**海量存储技术结课论文**

**对BetrFS的学习**

**樊成阳**

**2017202110089**

目录

[前言 3](#_Toc500703451)

[1. BetrFS 3](#_Toc500703452)

[1.1 动机与背景 3](#_Toc500703453)

[1.1.1 微写入问题 3](#_Toc500703454)

[1.1.2 写优化索引 4](#_Toc500703455)

[1.2 BetrFS设计 5](#_Toc500703456)

[1.3 BetrFS文件系统模式 6](#_Toc500703457)

[1.3.1 BetrFS数据和元数据索引 6](#_Toc500703458)

[1.3.2 BetrFS操作实现 7](#_Toc500703459)

[1.4 系统设计中的写入优化 9](#_Toc500703460)

[1.4.1 Eternal Sunshine of the Spotless Cache 9](#_Toc500703461)

[1.4.2 FUSE写入优化 9](#_Toc500703462)

[1.4.3 作为块管理器的Ext4 10](#_Toc500703463)

[2. BetrFS针对文件系统老化 10](#_Toc500703464)

[2.1 文件系统老化 10](#_Toc500703465)

[2.2 创建老化的文件系统 11](#_Toc500703466)

[2.3 测量文件系统老化 11](#_Toc500703467)

[2.4 现有的缓解老化策略 11](#_Toc500703468)

[2.5 针对现有文件系统的老化测试 13](#_Toc500703469)

[3. 结论 14](#_Toc500703470)

[参考文献 14](#_Toc500703471)

# 前言

此文章为我海量存储技术结课所写，本文是通过对“BetrFS: Write-Optimization in a Kernel File System”，“BetrFS: A Right-Optimized Write-Optimized File System”，“File Systems Fated for Senescence? Nonsense, Says Science!”，“Optimizing Every Operation in a Write-optimized File System”等文章阅读学习后所作。

# BetrFS

-tree文件系统（BetrFS），是第一个为充分利用写入优化而设计的内核文件系统。具体而言，BetrFS是使用TokuDB（Tokutek的分形树索引）中设计良好的树实现的。这里的设计是针对分形树索引的性能特点进行量身定制的，除此之外，这些决策可能适用于其他写优化文件系统。

## 1.1 动机与背景

### 1.1.1 微写入问题

微写入是指写入操作中建立时间（即在传统磁盘上的寻道时间）超过数据传输时间。传统的文件系统数据结构迫使文件系统设计者在有效的微写入优化和高效扫描之间进行选择。

Update-in-place文件系统通过将相关项目（例如目录中的条目或文件中的连续块）保持在彼此附近来针对扫描进行优化。但是更新性能往往受到底层磁盘随机写入延迟的限制。

基于B-TREE的文件系统在B-TREE中存储逻辑上相邻的相关项，但B-TREE不能保证逻辑上的相邻项在物理上相邻。随着B-TREE的老化，由于插入的节点分裂和删除的节点合并，叶节点散落在磁盘上。在老化的B-TREE中，叶子的逻辑顺序和物理顺序之间几乎没有关系，而读取新叶子的代价设计数据传输成本和搜索成本。如果叶子太小而无法分摊搜索成本，那么范围查询可能会很慢。搜索成本可以通过使用较大的叶来分摊，但是这进一步限制了更新的性能。

在另一个极端，日志文件系统优化写操作。日志记录确保文件的创建和更新速度很快，但是生成的数据和元数据可能在整个日志中传播，导致从磁盘读取数据或元数据时性能较差。在大型扫描（递归目录遍历和备份）中，这些性能问题尤其值得注意，这些扫描无法通过缓存加速。

微写入的瓶颈造成了一系列的应用问题。高性能计算检查点系统生成如此多的微写入，因此定制的文件系统PLFS被设计为通过利用检查点工作负载的细节来有效地处理他们。电子邮件服务器通常很难管理大量小消息和关于这些消息的元数据，例如读标志。桌面环境将首选项和活动状态存储在键值存储区（即注册表）中，以便访问和更新密钥不需要文件系统级微数据操作。 Unix和Linux系统管理员通常通过禁用atime选项来提高10-20％的性能。

即使存储系统使用SSD，微写入操作也会导致性能问题。在基于B-TREE的文件系统中，小的写入触发整个B-TREE节点的较大写入，这可以进一步放大到SSD上的整个擦除块。在一个日志结构的文件系统中，微写入可以引起沉重的清理活动，特别是当磁盘快满时。无论是哪种情况，额外的写入操作都会缩短SSD的使用寿命，并会浪费带宽来限制性能

### 写优化索引

**-tree。**一个-tree是增加了每个节点缓冲区的B-TREE。新项目被插入到-tree的根节点的缓冲区中。当一个节点的缓冲区变满时，消息从该节点的缓冲区移动到其子缓冲区之一。-tree的叶子存储键值对，如同B-tree一样。点和范围查询的行为与B-tree类似，除了从根到叶的路径上的每个缓冲区都必须检查影响查询的项。

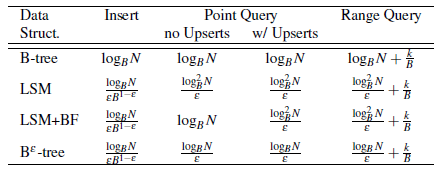


表 1

如表1所示，-tree比B-tree渐进地快。为了解释原因，考虑一个有N项的B-tree，其中每个节点可以保存B个键。（为了简单起见，假定键的大小是固定的，并且与每个键相关的数据的大小可以忽略不计）树具有扇出B，因此其高度将是。插入和查找将因此需要 I/O。返回k项的范围查询将需要I/O。

为了比较，具有大小B的节点的-tree将具有个子元素，其中0 <ε≤1。每个节点将为每个子元素存储一个“枢轴键”，每个节点消耗空间。每个节点中剩余的空间将用于缓冲新插入的项目。由于树的扇出是，其高度是。因此，搜索速度会减为。

然而在实践中，-tree比B-tree使用更大的节点。例如，典型的B-tree可能使用4KB或64KB节点，而Tokutek的分形树索引中的节点则为4MB。B-tree必须使用小节点，因为每次将新项添加到数据库时，都必须完全重写节点，这与在批量写入的-tree中不同。大节点意味着在实践中，-tree的高度不会比相同数据上的B-tree的高度大得多。因此，-tree实现中的点查询的性能可以与B-tree中的点查询性能相比较。大型节点也加快了范围查询的速度，因为数据分布在更少的节点上，需要更少的磁盘查询来读取所有的数据。

**Upserts。**-tree支持upserts，这是一种更新树中键值对的有效方法。当应用程序想要更新与-tree中的关键字k相关联的值时，它会在树中插入一条消息()，其中f指定了一个回调函数，可用于应用由指定的改变为与关键字k相关的旧值。这条消息像任何其他数据一样插入到树中。但是，每次将消息从一个节点刷新到子节点C时，-tree检查C的缓冲区是否包含于关键字k相关联的旧值v。如果是这样，那么他用替换v并丢弃树中的upsert消息。如果应用程序在应用回到函数之前查询到k，那么-tree会在回答查询时动态地计算。这个查询时有效的，因此对于关键字k的upsert总是位于从-tree的根到包含k的叶的路径上。因此，upsert机制可以加快更新一到两个数量级，而不会减慢查询速度。

**日志结构合并树。**日志结构合并树（LSM）是一个有许多变种的WOI。它们通常由指数级增长大小的对数数目的索引（例如B-tree）组成。一旦一个级别的指数填满，则通过将其合并到下一个较大级别的指数中而被清空。

LSM树可以被调整为与-tree具有相同的插入复杂度，但是在原来执行的LSM树中的查询可能会很慢，如表1所示。LSM树的实现者开发了一些方法来提高查询性能，最显着的是对每个B-tree使用布隆过滤器。

布隆过滤器对范围查询没有帮助，因为任何关键字的后继可能处于任何级别。另外，布隆过滤器的效用随着使用upserts而降低，而upserts是BetrFS性能的关键。要明白为什么，请注意为了计算查询的结果，所有相关的upserts必须应用于键值对。如果在LSM树的不同级别上存在许多“正在运行”的高位，则需要在每个这样的级别上执行搜索。由于不同LSM树B-TREE的叶子可能需要获取，所以搜索性能可能降低。

由于分形树索引是与所有工作负载的B-tree查询性能相匹配的唯一WOI实现，包括由BetrFS生成的高端高密度工作负载，因此BetrFS使用-tree（在Tokutek的分形树索引中实现）。简而言之，在特殊情况下，LSM匹配B-tree查询时间，-tree一般匹配B-tree查询时间。

## BetrFS设计

BetrFS是一个内核文件系统，旨在充分利用-tree的性能优势。整个系统体系结构如图1所示.

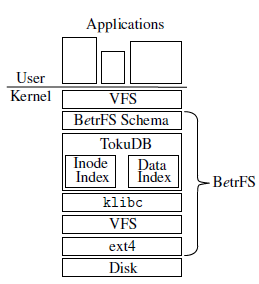


图 1

BetrFS VFS模式尽可能将文件系统操作转换为高效的-tree操作。从-tree获得良好性能的关键是：（1）使用upsert操作来更新文件系统状态;（2）组织数据，以便文件系统扫描可以在-tree中作为范围查询来实现。

通过将BetrFS作为内核文件系统实现，我们避免了FUSE的性能开销，这对于写优化的文件系统可能是特别有害的。同时还公开了优化文件系统与内核页面缓存交互的机会。

BetrFS的堆叠式文件系统设计清楚地将执行写入优化索引的复杂任务从块分配和空闲空间管理中分离出来。TokuDB的内核端口在底层的ext4文件系统上存储数据，但是任何文件系统都足够了。

将一个45KLoC数据库移植到内核中是一项非常重要的任务。通过编写一个shim层（称之为klibc）将TokuDB移植到内核中，该层将TokuDB外部依赖转换为用于锁定，内存分配和文件I / O的内核函数。

## BetrFS文件系统模式

-tree实现了一个键值存储，所以BetrFS必须将文件系统操作转换为键值操作。下面将介绍用于执行此转换过程的BetrFS模式，并说明此模式如何利用-tree的性能优势。

### 1.3.1 BetrFS数据和元数据索引

BetrFS使用底层数据库中的两个索引来存储文件系统数据和元数据：元数据索引和数据索引。由于键和值都可以被变量化，所以BetrFS能够将许多索引条目打包到每个B∈树节点中。

**元数据索引。** BetrFS原型维护一个将完整路径名（相对于挂载点）映射到文件元数据（大致相当于struct stat内容）的索引：

路径→（大小，所有者，时间戳等）

元数据索引旨在支持高效的文件创建，删除，查找和目录扫描。索引首先按照斜杠的数量排序，然后按照字典顺序排列。因此，如图2所示，相同目录内的项目被连续地存储。按照这个顺序，递归或不递归地扫描目录可以被实现为范围查询。



图 2

**数据索引。**虽然键和值可能是可变大小的，但BetrFS原型将文件分割成4096字节块，以便更好地与页面缓存集成。因此，数据索引映射（文件，偏移）元组到块：

（路径，块号）→data[4096]

数据索引中的键也按字典顺序排序，这保证了文件内容在逻辑上相邻，几乎总是物理上相邻的磁盘上。这使得文件内容可以在磁盘带宽附近顺序读取。 BetrFS通过简单地忽略数据索引中的稀疏块来实现稀疏文件。

BetrFS使用可变大小的值来避免零填充每个文件的最后一个块。这种优化避免了将缓冲区的未使用区域归零的CPU开销，然后在将节点写入磁盘之前将零压缩。对于小文件基准测试，这种优化大大降低了开销。例如，这种优化将TokuBench的吞吐量提高了50-70％。

### BetrFS操作实现

**支持盲写入。**在很多文件系统代码中，潜在的假设是数据必须以磁盘扇区粒度写入。因此，一个小的写入操作必须首先将周围的磁盘块放入缓存中，修改该块，然后将其写回。这种模式体现在Linux页面缓存助手功能 \_\_block\_write\_begin() 中。 BetrFS避免了这种读取修改-写入模式，而是在可能的情况下发出无读写的盲写入。

**在BetrFS中读写文件。** BetrFS使用数据索引中的范围查询来实现文件读取。-tree可以在有效磁盘带宽下从磁盘加载大范围查询的结果。

BetrFS通过upsert和insert来支持任何大小的有效文件写入。应用程序写入小于一个4K块成为格式的消息：

这意味着应用程序在指定的偏移量处将数据字节v写入由path指定的文件的块n中。 Upsert消息完全封装了块修改，避免了读取-修改-写入的需要。整个块的写入是通过插入（也称为put）来实现的，该插入是块的替换，并且行为相似。

upserts和inserts是插入到根节点的消息，这些消息渗透到树中。通过使用upserts进行文件写入，写操作文件系统可以将许多小的随机写入聚合到一个大写入磁盘中。因此，通过执行单个查找和一个大写操作，可以将数据提交到磁盘，从而在小型随机写入时可以提高数量级的性能。

在跨越多个块的大写入的情况下，插入遵循从根到树叶复制数据的相似路径。对于大写入，-tree实现有一些优化，可以跳过到叶节点的路径上的中间节点，但是对于大型顺序文件写入来说，它们不够积极，无法实现完整的磁盘带宽。

**BetrFs中的文件系统元数据操作。**如表2所示，BetrFS还将几乎所有的元数据更新（如时间戳更改，文件创建和符号链接）转换为upserts。

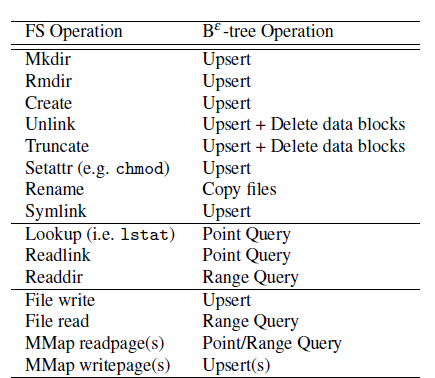


表 2

BetrFS中唯一没有插入的元数据更新是取消链接，截断和重命名。我们现在解释执行这些操作的障碍。

取消链接和截断可以从文件中删除块。 BetrFS以最简单的方式执行此操作：对要删除的块执行范围查询，并为数据索引中的每个块发出TokuDB删除。虽然TokuDB删除操作是使用upserts实现的，但是发出O(n) upserts可能会使这个任务的代价很高昂。

其次，通过完整路径进行键控使得递归目录遍历有效，但是实现高效重命名并不重要。例如，我们当前的实现通过重新插入所有键-值对，然后删除旧键来重命名文件和目录，从而有效地执行被重命名的文件或目录的深层副本。一个简单的解决方案是添加一个inode风格的间接层，第三个索引。这种方法是很好理解的，并且可以随着树的老化牺牲一些读取点。

尽管上述架构可以使用upsert来对文件系统进行大部分更改，但许多POSIX文件系统函数指定了在更改文件系统之前操作系统必须检查的先决条件。例如，在创建文件时，POSIX要求操作系统检查文件是否已经存在，并且用户对包含目录有写权限。

**崩溃一致性。**我们使用TokuDB事务机制来实现崩溃一致性。 TokuDB事务相当于完整的数据日志记录，所有数据和元数据更新记录到底层ext4文件系统中的文件中。日志条目按顺序停用，并且在TokuDB日志记录机制之前没有更新应用于磁盘上的树。条目被附加到两个内存中的日志缓冲区之一（默认情况下为16 MB）。这些缓冲区每秒都会旋转并刷新到磁盘，或者缓冲区溢出。

尽管公开事务API是可能的，但是BetrFS目前仅将事务用作内部一致性机制。除了写数据（每个数据块使用一个事务）以外，BetrFS通常每个系统调用使用一个事务。在我们当前的实现中，元数据上的事务在执行时保持适当的VFS级别的互斥锁，使得事务冲突和异常终止异常罕见。

**压缩。**压缩对性能非常重要，特别是对于键。两个索引都使用完整路径作为键，这个键可以是长而重复的，但TokuDB的压缩缓解了这些开销。使用quicklz将实验中的排序路径名压缩了20倍，使得磁盘空间开销可管理。

数据压缩的使用也意味着在读取文件系统块和从磁盘读取块之间不存在一对一的对应关系。叶节点通常为4 MB，并且压缩可以将多于64个文件系统块打包到叶中。

## 系统设计中的写入优化

### 1.4.1 Eternal Sunshine of the Spotless Cache

BetrFS利用Linux页面缓存实现有效的小读取，避免一般的磁盘读取，并促进内存映射文件。默认情况下，当应用程序写入当前处于缓存中的页面时，Linux会将其标记为dirty，并在稍后写出。这样，多个应用程序级写入页面可以吸收在缓存中，只需要一次写入磁盘。但是，在BetrFS中，小的写入操作非常便宜，以至于这种优化并不总是有意义的。

在BetrFS中，写入系统调用永远不会在缓存中弄脏干净的页面。当一个应用程序写入一个干净的缓存页面时，BetrFS向-tree发出一个upsert，并将写入应用到该页面的缓存副本。因此，缓存的内容仍然与磁盘上的数据同步，并且缓存的页面保持清洁。

请注意，BetrFS的方法并不总是比吸收缓存写入和写回整个块更好。例如，如果应用程序对同一个块执行了数百次小的写入，则将页面标记为dirty并且等到应用程序完成将最终内容写回磁盘会更有效。

缓存中的页面变脏的唯一情况是当文件被写入内存映射时。内存管理硬件不支持对写入内存映射文件的精细追踪-操作系统只知道内存页面内的内容已被修改。因此，BetrFS的mmap实现使用默认的读取和写入页面机制，以页面粒度运行。

### FUSE写入优化

BetrFS作为内核文件系统实现，因为FUSE体系结构包含若干设计决策，可能会破坏写入优化文件系统的潜在性能优势。当与用户空间文件系统进行通信时，FUSE具有额外的上下文切换和数据编组的开销。但是，由于完全不同的原因，FUSE对编写优化的文件系统造成了特别的损害。

FUSE可以将写入密集型转换为读取密集型工作负载，因为它在大多数文件系统更新之前（实际上是在其之后）向用户空间文件系统发出查询。例如，FUSE发出GETATTR调用（类似于调用stat（））以查找文件的整个路径，每次应用程序查找文件。对于大多数内核文件系统，随后的查找可以由内核的目录缓存来处理，但FUSE保守地假设底层文件系统可以异步更改（例如在网络文件系统中可能是这样）。

这些搜索可以阻塞写入优化的数据结构，其中插入比搜索快两个数量级。 TokuFS作者明确指出这些搜索是FUSE实现令人失望的原因。

TableFS作者确定了FUSE开销的另一个来源：内核中inode信息的双缓存。这会降低缓存的有效命中率。对于慢速文件系统来说，额外缓存未命中的开销可能并不重要。对于写入密集型工作负载的写优化数据结构，高速缓存未命中的开销可能很大。

### 作为块管理器的Ext4

由于TokuDB将数据存储在可变大小的压缩节点中，因此TokuDB依赖底层文件系统充当磁盘的块和空闲空间管理器。传统的文件系统在磁盘上相邻地存储大文件块的做法很好，特别是在以大块执行文件写操作的时候。

BetrFS将树节点表示为基础文件系统上的一个或多个大文件中的块，它是未经修改的ext4，具有有序数据模式和直接IO。通过依靠ext4来正确地发出磁盘写入缓存的障碍，尽管禁用磁盘的写入缓存并不会显着影响工作负载的性能。换句话说，所有的BetrFS文件系统更新，数据或元数据通常以数据写入的形式出现，而底层ext4文件系统的fsync则确保了BetrFS日志写入的持久性。尽管层之间存在一些重复的工作，但是我们期望有序的日志模式将这种情况降到最低，因为典型的BetrFS实例从ext4的角度跨越了11个文件。

# BetrFS针对文件系统老化

## 2.1 文件系统老化

随着文件的创建，删除，移动，附加到和截断，文件系统往往会变得碎片化或老化。当逻辑连续的文件块（来自大文件的块或来自同一目录的小文件）分散在磁盘上时，会发生碎片。读取这些文件需要额外的搜索，而在硬盘驱动器上，一些搜索可以对性能产生巨大的影响。例如，如果一个文件系统将100 MiB文件放在具有100 MiB / s带宽和5 ms寻道时间的磁盘上的200个不相交的片段（即200个搜索片段）中，则读取该数据将花费两倍于在理想的布局。即使在SSD上，逻辑块局部性的下降也会损害性能。

缓解老化的技术在分配的时候采用了更为先进的启发式方法，以避免碎片化。例如，文件系统尝试将相关文件放在一起放在磁盘上，同时为将来的文件留下空的空间。一些文件系统还包括碎片整理工具，它们尝试将文件和文件块重新组织到连续区域以抵制老化。

过去二十年来，对老化的意义有不同的看法。 Smith和Seltzer的开创性工作表明，文件系统在实际工作负载下老化，而这种老化影响性能。另一方面，开发人员普遍认为，老化是生产文件系统中解决的问题。

存储技术和文件系统设计方面的变化也可能会严重影响老化。例如，回归分析表明随着旋转磁盘变大，老化会变得更糟，因为寻道时间相对稳定，但带宽随着容量的平方根（大约）增长。考虑与上面的例子相同的碎片级别，但是在带有600MiB / s带宽但仍然是5ms寻道时间的新的，更快的磁盘上。那么这200个寻求就会导致四倍的放缓，而不是两倍的放缓。因此，随着随机I / O与顺序I / O之间的差距扩大，碎片化将成为一个越来越重要的问题。

至于固态硬盘，人们普遍认为碎片不是问题。例如，PCWorld通过对SSD上的NTFS文件系统进行碎片整理来衡量性能的提升，并得出结论：“在我的有限测试中，我坚信即使是最好的SSD碎片整理程序所带来的微小差异也不值得降低您SSD的寿命。“

## 2.2 创建老化的文件系统

Smith和Seltzer的开创性工作创建了一种模拟和测量文件系统老化的方法--比在新的空文件系统上运行更具代表性的基准测试结果。该研究基于从五台服务器的超过五十个真实文件系统的每日快照收集的数据，持续时间从一年到三年不等。 Smith和Seltzer的工作的一个总体目标是评估具有代表性老龄化的文件系统。

随后开发了其他工具来合成老化文件系统。为了衡量NFS的性能，TBBT 被设计为合成一个磁盘的老化时间，为NFS跟踪重放创建一个初始状态。

Impressions框架的设计使用户可以通过设置少量参数（如目录层次结构的组织）来合成老化文件系统。展示次数还可以让用户为结果图像指定目标布局分数。

## 2.3 测量文件系统老化

这里介绍了两种文件系统老化度量：递归扫描延迟和动态布局评分，这些措施旨在捕获文件内碎片和文件间碎片。

**递归grep测试。**在以下部分中介绍的一种方法是在文件系统的根目录中执行递归grep所需的时间。这样可以捕获文件内部和文件内部的影响，因为它可以搜索包含许多小文件的大文件和大文件夹。记录每单位数据的搜索时间，使用ext4的du输出进行标准化。我们将这个称为grep测试。

**动态布局分数。**测量文件中的块的比例，或者在逻辑块空间中以连续的顺序分配（合计）文件系统。将这个分数扩展到文件系统的动态I / O模式。在给定的工作负载期间，使用blktrace捕获由文件系统创建的逻辑块请求，并测量连续的分数。此方法捕获文件系统访问模式的放置决策的影响，包括元数据访问或跨文件访问的影响。高动态布局评分表示良好的数据和元数据位置，以及对于给定工作负载的高效磁盘组织。

这个措施的一个潜在的缺点是它不能区分小的和大的不连贯性。硬盘驱动器上的小的不连续性应该比一般的大的不连续性引起更少的昂贵的机械寻找，然而诸如轨道长度，角度位置的不同和其他几何考虑因素可能使这种关系复杂化。在SSD上，不连续的长度影响较小。因此，将展示动态布局得分与SSD上的grep测试性能强烈相关，并在硬盘驱动器上适度相关。

## 2.4 现有的缓解老化策略

在创建或扩展文件时，必须分配块来存储新数据。尤其是在数据很少或从不重新定位的情况下，如在ext4等就地更新文件系统中，初始块分配决策决定了文件系统在整个生命周期中的性能。在这里，表3介绍了一些在现代文件系统中用来解决老化的策略，主要是在分配时。

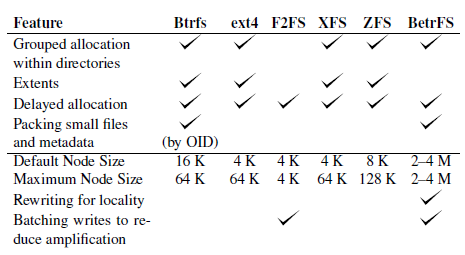


表 3

FFS介绍了柱面组的思想，后来演变成块组或配置组（XFS）。每个组维护有关其inode和块的位图的信息。一个新目录放置在包含多于空闲inode平均数量的柱面组中，而同一个目录中文件的inode和数据块尽可能放置在同一个柱面组中。

ZFS设计用于跨多个设备共享存储。 ZFS根据几个因素的加权计算从设备上的几百个元字节中选择一个，包括最小化搜索距离。选择权重最高的元字节。

在F2FS（一个日志结构化的文件系统）的情况下，磁盘被分成几部分-日志被垃圾收集或清理的粒度。F2FS中与主要的局部相关的优化是写入被分组以提高局部性，并且在找到要写入的另一个片段之前填充脏的片段。换句话说，具有时间地点的写作更可能与物理地点相关联。

组是目录局部性的一种尽力而为的方法：空间用于共同定位同一目录中的文件，但当空间耗尽时，同一目录中的文件可散布在磁盘上。同样，如果一个文件被重命名，它并没有物理移动到一个新的组。

除了F2FS和BetrFS之外，测量的所有文件系统都使用扩展区分配空间，或者物理上连续的块的分配。例如，ext4的范围可以达到128MB。范围减少簿记开销（存储范围与块的详尽列表）。启发式选择较大的扩展可以改善大文件的局部性。例如，ZFS使用首选策略从metaslab中的可用范围中进行选择。

大多数现代文件系统（包括ext4，XFS，Btrfs和ZFS）都实现了延迟分配，其中在将缓冲区写入磁盘之前不分配逻辑块。通过在文件增长时延迟分配，文件系统可以为附加到同一文件的数据分配更大的范围。但是，分配只能拖延很长时间，而不会违反耐久性和/或一致性要求。一个典型的文件系统确保数据肮脏不超过几秒钟。因此，延迟分配只会改善局部性，因为相邻的数据也被写在相同的时间尺度上;在大时间范围内添加或删除数据时，单独延迟分配无法防止碎片化。

应用程序开发人员也可以使用fallocate来请求连续块的持久预分配。为了充分利用这个接口，开发人员必须提前知道每个文件的大小。而且，fallocate只能帮助内部文件碎片;目前还没有一个类似的界面来确保目录的局部性。

对于具有许多小文件的目录，重要的优化可以是将文件内容和潜在的元数据打包成少量的块或范围。 Btrfs将文件和目录的元数据存储在写时拷贝B-tree中。小文件被分解成一个或多个碎片，这些碎片被打包在B-tree内。对于小文件，片段由对象标识符索引（与inode编号相当）;具有多个小文件的目录的位置取决于对象标识符的接近度。

BetrFS将元数据和数据作为键值对存储在两个树中。 树的节点很大（2-4 MiB），以此摊销寻求成本。键/值对按照排序顺序打包在一个节点中，并且随着批量应用更改，节点被周期性地重写，写时复制。

BetrFS还将文件系统的名称空间划分为所需大小（默认为512 KiB）的区域，以便保持目录中的局部性以及实现高效的重命名。每个区域根目录可以是单个大文件，也可以是小文件的子目录。文件或目录的关键是其区域根目录的相对路径。区域中的键/值对是连续的，从而保持局部性

## 2.5 针对现有文件系统的老化测试

使用现实的应用程序工作负载，将五个广泛使用的文件系统Btrfs，ext4，F2FS，XFS和ZFS以及BetrFS研究文件系统。一个工作负载通过对Linux内核源代码执行连续的git checkout出来老化文件系统，模拟开发人员在工作站上可能遇到的老化。第二个工作负载通过运行邮件服务器基准测试文件系统的老化，模拟对持续使用服务器的老化。

评估老化的影响如下：定期停止老化的工作负载，并测量文件系统的整体读取吞吐量--更大的碎片会导致读取吞吐量变慢。为了隔离老化的影响，而不是由于文件大小分布的变化而导致的性能下降，将文件系统复制到新的分区上，本质上产生文件系统的碎片整理或“未老化”版本，并执行相同的测量。然后将老化和未老化拷贝之间的读取吞吐量差异作为老化的结果。

测试结果为：

•所有生产文件系统都在机械硬盘和SSD上使用。例如，在git工作负载下，观察到机械硬盘速度减慢超过50倍，固态硬盘速度减慢超过2-5倍。同样，由于老化，邮件服务器在HDD上减慢了4-30倍。

•老化可以很快发生。例如，ext4显示了在100次git pull之后的2倍减速。在300次pull之后，Btrfs和ZFS同样减速。

•BetrFS基本上不会老化。除了Btrfs之外，在几乎所有的基准测试中，BetrFS的老化性能都比其他文件系统的未老化性能要好。例如，在邮件服务器工作负载上，未老化的ext4比老旧的BetrFS慢6倍。

老化的成本在具体情况下可能是惊人的。例如，在硬盘上的git工作负载结束时，所有四个生产文件系统花费了8分钟的时间来处理1GiB的数据。四个中的两个花了25分钟。 而BetrFS只花了10秒钟。

在执行了几个微观基准来深入研究老化的原因，发现生产文件系统的性能对许多因素敏感：

•Btrfs，ext4，F2FS，XFS和ZFS不能实现5 MiB / s的吞吐量，如果只有10％的文件是相对于目录结构乱序创建的（因此相对于目录树的深度优先搜索）秒。如果文件被完全无序地拷贝，那么这些只有XFS显着超过1 MiB / s。情况不一定如此; BetrFS保持大约50 MiB / s的吞吐量。

•如果应用程序以小块写入文件，则该文件的块可能最终分散在磁盘上，从而在读取文件时损害性能。例如，在硬盘上以循环方式将4个KiB块附加到10个文件的基准测试中，Btrfs和F2FS的读取吞吐量比每个文件完整写入一次的读取吞吐量低10倍。 ext4和XFS更稳定，但最终老化了2倍.ZFS整体上相对较低，但并没有老化。在整个测试中，BetrFS吞吐量稳定在三分之二的全磁盘带宽上。

# 结论

BetrFS演示了写入优化索引是文件系统开发人员的强大工具。在某些情况下，BetrFS比传统设计好几个数量级，比以前的结果更能提高艺术水平。尽管如此，在某些情况下还需要额外的工作，例如对大型流I / O进行进一步的数据结构优化以及对目录进行高效的重命名。研究结果表明，进一步的整合和优化工作可能会产生更好的性能结果。

并且针对现有的文件系统进行对比测试，发现BetrFS有很好的针对文件系统老化的能力，并且除了BtrFS之外，已老化的BetrFS文件系统的性能甚至超过了其他的未老化的文件系统。

# 参考文献

[1] JANNEN, W., YUAN, J., ZHAN, Y., AKSHINTALA, A., ESMET, J., JIAO, Y., MITTAL, A., PANDEY, P., REDDY, P., WALSH, L., BENDER, M., FARACH-COLTON, M., JOHNSON, R., KUSZMAUL, B. C., AND PORTER, D. E. BetrFS: A right-optimized write-optimized file system. In Proceedings of the USENIX Conference on File and Storage Technologies (FAST) (Santa Clara, CA, USA, Feb. 22–25 2015), pp. 301–315.

[2] JANNEN, W., YUAN, J., ZHAN, Y., AKSHINTALA, A., ESMET, J., JIAO, Y., MITTAL, A., PANDEY, P., REDDY, P., WALSH, L., BENDER, M., FARACH-COLTON, M., JOHNSON, R., KUSZMAUL, B. C., AND PORTER, D. E. BetrFS: Write-optimization in a kernel file system. ACM Transactions on Storage (TOS) 11, 4 (Nov. 2015), art. 18.

[3] YUAN, J., ZHAN, Y., JANNEN, W., PANDEY, P., AKSHINTALA, A., CHANDNANI, K., DEO, P., KASHEFF, Z., WALSH, L., BENDER, M., FARACH-COLTON, M., JOHNSON, R., KUSZMAUL, B. C., AND PORTER, D. E. Optimizing every operation in a write-optimized file system. In Proceedings of the USENIX Conference on File and Storage Technologies(FAST) (Santa Clara, CA, USA, Feb. 22–25 2016), pp. 1–14. <https://www.usenix.org/conference/fast16/technical-sessions/presentation/yuan>.

[4] Conway, A., Bakshi, A., Jiao, Y., Zhan, Y., Bender, M. A., & Jannen, W., et al. (2017). File systems fated for senescence? nonsense, says science!. Usenix Conference on File and Storage Technologies (pp.45-58). USENIX Association.