武汉大学海量存储课程论文

HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases

HopsFS: 基于 NewSQL 数据库扩展分层文件系统元数据

院（系）名 称 ： 计算机学院

专 业 名 称 ： 计算机技术 5 班

学 生 姓 名 ： 王嘉伟

学 生 学 号 ： 2017282110205

二〇一七年十二月

摘　要

目前云存储的主要消耗来自于数据传输和数据存储的消耗，而分布式文件系统的主要受到单节点内存中元数据服务的限制。结合关于云存储的论文“Knockoff: Cheap version in the cloud”和关于分布式文件系统的论文“HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databased”,我认为可以实现一个存储消耗小、高吞吐量、文件操作延迟低且不受内存限制的云存储服务。Knockoff 主要是提出了一种上传日志文件来代替数据文件的方法，来减少数据传输和数据存储的消耗；HopsFS 主要是提出了一种新的分布式文件系统框架，使用 NewSQL 数据库集群来代替内存存储元数据，这样元数据不再受到内存大小的限制，并且将元数据的存储和管理分割为两个部分。由于之前演讲 PPT 内容是 Knockoff，所以本次论文主要对 HopsFS 进行介绍。客户端和服务器之间的数据传输和数据存储的方式则是借鉴 Knockoff 论文中的方法，在第二章背景知识中对 Knockoff 进行简单介绍。下面是对 HopsFS 系统的介绍。

最近，由于 NewSQL 数据库的性能和可拓展性得到了改进，关于是否可以使用商品数据库(commodity databases)对分层文件系统的分布式元数据进行管理的研究问题又重新成为热潮。这篇论文主要介绍了 HopsFS，HopsFS 是 Hadoop 分布式文件系统（HDFS）的新一代发行版，它用建立在 NewSQL 数据库上的分布式元数据服务取代 HDFS 的单节点内存中元数据服务(内存空间有限，只能存放一部分元数据，仅可以提供查询服务)。通过消除元数据瓶颈，与 HDFS 相比，HopsFS 能够实现更大、更高的吞吐量集群，元数据容量已经增加到 HDFS 容量的至少 37 倍。在基于 Spotify 工作负载跟踪的实验中，HopsFS 的性能支持 Apache HDFS 吞吐量的 16 到 37 倍。对于许多并发客户端，HopsFS 也具有较低的延迟，并且在故障切换期间不会出现停机的情况。最后，由于元数据现在存储在商品数据库中，因此可以安全地扩展并轻松导出到外部系统进行在线分析和自由文本搜索。

目　录

1　绪论 1
1.1 研究背景及意义 1
1.2 论文的结构安排 2
2　背景知识 3
2.1 HDFS: Hadoop 分布式文件系统 3
2.2 网络数据库(NDB) 3
2.3 Knockoff 4
3　系统概述 6
4　HopsFS 分布式元数据 7
4.1 元数据存储模型 8
5　HopsFS 事务操作 10
5.1 单个文件、目录、块的 inode 操作 10
5.2 大型操作处理(大型子树) 11
5.2.1 子树操作协议 12
5.2.2 失败子树操作处理 13
总结 14

1 绪论

1.1 研究背景及意义

许多大规模数据并行处理系统都是基于分布式文件系统来处理数据。预计在2020 年将会大量出现存储 EB 规模的数据中心。对于如此大规模的数据量，当前的大型分布式分层文件系统需要进行拓展来适应新时代的数据爆炸，而元数据管理服务是限制拓展的瓶颈之一。许多现有的分布式文件系统将元数据存储在单个内存节点或共享磁盘中，例如 GFS，HDFS，QFS，GPFS 等，然而这两者都比较难以拓展，比如说内存节点受限于内存大小，而磁盘受限于读写速度。还有一些系统通过静态分片命名空间并在不同的主机上存储分片来扩展元数据，例如 NFS，AFS，MapR，Coda 等。但当我们跨越不同分区对文件进行操作时会存在问题，特别是文件移动到另外分区的操作。此外，管理员必须将元数据服务器映射到随时间变化的命名空间分片中，这会使文件系统的管理复杂化。

随着 NewSQL 数据库的性能和可扩展性的发展，将分布式文件系统的元数据存储在商品数据库中是十分可行的。而在此之前，传统观点认为在分布式数据库中规范化地存储元数据是十分昂贵的(就吞吐量和延迟而言)。

这篇论文介绍了如何使用 NewSQL 数据库搭建高吞吐量和低延迟的分布式文件系统。HopsFS 是 Hadoop 分布式文件系统（HDFS）的新一代发行版，它分离了文件系统元数据存储和管理服务。HopsFS 将所有的元数据都存储在高度可用的内存中的分布式关系数据库(称为 NDB，一个用于 MySQL 集群的 NewSQL 存储引擎)，并对所有元数据都进行了标准化处理，之所以需要完全标准化，我认为是因为标准化是可以实现分布式元数据共享的关键。HopsFS 设计了多个无状态服务器(stateless servers)，称为 namenode(NN)，并行读取和更新存储在数据库(NBD)中的元数据。

HopsFS 在分布式事务中封装文件系统操作。为了提高文件系统操作的性能，论文利用了传统的数据库技术，如批处理和事务内的写头缓存，以及 NewSQL 数据库中常见的分布感知技术。NewSQL 数据库的分布感知技术包括 (1)元数据分区(对命名空间进行分区，使对应于某个目录的所有子目录/文件的元数据存储在同一数据库分区中，以便高效生成目录列表)和 (2)分配感知事务(启动一个在数据库分

区上的事务，这个数据库分区中存储了文件系统操作所需的全部/大部分元数据)，和 (3)分区修剪索引扫描(扫描操作适用于单个数据库分区)。论文也介绍了一个 inode 缓存，用于更快地解析文件路径。当解析深度为 N 的路径时，缓存命中的话可以将数据库往返次数从 N 减少到 1。

然而大型目录子树上的某些文件系统操作(例如移动和删除)可能太大以至于不适合单个的数据库事务。比如说，由于数据库管理系统规定了在单个事务中操作的数量限制，删除一个包含数百万个文件的文件夹不能使用单个数据库事务执行。那么对于这种大型子树上的操作，论文引入了一种新的协议，即使用应用程序级别的分布式锁机制来分割大型子树。分割大型子树后，文件系统操作也被分解为并行执行的小操作，这样就可以使用并行的数据库事务执行这些操作了。如果执行操作的 namenode 失败，子树操作协议也可以保护命名空间的一致性不被破坏。

HopsFS 是 HDFS 的直接替代品。实验数据显示在基于 Spotify 实际工作负载跟踪的情况下，HopsFS 的吞吐量比 HDFS 高出 16 倍，并在故障切换期间没有停机。对于更多的写入密集型工作负载，HopsFS 提供了 37 倍的 HDFS 吞吐量。HopsFS 是第一个在分布式关系数据库中存储完全标准化元数据的开源分布式文件系统。

1.2 论文的结构安排

第一章是绪论部分，对本文的研究背景及意义进行简单叙述。

第二章是背景知识部分，首先介绍了 Hadoop 分布式文件系统的工作原理，紧接着对网络数据库的优缺点进行简单的概括。最后描述了 Knockoff 系统的原理。

第三章是对 HopsFS 分布式文件系统的概述。

第四章是描述 HopsFS 怎样将元数据进行分区，使数据库访问开销较低。

第五章是 HopsFS 的具体事务操作，并描述了对单独的 inode 操作以及大规模子树的具体操作。

2 背景知识

2.1 HDFS: Hadoop 分布式文件系统

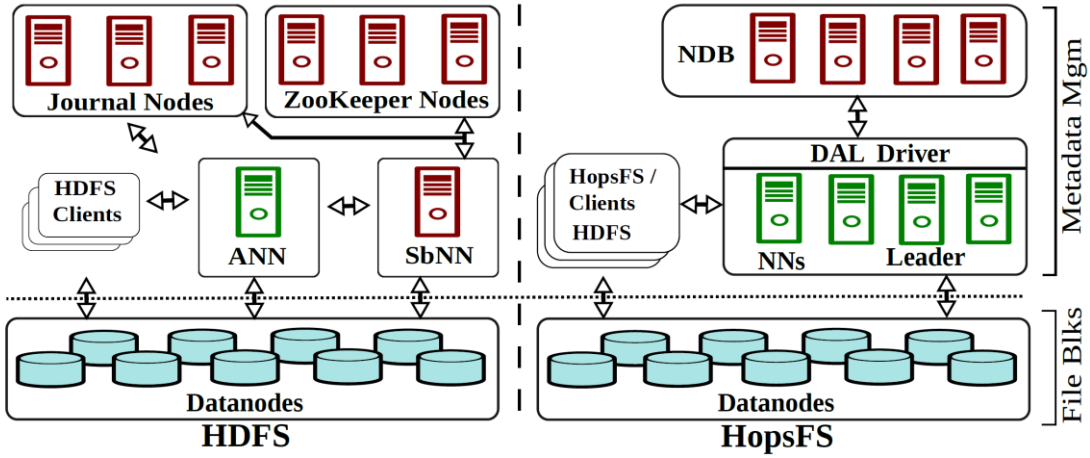

图 1　HDFS 和 HopsFS 系统框架图

HDFS 是 google 文件系统的开源实现。如图 1 所示，HDFS 的元数据存储在称为 Active NameNode(ANN)的进程中，数据文件存储在 datanode 中。文件被分割为块(通常为 128MB)，默认情况下在 datanode 中进行三重复制。为了保证元数据管理服务的高可用性，Active NameNode 使用 quorom-based 的复制方法将元数据的更改记录复制到日志服务器(Journal Node)，并将元数据更改日志异步复制到备用节点(SbNN)，SbNN 还执行 checkpointing 的功能，一旦离上一次已经有一定时间或者一定数量的操作，那么就开始将最新的元数据同步到 ANN 中。ZooKeeper 是 HDFS 的整体监控系统，如果 ANN 宕机后，这时候 ZooKeeper 重新选出 leader 作为 ANN，并且协调从 ANN 到 SbNN 的故障转移。

Datanode 连接 ANN 和 SbNN，多有的 Datanode 定期生成一个块报告，其中包括关于每个 Datanode 的数据块的信息。Namenode 处理块报告以验证 Namenode 块映射与实际存储在 Datanode 上的数据块的一致性。

在 HDFS 中，相对于文件数据，元数据量较小。目前的趋势是更大的 HDFS 集群，但是当前的 JVM 垃圾收集技术和 HDFS Namenode 的单一体系结构是 Hadoop 可扩展性的瓶颈，并且导致元数据难以修改或导出到外部系统。

2.2 网络数据库(NDB)

网络数据库(Network Database)是 MySQL 集群的存储引擎，网络数据库与传统

的数据库相比有以下的特点：

(1)扩大了数据资源共享范围。由于计算机网络的范围可以从局部到全球，因此，网络数据库中的数据资源共享范围也扩大了。

(2)易于进行分布式处理。在计算机网络中，各用户可根据情况合理地选择网内资源，以便就近快速地处理。对于大型作业及大批量的数据处理，可通过一定的算法将其分解给不同的计算机处理，从而达到均衡使用网络资源，实现分布式处理的目的，大大提高了数据资源的处理速度。

(3)数据资源使用形式灵活。基于网络的数据库应用系统开发，既可以采用 C/S（Client/Server，客户机／服务器)方式，也可以采用 B／S(Browser／Server，浏览器/服务器)方式，并发形式多样，数据使用形式灵活。

(4)便于数据传输交流。通过计算机网络可以方便地将网络数据库中的数据传送至网络覆盖的任何地区。

(5)降低了系统的使用费用，提高了计算机可用性。由于网络数据库可供全网用户共享，使用数据资源的用户不一定拥有数据库，这样大大降低了对计算机系统的要求，同时，也提高了每台计算机的可用性。

(6)数据的保密性、安全性降低。由于数据库的共享范围扩大，对数据库用户的管理难度加大，网络数据库遭受破坏、窃密的概率加大，降低了数据的保密性和安全性。

2.3 Knockoff

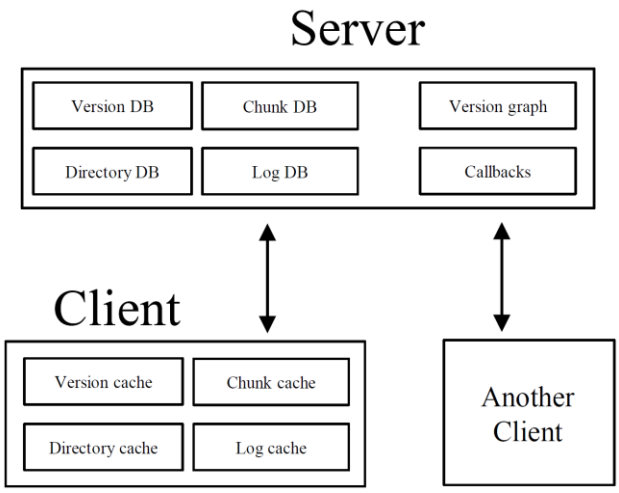

图 2　Knockoff 系统框架图

如图 2 所示，Knockoff 分为客户端和服务器，对于客户端，整个文件版本存

储在版本缓存（Version Cache）中，块缓存（Chunk Cache）存储版本缓存中的每个文件产生的块，目录数据存储在目录缓存（Directory Cache）中。服务器维护与客户端类似的数据。服务器的版本数据库存储每个文件的当前版本，并根据版本策略确定过去的文件版本。此外，服务器还维护客户端为每个文件设置的回调，如果客户端更新文件，则服务器使用这些回调来确定向哪些客户端通知更新信息。

文件写入时，客户端会计算传输原始数据文件和传输日志文件的成本，成本计算公式如式(1)(2)所示：

$$cost_{log} = size_{log} * cost_{comm} + time_{replay} * cost_{comp} \quad (1)$$

$$cost_{data} = size_{chunks} * cost_{comm} \quad (2)$$

如果 $cost_{log}$ 小于 $cost_{data}$，那么就将日志文件传输到服务器，否则传输原始数据文件。

对于文件存储，Knockoff 设置了一个最大重放时延（Maximum Materialization Delay），若当客户端请求文件时，重放日志文件所需的时间大于该时延，则存储原始文件，否则存储日志文件，默认的最大时延是 60s。

3 系统概述

HopsFS 与 HDFS 最大的不同点是 HopsFS 分离了元数据的存储和管理，从而可以提供向外扩展的元数据层。HopsFS 还支持用 java 编写的多个无状态服务器来处理客户端的请求，并处理存储在元数据层的元数据。换句话说，如图 1 所示，HopsFS 将元数据存储于元数据层(NDB)，将可以管理元数据的功能存储在namenode(NN)中，可以看到每个 namenode 都有一个类似于 JDBC 的数据访问层(DAL)驱动程序，封装了所有的数据库操作，这些操作允许 HopsFS 将元数据存储在各种 NewSQL 数据库中。系统内部的管理需要在各个 namenode 之间进行协调，例如 Datanodes 故障处理。那么既然是 namenode 服务器集群，那么应该选取一个负责管理集群的领导者节点来解决问题。HopsFS 使用数据库作为共享内存来实现领导者的选举和成员管理。领导者选举协议为每个 namenode 分配一个唯一的 ID，并且当 namenode 重启时，会重新为它分配 ID。我认为领导者节点的本质其实是一个可以在有限时间内写数据库的活跃的节点，类似于 HDFS 的 ANN。

客户端可以选择要进行文件系统操作的 namenode，可以随机选择，也可以循环或者粘性选择。由于每次重启 namenode 的 ID 都会发生变化，因此客户端要定期刷新 namenode 的列表。HopsFS 和 HDFS 的相似点是 Datanode 连接到所有的namenode，但是 Datanode 只给一个 namenode 发送块报告，领导者节点负载平衡了所有活动的 namenode 的块报告。

4 HopsFS 分布式元数据

分布式文件系统的元数据可以称为数据的数据，主要包括信息节点(inode)、块、块的副本以及映射(目录树到文件，文件到块，块到副本)等信息，用来支持如指示存储位置、历史数据、资源查找、文件记录等功能。由于网络数据库(NDB)是 MySQL 集群的存储引擎，用于存储分布式元数据，那么就需要一个分区方案来对元数据进行分片，并且当某文件系统操作跨越了不同的分区，还需要一致的协议来确保元数据的完整性。那么怎么来选择合适的分区方案呢？

表 1　HDFS 文件系统操作相关频率

Op Name	Percentage	Op Name	Percentage
append file	0.0%	content summary	0.01%
mkdirs	0.02%	set permissions	0.03% [26.3%*]
set replication	0.14%	set owner	0.32 % [100%*]
delete	0.75% [3.5%*]	create file	1.2%
move	1.3% [0.03%*]	add blocks	1.5%
list (*listStatus*)	**9% [94.5%*]**	**stat (*fileInfo*)**	**17% [23.3%*]**
read (*getBlkLoc*)	**68.73%**	**Total Read Ops**	**94.74%**

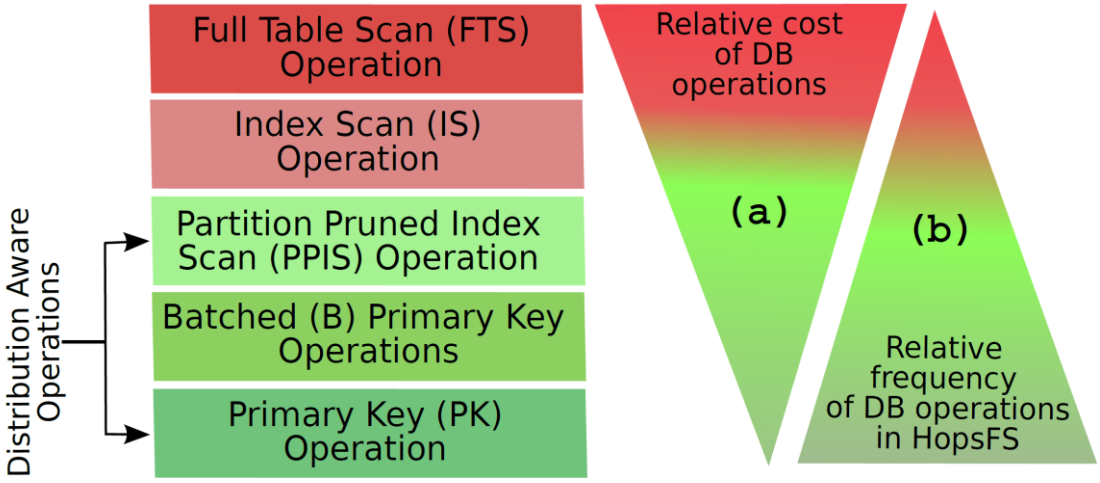

图 3　数据库操作的消耗以及频率

可以从文件系统操作角度来对元数据进行分区。这篇论文统计了 HDFS 中 Hadoop 应用经常进行的操作，发现大部分操作都是 read(读)、list(目录列表)、stat(分析信息)，如表 1。然后又统计了数据库相关操作的频率和消耗，发现一般分区修剪索引扫描操作、批处理逐渐操作和主键操作频率最高，并且消耗是最小的，如图 3。为什么要从操作角度来进行分区呢，可以这样理解，既然 read、list、stat 是最频繁

的文件系统操作，那么分区需要使这三种操作更加便捷更加高效。那么直接来根据文件目录来分区就好了，具有相同父节点的文件或目录的元数据就分到一个分区，这样方便 list 也方便索引。

4.1 元数据存储模型

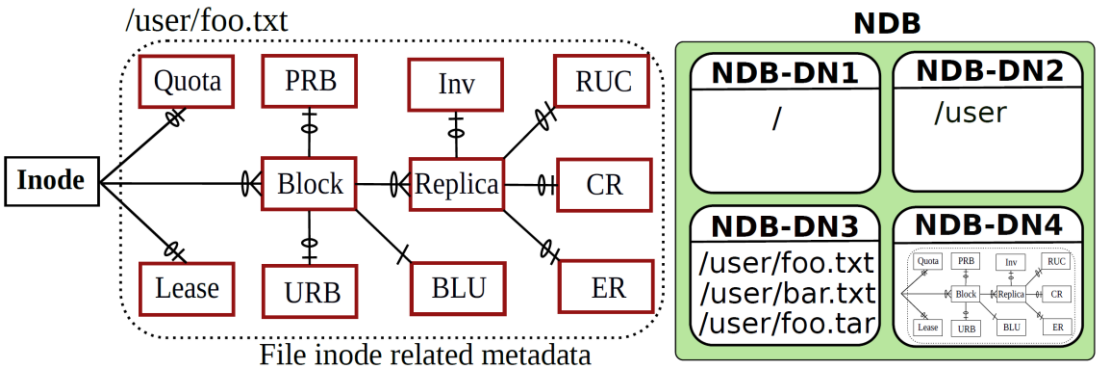

图 4　元数据分区实际模型

在本系统中，元数据存储在数据库表中，那么目录 inode 就由表中的一个单独的行来表示，一般目录 inode 包括两个部分：所包含文件的文件名，以及该文件名对应的 inode 编号。那么目录 inode 用表的单独行表示是说每一行表示一个文件或者一个目录，并且有对应的 ID，也就是 inode 编号。而文件 inode 则包含文件的元信息，比如说文件的字节数、时间戳、权限、文件数据块的位置等信息，那么如果是一个单独的文件而不再是目录的话，那么就用单独的表来存储该文件的元数据。

图 4 是元数据存储在数据库中的一个实际模型。可以看到所有父节点 inode 相同的 inode 都存储在一个分区中，比如说目录“/user”的所有直接子目录或文件都存储在分区 NDB-DN-3 中，这种分区的方式可以使目录列表更加高效的实现。文件 inode 相关的元数据使用文件的 inode 分区，那么一个文件的元数据全部存储在单个数据库分片(表)中，图 4 中 NDB-DN-4 就只存储了文件“/user/foo.txt”的所有元数据。这里需要注意的是，这些 inode 都包含了父索引节点的引用(其实就是主键)，这样才能将文件系统层次结构中的各个分区相连接起来。

图 4 还介绍了每个文件的实体关系模型，每个文件包含存储在 Block 实体中的多个块，每个块的副本存储在副本实体中。每个文件都有一个索引节点的外键，也就是分区的主键，这个主要是方便 HopsFS 来读取文件索引节点相关的元数据。每个块在其生命周期中会经历不同的阶段，比如说如果 datanode 发生故障，那么一些块数据可能不会被复制，那么这些块就存储在未复制表(URB)中，然后领导者

节点就发送命令给 datanode 来创建更多未复制块的副本。正在进行复制的块就存储在挂起的复制块表(PRB)中。当一个副本损坏时，它就被移动到损坏的副本表(CR)中；当写入新块的副本时，副本存储在正在构建的副本表(RUC)中；如果块的副本太多，多余的副本就存在副本表(ER)中；计划删除的副本存储在失效表(Inv)中。

HopsFS 缓存所有的 namenode 的根节点，所有根节点都是存储在同一个数据库分片中的，并且是不可以改变的。所有的路径解析操作都是从第二个路径开始。

5 HopsFS 事务操作

HopsFS 对于元数据的操作可以分为两类，第一类是对单个文件、目录、块的 inode 操作，例如创建/读取文件、创建目录或者块的状态更改；第二类是对一个可能有数百万个 inode 的子树进行操作，例如递归删除、移动目录等。同时由于是分布式文件系统，那么可能存在多个用户同时操作一个 inode 的情况，所以要采用行级锁来对冲突的 inode 操作进行序列化，防止数据共享出现问题。

在大多数 HDFS 工作负载中，解析路径和检查权限是最常见的操作，HDFS 将完整路径解析为单独的部分。在 HopsFS 中，对于深度为 N 的路径，需要在数据库中 N 次往返，这导致文件系统的延迟高。本文使用 inode 缓存的方法，每个 inode 都是一个由父索引节点 ID 和索引节点名称(即文件或目录名称)组成的复合主键，其中父索引节点 ID 充当分区的键。每个 namenode 只缓存 inode 的主键。这样就可以使用仅包含主键查找的单个数据库批量查询来发现对应路径的元数据。

5.1 单个文件、目录、块的 inode 操作

```
 1. Get hints from the inodes hint cache
 2. Set partition key hint for the transaction
BEGIN TRANSACTION
LOCK PHASE:
 3. Using the inode hints, batch read all inodes
    up to the penultimate inode in the path
 4. If (cache miss || invalid path component) then
      recursively resolve the path & update the cache
 5. Lock and read the last inode
 6. Read Lease, Quota, Blocks, Replica, URB, PRB, RUC,
     CR, ER, Inv using partition pruned index scans
EXECUTE PHASE:
 7. Process the data stored in the transaction cache
UPDATE PHASE:
 8. Transfer the changes to database in batches
COMMIT/ABORT TRANSACTION
```

图 5　inode 事务操作模板

HopsFS 采用了一个并发模型来支持并发读写操作、序列化冲突的 inode 和子树操作。对于单个的 inode 事务操作，inode 有三个阶段：锁定、执行以及更新。图 5 显示了 HopsFS inode 事务操作模板。第 1、2 行为准备工作，第 3-6 行是锁定阶段，第 7 行是执行阶段，第 8 行是更新阶段。图 5 第 1 行是说使用 inode 缓存发现文件路径每一部分的主键，例如路径是 rm / etc / conf，那么所发现的主键包括文件夹 rm、etc、conf 的主键。第 2 行是设置事务在某个数据库分片上开始，该分片

保存所有或大部分所需元数据，例如是在 rm 所对应的分片上开始执行事务。第 3 行是说批处理操作读取所有文件路径，一直到倒数第二个没有锁定元数据的路径组件。第 3 行对于深度为 N 的路径，这将删除访问数据库的前 N-1 次往返，大大降低了延迟。如果索引节点的缓存是无效的，那么就使用 HDFS 的方法：递归解析文件路径，并在第 4 行更新索引节点的缓存信息，以便下次使用。

路径解析完成后，路径中的最后一个 inode 组件（第 5 行）将采用共享锁或独占锁，也就是对于本次事务最终要操作的文件或目录进行上锁操作，防止 inode 操作冲突。共享锁用于 inode 只读操作，而独占锁用于 inode 修改操作。此外，根据操作类型和提供的操作参数，使用分区修剪扫描操作（第 6 行），以预定义的顺序从数据库中读取 inode 相关数据（如块，副本和 PRB 等）。

从数据库读取的元数据都存储在每个事务对应的事务缓存中。之所以将数据保存在缓存中，是因为元数据通常在同一事务中会被多次读取和更新，所以缓存可以节省许多往返数据库的时间。需要注意的是，直到该事务结束，对应的事务缓存将保留更新缓存对数据库的更新，也就是说这时更新缓存将不直接对数据库进行更新，而是把更新保留在事务缓存中。元数据的行级锁确保了缓存的一致性，即没有其他事务可以操作元数据。当事务完成后解锁时，缓存将被释放。

inode 操作是通过处理每个事务缓存中的元数据来执行的，比如读写操作等。inode 的更新是指首先事务缓存保留了对元数据的更新，在事务的最后阶段将更新批量发送给数据库，然后事务被提交或回滚。这样做的好处是一次性提交所有的更新，而不是一次一次对数据库进行更新，这样节省了访问数据库的时间，提高了效率。

5.2 大型操作处理(大型子树)

包含数百万个索引节点的大型目录的递归操作已经超出了数据库单个事务所能进行的操作数量，并且不支持在事务中锁定数百万行元数据，所以这种大型操作不可能用单个事务进行操作，例如这样的大型操作包括移动、删除、更改所有者、更改权限以及设置配额操作。移动操作会更改所有后代索引节点的绝对路径，而删除操作会删除所有后代索引节点，设置配额操作会影响所有后代索引节点如何使用磁盘空间或创建多少个文件/目录。同样，更改目录的权限或所有者可能会使在较低子树上执行的操作无效。

5.2.1 子树操作协议

那么怎么处理大型操作？解决方案是在批量事务中递增执行子树操作的协议。子树操作协议不使用行级数据库锁，而是使用应用程序级别的分布式锁机制来标记和分割子树。对于子树操作，我们也需要对其进行序列化，防止出现数据混淆的问题，那么就要求子树中所有正在进行中的 inode 和子树操作要完成于新的子树操作请求执行之前。文中主要是通过执行固定的规则来实现序列化：(1)在当前操作完成之前，没有新的操作访问子树；(2)子树在子树操作开始之前被停顿；(3)当故障发生时，不存在孤立的 inode 或者不一致的 inode。

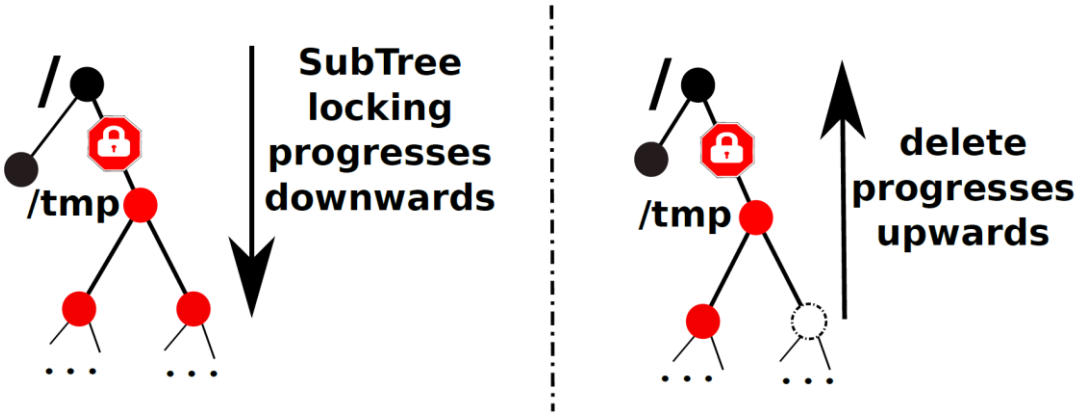

图 6　删除子树操作的执行示例

大型子树操作包括三个阶段：

阶段 1：在第一阶段，独占锁将子树的根节点锁定，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锁定为写入锁定，这样就可以保证在当前操作完成之前，没有新的操作访问子树。在设置锁定之前，我们要确保在该子树较低级别的子树上都没有任何的子树操作。因为设置子树锁可能会使在子树的子集上执行的子树操作失败。因此，我们将所有活跃的子树操作存储在一个表中并进行查询，以确保在子树的较低级别上没有正在执行的子树操作。需要注意的是，在路径解析过程中，当 inode 和子树操作遇到一个启用了子树锁的 inode，inode 和子树操作将中止事务并等待子树锁被移除。

阶段 2：如图 6 所示，在第二阶段，为了使子树停顿，我们通过在所有子树节点上设置和释放数据库写入锁，从而等待所有正在进行的 inode 操作完成。加锁进程是沿着子树向叶节点重复的，并且包含 inode 的树数据结构被建立在 namenode 的内存中。加锁完毕后，批处理子树操作开始从叶节点向根节点方向进行，通过释放锁来对子树进行子树操作，只有释放了锁的节点才可以进行子树操作。举个例子，

现在要对子树进行删除操作，首先我们从根节点向叶节点出发，对该子树的节点进行加锁；在加锁完毕后，使用并行事务的以批处理形式的删除操作从叶节点向根节点出发，如果遇到了某节点是加锁的，那么子树操作就停顿并释放锁，然后将该节点删除。由于是并行事务对大型子树进行操作，效率可以得到保证。

在最后阶段文件系统操作被分解成并行执行的更小的操作。为了提高性能，在每次事务中批量操作 inode。

5.2.2 失败子树操作处理

如果某个操作遇到一个带有子树锁定的 inode，并且该子树的 namenode ID 属于一个失效的 namenode，则清除子树锁定。当一个 namenode 正在执行一个子树操作并且失败了，这时对应的子树不应该处于一个不一致的状态。在第二阶段我们将包含 inode 的树数据结构存储在 namenode 的内存中，这个树数据结构可以在操作失败后保证命名空间的一致性。比如说，再删除操作的情况下，事务以后序遍历的方式删除子树，如果操作中途 namenode 失败，那么未被删除的 inode 将与命名空间树保持连接，HopsFS 客户端将重新提交文件系统操作到另外一个 namenode，对子树的剩余部分继续删除。

总结

本文主要对HopsFS分布式文件系统进行介绍，HopsFS文件系统利用NewSQL数据库集群来代替单内存节点中的元数据服务，这样就不会再受到内存空间大小的限制。并且 HopsFS 还分离了元数据的存储和管理，将元数据存储在数据库中，对元数据管理则是通过设计了多个无状态的服务器，并行读取和更新存储在数据库(NBD)中的元数据。通过消除元数据瓶颈，与 HDFS 相比，HopsFS 能够实现更大、更高的吞吐量集群，元数据容量已经增加到 HDFS 容量的至少 37 倍。同时，对于单个文件、目录或块的 inode 操作以及对于大型子树操作，HopsFS 都进行了相应的处理，使用 inode 缓存的方法，大大降低了系统的延迟。

如果利用HopsFS分布式文件系统来搭建云存储平台，根据Knockoff的思想，客户端对服务器上传文件时，可以对文件进行计算，若上传文件消耗较大，那么就改为上传日志文件，否则继续上传数据文件，并且上传的频率也是可控的。当日志文件上传到云中时，服务器还会对存储进行计算，如果重现日志文件的响应时间大于设置的阈值时间，那么就即时将日志文件重现为数据文件并存储，否则直接存储日志文件即可。当客户端请求读取或者修改云中的文件时，基于 HopsFS 的服务器对数据库中元数据进行相应操作，然后返回元数据给客户端，客户端再利用元数据访问云中的文件。