中国科学: 信息科学 2020 年 第 50 卷 第 12 期: 1834–1849
SCIENTIA SINICA Informationis
论文

# 一种近似最优的分布式存储系统磁盘修复算法

孙婧[1*], 梁松涛[2], 路新江[3]

1. 华东政法大学互联网 + 法律大数据平台, 上海 201620
2. 七牛信息技术有限公司富媒体部, 上海 200433
3. 百度研究院商业智能实验室, 北京 100085
* 通信作者. E-mail: jingsuncs@126.com

收稿日期: 2019–01–04; 修回日期: 2019–04–12; 接受日期: 2019–08–13; 网络出版日期: 2020–11–20

国家重点研发项目 (批准号: 2018YFC0830900, 2018YFC0830903) 资助

**摘要** 如何提升分布式存储系统中磁盘修复的速度, 一直是磁盘修复问题中的难点. 优化的途径有两种: 一种是通过对解码算法的优化, 减少修盘数据的传输量. 另外一种方法是通过对修盘过程中数据流的调度, 最大化地利用节点的计算能力、传输能力, 进而加速修盘进程. 本文从数据流的调度出发, 根据数据流图和拓扑结构, 计算出了节点的近似最优的修盘数据比例, 并依照此比例, 设计了分布式存储系统下的近似最优修盘调度算法 (NOPT). 对于主流的两种 Reed-Solomon (RS) 编码方式, 本文做了等价性证明, 并给出了编码转换矩阵. 通过大量实验仿真可以看出, 在预知系统拓扑的前提下, 可以显著地减少通过交换机的流量, 进而缩短修盘的时间.

**关键词** Reed-Solomon 码, 磁盘修复技术, 分布式存储系统

## 1 引言

根据应用场景的不同, 分布式系统分为 3 类: 对象存储系统、块存储系统和文件存储系统. 对象存储系统常作为公有云的一部分为用户提供服务, 如阿里云的 OSS[1)]、Amazon 的 S3[2)]、腾讯的 COS[3)]和七牛的 KODO[4)]等对象存储为海量的图片、视频提供了方便的存储和访问机制, 通过一些约定的接口或规范, 用户可以对数据进行读写. 对象存储一般不支持随机读写, 数据都是以 append 方式写入. 块存储则常用在可以随机读写的场景下, 例如数据库场景、网络云盘等. 甚至于, 在块存储之上, 也可以提供对象存储或文件存储的功能, 如开源分布式系统 ceph (既支持块存储, 也支持对象存储). 非开源

1) Aliyun:oss. https://cn.aliyun.com/product/oss.
2) Amazon:s3. https://aws.amazon.com/cn/s3/.
3) Tencent:cos. https://cloud.tencent.com/product/cos.
4) Qiniu:kodo. https://www.qiniu.com/products/kodo.

**引用格式:** 孙婧, 梁松涛, 路新江. 一种近似最优的分布式存储系统磁盘修复算法. 中国科学: 信息科学, 2020, 50: 1834–1849, doi: 10.1360/N112019-00004
Sun J, Liang S T, Lu X J. An approximately optimal disk repair algorithm for distributed storage systems (in Chinese). Sci Sin Inform, 2020, 50: 1834–1849, doi: 10.1360/N112019-00004

 

块存储系统一般都作为网络云盘使用. 在块存储的基础上, 可以对接多种类型的协议, 如网络文件系统 (nfs, samba, ftp 等). 分布式文件系统是最常用的存储系统. 因为直接实现了文件语义, 所以分布文件的系统的应用场景比较广泛, 其中一个, 就是作为数据源, 为云计算提供支撑. 常见的分布式文件系统有 Google File System [1], Hadoop Distributed File System [2] 等.

而无论哪种类型的分布式存储系统, 因为硬件故障或软件的原因导致某些数据丢失是一种必然会发生的事情. 所以保证数据可靠性是实现分布式系统重点考虑的一个问题. 常见的方法有两种, 副本和纠删码. 相比副本的方式, 纠删码在成本和数据可靠性上占据优势. 最常用的纠删码是 RS 码. 作为底层的存储技术, 条带化技术是基于纠删码的存储系统必然用的一种设计方式. 而当磁盘发生故障的时候, 如何使用条带化技术恢复故障下数据, 则成为一个研究的热点. 当单个磁盘发生故障后, 减少磁盘修复时间的方法主要分两类: 一类是减少磁盘的修复带宽 [3~7], 另一类是减少修复过程中, 跨交换机的数据传输量. 第一类主要通过实现新的纠删码的编解码算法, 且修复过程中对同一节点数据的再编码, 达到减少交换机之间的传输量 [8~10] 的目的. Dimakis 等 [3] 通过对修复过程中数据流的分析, 使用最大流最小割原理, 证明了通过网络编码的方法可以实现磁盘的最优带宽修复, 提出了再生码 (regenerating code) 的概念. 我们用 $[n,k,d]$ 表示一种再生码, 且满足 (1) $n$ 个节点中的任意 $k$ 个节点都可以重构所有的数据; (2) 任意一个故障节点都可以由其他 $d$ 个节点完成修复. Dimakis 在文献 [3] 中虽然证明了磁盘修复带宽的理论边界, 但并没有给出确定性构造方法. Rashmi 等 [11] 利用矩阵积算法构造了能够满足所有 $[n,k,d\geqslant 2k-2]$ 参数的最小带宽修复的再生码. 矩阵积算法在修复带宽上虽然达到了理论界的最优值, 但在存储系统中, 却因需要将一个数据块切分成指数级别的分片而难以实现. 文献 [12] 则通过节点分组, 设计了局部重构码 (local reconstruction code, LRC). LRC 的校验节点分为两类, 一类属于局部分组, 一类属于全局分组. 这种设计可以显著减少磁盘修复时的带宽, 并且易于在系统中实现. 该编码已经应用到了 Windows Azure 存储系统中 [13]. 文献 [14] 提出了一种满足任意 $[n,k,r,t]$ 参数下的 LRC 编码构造方法, 其中 $r$ 是编码的局部性, $t$ 是编码的可用性. 文献 [15] 则提出了一种新的基于局部重构的并发再生码, 这种方法具有更高的编解码吞吐量. 以上的编码都没有考虑系统的拓扑, 比如磁盘修复的时候数据如何经过交换机, 影响磁盘修复的瓶颈点等问题. 交换机的带宽在修盘过程中总是稀缺的资源, 流量常常称为瓶颈点, 尤其是核心交换机, 在修复过程中承担着比较大的压力. 若能够减少交换机之间的数据传输, 则修盘的时间也将得到优化. Shen 等 [16] 设计的基于分段解码的 CAR 算法, 在修复过程中, 将同一个交换机内的同一条带的数据按照分段解码的理论做聚合编码, 而后将编码后的数据块替代未编码前的多个数据块, 通过上层的交换机进行传输, 这样可以显著地减少交换机的流量. 文献 [17] 则设计了双再生码 (double regenerating codes, DRC), 将编码构造与存储系统的拓扑结合, 降低磁盘修复的时间.

我们从上述的第 2 个角度出发, 通过修盘任务的调度, 减少交换机传输带宽的方式来优化修盘的时间. 本文的贡献总结如下:

(1) 基于三层拓扑结构, 我们通过数据流的分析, 给出了最优修复时间求解方法. 并且根据系统存储节点的数据量, 给出了近似最优的自修复比例计算方式.

(2) 我们拓展了 RS 分段解码算法的理论, 给出了基于生成多项式和基于生成矩阵两种 RS 码的等价性, 并且给出了具体的编码转换方法, 分段解码可以减少通过交换机的数据量.

(3) 基于近似最优修复比例的计算思路, 我们设计了新的近似最优修盘调度算法 (NOPT), 通过修盘实验可以看出, NOPT 可以显著地减少修盘过程中经过交换机的数据量, 进而达到优化修盘时间的目的.

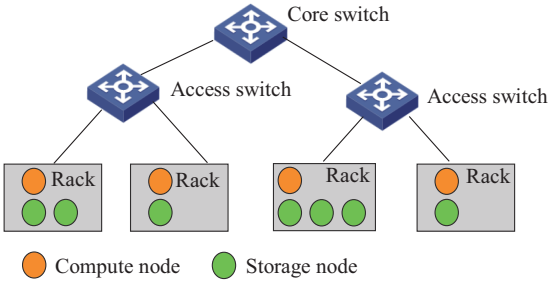


图 1 (网络版彩图) 存储系统模型

**Figure 1** (Color online) Storage system model

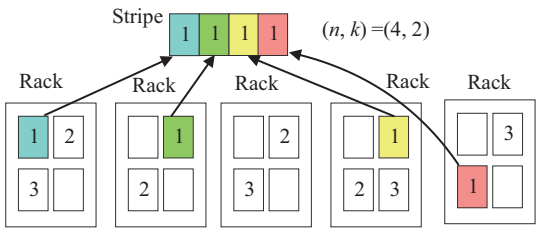


图 2 (网络版彩图) 条带存储模型

**Figure 2** (Color online) Stripe storage model

## 2 系统模型

分布式存储系统中, 与用户数据存储相关的存储方式有两种: 一种是中心化的 (多数存储系统仍然采用中心化的方式), 一种是去中心化的 (如 Ceph[5]存储系统通过 crush 算法进行数据存储位置的映射). 去中心化有众多的优点, 如不再需要额外的存储空间和相应的机制去记录与文件存储相关的元数据信息; IO 访问的时候, 不再需要与文件存储位置相关的元数据的查询; 集群的规模不再受限于 CPU、内存和磁盘的约束, 可以无限地扩展. 去中心化并非没有坏处, 尤其是一致性 Hash 算法在扩容时带来的局部数据的迁移问题, 仍然会给系统带来较大的网络及磁盘访问的开销. 在系统整体负载比较高的时候, 尤其考验系统设计的架构能力和异常处理能力. 所以如 EMC, NetApp 等大部分商用的存储系统仍然采用中心化的方式, 记录文件的元数据信息. 本文的磁盘修复模型建立在中心化的存储系统之上. 这里先介绍与修盘流程相关的对象存储模型.

### 2.1 分布式存储模型

不同的分布式存储系统的存储模型各不相同. 忽略掉具体的实现细节, 分布式存储系统的模型可以抽象为图 1. 存储系统采用 3 层的结构: 最顶层为核心层交换机, 控制着跨交换机的数据通信流. 中间层为接入层交换机, 与存储系统具有某种功能的节点直连. 第 3 层为存储系统的计算节点、存储节点和其他节点 (监控节点等). 在某些存储系统中, 会将计算功能和存储功能合并为一个节点, 比如 ceph 的 OSD 节点, 可以作为我们系统模型的一种特例. 存储节点属于 IO 密集型, 主要负责对磁盘进行读写操作. 而计算节点则属于 CPU 密集型, 负责数据的迁移、修复、回收等任务的处理和 RS 的编解码. 其他节点比如监控节点, 控制存储系统的拓扑结构、监控系统状态、控制修盘流程甚至于控制迁移流程等功能. 存储系统的设计理念不同, 监控节点的功能也会不一样. 所以, 我们忽略掉监控节点的细节.

### 2.2 基于 RS 码的条带存储模型

RS 码是一种广泛应用在存储系统的编码. RS 码的使用, 对存储系统的成本、可靠性等都带来了显著的提升, 已经成为了分布式存储系统中必然使用的一种技术. 当然, 系统需要解决使用 RS 码带来的 CPU 开销高、系统元数据设计复杂、一致性等问题. 基于条带的存储技术同样已经成为了分布式存储系统的模板. 我们忽略元数据设计的细节, 单独地抽象出条带存储的概念, 如图 2 所示.

5) ceph. https://ceph.com.

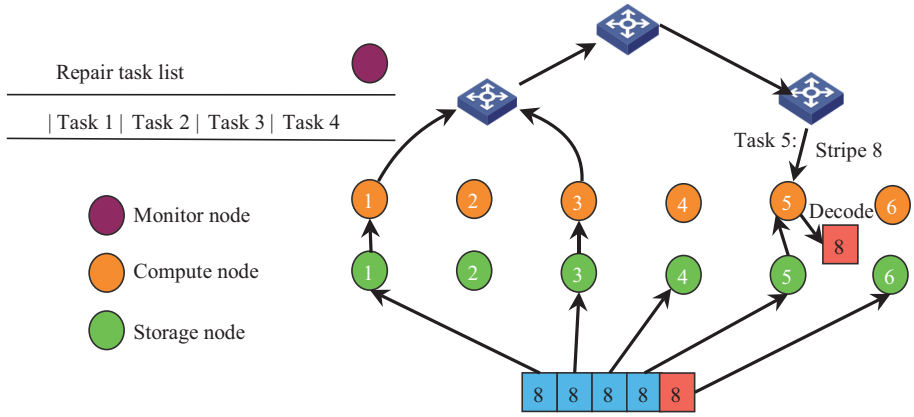


**图 3 (网络版彩图) 故障盘修复模型**

**Figure 3** (Color online) Broken disk repair model

图 2 给出了以条带为单元的存储方式, 数字编号一样的属于同一个条带. 如图 2 中的 $(4,2)$RS 码, 同一个条带的各个分片存储在不同的 rack 中, 见编号为 1 的数据分片. 基于 RS 的特性, 系统可以抵抗 rack 级别的故障, 即当 $n-k$ 个 rack 的损坏或丢失, 数据不会丢失. 这里的 rack 是一个虚拟的概念, 可以为一个磁盘、一台主机、一个机架, 或是数据中心. 如何定义 rack, 与分布式存储系统设计的目标相关. 分片的大小根据系统架构的设计而不同, 一般是 KB 或 MB 级别. 如 HDFS 系统默认的数据块大小为 64 MB.

### 2.3 磁盘修复模型

我们所讨论的磁盘修复过程基于前面描述的存储模型. 图 3 给出了一个故障盘修复的数据流图. 图 3 采用 $(5,4)$ 的编码方式, 共有 6 个存储节点, 其中节点 6 发生了故障. 监控节点获取坏盘信息后, 解析坏盘的条带信息, 并生成对应的修盘任务. 计算节点主动或者被动地与监控节点进行通信, 尝试做条带级别的修复. 如图 3 所示, 计算节点 5 得到了修盘的任务, 而后解析条带中没有损坏的块的信息, 从存储节点 1, 3, 4, 5 中读取对应的数据块, 通过解码, 就可以计算出存储节点 6 中的损坏的数据块. 解码完成后, 会将数据存储一个新的位置, 可能是热备盘, 可能存储节点 6 新换的盘, 也可能是通过分配算法, 分配的系统中某个还有空闲容量的磁盘, 不同的存储系统存储的方式不尽相同. 所以我们后面的分析中, 只进行到解码完成, 不再考虑后继的存储问题.

### 2.4 符号定义

如果没有特别说明, 本文后面用到的符号定义如表 1 所示.

## 3 设计实现

### 3.1 原理

在分布式存储系统的生产环境中, 如果数据中心达到了 PB 甚至 EB 的规模, 会需要更多的接入层交换连接计算节点和存储节点. 我们至顶向下地分析存储的拓扑结构, 以期望得出磁盘的修复会存在什么样的性质.

**表 1　符号定义**

**Table 1**　Symbol definition

| Symbol | Definition |
|---|---|
| $S_i$ | The storage node $i$ |
| $n_t$ | The number of access switches |
| $n_{t_s}$ | The number of switches associated with storage nodes participating in one repair process |
| $n_{t_c}$ | The number of switches associated with computational nodes participating in one repair process |
| $\hat{n}_c$ | The number of computational nodes |
| $n_c$ | The number of computational nodes participating in one repair process |
| $n_s$ | The number of storage nodes participating in one repair process |
| $n_{s_i}$ | The number of storage nodes connected with access switch $i$ |
| $n_{c_i}$ | The number of computational nodes connected with access switch $i$ |
| $W$ | The amount of data waiting for repairing |
| $d_{s_i}$ | The amount of data related to broken disk for storage node $i$ |
| $V_s$ | Disk read rate |
| $V_{l_i}$ | Data transmission rate from the access switch $i$ to the core switch |
| $V_{m_i}$ | Data transmission rate from the core switch to the access switch $i$ |
| $V_h$ | The maximum throughput of the core switch |
| $V_{t_i}$ | The maximum throughput of the access switch $i$ |

**定理1** (核心交换机的最小修盘时间)　设 $D$ 为修盘过程中经过核心交换机的数据流量. 令 $V_{l_i}$ 为第 $i$ 个为接入层交换机向核心层交换机的传输速率, 则当第 $i$ 个交换机传输的数据量 $D_i$ 为 $\frac{D}{\sum_{j=1}^{n} V_{l_j}} V_{l_i}$ 的时候, 修盘时间能够达到最小值, 且时间为 $\frac{D}{\sum_{j=1}^{n} V_{l_j}}$.

**证明**　用反证法证明. 设修盘时间 $t < \frac{D}{\sum_{j=1}^{n} V_{l_j}}$, 则经过核心交换机的上行流量为 $V_{l_1}t + V_{l_2}t + \cdots + V_{l_n}t < D$ 与传输数据量为 $D$ 矛盾. 而当 $D_i = \frac{D}{\sum_{j=1}^{n} V_{l_j}} V_{l_i}$ 时, 每个交换机参与修复时间均为 $D_i/V_{l_i} = \frac{D}{\sum_{j=1}^{n} V_{l_j}}$. 因整体的修盘时间等于最慢的节点修复时间, 所以最小修盘时间为 $\frac{D}{\sum_{j=1}^{n} V_{l_j}}$.

定理 1 反映了在异构环境下, 每个接入层交换机传输修盘数据的期望. 在生产环境中, 精确地预先安排每个接入层交换机修盘的数据量是一件非常困难的事情. 因受磁盘分配算法的影响, 每个接入层下的存储节点与坏盘在同一个条带的数据量各不相同. 例如, $S_i$ 节点可能有 200 GB 的数据与坏盘相关, 而 $S_j$ 节点却只有 16 MB 的数据与坏盘相关. 这种偏差之下, 让 $S_j$ 节点之上的交换机传输大于 16 MB 的数据是不符合实际的. 在我们设计修盘算法的过程中, 可以将定理 1 作为数据传输量的一个期望.

事实上, 我们可以通过存储节点和计算节点的限流算法来控制接入层交换机发送到核心交换机的速度. 在生产环境下, 当交换机快要达到最大的吞吐量的时候, 就会存在网络抖动, 并且有数据包丢失的情况. 所以, 我们一般只会分配一定比例的带宽用作某种业务. 比如, 为了给用户提供稳定的服务质量, 存储系统预留 1/3 的带宽, 防止高峰期的突增访问量带来存储系统服务质量的降低. 因各个存储节点的访问请求不同, 所以也会使各个存储节点的数据传输率不同.

这里, 我们分析一下单个业务层交换机内的修盘数据流, 见图 4. 设一个业务层交换机有两个计算节点 $i$, $j$ (每个计算节点分别对应一个存储节点, 这么设计的原理是因为存储节点主要使用磁盘空间读写数据, 对内存和 CPU 的要求并不是很高. 而计算节点则属于 CPU 密集型, 需要做编解码, 对

内存和 CPU 要求很高. 因此, 在生产环境中, 一台服务器上会部署一个存储节点和一个计算节点). 两个计算节点中, 与坏盘相关的数据量分别为 $d_{s_i}$ 和 $d_{s_j}$, 设系统使用的是 $(n,k)$ 的 RS 码, 令 $\alpha_i$ 和 $\alpha_j$ 分别为两个计算节点自解码 (计算节点从同服务器中读取一定的数据量, 然后从其他的存储节点读取 $(k-1)$ 倍于自身的数据量来进行解码, 原理参见 RS 码) 的比例, 则由图 4 可以看出, 自解码的数据量为 $d_{s_i}\alpha_i$ 和 $d_{s_j}\alpha_j$, 而为了解码, 需要从其他节点下载的数据量为 $(k-1)d_{s_i}\alpha_i$ 和 $(k-1)d_{s_j}\alpha_j$. 而发送出的, 需要在别的节点解码的数据量为 $d_{s_i}(1-\alpha_i)$ 和 $ds_j(1-\alpha_j)$.

根据修盘的数据流, 我们可以得到下列的定理.

**定理2** (最小修盘时间) 设 $\alpha_i$ 为第 $i$ 个存储节点参与自解码的磁盘数据量的比例, $i\in\{1,2,\ldots,n_s\}$, $0\leqslant\alpha_{n_s}\leqslant 1$. 则寻找最小修盘时间的问题, 等同于求解 $\min\ f(\alpha_1,\alpha_2,\ldots,\alpha_{n_s})$ 的最优化问题. 其中

$$f(\alpha_1,\alpha_2,\ldots,\alpha_{n_s})=\{\forall i\in\{1,2,\ldots,n_s\}|\max\{g(i,\alpha_i),h(i,\alpha_i),l\}\},\tag{1}$$

$$g(i,\alpha_i)=\frac{(k-1)d_{s_i}\alpha_i}{V_{c_i}},\tag{2}$$

$$h(i,\alpha_i)=\frac{d_{s_i}(1-\alpha_i)}{V_{s_i}},\tag{3}$$

$$l=\frac{\sum_{i=1}^{n_s}d_{s_i}(1-k\alpha_i)}{V_{c_i}(n_c-n_s)},\tag{4}$$

$$V_{c_i}=\min\left\{V_{t_i},\frac{V_h}{n_{t_c}}\right\}\frac{n_{t_s}}{(n_{t_s}+n_{t_c})n_{c_i}},\tag{5}$$

$$V_{s_i}=\min\left\{V_{t_i},\frac{V_h}{n_{t_s}}\right\}\frac{n_{t_c}}{(n_{t_s}+n_{t_c})n_{s_i}}.\tag{6}$$

**证明** 针对任意的计算节点 $i$, 参与自解码的数据量为 $d_{s_i}\alpha_i$, 则需要通过交换机传输, 发送给其他计算节点的数据量为 $d_{s_i}(1-\alpha_i)$. 通过 RS 自解码, 且经交换机传输的数据量为 $(k-1)d_{s_i}\alpha_i$. 耗时分别为 $\frac{d_{s_i}\alpha_i}{V_s}$, $\frac{d_{s_i}(1-\alpha_i)}{V_{s_i}}$ 和 $\frac{(k-1)d_{s_i}\alpha_i}{V_{c_i}}$. 显然, 3 个值之中的最大值即为修盘完成的时间. 这里, 我们忽略了解码的时间和解码后将数据写回的时间. 因为从修盘的角度来看, 经过 intel 的 SIMD, AVX 和 AVX2 等汇编指令, RS 编解码的速度非常高. 相比于传输带宽, 编解码的速度已经不再是瓶颈点. 而数据写回与分布式存储系统的机制强相关, 且读取的数据量是写回数据量的 $k-1$ 倍, 所以我们忽略掉这两个因素.

从解码的角度, 因为自解码需要同时下载 $d_{s_i}\alpha_i$ 和 $(k-1)d_{s_i}\alpha_i$ 的数据量才可以进行解码, 而在生产环境下 SATA 盘的速度为 100 MB/s 左右, SSD 的速度为 350 MB/s 左右, 基于机架结构的不同, 一台服务器会有多个盘位, 如 4U 的机器一般都有 36 盘位. 考虑到不同磁盘读取的并发, 所以 $V_s$ 的上限值非常大, 而 $(k-1)d_{s_i}\alpha_i$ 这一部分数据会受到交换机流量的限制. 所以, 不失一般性, 自解码的时间可以用 $\frac{(k-1)d_{s_i}\alpha_i}{V_{c_i}}$ 来计算. 从而可以得到式 (2) 和 (3).

我们再从图 5 来分析 $V_{s_i}$ 和 $V_{c_i}$ 的取值. 对任意的节点 $i$ 来说, 根据守恒定律, 都应该满足 $V_{\text{in}}=V_{\text{out}}$, 其中 $V_{\text{in}}$, $V_{\text{out}}$ 分别为数据进出节点 $i$ 的速率. 如图 5 所示, 应该满足下面的关系:

$$\begin{cases}V_{s_i}+V_{s_j}=V_l,\\ V_{c_i}+V_{c_j}=V_m,\\ V_{s_i}+V_{s_j}+V_m\leqslant V_{t_i},\\ V_{c_i}+V_{c_j}+V_l\leqslant V_h.\end{cases}\tag{7}$$

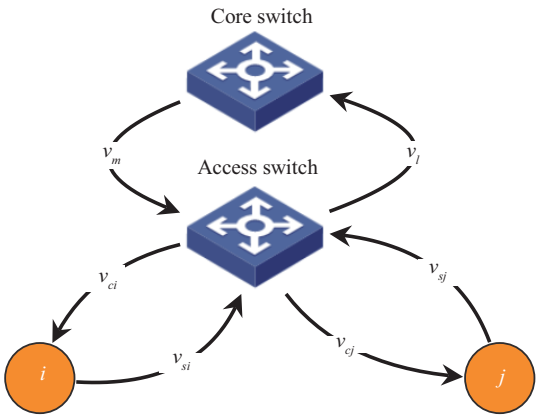


图 4 (网络版彩图) 修盘数据流图

**Figure 4** (Color online) Repair data flow

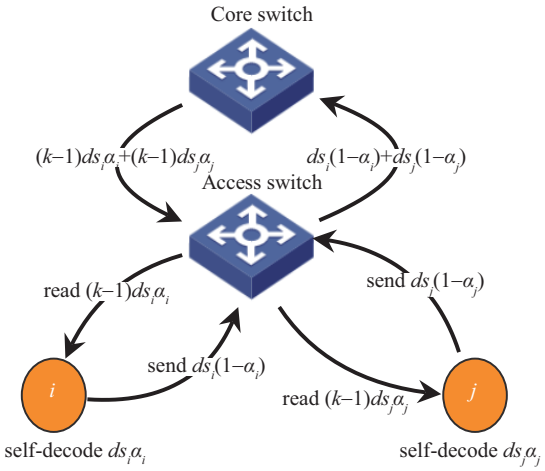


图 5 (网络版彩图) 修盘吞吐量图

**Figure 5** (Color online) Repair throughput

将前 2 个等式分别代入到后 2 个等式中, 可得 $V_l+V_m \leqslant V_{t_i}$ 且 $V_l+V_m \leqslant V_h$, 即 $V_l+V_m \leqslant \min\{V_{t_i}, V_h\}$. 根据流量守恒定律, $n_{t_s}V_l = n_{t_c}V_m$, 其中 $n_{t_s}$ 为修复过程中, 存储节点关联的交换机的总数 (发送数据到核心交换机), $n_{t_c}$ 为计算节点关联的交换机的总数 (核心交换机发出数据到计算节点). 则 $V_l = \frac{n_{t_c}}{n_{t_s}}V_m$. 进而可得 $V_l \leqslant \min\{V_{t_i}, V_h\}\frac{n_{t_c}}{n_{t_s}+n_{t_c}}$, $V_m \leqslant \min\{V_{t_i}, V_h\}\frac{n_{t_s}}{n_{t_s}+n_{t_c}}$.

我们将图 5 的分析推广到整个拓扑中, 设当前的计算节点为 $C_i$, 则 $V_l = \min\{V_{t_i}, \frac{V_h}{n_{t_s}}\}\frac{n_{t_c}}{n_{t_s}+n_{t_c}}$, $V_m = \min\{V_{t_i}, \frac{V_h}{n_{t_c}}\}\frac{n_{t_s}}{n_{t_s}+n_{t_c}}$. 因在存储系统中, 限流算法的申请流量配额的方式是小额申请 (KB 级别), 多次迭代的方式, 所以可以认为同一个接入层交换机下, $d_{s_i} = d_{s_j}$. 进而可得 $V_{s_i} = V_l/n_{t_s} = \min\{V_{t_i}, \frac{V_h}{n_{t_s}}\}\frac{n_{t_c}}{(n_{t_s}+n_{t_c})n_{s_i}}$, $V_{c_i} = V_m/n_{c_s} = \min\{V_{t_i}, \frac{V_h}{n_{t_c}}\}\frac{n_{t_s}}{(n_{t_s}+n_{t_c})n_{c_i}}$. 则式 (5) 和 (4) 成立.

我们再分析整个网络拓扑, 对于存储节点 $i$, 数据分为两个部分, 一部分做自修复, 一部分发送其他业务层交换机. 而发送其他业务层交换机的数据也分两个部分: 第一部分是, $i$ 节点和该交换机下的某个存储节点 $j$ 属于同一个条带的数据. 计算节点 $j$ 修盘时, 从存储节点 $j$ 中读取的数据不需要消耗核心交换机的流量. 所以, 自修复读取的数据总量为 $k\sum_{i=1}^{n_s} d_{s_i}\alpha_i$. 而非自修复的数据总量 (这一部分都要走核心交换机) 为 $\sum_{i=1}^{n_s} d_{s_i} - k\sum_{i=1}^{n_s} d_{s_i}\alpha_i = \sum_{i=1}^{n_s} d_{s_i}(1-k\alpha_i)$. 完成这一部分数据修复的计算节点的个数为 $n_c - n_s$ (每个存储节点都有自己唯一的计算节点做自修复), 根据定理 1 可知, 最短的修复时间为 $\frac{\sum_{i=1}^{n_s} d_{s_i}(1-k\alpha_i)}{V_{c_i}(n_c-n_s)}$. 由此, 我们完成了定理 2 的证明.

需要指出的是, $V_{s_i}$ 为计算节点发出非自修复数据的速率, $V_{c_i}$ 为计算节点接收自修复数据的速率. 当得知坏盘的 ID 时, 条带修复的策略已经确定, 参与修复的计算节点 $n_c$ 也指定的时候, $V_{s_i}$ 和 $V_{s_i}$ 为固定值. 从定理 2 可知, 如果我们能够找到一组 $(\alpha_1, \alpha_2, \ldots, \alpha_{n_s})$, 使函数 $f(\alpha_1, \alpha_2, \ldots, \alpha_{n_s})$ 的值最小, 则我们就可以寻找到最优的解码方案. 然而, 解决上述最优化的问题非常具有挑战性. 但从 $g$, $h$ 和 $l$ 这三个函数上, 我们可以通过算法, 求出一些较优方案, 使得在相同网络拓扑和性能参数的情况下, 修盘时间尽可能的短.

下面我们从自修复和非自修复的数据量上, 分析磁盘修复时间的最值问题.

**定理3** (自修复数据量评估定理) 设 $W$ 为坏盘的有效数据量 (这里, 有效指的是已经被写入的用户数据量, 而不是磁盘的大小), $\beta$ 为参与自修复的数据量比例. 系统采用的编码方案为 $(n, k)$ 的 RS

码, 则最优的修盘时间为

$$R(\beta, n_c) = \min\left\{\frac{(k-1)W\beta}{\sum_{i=1}^{n_s} V_{s_i}}, \frac{kW(1-\beta)}{\sum_{i=1}^{n_c} V_{c_i}}\right\}, \tag{8}$$

其中 $V_{s_i}$ 和 $V_{c_i}$ 的求解见式 (6) 和 (5).

**证明** 因系统使用的是 $(n,k)$ 的 RS 码, 所以参与修复的数据总量为 $kW$. 而参与自修复的数据量为 $kW\beta$, 需要经过交换机的数据量为 $(k-1)W\beta$. 非自修复的数据量为 $kW(1-\beta)$, 需要经过核心交换机的数据量为 $kW(1-\beta)$. 因参与自修复的计算节点个数为 $n_s$ (每个存储节点有唯一的计算节点进行自修复), 参与非自修复的计算节点个数为 $n_c - n_s$. 对自修复的节点, 由定理 1 可知, 对自修复节点, 当第 $i$ 的计算节点进行修复的数据量为 $\frac{(k-1)W\beta}{\sum_{i=1}^{n_s} V_{s_i}} V_{s_i}$ 时, 修盘能够达到最优值. 对非自修复的节点, 当第 $i$ 个计算节点进行修复的数据量为 $\frac{kW(1-\beta)}{\sum_{i=1}^{n_c} V_{c_i}} V_{c_i}$ 时, 修盘能够达到最优值. 自修复节点的修盘时间为 $\frac{(k-1)W\beta}{\sum_{i=1}^{n_s} V_{s_i}}$, 非自修复节点的修盘时间为 $\frac{kW(1-\beta)}{\sum_{i=1}^{n_c} V_{c_i}}$. 进而可得, 最小修盘时间为 $\min\{\frac{(k-1)W\beta}{\sum_{i=1}^{n_s} V_{s_i}}, \frac{kW(1-\beta)}{\sum_{i=1}^{n_c} V_{c_i}}\}$.

根据定理 3, 我们可以找到比较优化的 $\beta$ 和 $n_c$, 方法见算法实现部分算法 1. 我们这里假设已经得到了这两个参数, 然后我们就可以通过这两个参数来获取比较优的 $(\alpha_1, \alpha_2, \ldots, \alpha_{n_s})$. 因自修复的数据量为 $\sum_{i=1}^{n_s} d_{s_i}$, 且等于 $W\beta$. 根据定理 1, 我们可以将自修复的数据量分配为

$$\left(\frac{W\beta V_{c_1}}{\sum_{i=1}^{n_c} V_{c_1}}, \frac{W\beta V_{c_2}}{\sum_{i=1}^{n_c} V_{c_2}}, \ldots, \frac{W\beta V_{c_{n_s}}}{\sum_{i=1}^{n_c} V_{c_{n_s}}}\right).$$

这样可以使修盘的时间达到最优的值. 但在实际场景中, 可能存在节点 $i$ 与坏盘相关的数据比上面分配值要少, 则该节点的所有数据都将参与自修复, 即 $\alpha_i = 1$. 由此, 我们可以得到以下引理.

**引理1** $(\alpha_1, \alpha_2, \ldots, \alpha_{n_s}) = (y(i=1), y(i=2), \ldots, y(i=n_s))$ 是修盘数据分配方案的一组近似最优解, 其中

$$\alpha_i = y(i) = \begin{cases} 1, & \dfrac{W\beta V_{c_i}}{\sum_{i=1}^{n_c} V_{c_i}} >= d_{s_i}, \\ \dfrac{W\beta V_{c_i}}{(\sum_{i=1}^{n_c} V_{c_i}) d_{s_i}}, & \dfrac{W\beta V_{c_i}}{\sum_{i=1}^{n_c} V_{c_i}} < d_{s_i}. \end{cases} \tag{9}$$

### 3.2 RS 码的分段解码算法及等价性理论

通过对修复过程中经过交换机的数据量的调度, 可以优化修盘时间. 而 RS 码的分段解码算法同样可以减少修复过程中的数据传输量, 编解码实现见 CAR 算法 [16]. 分布式系统中常用的 RS 码编解码的实现分两种: 一种是基于生成矩阵, 一种是基于生成多项式. CAR 算法采用基于生成矩阵的 RS 分段解码算法. 下面我们通过对生成多项式的 RS 码的分析, 指出其无法使用分段解码的原因, 并且通过等价性的证明, 推导出与任一 RS 生成多项式等价的生成矩阵, 从而可以间接地采用分段解码的算法减少修盘过程中的数据传输量.

RS 码基于生成多项式的解码算法的步骤可以总结如下: (1) 计算伴随式多项式 (syndromes polynomial). 伴随式多项式可以快速地判断接收到的编码信息是否存在错误. (2) 构造错误定位多项式 (error locator polynomial). 错误定位多项式用来定位哪些位置的数据出现了错误. (3) 根据错误定位多项式的根确定错误的位置. 比如, Chien [18] 搜索的方法. (4) 纠错. 用 Forney [19] 算法纠正错误的信息, 并从错误的信息位减去该值.

标准的 RS 解码过程中, 可以不用知道错误位置及出错的个数. 通过上述步骤 2 对错误定位多项式的构造, 并且在步骤 3 对错误定位多项式求解后, 就可以确定错误的位置及数目. 最后, 通过步骤 4

的 Forney 算法求出出错的信息位, 并且完成纠错. 而在分布式存储系统中有广泛的错误鉴别机制, 能够确定错误的位置. 事实上, 我们无法通过标准的解码算法在交换机层面节省带宽, 理论分析如下.

设 $\alpha$ 有限域 GF($p^r$) 的本源元, RS 码的传输多项式 $s(x)$ 和生成多项式定义 $g(x)$ 分别定义为

$$s(x)=\sum_{i=0}^{n-1}c_ix^i,\quad g(x)=\prod_{j=1}^{n-k}(x-\alpha_i),\tag{10}$$

其中传输多项式 $s(x)$ 的前 $k$ 项的系数为数据位, 而后面的 $n-k$ 项为校验位. 根据生成多项式的定义, $s(x)$ 能够被 $g(x)$ 整数, 则可以得到 $s(\alpha^i)=0$, 其中 $i=\{1,2,\ldots,n-k\}$.

假设某些存储节点的磁盘出现坏盘, 则存储的数据无法取出, 我们假设该数据为 0, 接收多项式和错误多项式可以定义如下:

$$r(x)=s(x)+e(x),\quad e(x)=\sum_{i=0}^{n-1}e_ix^i.\tag{11}$$

因非出错的信息位 $e_i=0$, 所以 $e(x)$ 的单项式的个数为错误的个数. 如果只有一个坏盘, $e(x)$ 是单项式. 设出现错误的个数为 $\nu$, 则 $e(x)$ 可以简化为

$$e(x)=\sum_{k=1}^{\nu}e_{i_k}x^{i_k}.\tag{12}$$

通过解码, 找到对应位置 $i_k$ 和错误 $e_{i_k}$. 因分布式存储系统中 $i_k$ 是已知的, 所以只要解出 $e_{i_k}$ 即可. 如果坏盘的个数是 1, 则对应的码字中只有一个位置错误, 是可以通过读取其他 $n-1$ 个位置的信息解出 $e_{i_k}$. 这个很容易证明, 因 $\alpha$ 是 $g(x)$ 的根, 所以 $s(\alpha)=0$. 即 $\alpha^{n-k}\sum_{i=0}^{k-1}\alpha=\sum_{j=0}^{n-k-1}b_i$, 从这个表达式中可以直接解码出出错的信息位. 但从传输带宽的角度, 需要 $n-1$ 个节点都输出一份数据. 这样会给带宽造成很大的压力. 而对 $(n,k)$RS 码来说, 确定错误的位置后, 能够纠错的最大值为 $n-k$, 所以, 我们可以通过标记 $n-k$ 个错误的码字, 来降低传输的修盘带宽.

设编码中有 $n-k$ 个错误的码字. 如果错误码字的个数为 $\hat{n}<n-k$, 则可以随机从其他 $n-\hat{n}$ 个中选 $n-\hat{n}-k$ 个节点作为错误节点. 如果错误码字的个数超过 $n-k$, 则直接返回解码失败. 定义伴随式 $S_j$ 如下:

$$S_j=r(\alpha_j)=s(\alpha_j)+e(\alpha_j)=0+e(\alpha_j)=\sum_{k=1}^{\nu}e_{i_k}(\alpha^j)^{i_k},\quad j=1,2,\ldots,n-k.\tag{13}$$

则只要能够解出来 $\nu$ 个 $e_{i_k}$, 就可以修复出错的码字. 但根据 Forney [19] 算法, 同一个交换机下的存储节点需要发送出去 $\nu$ 个 $S_{i_k}$, $k\in\{1,2,\ldots,\nu\}$, 通过核心交换机传输的数据量并没有减少. 所以, 我们这里可以将生成多项式的编码方法转换成基于生成矩阵的方法. 下面, 我们证明基于生成多项式和生成矩阵方法的等价性.

不失一般性, 我们使用 $g(x)=(x-1)(x-\alpha)(x-\alpha^1)\cdots(x-\alpha^{n-k-1})$ 作为生成多项式, 且定义 $m(x)=\sum_{i=0}^{k}m_ix^{k-i-1}$,

$$b(x)=x^{n-k}m(x)\pmod{g(x)},\tag{14}$$

则 $x^{n-k}m(x)=q(x)g(x)+b(x)$. 进而可得

$$s(x)=x^{n-k}m(x)-b(x).\tag{15}$$

令 $b(x)=\sum_{i=0}^{n-k-1}b_ix^{n-k-i-1}$, 则 $s(x)=(m_0,m_1,\ldots,m_{k-1},-b_0,\ldots,-b_{n-k-1})$. 在 GF($2^8$) 内, $-\alpha=\alpha$, 则编码后的码字为 $(m_0,m_1,\ldots,m_{k-1},b_0,\ldots,b_{n-k-1})$.

**引理2** 对于 $g(x)$ 任意的根 $\beta$, 有 $\beta^{n-k}m(\beta)=b(\beta)$.

**证明** 因 $x^{n-k}m(x)-b(x)=g(x)d(x)s(x)$ 是生成多项式 $g(x)$ 的整数倍, 所以 $\beta^{n-k}m(\beta)=b(\beta)$.

**引理3** (生成多项式与生成矩阵的等价关系) 对于 $g(x)=(x-1)(x-\alpha)(x-\alpha^1)\cdots(x-\alpha^{n-k-1})$ 的生成多项式, 设 $G(x)$ 为 RS 码的校验位的生成矩阵, 且 $n\leqslant 2k-1$, 则

$$G(x)=A^{-1}\Lambda[A|B], \tag{16}$$

其中

$$A=\begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & \alpha & \cdots & \alpha^{n-k-1} \\ \vdots & \vdots & \cdots & \vdots \\ 1 & \alpha^{n-k-1} & \cdots & \alpha^{(n-k-1)(n-k-1)} \end{bmatrix}, \tag{17}$$

$$\Lambda=\begin{bmatrix} 1 & & & \\ & \alpha^{n-k} & & \\ & & \cdots & \\ & & & \alpha^{(n-k-1)(n-k)} \end{bmatrix}, \tag{18}$$

$$B=\begin{bmatrix} 1 & \cdots & 1 \\ \alpha^{n-k} & \cdots & \alpha^{k-1} \\ \alpha^{2(n-k)} & \cdots & \alpha^{2(k-1)} \\ \vdots & \cdots & \vdots \\ \alpha^{(n-k-1)(n-k)} & \cdots & \alpha^{(n-k-1)(k-1)} \end{bmatrix}. \tag{19}$$

**证明** 根据引理 2 可知,

$$m(1)=b(1), \tag{20}$$

$$\alpha^{n-k}m(\alpha)=b(\alpha), \tag{21}$$

$$\vdots$$

$$\alpha^{(n-k)(n-k-1)}m(\alpha^{n-k-1})=b(\alpha^{n-k-1}), \tag{22}$$

即

$$\Lambda[A|B][m_0,m_1,\ldots,m_{k-1}]^t=A[b_0,b_1,\ldots,b_{n-k-1}]^t. \tag{23}$$

然后可得该定理

通过上面的引理, 我们可以将基于生成多项式的解码算法转换成基于生成矩阵的算法, 而后采用 RS 分段解码算法, 间接地达到优化交换机传输带宽的目的.

### 3.3 算法实现

近似最优修盘算法的实现包括 3 个子算法: (1) 最优自修复比例的求解算法; (2) 磁盘修复任务调度算法; (3) 计算节点修复算法.

算法 1 的目的是计算出每个存储节点参与自修复的数据比例, 设为 $\alpha_i$. 算法第 1 行, 由定理 3, 获取求解存储系统最优修复盘时间的函数. 算法的第 2~9 行, 通过迭代的方式, 获取每个存储节点最优的参与自修复的数据比例 $\beta$ 及参与修复的计算节点的个数 $n_c$. 算法中 step 为迭代的步长, 可以根据系统的实际情况而定. 显然, step 越小, 得到的 $(\beta, n_c)$ 越精确. 第 11~17 行, 根据引理 1, 对近似最优自修复的数据比例进行调整.

**Algorithm 1** The optimal self-repair ratio algorithm

**Input:** $(k, W, V_{s_i}, V_{c_i}, n_s, n_c, d_{s_i})$;
**Output:** $(\alpha_1, \alpha_2, \ldots, \alpha_{n_s})$;
1: $f(\beta, n_c) = \min\{\frac{(k-1)W\beta}{\sum_{i=1}^{n_s} V_{s_i}}, \frac{kW(1-\beta)}{\sum_{i=n_s+1}^{n_c} V_{c_i}}\}$;
2: $(f_{\min}, \beta_{\min}, n_{c_{\min}}, \text{step}) = (\infty, 0, 0, 0.01)$;
3: **for** each $n_{c_i} \in [n_s, \hat{n}_c]$ **do**
4: **for** $\beta_v = 0$; $\beta_v \leqslant 1$; $\beta_v +=$ step **do**
5: **if** $f(\beta_v, n_{c_i}) < f_{\min}$ **then**
6: $(f_{\min}, \beta_{\min}, n_{c_{\min}}) = (f(\beta_v, n_{c_i}), \beta_v, n_{c_i})$;
7: **end if**
8: **end for**
9: **end for**
10: $(\beta, n_c) = (\beta_{\min}, n_{c_{\min}})$;
11: **for** each $i \in [1, n_c]$ **do**
12: **if** $\frac{W\beta V_{c_i}}{\sum_{i=1}^{n_c} V_{c_i}} \geqslant d_{s_i}$ **then**
13: $\alpha_i = 1$;
14: **else**
15: $\alpha_i = \frac{W\beta V_{c_i}}{(\sum_{i=1}^{n_c} V_{c_i}) d_{s_i}}$;
16: **end if**
17: **end for**

算法 2 整体分两个步骤: 第 1 步, 解析系统的元数据信息, 以条带为粒度生成待修复任务. 第 2 步, 等待计算节点的连接, 根据集群的状况动态地为计算节点分配修复任务, 并且在任务完成后, 及时地更新分配状态. 第 2 步使用了 allocTask 和 finishTask 两个函数, 见算法 3 和 4, 分别实现了监控节点分配任务和完成任务的逻辑. allocTask 对任务的调度是基于对 diskWeight 集合的控制而完成的.

算法 2 的第 3~15 行, 对所有条带参与修复的 $k$ 个数据块进行了选择. 选择的原则是, 尽可能地增加关联的 diskId. 因为参与修复的磁盘越多, 均衡到每个磁盘上, 参与修复的数据量就越少, 并发度也就越高. 第 19~29 行, 监控节点被动地等待计算节点领取修盘任务. 被动的方式有助于计算节点根据自己的负载情况按需地完成修盘任务.

修盘算法的第 3 个子算法是计算节点的修复算法, 见算法 5. 第 4~6 行, 算法可以自主地选择是否采用 CAR 算法进行解码时传输带宽的优化, 如果选择, 则存储系统内需要实现基于 RS 的分段解码. 前面的等价性已经证明, 无论采用基于生成多项式的方法, 还是基于生成矩阵的方法, 其本质上是等价的. 所以无论是否采用分段解码算法, 我们都可以找到等价的解码实现. 第 7~11 行, 计算节点完成了指定 stripeId 的修盘任务, 并且通知监控节点, 对已完成修复的任务进行处理.

**Algorithm 2** The repair task scheduling algorithm

Find all the {stripeId} of the broken disk diskId;
Let diskWeight be a map structure that key is diskId and value is repairing blocks count of broken disk;
**for all** stripeId in {stripeId} **do**
  **for** each $i \in \{1, \ldots, n\}$ **do**
    **if** after adding the $i$ block into the map diskWeight, the capacity of map is more than before **then**
      Choose block $i$ for stripe repair;
    **else**
      Skip block $i$;
    **end if**
  **end for**
  **if** the number of choose blocks satisfy $\hat{k} < k$ **then**
    Walk all the blocks in stripe stripeId, and random choose $k - \hat{k}$ blocks that is not choosed yet for stripe repair;
  **end if**
  Add choosed $k$ blocks into map diskWeight;
**end for**
According Algorithm 1, compute the approximately optimal repair ratio $(\alpha_1, \alpha_2, \ldots, \alpha_{n_c})$;
Generate the repair task list by repair ratio and wait computer node acquire;
Through socket mechanism, run the following process;
**repeat**
  **if** the event is acquire repairing task **then**
    Decode the request body and get node index $i$;
    Run allocTask($i$) (see Algorithm 3);
  **else if** the event is finish repaired task **then**
    Decode the request body and get node index $i$;
    Run finishTask($i$) (see Algorithm 4);
  **else**
    Handle other tasks;
  **end if**
**until** Close socket.

**Algorithm 3** allocTask(int $i$)

$n_s$ = len(diskWeight);
$c_s = n_c - n_s$, let $c_s$ be number of compute node that not handle self-repair tasks;
**if** $i$ a key of map diskWeight **then**
  Let $\hat{\alpha}_i$ be the ratio of self-repair tasks which have been alloc before;
  **if** $\hat{\alpha}_i < \alpha_i$ **then**
    **return** a self-repair task for node $i$;
  **else if** $\alpha_i < \hat{\alpha}_i < 1$, and the last alloc time of task for diskWeight[$i$] exceeds a particular value **then**
    **return** a self-repair task for node $i$;
  **else if** $\alpha_i < \hat{\alpha}_i < 1$, and the last alloc time of task for diskWeight[$i$] less than a particular value **then**
    **return** nil;
  **end if**
**else if** node $i$ is a compute node in $c_s$ **then**
  **return** a random repair task;
**else**
  Set compute node $i$ be a backup node for repair;
  **return** nil.
**end if**

**Algorithm 4** finishTask(int $i$)

```
Refresh diskWeight, and modify α̂_i;
if the key of diskWeight is empty then
    Report the repair process which has been finished;
end if
```

**Algorithm 5** Repair algorithm for compute node

```
repeat
  if the compute node acquire a non-nil task from monitor node then
    Decode the repair task;
    if fragment RS decode algorithm is used then
      Decode by CAR ( [16]) algorithm;
    else
      By stripeid, find all the hosts relative to k blocks;
      Read k blocks;
      Set the index of broken disk be j in stripe, decode by RS algorithm which code is created by generate matrix;
      Put the j block to new alloc disk (the monitor will alloc new disks for broken blocks);
      Send a finishTask request to monitor node;
    end if
  else
    The thread sleep for a while;
  end if
until
```

## 4 实验评估

实验评估的指标包括两个: (1) 通过交换机的数据总量, (2) 修盘时间. 实验环境共有 30 个 host. 每个 host 包括 4 个计算节点和 4 个存储节点. 为了实现方便, 我们将计算节点和存储节点进行合并, 统称为计算节点. 每个计算节点对应一块 960 G 的 SATA 盘, 用作数据存储. 系统盘则采用了 320 G 的 SSD 盘. 内存均为 16 G, CPU 统一为 Intel Xeon, X5472 3.00 GHz. 系统均为 Ubuntu 14.04.

存储单元数据块的大小为 4 MB, 验证的数据量为 500 个条带. 实验采用 3 种不同参数的 RS 码, 分别为 $(4,2)$, $(6,3)$ 和 $(10,4)$, 这 3 种编码在分布式存储系统中使用的最为频繁. 其中 $(4,2)$RS 方案采用 3 个接入层交换机的拓扑结构, $(6,3)$RS 方案采用 5 个接入层交换机的拓扑结构, 而 $(10,4)$RS 则采用 10 个接入层交换机的拓扑结构. 对应的存储节点的数目分别为 12, 20, 40 个. 条带经过 RS 编码后的存储规则是: 同一个条带组内的不同块, 可以存储在同一个接入层交换机内, 但不在一个存储节点上. 我们设定接入层的交换机最大带宽为 10 MB/s, 核心层的交换机最大带宽为 20 MB/s. 交换机的带宽设定, 通过软件层的限流算法实现.

其中, 服务的通信机制是通过 go-micro[6)]框架实现, ec 库则采用 golang 版本的实现[7)]. 实验通过随机洗牌的方法, 将 500 个条带分配到各个存储节点上, 而后随机地标记一个坏盘, 然后对坏盘进行修复. 需要注意的是, 仿真存储系统中数据的总容量为 500 个条带, 但单个坏盘的数据量要小于这个值. 而且集群越大, 单个坏盘涉及到的条带个数会越少. 为了使时间更具普遍意义, 我们重复多次随机地设置坏盘, 实验中的数据取其均值. 经过 20 次坏盘修复实验, 我们得到了实验数据, 见图 6 和 7. 其

6) go-micro. https://micro.mu.

7) go-ec. https://github.com/klauspost/reedsolomon.

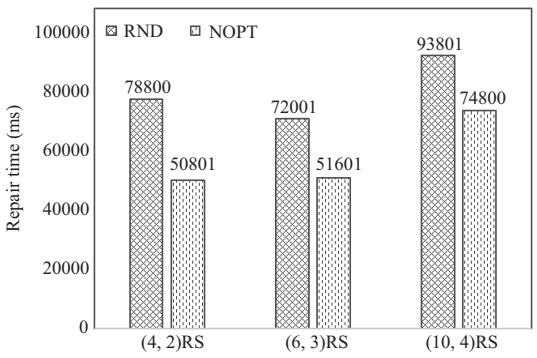


**图 6 修盘时延**

**Figure 6** Repair time

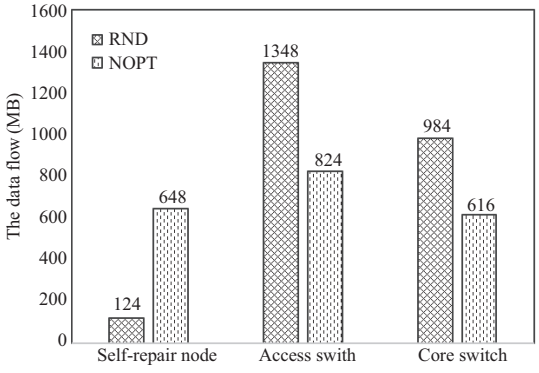


**图 7 数据流量**

**Figure 7** Data throughput

中, 图 7 统计了数据流的 3 个指标: 自修复数据量、接入层交换机的数据量和核心层交换机的数据量.

图 6 中, RND 算法采用的是随机分配条带的策略 (这种策略在存储系统中比较常用), 即计算节点根据自己的负载, 主动地向监控节点请求修盘任务. 监控节点则随机地派分任务给计算节点. 从图 6 可以看出, 相比 RND 算法, NOPT 算法的修盘时间缩短了 20.3%~35.5%. 修盘时延减少最主要的原因, 是因为在这个用例中, 交换机的流量, 是整体修盘过程中的瓶颈点. 减少接入层交换机和核心交换机的数据流量, 就可以显著地减少修盘的时间.

图 7 统计了 (4, 2)RS 的数据流量. 相比 RND 算法, NOPT 算法在分配任务的过程中, 显著地提高了自修复任务的比例, 可参见自修复流量. 需要注意的是, 自修复流量 + 接入层交换机流量 = 坏盘涉及条带的总修复数据量. 因为在坏盘相同的情况下, 前两部分的流量和是相等的, 所以, 我们通过调度, 能够分配多少任务进行自修复, 就能够避免多少的流量流经交换机, 进而加快修盘的进程.

除了分析经过交换机的传输量与修盘时间的关系外, 我们对是否计算节点越多, 磁盘的修复时间得到的优化就越大这个问题进行了仿真实验. 我们部署了 3 组验证方案, 采用了一致的 (6, 3)RS 码. 3 组方案交换机的个数分别为 10, 20 和 30. 我们通过分配算法, 将数据存储到 10 个交换机下. 测试前, 根据 3 种不同的方案, 选择不扩容、扩容 10 个交换机及计算节点和扩容 20 个交换机及计算节点. 实验数据如图 8, 每种方案中, 统计 3 种算法带来的实验结果. 第 1 种是基于随机调度算法的修盘时间, 第 2 种是基于最优自修复比例的求解算法 (算法 1), 但是对计算节点的个数不加限制下的修盘时间. 第 3 种是基于算法 1 的修盘时间. 由数据可以得出: 并不是参与修复的计算节点越多, 修盘时间就越少. 当参与修复的计算节点个数超过最优值, 将会导致更多的修盘数据流经过交换机, 所以修盘时间反而更长.

## 5 总结

随着硬件存储容量不断地变大, 修盘时间长的问题一直是分布式存储系统中的难题. 虽然通过纠删码的编解码构造 (例如 regenerating code) 可以优化修盘的数据量, 但同样存在着难以在系统中实现等问题. 本文从分布式存储系统的拓扑结构出发, 通过对数据流的分析, 推导出了近似最优的修盘传输量, 并由此设计了近似最优的磁盘修复算法. 通过对修盘任务的调度, 可以显著地减少通过交换机的修盘流量, 进而减少修盘时间.

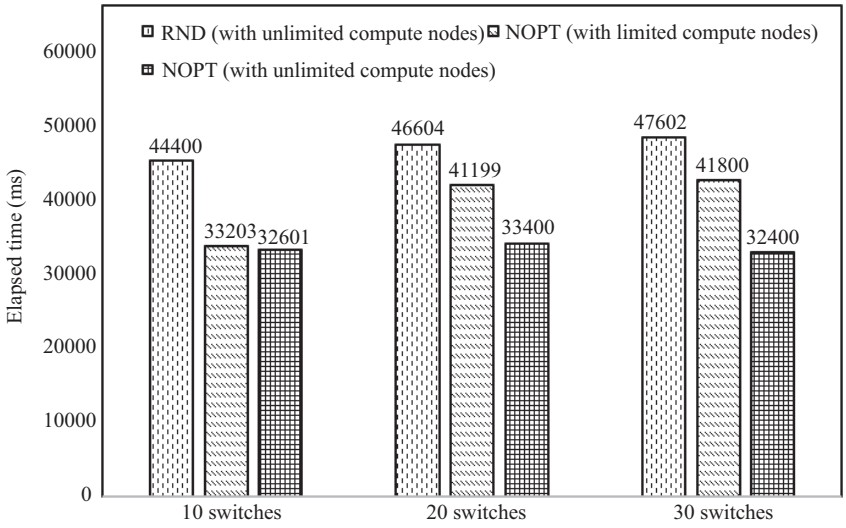


**图 8　自由计算节点个数对修盘的影响**

**Figure 8**　The influence of the number of compute nodes against disk repair

我们通过仿真验证了算法的有效性及优势. 后面, 我们会将算法逐步地移植到生产环境下的存储系统中, 例如 ceph, swift 等. 通过对任务的调度, 减少修盘的带宽, 并通过分段解码的算法, 减少通过交换机的带宽. 任务调度及编解码算法的构造仍然是我们需要做进一步深入研究的重点.

## 参考文献


1 Ghemawat S, Gobioff H, Leung S T. The Google file system. SIGOPS Oper Syst Rev, 2003, 37: 29–43

2 Shvachko K, Kuang H, Radia S, et al. The hadoop distributed file system. In: Proceedings of IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), 2010. 10: 1–10

3 Dimakis A G, Godfrey P B, Wu Y, et al. Network coding for distributed storage systems. IEEE Trans Inform Theor, 2010, 56: 4539–4551

4 Rashmi K V, Shah N B, Gu D, et al. A “hitchhiker’s” guide to fast and efficient data reconstruction in erasure-coded data centers. SIGCOMM Comput Commun Rev, 2015, 44: 331–342

5 Rashmi K V, Nakkiran P, Wang J, et al. Having your cake and eating it too: jointly optimal erasure codes for I/O, storage, and network-bandwidth. In: Proceedings of the 13th USENIX Conference on File and Storage Technologies, 2015. 81–94

6 Xu L H, Bruck J. X-code: MDS array codes with optimal encoding. IEEE Trans Inform Theor, 1999, 45: 272–276

7 Xu S, Li R, Lee P P C, et al. Single disk failure recovery for X-code-based parallel storage systems. IEEE Trans Comput, 2014, 63: 995–1007

8 Chowdhury M, Kandula S, Stoica I. Leveraging endpoint flexibility in data-intensive clusters. SIGCOMM Comput Commun Rev, 2013, 43: 231–242

9 Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. Commun ACM, 2008, 51: 107–113

10 Li R, Hu Y, Lee P P C. Enabling efficient and reliable transition from replication to erasure coding for clustered file systems. IEEE Trans Parallel Distrib Syst, 2017, 28: 2500–2513

11 Rashmi K V, Shah N B, Kumar P V. Optimal exact-regenerating codes for distributed storage at the msr and mbr points via a product-matrix construction. IEEE Trans Inform Theor, 2011, 57: 5227–5239

12 Huang C, Simitci H, Xu Y, et al. Erasure coding in windows azure storage. In: Proceedings of Presented as Part of the 2012 USENIX Annual Technical Conference, 2012. 15–26

13 Calder B, Wang J, Ogus A, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In: Proceedings of the 23rd ACM Symposium on Operating Systems Principles. New York: ACM, 2011.

143–157
14 Zhang Y, Kan H. Locally repairable codes with strict availability from linear functions. Sci China Inf Sci, 2018, 61: 109304
15 Xu Q Q, Xi W Y, Yong K L, et al. CRL: efficient concurrent regeneration codes with local reconstruction in geo-distributed storage systems. J Comput Sci Technol, 2018, 33: 1140–1151
16 Shen Z, Shu J, Lee P P C. Reconsidering single failure recovery in clustered file systems. In: Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). New York: IEEE, 2016. 323–334
17 Hu Y, Li X, Zhang M, et al. Optimal repair layering for erasure-coded data centers. ACM Trans Storage, 2017, 13: 1–24
18 Chien R. Cyclic decoding procedures for Bose- Chaudhuri-Hocquenghem codes. IEEE Trans Inform Theor, 1964, 10: 357–363
19 Forney G. On decoding BCH codes. IEEE Trans Inform Theor, 1965, 11: 549–557

# An approximately optimal disk repair algorithm for distributed storage systems

Jing SUN[1*], Songtao LIANG[2] & Xinjiang LU[3]

1. *Internet Plus Law Big Data Platform, East China University of Political Science and Law, Shanghai* 201620*, China*;
2. *Rich Media Dept., Qiniu Information Technology Co., Ltd, Shanghai* 200433*, China*;
3. *The Business Intelligence Lab, Baidu Research Center, Beijing* 100085*, China*
* Corresponding author. E-mail: jingsuncs@126.com

**Abstract** Effectively reducing disk repair time for distributed storage systems is an open problem. Generally, there are two directions, i.e., constructing better erasure codes to reduce disk repair throughput, and dispatching repair tasks to reduce the data flow in switches. In this paper, by analyzing the repair data flow, we provide an approximately optimal disk repair time. We also prove the best ratio of storage nodes for repairing and present the NOPT repair algorithm. We also prove the equivalence of the two decoding algorithms. The results of simulation experiments demonstrate that the repair time is improved significantly and repair traffic is reduced notably.

**Keywords** Reed-Solomon code, disk repair algorithm, distributed storage systems

**Sun JING** was born in 1985. She received her Ph.D. degree in computer science from Fudan University in 2015 and performed postdoctoral studies in software engineering at Fudan University. She is currently a lecturer at the East China University of Political Science and Law. Her research includes data mining, distributed storage systems, machine learning, and big data.

**Songtao LIANG** was born in 1985. He received his Ph.D. degree in computer science from Fudan University in 2015. He has previously worked at Huawei Technologies Co. Ltd and is currently working on object storage design at Qiniu Technologies. His current research includes erasure coding, distributed storage systems, and machine learning.

**Xinjiang LU** was born in 1984. He received his Ph.D. degree from Northwestern Polytechnical University, Xi'an, China in 2018.He is currently a researcher at Baidu. His research interests include mobile intelligence, urban computing, and data mining.