研 究 生（数 据 中 心 技 术）课 程 论 文（报 告）

题目：数据中心重复数据删除的性能优化综述

学　　号　　**M202073162**

姓　　名　　严伟杰

专　　业　　电子信息

课程指导教师　　施展、童薇

院（系、所）　武汉光电国家研究中心

2020 年 12 月 10 日

数据中心重复数据删除的性能优化综述

严伟杰[1)]

[1)](华中科技大学武汉光电国家研究中心 湖北 武汉 430074)

摘 要 由于当今时代数据的爆炸性增长，数据中心中存在着大量的冗余数据。重复数据删除作为一种有效的数据缩减方法，在数据中心等大型存储系统中受到越来越多的关注和欢迎。它消除了文件或子文件级别的冗余数据，并通过其加密安全的哈希签名识别重复的内容，与传统的压缩方法相比，该方法显示出更高的计算效率。典型的在线重复数据删除系统遵循着数据分块，指纹计算，指纹索引，差量压缩和存储管理的工作流程，其中指纹索引、差量压缩和数据恢复等阶段存在着各自的瓶颈问题，严重影响重复数据删除系统的性能。因此，重复数据删除技术的性能优化是近年来学术界和工业界共同关注的热点。本文通过梳理相关研究工作，研究了针对重复数据删除系统的指纹索引、差量压缩和数据恢复三个方面进行性能优化的思路和方法，深入分析了每个方法的原理和算法细节。

关键词 数据去重；数据压缩；性能优化；数据恢复

Survey on Performance Optimization of Data Deduplication in Data Center

Weijie Yan[1)]

[1)](Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, Hubei 430074)

Abstract Due to the explosive growth of data in today's era, there is a large amount of redundant data in the data center. As an effective data reduction method, data deduplication has received more and more attention and welcome in large storage systems such as data centers. It eliminates redundant data at the file or sub-file level and recognizes duplicate content through its cryptographically secure hash signature. Compared with traditional compression methods, this method shows higher computational efficiency. A typical online data deduplication system follows the workflow of data partitioning, fingerprint calculation, fingerprint indexing, differential compression and storage management. Among them, fingerprint indexing, differential compression and data recovery have their own bottleneck problems, which seriously affect the performance of the deduplication system. Therefore, the performance optimization of data deduplication technology is a hot spot that academia and industry have paid attention to in recent years. Through combing related research work, this paper studies the ideas and methods of performance optimization for the fingerprint index, difference compression and data recovery of the deduplication system, and deeply analyzes the principle and algorithm details of each method.

Key words Data Deduplication; Data Compression; Performance Optimization; Data Restore

1 引言

随着互联网时代的到来，全球数据量呈现指数式增长。根据 IDC 的报告，2020 年全球生成的数据量将超过 44ZB[1]。随着数据中心中存储的数据量的增加，冗余数据的比例也在不断增加。根据 Microsoft[2][3]和 EMC[4]的研究，它们的数据中心中分别约有 50%和 85%的数据是冗余的重复数据。IDC 最近的一项研究显示，接近 80%接受调查的公司正在探索其存储系统中的重复数据删除技术以减少冗余数据[5]。如何有效地管理数据，提高数据存储效率并减低存储成本成为了当前学术研究的热点之一。为了有效地管理大量冗余数据，重复数据删除技术已在数据中心得到广泛应用[6]。

重复数据删除技术是一类无损的数据压缩技术。它不仅可以通过消除重复数据来减小存储空间，还可以减小低带宽网络环境中冗余数据的传输，从而提高传输性能。重复数据删除技术通常可以分为两类：离线重复数据删除和在线重复数据删除。离线重复数据删除是在数据写入存储设备后，在后台消除存储设备中的冗余数据。而在线重复数据删除则是实时地边存储数据边进行冗余数据的消除。离线重复数据删除会占据更多的存储空间，并且不会减少数据的写操作。因此，在线重复数据删除具有更好的研究价值。

典型的在线重复数据删除系统遵循着数据分块，指纹计算，指纹索引，进一步压缩和存储管理的工作流程[6]。重复数据删除系统中的存储管理可以分为几类，包括数据恢复，垃圾回收，可靠性，安全性等，如图 1 所示。数据分块是利用数据分块算法将数据流分成许多大小不等的数据块。相比于早期的文件级的重复数据删除技术，数据块级的重复数据删除技术可以实现更高的去重率。指纹计算是利用哈希算法计算出每个数据块的指纹。数据块指纹作为数据块的唯一标识，系统可以通过比较数据块指纹是否相同来判断数据块是否重复。指纹索引是重复数据删除系统的关键阶段。系统为已经存储的数据块指纹建立索引，通过查询索引来判断数据是否重复。进一步压缩阶段指的是将那些非重复但是非常相似的数据块进行压缩存储，即差量压缩。

重复数据删除的性能对于数据中心至关重要，而重复数据删除系统中涉及许多计算和磁盘索引等严重影响 I/O 性能的操作，因此需要采用优化技术来提高重复数据删除系统的吞吐量。近年来，不同研究人员分别从重复数据删除的多个阶段对重复数据删除系统进行了优化，取得了显著的效果。

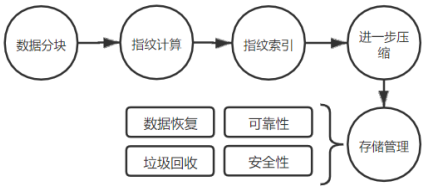


图 1 重复数据删除技术的工作流程

为了进一步深入研究数据中心的重复数据删除技术的性能优化，接下来本文会分别介绍三篇文献。它们分别从指纹索引、差量压缩和数据恢复三个方面对重复数据删除的瓶颈部分进行了优化。本文会详细分析其原理和优化效果，最后对重复数据删除的性能优化做出总结。

2 背景

在基于数据块的重复数据删除系统中，一个非常重大的挑战是如何构建有效的指纹索引以帮助识别重复的数据块。当数据量很大时，在 RAM 中保持非常大的指纹索引是不切实际的。但是如果将索引放在磁盘里，频繁的磁盘读取又会严重影响 I/O 性能。这个问题被称为块查找磁盘瓶颈问题[10]。例如，对于 1PB 的唯一数据集，假设平均块大小为 8KB，它将生成大约 2.5TB SHA-1 指纹（即每个块 160 位）。指纹索引结构在查找等待时间和重复数据删除率方面会严重影响重复数据删除性能，因为每次只能访问这些指纹的一部分。

针对块查找磁盘瓶颈问题，很多研究机构已经提出了一些解决方案。Sparse indexing[11]通过对内存中的数据块指纹索引进行采样来提高重复数据删除系统的内存利用率，从而将内存使用率降低到一半以下。Extreme Binning[12]通过利用文件相似性来实现每个文件的单个磁盘索引访问以进行块查找，从而提高了重复数据删除的可伸缩性。

在存储系统中，差量压缩通常用作重复数据删除技术的补充技术，因为它能够消除非重复但高度相似的数据块之间的冗余。例如，如果数据块 A2 与数据块 A1 相似，那么差量压缩方法可以仅存储或传输 A2 和 A1 之间的差异和映射关系，从而删除冗余数据以提高存储空间效率和网络带宽效率。如

图 2 所示，差量压缩的一般工作流程是：①计算数据块的相似度，即计算特征和特征分组。②检查相似的数据块。③对两个相似的块进行编码，即计算它们的差量。

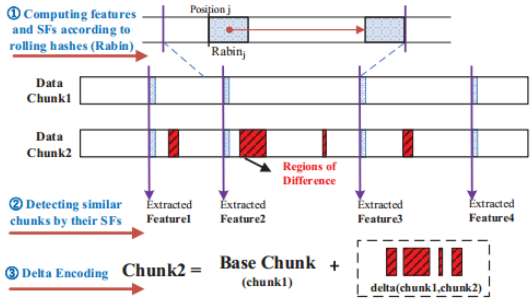


图 2 差量压缩的基本思路流程

为了获得高的差量压缩效率，最近一些关于差量压缩的研究建议将四个或更多的特征分组到一个 SF 中以减少相似检测中的误报，并使用三个或更多的 SF 来检测高度相似的块进行差量压缩。但根据我们在差量压缩原型系统中的观察，计算数据块的相似度，即生成它们的 SF，是相当耗时的。

当前，N-transform SF[15]是最广泛使用的计算数据块相似度以检测差量压缩候选数据块的方法。N-transform SF 的具体算法是：利用 Rabin 指纹在数据块上逐字节计算，然后对每个 Rabin 指纹分别线性变换 N 次以计算 N 维哈希值集。最后，从 N 个维度中的每个维度中选取 N 个最大值作为特征。由于算法需要进行 N 次线性变换，所以计算非常耗时。

恢复原始数据是重复数据删除的反向过程。在重复数据删除系统中，由于数据块分散在许多容器中，数据恢复时会频繁读取容器，因此数据块碎片会严重阻碍数据恢复性能，造成 I/O 性能的下降。目前基于容器的缓存，基于块的缓存和前向组装等的几种方案是当前主流的用于减少恢复过程中的磁盘读取次数的方法。基于容器的缓存具有较低的运行成本，但是基于块的缓存可以更好地过滤出与将来恢复无关的数据块，从而更好地提高缓存空间的利用率。Lillibridgeet 等人提出的一种基于块的缓存的特殊方式称为前向组装[16]。该方法将多个容器用作称为前向装配区（FAA）的组装缓冲区，并且使用大小相同的前瞻窗口来标识要在下一个容器中恢复的数据块。与基于块的缓存相比，前向汇编的开销较低。如果每个唯一的数据块将在恢复后的短时间内重新出现，则前向汇编会非常有效。

3 研究方法

3.1 基于指纹索引的优化

（1）思路动机

论文作者经过研究发现，现有重复数据删除系统中存在以下事实：对于基于采样的指纹索引，采样率与内存开销成比例并进一步直接影响重复数据删除率。同时现有系统的采样的指纹与数据段之间的映射关系是静态的，缺乏自适应反馈机制来调整映射关系以反映数据流的动态。因此，论文提出了利用一种简单的强化学习方法建立动态预取数据段的索引，简称 LIPA。LIPA 采用了一种称为 K 臂赌博机的特定强化学习模型[13]，该模型广泛用于内容推荐和预取。由于数据段可以在数据备份流中具有相同的特征，从而彼此关联。因此，模型将每个段都对应一个分支，然后通过特征从多个分支（也就是多个数据段）中动态地选择更优的数据段进行预取，从而减少磁盘读取的次数，提高 I/O 的性能。

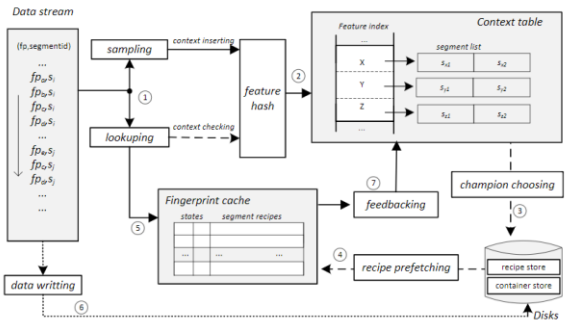


图 3 LIPA 架构图

（2）算法细节

与传统的指纹缓存的方法相比，LIPA 增加了一个核心组件：上下文表。上下文表用于存储重复数据删除过程中指纹查找的运行时上下文信息。对于即将到来的数据段，系统将数据段采样的特征映射到上下文表中，并建立其特征和数据段的映射关系。上下文表具有两个重要的属性：段的分数和段的追随者。分数表示随着时间的推移命中对该数据段进行查找的奖励。它由数据段在指纹缓存中的平均命中次数表示。追随者表示当前数据段被选择为冠军后将预取到缓存中的连续数据段的数量。追随者的数量由预取的效果来动态决定。当从指纹缓存中获取反馈时，它们会被更新。

LIPA 的指纹缓存存储着数据段的指纹信息和

命中次数。从系统上下文表中选择冠军数据段时，该冠军数据段及其一些追随者数据段将被预取到指纹缓存中，同时某些数据段将被驱除出指纹缓存。当数据段被驱逐出指纹缓存后，它会将它记录的命中次数反馈到上下文表中，作为该数据段的分数信息。

LIPA 的架构图如图 3 所示。它的流程如下：①对于传入的数据段，首先通过采样算法获得其特征。②将特征与数据段之间的映射关系存储到上下文表中。上下文表的每个条目对应于大多数关联中的固定的特征。多个数据段共享相同的特征。 我们将这种情况建模为多武装匪徒问题。③从当前上下文中提高选择策略对可选择的臂（即数据段）进行选择。被选择的数据段被称为一个冠军数据段。④将冠军数据段和后续的数据段都被按顺序预取到指纹缓存中。⑤在指纹缓存中检查对数据段的每个数据块是否重复。⑥非重复的数据块被写回磁盘。⑦如果从指纹缓存中逐出了一个缓存的段，它会将奖励反馈给选定的分支，并通过缓存中的指纹查找命中来更新上下文表。数据段在高速缓存中的期间，记录指纹查找的命中次数。我们以 K 臂武装匪徒模型中的命中率为依据，进一步设计了奖励政策和反馈机制。

LIPA 的冠军选择策略有以下三种政策：最近，随机和贪婪。 顾名思义，最近的策略总是选择最新的细分作为冠军，而随机策略则总是从候选集中随机选择具有相等概率的细分，这可以认为是贪婪策略ε= 0 的特殊情况。ε −贪心选择策略可以将其视为随机策略的改进。在每个试验中，它都可以选择具有较高得分的段，其概率为 1-ε（即对过去的利用）；否则，它可能会选择具有概率ε的随机细分（即对未来的探索）。实现在利用和探索之间找到更好的平衡至关重要。这是强化学习算法中探索与利用的平衡。

上下文表的更新是由以下两个方面引起的：条目的添加/删除和奖励的反馈。上下文表使特征和映射的段之间的关联保持了联系。对段的特征进行采样后，如果该特征在上下文表中是新的，则将特征与该段的关联作为新条目添加到上下文表中；否则，将数据段添加到特征映射的数据段队列中。在上下文表的每个条目中，每个队列中具有相同最大数量的数据段。当队列已满时，必须从队列中删除之前的数据段。LIPA 提供了两种可选择的策略：FIFO 和 Mininum。FIFO 策略是按照添加到队列中的顺序删除条目。Mininum 策略是删除得分最低的数据段。这两种策略在内存和计算上都没有太多成本。

一旦将冠军及其一定数量的追随者数据段预取到指纹缓存中，则重复数据删除将查找传入段的指纹。查找命中意味着已识别出重复的数据。在缓存中获得冠军数据段的期间，我们将其从缓存中逐出旧的数据段。旧数据段之前的所有查找结果相加，然后反馈奖励并更新上下文表中的相应得分。设 s 是一个数据段，$Q_n(s)$为段 n 次选为冠军数据段后的奖励的r_1, r_2, …, r_n的估计得分。其中 r 分别作为奖励值可以是段的查找次数的计数，计算$Q_n(s)$的一种方法是计算平均值。

$$Q_n(s) = \frac{1}{n}\sum_{i=1}^{n} \mathrm{r}_i$$

然而，该公式会导致内存计算的开销增大，所以论文采用了增量的替代方法计算平均值。

$$Q_n(s) = Q_{n-1}(s) + \frac{1}{n}(\mathrm{r}_n - Q_{n-1}(s))$$

一开始$Q_0(s) = 0$。此实现仅需要对$Q_{n-1}(s)$和 n 进行存储，而对于每个新奖励仅需进行少量计算

在 LIPA 的实现过程中，非常重要一点的是要自适应地确定一个冠军被选中后预取的追随者的数量。一种简单的预取方法是设置为一个固定值。例如，每次预取 4 个追随者数据段。由于每个缓存的数据段都会获得反馈奖励，因此可以采用自适应预取方式动态调整追随者的数量 n。为了实现自适应方法，在上下文表的每个条目中，将预取的跟随者的数量记录为段的属性。最初，每个新条目设置默认值 4。高速缓存中的每个数据段都会记录信息，包括它是否为最后一个追随者以及它跟随的冠军。当从缓存中撤出最后一条追随着时，它将根据其奖励调整其冠军的追随者数量 n。

（3）优化效果

针对 Linux Kernel、Vmdk、Fslhomes 和 Macos 四个数据集， LIPA 与传统的 Sparse Indexing 的方法进行了对比测试。实验基于 Destor[14]的实验测试平台，采用 TTTD 和 CDS 作为分块和分段算法，利用 SHA-1 计算数据块的指纹。实验的采样方法为最小采样。如图 4 所示，实验结果显示 LIPA 不仅可以提高数据去重率，在部分数据集上还可以极大提升 I/O 性能。

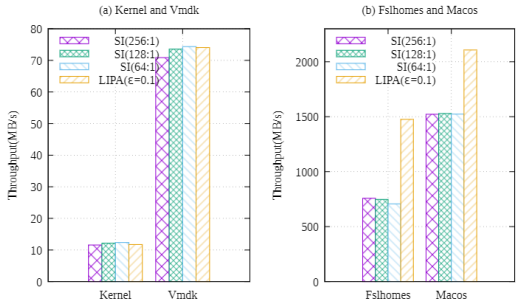


图 4 LIPA 架构图

3.2 基于差量压缩的优化

（1）思路动机

差量压缩中最广泛使用的计算数据块相似度的算法是 N-transform SF 算法。N-transform SF 算法需要进行 N 次线性变换，是计算密集型算法。因此，差量压缩中的相似度检测阶段成为严重影响重复数据删除性能的阶段。

论文观察到相似块之间也存在细粒度的局部性。数据块的相应子区域（子块）及其特征在相似的数据块中也以很高的概率出现在相同的顺序中，这被称为特征局部性。基于以上的事实，论文提出了 Finesse，这是一种基于特征局部性的细粒度快速重相似度检测方法，该方法将每个块划分为几个固定大小的子块，分别计算这些子块中的特征，然后将特征分组为超特征，从而简化了相似度计算。

（2）算法细节

Finesse 的具体实现包括特征提取和特征分组。特征提取：为了利用相似块的细粒度特征局部性来提取特征，Finesse 首先将数据块划分为几个固定大小的子块，然后根据数据内容的 Rabin 指纹在每个子块上计算特征，此处与传统的 N-transform SF 方法相同。由于基于内容分块算法会将数据块分成大小不一的子块，从而使计算变得非常复杂。因此论文采用固定大小分块的方法划分子块。特征提取算法如图 5 所示。

```
Require: chunk content, Str; length of the chunk, L;
Ensure: N features, Feature[N];
1: function FEATURE-EXTRACT-FINESSE(Str, L)
2:     subChunkSize ← L/N;
3:     Feature[0, ··· , N−1] ← 0;
4:     for m = 0 to N−1 do
5:         for i = 0 to subChunkSize−1 do
6:             FP ← RabinFunction(Str, m*subChunkSize+i);
7:             if Feature[m] ≤ FP then
8:                 Feature[m] ← FP;
9:             end if
10:        end for
11:    end for
12: end function
```

图 5 Finesse 特征提取算法

特征分组：Finesse 中的分组策略不同于传统的方法，因为在 Finesse 中更改了特征提取方式。具体来说，Finesse 首先将子块及其对应的特征划分为几个相同大小的连续集合，然后将最大的特征（每个集合中具有最大哈希值）分组以构成第一个 SF，将集合的第二个最大特征分组以形成第二个 SF，依此类推。这种分组策略可以确保在每个块中均匀一致地选择每个 SF 中的分组特征，从而实现类似于 N-transform SF 的分组效率。如图 6 所示的三个 SF 和每个 SF 的四个特征的详细示例。我们首先将块划分为十二个子块，以提取 12 个特征 F0 ... F11。这些特征进一步分为四组，并按它们的值{F0 <F2 <F1} ... {F11 <F10 <F9}进行排序。最后，SF0 由上述四组的最大值组成，即{F1，F5，F7，F9}，第二大值{F2，F3，F8，F10}的 SF2 和第三大值{F0，F4，F6，F11}的 SF3。因此，与特征提取相比，特征分组是快速的，因为它仅处理少量特征而不是整个数据块。

N-transform SF 需要至少 3xN 次操作，而 Finesse 只需要一次分支操作，因此大大加快了系统的性能。Finesse 的局限性在于它无法检测到大小相差很大的相似块。因为如果两个相似块的大小相差很大，那么它们通过划分子块并分组所得到的特征将完全不同。

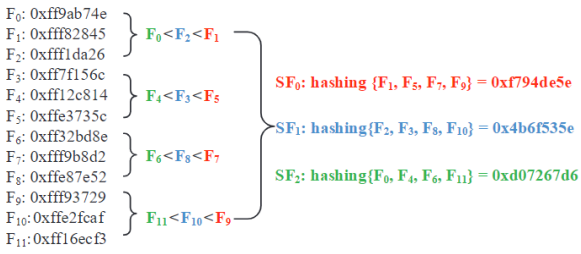
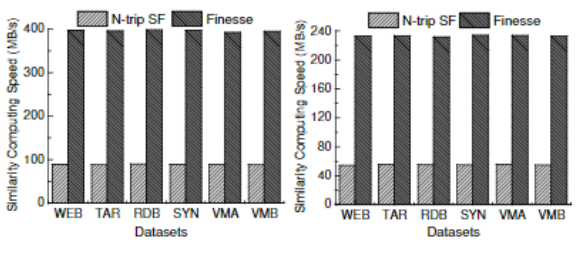


图 6 Finesse 特征分组

（3）优化效果

实验将 Finesse 与传统的 N-transform SF 在两个不同机器上进行对比实验。



(a) Intel i7-4770　　(b) Intel E5-2620

图 7 吞吐量对比

实验测量在六个不同数据集上的差量压缩率（DCR）和差量压缩效率（DCE）以及吞吐量。DCR 反映了差量压缩的空间缩减效果，DCE 反映了相似块之间的相似程度，如图 7 和图 8 所示。

Dataset	Approaches	DCR	DCE
WEB	*N-transform SF*	7.60	0.8749
	Finesse	7.52 (–1.05%)	0.8795 (+0.53%)
TAR	*N-transform SF*	15.00	0.9516
	Finesse	15.34 (+2.27%)	0.9846 (+3.47%)
RDB	*N-transform SF*	3.67	0.9129
	Finesse	3.94 (+7.36%)	0.9448 (+3.49%)
SYN	*N-transform SF*	1.75	0.9326
	Finesse	1.70 (–2.86%)	0.9640 (+3.37%)
VMA	*N-transform SF*	1.56	0.9088
	Finesse	1.51 (–3.21%)	0.9161 (+0.80%)
VMB	*N-transform SF*	1.30	0.9093
	Finesse	1.28 (–1.54%)	0.9193 (+1.10%)

图 8 DCR 和 DCE 对比

实验结果可知，Finesse 与 N-transform SF 具有相近的压缩率，而 Finesse 可以检测到更加相似的数据块。在相似度计算阶段，Finesse 在两台机器上分别实现了 3.5x 和 3.2x 的速度提升，大大改善了系统的 I/O 性能。

3.3 基于数据恢复的优化

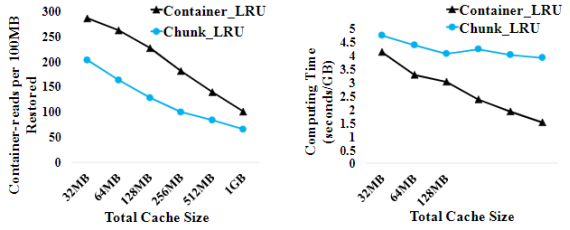
（1）思路动机

由于重复数据删除之后，数据块分散在许多不同的容器中，数据恢复时会频繁读取容器，因此数据块碎片会严重阻碍数据恢复性能，造成 I/O 性能的下降。这是数据恢复阶段的瓶颈问题。当前主流的基于容器的缓存、基于块的缓存和前向组装具有各自的优缺点，适用于不同的工作环境。论文作者系统通过分析不同算法的优缺点，将其动态地组装起来从而提高适用于多数环境下，提高数据恢复的性能。

（2）算法细节

论文首先通过不同数据集对比分析基于容器的缓存、基于块的缓存和前向组装的缓存效率，如图 9 图 10 所示。

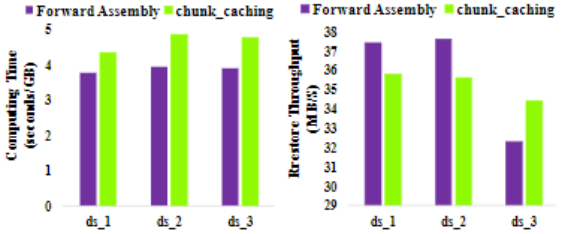
通过对比分析可知以下几点：①在大多数情况下，缓存的容器中的某些数据块与当前和不久的将来的恢复过程无关，同时有用的数据块可能会被迫与整个容器一起被逐出缓存。因此，缓存块比缓存容器效果更好。②前向组装的计算时间比基于块的缓存的计算时间低得多。如果大部分数据块时非重复数据块或者数据块重用的距离小于 FAA 的范围，则前向组装的性能会优于基于块的缓存。否则，它的性能会优于前向组装。由于前向组装和基于块的缓存在不同场景下各有优势，所以论文希望将二者结合起来，利用前瞻窗口(LAW)、FAA 和块缓存来进行数据恢复。



(a) # Containers-reads per 100MB restored as cache size varies from 32MB to 1GB

(b) Computing time per 1GB restored as cache size varies from 32MB to 1GB

图 9 基于容器缓存和基于块缓存的对比



(a) Computing time per 1GB restored

(b) Restore throughput

图 10 FAA 和基于块缓存的对比

FAA 由几个容器大小的存储缓冲区组成，它们以线性顺序排列。我们将每个容器大小的缓冲区称为 FAB。恢复指针会精确指出要查找的配方中的数据块，然后将其复制到其在第一个 FAB 中的相应位置。指针之前的数据块已经被组装。如果这些数据块也出现在这些 FAB 中，则将使用其他 FAB 来保存访问的数据块。LAW 从第一个 FAB 中的第一个数据块开始（在示例中为 FAB1），并且覆盖的范围大于配方中 FAA 的范围。实际上，LAW 的第一部分（相当于 FAA 的大小）用于组装目的，而 LAW 的其余部分则用于基于块的缓存。其中 F 块表示根据 LAW 得知的未来会用到的数据块，P 块表示 FAA 刚刚组装完的数据块。作者没有固定 FAA，LAW 和块缓存的大小，而是设计了一个动态自适应算法 ALACC 来适应不同的工作负载。ALACC 可以动态调整 FAA 和块缓存的存储空间比例以及 LAW 的大小，如图 10 所示。算法首先检测调整 FAA 的大小的条件。如果条件满足，FAA 将相应地增加 1 个容器大小，而块缓存的大小将相应地减少 1 个容器。否则，我们将检查调整块缓存大小的条件。最后，算法会修改 LAW 的大小。

当以下情况发生时，FAA 的性能要优于基于块的缓存：①第一个 FAB 中的数据块主要被标识为唯一数据块，并且这些块存储在相同或紧密的容器中；②FAB 里的大部分重复数据块的重用距离在 FAA 范围内。关于第一个条件，我们认为可以通过读取不超过 2 个容器来填充 FAB，并且 FAB 所需的数据块都不来自块缓存。发生这种情况时，我们认为该组装周期 FAA 有效。对于条件 2，我们观察到，如果此 FAB 中 80%或更多重复块的重用距离小于 FAA 大小，则前向组装的性能更好。

因此，使用以下两个条件之一来增大 FAA 大小：首先，如果连续的 FAA 有效组装周期数大于给定阈值，则将 FAA 大小增加 1 个容器。在这里，我们使用当前 FAA 大小S_{FAA}^{i}作为阈值，以在第 i 个组装周期结束时测量此条件。当S_{FAA}^{i}尺寸较小时，条件更容易满足，并且 FAA 的尺寸可以更快地增加。当S_{FAA}^{i}较大时，条件很难满足。在将 FAA 大小增加一之后，将连续的 FAA 有效组装周期的计数重置为 0。第二，检查在该组装周期中用于填充 FAA 中第一个 FAB 的数据块。如果在组装周期中这些被检查的块中 80%以上的重用距离小于S_{FAA}^{i}+ 1 容器大小，则 FAA 的大小将增加 1 个容器。即$S_{FAA}^{i+1} = S_{FAA}^{i}$ + 1。ALLACC 算法实例如图 11 所示。

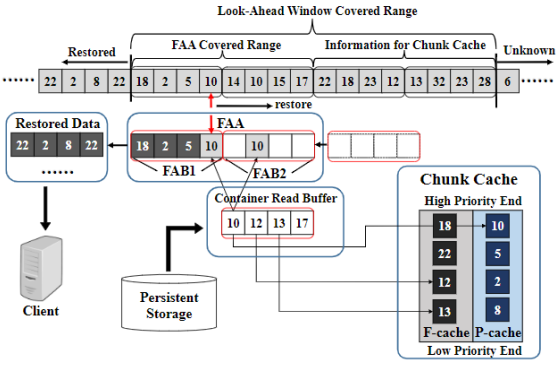


图 11 ALACC 算法实例

如果将 FAA 大小增加 1 个容器，则块缓存的大小将相应减少 1 个容器。最初，LAW 中S_{LAW}^{i}-S_{FAA}^{i}个容器大小供S_{cache}^{i}缓存使用。经过 FAA 调整后，$S_{cache}^{i}-1$个容器大小的高速缓存使用了S_{LAW}^{i}-S_{FAA}^{i}+ 1 个容器大小的 LAW 信息，这浪费了 LAW 中的信息。因此，将 LAW 大小减少 1 个容器大小，以避免现在较小的块缓存使用相同的 LAW 大小。在确定了新的 FAA 大小，块缓存和 LAW 之后，将在 FAA 的末尾添加两个空的 FAB（一个用于替换重新存储的 FAB，另一个用于反映 FAA 的大小增加的数据块，然后将开始第 i+1 个组装周期。

如果没有调整 FAA 大小，我们现在考虑调整块缓存大小。在完成上述组装周期后，对 F 缓存中 F 块的总数$N_{F-chunk}$和 P 缓存中 P 块的总数$N_{P-chunk}$进行计数。同样，在第 i 个组装周期中新添加到 F 缓存的 F 块的数量用$N_{F-added}$表示。这些新添加的 F 块是在装配周期中来自读入的容器或者由于 LAW 的扩展而从 P 块转换而来。我们检查以下三个条件。首先，如果$N_{P-chunk}$变为 0，则表明所有高速缓存空间都被 F 块占用。当前的 LAW 大小太大，并且基于当前 LAW 的 F 块的数量大于块缓存的容量，因此块缓存的大小将增加 1 个容器。同时，LAW 的大小将减少 1 个容器，以减少不必要的开销。其次，如果添加的$N_{F-added}$大于 1 个容器，则表明 F 块的总数迅速增加。因此，我们将块缓存的大小增加 1 个容器，将 LAW 减少 1 个容器。注意，当$N_{P-chunk}$太大或太小时，可能会添加一个较大的$N_{F-added}$。这种情况将使算法对工作负载的变化迅速做出反应。第三，如果$N_{P-chunk}$非常大，则块缓存大小将减少 1 个容器。 在这种情况下，将根据以下两个条件之一来不同地调整 LAW 的大小：①当前的 FAB 中很少有数据块可在将来重用；②LAW 的大小太小，无法为当前的工作量寻找足够多的 F 块。对于条件 1，我们将 LAW 大小减小 1 个容器。对于条件 2，我们将 LAW 大小增加 K 个容器大小。在此，K 由 K $=(MAX_{LAW}-S_{LAW}^{i})/(S_{FAA}^{i}+S_{cache}^{i})$计算。如果 LAW 很小，则其大小会增加很多。如果 LAW 大，其大小将缓慢增加。

如果上述条件都不满足（FAA 和块缓存的大小保持不变），则 LAW 大小将独立调整。在这里，我们使用$N_{F-chunk}$来决定调整。如果$N_{F-chunk}$小于给定的阈值（例如，总块缓存大小的 20%），则 LAW 大小将稍微增加 1 个容器以处理更多的将来信息。如果$N_{F-chunk}$高于阈值，则 LAW 大小将减小 1 以减少计算开销。ALACC 在计算开销和减少容器读取之间进行权衡，从而可以实现更高的还原吞吐量。相比于使用固定值作为阈值，算法动态更改 FAA 大小和 LAW 大小。当开销大时（当 LAW 大小很大时），它可以减慢调整；当开销小时，它可以加快调整速度，以快速减少容器读数（当 FAA 大小小时）。

（3）优化效果

为了评估 ALACC 的性能，论文采用了五个恢

复算法进行对比，分别是 ALACC，基于 LRU 的容器缓存，基于 LRU 的块缓存，FAA 和 FAA 与基于块缓存的固定组合 Fix_Opt。在四个数据集上的测试表现如图 12 所示。实验结果图中从左到右分别表示计算速度，计算开销和吞吐量。对于所有 4 个数据集，ALACC 的总体平均速度最快，吞吐量最高。ALACC 至少可以达到与 Fix_Opt 相似或更好的性能。

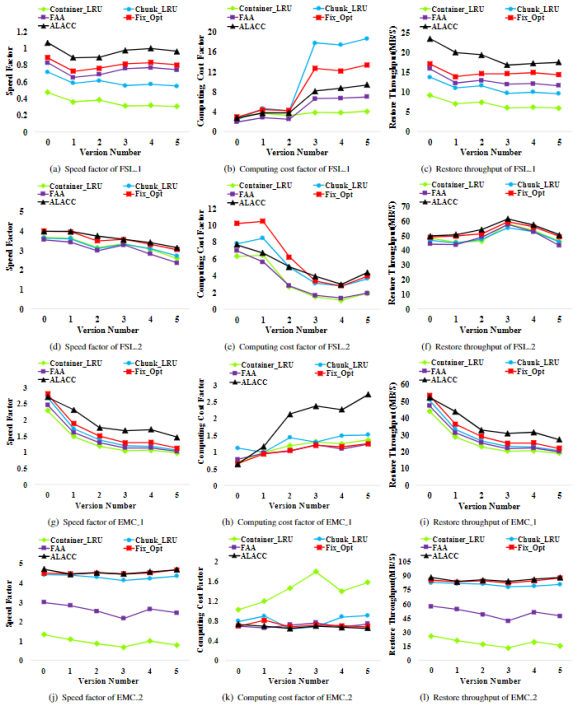


图 12 ALACC 算法实验对比

4 论文总结

由于数据中心存在大量冗余数据，影响了存储的性能。作为消除冗余数据，提高存储空间和网络传输带宽的关键技术，重复数据删除技术已经成为数据中心中不可缺少一部分。如何降低重复数据删除过程中的性能开销，提高数据吞吐量是重复数据删除更好地应用于数据中心的关键之一。以上三篇论文分别从指纹索引，差量压缩和数据恢复三个方面对重复数据删除技术进行了优化，提高了 I/O 性能。

LIPA 的最大贡献在于首次在重复数据删除技术中加入机器学习的方法，提出了一种基于强化学习的指纹索引检测与预取技术以解决磁盘瓶颈问题。系统通过反馈机制更新了特征及其对应数据段之间的关联关系并动态调整了缓存机制以提高重复数据删除系统的性能。实验结果表明基于学习的方法只需要很少的内存开销即可存储索引，同时与以前的方法相比，其重复数据删除率和吞吐量甚至更高。

Finesse 针对当前的差量压缩相似性检测部分的主流算法 N-transform SF 进行研究，发现了数据块之间的细粒度特征局部性。根据高度相似的块的特征局部性，Finesse 改进了传统的 N-transform SF 方法，将数据块划分为多个子块并从每个子块中提取特征并分组为 SF。算法针对差量压缩的瓶颈阶段进行了改进，减少了相似度检测的计算开销，极大提升了重复数据删除系统的 I/O 性能。

第三篇论文研究了应用于数据还原过程的不同缓存机制的效率和效率。基于对缓存效率实验的观察，论文设计了一种称为 ALACC 的自适应算法，该算法能够根据工作量的变化来自适应地调整 FAA，块缓存和 LAW 的大小。通过在容器读取数和计算开销之间进行更好的权衡，ALACC 的还原性能要比基于容器的缓存，基于块的缓存和前向组装更好，拥有最佳的恢复速度，提高了数据恢复的 I/O 吞吐量。

参 考 文 献

[1] L. DuBois, M. Amaldas, and E. Sheppard, “Key considerations as deduplication evolves into primary storage,” White Paper 223310, 2011

[2] A. El-Shimi et al., “Primary data deduplication - large scale study and system design,” USENIX Annual Technical Conference, 2012

[3] D. T. Meyer and W. J. Bolosky, “A study of practical deduplication,” ACM Transactions on Storage (TOS), 2012.

[4] P. Shilane et al., “WAN optimized replication of backup datasets using stream-informed delta compression,” USENIX Annual Technical Conference, 2012

[5] L. DuBois, M. Amaldas, and E. Sheppard, “Key considerations as deduplication evolves into primary storage,” White Paper 223310, Mar. 2011. [Online]. Available: http://www.bedrock-tech.com/wp-content/uploads/2010/05/wp_key-considerations.pdf

[6] W. Xia et al., “A comprehensive study of the past, present, and future of data deduplication,” Proceedings of the IEEE, vol. 194, no. 9, pp. 1681-1710, 2016

[7] G. Xu, B. Tang, H. Lu, Q. Yu and C. W. Sung, "LIPA: A Learning-based Indexing and Prefetching Approach for Data Deduplication," 2019 35th Symposium on Mass Storage Systems and Technologies (MSST), Santa Clara, CA, USA, 2019, pp. 299-310, doi: 10.1109/MSST.2019.00010.

[8] Zhang Y, Xia W, Feng D, et al. Finesse: Fine-grained feature locality

based fast resemblance detection for post-deduplication delta compression[C]//17th {USENIX} Conference on File and Storage Technologies ({FAST} 19). 2019: 121-128.

[9] Cao Z, Wen H, Wu F, et al. {ALACC}: Accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching[C]//16th {USENIX} Conference on File and Storage Technologies ({FAST} 18). 2018: 309-324.

[10] B. Zhu, K. Li, and R. H. Patterson, ”Avoiding the Disk Bottleneck in the Data Domain Deduplication File System,” in Proc. USENIX FAST, 2008.

[11] M. Lillibridge, K. Eshghi, D. Bhagwat et al., ”Sparse indexing: Large scale, inline deduplication using sampling and locality.” in Proc. USENIX FAST, 2009.

[12] D. Bhagwat, K. Eshghi, D. D. Long et al., ”Extreme binning: Scalable, parallel deduplication for chunk-based file backup,” in Proc. IEEE MASCOTS, 2009

[13] L. Zhou, ”A survey on contextual multi-armed bandits”, arXiv preprint arXiv:1508.03326

[14] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, Y. Zhang, and Y. Tan, “Design tradeoffs for data deduplication performance in backup workloads,” in Proc. USENIX Conference on File and Storage Technologies (FAST’15). Santa Clara, CA, USA, 2015, pp.331-344

[15] SHILANE, P., HUANG, M., WALLACE, G., AND ET AL. WAN optimized replication of backup datasets using stream-informed delta compression. In the 10th USENIX Conference on File and Storage Technologies (FAST’12) (San Jose, CA, USA, 2012), USENIX Association, pp. 49–63

[16] Mark Lillibridge, Kave Eshghi, and Deepavali Bhagwat. Improving restore speed for backup systems that use inline chunk-based deduplication. In 11th USENIX Conference on File and Storage Technologies (FAST 13), pages 183–198, 2013