|  |
| --- |
| **文件系统设计概述** |

|  |  |
| --- | --- |
| 院（系）名 称 ： | 计算机学院 |
| 专 业 名 称 ： | 信息安全 |
| 学 生 姓 名 ： | 郝世荣 |
| 学生学号 ： | 2017202110096 |

二〇一七年十二月

目 录

[1 绪论 1](#_Toc500668135)

[1.1 课题介绍 1](#_Toc500668136)

[2 相关理论基础 2](#_Toc500668137)

[2.1 写优化字典 2](#_Toc500668138)

[2.2 BetrFS 2](#_Toc500668139)

[2.3 分区 4](#_Toc500668140)

[2.4 迟绑定日志条目 4](#_Toc500668141)

[2.5 小文件优化 5](#_Toc500668142)

[2.6 预取 5](#_Toc500668143)

[3 改进的文件系统设计 7](#_Toc500668144)

[3.1 优化写文件系统 7](#_Toc500668145)

[3.1.1 避免重复写入 7](#_Toc500668146)

[3.1.2 平衡搜索和重命名 9](#_Toc500668147)

[3.1.3 有效范围删除 11](#_Toc500668148)

[3.1.4 优化的堆 13](#_Toc500668149)

[3.2 复合文件文件系统 13](#_Toc500668150)

[3.2.1 数据表示形式 14](#_Toc500668151)

[3.2.2 元数据表示和操作 14](#_Toc500668152)

[3.2.3 识别复合文件成员 16](#_Toc500668153)

[4 总结 18](#_Toc500668154)

[参考文献 20](#_Toc500668155)

# 绪论

## 课题介绍

文件系统性能优化是一个很好的领域。然而，大多数优化技术（例如，高速缓存）保留了逻辑文件到其物理元数据表示的一对一映射（每个文件与其在UNIX平台上的它自己的i-node相关联）。这样的映射是可取的，因为元数据结构是根深蒂固的数据结构，许多存储组件和机制（如VFS API，预取和元数据高速缓存）依赖于这样的结构。然而，这种严格的映射也是一类性能优化的盲点。许多逻辑文件可以组合在一起并与单个i-node（加上作为扩展属性存储的额外信息）相关联。这样的安排是合理的，因为一起访问的许多文件共享相似的元数据子字段，可以进行重复数据删除。因此，CFFS[1]可以减少对存储的元数据访问，这是访问小文件的一个重要的开销，这仍然是现代工作负载的大多数文件引用的主导。

诸如LSM-trees和Bε-trees之类的写优化字典（WOD）是用于管理文件系统中的磁盘数据的有前景的构建块。相比传统文件系统，先前的基于WOD的文件系统已经提高了随机写入，元数据更新和递归目录遍历的性能数量级。但是，以前的基于WOD的文件系统并没有在不牺牲某些其他操作的性能的情况下获得这三个性能增益。例如，TokuFS和BetrFS文件删除，重命名和顺序文件写入速度较慢。 KVFS和TableFS中的目录遍历基本上不比传统的文件系统更快。TableFS将大型文件存储在底层的ext4文件系统中，因此不会为随机文件写入带来性能提升。基于WOD的文件系统可以保持元数据更新，小型随机写入和递归目录遍历（有时是数量级）的性能改进，同时与其他操作上的传统文件系统相匹配。

# 相关理论基础

## 写优化字典

WOD包括LSMtrees及其变体，Bε-trees，xDicts和cache-oblivious lookahead array。WOD提供了一个支持插入，查询，删除和范围查询操作的键值界面。WOD接口类似于B树的接口，但是性能配置文件是不同的。WOD可以执行快于B树的随机密钥的插入。在旋转的磁盘上，B树在最坏的情况下每秒只能执行几百个插入，而WOD可以执行几万个。在WOD中，删除是通过插入一个非常快的逻辑删除信息来实现的。 一些WOD，如Bε-trees，可以像B树一样快速地执行点查询。 Bε树提供查询和插入性能的可证明的最佳组合。

WOD以接近磁盘带宽执行范围查询。因为WOD可以使用超过兆字节大小的节点，所以扫描每MB数据所需的磁盘搜索少于一个，因此具有带宽限制。写入优化的关键思想是推迟和批量小的随机写入。一个Bε树会在树的根部记录插入或删除消息作为消息，并且当已经产生了足够的消息来抵消访问子代的成本时，只在树中的某一级刷新消息。结果，单个消息可能被多次写入磁盘。由于每个消息总是作为更大批量的一部分写入，因此每个插入的摊销成本通常远小于一个I/O。相比之下，将一个随机元素写入一个大的B树需要最少一个I/O。大多数生产质量的WOD是专门用于数据库而不是文件系统，因此设计时具有不同的性能要求。例如，基于BetrFS的开源WOD实现是TokuDB2到Linux内核的一个端口。TokuDB记录所有插入的密钥和值以支持事务，将写入带宽限制为磁盘带宽的至多一半。因此，BetrFS0.1提供了完整的数据日志功能，尽管需要大量的顺序写入。

## BetrFS

B树数据结构可以说是BTRFS[2]文件系统的基础。它提供了一个通用的方式去存储不同的数据类型。它仅仅存储3个数据类型：key，item和block header。

写优化可是一种很重要的设计策略，即Buffer缓冲区的思想。问题背景就是B树家族的普遍问题，每次插入都会将新加入Key放置到最终位置，从而导致插入性能瓶颈。而WODs家族的数据结构使用缓冲思想，将树内部结点中的一部分划出来当作缓冲区，当插入数据时先暂存到缓冲区，之后再一点点Flush到Key应去的位置。WODs家族的成员有Buffer Tree、Bε和Cache-Oblivious Streaming B树s（COLA）等变种。Bε树。它将Key抽象成异步消息，整个树仿佛成了一个MQ。根据对epsilon参数的选择，决定每个树结点中要留出多少空间作为Buffer。不仅对B树的写性能有大幅提升，而且能够充分利用磁盘带宽

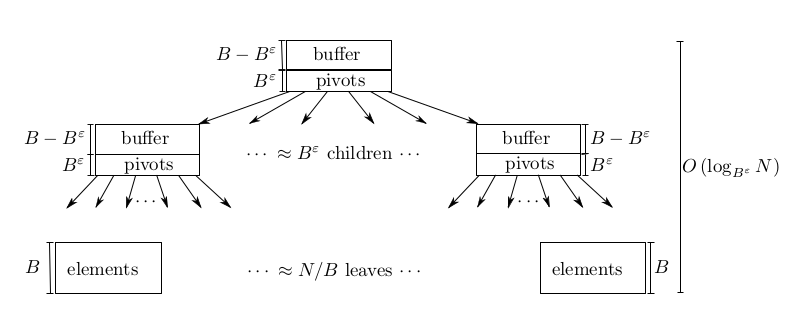


Figure A Bε -tree。每个节点的大小大致是B，ε控制多少内部节点的空间用于枢轴（Bε），多少用于缓冲未决更新（B-Bε）。B树中一样，项目存储在树叶中，树的高度根据分支因子（此处为Bε）在项目总数（N）中为对数，

BetrFS在Bε-trees 中存储所有的文件系统数据，包括元数据和文件内容。 BetrFS使用两个Bε-trees：一个元数据索引和一个数据索引。数据索引将完整路径映射到相应的结构统计信息。数据索引映射（路径，块号）对指定文件块的内容。

传统的文件系统使用间接方式，如索引节点号，用单个指针交换来有效地实现重命名。这种间接性会伤害目录遍历，因为在退化的情况下，每个文件可能有一个搜索。

BetrFS0.1基于完整路径的模式改为优化目录遍历，代价是重命名大文件和目录。递归目录遍历直接映射到底层BΕ树中的范围查询，它可以在几乎磁盘带宽下运行。另一方面，在BetrFS0.1中进行重命名必须将旧密钥中的所有数据移动到新密钥，这对于大型文件和目录可能变得昂贵。架构更改使得BetrFS0.2能够在接近磁盘带宽的情况下执行递归目录遍历，并以与基于 i-node的文件系统相当的速度进行重命名。按完整路径对数据和元数据编制索引也会损害删除性能，因为必须单独删除大文件的每个块。按完整路径对数据和元数据编制索引也会损害删除性能，因为必须单独删除大文件的每个块。 BetrFS0.1中的这些删除消息的大量导致大文件的断链时间更差。

一致性。在BetrFS中，文件写入和元数据更改首先记录在内核的通用VFS数据结构中。VFS可以将数据和元数据高速缓存5秒钟，然后将其写回底层文件系统，BetrFS将其转换为BΕ树操作。因此，BetrFS在崩溃期间最多可能会丢失6秒钟的数据，从VFS层开始5秒钟，从Bε树日志缓冲区开始1秒钟。BetrFS中的fsync首先写入与 i-node关联的所有数据和元数据，然后将整个日志缓冲区写入磁盘。

## 分区

动态子树分区是为大规模分布式系统设计的技术，如Ceph，以减少元数据竞争和平衡负载。这些系统贡献了（1）跨节点的元数据对象的数量和（2）元数据访问的频率。区域会根据其聚合大小来分区对象以重新绑定重命名成本。

Spyglass引入了基于KD树的多维元数据索引的分区技术。分区技术也被用来确定哪些数据比较快的介质更慢，有效地维护倒排文档索引，或者插入不同的存储数据结构以优化读取或写入密集型工作负载。Chunkfs对ext2文件系统进行分区以提高恢复时间。许多系统还将磁盘带宽和高速缓存空间分开以实现性能隔离。一般来说，这些系统主要关心用户或客户之间的公平性，而不是限制最坏的执行时间。这些技术打击了与分区的目录搜索和重命名的平衡不同的特定于域的权衡。

IceFS使用立方体（与区域相似的概念）来隔离错误，消除数据结构和事务机制中的物理依赖性，并允许更精细的粒度恢复和日志配置。立方体由用户明确定义为由整个目录子树组成，随着用户添加更多数据，立方体可以随意增大。相反，区域对用户来说是完全透明的，并且被动态分割和合并。

## 迟绑定日志条目

KVFS通过为每个事务创建一个新的VT-树快照来避免两次写入大部分数据的日志开销。当事务提交时，来自事务的快照VT-树的所有内存数据被提交到磁盘，并且该事务的VT-树被添加到依赖的VT-树之上。在这种情况下数据不会被写入两次，但VT-tree可能会随意增长，导致搜索性能难以推理。

日志结构的文件系统通过只写入日志来避免重复写入的问题。这可以在最佳情况下提高写入吞吐量，但不会强化查询时间的最佳下限。

物理日志记录存储单个数据库页面的前后图像，这对大型更新或大型对象的小型更新可能是昂贵的。当操作有简洁的表示时，逻辑日志记录可以减少日志大小，但是对于大数据插入则不会。

zfs意向日志结合了写时拷贝更新和间接寻址以避免大记录的日志写入放大。我们采用这种技术来实现在BetrFS0.2中对大消息（或大量相关的小消息）进行的后期绑定日志记录。

先前的系统已经实现了软更新的变化，其中首先写入数据，然后是从叶到根的元数据。这种方法下令写入，以便磁盘上的结构始终是一个一致的检查点。虽然软件更新可能在B树中，但是这将是有挑战性的。与软更新一样，后期绑定日志避免了大写操作翻倍的问题，但是与软更新不同，它主要封装在块分配器中。后期绑定对Bε树本身施加了额外的要求，并且不会延迟写入任何树节点来强制排序。因此，具有约束力的期刊特别适合于WOD。

## 小文件优化

虽然复合文件系统的研究主要关注逻辑文件和物理元数据的多对一映射，但这项工作与通过减少存储访问次数来优化小文件访问密切相关。在这方面的早期工作涉及将文件的i节点与其第一个数据块搭配并将i-节点嵌入到目录中。之后，hFS使用单独的存储区来优化小文件和元数据访问。Btrfs将元数据和小文件打包到写复制b-树上。TableFS将元数据和小文件打包到表中，并刷新表条目修改的2MB日志，组织为日志结构合并树。

CFFS通过整合通常一起访问的文件的i节点来补充现有的许多方法。并行和分布式计算领域也探索访问子文件区域和合并元数据的想法，在这个领域中，多台计算机上的CPU需要访问相同的大数据文件。Facebook的照片存储[BEA10]利用观察图像的权限大致相同，可以巩固。但是，这些机制是为非常同类的数据类型量身定制的。通过不同的方式来形成组合文件，CFFS可以处理具有更多不同内容和访问语义的子文件。

## 预取

尽管可以发现大量的工作来改善预取，但也许C-Miner离我们的工作最近。特别是C-Miner在块级应用频繁序列挖掘，优化了文件和元数据块的布局，提高了预取能力。然而，CFFS减少了频繁访问的元数据块的数量，并且避免了需要大表来映射逻辑块到物理块。另外，我们的文件系统级挖掘处理的对象和关联开销也明显减少。有研究结合了磁盘布局的知识来改善预取，预取可以跨文件和元数据边界。CFFS主动减少物理元数据项的数量，并改变存储布局以促进顺序预取。通过将高级执行上下文（例如线程，应用程序ID）传递到块层，结果数据挖掘可以在并发工作负载下生成具有较长运行的预取规则。由于CFFS在文件系统级执行数据挖掘，因此我们可以使用PID和IP地址来解开并发文件引用。然而，CFFS的重点是改变逻辑文件到其物理表示的映射，并且可以采用各种挖掘算法来合并元数据并改善存储布局。

# 改进的文件系统设计

## 优化写文件系统

### 避免重复写入

后期绑定日志记录，这是一种用于在保证全数据记录语义的同时提供仅元数据日志的顺序写入性能的技术。

BetrFS0.1无法匹配传统文件系统的顺序写入性能，因为它将所有数据写入至少两次：一次写入预先写入日志，至少一次写入BΕ树。商品磁盘上的BetrFS0.1以28MB/s执行大量顺序写入，而其他本地文件系统以78-106MB/s执行大量顺序写入 - 利用几乎所有硬盘的125 MB/s的带宽。对于日志记录的额外写入不会显着影响小随机写入的性能，因为它们可能会在批量向下移动Bε树时多次写入磁盘。但是大量的顺序写入可能会直接进入树叶。由于它们在Bε树中只能被写入一次，所以记录一半的BetrFS0.1顺序写入带宽。类似的开销对于就地更新文件系统而言是众所周知的，比如ext4，因此默认为仅包含元数据的日志记录。

无覆盖文件系统解决了日记写间接的问题。对于小数值，zfs将数据直接嵌入到日志条目中。对于较大的值，它将数据写入磁盘重定向写入，并将指针存储在日志中。通过刷新日志，这为zfs提供了快速持久性，避免了两次写入大值的开销，并保留了数据日志的恢复语义。另一方面，无论大小如何，btrfs对所有写入使用间接寻址。它将数据写入到新分配的块中，并将指针写入其日志中。

**BetrFS磁盘结构**。Bε-tree实现将Bε-tree节点写入磁盘，并维护一个逻辑预写重做日志。每个插入或删除消息首先记录在日志中，然后插入到树的内存节点中。日志中的每个条目指定操作（插入或删除）以及相关的键和值。

崩溃一致性是通过定期检查BES树和检查点之间的日志操作来实现的。一旦其日志条目在磁盘上，操作就是持久的。在每个检查点，所有脏节点都被写入，以确保磁盘上有一个完整一致的BΕ树快照，并且丢弃该日志。例如，在检查点完成之后，有一个B树，Ti和一个空的日志。在Ti中无法访问的块可以被垃圾回收和重新分配。在检查点i和i + 1之间，所有操作都以Logi + 1记录。如果在完成检查点i和检查点i + 1之间的任何时间系统崩溃，它将从树Ti恢复并重放Logi + 1。

**后期约束日志**。 BetrFS0.2处理大量消息或大量连续消息：

一个特殊的未绑定日志条目附加到内存日志缓冲区1中。未绑定的日志条目指定一个操作和一个键，但不是一个值。这些消息记录插入的逻辑顺序。

一个特殊的未绑定消息被插入BΕ树2。未绑定消息包含其相应未绑定日志条目的键，值和日志条目标识。未绑定的消息像任何其他消息一样向下移动树。

要使日志持久，包含未绑定消息的所有节点都首先写入磁盘。作为将节点写入磁盘的一部分，将每个未绑定的消息转换为正常的插入消息（非叶节点）或正常的键值对（叶节点）。将节点中的未绑定消息写入磁盘之后，会将绑定日志条目附加到内存中的日志缓冲区3。每个绑定日志条目都包含来自未绑定消息的日志条目标识和节点的物理磁盘地址。一旦在内存中的日志缓冲区中的所有插入都被绑定，则内存中的日志缓冲区将被写入磁盘。

节点回写的处理方式相似：当包含未绑定消息的节点作为高速缓存驱逐，检查点的一部分写入磁盘或出于任何其他原因时，绑定条目会附加到内存日志缓冲区中，以用于所有未绑定消息在节点中，并且节点中的消息被标记为绑定。

系统可以随时写入所有包含未绑定消息的树节点，然后将日志刷新到磁盘，从而使日志操作持久。磁盘日志中的所有未绑定插入将具有匹配的绑定日志条目是一个不变的。因此，恢复总是可以进行到日志的结尾。

磁盘格式不会更改为未绑定的插入：未绑定的邮件仅存在于内存中。

后期绑定日志加速大量消息。数量可以忽略的数据写入日志，但树节点被强制写入磁盘。如果要写入给定树节点的数据量等于节点的大小，则这会将带宽成本降低一半。

在一个或多个插入仅占节点一小部分的情况下，记录这些值优于未绑定的插入。问题是未绑定的插入可能会过早地强制节点到磁盘（在日志刷新时，而不是下一个检查点），从而失去批量更改小修改的机会。编写大部分不变的节点会浪费带宽。因此，BetrFS0.2只有在连续写入至少1MB的页面到磁盘时才使用未绑定的插入。

**崩溃恢复**。后期绑定在恢复过程中需要两次通过日志：一次是识别包含未绑定插入的节点，另一次是重放日志。

核心问题是每个检查点只记录正在使用的检查点的磁盘节点。在BetrFS0.2中，由绑定日志条目引用的节点在检查点的分配表中没有标记为已分配。因此，第一遍需要更新分配表以包括由绑定日志消息引用的所有节点。第二遍重放日志中的逻辑条目。在下一个检查点之后，日志将被丢弃，并且日志引用的所有节点上的引用计数将递减。引用计数达到零的任何节点（即，因为它们不再被树中的其他节点引用）在那个时间被垃圾收集。

**实现**。BetrFS0.2保证直到最后的日志刷新或检查点一致的恢复。默认情况下，在同步操作中，每秒钟或32 MB日志缓冲区填满时触发日志刷新。使用未绑定的日志条目冲洗日志缓冲区还需要在内存树节点中搜索包含未绑定消息的节点，以便首先将这些节点写入磁盘。因此，BetrFS0.2也为绑定日志消息在日志缓冲区的末尾保留了足够的空间。实际上，日志刷新间隔足够长，以便在日志刷新之前将大部分未绑定的插入写入磁盘，从而最大限度地减少日志写入的延迟。

**其他优化**。一个例子是当一个密钥被插入然后被删除;如果插入和删除在同一个消息缓冲区中，则可以删除该插入，而不是刷新到下一个级别。在未绑定的插入的情况下，允许删除在下面的条件写入磁盘之前删除未绑定的插入：（1）所有涉及未绑定的键值对的事务已经提交，（2）删除事务已经承诺，（3）日志尚未刷新。如果满足这些条件，则文件系统可以在没有此绑定值的情况下始终得到恢复。在这种情况下，BetrFS0.2将消除的插入绑定到特殊的NULL节点，并从Bε树中删除插入消息。

### 平衡搜索和重命名

传统的文件系统支持快速重命名，代价是缓慢的递归目录遍历。每个文件和目录都分配有自己的 i-node，目录中的名称通常映射到指针的 i-node。重新命名文件或目录可以非常有效，只需要创建和删除一个指向 i-node的指针和一个固定数量的I/O。但是，搜索目录中的文件或子目录需要遍历所有这些指针。当一个目录下的i-node没有被一起存储在磁盘上时，例如由于重命名，那么每个指针遍历都可能需要磁盘寻道，严重限制了遍历的速度

BetrFS0.1另一个极端。它们通过文件系统中的完整路径对每个目录，文件和文件块进行索引。路径上的排序顺序保证了目录下的所有条目都是按照逻辑顺序连续地存储在BΕ树的节点中，从而能够在目录层次结构的整个子树上进行快速扫描。然而，重命名文件或目录需要物理地将每个文件，目录和块移动到新的位置。

这种权衡在文件系统设计中是很常见的。这些极端之间的中间点是可能的，例如在目录中嵌入i-node但不移动重命名文件的数据块。快速目录遍历需要磁盘上的位置，而重命名只能发出少量的I/O快速。

BetrFS0.2的模式通过将目录层次划分为连接区域（我们称之为区域）来实现这种折中参数化和可调整性。每个区域都有唯一的区域ID，类似于传统文件系统中的 i-node编号。每个区域包含一个文件或者一个根目录，我们称之为区域的根目录。

文件和目录由区域ID和区域内的相对路径来标识。区域中的目录和文件一起存储，可以在该区域内快速扫描。穿越区域边界可能需要寻找树的不同部分。重命名区域根目录下的文件将移动数据，而重命名大文件或目录（区域根目录）则只需更改指针。

分区支持上述两个极端之间的一系列权衡点。当区域被限制为1时，BetrFS0.2模式相当于一个基于 i-node的模式。如果将区域大小设置为无穷大（∞），那么BetrFS0.2的模式等同于BetrFS0.1的模式。在中间设置下，BetrFS0.2可以平衡目录扫描和重命名的性能。

BetrFS0.2中的默认区域大小是512 KiB。直观地说，移动一个非常小的文件是足够便宜的，间接性可以节省很少，尤其是在WOD中。另一个极端是，一旦文件系统在每次查找之间读取几MB，主要成本就是转移时间，而不是寻求。因此，人们会期望最佳区域大小在几十KB和几MB之间。我们还注意到，这种权衡取决于实现：文件系统可以更有效地移动一组键和值，区域越大，不会影响重命名性能。

作为分区的效果，BetrFS0.2通过将具有多于一个链接的文件放入其自己的区域来支持硬链接。

**元数据和数据索引**。BetrFS0.2 数据索引映射（区域ID，相对路径）到关于文件或目录的元数据的密钥。对于同一区域中的文件或目录，元数据包含统计结构的典型内容，如所有者，修改时间和权限。例如，在区域0中，路径“/ local”映射到该目录的统计信息。如果这个键（即区域内的相对路径）映射到不同的区域，则元数据索引映射到该区域的ID。例如，在区域0中，路径“/ docs”映射到区域ID 1，即区域的根目录。数据索引映射到指定文件块的内容。

**路径排序顺序。** BetrFS0.2首先按区域ID对键进行排序，然后按相对路径排序。由于区域中的所有项目将按照此排序顺序连续存储，因此递归目录扫描可以高效地访问区域内的所有条目。在一个区域内，条目以“深度优先”的顺序排列。这种排序顺序可以确保目录下的所有条目在逻辑上连续地存储在底层的键值存储中，然后递归列出该目录的子目录。

**改名**。重命名作为其区域根目录的文件或目录只需在其新位置插入对其区域的引用并删除旧的引用即可

重命名不是其区域根目录的文件或目录，需要将该文件或目录的内容复制到新的位置。重命名目录从不需要递归移动到子区域。因此，通过将目录子树的大小限定在单个区域内，我们还限定了执行重命名所需的工作量。

**分裂和合并**。为了在整个系统生命周期中保持一致的重命名和扫描性能权衡，必须对区域进行拆分和合并，以便维护以下两个不变量：

**Zone Min**：每个区域的大小至少为C0。ZoneMax：不是其区域根目录的大小最多为C1。ZoneMin不变量确保递归目录遍历在启动另一个区域的扫描之前能够扫描键值存储区中的至少C0个连续字节，这可能需要磁盘寻道。ZoneMax不变量确保不需要移动多于C1字节的目录重命名。

BetrFS0.2设计如下维护这些不变量。每个i-node都有两个计数器来记录其子树中的数据和元数据条目的数量。无论何时添加或删除数据或元数据条目，BetrFS0.2都会递归更新相应文件或目录中的计数器直到其区域根目录。如果文件或目录的计数器超过C1，则BetrFS0.2为该文件或目录中的条目创建一个新的区域。当区域大小低于C0时，该区域与其父级合并。 BetrFS0.2避免了级联拆分和合并，只有当这样做不会导致父级拆分时才会合并区域。为了避免大型目录删除时不必要的合并，BetrFS0.2推迟合并，直到写回脏的 i-node。

通过调整C0和C1来调整重命名和目录遍历性能之间的权衡。较大的C0将改善递归目录遍历。然而，将C0增加到基础数据结构的块大小之后将具有递减的回报，因为在扫描单个区域期间系统将不得不寻找块来阻止。较小的C1将改善重命名性能。所有大于C1的对象都可以在一个固定数量的I / O中重命名，最坏的重命名只需要移动C1字节。

### 有效范围删除

BetrFS0.1文件和目录的删除性能在被删除的数据量上是线性的。尽管在任何文件系统中都是如此，但由于释放的磁盘空间在文件大小上是线性的，所以BetrFS0.1的斜率是惊人的。两个潜在的问题是绝对的删除消息的数量必须被插入到BΕ树中，并且在BΕ树实现中没有优化。由于BΕ树实现不会烘焙关于模式的任何语义，所以BΕ树不能推断出两个密钥在密钥空间中是相邻的。如果没有来自文件系统的提示，BΕ树不能优化删除大的连续密钥范围的常见情况。

范围消息:为了支持删除单个消息中的密钥范围，在BΕ-tree实现中添加了范围消息类型。在基线BΕ树实现中，各种形式（例如，插入和删除）的更新被编码为寻址到单个密钥的消息，从根到叶的路径被刷新。范围广播消息可以被寻址到由开始和结束键（包括端点）指定的连续范围的键。这些开始和结束键不需要存在，并且范围可以是稀疏的;该消息将应用于存在的范围内的任何键。

**范围广播消息传播**。当单键消息从父项传播到子项时，它们只是按照逻辑顺序（或者应用于叶子时按键顺序）插入到子项的缓冲区空间中

首先，范围广播消息可以在不同的时间应用于多个孩子。当一个范围广播消息被刷新到一个孩子时，传播功能必须检查范围是否跨越多个孩子。如果是这样，范围广播消息被透明地分割并复制给每个孩子，并具有原始范围的适当子集。如果范围广播消息覆盖节点的多个孩子，则范围广播消息可以被分割并且在不同的时间点被应用到每个孩子上 - 最通常地，这些消息被推迟，直到有足够的消息来为该孩子分摊冲洗成本。随着消息沿着树向下传播，它们将以相同的提交顺序存储并应用于叶。因此，即使范围跨越多个节点，对密钥的任何更新或已删除密钥的重新插入都会维持全局串行顺序。

其次，当范围删除刷新到叶子时，可能会删除多个键/值对，甚至是整个叶子。由于取消链接使用范围删除，所以文件的所有数据块都是相对于崩溃自动释放的。

**查询**。 BΕ树查询必须将节点缓冲区中的所有挂起的修改应用于相关的键。应用这些修改是有效的，因为所有相关的消息都将位于根到叶搜索路径上的节点缓冲区中。

每个BΕ树节点维护待决消息的FIFO队列，并且对于单个密钥消息来说，是由消息的密钥排序的平衡二叉树。对于范围广播消息，我们目前的原型检查一个范围广播消息的简单列表

**范围广播取消链接和截断**。在BetrFS0.2模式中，4KB数据块由区域ID，相对路径和块编号连接而成。取消链接文件涉及一个删除消息，从元数据索引中删除该文件，并在同一个BΕ树等级事务中使用范围删除所有的块。删除文件中的所有数据块只需使用相同的前缀进行编码，但是从块0到无限大。截断文件的方式相同，但可以以零以外的块号开头，并且不会删除元数据键。

Bε树内部优化：对大量删除消息进行分组的能力不仅减少了删除文件所需的全部删除消息的数量，而且还为BΕ树内部优化创造了新的机会。

**叶子修剪**。当BΕ树从一个层次刷新到下一个层次时，必须先读取子进程，合并进入的数据并重写子进程。在大的顺序写入的情况下，可以从磁盘读取大量的取消数据，只是被覆盖。在使用BetrFS0.1的情况下，不必要的读取会比第一次写入文件时慢10到30GB/s。

叶子修剪优化标识何时整个叶子被范围删除消除，并且不从叶片读取叶子。如果插入大量连续的键和值，例如覆盖大文件区域，则BetrFS0.2会在同一事务中包含范围删除。这个范围删除消息是必要的，因为BΕ树不能推断出插入的键的范围是连续的;范围删除传达关于密钥空间的信息。在刷新消息给孩子时，Bε树可以检测范围删除何时包含孩子的密钥空间。BetrFS0.2使用BΕ树实现中的事务来确保移除和覆盖是原子的：在任何时候崩溃都不会丢失修改块的旧内容和新内容。陈旧的叶子节点被回收作为正常的BΕ树垃圾收集的一部分。

因此，当一个大文件被覆盖时，这个叶子修剪优化避免了昂贵的读取。这种优化对于连续的I / O性能是必不可少的，只有在范围删除的情况下才可能实现。

**Pac-Man**。范围删除还可以消除大量的缓冲消息。例如，如果用户创建一个大文件并立即删除文件，BΕ树可能会包含许多不再有利于传播到树叶的有用插入消息。

BetrFS0.2为消息刷新添加了一个优化，其中范围删除消息可以在提交序列之前吞噬消息。我们把这个优化称为Pac-Man，表达对以吞噬鬼魂而闻名的街机游戏角色的敬意。这种优化进一步减少了树中的后台工作，消除了“死”的消息，在他们到达叶。

### 优化的堆

BetrFS有一个堆的文件系统设计; Bε树节点和日志在ext4文件系统上存储为文件。BetrFS0.2纠正了BetrFS0.1使用基础的ext4文件系统次优的两点。 首先，为了确保节点物理放置在一起，TokuDB将零写入节点文件中，以强制更大范围内的空间分配。对于顺序写入新的FS，BetrFS0.1将这些节点置零，然后立即用文件内容覆盖节点，浪费磁盘带宽的三分之一。 我们用更新的fallocate API取代了这个，它可以物理地分配空间，但逻辑上将内容归零。 其次，用于刷新BetrFS日志文件的I/O正被ext4日志所放大。 每个BetrFS日志刷新附加到ext4上的文件，这需要更新文件大小和分配。BetrFS0.2通过为日志文件预先分配空间并使用fdatasync来减少此开销。

## 复合文件文件系统

符合文件文件系统（CFFS），它允许多个小文件合并，共享一个i-node。CFFS引入了一个称为**复合文件（composite file）**的内部物理表示，该文件保存经常一起访问的小文件的内容。复合文件对最终用户是不可见的，并且与在小文件中共享的单个复合i-node相关联。存储在小文件i-node中的原始信息被重复数据删除并存储为复合文件的扩展属性。各个小文件的元数据属性仍然可以被重构，检查和更新，使得传统访问语义（例如，类型，权限，时间戳）不变。扩展属性还记录了单个小文件的复合文件内的位置。通过这种表示，CFFS可以将物理复合文件转换为逻辑文件。

哪些文件要组合成一个复合文件是一个重要的依赖于工作负载的策略决策。CFFS已经被配置了三种方式。第一种方案是**基于目录的合并**，目录中的所有文件（不包括子目录）形成一个复合文件。第二种方案是**嵌入引用合并**，其中提取文件内容中的文件引用以识别可以形成合成文件的文件。三是**基于频次挖掘的整合**，通过设置频次挖掘对文件参考进行分析，以便经常一起访问的文件形成复合文件。

一个复合文件利用传统的VFS预取机制，因为整个复合文件可以以与通过将小数据块组合成更少的较大块所实现的优点FFS类似的方式被预取为一个单元。

### 数据表示形式

复合文件的内容是通过连接小文件（称为**子文件**）形成的。复合文件中的所有子文件共享相同的i-node，以及间接块，双重间接块等。复合文件的最大大小限制不是一个问题，因为复合文件被设计为对小文件进行分组。如果子文件大小的总和超过了最大文件大小限制，我们可以使用多个复合文件。

通常，复合文件中的第一个子文件是**入口点**，其入口将触发预取其余子文件。例如，当浏览器访问一个html文件时，它会加载一个css文件和flash脚本。该html文件可以作为这个三子文件合成文件的入口点和预取触发器。对于基于频率的合并，子文件的排序反映了它们如何被访问。尽管同一组文件可能具有不同入口点的不同访问模式，但是数据布局是基于最普遍的访问模式。

### 元数据表示和操作

**组合文件创建**：创建组合文件时，CFFS分配一个i-node，并将子文件的内容作为其数据进行复制和拼接。组合文件将各个子文件的复合文件偏移量和大小以及重复数据删除的i-node信息记录到其扩展属性中。原始子文件然后被截断，其目录条目被重新映射到复合文件的i-node扩展为还包括子文件ID和其原始i-node的重新分配。因此，最终用户仍然可以感知名称空间中的单个逻辑文件，而单个子文件仍然可以被定位。

**I-NODE内容重构**：重复删除的子文件i-node在不停机状态下重建。默认情况下，除非在扩展属性中另行指定，否则子文件的i-node字段会继承复合文件的i-node字段的值。

**权限**：在文件打开时，首先根据复合i节点检查权限测试。如果失败，则不需要进一步检查。否则，如果子文件具有作为扩展属性存储的不同权限，则将再次检查权限。因此，复合i节点将具有跨所有子文件的最广泛的权限。例如，如果在复合文件中，我们有一个只读的子文件A和一个可写的子文件B，复合i节点的权限将是读写。但是，打开具有写入权限的子文件A时，扩展属性中的只读权限限制将捕获违规。

**时间戳**：每个文件操作都会更新单个子文件和复合文件的时间戳。但是，在检查期间（例如统计系统调用），我们返回子文件的时间戳。

**大小**：对于数据访问，偏移量通过扩展属性中编码的子文件偏移量和大小进行转换和绑定检查。复合文件的大小是复合文件的长度，可以大于其子文件的总大小。例如，如果复合文件中间的子文件被删除，则该区域将被释放，而不会更改复合文件的大小。 i-节点名称空间：对于大于阈值X的i节点数字，使用较高零扩展的N位用于复合i节点号码，较低的M位保留用于子文件ID。我们将这个范围的i-节点号码称为CFFS唯一ID（CUID）。

**子文件查找和重命名**：如果目录中的名称映射到CUID，则可以通过子文件ID查找子文件的属性。重命名将继续进行，就像一个CUID是非CFFS系统中的一个i-node编号一样。由于将子文件移入和移出复合文件将改变其CUID，因此我们需要存储后台指针，以更新映射到CUID的所有名称。

CUID中的改变可能会中断通过其i节点号唯一标识文件的应用程序（例如，备份）。然而，今天的文件系统也可能导致不同的文件在不同的时间共享相同的i节点号码; CFFS设计放大了应用程序不应该假设i节点号码是文件的唯一属性的原因。

**子文件和子文件成员资格更新**：将子文件添加到组合文件时，它会附加到组合文件。从复合文件中删除子文件时，复合文件中相应的数据区域在扩展属性中被标记为已释放。

**子文件打开/关闭操作**：对子文件的打开/关闭调用与对复合文件的打开/关闭调用相同，并转换文件位置指针。

**子文件写操作**：就地更新的处理方式与传统文件系统中的相同。但是，如果更新涉及在复合文件中间生成子文件，并且在子文件末尾没有可用空间，则将更新的子文件移动到复合文件的末尾。该方案利用潜在的时间局部性，即在不久的将来，不断增长的子文件可能再次增长。

**硬链接：**目录中的不同名称可以映射到相同的i节点编号或CUID。

**空间压缩**：当一半分配的大小不包含有用数据时，复合文件压缩其空间。

**并发更新到复合文件中的子文件**：对复合文件中的子文件的并行更新与对普通文件的并发更新具有相同的语义。为避免锁争用，检测到涉及并发更新的文件可能必须提取到多个常规文件中。

**锁定和一致性**：CFFS不支持群集，但我们相信可以实现子文件锁定子系统。

### 识别复合文件成员

基于目录的合并：鉴于传统文件系统具有围绕目录的深层次的空间局部优化，目录是文件访问模式和用于形成合成文件的良好近似。目前，这个合并计划不包括子目录。可以在所有目录上执行基于目录的整合，而无需跟踪和分析文件参考。但是，它不会捕获跨目录的文件关系

基于嵌入式参考的整合：基于嵌入式参考的整合可以根据文件中的嵌入式文件参考标识复合文件成员资格。例如，超链接可能被嵌入在一个html文件中，一个网络爬虫很可能通过这些链接访问每个网页。在这种情况下，我们合并原始的html文件和引用的文件。类似的想法适用于编译。我们可以从Makefile中提取依赖关系规则，并合并导致生成相同二进制文件的源文件。由于文件更新可能会打破依赖关系，CFFS可以定期通过修改的文件进行筛选以协调复合文件成员资格。

基于嵌入式参考的方案可以识别跨目录访问的相关文件，但是可能不容易提取超出基于文本的文件格式（例如，HTML，源代码）的嵌入式文件引用。另外，它需要知道特定的文件格式。

基于频率挖掘的合并：在探索基于频率挖掘的整合，使用Apriori算法的一个变种。关键的观察是，如果经常访问一组文件，它的子集也必须是（Apriori属性）。

**初始通过**：首先，计算每个文件的访问次数，然后删除计数小于阈值的文件（说两个）进一步分析。

**二次通过**：对于其余的文件，我们排列，构建和计数所有可能的双文件引用集。只要在B之后立即访问文件A，反之亦然，我们增加文件集{A，B}的计数。计数小于阈值的集被删除（例如，{B，D}）。

**三次通过**：我们可以根据剩余的两个文件参考集合生成所有三个文件的参考集。但是，如果一个三文件参考集经常出现，那么它所有的两文件参考集也需要频繁出现。因此，{A，B，D}等文件集被修剪，因为在第二遍中{B，D}被消除了。

**终止**：由于我们不能再生成四个文件的参考集，算法结束。现在，如果一个文件可以属于多个文件集，我们将集合{A，B，C}和{A，D}作为两个经常访问的集合返回。诸如{A，B}的集合被删除，因为它们是{A，B，C}的子集。

**变化**：另一种方法是使用标准化的阈值或支持度，即设置事件的百分比（一组事件的发生次数除以总事件的次数，范围介于0和1之间）。

我们也可以跟踪文件参考序列来确定合成文件的入口点和内容布局，而不是跟踪文件集。

我们目前不允许重叠的文件集，以避免复制的复杂性和保持一致性。 要选择两个复合文件之间的子文件成员资格，决定取决于复合文件是否有更多的子文件，更高的支持和更新的创建时间戳。 基于频率挖掘的合并可以基于动态文件引用来识别复合文件候选。但是，运行它的成本将其应用限制在更受欢迎的文件参考序列中。

# 总结

两篇论文立足于文件系统性能优化，分别提出了基于减少对存储的元数据访问的CFFS和优化写文件系统措施。

复合文件系统CFFS允许许多逻辑文件可以组合在一起并与单个i-node（加上作为扩展属性存储的额外信息）相关联，减少对存储的元数据访问，这是访问小文件的一个重要的开销。CFFS改变逻辑文件到其物理表示的映射，并且可以采用各种挖掘算法来合并元数据并改善存储布局。进而达到优化文件系统的目的。

BetrFS 0.2通过以下增强，可以与Linux上的其他本地文件系统大致匹配。在某些情况下，它比其他文件系统要快得多，或者以相当的成本提供强有力的保证。

使用后期绑定日志来在磁盘带宽上执行大量顺序写入，同时保持全数据日志的强大恢复语义。BetrFS0.1提供全数据日记功能，但由于所有数据至少被写入两次，因此大写操作的系统吞吐量减半。后期绑定日志采用了不覆盖文件系统的方法，比如zfs和btrfs，它只将数据写入空闲空间一次。将这种技术应用于Bε树的一个特殊挑战是平衡数据的崩溃一致性与充分的I / O调度灵活性，以避免在Bε树消息刷新中重新引入大量的重复写入。

其次，BetrFS0.2引入了一种名为zoning的可调整目录树分区技术，用于平衡快速递归目录遍历与快速文件和目录重命名之间的紧张关系。快速遍历需要在磁盘上共同定位相关的项目，但为了保持这个位置，重命名必须物理地移动数据。快速重命名可以通过更新几个元数据指针来实现，但是这可以将目录的内容分散在磁盘上。分区产生这两种设计的大部分好处。BetrFS0.2以接近磁盘带宽的方式遍历目录，并以与基于 i-node的系统相当的速度进行重命名。

最后，BetrFS0.2提供了一个新的范围删除WOD操作，加速解除链接，顺序写入，重命名和分区。BetrFS0.2使用范围删除来告诉WOD何时不再需要大量的数据。范围删除启用进一步的优化，如避免读取和合并陈旧的数据，否则将是困难的或不可能的。通过这些增强功能，BetrFS0.2可以大致与Linux上的其他本地文件系统相匹配。在某些情况下，它比其他文件系统要快得多，或者以可比的成本提供更强有力的担保。在少数情况下，速度较慢，但​​在合理的范围内。

这两篇论文分别解决了文件系统运行过程中的主要开销，通过在不影响性能的情况下，减少开销来进行文件的优化，我们在研究过程中应该抓住主要问题后，通过提出创新性的技术，比如改进数据结构甚至提出新的数据结构，状态策略或者存储机制，挖掘算法等来进行文件系统优化。

参考文献

[1] Shuanglong Zhang.The Composite-file File System:Decoupling the One-to-One Mapping of Files and Metadata for Better Performance  
https://www.usenix.org/conference/fast16/technical/sessions/presentation/zhang-shuanglong

[2] Yuan J, Zhan Y, Jannen W, et al. Optimizing every operation in a write-optimized file system[C]// Usenix Conference on File and Storage Technologies. USENIX Association, 2016:1-14.