在写优化的文件系统中优化每个操作

摘要

使用写优化字典（WOD）的文件系统可以执行随机写入，元数据更新和递归目录遍历，比传统文件系统快几个数量级。但是，以前的基于 WOD 的文件系统并未在不牺牲其他操作（如文件删除，文件或目录重命名或顺序写入）的性能的情况下获得所有这些性能提升。

使用三种技术，迟绑定日志，分区和范围删除，我们表明，写优化没有根本的权衡。在所有其他操作上匹配传统文件系统时，可以保留这些显着的改进。

BetrFS0.2 在目录扫描和小型随机写入方面比传统文件系统的性能提高了一个数量级，并且与重命名，删除和顺序 I/O 上的传统文件系统的性能相匹配。例如，BetrFS0.2 执行的目录扫描速度提高了 2.2 倍，而且随机写入速度比传统文件系统快两个数量级。但与 BetrFS0.1 不同的是，它重命名和删除与传统文件系统相当的文件，并以接近磁盘带宽执行大型顺序 I/O。这些技术的性能优势也延伸到应用程序。BetrFS0.2 在诸如 rsync，git-diff 和 tar 等许多应用程序上继续优于传统文件系统，但是相比于其他文件系统，git-clone 性能比 BetrFS0.1 提高了 35%。

1 引言

写入优化字典（WODs）， 例如对数类型结构合并树（LSM-树）和B_ε树，构建块用于在文件系统中管理的磁盘上的数据是很有可能的。 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，之前的基于 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 文件的删除，重命名和顺序文件写入都很慢。 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。

本文表明，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。

本文确定了三种技术来解决基于 WOD 的文件系统的基本性能问题，并在 BetrFS 中实现。 我们称之为最终系统 BetrFS0.2 和基线 BetrFS0.1。 尽管我们在 BetrFS 中实现了这些想法，但我们希望他们能够改进任何基于 WOD 的文件系统，并且可能会有更通用的应用。

首先，我们使用 后期绑定日志来在磁盘带宽上执行大量顺序写入，同时保持全部数据的强大恢复语义日记。 BetrFS0.1 提供全部数据日志功能，但对于大写操作，系统吞吐量减半，因为所有数据至少被写入两次。 我们的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。 在适应该技术进行B_ε树 的一个特殊挑战是平衡针对足够的 I/O 调度灵活性数据，以避免再引入大的崩溃一致性，重复在B_ε树消息覆盖写入 。

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。 zoning 同时拥有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 inode 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包括删除 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。

本文的贡献是：

- 一个后期绑定日志，用于大量写入面向消息的 WOD。 BetrFS0.2 以 96MB/s 写入大文件，而 BetrFS0.1 则为 28MB/s。
- 区域树模式和分析骨架 对于在目录遍历中推理地区间的权衡，以及快速文件和目录重命名的间接性。 我们确定了保留原始 BetrFS 大部分扫描性能的一点，并支持对大多数文件和目录大小的传统文件系统重新命名。 最高的重命名开销是比 ext4 慢 3.8 倍。

- 一个范围删除原语，它使 WOD 内部优化文件删除，也避免昂贵的读取和合并死树节点。使用范围删除功能，BetrFS0.2 可以在 11ms 内解开一个 1GB 的文件，而在 BetrFS0.1 和 ext4 上的超过一分钟。
- 一个彻底评测的这些优化和它们对真实应用程序的影响。

因此，BetrFS0.2 表明，一个 WOD 可以提高随机写入，元数据更新，和几个数量级目遍历文件系统的性能，而不会牺牲其他文件系统操作的性能。

2 背景

本节提供了必要的了解和分析 WOD 文件系统的性能的背景，重点是基于B_ε-树和 BetrFS。见 Bender 等。更全面的内容如下。

2.1 写优化字典

WODs 包括日志结构合并树（LSM-树）和它们的变体B_ε-树，xDicts，和高速缓存不经意外表前面的数组（COLAs）。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 界面与 B 树相似，但是性能配置文件不同：

- WOD 可以比 B-树更快地执行随机密钥的插入。在旋转磁盘上，B-树在最坏的情况下每秒只能执行几百次插入，而 WOD 可以执行数万次。
- 在 WOD 中，删除是通过插入一个非常快的逻辑消息来实现的。
- 一些 WODs，如B_ε-树，可以尽可能快执行点查询作为 B-树。B_ε-树（但不是 LSM-树）提供查询的可证明的最佳组合，并插入性能。
- 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。B_ε-树日志插入或缺失作为树的根的消息，只有当足够的消息已产生抵消产生子节点的代价时，才会在树中的某个级别上刷新消息。结果是单个消息可能被多次写入磁盘。由于每条消息总是作为大批量的一部分写入，因此每个

插入的分摊成本通常比一个 I/O 小得多。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，开源 WOD 在 BetrFS 之下的执行是到 Linux 内核中 TokuDB2 的端口。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，尽管需要大量的顺序写入，但是 BetrFS0.1 提供了完整的数据日志功能。

缓存和恢复。我们现在总结一下 TokuDB 的相关日志和缓存管理功能。

TokuDB 更新使用B_ε-树节点重定向写。换句话说，每当脏节点写入磁盘时，该节点就被放置在一个新的位置。恢复基于树的周期性，稳定的检查点。在检查点之间，提前写入，逻辑日志跟踪所有的树更新，并可以重播对最后一个稳定的检查点进行恢复。这个日志被缓存在内存中，并且每秒至少持续一次。

检查点和预写日志的这一方案允许在B_ε-树在内存中缓存脏节点，并将它们写回以任意顺序，只要树的一致版本是在检查点时间写入到磁盘。每个检查点之后，旧的检查点，日志和无法访问的节点都将被垃圾回收。

缓存脏节点会提高插入性能，因为 TokuDB 通常可以避免将内部的树节点写入磁盘。当一个新的消息被插入到树中时，它可以立即尽可能地向下移动，而不会污染任何新的节点。如果消息是长序列插入的一部分，那么整个根到叶的路径很可能是脏的，并且消息可以直接到叶子节点。缓存与预写日志结合起来，解释了为什么在 BetrFS0.1 大顺序写实现至多占用一半磁盘的带宽：大多数消息都写入一次的记录只有一次的叶子。第 3 节描述了一个后期绑定日志，它可以让 BetrFS0.2 只写一次大数据值，而不会牺牲数据的崩溃一致性。

消息传播。当内部B_ε缓冲树节点中的缓冲区填满时，B_ε缓冲树估计哪个或哪些子节点将收到足够的消息来分摊将这些消息刷新到一个级别的成本。消息在节点缓冲区内在逻辑上保持一致，按提交顺序存储。即使消息在不同的时间被物理地应用于叶子，任何读取都会以提交顺序在根和叶子之间应用所有匹配的缓冲消息。第 5 节介绍了“范围强制转换”的消息类型，可以传播给多个子节点。

2.2 BetrFS

BetrFS 存储所有文件系统数据，元数据和文件内容-B 中B_ε-树。BetrFS 使用两个B_ε-树：元数据索引和数据索引。元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。

间接。传统的文件系统使用间接索引（例如索引节点号）来实现单个指针交换的高效重命名。这种间接性会伤害目录遍历，因为在退化情况下，每个文件可能会有一个搜索。

BetrFS0.1 基于完整路径的模式改为以重命名为代价来优化目录遍历大文件和目录。递归目录遍历直接映射到在底层B_ε树范围查询，其可以运行在近磁盘带宽。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。第 4 部分介绍了架构更改，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 inode 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。BetrFS0.1 中这些删除消息的大量数量导致大文件的解链时间数量级下降。第 5 节介绍了我们新的“范围删除”原语，用于在 BetrFS0.2 中实现高效的文件删除。

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。该 VFS 可以写回底层文件系统，其 BetrFS 转换到B_ε操作之前缓存脏数据和元数据长达 5 秒。因此 BetrFS 可以碰撞-5 从 VFS 层秒，从B_ε日志缓冲区 1 秒期间丢失数据的最多 6 秒。BetrFS 中的 fsync 首先写入与 inode 关联的所有脏数据和元数据，然后将整个日志缓冲区写入磁盘。

3 避免重复写入

本节讨论后期绑定日志记录，这是一种技术，用于在保证全数据日志记录语义的同时，实现仅元数据日志的顺序写入性能。

BetrFS 0.1 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B_ε树。正如我们在第 7 节中的实验所示，商品磁盘上的 BetrFS 0.1 以 28MB / s 执行大量的顺序写入，而其他本地文件系统以 78-106MB / s 执行大量顺序写入 - 利用几乎所有的硬盘驱动器 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B_ε树时被写入磁盘几次。但是，如 2.1 节所述，大量的顺序写入可能会直接进入树叶。由于它们在B_ε树中只写入一次，所以记

录　　一半的 BetrFS0.1 次写入带宽。类似的开销对于就地更新文件系统而言是众所周知的，例如 ext4，因此默认为仅包含元数据的日志记录。

流行的无覆盖文件系统解决了日记写间接的问题。对于较小的值，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。这通过刷新日志为 zfs 提供了快速的持久性，避免了两次写入较大值的开销，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。

在本节的其余部分，我们解释我们如何为大型写入到 BetrFS 恢复机制整合间接，和大家讨论由B_ε-树的面向消息的设计所带来的挑战。

BetrFS 磁盘结构。该 BetrFS B_ε-树执行写入B_ε-树节点使用重定向-ON-写入磁盘，并保持合理的预写重做日志。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。

崩溃一致性通过周期性检查点的B_ε和由检查点之间的测操作实现。一旦其日志条目在磁盘上，操作就是持久的。在每个检查点，所有脏节点写入，以确保完整和一致的B_ε-树快照磁盘上，日志被丢弃。例如，检查点完成后，有一个单个B_ε-树，*Ti*和一个空的日志。任何在 *Ti* 中无法访问的块都可以被垃圾收集和重新分配。

在检查点 *i* 和 *i*+1 之间，所有操作都记录在 Log*i*+1 中。如果系统在检查点 *i* 和检查点 *i*+1 完成之间的任何时间崩溃，则将从树 *Ti* 恢复并重播 Log*i*+1。

后期约束期刊。BetrFS0.2 处理大消息或大量连续消息，如下所示，如图 1 所示：

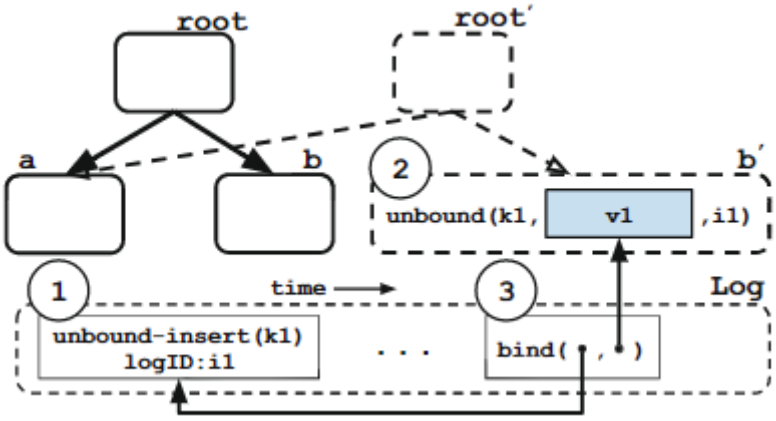


图 1：B_ε-树中的后期绑定日志

• 特殊的未绑定日志条目被追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到B_ε-树 2 中。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，包含非绑定消息的所有节点首先写入磁盘。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。该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，每隔一秒进行一次同步操作或 32MB 日志缓冲区填满时触发日志刷新。使用未绑定的日志条目刷新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。在实践中，日志刷新间隔足够长，大部分未绑定的插入在日志刷新之前写入磁盘，从而最大限度地减少日志写入的延迟。

其他优化。第 5 部分解释了一些优化，其中逻辑上可以避免的操作可以作为从树的一级刷新消息的一部分而被丢弃。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。在未绑定插入的情况下，我们允许在下列情况下将值写入磁盘之前删除未绑定的插入：（1）涉及未绑定键值对的所有事务已经提交，（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 结合消除插入到特殊 NULL 节点，并丢弃从B_ε-树插入消息。

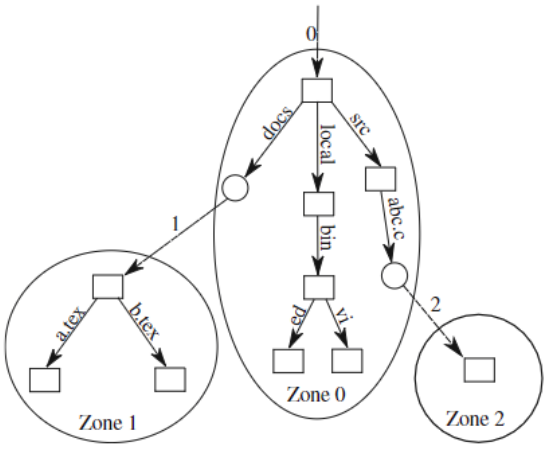
4 平衡搜索和重命名

在本节中，我们认为在重命名和递归目录扫描之间存在设计权衡。我们提出了一个在这个折衷曲线上选择一个点的算法框架。

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 inode，目录中的名称通常映射到指针的 inode。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 inode 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 inode 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 和 TokuFS 在另一个极端。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。在路径上的排序顺序保证一个目录下的所有条目被连续存储在逻辑顺序中这些B_ε-树的节点内，使在目录层次结构的子树整个快速扫描。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 inode 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。



(a) An example zone tree in BetrFS 0.2.

Metadata Index

(0,"/")	→	stat info for "/"
(0,"/docs")	→	zone 1
(0,"/local")	→	stat info for "/local"
(0,"/src")	→	stat info for "/src"
(0,"/local/bin")	→	stat info for "/local/bin"
(0,"/local/bin/ed")	→	stat info for "/local/bin/ed"
(0,"/local/bin/vi")	→	stat info for "/local/bin/vi"
(0,"/src/abc.c")	→	zone 2
(1,"/")	→	stat info for "/docs"
(1,"/a.tex")	→	stat info for "/docs/a.tex"
(1,"/b.tex")	→	stat info for "/docs/b.tex"
(2,"/")	→	stat info for "/src/abc.c"

Data Index

(0,"/local/bin/ed",*i*)	→	block *i* of "/local/bin/ed"
(0,"/local/bin/vi",*i*)	→	block *i* of "/local/bin/vi"
(1,"/a.tex",*i*)	→	block *i* of "/docs/a.tex"
(1,"/b.tex",*i*)	→	block *i* of "/docs/b.tex"
(2,"/",*i*)	→	block *i* of "/src/abc.c"

(b) Example metadata and data indices in BetrFS 0.2.

图 2：BetrFS 0.2 中的区域树的图形和模式说明

BetrFS0.2 的模式通过将目录分层结构划分为连接区域（我们称之为区域）来实现这种可折衷的参数化和可调整性。图 2(a)显示了如何将子树内的文件和目录收集到 BetrFS0.2 的区域中。每个区域都有唯一的区域 ID，类似于传统文件系统中的 inode 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）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 inode 的模式。如果将区域大小设置为无穷大（∞），则 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 512KiB。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，可以不损害重命名性能。第 7 部分经验性地评估了这些权衡。

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 元数据索引映射（zone-ID，相对路径）到关于文件或目录的元数据，如图 2b 所示。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 ID1，即区域的根目录。

数据索引映射到指定文件块的内容（zone-ID，relative-path，block-number）。

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，如图 2b 所示。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。因此，一个对目录执行 readdir，然后以 readdir 返回的顺序递归扫描其子目录的应用程序将有效地对该区域及其下面的每个区域执行范围查询。

重命名。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。因此，例如，将图 2 中的“/src/abc.c”重命名为“/docs/def.c”，需要删除关键字（0，“/src/abc.c”）元数据索引并插入映射（1，“/def.c”）→区域 2。

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其新位置。因此，例如，重命名“/local/bin 中”向“/文档/工具”需要（1）删除形式的所有密钥（0，“/local/bin 中/*P*”）在元数据索引，（2）（1，“/tools/*p*”），（3）从数据索引中删除表单的所有键（0，“/local/bin/*p*”，*i*）将它们重新插入为表单的键（1，“/tools/*p*”，*i*）。请注意，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：ZoneMin：每个区域的大小至少为 $C0$；ZoneMax：不是其区域的根目录的大小最多为 $C1$。ZoneMin 不变量确保递归目录遍历能够

在启动对另一个区域的扫描之前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 不变量确保没有目录重命名需要移动超过 $C1$ 字节。

BetrFS0.2 设计如下维护这些不变量。每个 inode 都维护两个计数器，以记录其子树中的数据和元数据条目的数量。每当添加或删除数据或元数据条目时，BetrFS0.2 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，该区域与其父区域合并。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 inode。

我们可以通过调整 $C0$ 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外将具有递减的回报，因为在单个区域的扫描期间系统将不得不寻找块到块。较小的 $C1$ 将提高重命名性能。所有大于 $C1$ 的对象可以在固定数量的 I/O 中重命名，而最坏的重命名只需要移动 $C1$ 字节。在当前的实现中，$C0 = C1 = 512\text{KiB}$。

区域模式使 BetrFS0.2 能够支持重命名性能和目录遍历性能之间的一系列权衡。这些我们将在第 7 部分探讨。

5 有效的范围删除

本节介绍如何 BetrFS0.2 通过引入新的 rangecast 消息类型到B_ε-树，和实施使用这种新的消息类型几个B_ε-树内部优化获得几乎平坦删除时间。

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。例如，取消连接一个 4GB 的文件需要 5 分钟的 BetrFS0.1！

两个根本问题是必须插入转换为B_ε-树，错过了在B_ε-树实现优化删除邮件的绝对数量。因为B_ε-树实现不绕模式中的任何语义烘烤中B_ε不能推断出两个键是在密钥空间相邻。而不从文件系统的提示，B_ε-树不能优化用于删除大的，连续的键范围的一般情况。

5.1Rangecast 消息

为了支持一个键范围的删除在单个消息中，我们添加了一个 rangecast 消息类型到B_ε-树实现。在基于B_ε-树实现方式中，各种形式的更新（例如，插入和删除）被编码成的消息寻址到单个键，其中，因为在§2 解释的那样，被冲入从根到叶的路径，消息可以寻址到一个连续范围密钥，由开始和结束键，包括指定。这些开始和结束键不存在，且范围可以是稀疏的;该消息将被应用到的范围内确实存在任何密钥。目前，我们已经增加了 rangecast 删除消息，但我们可以预见范围内插入和更新插入是有用的。

Rangecast 消息传播。当单密钥消息从父到子传播的，它们被简单地插入到在逻辑顺序孩子的缓冲空间（或在键顺序当施加至叶）。Rangecast 消息传播类似于普通消息传播，有两点不同。

首先，rangecast 消息可能会在不同的时间应用到多个孩子。当 rangecast 消息被刷新到一个孩子，传播功能必须检查的范围是否跨多个孩子。如果是，则 rangecast 消息被透明地分裂和复制为每个子，与原来的范围的适当子集。如果 rangecast 消息覆盖了节点的多个儿童，rangecast 消息可以在时间最常被拆分并在不同的点应用到每一个孩子，推迟直到有针对孩子摊销成本刷新足够的信息。作为信息传播下来的树，它们存储在相同提交顺序应用到叶片。因此，要删除键的键或 reinsertions 任何更新保持全球串行顺序，即使 rangecast 跨越多个节点。

其次，当一 rangecast 删除被刷新到叶，其可移除多个键/值对，或甚至整个叶。由于取消链接使用 rangecast 删除所有文件的数据块的相对于碰撞原子释放。

查询。B_ε-树查询必须在节点缓存应用所有待修改相关的键（一个或多个）。应用这些修改是有效的，因为所有的相关信息将在一个节点的根到叶的搜索路径上的缓冲。Rangecast 信息保持不变式。

每个B_ε-树节点保持未决消息的 FIFO 队列，并且，对于单关键信息，平衡二叉树由消息键进行排序。对于 rangecast 的消息，我们目前的原型检查 rangecast 消息的简单列表，并根据提交顺序关键信息交织的消息。这种搜索在 rangecast 消息的数量线性成本。更快的实施将使用间隔树存储在每个节点中的 rangecast 消息，使其能够找到在所有相关于查询的 rangecast 消息 $\hat{O}$（$\acute{k}$+登录 $\tilde{N}$）时间，其中 $\tilde{N}$是在节点和 rangecast 消息数 $\acute{k}$是相关的当前查询的信息的数量。

Rangecast 断开链接并截断。在 BetrFS0.2 模式，4KB 的数据块由区 ID，相对路径，和块数的一个级联元组为关键字。取消链接文件涉及一个删除消息，以除去从元数据索引的文件，并在相同的乙B_ε-tree-级别的事务，一个 rangecast 删除，以除去所有的块。删除所有数据块在文件中简单地通过使用相同的前缀进行编码，但是从块 0 到无穷大。截断文件的工作方式相同，但可以用零以外的块号开始，并且不会删除元数据键。

5.2 B_ε-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，而且为B_ε树内部优化创造了新的机会。

叶片修剪。当一个B_ε-树将数据从一个水平刷新到另一个水平时，它必须先读取孩子，合并输入数据，然后重写孩子。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 BetrFS 0.1 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。

叶修剪优化识别当整个叶是由一系列消除删除和 elides 从磁盘读取叶。当插入大范围的连续的键和值，诸如覆盖一个较大的文件区域，BetrFS0.2 包括一系列删除在同一事务中的键范围。该范围内删除消息是必要的，作为B_ε-树不能推断出所插入的键的范围是连续的，删除的范围内传递与密钥空间的信息。在刷新消息一个子节点时，B_ε-树可检测出各种删除包括孩子的密钥空间。BetrFS0.2 采用 B 内交易B_ε-树的实现，以确保清除和覆盖是原子的：在任何时候，可以在碰撞失去两个被修改块的新老内容。陈旧叶节点被回收作为正常 B 的部分B_ε-树垃圾收集。因此，该叶片修剪优化避免了昂贵的当一个大文件被覆盖读取。这种优化是顺序 I/O 性能和可用的只有 rangecast 删除这两个至关重要的。

6 优化堆叠

BetrFS 具有堆叠文件系统的设计;B_ε-tree 节点和轴颈被存储为 ext4 文件系统上的文件。BetrFS0.2 校正其中 BetrFS0.1 使用底层 ext4 文件系统未达最佳的两点。、

首先，为了使各节点在物理上放置在一起，TokuDB 写入零到节点文件，以迫使在更大程度上的空间分配。连续写入到一个新的 FS，BetrFS0.1 零点这些节点，然后立即用覆盖文件内容的节点，浪费了磁盘的带宽的三分之一。我们与新的 fallocateAPI，它可以实际分配空间，但为逻辑 0 的内容替换此。

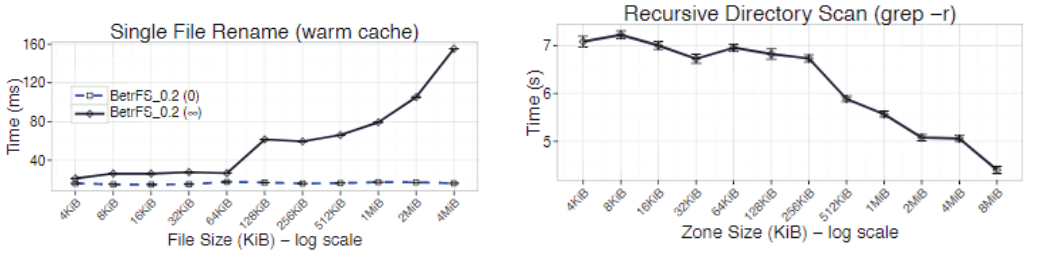
其次，I/O 刷新 BetrFS 日志文件正在由 ext4 的日记放大。追加到上 EXT4 一个文件，这需要更新的文件大小和分配各 BetrFS 登录平齐。BetrFS0.2 通过预分配的日志文件空间，并减少这方面的开销。

7 评价

我们的评估针对以下问题：

- 怎样才能选择区域大小？
- 是否 BetrFS0.2 在最坏的情况下为 0.1BetrFS 同等执行到其他文件系统？
- 是否 BetrFS0.2 执行 comparablytoBetrFS0.1on
- 最好的情况下为 0.1BetrFS？
- 如何 BetrFS0.2 优化影响应用的性能？这是性能堪比其他文件系统和一样好，甚至 BetrFS0.1 更好？
- 什么是后台工作 BetrFS0.2 的成本？

所有的实验结果收集于戴尔 OptiPlex790 采用了 4 核 3.40GHz 的英特尔酷睿 i7CPU，4GBRAM 和 500GB7200 转 ATA 磁盘，以 4096 字节的块大小。每个文件系统的块大小为 4096 个字节。该系统运行的 Ubuntu13.10，64 位，与 Linux 内核版本 3.11.10。每个实验用多个文件系统，包括 BetrFS0.1，BTRFS，EXT4，XFS，和 ZFS 相比。我们使用是 3.11.10 内核的一部分 XFS，BTRFS，EXT4 的版本，和 ZFS0.6.3。磁盘被分成 2 个分区大致 240GB 每个;一个用于根 FS，另一个用于实验。我们使用默认推荐的文件系统设置，除非另有说明。inode 表和期刊进行初始化上的 ext4 关闭。每实验运行至少 4 倍。误差棒和±范围表示 95 个%置信区间。除非另有说明，所有的基准是冷缓存测试。



(a) BetrFS 0.2 file renames with zone size ∞ (all data must be moved) and zone size 0 (inode-style indirection).

(b) Recursive scans of the Linux 3.11.10 source for "cpu_to_be64" with different BetrFS 0.2 zone sizes.

图 3：区域大小对重命名和扫描性能的影响。 越低越好。

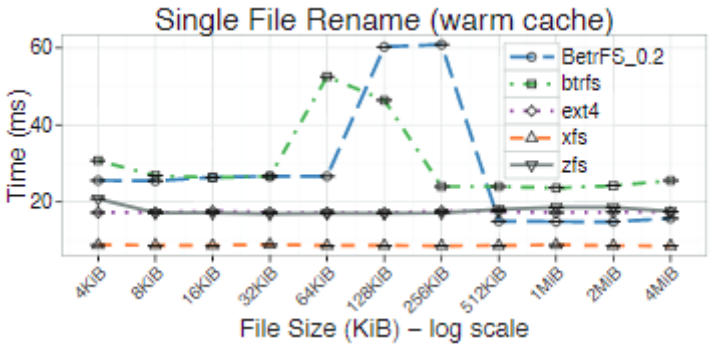


图 4：时间来重命名单个文件。 越低越好

7.1 选择一个区域大小

良好的区大小限制重命名的最坏情况下的成本，但维护数据局部性快速目录扫描。图 3a 示出重命名文件和 FSYNC 父目录的平均费用，经 100 次迭代，绘制为大小的函数。我们展示 BetrFS0.2 具有无限区尺寸（不带创建，重命名移动所有的文件内容）和 0（每个文件是在自己的区域-重命名是一个指针交换）。一旦文件是在它自己的区域，在性能与大多数其他文件组弱（上 BetrFS0.216ms 的相比，17MS 上的 ext4）。这是对图 3b，其示出的 grep 性能与区大小平衡。正如在第 4 预测目录遍历性能提高作为区规模的增大。

我们选择 512KIB，这强制对最坏的情况下的重命名合理结合的默认区域大小（相对于未受限制的 BetrFS0.1 最坏的情况下），并保持渐近线的 25%以内的搜索性能。0.2 重命名时间图 4 比较 BetrFS 到其它文件系统。具体而言，在这个区域的大小最坏情况下的重命名性能 66ms，3.7×低于 18 毫秒的平均文件系统的命名成本慢。然而，重命名文件 512KIB 或更大的比得上其他文件系统，并且搜索性能是 2.2×最好基线文件系统和 8×中值。我们使用此区域大小评估的其余部分。

7.2 改善最坏的情况下

该措施 BetrFS0.1 的三最坏的情况下，并且示出的是，对于典型的工作负载，BetrFS0.2 或者是更快或其他文件系统的约 10%之内。

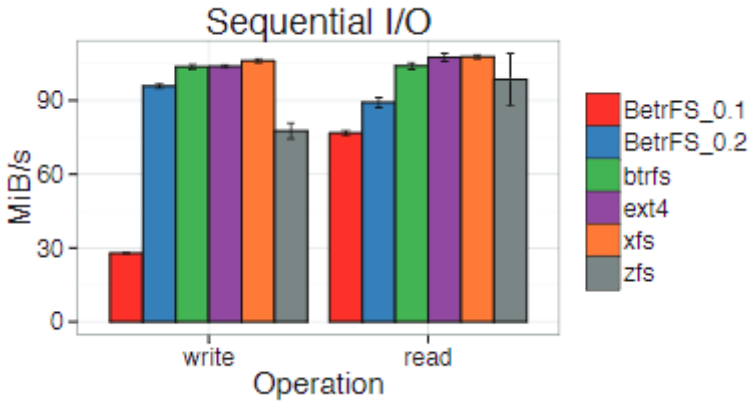


图 5： 大文件 I / O 性能。 我们依次读写一个 10GiB 文件。 越高越好

顺序写入。图 5 示出了可以通过顺序地读取和写入一个 10GiB 文件（机器的 RAM 的两倍以上的大小）。中所描述的优化在§30.1BetrFS 的 BetrFS0.2 顺序写入吞吐量提高到 96MiB/秒，从 28MiB/秒。除了 ZFS，其他的文件系统实现更高的大约 10%的吞吐量。我们还注意到，这些文件系统提供不同的崩溃一致性特性：EXT4 以及 XFS 唯一保证元数据恢复，而 ZFS，BTRFS 和 BetrFS 保证数据恢复。

顺序读取吞吐量 BetrFS0.2 由大致 12MiB/s，这是归因于精简码超过 BetrFS0.1 提高。这地方 BetrFS0.2 其他文件系统的攻击距离之内。

重命名。表 1 显示了在 Linux3.11.10 几种常见的目录操作的执行时间的源代码树。重命名测试重命名整个源代码树。BetrFS0.1 目录重命名是 MAG-nitude 比任何其他文件系统慢两个数量级，而 BetrFS0.2 比除 XFS 每个其他文件系统更快。通过划分目录层次为区域，BetrFS0.2 确保重命名的成本相当于其他文件系统。

取消链接。表 1 还包括递归删除 Linux 源代码树中的时间。再次，而 BetrFS0.1 比其他任何文件系统量值慢的顺序，BetrFS0.2 更快。我们认为这是改善 BetrFS0.2 快速目录遍历和范围缺失的短跑运动员，那些收不到。

我们还测量了链接大小不断增大的文件的延迟。 由于规模的限制，我们将图 6 中的 BetrFS0.1 与 BetrFS0.2 进行对比，并且将 BetrFS0.2 与图 7 中的其他文件系统进行比较。在 BetrFS0.1 中，删除文件的成本随文件大小线性缩放。 图 7 显示了 BetrFS 0.2 删除延迟对文件大小不敏感。 测量表明，zfs 性能相当慢和噪声较大; 我们怀疑这个变化是可以分开的，并且发生了大量的家务劳动。

7.3 保持最佳案例

本小节评估最好的情况下为直写优化的文件系统，包括小型随机写入，文件创建和搜索。我们确认，我们的优化还没有被侵蚀的写入优化的好处。在大多数情况下，没有任何损失。小型，随机写入。表 2 示出了微基准发出万 4 字节的执行时间将覆盖在 10GiB 文件内随机偏移，接着是 FSYNC。BetrFS0.2 不仅保留两个数量的数量级的改进在其他文件系统，但由 34%改善了 BetrFS0.1 的延迟。

File System	find	grep	mv	rm -rf
BetrFS 0.1	0.36 ± 0.0	3.95 ± 0.2	21.17 ± 0.7	46.14 ± 0.8
BetrFS 0.2	0.35 ± 0.0	5.78 ± 0.1	0.13 ± 0.0	2.37 ± 0.2
btrfs	4.84 ± 0.7	12.77 ± 2.0	0.15 ± 0.0	9.63 ± 1.4
ext4	3.51 ± 0.3	49.61 ± 1.8	0.18 ± 0.1	4.17 ± 1.3
xfs	9.01 ± 1.9	61.09 ± 4.7	0.08 ± 0.0	8.16 ± 3.1
zfs	13.71 ± 0.6	43.26 ± 1.1	0.14 ± 0.0	13.23 ± 0.7

Table 1: Time in seconds to complete directory operations on the Linux 3.11.10 source: `find` of the file "wait.c", `grep` of the string "cpu_to_be64", `mv` of the directory root, and `rm -rf`. Lower is better.

File System	Time (s)
BetrFS 0.1	0.48 ± 0.1
BetrFS 0.2	0.32 ± 0.0
btrfs	104.18 ± 0.3
ext4	111.20 ± 0.4
xfs	111.03 ± 0.4
zfs	131.86 ± 12.6

Table 2: Time to perform 10,000 4-byte overwrites on a 10 GiB file. Lower is better.

小文件的创建。为了评价文件的创建，我们使用了 TokuBench 基准创造一个平衡的目录树 3000000 个 200-字节的文件与 128 的扇出我们用 4 个线程，每个机器的核心之一。每秒创建为创建的文件的数量的函数图 8 的曲线图的文件。换句话说，在百万在 x 轴上的点是在创建百万分之一文件中的时间的累积吞吐量。围绕创建一个半百万个文件后 ZFS 排气系统内存。

对于 BetrFS0.2 行是大多比线更高为 BetrFS0.1，并且都维持吞吐量至少 3 倍，但通常一个数量级，比任何其他的文件系统更高（注意 Y 轴是对数标度）。由于 TokuBench 的平衡目录层次结构和编写模式，BetrFS0.2 执行 16,384 区在大约 2 万个文件分割临门。这导致在性能和立即恢复的突然下降。

搜索。表 1 显示了搜索名为“wait.c”（查找）并搜索文件内容的字符串“CPU 到 be64”（grep）来处理文件的时间。这些操作是在两个写入优化文件系统不相上下，尽管 BetrFS0.2grep 的 46%，这归因于所述的权衡来添加分区减慢。

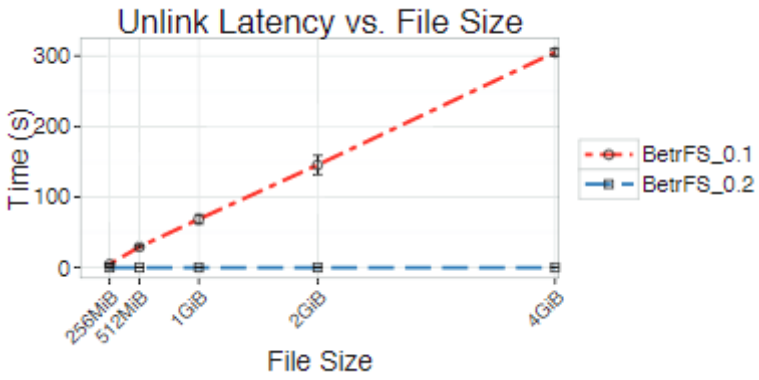


图 6：通过文件大小取消延迟延迟

7.4 应用性能

本小节评估 BetrFS0.2 优化对若干应用，在图 9 中图 9a 中所示的性能的影响示出的 rsync 的吞吐量，具有和不具有--in 就地标志。在这两种情况下，BetrFS0.2 提高了吞吐量超过 0.1BetrFS 并保持超过其他文件系统一个显著的改善。更快的顺序 I/O，并在第二种情况下，更快重命名，有助于这些收益。

在 git 克隆的情况下，顺序写入的改进使 BetrFS0.2 性能堪比其他写一个 10GiB 文件（左）和文件（右）在部分重写 10,000 随机块之后。越高越好。文件系统，不像 BetrFS0.1。同样，BetrFS0.2 略有改善的 git-DIFF 的性能，使得它显然比其他 FSes 更快。

无论 BetrFS0.2 和 ZFS 跑赢上达夫科特 IMAP 工作量与其他的文件系统，虽然 ZFS 是最快的。该工作负载的特点是频繁的小写入和 fsync 作业，和这两个文件系统通过刷新其日志快速持续小的更新。在 BetrFS0.2，焦油比 BetrFS0.1 慢 1%，因分割区域的额外的工作。

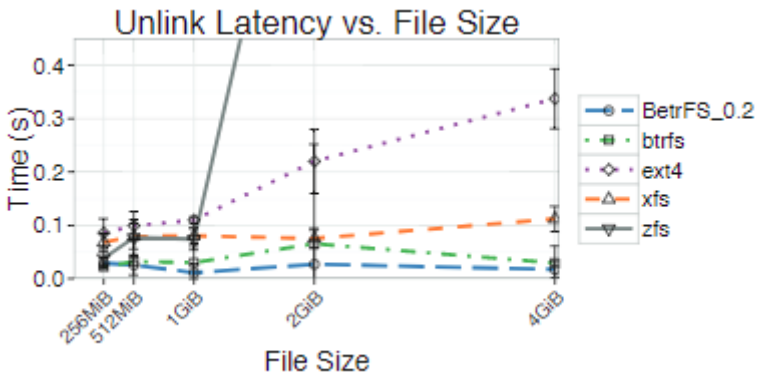


图 7：通过文件大小取消延迟延迟

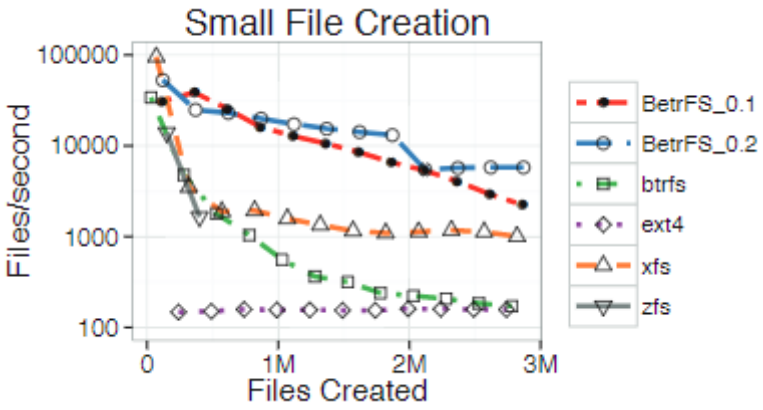


图 8：使用 4 个线程持续创建 300 万个 200 字节的文件。 越高越好，y 轴是对数级。

8 相关工作

分区。动态子树分割是被设计用于大规模分布式系统的技术中，像 Ceph 的，以减少元数据争用和平衡负载。这些系统分发（1）的元数据的对象的数量和（2）的元数据的访问，跨节点的频率。区域，而不是根据其聚合分区中的对象*的大小*与结合的重命名成本。

引入了基于 KD-树木多维元数据索引的分区技术。分割技术也被用来确定哪个数据的推移较慢与更快媒体，以有效地保持倒文件指数，或在不同的存储的数据结构，堵塞，以优化只读或写密集型工作负载。Chunkfs 分隔 ext2 文件系统，以改善恢复时间。一些系统还划分磁盘带宽和性能隔离缓存空间;一般来说，这些系统主要关注的是公平跨越用户或客户端，而不是边界最坏情况下的执行时间。这些技术攻击特定领域的取舍不同的目录搜索分区的平衡和重命名。

IceFS 采用立方体，类似的概念，以区域，以隔离故障，在数据结构和事务机制除去物理的依赖关系，并允许更精细的粒度的回收和轴颈配置。立方体明确用户定义为包括整个目录树，并作为用户添加更多的数据可以尽量增大。相反，区域是对用户完全透明，并动态拆分和合并。

后期绑定日志条目。KVFS 避免了通过创建一个新的 VT-树快照为每个事务写的大部分数据的两倍日记开销。当一个事务提交时，从交易的快照 VT-树内存中的所有数据被提交到磁盘，而交易的 VT-树添加*上述*相关 VT-树。数据未在这种情况下写了两次，但 VT-树成长任意高大，使得搜索性能很难原因有关。

日志结构文件系统避免重复的写入仅写入到日志的问题。这提高了在最好的情况下写的吞吐量，但不会强制执行最佳的查询时间下限。

物理记录存储前和个人数据库页后图像，其可以是用于大的更新或少量更新以大对象昂贵。逻辑日志可减少日志大小时操作有简洁的表述，但不适合大数据的插入。ZFS 意图日志的结合写入时复制更新和间接以避免大的记录，日志写入放大。我们适应这种技术来实现在晚期 0.2BetrFS 大邮件（或大组相关的小消息）的结合日志。

以前的系统已经实现软更新，在那里数据被首先写入，随后的元数据的变化中，从叶到根。这种方法写的订单，使磁盘上结构总是一致的入住点。虽然软更新可以在 B 能够B_ε-树，这将是具有挑战性的，后期结合杂志避免大型写入翻一番的问题，但是，不像软的更新，主要是封装在块分配器。后期绑定强加一些额外的要求，在 BB_ε-树本身并不会耽误任何树节点的写入强制排序。因此，后期绑定轴颈是特别适合于 WOD。

9 结论

本文表明写入优化字典可以实际不只是加速特殊情况，但作为一个构建块的通用文件系统。BetrFS0.2 提高某些操作的数量级，并提供性能堪比上所有其他商品的文件系统的订单效果。这些改进是在写入优化字典的设计的基础的进步的产物。我们认为，其中的一些技术可以应用于更广泛的类文件系统这是未来的工作。