|  |  |
| --- | --- |
| **题目：** | 消除器：重复数据删除 |

|  |  |
| --- | --- |
| **学 院:** | 计算机科学与技术 |
| **专 业:** | 计算机技术 |
| **学 号:** | 2017282110227 |
| **学生姓名:** | 胡咏 |

**一、估计重复数据删除率**

在备份世界蓬勃发展多年之后，重复数据删除已经成为主要存储的关键技术。随着全闪存存储系统比旋转磁盘成本更高，随机读取性能更好，通用的重复数据删除和数据还原功能比以往任何时候都更有意义。结合现代虚拟环境的流行和高度重复性，整合重复数据为这种高端存储系统带来了巨大的利益。这种趋势势必会继续出现新的存储级备忘录，预计随着闪存的不断上涨和更高的每GB成本。

估计一个非常大的数据集的重复数据删除率是非常有用的，但真正很难执行。如何估计给定数据集的重复数据删除优势？潜在客户需要这些信息才能做出明智的决策，确定高端存储与删除是否值得。更重要的是，规模和容量规划的问题与具体数据预期的重复数据删除效果息息相关。事实上，所有的重复数据删除解决方案供应商都面临着这个问题，不幸的是没有简单的解决方案。

这个困难源于重复数据删除是一个全球性的事实，因此需要搜索大量的数据。为了得到准确的估计，需要从磁盘读取大部分数据。相反，压缩是一个局部过程，因此可以非常有效地解决压缩估计问题。

市场上现有的解决方案采取以下两种方法之一：第一种方法是根据先前的知识和基于手头工作量的信息进行有根据的猜测。举例来说：虚拟桌面基础架构（VDI）环境能提供平均的0.16重复数据删除率（减少1：6）。但实际上，根据特定的环境，结果可以在0.5到0.02的重复数据删除率（1：2和1：50之间）之间变化。因此，使用这种模糊的估计大小是非常不准确的。

另一种方法是对手头数据进行全面扫描。实际上，一个典型的用户可以对尽可能多的数据进行全面扫描，并获得准确的估计，但仅限于扫描的数据。这种方法不是没有挑战，因为评估扫描数据集的重复数据删除率需要大量的内存和磁盘操作，通常远高于估算扫描所允许的数量。因此，对这个主题的研究集中在获取低内存需求的精确估计上，同时仍然从磁盘读取所有数据（并且在所有数据上计算哈希值）。

随着技术的普及，过去几年估计重复数据删除的问题已经出现。但是，这个问题与计算机科学中一个长期存在的问题直接相关，那就是估计大量人群中不同元素的数量。从生物学（估计物种数量）到数据库（表格/列中不同的值）的动机，这个问题受到了很大的关注。有一大堆启发式统计估计器，但是这些统计估计器的准确性并没有严格的保证，而且大多数目标集的元素数量相对较少。他们的实证检验在长尾巴分布上表现很差（其中大部分数据的重复计数较低），这是删除常见的情况。图1显示了不准确的启发式估计如何可以在真实的数据集上的例子。它还显示了样本的重复数据删除率与整个数据集的重复数据删除率有多远。

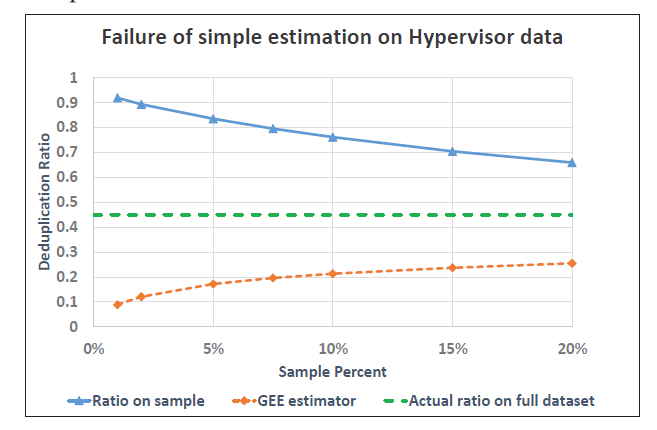


图1：当前采样失败的一个例子

方法：每个图表描述了重复数据删除率作为采样百分比的函数。 这些点是100个相同百分比的随机样本的平均值。 我们看到，仅仅考虑样本的比率给出了一个非常悲观的估计，而在这个例子中的估计总是过于乐观。

这个经验性的困难也得到了理论下界的支持，证明了一个准确的估计需要扫描大部分的数据。因此，大量关于不同元素的工作集中在数据流的低内存估计上。这些估算方法需要对数据进行全面扫描，并为前几节中提到的用于重复数据删除估计的低内存扫描奠定了基础。

2011年，在一个突破性的论文中，Valiant表明，至少必须检查Ω（lognn）的数据，并且还有一个匹配的上界。也就是说，他们展示了一个理论算法，如果至少检查了O（lognn）个元素，就可以达到可证明的准确度。

随后，这个算法的变化也由同一个作者实现[18]。值得注意的是，Valiants的工作，名为“估计看不见的”更为笼统，而不仅仅是截然不同的元素估计，可以用来估计其他的度量，如数据的熵。

在理论和实践之间往往不是直截了当的，估计重复数据删除的情况是明确的。对此，Danny Harnik提出了一种新的方法来准确估计去重效益，比现有技术的运行速度快3倍到15倍。改进的程度取决于数据本身及其驻留的存储介质。该技术基于Valiant从2011年开始的突破性的理论工作，为仅仅看到一小部分数据提供了一种可靠的方法来估计各种措施。但是，对于重复数据删除估算的使用情况，将这一理论付诸实践却遇到了重大障碍。在这项工作中，他们发现了新的解决方案和新颖的技术，发现了评估估计准确性的新颖方法，运行内存消耗低的技术，评估组合压缩和重复数据删除比率的方法以及在实际存储系统中执行实际采样以实际获得好处的方法这些算法。并在许多现实世界数据集上作出了实践。在这项工作中，Danny Harnik解决了尝试在实际环境中成功使用这种新技术时出现的许多挑战。对于初学者来说，以小粒度进行随机采样对于基于HDD的系统中的完整顺序扫描有很大的好处。但是还有一些更深层次的问题需要解决，以便从新的方法中获益。

以下是主要主题和Danny Harnik的解决方案的概述：

了解估计的准确性。 Unseen算法的准确性证明本质上是渐近的，并且不会转化为具体的实际数字。此外，它们提供了最糟糕的情况分析，并且不能保证数据集更容易分析。因此，没有真正的方法可以知道要采样多少数据，以及哪一部分实际上足够。在这项工作中，Harnik提出了一种新的方法来衡量算法的准确性。他们的技术不是返回一个估计，而是输出一个预期实际结果的范围。这在许多方面是可行的，特别是允许逐步执行：首先取一个小样本并评估其结果，如果范围太大，则继续增加样本大小。虽然他们的测试表明，15％的样本足以对所有工作负载进行较好的估计，但是一些真实生活中的工作负载可以达到一个足够好的估计，样本小到3％，甚至更少。使用他们的方法，可以在达到足够的估计时提前停止。

算法的内存消耗。在实际系统中，能够以较小的内存占用量进行估算并且不需要额外的磁盘IO是非常需要的，并且在某些情况下是必须的。问题是，只要按照规定运行Unseen算法，就需要映射并计算样本中所有不同的块。在重复数据删除的使用情况下，这个数字可能非常大，与整个数据集中块的数量相同。请注意，当在分布式系统中执行估计时，低内存使用率也有利于较低的通信带宽。用于不同元素的低内存估计的现有解决方案不能与Unseen算法直接相结合。相反，它们需要一些修改和小心的组合。他们提出了两种这样的方法：一种具有更严格的准确性，另一种失去更多的估计紧密度，但对分布式和自适应设置更好。在这两种情况下，整个算法都可以运行，只有10MB的内存。

结合重复数据删除和压缩。许多部署重复数据删除的系统也使用压缩来补充数据减少。在多个作品中表明，为了实现所有工作负载的数据减少，这种组合非常有用。估计综合效益的一种自然方法是分别估计每一个，并将两者的比率相乘。但是，如果重复数据删除的有效性和压缩效率是独立的，这在某些情况下是不正确的，这只会产生正确的结果。他们提出了一种方法来压缩到Unseen算法的压缩，并显示它会产生准确的结果。

如何进行抽样？如上所述，在基于HDD的系统中（在一些情况下，只有2％的采样可能已经花费比全扫描多），以小粒度（例如4KB）执行直接采样是非常昂贵的。相反，他们采取大量的“超级块”（1MB）进行采样，并以分类的方式进行读取。这样的采样比全面扫描运行速度快得多，这是时间收益的主要来源。

同样重要的是，他们的方法可以被调整为在这个限制下给出正确和可靠的估计（以略微宽松的估计范围为代价）。他们还提出了一个总体的抽样策略，要求低内存，产生排序的非重复读取，并可以以渐进的方式运行（例如，如果我们要先读取5％的样本，然后将样本放大到10％）。

结果总结。总之，他们设计和评估了一个估算大数据集中去除率的新方法。他们的策略利用了不到10MB的RAM空间，可以分配，并且可以准确地估计压缩和重复数据删除（以及它们各自的优点）的联合优势。由此得出的估计值表示为实际比率预期存在的范围（而不是单个数字）。这允许一种渐进的估计模式，在这种模式下，可以对一小部分进行采样，对其进行评估，如果结果范围过宽，则继续采样。

请注意，实际估计算法的执行时间与扫描数据所花费的时间相比可以忽略不计，因此能够以小的采样率停止对于实现整体时间改进至关重要。

他们评估一些实际工作负载的方法，并验证其高准确度。 总的来说，他们的方法至少比现有扫描技术提高了3倍的时间。 时间的改善根据数据和存储数据的介质而变化，并且可以达到15倍以上的时间改进。

**二、OrderMergeDedup：Flash上高效，失败一致的重复数据消除**

闪存在移动设备，传感器和云服务器上很常见。 I / O重复数据删除对于节省存储空间并降低高昂的闪存写入是有益的。 I / O重复数据删除被广泛用于节省存储空间和I / O负载。 I / O重复数据消除对于数据中心的存储服务器以及个人设备和现场部署的感知系统都是有利的。在软件包安装过程中或通过频繁使用SQLite事务处理，智能手机和平板电脑上的Flash写入可能会发生。在网络物理系统中，大量的数据可能被现场部署的摄像机捕获，并为智能交通等应用进行存储/处理。

重复数据删除系统维护元数据，例如逻辑到物理块映射，物理块引用计数器，块指纹等。这些元数据和数据结构在系统故障后必须保持一致。为了满足存储耐久性的要求，还需要坚持写入。在闪存上，I / O重复数据删除还必须最大限度地减少由元数据管理导致的昂贵的Flash写入。

Zhuan Chen 和Kai Shen出了一种新的方法—OrderMergeDedup，它可以在保持同步的同时实现失败一致性，效率和持久性的同时，对存储写入进行重复数据删除。他们设计了一种软更新式元数据写入顺序，可以保持存储数据的一致性，而不会产生一致性引起的额外I / O。他们进一步探索I / O延迟和合并的机会，以减少元数据I / O写入。

特别地，他们对重复数据和数据进行了仔细的排序，以便任何故障停止失败都不会产生与未收集的垃圾相比的其他数据不一致(在持久存储上)。 这种方法在概念上类似于基于软件更新的文件系统设计。通过不需要额外的I / O流量来保持一致性（与记录/日志记录或基于阴影的I / O原子性相反），效率是非常高的。虽然原始文件系统软更新遭受依赖周期和回滚，但相对简单的重复数据存储结构使我们能够识别和移除所有可能的依赖周期，而不会影响性能。

通过合并共享公用重复数据删除元数据块的多个逻辑写入，可以进一步降低元数据I / O开销。特别是，我们在不久的将来会预期元数据I / O合并机会，从而延迟一些I / O操作。如果延迟的I / O由于数据持久性语义而等待，则预防性I / O延迟和合并可能延长对用户的响应。我们表明，当延迟时间被限制时，性能影响是轻微的。由于减少了I / O负载，它甚至可以改善应用程序延迟。

I / O重复数据消除消除了I / O流中的重复写入。他们捕获设备层上的所有写入I / O块以进行重复数据删除。采用固定大小的分块方法，每个4 KB传入数据块被截取，并从其内容计算哈希指纹。根据现有存储块的指纹查找该指纹以识别重复。

重复数据删除系统维护额外的元数据信息。具体而言，逻辑到物理块映射将逻辑块访问（具有其逻辑地址）引导到其存储上的物理内容。对于每个物理块，相关联的参考计数器记录映射到其的逻辑块的数量，并且计算指纹以促进块内容匹配。重复数据删除系统收到的写入请求可能会导致对数据块和元数据的一系列物理写入。对于重复数据删除元数据管理，实现（1）故障一致性（数据/元数据写入必须仔细执行以实现故障之后快速一致的恢复）（2）效率 - 由元数据写入引起的额外I / O成本不应该显着减少重复数据删除I / O保存; （3）持久性 - 重复数据删除层不应过早地返回违反持久性语义的I / O写入。

文件和存储系统已经认识到原子I / O的重要性，以支持一致的故障恢复。现有的技术包括日志记录，shad-due和软更新。在日志中，在写入文件系统之前，将原子I / O操作记录在重做日志中。通过运行重做日志可以在系统重新启动时恢复部分写入失败。在投影中，对现有文件的写入以写入时拷贝的方式处理为临时阴影块。最后的提交是通过一个原子I / O写入文件索引块来实现的，该块指向更新的影子数据/索引块。必须重新编写索引块（可能在多个层次级别）以创建完整的阴影。日志和阴影都需要额外的写入I / O来实现持久数据的失败一致性。

软更新方法小心地命令文件系统操作中的写操作，使得任何中间操作故障总是使文件系统结构保持一致状态（除了临时写入块上可能的空间泄漏之外）。虽然在正常操作过程中不需要I / O开销，但回滚可能需要在块提交或解析中解决循环依赖关系。 Seltzer等人[18]表明，这种回滚有时导致在UNIX快速文件系统上软性更新性能不佳。由于重复数据删除存储的相对简单的语义（与文件系统相比），他们表明仔细设计所有重复数据删除I / O路径可以有效地解决可能的依赖关系周期。接下来我们将展示我们的软更新式重复数据删除设计。

他们设计的一个独特之处在于他们的物理块参考计数器对逻辑块参考进行计数，并对物理块的指纹进行参考。因此，块指纹的回收不必与删除引用物理块的最后逻辑块一起发生。将它们分成两个失败一致的事务使得每个事务都不那么复杂，并且减少了循环依赖写入顺序的可能性。它还允许指纹回收被延迟 - 例如，周期性地离线执行。懒惰的指纹回收可以提高性能，因为在一段不存在期间重写的相同数据仍然可以被重复数据删除。这种情况已经被证明在某些工作量中发生[16]。

具体而言，他们在重复数据消除期间的I / O操作之间保持以下顺序。

1.物理数据块应始终在与逻辑地址或计算的指纹链接之前保持不变。故障恢复可能会使某些数据块无法访问，但不会导致指向不正确内容的任何逻辑地址或指纹。

2.对于参考计数器，他们保证当突发故障发生时，唯一可能导致一致性的是一些物理块的高于实际的参考计数器。高于实际值的参考计数器可能会产生垃圾（可以异步重新申报），而低于实际值的参考计数器会导致过早的块删除严重损失。为了达到这个目的，从某个逻辑地址或指纹指向一个物理块的一个新的链接必须在物理块的引用计数器的增量之前，并且相应的解除链接操作必须在物理块的引用计数器的递减之前。

3.同时，逻辑到物理块映射和指纹的更新可以并行处理，因为它们之间没有故障一致的依赖关系。

图1说明了他们在不同写入条件下的完整软更新式写入顺序。

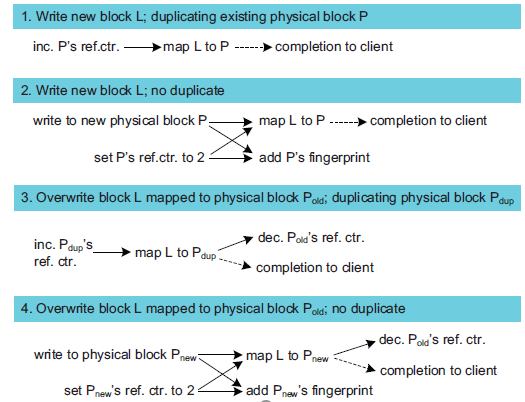


图1：不同写入条件下的失效一致重复数据删除写入顺序（新写/覆盖，重复识别与否等）。 实箭头表示写入的顺序。 每种情况下的虚线箭头显示完成信号何时发送到客户端。 请注意，一个新的物理块的参考计数器从第一个逻辑块的两个引用开始，另一个从指纹开始。

块元数据项比块本身小得多（例如，他们使用物理块索引的64位地址和8位参考计数器1）。如果将属于同一个元数据块的多个元数据写入合并到一个I / O操作中，则可以降低I / O成本。合并机会出现在元数据写入，只要他们不受任何排序约束。

虽然元数据写入合并为I / O减少带来明显的好处，但合并可能会造成附加的写入顺序限制，并导致循环依赖或死锁。例如，对于图1中的情况3和4，如果新块Pdup / Pnew和旧块Pold的引用计数器在同一个元数据块上，那么引用计数器更新的合并将导致死锁。对于这种情况，不允许元数据写入合并。

循环依赖性可以防止元数据写入合并，从而降低成本并且使实现复杂化。他们通过延迟循环依赖关系中涉及的非关键元数据更新来解决此问题。非关键更新是客户完成信号不依赖的写入操作，特别是对于上述相同的示例，图1的情况3和4中的Pold的参考计数器的减少和后续工作。这些操作的延迟，直到它们相关的依赖被清除，简单地消除了循环依赖。由于延迟动作不在I / O完成的关键路径上，客户端I / O响应不会受到影响。幸运的是，在他们的软更新式重复数据删除元数据管理下，所有潜在的死锁都可以通过这种方式解决。

1引用计数器溢出会导致分配另一个新的物理块来保存相同的块内容。 后来写这样的内容将被映射到新的块。

如前所述，大量的I / O成本降低可能会归因于归入相同元数据块的多个元数据写入。我们可以通过延迟元数据更新I / O操作来增加元数据I / O合并的机会，从而增加与未来元数据写入合并的机会。

弱持久性他们的重复数据删除系统支持两种具有不同性能权衡的持久性模型。具体而言，强大的持久性模型完全保留了底层设备的持久性支持 - 只有在从设备返回所有相应的物理I / O操作之后，重复数据删除层才会返回I / O操作。另一方面，一个弱持久性模型在异步写入的情况下执行写操作，而相应的物理I / O操作可以延迟到下一个刷新或强制单元访问（FUA） - 挂起的请求。

在持久性较弱的情况下，I / O操作可以被积极地分解以呈现元数据I / O合并机会。然而，这样的延迟可能会受到某些应用程序和数据库中的同步写入操作的阻碍。

非关键I / O延迟显示了通过延迟一些非关键元数据更新来消除循环依赖关系。实际上，如果延迟持续时间足够长，以便与位于同一个元数据块中的将来的元数据写入合并，那么这些更新可以是免费的。而且，这适用于所有非关键的元数据写入。他们也积极推迟P / P new的指纹插入操作。突然的系统故障可能会使一些参考计数器高于实际值，导致未复制的垃圾，或丢失物理块重复删除机会的一些指纹，但不会发生其他严重的不一致情况。

预期I / O延迟和合并如果两个元数据执行的间隔时间比重复数据删除系统写入物理设备的典型周期长，那么两个元数据写入同一个物理块通常会导致单独的设备提交。如果这样的间隔很小，则可以给物理设备施加一个短暂的空闲时间（通过停止向其写入数据）来产生更多的元数据写入合并机会。这让人联想到I / O预期调度，它被认为是一种用于机械磁盘的性能增强寻道减少技术。在他们的情况下，暂时闲置物理设备，以期望即将到来的可用请求将元数据更新与现有设备合并。

预期的I / O延迟和合并可以在高密度的写入请求下获得高收益，因为短暂的预期器件空闲时间会产生高度的合并。另一方面，轻负载几乎没有合并的机会，所以预期的设备空闲只能延长I / O响应延迟。为了最大限度地利用预期的I / O延迟和合并，他们应用一个简单的启发式提示作为指导 - 重复数据删除系统收到的传入写入请求的频率。直观地说，如果空闲期可以覆盖多个传入的写入请求，则可能发生元数据写入合并。在这种情况下，他们只启用预期的I / O延迟和合并。

他们在通用操作系统块设备层实施了重复数据删除机制，为软件应用程序提供透明的全设备重复数据消除功能。具体而言，他们的机制在Linux 3.14.29内核中作为自定义设备映射器目标来实现。他们的实现遵循开源Dmdedup框架中用于重复数据删除检查的基本块读取/写入接口。

重复的4 KB I / O块通过散列指纹来识别。他们使用广泛采用的SHA-1密码散列算法来产生160位（20字节）块指纹。 SHA-1哈希是抗colLision ，我们认为两个块作为重复如果他们有匹配的指纹，而不进行全块比较。这是在重复数据删除中被广泛接受的做法，因为两个不同块之间的哈希冲突的机会是可以忽略的 - 小于内存和网络传输的错误率。一个哈希表维护在内存中组织鳍印。根据物理数据块的总数，将指纹值空间划分为N个区段，并将每个指纹f映射到对应的区段（f mod N）。

为了简单起见，他们将存储区中的元数据块组织为线性表。未来可能的增强是使用基数树结构。基数树分层写入可以并入失败一致的写入顺序而不引入循环依赖。

文件系统保持关键信息的冗余持久副本，例如超级块的可靠性。对于Ext4文件系统，超级块和块组描述符的多个副本保留在文件系统中，而主副本位于前几个块。 去重复这些块可能会损害这种面向可靠性的冗余度量。 他们采用一种简单的方法来防止重要的文件系统信息（具有可识别的块地址）的主副本的重复数据删除。 具体而言，他们不会在重复数据删除的高速缓存中保留它们的印记。 他们也不试图去重复写一个这样的块。 未来一个可能的增强是基于直接从文件系统传递的提示来协助这些决策。

在他们的实现中，在清除失败一致的依赖关系（在此期间，它们可能会与其他在同一个元数据块中的传入元数据更新合并）之后，他们将非关键元数据写入延迟30秒。他们选择1毫秒的空闲时间来进行预期的I / O延迟和合并，这与他们实验平台的闪存写入延迟相同。他们的删除系统默认配置了弱持久性模型。为了在性能和持久性之间取得更好的平衡，他们定期每隔1秒向物理设备提交分层的元数据写入（除同步刷新或FUA标记的请求和非关键元数据写入外）。这与Linux中其他设备映射程序目标支持的设置相同（例如，dm-cache）。当数据持久性至关重要时，他们的重复数据删除系统也支持强大的持久性模型。

他们使用多种移动和服务器工作负载（包安装和更新，BBench网页浏览，车辆计数，Hadoop和Yahoo云服务基准）评估他们的基于Linux设备映射器的实施。

结果显示OrderMergeDedup可以实现写入内容重复度为23-73％的工作负载的写入减少18-63％。还节省高达56％的空间使用率。在支持强大的I / O持久性模型时，预期的I / O延迟优化对于增加元数据合并机会特别有效。与替代的基于I / O shadowing的重复数据删除相比，它的元数据写入开销要小得多。他们的方法对应用程序延迟有轻微的影响，甚至可能由于减少I / O负载而改善性能。

1. **CacheDedup：Flash缓存的联机重复数据删除**

闪存缓存采用基于闪存的存储器作为基于DRAM的主存储器和存储系统的典型I / O堆栈中基于HDD的主存储器之间的缓存层，以利用I / O中固有的局部这层和提高应用程序的性能。近年来备受关注[7,5,2,23]，这可以归结为两个重要的原因。首先，随着整合水平（就整合到单个主机的工作负载数量和统一到单个存储系统的主机数量而言）在典型的计算系统（如数据中心和云）中继续增长，存储系统的可扩展性成为一个严重的问题。其次，基于闪存的存储设备的高性能使得闪存缓存成为解决这种可扩展性问题的有希望的选择：通过使用缓存数据为I / O提供服务，可以减少主存储上的负载并提高工作负载性能。

然而，闪存有效缓存有几个关键限制。首先，随着现代工作负载数据密度的不断提高以及系统中并发工作负载的数量的增加，相对于商用闪存设备容量的有限性，缓存容量的需求急剧上升。其次，由于闪存会随着写入而磨损，所以使用闪存进行缓存会加重问题，因为工作负载的固有写入以及错过缓存的读取都会导致磨损。

CacheDedup是一种使用在线重复数据删除解决这些限制的解决方案。首先，重复数据删除减少了工作负载的缓存占用空间，从而允许缓存更好地存储其工作集并减少容量缺失。其次，重复数据删除技术减少了由强制缺失和容量缺失导致的必要的缓存插入次数，从而减少了闪存磨损并提高了缓存的耐用性。虽然重复数据删除技术已经在包括基于闪存的主存储在内的各种存储系统中进行了研究，但文章讨论了以重复数据删除和高速缓存整合的有效整体方式所面临的独特挑战。

高效的缓存重复数据删除需要缓存和重复数据删除管理的无缝集成。为了满足这种需求，CacheDedup体现了一种新颖的体系结构，它使用独立的数据缓存和元数据缓存来集成数据缓存和重复数据删除元数据（数据的源地址和指纹）。这个设计解决了两个关键问题。首先，它允许CacheDedup绑定元数据的空间使用情况，使其足够灵活，可以部署在存储系统的客户端或服务器端，并以软件或硬件实现。其次，它可以在数据从数据高速缓冲存储器中逐出之后优化在元数据高速缓冲存储器中高速缓存历史源地址和指纹。这些历史性重复数据删除元数据允许CacheDedup使用缓存的指纹快速识别重复，并在再次引用这些源地址时生成缓存命中。

基于这种架构，Wenji Li进一步研究了能够利用重复数据删除技术来提高闪存缓存性能和耐用性的重复感知缓存替换算法。首先，他们介绍D-LRU，LRU的重复感知版本，可以通过在数据和元数据高速缓存上强制执行LRU策略来有效实现。其次，他们提供了D-ARC，一个ARC的重复感知版本，利用ARC的抗扫描能力来进一步提高缓存性能和耐用性。对于这两种算法，他们还从理论上证明，它们不会导致数据和元数据缓存中的浪费，并且可以有效地利用它们的空间。

CacheDedup是在块设备虚拟化的基础上实现的，可以透明地部署在现有的存储系统上。他们使用真实世界的踪迹对其进行评估，包括FIU踪迹以及从一组Hadoop VM收集的自己的踪迹。结果显示CacheDedup通过将缓存缺失率降低高达20％，I / O延迟降低51％，写入闪存达89％，大大优于传统的缓存替换算法（LRU和ARC）。它可以有效地在共享缓存的工作负载和多个工作负载内重复数据删除。我们还使用fio基准测量了CacheDedup的开销，这表明通过开销的开销可以忽略不计，并且fin-gerprinting的延迟开销可以与并发I / O操作重叠，并且受到命中率重复数据删除。就空间开销而言，CacheDedup需要<4％的闪存缓存来存储重复数据删除元数据，以便我们的算法达到最佳性能。

CacheDedup是研究缓存重复数据删除重复数据缓存管理的首批研究者之一。与相关工作相比，也考虑了服务器端闪存缓存技术，他们的方法可以自然扩展，以支持复制和压缩感知缓存管理，它可以提高读取命中率12.56％。他们的方法并不是基于闪存的缓存，它只利用闪存设备与HDD相比更快的速度，与DRAM相比更大的容量。因此，也适用于DRAM和较慢的二级存储器之间用作缓存层的其他非易失性存储器技术。尽管新技术可能具有更好的耐用性，但与NAND闪存相比，它们可能具有相当有限的容量，而CacheDedup仍将大大受益，CacheDedup可以大幅降低缓存占用量，并将写入发送到缓存设备。

CacheDedup无缝集成了缓存和重复数据删除的管理，并重新设计了这两个功能需要的关键数据结构。传统的缓存需要管理从主存储上的块的源地址到缓存设备上的块的缓存地址的映射。重复数据删除层需要跟踪数据块的指纹，以便识别重复的数据块。在这些数据结构的设计中，将缓存和重复数据删除引起了一些独特的问题。

首先，与传统缓存不同，CacheDedup中的源到缓存地址映射的数量不受缓存大小的限制，因为在重复数据删除的情况下，这两个地址空间之间现在存在多对一的关系。其次，即使高速缓存必须跟踪的指纹数量受到高速缓存大小的限制，CacheDedup还有一个重要的原因是跟踪超出当前存储在高速缓存中的数据的指纹。具体而言，保留已经从高速缓存中逐出的块的历史指纹是有益的，以便当再次请求这些块时，CacheDedup不必从主存储器中取出它们以确定它们是否与当前缓存的块。当CacheDe-dup被用作客户端缓存时，这样的优化尤其重要，因为它可以减少昂贵的网络访问。但是，指纹存储仍然需要遵守CacheDedup的空间使用限制。

为了解决这些问题，他们提出了一个新的Metadata Cache数据结构来缓存源地址及其指纹。这种设计使我们能够解决这些元数据的管理问题，将其视为缓存替换问题，并将其与存储数据块的数据缓存（图1）分开考虑。元数据缓存包含两个关键的数据结构。源地址索引将主存储器的源地址映射到元数据高速缓存中的指纹。每个缓存的源地址都与高速缓存的指纹关联，并且由于重复数据删除，多个源地址可能映射到相同的指纹。指纹存储区映射指纹以阻止数据高速缓存中的地址。它也包含历史指纹，其相应的块当前没有存储在数据高速缓存中。当历史指纹所指向的数据块被带回到数据高速缓存时，映射到该指纹的所有源地址在再次被引用时可以产生高速缓存命中。每个指纹都有一个引用计数来指示包含相同数据的源块的数量。当它下降到零时，指纹将从元数据缓存中移除。

解耦的元数据高速缓存和数据高速缓存为我们的重复感知算法提供单独的控制“旋钮”来优化高速缓存管理。这些算法通过将其替换策略应用于元数据缓存并利用缓存的历史指纹来机会地提高读取性能来限制元数据空间使用。这两个缓存可以永久存储在闪存设备上，以容忍失败。而且，其架构使得闪存空间可以灵活地在两个缓存之间进行分区。对于给定的数据高速缓存大小，最小的元数据高速缓存大小是存储所有高速缓存数据的元数据所需的空间。他们使用真实世界的评估的评估表明，“最小”尺寸足够小，在实践中不成问题。更重要的是，元数据缓存可以通过从数据缓存中拿走一些空间来存储更多的历史指纹并可能提高性能。

基于上面讨论的体系结构，CacheDedup的一般操作如下：（必要的缓存替换是由重复感知算法所管理的。）

读。在元数据高速缓存中发现其指向数据高速缓存块的指纹的源块地址的读取在数据高速缓存中是命中的。否则，这是一个小姐。请注意，读取可能与元数据高速缓存中的历史指纹匹配，该历史指纹不映射到数据高速缓存中的任何数据块，并且仍然是未命中。在未命中时，所请求的数据块从主存储器中取出，并且如果它不是任何现有的高速缓存数据块的重复，则将其插入到数据高速缓冲存储器中。如有必要，将相应的源地址和指纹插入到元数据高速缓存中。如果指纹已经存在于元数据缓存中，则通过指向新的数据缓存块来“重新启动”，并且指向该指纹的所有历史源地址也被“恢复”，因为它们在被访问时可以产生命中再次。

写。这些步骤因CacheDedup支持的各种写策略而不同：

1.写入无效 - 所请求的源地址和该地址的缓存数据在元数据和数据高速缓存（如果存在的话）中是无效的。写入直接到主存储。

2.通过写 - 写入存储在缓存中，同时提交给主存储器。如果源地址和指纹不在那里，则将其插入到元数据缓存中。如果写入中包含的数据与现有的缓存块匹配，则不需要对数据缓存进行更改，而其指纹的引用计数会加1。如果此块的前一个数据已经在数据缓存中，则先前指纹的引用计数减1。

3.回写 - 写入仅存储在高速缓存中，稍后提交到主存储器，块被清除或脏数据总量超过预定义阈值时。除了不立即将写入提交到主存储器之外，这些步骤与直写式策略相同。

闪存的非易失性允许缓存数据在缓存的主机崩溃时保留，但为了恢复缓存的数据，其元数据也需要一直存储。如果目标是为了避免数据丢失，则只有使用回写策略的本地修改数据的源到缓存地址映射必须是持久的。 CacheDe-dup同步提交元数据和数据到缓存设备，以确保一致性。如果目标是在主机重新启动之后避免热身，则整个元数据缓存（包括源块索引和指纹）将保持一致。使元数据缓存持久化的时间开销不如闪存设备上的空间开销那么重要，因为干净块的元数据可以批量和异步方式写入。对于故障容错目标，元数据缓存也保存在主存储器中以加速缓存操作，并且其内存使用是有限的。最后，如果目标是容忍闪存器件故障，则需要采用其他机制。还可以利用相关工作来提供更好的闪存缓存一致性。

CacheDedup可以部署在存储系统的客户端和服务器端：客户端CacheDedup可以通过隐藏高网络I / O延迟来更直接地提高应用性能，而服务器端CacheDedup可以使用多个I / O客户端实现更高的数据还原水平。当共享数据的多重客户端使用CacheDedup时，需要缓存一致性协议来确保每个客户端都拥有共享数据的一致视图。虽然这不是本文的重点，但Cache-Dedup可以直接扩展已深入研究的缓存协作协议，以同步数据缓存中的数据和元数据缓存中的指纹，从而确保一致性客户。

CacheDedup是第一个使用重复感知缓存管理将重复数据删除与闪存缓存集成在一起的研究。其新颖之处在于无缝集成了元数据和数据缓存的新架构设计，以及新的高速缓存替换算法D-LRU和D-ARC，可以同时优化性能和耐用性。文章通过理论分析和实验验证，对这些算法进行了深入的研究，证明了它们的无缓存损耗性能，并且显示出缓存命中率，I / O延迟，以及写入发送到缓存设备。在这两种算法之间，D-ARC达到了最佳的性能，D-LRU因其简单性而具有吸引力。两者在时间和空间使用方面都是高效的。高速缓存重复数据删除是一个多功能框架，用于启用各种算法，包括一个（CD-ARC），通过重复数据删除技术改进压缩的使用。由于其设计不是针对闪存设备的，因此他们认为CacheDedup方法也可以应用于新的非易失性存储器技术，并在用于缓存时提高其性能和耐用性。

1. **使用提示来改进内嵌块重复数据消除**

组织存储的数据量正在迅速增长。硬盘和固态硬盘价格的下降无法弥补这一增长。结果是企业在存储上花费越来越多。降低存储成本的一项技术是重复数据删除技术，允许网站存储更少的原始数据。在其核心，重复数据删除系统地用引用替换重复的数据块。对于许多真实世界的数据集，重复数据删除显着降低了原始存储使用率。

重复数据删除可以在存储栈中的多个层次上实现。大多数现有的解决方案都建立在文件系统中，因为它们有足够的信息来高效地进行重复数据删除而不会影响可靠性。例如，文件大小，元数据和磁盘布局对于文件系统是已知的;通常文件系统意识到执行I / O的进程和用户。可以利用该信息来避免重复删除某些块（例如，元数据），或者预取去重元数据（例如，对于可能一起访问的块）。

另一种方法是将重复数据删除添加到块层，它提供了一个简单的读/写接口。因为这种简单性，添加功能到块层比改变文件系统更容易。这种观察同样适用于直接在块层工作的系统，如数据库和对象存储。

但是，块级重复数据删除系统并不知道其操作数据的上下文。典型的I / O请求仅包含操作类型（读取或写入），大小和偏移量，而不包含元数据和用户数据之间的差异等附加语义。 （1）损害可靠性，例如，因为许多文件系统故意保存关键数据的一些副本，如超级块，（2）浪费计算资源，因为典型的元数据（inode表，目录条目等）表现出低冗余度。特别是，在线重复数据删除是昂贵的，因为在将数据写入磁盘之前执行形成块（固定长度或可变长度），哈希计算和哈希搜索;不希望将资源花费在不能从重复数据消除中受益的数据上。

为了允许块层重复数据删除考虑到上下文，Sonam Mandal提出了一个接口，允许文件系统和应用程序提供关于执行重复数据删除的上下文的简单提示。这样的提示只需要较小的文件系统更改，使它们可以添加到现有的成熟的文件系统中。他们实现了两个提示：NODEDUP和PREFETCH，在很多情况下都是有用的。

为了评估提示的潜在好处，他们使用了开源的块层重复数据删除系统dmd-edup。 Dmdedup适用于内联主存储重复数据删除，并在Linux内核中作为可堆栈块设备实现。我们在各种工作负载下对我们的提示进行了评估，并对独特数据与可消除重复数据进行了组合。他们的实验表明，通过NODEDUP提示在现代机器上加速应用程序的速度高达5.3倍，因为重复数据删除的开销在不太可能是有益的时候可以避免。他们还表明，通过预加载哈希数据，可能很快将被重复数据删除，PREFETCH提示可以通过为可能很快访问的数据缓存哈希来将应用程序加速至1.8倍。

为了让块层知道上下文，我们取消了一个让存储栈中的提示从较高层到较低层流动的系统。然后，应用程序和文件系统可以将关于其数据的重要信息传送到较低层。图2中的红色箭头显示了如何将提示传递给块图层。我们已经实现了两个重要的提示：NODEDUP和PREFETCH。

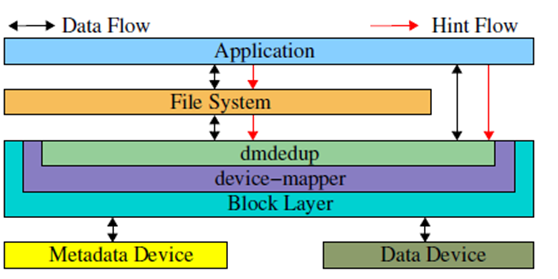


图2：整个存储层的提示流程

Nodedup。由于重复数据消除使用计算资源并可能增加延迟，因此只能在有潜在利益的情况下执行。 NODEDUP提示指示块层不要在写入时重复删除特定的块（块）。它有两个用例：

（1）独特的数据：在重复数据不浪费的情况下，浪费资源是不可能有重复的，例如传感器或加密数据; （2）可靠性：保持某些块的多个副本可能是必需的，例如，在许多文件系统中的超级块副本。

预取。重复数据消除系统中最耗时的操作之一就是哈希查找，因为它十年后需要额外的I / O操作。更糟糕的是，哈希本质上是随机分布的。因此，查找哈希通常需要随机I / O，这在大多数存储系统中是最慢的操作。而且，正如之前的研究[38]所显示的，将所有哈希值保存在内存中是不切实际的，因为哈希指数太大了。

PREFETCH提示用于通知重复数据删除系统可能产生进一步重复的I / O操作（例如，在文件复制期间），以便可以预取和缓存它们的散列以最小化随机访问。这个提示可以在应用程序的读取路径上设置，这些应用程序会再次访问相同的数据。 （注意，读取通常只需要访问LBN→PBN索引，绕过散列索引。）

数据块层的重复数据删除有两个主要优势：（1）允许任何文件系统和应用程序从重复数据删除中获益;（2）易于实现。不幸的是，应用程序和文件系统上下文在块层丢失，这会损害重复数据删除的有效性。但是，通过添加简单但功能强大的提示，我们能够向块图层提供丢失的语义，从而使重复数据删除系统能够提高性能，并可能提高可靠性。我们的实验显示，在dd等应用程序中添加NODEDUP提示可以在复制唯一数据时将性能提高5.3倍，因为可以避免重复数据消除的重复开销。这个提示可以扩展到其他应用程序，如压缩或加密的应用程序。添加PREFETCH提示给像dd这样的应用程序提高了1.8倍的拷贝时间，因为缓存哈希值，不需要访问元数据设备在写入路径上获取它们。向宏工作负载添加提示（如Filebench的Fileserver工作负载）可将吞吐量提高4.5倍。另一个重要的提示是，提示的有效性取决于重复数据删除系统所添加的开销，所写入数据的性质（例如，重复数据删除率）以及工作负载，所以选择时需要考虑所有因素使用提示。