠删码中基于增量的更新。假设{D1,D2,...,Dk}和{P1,P2,...,Pm}分别表示一个条带的 k 个数据块和 m 个校验块。每个校验块 Pj(1≤j≤m)可以通过伽罗瓦域算法计算 k 个数据块的联合线性计算得出，由下式给出

$$P_j = \sum_{i=1}^{k} \gamma_{i,j} D_i, \qquad (1)$$

其中 $\gamma_{i,j}$(1≤i≤k 和 1≤j≤m)是数据块 Di 用来计算奇偶校验块 P_j 的编码系数。

假设数据块 D_h 被更新为 D_h’(1≤h≤k)。

为了保证数据块和校验块之间的编码一致 y，每个奇偶校验块 P_j(1≤j≤m)应该基于等式(1)进行如下地更新:

$$P_j' = P_j + \gamma_{h,j}(D_h' - D_h). \qquad (2)$$

等式(2)表示新的校验块可以通过利用旧的校验块 P_j 和数据块的增量(即 D_h’-D_h ,新旧数据块之间的差)或校验块的增量(即新旧校验块之间的差异)，而不必访问未改变的数据块。此外，作为编码系数 $\gamma_{i,j}$，一旦建立了参数 k 和 m，它们对所有节点都是公开的，而不必重新传输。

B.纠删码数据中心的校验更新

假设机架 Rx 中的数据块{D_1,D_2,...,D_{ux} }被更新为{D_1',D_2',...,D_{ux}'}，其中 ux 表示 Rx 中更新的数据块的数量。基于等式(2)，文中可以将 P_j'的计算概括为:

$$P_j' = P_j + \sum_{h=1}^{u_x} \gamma_{h,j} \Delta D_h = P_j + \Delta P_j \qquad (3)$$

综上，这里有两种方法去更新校验块，分别是基于数据增量的更新和基于校验增量的更新。

基于数据增量的更新:通过直接传输数据增量块，批量更新一个机架的校验块。它首先计算在机架 R_x 中更新的 u_x 个数据块的 u_x 个数据增量块,并将它们发送到机架 R_y 中的中继节点，该中继节点然后将 u_x 个数据增量块转发到存储奇偶校验块的 R_y 的相应 t_y 个节点。对于保存奇偶校验块 P_j 的节点，它将根据等式(3)读取旧的校验块(即 P_j)计算新的校验块(即 P_j’)。

基于校验增量的更新:它通过传输相应的校验增量块来更新另一个机架中的每个校验块。为更新机架 R_y 机架中的校验块 P_j，基于校验增量的更新方法首先在数据机架 R_x 中计算校验增量块 ΔP_j，然后发送到 R_y。最后，根据等式(3)基于旧的校验块 P_j 和接收到的 ΔP_j 生成新的校验块 P_j’。

原则上，机架协调更新是基于数据增量的更新和基于校验增量的更新方法的综合。主要的思想是在保证系统可靠性的情况下减少跨机架更新流量，允许机架在数据块更新后立即协调校验块更新。它将整个校验更新过程分为增量收集阶段和另一个接续执行的选择校验更新阶段。具体地说，在增量收集阶段，机架协调更新机制将选择几个收集器机架，它们负责从其他机架收集数据增量块。另一方面，选择校验更新阶段将根据数据中心的更新模式和校验块布局选择基于数据增量的更新或基于校验增量的更新来更新校验块，主要目的是减少跨机架更新流量。

具体来说，假设一个机架 R_x 有 u_x 个更新的数据块，另一个机架 R_y 存储 t_y 个校验块。选择校验更新执行以下动作:如果 $u_x \leqslant t_y$，它使用基于数据增量的更新，从 R_x 向 R_y 发送 u_x 个数据增量块，批量更新 t_y 个数据块；否则，它会使用基于校验增量的更新，传输相应的 t_y 个校验增量块。因此，通过机架 R_x 中的 u_x 个更新的数据块来更新 R_y 中的 t_y 个校验块，选择校验更新仅需要跨机架传输 $\min\{u_x,t_y\}$ 块。

给定一个条带，假定更新的数据块分布在 d 个数据机架上 $\{u_1,u_2,...,u_d\}$（$u_i \leqslant m$），需要更新的 m 个校验块分布在 p 个校验机架 $\{t_{d+1},t_{d+2},...,t_{d+p}\}$ 上。文中选择 R_d*和 R_p*来分别表示拥有更新数量最多的数据机架和校验机架。根据以下规则来决定收集器机架 L：

如果 R_d*上更新的数据块的数量不少于 R_p*上的校验块的数量，则选择 R_d*作为收集器机架 L；

否则，选择 R_p*作为收集器机架 L。

通过分析，可得以下结论：

结论 1：在所有的解集中，有包含 L 收集器的解，需传输的数据块最少。

结论 2：仅有一个收集器且为 L 的解，所产生的流量最少。

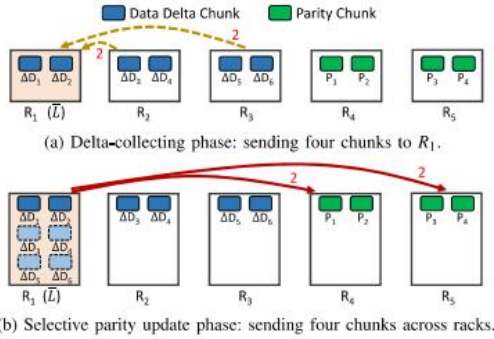

图 1.单一收集器

如图 1 所示，将 R_1 作为唯一的收集器 L，增量收集阶段，R_1 会收集 R_2 与 R_3 的数据增量块，此阶段传输 4 个数据块；选择校验更新阶段，由于 R_1 的数据增量块为 6 个，大于 R_4 与 R_5 分别拥有的 2 个校验块，采用校验增量更新方法，此阶段共传输 4 个数据块。综上整个更新过程只传输了 8 个数据块，是为数据传输最少。

关于上述 L 收集器的校验更新过程，使用 C 语言进行如下代码实现：

Algorithm 1 Procedure of RackCU

Input: $\{R_1, R_2, \cdots, R_d\}$ (data racks)
$\{u_1, u_2, \cdots, u_d\}$ (distribution of updated data chunks)
$\{R_{d+1}, R_{d+2}, \cdots, R_{d+p}\}$ (parity racks)
$\{t_{d+1}, t_{d+2}, \cdots, t_{d+p}\}$ (distribution of parity chunks)
Output: The new $n-k$ parity chunks of the same stripe

Find the data rack R_{d^*}, where $u_{d^*} = \max\{u_i | 1 \leq i \leq d\}$
Find the parity rack R_{p^*}, where $t_{p^*} = \max\{t_{d+j} | 1 \leq j \leq p\}$
// Determine the sole collector rack
if $u_{d^*} \geq t_{p^*}$ **then**
    $\overline{L} = R_{d^*}$
else
    $\overline{L} = R_{p^*}$
end if
// Delta-collecting phase
for $1 \leq i \leq d$ **do**
    Send the u_i data delta chunks from R_i to $\overline{L}$
end for
// Selective parity update phase
for $1 \leq j \leq p$ **do**
    if $\overline{l} > t_{d+j}$ **then**
        Send the t_{d+j} parity delta chunks to R_{d+j}
    else
        Send the $\overline{l}$ data delta chunks to R_{d+j}
    end if
    Update the t_{d+j} parity chunks
end for

2）结构设计

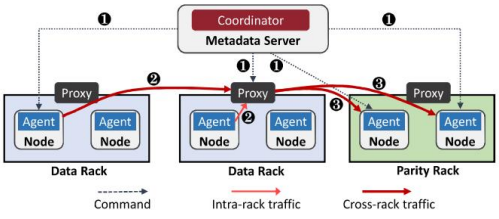

图 2.RackCU 模型

如图 2 所示，文中设计了 RackCU 模型用以实现最佳机架间流量更新机制。每个机架上都有一个 Proxy，每个数据节点都有一个 Agent，整个条带上有一个 Coordinator。

Coordinator 根据更新的数据块和对应的校验块的分布决定收集器机架，并发布指令给 Agent 和 Proxy 建立校验更新机制。接收到指令后，每个更新的数据节点的 Agent 会计算出增量数据块并发送给收集器的 Proxy。收集到足够的数据增量块后，Proxy 决定采用何种机制来更新校验块。一旦产生了新的校验块，校验块会发送一个确认 ACK 给 Coordinator，Coordinator 从同一条带上所有的 m 个校验块都收到了确认 ACK 后，认定此次更新成功。

2.4 小结

文章提出了一种新的机架协调更新机制，旨在显著减少跨机架更新流量。作者设计了 RackCU，一个最优的机架协同更新方案，达到跨机架更新流量的下界。RackCU 是一种适用于不同代表性纠删码的通用设计。文中实现了 RackCU 模型系统，并通过大规模模拟实验和阿里巴巴云弹性计算服务(ECS)实验进行了验证，结果表明 RackCU 减少了 22.1%～75.1%的跨机架更新流量，提高了 34.2%～292.6%的更新吞吐量。

3. 相关研究

本节主要是对四篇关于纠删码存储系统进行性能优化的相关文献[1][2][3][4]进行简单讲述与介绍，并列出文献中作者观测得到的结论以及实验后的研究成果。

3.1 PDL：纠删码存储系统的数据布局

3.1.1 介绍

本论文出自 INFORM 21，主要是一种面向纠删码分布式存储系统故障快速恢复的数据布局方法。

纠删码以低存储开销提供高可靠性，在分布式存储系统中得到越来越广泛的应用。然而，传统的随机数据放置方法在故障恢复过程中会造成跨机架流量过大、负载严重不均衡，严重影响恢复性能。此外，DSS 中多种纠删码策略共存加剧了上述问题。本文提出 PDL，一种基于 PBD 的数据布局，用于优化决策支持系统的故障恢复性能。PDL 基于具有统一数学性质的组合设计方案——两两平衡设计构造，从而给出了一种统一的数据布局。然后，作者提出了一种基于 PDL 的故障恢复方案 rPDL。rPDL 通过均匀地选择替换节点并提取确定的可用数据块来恢复丢失的数据块，有效地减少了跨机架流量，并提供了近乎均衡的跨机架流量分布。

3.1.2 研究背景

用一个(n，m)传统编码方法由以下 3 个步骤组成:(i)选择一个替换节点;(ii)从源节点读取同一条带的 n 个块到替换节点;(iii)重构替换节点中的失效块。为了恢复失效节点，需要重构其中所有的数据块/校验块，这导致了巨大的 I/O 负载和网络流量。因此，加快故障恢复过程对 DSS 至关重要。

除了故障恢复所需要的大量 I/O、流量和 CPU 负载外，本文还发现以下因素进一步降低了故障恢复的速度。

由于故障恢复，机架间的通信量很大。采用随机数据布局方法，当机架数量足够时，每个条带只需存储一个数据块即可达到机架级最大容错能力。但由于故障恢复，它会导致大量的横线交通。

块的非均匀分布。随机数据布局在概率上达到了条带数量足够多的块的近似均匀分布。但由于 I/O、内存空间和计算资源有限，在每轮故障恢复过程中，恢复的故障条带数通常与节点数处于同一数量级，远远不能得到均匀的块分布。故障恢复时跨机架流量不均衡。在失效恢复过程中，替换节点和检索块的选择对负载均衡也起着至关重要的作用。如果替换节点聚集在某些机架上(如 HDFS 选择邻近的替换节点，导致少数机架上出现替换节点)，那么将检索到的数据读取到替换节点的内存中，会大大增加 ToR 在这些机架上的跨机架下行负载。如果将检索到的数据块聚集在某些机架中，则会从这些机架中读取大量数据进行故障恢复，大大增加了 ToR 跨机架的上行负载。

综上所述，作者在 DSS 中设计了支持混合纠删码策略的统一数据布局，并提出了一种高效的恢复方案，以减少和均衡机架间的流量以实现故障恢复。

3.1.2 PDL 设计及快速恢复

（1）PDL 设计

PBD 和 BIBD 是组合理论中的两类重要设计，在实验设置、软件测试、数据布局等方面有着广泛的应用。PBD 用于描述 DSS 中纠删码的数据布局，BIBDs 可以构造该数据布局。

为了提高混合纠删码 DSS 系统的故障恢复速度，作者提出了一种基于 PBD 的数据布局 PDL，使数据/校验块均匀分布。

在 PDL 算法中，首先对条带中的所有数据块进行分组，将分组后的数据块分配到机架中，并将同一组中的数据块分配到同一机架的节点上，以达到数据块均匀分布的目的。因此，可以在故障恢复过程中在机架内部部署部分解码，以减少跨机架的流量。

A.分组条带的块

对于(n_i,m_i)码，应该分配满足以下要求的块。

•同一条带的最多 m_i 个块分配到同一机架，以容忍一个机架故障;

•同一条带的任意两个块不分布到同一节点，以容忍 m_i 并发节点故障。

为了最大化部分解码的效果，应该最小化条带的组数。但从另一个角度来看，一个机架包含最多 m_i 块的相同条带，以容忍一个机架故障的 (n_i,m_i) 代码。因此 $Ng(n_i,m_i) = [(n_i+m_i)/m_i]$是可以容忍一个机架故障的最小组数，称包含精确 m_i 块的组为满的，否则为非满的。为了使块的分布均匀，试图将所有块分成组，使任何组中的块数量最多相差 1。

B.使用 PBD 表分发数据

根据最后一小节中所有纠删码的分组构造 PBD 表，并基于 PBD 表将分组分配到机架中。

将组映射到机架。当编写一个编码为(n_i,m_i)码的条带时，按顺序在 PBD 表中查找一个大小为 $k_i = Ng(n_i,m_i)$的元组，然后通过随机映射将条带的 ki 组分配到指定的机架上，每组分配到一个机架上。例如，要写一个 RS(10,4)编码的条带，我们可以根据图 3(b)中的第一个元组(0,3,4,5)，将条带的四组 G_0、G_1、G_2 和 G_3 分别分配到机架 R4、R0、R3 和 R5 中，其中源机架 R3 和 R5 用于非完整的 G_2 和 G_3。在图 3(a)中，$G_{l,i,j}$ 是第 i 条纠删码的 j 组，其中 l = 0,1,2,3 分别表示 RS(10,4)， RS(12,4)，RS(4,3)和 RS(6,3)的使用。从图 3(a)中，可以发现所有组都均匀分布在所有机架上。

将块映射到机架内的节点。在每个机架内，同组的块按轮询方式分配给各节点，因此块分布均匀，即每个机架内各节点的块数相等或不相等 1。

通过以上两步，实现了基于 PBD 表的均匀数据分布 PDL。

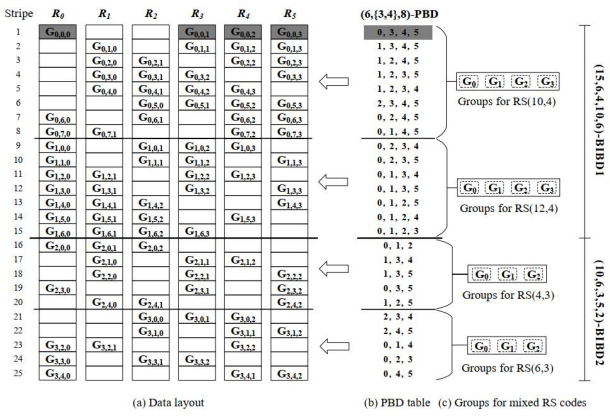

图 3.混合纠删码的数据分布

（2）故障快速恢复

A.基于 PDL 的单故障恢复方案

基于 PDL 的均匀数据布局，文章提出了一种单故障恢复方案 rPDL。该方案由 3 个步骤组成，即选择替换节点、选择剩余块恢复丢失块和解码方案设计。假设机架 Rf 中的一个节点 Nf 失效。同上，将一个条带划分为(n,m)编码为 k = Ng(n,m)组，每个组由 m 个或小于 m 个块组成，分别称为全组或非全组。我们将每个组分配到一个单独的机架，并将具有全(非满)组的机架称为全(非满)机架。给定一个(n,m)编码，将存储 S 的块的机架称为源机架。如果除 R_f 之外的所有源机架都是满的，则称条带为满的；如果除 R_f 之外的某些源机架是非满的，则称条带为非满的。

1)更换节点的选择:为保证容忍一个机架故障，文章提出以下更换节点的选择方案。

•当条带 S 满时，从非源机架中随机选择待更换节点。

•否则，更换节点需在非全源机架中随机选取。

在 Rf 中的一个故障节点上，Rf 是所有条纹的源机架，而其他机架中的任何一个都是只有一小部分条纹的源机架，因为实际 dss 中机架的数量远远大于纠删码的分组大小。如果我们从随机选择的包含 Rf 的源机架中选择一个替换节点，那么 R_f 将比其他机架承受更大的跨机架流量。因此，我们不从 R_f 中选择替换节点。

2)选择幸存块来恢复丢失的块:注意，我们的恢复算法不会从 R_f 中读取幸存数据以达到负载均衡(与上述替换节点的选择方案类似)。由下面引理 1 可知，除 Rf 外的所有源架都将参与故障块的重构。

3)译码方案设计:对于一个 (n,m)码，任意块是其他 n 个块的线性组合，如 $B0 = \sum c_i * B_i$，其中 c_i 为译码系数，i =1,2,..,n.因此，为了减少跨机架重构的流量，可以在机架中随机选取一个源节点(称为计算节点)，通过部分解码将同一机架中检索到的块进行聚合。替换节点接收除 R_f 外的所有源机架的汇聚块，用于重构故障块 B0。

简单来说，将用于故障恢复的跨机架流量称为跨机架重构流量，即从源机架跨机架读取数据块到替换节点时，会发生源机架的跨机架重构读和替换节点所在机架的跨机架重构写。基于以上的架构分析，文中得出 rPDL 引入了重构每个条带单节点故障的最小 CRRT 和 rPDL 在纠删码中使用足够的条带可以实现几乎均衡的恢复流量。

B.多故障修复讨论

rPDL 用于多个故障。当混合纠删码决策支持系统中存在多个并发故障时，可以使用 rPDL 依次对故障进行恢复，从而实现负载均衡和每个故障恢复的最小 CRRT。但是用于所有故障恢复的总 CRRT 将很高。PDL 数据布局有利于故障恢复时数据块和 I/ O 的均匀分布，加快了多故障的恢

复速度。

rPDL 恢复单故障后保留 PDL。如果 Rf 中存在带 mi 块的非全失败条带，则恢复的条带保持 PDL 数据布局的属性。对于其他失效的条带，只需在集群空闲时将重构块迁移到 Rf。对于一个(n,m)纠删码，迁移的块只占总块的 1/(n+m)。此外，重构块在除 Rf 外的机架间分布均匀，因此每个机架向 Rf 迁移的负载也均匀。

3.1.3 小结

文章提出了一种基于 PBD 的数据布局 PDL，以有效地加速混合纠删码 DSS 的故障恢复。PDL 为采用多纠删码的分布式系统故障恢复提供了数据均匀分布和均衡访问的通用支持。更重要的是，与现有的随机数据分布相比，PDL 具有更短的降级读延迟、最小的跨机架重构流量、更好的跨机架重构读/跨机架重构写负载均衡和更高的恢复吞吐量等性能。在故障恢复窗口内，PDL 显著降低了前台用户请求的响应延迟。

3.2 CRaft:支持纠删码的 Raft 版本

3.2.1 介绍

本论文出自 FAST 20，作者在 Raft 的基础上提出支持纠删码的 CRaft，旨在用于减少存储成本和网络成本。

共识协议能够提供高可靠、高可用的分布式服务。在这些协议中，日志条目被完全复制到所有服务器。这种全表项复制会导致较高的存储和网络成本，影响性能。纠删码是一种常用的技术，可以在保持容错能力不变的情况下降低存储和网络成本。如果将一致性协议中的完全复制替换为纠删码复制，可以大大降低存储和网络成本。RS-Paxos 是第一个支持纠删码数据的一致性协议，但与常用的一致性协议(如 Paxos 和 Raft)相比，其可用性较差。指出了 RSPaxos 的活性问题，并尝试解决这个问题。在 Raft 协议的基础上，提出了一种新的协议 CRaft。CRaft 提供两种不同的复制方法，可以像 RS-Paxos 一样使用纠删码来节省存储和网络成本，同时保持与 Raft 相同的活性。作者基于 CRaft 构建了一个 key-value 存储，并对其进行了评估。实验结果表明，与原始 Raft 相比，CRaft 可节省 66%的存储空间，提高 250%的写吞吐量，降低 60.8%的写延迟

3.2.2 研究背景

Raft 协议是共识协议之一，其规则保证了活跃性。通常，使用共识协议的系统中的服务器数量 N 是奇数。假设 N = 2F + 1，那么 Raft 可以容忍任何 F 个故障。如果将共识协议的活性级别定义为它可以容忍的故障数量，因此 Raft 具有 F 活性级别。活性水平越高，活性越好。没有协议可以达到(F+1)活性级别。如果存在一个(F+1)活性级别的协议，则该协议可以分为两组，由 F 台健康服务器组成，这两组服务器可以分别协商不同的内容，这违背了协议的安全性。安全性和活性是共识协议最重要的性质。Raft 可以保证系统的安全性能始终保持，并达到可能的最高活性水平 F，为系统的建立奠定了良好的基础。根据这些特性，作者选择 Raft 作为设计新协议的基础。

RS-Paxos 是 Paxos 的改进版本，它将纠删编码与 Paxos 相结合，节省了存储和网络成本。在 Paxos 中，命令是完全传输的。然而，RS-Paxos 采用编码片段的方式传输命令。根据这一改变，服务器在 RS-Paxos 中可以只存储和传输碎片，从而降低存储和网络成本。

3.2.3 CRaft 设计

A.节点片段复制

当一个 leader 试图通过节点片段复制方法复制一个条目时，它首先对条目进行编码。在 Raft 中，每个日志条目都应该包含来自客户机的原始内容，以及协议中的 Term 和 index。当 leader 试图编码一个条目时，内容可以被协议选择的(k,m)-RS 编码成 N = (k+m)个片段。Term 和 index 在协议中起着重要的作用，因此不应该进行编码。图 4 显示了编码过程。

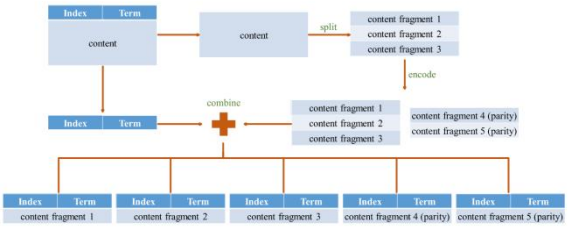

图 4.CRaft 的编码流程

编码后，leader 将拥有 N 个节点片段的条目。然后，它将向每个 follower 发送相应的节点片段。在收到对应的节点片段后，每个 follower 会回复 leader。当 leader 确认至少(F + k)个服务器存储了一个节点片段时，条目和它之前的条目都可以安全地应用。leader 会提交并应用这些条目，然后通知 followers 去应用。节点片段复制的条件比 Raft 更严格。此提交条件还意味着，当没有(F+k)个健康服务器时，leader 不能使用节点片段复制来复制条目，然后提交它。

当一个 leader 下线时，新的 leader 会被选举出来。如果一个条目已经提交，Raft 的选举规则保证新 leader 至少有一个条目的片段，这意味着安全属性可以得到保证。由

于至少有(F +k)台服务器存储了已提交数据的片段，因此在任何(F + 1)台服务器中至少有 k 个已编码的片段。因此，当至少有(F+1)个健康的服务器时，新的 leader 可以收集 k 个节点片段，然后恢复完整的条目，这意味着活性可以得到保证。在节点片段复制中，关注者可以接收和存储条目的编码片段。但是，在 Raft 中，跟随者必须接收并存储完整的条目。根据编码过程的不同，节点片段的大小约为全集大小的 1/ k。因此，使用节点片段复制可以大大降低存储和网络开销。

B.全条目复制

为了降低存储和网络成本，鼓励 leader 使用编码片段复制。然而，当没有(F +k)个健康服务器时，节点片段复制将无法工作。当健康服务器的数量大于 F 且小于(F +k)时，leader 应该使用全条目复制方法来复制条目。

在全条目复制中，leader 必须在提交输入之前将完整输入复制到至少(F +1)个服务器，就像 Raft 一样。由于提交规则与 Raft 相同，安全性和活跃性都不是问题。然而，由于 CRaft 支持节点片段，在提交条目后，leader 可以按节点片段复制条目，而不是向剩余的 follower 复制完整的条目。

在实际实现中，有许多策略可以通过全项复制来复制一个条目。定义一个整数参数 0 < p < F。leader 可以先将一个条目的完整副本发送给(F + p) followers，然后将节点片段发送给剩余的(F + p) followers。较小的 p 意味着更少的存储和网络成本，但也意味着更长的提交延迟(如果(F + p)中没有 F 的响应及时返回，在提交之前可能需要更多轮的通信)。当 p = F 时，策略与 Raft 的复制方法相同。

C.预测

在两种复制方法都能成功的情况下，使用编码片段复制比使用完全复制可以获得更好的性能。贪婪策略是 leader 总是试图通过节点片段复制来复制条目。如果它发现没有(F+k)健康的服务器，它转而通过完全复制复制条目。然而，如果 leader 已经知道健康服务器的数量小于(F + k)，那么通过节点片段进行的第一次复制是没有意义的。

准确地选择复制方法可以达到最佳性能。但是 leader 不能确定其他服务器的状态。因此，当它试图复制条目时，它只能预测可以与多少台健康服务器通信。leader 可以使用最新的心跳响应来估计健康服务器的数量。这个预测应该足够准确。然而这个预测与 leader 选择复制上一个条目的方法无关。它只依赖于最新的心跳答案。因此，leader 很有可能使用完全复制来复制最后一个条目，然后它自动选择节点片段复制来复制新条目。

D.新当选的 leader

两种复制方法都能在 leader 拥有所有完整条目的情况下保证安全性和活性。然而，当新当选领导人时，新当选领导人的日志很可能没有完整的副本，而只有一些条目的节点片段。当只有(F +1)个健康服务器时，不能保证这些不完整的条目可以恢复。如果新当选的 leader 没有应用这些不可恢复的项目，该 leader 没有办法处理这些项目。根据 Raft 的规则，leader 从客户端收到的新条目也不能复制到 followers。因此协议无法充分发挥作用，协议的活性也无法得到保证。因此，需要一些额外的操作来保证活性。

因此，作者引入 LeaderPre 操作：当新领导人当选时，它首先检查自己的日志，找出未应用的代码片段。然后，它向关注者请求他们自己的日志，重点关注未应用的代码片段的索引。至少收集(F + 1)个答案(包括新 leader 本身)，或者新 leader 继续等待。新的 leader 应该尝试按顺序恢复其未应用的代码片段。对于每个条目，如果在(F+ 1) 应答中至少有 k 个节点片段或 1 个完整副本，则该条目可以被恢复，但不允许立即提交或应用。否则，新的 leader 应该在其日志中删除此条目和所有以下条目(包括完整条目)。在恢复或删除所有未使用的表项后，可以进行整个 LeaderPre 操作。在 LeaderPre 期间，新当选的领导人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。

3.2.4 总结

文章在 Raft 的基础上扩展成 CRaf，使之支持纠删编码。借助纠删编码，可以大大减少存储和网络开销。

之前的纠删码支持协议 RSPaxos 未能保持像 Paxos 或 Raft 那样的活性级别。CRaft 解决了这个问题。通过分析不同协议的性能，可得 CRaft 在具有最佳活性的同时，能够最大程度地降低存储和网络开销的结论。文章中设计了一个键值存储并在上面进行了实验。实验结果表明，与 Raft 相比，CRaft 降低了 60.8%的延迟，提高了 250%的吞吐量。

3.3 纠删码存储中全节点修复的提升

3.3.1 介绍

本文出自 USENIX 21，研究方向是纠删码的修复算法的优化。在本文中，文中提出了一种调度框架 RepairBoost，可以辅助现有的线性纠删码和修复算法提高全节点修复性能。RepairBoost 建立在三个设计原语之上:(i)修复抽象，采用有向无环图来描述单块修复过程;(ii)修复流量均衡，即同

时均衡上传和下载修复流量;(iii)传输调度，谨慎地分发请求的数据块，使空闲带宽达到饱和。在 Amazon EC2 上的大量实验表明，RepairBoost 对各种纠删码和修复算法的修复速度提高了 35.0～97.1%

3.3.2 研究背景

通过研究现有的加速纠删码存储修复的尝试，发现存在 4 个限制，如果不适当解决，将直接影响全节点修复时的性能增益。

限制 1 :无法使用全双工传输。大多数现有研究忽略了修复中的全双工传输，它使节点能够同时且独立地发送(上传)和接收(下载)数据;导致在全节点修复时无法很好地均衡上传和下载修复流量。

限制 2:每个时隙没有充分利用带宽。现有的修复算法虽然可以缓解单块修复中的下载瓶颈，但它们只是将多个块的修复方案组合起来，以应对全节点修复。这可能在无意中导致链路再次拥塞，使全节点修复时带宽利用率不足。

限制 3 :不灵活。许多修复算法平等地对待每个块，并使用相同的修复算法修复所有丢失的块。这种修复方式简单但缺乏灵活性，不能将不同的修复算法进行弹性组合，使它们能够协同应对现实存储系统中不同的可靠性需求和倾斜访问流行度。

限制 4 :缺乏全节点修复的通用框架。现有的分布式存储系统针对单块修复部署了不同的修复算法。例如，许多商用存储系统(如 Hadoop HDFS 和 Windows Azure storage)由于其简单性，仍然依赖 CR 进行单块修复，而 PPR 和 CAR 可以用于层次存储系统，以减少机架间修复流量。故需要有一个通用的框架，同时支持针对不同部署场景的不同类型的修复算法。

3.3.3 RepairBoost 设计

A 修复概述

RDAG 构建:首先通过有向无环图(directed acyclic graph, DAG)公式化单块修复方案，称为修复 DAG (repair DAG, RDAG)。对于 RS(k,m)，丢失数据块的 RDAG 可以初始化为 k + 1 个顶点，其中 k 个顶点$\{v_1,v_2,...,v_k\}$表示存储的用于修复的幸存 k 个节点，而 v_{k+1} 表示目标节点。同时，用顶点间的有向边表示修复算法中指定的数据路由方向。

根据以下规则构建边缘。给定两个顶点 v_i 和 $v_j(1\leqslant i\neq j\leqslant k+1)$，用一条有向边 $e_{i,j}$ 表示 v_i 指定向 v_j 发送一个幸存的块。因此，如果 $e_{i,j}$ 存在，我们说 v_i 是 v_j 的子节点，反之 v_j 是 v_i 的父节点。对于 v_j(其中 $j\neq k+1$)，它必须从其子节点收集所有幸存块，使用解码系数将它们与存储的本地数据相加，并将结果发送给父节点。因此，在这个 RDAG 中，v_j 可以在接收到所需的它的子节点所有数据块后，发送一个数据块进行修复。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 中有多个父节点和子节点。

RDAG 指导的修复过程:修复从叶顶点(即没有任何子节点)开始，结束于 v_{k+1}:对于每条边 $e_{i,j}(1\leq i\neq j\leq k+1)$，如果 v_i 已经将请求的块发送给 v_j 进行修复，则删除来自 RDAG 的 $e_{i,j}$；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输给它的父节点)，我们也将 v_i 从 RDAG 中删除。因此，随着修复的进行，RDAG 中的顶点数量将逐渐减少，当顶点 v_{k+1} 最终成为叶顶点时，丢失的数据块将被成功修复。

RDAG 的优点:首先，RDAG 是单块修复解决方案的一般形式。也就是说，只要确定了需要修复的剩余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 RDAG。因此，RepairBoost 适用于多种纠删码(如 RS 码、LRCs 码、再生码等)和单条带修复算法(如 CR、PPR、ECPipe 等)。这种设计还解决了部署特定修复算法(即通用性)的紧张关系，并促进了它们的共存(即灵活性)。

其次，RDAG 描述了数据是如何在网络上传输的，以及参与修复的 k 个幸存块之间的依赖关系。给定一个 RDAG，我们可以很容易地确定在一个 RDAG 中具有不同父节点的叶节点可以潜在地并行传输，以占用可用带宽而不会出现网络拥塞(例如图 5 中的 v1 和 v3)。

第三，RDAG 表示每个顶点的修复任务。例如，我们可以了解到图 5 中的 v4 需要下载两个块，并在修复中上传一个块。

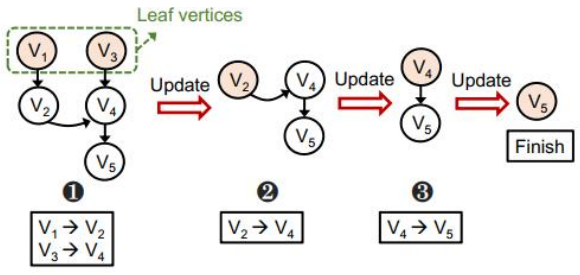

图 5.RDAG 的一个例子

B.均衡修复流量

在为失效节点上的丢失数据块构建 RDAG 后，RepairBoost 通过映射顶点到节点来分配修复任务，使得(i)修复后必须保持纠删编码提供的容错程度(即可容忍的失效节点数量)，(ii)整个系统的上传和下载修复流量尽可能均衡。

容错度保持:给定一个丢失数据块的 RDAG，将其 k

个顶点{v_1,v_2,...,v_k}分配给存储同一条带中幸存数据块的 k 个节点。我们还将 v_{k+1} 映射到修复前不存储同一条带的任何块的节点。通过这样做，RepairBoost 确保同一条带的 k +m 块在修复后仍然驻留在 k +m 个不同的节点中，从而保持纠删码提供的节点级容错。

均衡修复流量:给定一个 RDAG，用元组(u_{vi},d_{vi})表示顶点 vi 的修复任务(其中 1≤i≤k+1)，其中的参数分别表示该 RDAG 中顶点 v_i 上传和下载的数据块数量。

将 RDAG 的顶点分为 3 类:(i)叶顶点 v_i 只发送(上传)用于修复的幸存块(即 $u_{vi} > 0$ 和 $d_{vi} = 0$);(ii)根顶点，只接收(下载)修复数据(即 $u_{vi} = 0$ 和 $d_{vi} > 0$);(iii)中间顶点，从子节点接收(下载)多个数据块，并向父节点发送(上传)一个中间数据块(即 $u_{vi} = 1, d_{vi} > 0$)。例如图 5 中，v_1 是叶节点，v_2 是中间节点，v_5 是根节点。

在从多个 RDAG 中收集这 3 类顶点后，首先以均衡下载修复流量为首要目标，将中间顶点和根顶点优先映射到节点上;然后，仔细分配叶顶点，以进一步平衡上传修复流量。如下算法给出了中间顶点映射算法的伪代码。

算法描述:设 T 为尚未赋值的中间顶点的集合。设 M

Algorithm 1 Mapping of Intermediate Vertices

Input: $\mathcal{T}$ (set of intermediate vertices)
Output: The mapping $\mathcal{M}$ from intermediate vertices to nodes

```
1:  procedure MAIN(T)
2:      Set M = ϕ
3:      Sort T in descending order
4:      while T ≠ ϕ do
5:          v̄ = POP(T)
6:          Establish N
7:          N* = MAP(v̄, N)
8:          Append (v̄, N*) to M
9:      end while
10:     return M
11: end procedure
12: function MAP(v̄, N)
13:     Set N* = arg min{d_Ni | N_i ∈ N}
14:     T = T − v̄
15:     u_N* = u_N* + u_v̄
16:     d_N* = d_N* + d_v̄
17:     return N*
18: end function
```

为建立的映射。RepairBoost 首先初始化 M 作为一个空集和各种中间顶点在 T 块需要下载的数量降序排列(算法 1 的 2 - 3 行)。然后它定义顶点$\bar{v}$为需要下载修复的最大块的顶点 T(第 5 行)。RepairBoost 发现幸存的块存储在$\bar{v}$的 RDAG 的节点的集合(用 N 表示),却没有被分配的维修任务(第 6 行),然后通过调用 MAP 函数(第 7 行)为$\bar{v}$搜索节点。

在 MAP 函数中，RepairBoost 选择 N 中下载流量最少的节点 N*(第 13 行，其中 d_{Ni} 表示节点 N_i 的下载块数)。然后它将$\bar{v}$映射到 N*。RepairBoost 随后将$\bar{v}$从 T 集合中排除(第 14 行)，并通过$\bar{v}$的任务(第 15 行至第 16 行)增加 N* 的上传和下载块的数量(以 u_{N*} 和 d_{N*} 表示)。该映射表明，通过上传和下载$\bar{v}$指定的请求幸存块，N* 将作为相应 RDAG 中的$\bar{v}$。RepairBoost 重复上述步骤，直到所有中间顶点都被分配到相应的节点(行 4-9)。

然后，RepairBoost 以类似的方式将根节点和叶节点映射到相应的节点上，不同之处在于:(i)在映射 RDAG 的根节点时，RepairBoost 选择下载修复流量最小的节点，该节点在修复块的同一条带内不存储任何块;(ii)在映射 RDAG 的叶子顶点时，RepairBoost 选择上传修复流量最小的节点，这些节点不仅存储在被修复块的同一条带内，而且没有映射到该 RDAG 的任何顶点。

C.传输调度

在建立从 RDAG 到节点的映射以平衡整体的上传和下载修复流量后，它不一定达到修复时间的下限，因为在修复期间，每个时隙的带宽可能不会被利用。

为了进一步提高带宽利用率，RepairBoost 将传输调度建模为最大流问题。具体地，假设有 n 个节点参与全节点修复，我们可以根据丢失数据块的 RDAG 构建一个网络。该网络构建在 2n + 2 个顶点上(图 7)，一个源 s，一个通道 t，n 个发送节点{S_1,S_2,...,S_n}表示 n 个可能发送数据进行修复的节点，n 个接收节点{R_1,R_2,...,R_n}。然后建立连接:对于任意两个顶点 S_i 和 R_j(1≤i≠j≤n)，一旦 S_i 可以根据 R_j 的 RDAG 向 Rj 发送一个块，我们就建立 S_i 和 R_j 之间的连接。S_i 和 R_j 之间的每个连接都被分配了 1 的容量，这意味着我们可以从 S_i 和 R_j 一次发送一个幸存的块。这样就能够找到网络中的最大流，该最大流的容量表示在该时隙可以同时传输的数据块的数量，从而使可用的上传和下载带宽达到饱和。

建立最大流后，根据最大流选取的边进行分块。如果 S_i 有很多块要发送，就选择发送一个块，这样发送这个块可以使 Si 的父节点在下一个时隙成为 RDAG 的叶节点。这样的目标是建立最大流，同时在下一次调度中帮助增加网络的边数。这可能会增加下一次传输中最大流的容量。

一旦与最大流边相关的数据块都传输完毕，就删除 RDAG 中相应的边，并根据剩余的 RDAG 更新网络。如此重复调度，直到故障节点的所有丢失块都被修复。

D.拓展应用

多节点修复:这里有两种选择处理多节点修复。第一种方法是简单地分别修复每个故障节点，直到所有的故障节点修复成功。二是优先修复故障块较多或访问频繁的条带，以满足对系统可靠性和访问性能的要求。

异构环境:RepairBoost 也可以适应异构环境。当已知可用链路带宽时，RepairBoost 可以推导出上传和下载一个数据块的时间。它可以根据上传和下载数据块的次数(而不是同质环境下上传和下载数据块的数量)将顶点映射到节点，从而修正修复流量平衡。这可以确保各个节点的上传和下载时间几乎相同。在轮询模式下，RepairBoost 采用以下方式进行传输调度:(i)确定上传下载数据时间最少的节点(N')进行修复;(ii)根据可用带宽对 L 连接到 N'的链路集合进行排序;(iii)选择 L 中上传或下载带宽最大的链路进行数据传输。RepairBoost 重复上述步骤，直到传输所有请求的数据块。

适应网络条件:RepairBoost 还可以适应网络条件(例如，随机网络拥塞)。具体来说，RepairBoost 可以将整个全节点修复过程分解为几个子过程，每个子过程修复若干单个块。然后，RepairBoost 可以通过监控各子进程中节点的完成时间来推断网络状态，并主动调整下一个子进程的修复方案。对于子进程中耗时最长的节点，可以预测性地减少它在下一子进程中提供的上传和下载修复流量，以平衡幸存节点的修复时间。

3.3.4 小结

本文提出 RepairBoost，一种调度框架，促进各种纠删码和修复算法的全节点修复。RepairBoost 采用了一种称为 RDAG 的图抽象来描述单块修复解决方案。然后细致地将 RDAG 的修复任务分配给节点，以平衡上传和下载修复流量。RepairBoost 进一步调度块的传输，以使空闲带宽饱和。在 Amazon EC2 上的大量实验验证了 RepairBoost 的通用性、灵活性和有效性。

3.4 内存键值存储与校验日志的耦合

3.4.1 介绍

本论文出自 SC 21，研究方向是基于日志的纠删码校验更新方案的优化。文中提出了一种新的基于校验的日志机制架构 HybridPL，它创建了一种就地更新(用于数据和异或校验块)和基于日志的更新(用于剩余的校验块)的混合，从而平衡更新性能和内存成本，同时保持高效的单故障修复。将 HybridPL 实现为内存中的键值存储 LogECMem，并进一步设计了针对多种故障的高效修复方案。最后构建了 LogECMem 原型，并在不同的工作负载下进行了实验。实验结果表明，与现有的纠删码更新方案相比，LogECMem 以较低的内存开销实现了更好的更新性能，同时保持了较高的基本 I/O 和修复性能。

3.4.2 研究背景

现今，关于纠删码更新的研究已经开始关注大规模集群。在这种情况下，每个条带都很宽，因此在更新期间，一个条带上可能会有许多未改变的活跃数据块，而全条带更新为进行校验重新计算以形成新的分条，从而导致大量的网络传输。同时，作者通过实验发现全条带更新很难权衡任何更新工作负载的更新成本(以块传输为标准)和内存开销。此外，最近学术界新提出了一种宽条带概念，促使作者在大规模集群中探索就地更新和校验日志，因为这两种更新方案只基于增量更新校验块，而不管条带有多宽。而在任何更新工作负载下，就地更新总是会读取校验块而产生额外的块传输。与上述两个相比，作者发现校验日志可以在很大程度上消除更新期间的校验块读取，避免了更新期间的繁重内存开销。

因此，文中是在内存 KV 存储中引入校验日志，以降低更新和存储的开销。然而，基于磁盘的日志设备的传输速率比 dram 低得多，降低了更新和修复性能。为此如何在内存 KV 存储上部署面向磁盘的校验日志仍然具有挑战性。

3.4.3 方法介绍及结构设计

A.HybridPL 架构

本文提出 HybridPL 架构，以混合模式将就地更新和校验日志应用于内存 KV 存储。图 6 显示了带有(6,3)节点的 HybridPL。HybridPL 由多个客户端、一个代理、多个 DRAM 节点和日志节点组成。具体来说，

1)客户端与用户应用程序的接口;

2)代理作为客户端执行各种对象前端接口的请求;

3) DRAM 节点存储所有数据块和异或校验块;

4)多个日志节点存储剩余的校验块。

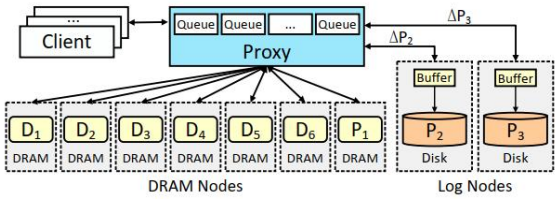

图 6.HybridPL 架构

以往对内存 KV 存储系统的负载研究中，考虑到存储对象通常较小，难以直接部署纠删码。为此，作者将多个对象组织成一个更大的固定大小单元。为了实现这一点，代理具有多个包含编码缓冲区的队列，以收集新写入的对象:固定大小的数据块(默认大小 4 KB 为)并将它们编码为

一个同等大小的校验块。此外，代理还需要维护条带的原始数据，用于管理客户端的请求(如更新)、降级读和修复操作。主要包括标识每个条带的条带 ID、将对象映射到分条 ID 并提供详细的数据块元数据的对象索引，以及记录每个分条的所有:数据块和校验块的分条索引。

使用(n,k)节点，我们可以容忍多达 n-k 个 DRAM(或 log)节点故障，即使三个 DRAM(或 log)节点与图 6 中的(6,3)节点脱机，存储在 HybridPL 架构中的对象仍然可以可用。此外，为了避免代理服务器的单点故障，与之前的研究类似，可以对代理服务器进行多次热备份，同时也可以保护元数据的可靠性，包括 Stripe 索引和对象索引，这些数据结构存储在代理服务器上，并在备份代理上进行复制。

简单来讲，HybridPL 架构通过部署数据块的就地更新来减少内存开销；通过检索同一条带剩余的 k-1 数据块和异或校验块，在 DRAM 节点内解码请求的数据块，从而保证高性能的修复单点故障；将校验增量存储到日志设备中，通过校验日志来更新除异或以外的校验块；通过批量转发增量副本实现快速更新。

B.LogECMem 设计

LogECMem 支持 4 种基本请求:写、读、降级读和删除，其中每个对象的键和值都是任意字符串。

写入:对于要写入 LogECMem 的新对象，代理首先选择一个 DRAM 节点来存储对象的键。为了实现纠删编码，代理为每个 DRAM 节点维护一个队列，队列中的每个元素都是固定长度的单位(如 4KiB)，可以通过先到先服务的方式收集对象的值，形成数据块。请注意，我们可以通过数据块中的偏移量和长度来定位对象。当所有队列中的 k 个单位至少有一个完整时，这 k 个单位就可以转换成 k 个数据块，并编码成 r 校验块。代理将相同条带的 k+r 个数据和校验块与唯一的分条 ID 关联起来。然后将异或校验块(其键来自条带 ID)分配到其中一个 DRAM 节点中，将其他 r-1 校验块分配到不同的日志节点中。特别地，这些非异或校验块不会立即存储在日志节点中。相反，它们可以临时存储在日志节点的缓冲区中，以便快速写入。

这里，将写操作的元数据组织如下。首先，该代理将写入对象的键映射到对象索引的一个元素上，其中每个元素由条带 ID、条带内数据块序号、数据块内偏移量和数据块内长度组成;其次，代理按顺序记录所有 k+r 数据/校验块，并通过条带索引记录每个数据块中所有对象的键;这样，当无法直接读取该写入对象时，可以对其进行解码。

读取:为了获取一个已存在的对象，代理选择 DRAM 节点，并通过对象的键从 LogECMem 检索它。

降级读:对于从 LogECMem 正常读取失败的对象，代理触发解码进程重新获取，以降级模式进行读操作。作者研究了瞬时的网络拥塞和永久的节点故障都会导致数据块不可用。

降级读的对象,属于不可用数据块,LogECMem 首先获得条带 ID、序列号的数据块,通过查找对象索引偏移量和长度,重建 k - 1 通过收集所有可用的数据块包含的对象的条带与条带的数据块通过条带索引 ID,以及异或校验块,从这些 k - 1 数据块(基于序列号)和一个异或校验块解码出不可用数据块,并从解码数据块中根据其偏移量和长度重新获取该对象。

删除:为了从 LogECMem 中删除一个已经存在的对象，代理可以将对象的值直接更新为 0 字节作为删除请求，但需要部署垃圾收集方法来回收这些 0 字节空间。

C.LogECMem 更新

在将一个已存在对象的旧值更改为一个新值，还需要更新包含该对象的条带的校验块。

更新对象时，代理首先从对象索引中获取对象的条带 ID、序列号、偏移量和长度。然后，代理检索对象的旧值和条带的异或校验块(它的键可以通过条带 ID 获得)通过读取操作。在这里，代理首先从新旧数据块中计算增量，使用增量和相应的系数(由序号决定)计算异或校验块的校验增量，然后通过合并旧的校验块与校验增量计算新的异或校验块。代理还将增量连同相应的系数(由序列号决定)发送给每个日志节点，每个日志节点以类似的方式计算自己的校验增量。最后，LogECMem 将新对象和异或校验块写入 DRAM 节点，并将校验差值与偏移量和长度写入 log 节点。

图 7 描述了(6,3)节点中属于 D1 的对象的更新工作流，其中 D1、D2、D3、D4、D5、P1 存储在 DRAM 节点中，P2、P3 存储在 log 节点中。注意，对于异或校验块，LogECMem 对整个块执行就地更新，但对于非异或校验块，记录基于日志文件的校验增量和对象的偏移量和长度。

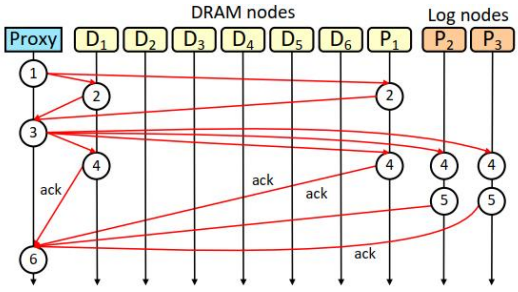

图 7.更新工作流

基于 HybridPL, LogECMem 将数据和异或校验块以

对象的形式存储在内存 KV 存储中，其他校验块和校验增量以文件的形式存储在日志节点中。在这里，LogECMem 为每个校验块及其对应的校验增量创建一个日志文件，其文件名通过其条带 ID 生成。

作者发现在每个日志节点的缓冲区中经常有多个相同条带的校验差。LogECMem 通过合并多个校验增量(称为基于合并的缓冲区日志)来改进 HybridPL 的缓冲区日志方法。通过这种方式，LogECMem 可以在日志节点更新时减少磁盘 IOs。如图 8 所示，对于这个传入的数据流 a, b, a', b', a"，HybridPL 中的校验块 a + 2b 必须存储三个校验

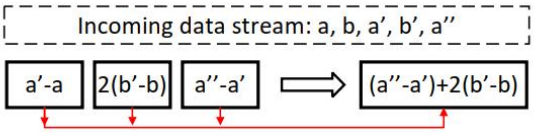

图 8.合并校验更新

增量，在 LogECMem 中可以减少到只有一个校验增量(a "-a) + 2(b' -b)。

D.LogECMem 节点修复

保留空间的校验日志：为了处理多故障块，除了异或校验块外，还需要使用更多的校验块，因此 LogECMem 需要使用日志节点来进行多故障修复，采用前人研究的一种最先进的 PL 方案——保留空间的校验日志(parity logging with reserved space, PLR)，它将校验增量保存在旧的校验块旁边，以减少计算最新校验块时的磁盘寻道。

修复分条多块故障：PLR 不能在 LogECMem 中使用，LogECMem 的目标是在写(更新)和修复方面都获得高性能。可以简单地通过合并(称为 PLR-m)来增强 PLR，它在刷写到磁盘之前立即将同一条带的校验增量合并到内存中。显然，由于内存空间有限，PLR-m 只能合并接近传入的校验增量。为了解决这个问题，作者使用连续的磁盘空间来进行惰性合并(称为校验日志合并，简称 PLM)，它首先顺序地将校验增量写入额外的连续磁盘空间，然后读取它们以合并相同条带的增量，最后将合并后的增量写入特定的校验块的保留空间。这样，PLM 可以合并比 PLR-m 更多的校验增量，从而减少更新时比 PLR 和 PLR-m 更多的磁盘 IOs。

3.4.4 小结

文章提出了 HybridPL，一种新的基于校验日志的架构，对 DRAM 中的数据和异或校验块进行就地更新，并对日志中剩余的校验块进行基于日志的更新，这样可以很好地平衡更新和单次故障修复的内存成本、性能。将 HybridPL 原型为基于 Memcached 的 LogECMem，并设计了两种提高多故障修复性能的方案(PLM 和 log-assist)。基于云平台的实验结果表明，LogECMem 在基本 I/O、更新和修复方面具有较高的效率，且具有较低的内存开销。

4. 总结

现有的针对纠删码存储系统内的更新和修复流量优化的方法，多是从某个角度或是某种条件下出发进行研究得到的，虽然从测试结果来看，都有着出色的表现。但在实际情况中，仍然有着较多的影响因素，方法的应用还有待持续的测试与改进。

RackCU 提出了收集器与选择校验更新来减少机架间的流量传输；PDL 通过提出一种新的数据布局加速故障恢复和减少重构流量；CRaft 为支持纠删码的 Raft，灵活使用节点片段复制和全条目复制从而降低网络开销；RepairBoost 利用修复有向无环图、最大流问题来解决全节点修复；LogECMem 采用键值存储与校验日志耦合的方法，减少了更新和修复时的磁盘 IO。以上几种方法都能不同程度的提升系统的数据吞吐量、读写性能和修复速率，可见学术界目前面对纠删码系统的性能优化已经有了一些比较成熟的方案。

但是，如何在保持整个存储系统的可靠性前提下，既能够有效降低数据更新时产生的更新流量，又能在节点修复时采用较少的流量进行快速的修复，LogECMem 的研究使其有了一个好的开端，也还需要后人不断的去整合、测试和研究，这也是未来学术界的研究方向之一。

5. 参考文献

[1] Guowen Gong, Zhirong Shen, Suzhen Wu, et al. Optimal Rack-Coordinated Updates in Erasure-Coded Data Centers//Proceedings of the IEEE INFOCOM 2021 - IEEE Conference on Computer Communications.Vancouver, BC, Canada,May 10-13，2021:1-10.

[2] Liangliang Xu, Min Lv, Zhipeng Li, et al. PDL: A Data Layout towards Fast Failure Recovery for Erasure-coded Distributed Storage Systems //Proceedings of the the IEEE INFOCOM 2020- IEEE Conference on Computer Communications.Toronto, ON, Canada,July 06-09 2020: 736-745.

[3] Zizhong Wang, Tongliang Li, Haixia Wang, et al. 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), Santa Clara, CA, USA,February 25-27, 2020 :297-308.

[4] Shiyao Lin, Guowen Gong, and Zhirong Shen, et al. Boosting Full-Node Repair in Erasure-Coded Storage//Proceedings of the 2021 USENIX Annual Technical Conference.July 14–16, 2021:641-655.

[5] Liangfeng Cheng,Yuchong Hu,Zhaokang Ke,et al.LogECMem: Coupling Erasure-Coded In-Memory Key-Value Stores with Parity Logging//Proceedings of the SC 2021, St. Louis, MO, USA ,November 14–19, 2021: 89.