海量存储读书报告

黎琳 2017202110036

摘要:本文主要介绍了两种针对元数据的去耦式文件系统方案。CFFS实现了一个复合文件系统，允许文件与元数据由原来的一对一映射转变成多对一映射，在网络服务器和软件开发工作负载下，实验显示能将性能提高达27%。HopsFS则利用无共享、事务、内存中的NewSQL数据库对HDFS进行了改进，将元数据的存储与管理分开，消除了元数据瓶颈实现了更大更高的吞吐量。

关键词：元数据；去耦；CFFS；HopsFS

1. 引言

文件系统性能优化是一个计算机存储深入研究的领域，如何能减小系统开销的同时提升文件系统效率和性能是众多研究者最为关注的问题。元数据是文件系统中用于管理文件数据的重要组件，文件系统的优化离不开对元数据的优化，众多研究者针对元数据管理及存储的优化，提出了各种各样的文件系统性能优化模型和解决方案。但现有的方案在改进文件系统时依然保留了逻辑文件到物理元数据的一对一映射，这种映射方式阻碍了文件系统性能的进一步提高，本文介绍了一种新型文件系统CFFS（复合文件文件系统，Composite-file File System）,该系统将被常被共同访问的小文件组合，将传统的逻辑文件到元数据的一对一映射去耦转变为多对一映射，极大的提升系统性能；此外，大多数文件系统的管理和存储都是放在内存中进行，当面对高吞吐量高工作负载的场景时，系统延迟加重。本文所介绍的另一个HopsFS文件系统，在原有的HDFS上进行改进，将元数据存储于外部数据库，分离其的存储和管理，不仅提高了系统吞吐量降低了延迟，还能够扩展元数据，使得文件系统能够应用到更多场景中。

本文介绍了关于文件系统优化的两个解决方案。第1节引言介绍了文件系统优化的背景和现状，而后引出本文所关注的两类针对元数据进行文件系统性能改进的去耦式方案：CCFS和HopsFS。第2节介绍了关于文件系统优化的一些相关工作。第3节则介绍了CFFS（Composite-File File system）的设计动机、CCFS的设计细节包括复合文件创建以及三个复合文件整合策略（基于目录、基于嵌入引用、基于频次挖掘的整合策略），以及实验与评估。第4节则介绍了HopsFS（HDFS的优化版本）涉及到的相关概念HDFS和NDB、HopsFS概述及操作以及HopsFS的评估。最后，第5节对本文进行总结。

1. CFFS(Composite-file File System)

现有的文件系统优化技术（例如缓存、更好的数据布局），通常保留了逻辑文件到其物理元数据表示的一对一映射，例如每个文件在UNIX平台上与其自己的i-节点相关联。选取这样的映射方式是因为许多存储组件和机制（例如VFS API，预取和元数据高速缓存）依赖于这样的数据结构，但也正是因为这样的映射方式阻碍了文件系统性能的进一步优化。本节从设计动机、CFFS设计、实验与评估三个方面来具体介绍CFFS。

2.1 设计动机

CFFS的出现主要是为了解决以下四个问题以提高文件系统性能。

2.1.1 频繁访问小文件

研究显示> 80％的访问是对小于32个字节的文件，同时访问磁盘上的小文件的访问时间的40％可归因于对元数据的访问，如果能减少这种访问开销就能极大的提高性能。

2.2.2 冗余元数据信息

传统文件与其自身的物理元数据相关联，该元数据跟踪诸如文件块的位置，访问许可等的信息，但事实上许多文件共享相似的文件属性，如文件所有者的数量，权限模式，造成了元数据信息冗余。而研究显示典型工作站的元数据压缩率可以高达75％。

2.3.3 分组访问的文件

访问文件时，许多文件相邻或者相关的文件往往会被一起访问，比如网络访问通常会涉及访问许多相关联的文件（例如HTML）。虽然表面上看来文件分组是一个有效的提高性能的做法，事实上识别和分组文件的过程会增大开销，抵消掉文件分组所带来的性能增益。

2.3.4 预取限制

虽然预取是一种有效的优化方式，但是单独获取每个文件和相关的元数据访问会带来很高的开销。因此，基于上述四个问题提出了CFFS，通过整合被共同访问的小文件，解耦逻辑文件和物理元数据之间的一对一表示以此提高性能。

2.2 CFFS设计

CFFS引入了一个称为复合文件（composite file）的内部物理表示，用于保存经常被共同访问的小文件的内容，并且与多个小文件所共享的单个复合i节点相关联，值得注意的是，该复合文件对终端用户是不可见的。存储在小文件inode中的原始信息经过去重操作之后，存储为复合文件的扩展属性，扩展属性记录了每个小文件在复合文件内的位置，利用这种物理表示方法，CFFS可以物理复合文件转换为逻辑文件。本节从复合文件创建和复合文件整合策略两个方面具体介绍了CFFS。

2.2.1 复合文件创建

复合文件创建：创建复合文件时，CFFS分配一个i节点并复制连接子文件的内容作为其数据。 该复合文件会将各个子文件的复合文件偏移量和大小以及重复数据消除的i节点信息记录到其扩展属性中。然后原始子文件被截断，其目录条目被重新映射到复合文件的i节点，还包括子文件ID和其原始i节点的重新分配。 因此，终端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。具体过创建程如图2.1所示。



图2.1 复合文件创建过程

2.2.2 复合文件整合策略

上面介绍了复合文件构建的过程，复合文件是由一个个的小文件组合形成的，但是如何选取用于组合复合文件的小文件呢，为此,CFFS配置了以下三种方式：基于目录的合并、嵌入式引用整合、基于频次挖掘整合。

（1）基于目录的整合

文件系统围绕目录来实现空间局部优化，因此目录是一种常用的文件访问模式，适合用其来形成组合文件，CCFS可以无需跟踪和分析文件引用，直接在所由目录上执行基于目录的整合（并不包括子目录）。但是，该策略并不能捕获跨目录的文件关系。

（2）基于嵌入式引用整合

嵌入式引用整合是基于文件中嵌入的文件引用来标识复合文件成员资格。例如，html文件中被嵌入了超链接，网络爬虫便可以通过这些链接访问每个网页，在这种情况下便可以将原始的html文件和引用的文件合并。再比如编译的过程，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可以定期筛选修改的文件以协调复合文件成员资格。基于嵌入式参考的策略可以识别跨目录访问的相关文件，但是它需要知道特定的文件格式，难以提取不是基于文本的文件格式（例如，html，源代码）的嵌入式文件引用。

（3）基于频次挖掘整合

基于频率挖掘的整合策略主要是利用Apriori算法实现的，根据Aprior属性，如果经常访问一组文件，它的子集也会被经常访问。图2.2解释说明了对文件A，B，C，D和E的访问流的算法。

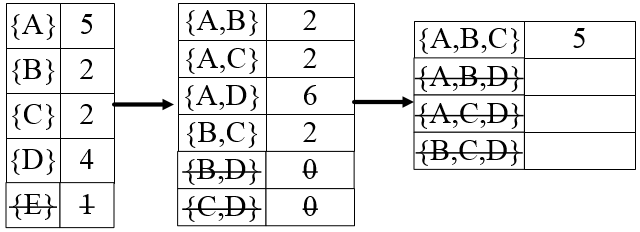


图2.2 Apriori算法

初始阶段：首先，我们计算每个文件的访问次数，然后删除计数小于阈值（如两个）的文件以供进一步分析。（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）

第二阶段：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，我们都增加文件集{A，B}的计数。计数小于阈值的集将会被被删除，例如{B，D}和{B，D}

第三阶段：我们可以根据剩余的双文件引用集生成所有三文件引用集集。但是，根据之前提到的Apriori的属性，如果一个三文件引用集经常出现，那么它所有的双文件引用集也会频繁出现。因此，{A，B，D}等文件集被删除，因为在第二阶段中{B，D}就已经被删除了。

终止阶段：由于我们不能再生成四个文件的引用集，算法结束，由于{A，B}、{A，C}，{B，C}等是是{A，B，C}的子集所以将它们除去，返回集合{A，B，C}和{A，D}作为两个频繁访问集。该方法不允许出现重叠的文件集，目的是为了保持一致性和避免复制的复杂性。

基于频率挖掘的整合策略可以基于动态文件引用来识别复合文件候选成员，但是，运行它的成本阻碍了其应用在更受欢迎的文件参考序列中。

2.3 实验与评估

2.3.1 实现

复合文件成员资格生成器和CFFS组成了系统原型，文章在Linux平台上的用户空间中运行FUSE框架，创建了用C++语言实现的CFFS原型。表1是实验平台介绍，图3则说明了CFFS组件和从应用层到ext4的数据路径。

表2.1 实验平台

|  |  |
| --- | --- |
| Processor | 2.8GHz Intel® Xeon® E5-1603, L1 cache 64KB, L2 cache 256KB, L3 cache 10MB |
| Memory | 2GBx4, Hyundai, 1067MHz, DDR3 |
| Disk | 250GB, 7200 RPM, WD2500AAKX with 16MB cache |
| Flash | 200GB, Intel SSD DC S3700 |



图2.3 数据路径

CFFS定期与生成器工具进行交流协商以创建新的复合文件（因此需要拦截所有与文件系统相关的调用以便于更新各个子文件的时间戳），利用类似于硬链接的机制来使得多个文件名映射到相同的复合文件i节点，该i节点及其扩展属性由底层的ext4文件存储。

对于基于目录的整合策略，我们使用Perl脚本将目录中的所有文件列为复合文件成员。对于基于嵌入式引用的整合策略，主要关注两类场景：对于Web服务器工作负载，合并HTML文件及其引用的文件，在复合文件成员资格冲突的情况下，首选index.html，然后是首先包含文件的html；对于源代码编译，利用Makefile作为参考来整合源代码文件。对于基于频率挖掘的方案，成员资格生成器工具可以使用http访问日志或strace输出。

2.3.2 性能评估

文章基于从网络服务器收集的http日志和软件开发工作台的strace数据，进行了重放实验，通过FUSE比较了部署在的CFFS和作为基准的ext4文件系统。在重放实验中，对于磁盘的Web服务器请求延迟，嵌入式引用整合表现最好，缓存命中率达到62％，比ext4高20％，整体重播时间也减少了20％左右；基于目录的整合也可以将缓存命中率提高15％，而基于频率挖掘的整合则表现次之。图4显示了具体结果。

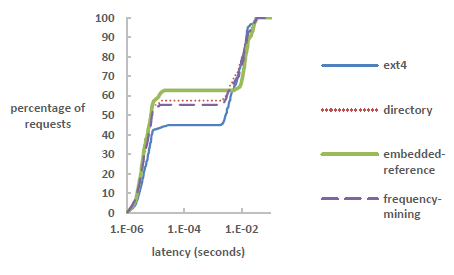


图2.4 实验结果

2.3.3 讨论与未来工作

通过使用不同的存储介质，由于IO数量的减少有利于读操作和读写操作的进行，提升了系统整体性能。未来工作的重点将放在寻找形成复合文件的策略，并量化它们与性能贡献的不同组件之间的相互作用，以及更全面的探索元数据的压缩、并发和安全。

3.HopsFS

HopsFS是对Hadoop分布式文件系统（HDFS，Hadoop Distributed File System）的改进优化，它用建立在NewSQL数据库上的分布式元数据服务取代HDFS内存中单节点内存中元数据服务。通过消除元数据扩展瓶颈，与HDFS相比，HopsFS是一个支持高吞吐量和低操作延迟的分布式文件系统。本章从相关概念、设计思路、实验设计与评估、Hops与HDFS四个方面来介绍HopsFS。

3.1相关概念

3.1.1 Hadoop Distributed File System

HDFS是Google文件系统（GFS）的开源实现，它将文件分割为默认大小为128MB的块，并将块复制到多个数据节点以实现高可用性（默认有为个副本）。在HDFS中，活动命名节点（ANN，Active NameNode)负责管理文件系统命名空间,服务于客户端发送的文件系统请求，以及许多存储和管理文件块的数据节点（高达5-10K）。为了确保元数据管理服务的高可用性，命名节点将对元数据的更改记录到日志服务器中，备用名称节点（SbNN，Standby Namenode）稍后读取日志并异步重新应用， ZooKeeper 则用于协调从活动命名节点到备用命名节点的故障转移，以及提供决定命名节点是活动的还是备用的协议。

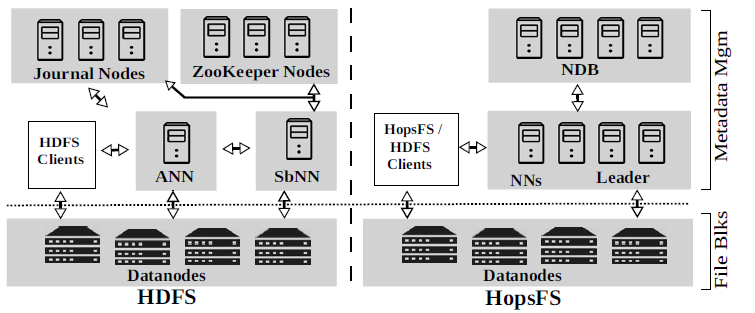


图3.1 HDFS和HopsFS

但HDFS的设计遭遇了两个扩展瓶颈：首先，是命名节点的设计使得元数据的可扩展性受到限制。在内存中的单个服务器——命名节点，管理着与文件系统命名空间和内务管理功能相关的元数据，但命名节点的Java虚拟机（JVM）体系结构对命名空间（文件/目录）的大小有实际的限制，超过200GB，垃圾收集效应会导致严重的性能下降。其次，为了限制对单个writer或多个reader的并发访问来确保文件系统的一致性，在命名空间上施加了单个全局锁，这限制了HDFS在高吞吐量场景下的使用。此外，为了减少内存占用，数据结构都需经过优化，导致了元数据难以修改或者导出到外部系统。

3.1.2 Network Database(NDB)

MySQL Cluster是一个内存中的无共享且高度可用的分布式关系型NewSQL数据库。网络数据库（NDB）是MySQL Cluster的存储引擎。在MySQL Cluster中至少包含一个用于监视和配置集群的管理节点，以及用于存储表数据和处理访问/处理存储数据等事务的多个NDB 数据节点。

与HDFS/HopsFS中的数据节点不同，NDB数据节点是根据复制因子组织成节点组，并且其数据是以面向行（row-oriented）的方式存储在表中，使用表分区键（通常是主键）的哈希值对表中的行进行分片，节点组中的所有NDB数据节点都包含分配给节点组的所有分片的完整副本。NDB支持以下功能：

应用程序定义的分区（ADP，Application Defined Partitioning）：这使得开发人员可以覆盖默认的NDB分区方案，以精细控制表的数据在NDB数据结点上的分布。

分布感知事务（DAT，Distribution Aware Transactions）：根据应用程序定义的分区来指定事务提示，以降低数据库操作的延迟。

3.2 HopsFS概述及操作

3.2.1 HopsFS概述

HopsFS是HDFS的一个分支，与HDFS不同，HopsFS通过分离元数据存储和管理服务来提供可向外扩展的元数据层，支持用Java编写的多个无状态命名节点来处理客户端的请求并处理存储在外部分布式数据库中的元数据，如图1所示。每个命名节点都有一个数据访问层（DAL，Data Access Layer）驱动程序，类似于JDBC，封装了所有的数据库操作，允许HopsFS将元数据存储在各种NewSQL数据库中。Hops通过选择一个领导命名节点（leader namenode）协内部管理操作，例如数据节点故障处理，具体是通过使用数据库作为共享内存来实现leader选举和成员管理，领导者选举协议为每个命名节点分配唯一的ID，并且当命名节点重新启动时将改变命名节点的ID。与HDFS一样，数据节点连接到所有的命名节点，然而数据节点只会给一个命名节点发送块报告，领导命名节点则负责平衡所有活动命名节点上的块报告。

3.2.2 元数据分区

分层分布式文件系统的元数据通常包含有关inode，块，副本和映射（目录到文件，文件到块和块到副本）的信息。当元数据被分布时，需要一个应用程序定义的分区方案对元数据进行分片并且需要共识协议以确保跨越分片的操作的元数据完整性。

首先介绍元数据模型中描绘关键实体的实体关系模型，以了解分区方案所立足的基础。在HopsFS中，文件系统元数据存储在表中，而目录i节点是由i节点表中的单个行表示的，图3.2显示了描述HopsFS元数据模型中关键实体的实体关系模型。文件和目录由i节点实体表示，其中包含对文件系统层次结构中其父索引节点（父索引节点ID）的引用，在i节点实体中存储的不是完整的路径，而是路径中的各个组件；每个文件都包含存储在Block实体中的多个块，每个块副本的位置存储在副本实体中。在一个块的生命周期中，如果数据节点发生故障并且这些块存储在未复制块表（URB，under-replicated blocks）中，块可能会被低复制，即被复制的数量少于原始block的数量；正在进行复制的块存储在挂起的复制块表（PRB,pending replication blocks table）中。同样，块的复制品在其生命周期中也有不同的状态。当一个副本被损坏时，它被移动到损坏的副本（CR，corrupted replicas）表；每当客户端写入新块的副本时，该副本将移动到正在构建的副本（RUC,replica under construction）表中；如果块的副本太多，例如恢复了包含重新复制块的数据节点，则额外的副本将存储在多余的副本（ER，excess replicas）表中，并且计划删除的副本将被存储在失效（Inv,invalidation）表中。

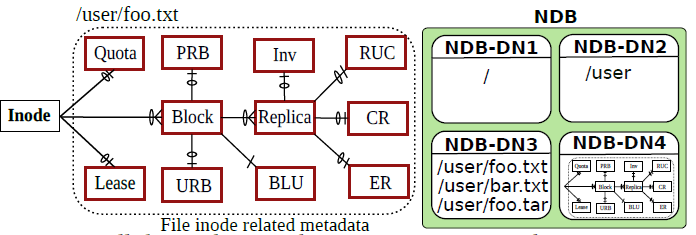


图3.2 实体关系模型

HopsFS通过其父节点的i节点ID对i节点进行分区，从而使得具有相同父节点inode的i节点存储在同一个数据库分片上，能够统一分配元数据并且有效地执行目录列表操作。在目录中列出文件时，首先使用提示机制来启动位于数据库碎片上的事务协调器上的事务，该事务协调器保存该目录的子索引节点，然后，使用修剪的索引扫描来检索本地目录的内容。文件inode相关的元数据，即块，副本映射和校验和，也都是使用文件的i节点ID分区，因此给定文件的元数据全部存储在单个数据库碎片中，可以实现高效的文件操作。

3.2.3 元数据一致性维护

在HopsFS中为了维护文件系统元数据的一致性，采取了将文件系统操作转换为分布式事务的策略，并使用每个事务（per-transaction）和节点级缓存来降低分布式事务的延迟。事务性元数据操作包括以下两类：对单个文件，目录或块进行操作的i节点操作例如创建/读取文件，创建目录或者更改块的状态；对未知数量的i节点的子树操作，例如递归删除、移动、更改权限等操作。HopsFS实现了悲观并发模型，该模型支持对命名空间进行并行读写操作，i节点和子树操作。与乐观并发模型相比，它在中高级别的资源利用方面表现出了良好的性能。

（1）i节点操作

i节点操作封装在由三个不同阶段组成的单个事务中，即锁定，执行和更新阶段。

在锁定阶段，元数据被锁定并从数据库中读取，并且在事务处理期间也需要锁定。i节点操作是基于路径的，如果它们不是只读操作，则它们只修改路径的最后一个组件，因此，只有文件路径的最后一个组件被锁定才能进行文件系统操作。图3.3显示了HopsFS中i节点操作的事务模板。

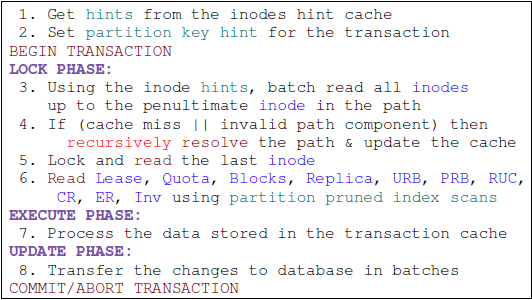


图3.3 事务模板

从数据库读取的所有数据都存储在每个事务缓存（快照）中，该缓存保留数据和数据库之间的更新缓存记录，由于元数据通常在同一事务中被多次读取和更新，因此缓存可以节省许多往返数据库的时间。元数据的行级锁定确保了缓存的一致性，即没有其他事务可以更新元数据。当事务完成释放锁定时，缓存将被清除。

i节点操作是通过处理每个事务缓存中的元数据来执行的。在第二阶段生成的更新和新的元数据存储在/最后的更新阶段中批量发送到数据库的缓存，然后事务被提交或回滚。

（2）子树操作

包含数百万个i节点的大型目录的递归操作太大，无法利用单个事务完成。因此HopsFS采用了提供一致性语义的协议，使得在批量事务中递增执行子树操作，同时采用应用程序级分布式锁定机制来标记和隔离子树。子树操作有以下几个阶段：在第一阶段，在子树的根节点上获得排他锁，并且在数据库中设置并保存子树锁标志，该标志表示该子树的所有后代都被独占锁定。在第二阶段，通过以相同的总次序锁定i节点来获取和释放数据库写入锁，等待所有进行中的i节点操作完成后停顿子树；在最后阶段，文件系统操作被分解为并行执行的较小操作。

3.3 HopsFS评估

HopsFS解决了如何扩展HDFS的元数据层的问题，基于大型生产集群中的实际文件系统负载，实验比较测试了HDFS和HopsFS中命名节点的性能和可扩展性。在实验中，HopsFS部署运行在Dell PowerEdge R730xd服务器，利用了22个线程配置了12个节点，而ApacheHDFS部署在5台服务器，并利用了240个客户端处理程序线程配置了HDFS命名节点，基于工业工作负载轨迹利用基准测试工具来测试文件系统工作负载的性能。由于实验中只是评估元数据性能，使用非空文件进行测试不仅不能获得更多信息，还会需要更多数量级的HDFS/HopsFS数据节点，因此实验中所有的测试文件的长度都为零。

3.3.1 工业工作负载实验

文章使用基于Spotify操作轨迹的工作负载对HopsFS进行基准测试，该轨迹操作一个由包含60PB数据的1600个+节点组成的Hadoop群集，命名空间包含13万个目录和2.18亿个文件，每个文件平均包含1.3个块。图3.4显示，对于实验所用的工业负载，使用60个命名节点和12个NDB节点，HopsFS可以执行每秒125万次运算，吞吐量能达到HDFS的16倍。即使使用等效的硬件（2个NDB节点和3个名称节点），HopsFS的吞吐量也比HDFS高出10％。

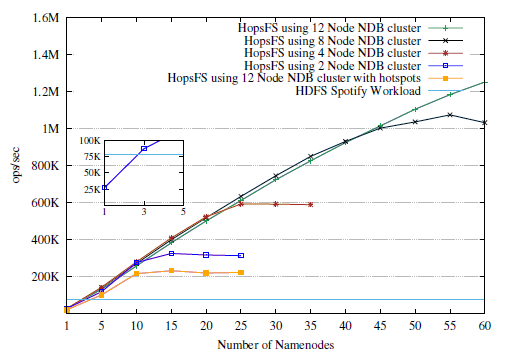


图3.4 吞吐量比较

3.3.2 元数据扩展性

在HDFS中，由于整个命名空间元数据必须放在单个JVM的堆上，所以对数据结构进行了高度优化以减少内存占用，使得元数据难以扩展。下图3.5显示了HDFS和HopsFS的元数据可伸缩性，HopsFS可以使用24TB的元数据存储多达170亿个文件，比HDFS高出37倍。

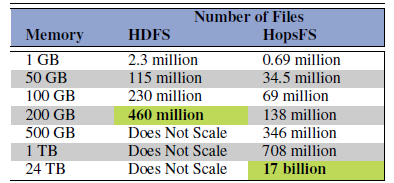


图3.5 元数据扩展性

3.3.3 文件系统操作吞吐量

在此实验中，对于每个文件系统操作，基准实用程序都会使用相同的文件系统操作来淹没命名节点，以此确定特定文件系统操作的最大吞吐量和可扩展性。图3.6显示了比较不同文件系统操作的吞吐量的结果。对于每个操作，HopsFS的结果显示为堆叠矩形的条形图。对于所有文件系统操作，HopsFS的性能优于HDFS，对于最常见的文件系统操作，其性能也是远优于HDFS。

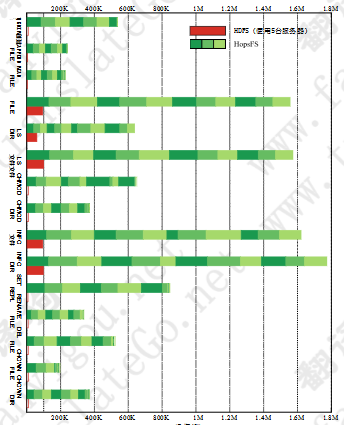


图3.6 文件系统操作吞吐量

3.3.4 操作延迟

同时运行Spotify工作负载，HopsFS比HDFS具有更低的操作延迟，因为在更新命名空间的HDFS文件系统操作中阻止了所有其他文件系统操作。大型HDFS部署可能有成千上万的客户端，并且客户端观察到的端到端延迟会随着文件系统操作在命名节点 的RPC调用队列中等待而增加。相比之下，HopsFS可以处理更多的并发客户端，同时保持较低的运行延迟。

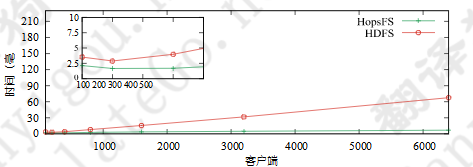


图3.7 操作延迟

4.结论

本文介绍了两种针对元数据的去耦式文件系统优化方案。CCFS复合文件系统合并经常被共同访问的文件元数据，将逻辑文件与物理元数据之间一对一的映射方式去耦为多对一的映射方式，并通过实验证明能够提高27%的性能；HopsFS则通过利用外部NewSQL数据库存储其元数据，分离了元数据的存储和管理，与HDFS相比不仅扩展了元数据，还提高了系统吞吐量和降低了时间延迟，是一个经济有效的分布式分层文件系统。总体看来，这两种优化方案各有优缺点，从不同的角度针对优化问题提出了两种不同的解决方案，但都是对元数据的改进以此实现对文件系统的改进，未来工作可以尝试将两者互相取长补短，提出一个更为优秀的文件系统优化方案。

参考文献

Niazi S, Ismail M, Haridi S, et al. HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases[C]//FAST. 2017: 89-104.

Ismail M, Niazi S, Ronström M, et al. Scaling HDFS to more than 1 million operations per second with HopsFS[C]//Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. IEEE Press, 2017: 683-688.

Zhang S, Catanese H, Wang A I A. The Composite-file File System: Decoupling the One-to-One Mapping of Files and Metadata for Better Performance[C]//FAST. 2016: 15-22.