# HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases

# HopsFS：使用NewSQL数据库扩展分层文件系统元数据

**Abstract**

Recent improvements in both the performance and scalability of shared-nothing, transactional, in-memory NewSQL databases have reopened the research question of whether distributed metadata for hierarchical file sys- tems can be managed using commodity databases. In this paper, we introduce HopsFS, a next generation distribu- tion of the Hadoop Distributed File System (HDFS) that replaces HDFS’ single node in-memory metadata service, with a distributed metadata service built on a NewSQL database. By removing the metadata bottleneck, HopsFS enables an order of magnitude larger and higher through- put clusters compared to HDFS. Metadata capacity has been increased to at least 37 times HDFS’ capacity, and in experiments based on a workload trace from Spotify, we show that HopsFS supports 16 to 37 times the through- put of Apache HDFS. HopsFS also has lower latency for many concurrent clients, and no downtime during failover. Finally, as metadata is now stored in a commodity database, it can be safely extended and easily exported to external systems for online analysis and free-text search.

对于无共享，事务型的，内存不足的内存NewSQL数据库的性能和可扩展性的最近的改进重新开启了关于分层文件系统的分布式元数据是否可以使用商品数据库进行管理的研究问题。在本文中，我们介绍HopsFS，它是Hadoop分布式文件系统（HDFS）的下一代发行版，用于将HDFS的单节点内存元数据服务替换为基于NewSQL数据库的分布式元数据服务。通过去除元数据的瓶颈，与HDFS相比，HopsFS启用了一个数量级更大和更高的通用集群。元数据容量已经提高到HDFS容量的至少37倍，在基于Spotify（全球最大的正版[流媒体](https://baike.baidu.com/item/%E6%B5%81%E5%AA%92%E4%BD%93)音乐服务平台？）的工作量跟踪的实验中，我们显示HopsFS支持（是）Apache HDFS的16到37倍。 HopsFS对许多并发客户端的延迟更低，在故障转移期间也不会有停机时间。最后，随着元数据现在存储在商品数据库中，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。

## Introduction

Distributed file systems are an important infrastructure component of many large scale data-parallel processing systems, such as MapReduce [13], Dryad [27], Flink [5] and Spark [77]. By the end of this decade, data centers storing multiple exabytes of data will not be uncommon [12, 47]. For large distributed hierarchical file systems, the metadata management service is the scalability bottleneck [62]. Many existing distributed file systems store their metadata on either a single node or a shared-disk file systems, such as storage-area network (SAN), both of which have limited scalability. Well known ex- amples include GFS [17], HDFS [61], QFS [41], Far-site [3], Ursa Minor [2], GPFS [58], Frangipani [67], GlobalFS [50], and Panasas [73]. Other systems scale out their metadata by statically sharding the namespace and storing the shards on different hosts, such as NFS [44], AFS [36], MapR [64], Locus [49], Coda [57], Sprite [40] and XtreemFS [26]. However, statically sharding the namespace negatively affects file system operations that cross different shards, in particular move operation. Also, it complicates the management of the file system, as ad- ministrators have to map metadata servers to namespace shards that change in size over time.

分布式文件系统是许多大规模数据并行处理系统的重要基础设施组件，如MapReduce [13]，Dryad [27]，Flink [5]和Spark [77]。到本十年底，到本十年底，存储多个数据的数据中心并不少见[12,47]。对于大型分布式分层文件系统，元数据管理服务是可扩展性瓶颈[62]。许多现有的分布式文件系统将其元数据存储在单个节点或共享磁盘文件系统上，如存储区域网络（SAN），这两者都具有有限的可扩展性。众所周知的例子包括GFS [17]，HDFS [61]，QFS [41]，Far-site [3]，Ursa Minor [2]，GPFS [58]，Frangipani [67]，GlobalFS [50] Panasas [73]。其他系统横向扩展他们的元数据通过静态分隔命名空间并将分片存储在不同的主机上，例如NFS [44]，AFS [36]，MapR [64]，Locus [49]，Coda [57]，Sprite [40]和XtreemFS [26]。但是，静态分配命名空间会对跨越不同分片（碎片？）的文件系统操作产生负面影响，特别是移动（move指文件移动的操作嘛？）操作。此外，由于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碎片，所以使文件系统的管理变得复杂化。

Recent improvements in both the performance and scalability of shared-nothing, transactional, in-memory NewSQL [42] databases have reopened the possibility of storing distributed file system metadata in a commodity database. To date, the conventional wisdom has been that it is too expensive (in terms of throughput and latency) to store hierarchical file system metadata fully normalized in a distributed database [59, 33].

In this paper we show how to build a high throughput and low operational latency distributed file system using a NewSQL database. We present HopsFS, a new distribution of the Hadoop Distributed File System (HDFS) [61], which decouples file system metadata storage and management services. HopsFS stores all metadata normalized in a highly available, in-memory, distributed, relational database called Network Database (NDB), a NewSQL storage engine for MySQL Cluster [38, 54]. HopsFS provides redundant stateless servers (namenodes) that in parallel, read and update metadata stored in the database.

最新的无共享内存NewSQL [42]数据库的性能和可扩展性的改进已经重新开放了将分布式文件系统元数据存储在商品数据库中的可能性。到目前为止，传统的观点是，在分布式数据库中将分层文件系统元数据完全归一化（标准化？），这是太贵了（在吞吐量和延迟方面）[59,33]。

在本文中，我们将展示如何使用NewSQL数据库构建高吞吐量和低运行时延分布式文件系统。我们提供HopsFS，这是Hadoop分布式文件系统（HDFS）的新发行版[61]，它解耦文件系统元数据存储和管理服务。 HopsFS将所有元数据归一化在高可用性，内存中，分布式的关系数据库中，称为网络数据库（NDB），一种MySQL群集的NewSQL存储引擎[38,54]。 HopsFS提供冗余的无状态服务器（Namenodes）并行读取和更新存储在数据库中的元数据。

HopsFS encapsulates file system operations in distributed transactions. To improve the performance of file system operations, we leverage both classical database techniques such as batching (bulk operations) and write-ahead caches within transactions, as well as distribution aware techniques commonly found in NewSQL databases. These distribution aware NewSQL techniques include application defined partitioning (we partition the namespace such that the metadata for all immediate descendants of a directory (child files/directories) reside on the same database shard for efficient directory listing), and distribution aware transactions (we start a transaction on the data- base shard that stores all/most of the metadata required for the file system operation), and partition pruned index scans (scan operations are localized to a single database shard [78]). We also introduce an inode hints cache for faster resolution of file paths. Cache hits when resolving a path of depth N can reduce the number of database round trips from N to 1.

**！** HopsFS将文件系统操作封装在分布式事务中。为了提高文件系统操作的性能，我们利用经典数据库技术，如批处理（批量操作）和事务中的预先提前缓存，以及NewSQL数据库中使用的分发感知技术。这些分发感知的NewSQL技术包括应用程序定义的分区？（我们对命名空间进行分区，使得目录的所有即时后代的元数据（子文件/目录）驻留在同一数据库分片上，以实现高效的目录列表）和分发感知事务（我们在这样的一个数据库分片上开始事务，这个数据库分片上存储了文件系统操作所需的所有/大部分元数据），以及分区修剪的索引扫描（扫描操作已本地化到单个数据库分片[78]）。我们还引入了一个inode提示缓存来更快地分辨文件路径。解析深度N的路径时的缓存命中可以将数据库往返次数从N减少到1。

However, some file system operations on large directory subtrees (such as move, and delete) may be too large to fit in a single database transaction. For example, deleting a folder containing millions of files cannot be performed using a single database transaction due to the limitations imposed by the database management system on the number of operations that can be included in a single transaction. For these subtree operations, we introduce a novel protocol that uses an application level distributed locking mechanism to isolate large subtrees to perform file system operations. After isolating the subtrees large file system operations are broken down into smaller transactions that execute in parallel for performance. The subtree operations protocol ensures that the consistency of the namespace is not violated if the namenode executing the operation fails.

但是，大型目录子树（如移动和删除）上的一些文件系统操作可能太大，无法适应单个数据库事务。 例如，由于数据库管理系统对可以包含在单个事务中的操作数量所施加的限制，删除包含数百万个文件的文件夹无法使用单个数据库事务执行。 对于这些子树操作，我们引入了一种新颖的协议，它使用应用级分布式锁定机制来隔离大型子树来执行文件系统操作。 分离子系统之后，大文件系统操作将分解为并行执行的较小事务。子树操作协议确保如果执行操作的namenode失败，则不会违反命名空间的一致性。

HopsFS is a drop-in replacement for HDFS. HopsFS has been running in production since April 2016, providing Hadoop-as-a-Service for researchers at a data center in Luleå, Sweden [63]. In experiments, using a real-world workload generated by Hadoop/Spark applications from Spotify, we show that HopsFS delivers 16 times higher throughput than HDFS, and HopsFS has no downtime during failover. For a more write-intensive workload, HopsFS delivers 37 times the throughput of HDFS. To the best of our knowledge HopsFS is the first open-source dis- tributed file system that stores fully normalized metadata in a distributed relational database.

HopsFS是HDFS的替代品。 HopsFS自2016年4月开始投入生产，为瑞典Luleå数据中心的研究人员提供Hadoop-as-a-Service [63]。 在实验中，使用来自Spotify的Hadoop / Spark应用程序生成的真实工作负载，我们显示HopsFS的吞吐量比HDFS高出16倍，HopsFS在故障转移期间不会有停机时间。 对于更加写密集的工作负载，HopsFS提供了HDFS吞吐量的37倍。 据我们所知，HopsFS是第一个在分布式关系数据库中存储完整归一化元数据的开源分布式文件系统。

1. **Background**

This section describes Hadoop Distributed File System (HDFS) and MySQL Cluster Network Database (NDB) storage engine.

## Hadoop Distributed File System

The Hadoop Distributed File System (HDFS) [61] is an open source implementation of the Google File System [17]. HDFS’ metadata is stored on the heap of single Java process called the Active NameNode (ANN), see Figure 1. The files are split into small (typically 128 MB) blocks that are by default triple replicated across the datanodes. For high availability of the metadata management service, the Active namenode logs changes to the metadata to journal servers using quorum based replication. The metadata change log is replicated asynchronously to a Standby NameNode (SbNN), which also performs checkpointing functionality. In HDFS, the ZooKeeper coordination service [25] enables both agreement on which machine is running the active namenode (preventing a split-brain scenario) as well as coordinating failover from the active to the standby namenode.

Hadoop分布式文件系统（HDFS）[61]是Google文件系统的开源实现[17]。 HDFS'元数据存储在名为Active NameNode（ANN）的单个Java进程的堆中，请参见图1.这些文件被拆分为小的块（每块大小128MB），在datanodes上有3个副本。 为了元数据管理服务的高可用性，Active namenode日志将使用基于候选团的副本协议将元数据更改的日志服务器（就是namenade上同时搭建了一个基于候选团协议的日志服务器，用来跟。。。。）。 元数据更改日志异步复制到备用namenode（SbNN），该备用namenode也执行检查点功能。 在HDFS中，ZooKeeper协调服务[25]使得能够在哪台机器上运行主namenode（防止裂脑情景）以及协调主动到备用节点的故障切换。

Figure 1: System architecture for HDFS and HopsFS. For high availability, HDFS requires an Active NameNode (ANN), at least one Standby NameNode (SbNN), at least three Journal Nodes for quorum-based replication of the write ahead log of metadata changes, and at least three ZooKeeper instances for quorum based coordination. HopsFS supports multiple stateless namenodes that access the metadata stored in NDB database nodes.

The namenode serves requests from potentially thousands of datanodes and clients, and keeps the metadata strongly consistent by executing the file system operations atomically. The namenode implements atomic operations using a single global lock on the entire file system metadata, providing single-writer, multiple-readers concurrency semantics. Some large file system operations are not atomic, as they would hold the global lock for too long, starving clients. For example, deleting large directories is performed in batches, with inodes first being deleted, then the blocks are deleted in later phases. Moreover, as writing namespace changes to the quorum of journal nodes can take long time, the global file system lock is released before the operation is logged to prevent other clients from starving. Concurrent clients can acquire the file system lock before the previous operations are logged, preventing starvation, at the cost of inconsistent file system operations during namenode failover. For example, when the active namenode fails all the changes that are not logged to the journal nodes will be lost.

namenode提供潜在的数千个datanodes和客户端的请求，并通过原子执行文件系统操作来保持元数据的一致性。 namenode使用单个全局锁对整个文件系统元数据实现原子操作，提供单写入器，多读取器并发语义。一些大文件系统操作不是原子操作，因为它们会把全局锁保存得太久，饿死客户端。例如，删除大目录是分批执行的，首先删除索引节点，然后在稍后阶段删除这些块。此外，由于将名称空间更改写入日志节点的仲裁时间可能需要很长时间，因此在记录操作之前释放全局文件系统锁以防止其他客户端挨饿。并发客户端可以在先前的操作被记录之前（即写到日志里之前）获取文件系统锁，从而避免当主namenode结点宕机，避免客户端饥饿和文件系统操作不一致的代价。例如，当主namenode结点宕机时，所有未记录到日志节点的更改都将丢失。

The datanodes are connected to both active and standby namenodes. All the datanodes periodically generate a block report containing information about its own stored blocks. The namenode processes the block report to vali- date the consistency of the namenode’s blocks map with the blocks actually stored at the datanode.

In HDFS the amount of metadata is quite low relative to file data. There is approximately 1 gigabyte of metadata for every petabyte of file system data [62]. Spotify’s HDFS cluster has 1600+ nodes, storing 60 petabytes of data, but its metadata fits in 140 gigabytes Java Virtual Machine (JVM) heap. The extra heap space is taken by temporary objects, RPC request queues and secondary metadata required for the maintenance of the file system. However, current trends are towards even larger HDFS clusters (Facebook has HDFS clusters with more than 100 petabytes of data [48]), but current JVM garbage collection technology does not permit very large heap sizes, as the application pauses caused by the JVM garbage collector affects the operations of HDFS [22]. As such, JVM garbage collection technology and the monolithic architecture of the HDFS namenode are now the scalability bottlenecks for Hadoop [62]. Another limitation with this architecture is that data structures are optimized to reduce their memory footprint with the result that metadata is difficult to modify or export to external systems.

datanodes连接到主和备namenode节点。所有datanodes周期性地生成包含有关其自己的存储块的信息的块报告。 namenode处理块报告以验证namenode的块映射与实际存储在数据库中的块的一致性。

在HDFS中，相对于文件数据，元数据量相当低。对于每一PB文件系统数据，大约有1 GB的元数据[62]。 Spotify的HDFS集群拥有1600多个节点，存储60 PB的数据，但其元数据需要140 GB的Java虚拟机（JVM）堆。临时对象，文件系统维护所需的RPC请求队列和辅助元数据占用额外的堆空间。然而，目前的趋势是朝着更大的HDFS集群（Facebook拥有超过100 PB的数据的HDFS集群[48]），但目前的JVM垃圾回收技术不允许非常大的堆大小，因为JVM垃圾收集器引起的应用程序暂停会影响HDFS的操作[22]。因此，JVM垃圾收集技术和HDFS namenode的单片架构现在是Hadoop的可扩展性瓶颈[62]。该架构的另一个限制是数据结构被优化以减少其内存占用，导致元数据难以修改或导出到外部系统。

## Network Database (NDB)

MySQL Cluster is a shared-nothing, replicated, in- memory, auto-sharding, consistent, NewSQL relational database [38]. Network DataBase (NDB) is the storage engine for MySQL Cluster. NDB supports both datanode-level and cluster-level failure recovery. The datanode-level failure recovery is performed using transaction redo and undo logs. NDB datanodes also asynchronously snapshot their state to disk to bound the size of logs and to improve datanode recovery time. Cluster-level recovery is supported using a global checkpointing protocol that increments a global epoch-ID, by default every 2 seconds. On cluster-level recovery, datanodes recover all transactions to the latest epoch-ID.

MySQL集群是一个无共享，复制，内存，自动分片，一致的NewSQL关系数据库[38]。网络数据库（NDB）是MySQL集群的存储引擎。 NDB支持数据库级和集群级故障恢复。使用事务重做(redo)和撤消日志(undo)来执行数据库级别的故障恢复。 NDB数据库还将其状态异步映射到磁盘以限制日志大小并提高数据恢复时间。使用全局检查点协议支持集群级别恢复，默认情况下每2秒增加一个全局epoch-ID。在集群级恢复时，datanodes将所有事务恢复到最新的epoch-ID。

NDB horizontally partitions the tables among storage nodes called NDB datanodes. NDB also supports application defined partitioning (ADP) for the tables. Transaction coordinators are located at all NDB datanodes, enabling high performance transactions between data shards, that is, multi-partition transactions. Distribution aware transactions (DAT) are possible by providing a hint, based on the application defined partitioning scheme, to start a transaction on the NDB datanode containing the data read/updated by the transaction. In particular, single row read operations and partition pruned index scans (scan operations in which a single data shard participates) benefit from distribution aware transactions as they can read all their data locally [78]. Incorrect hints result in additional network traffic being incurred but otherwise correct system operation.

NDB水平划分存储结点的表，叫做NDB datanodes。 NDB还支持表的应用程序定义分区（ADP）。事务协调器位于所有NDB数据节点上，从而实现数据分片之间的高性能交易，即多分区事务。通过提供基于应用程序定义的分区方案的提示，可以在包含由事务读取/更新的数据的NDB数据结点（datanodes）上启动事务，从而实现分发感知事务（DAT）。特别地，**单行读取操作和分区修剪的索引扫描（单个数据分片参与的扫描操作）受益于分布式感知事务，因为它们可以在本地读取所有数据[78]**。不正确的提示导致额外的网络通信，但是不会影响系统的正常运行。

* + 1. **NDB Data Replication and Failure Handling**

NDB datanodes are organized into node groups, where the data replication factor, R, determines the number of datanodes in a node group. Given a cluster size N, there are N/R node groups. NDB partitions tables (hash partitioning by default) into a fixed set of partitions distributed across the node groups. New node groups can be added online, and existing data is automatically rebalanced to the new node group. A partition is a fragment of data stored and replicated by a node group. Each datanode stores a copy (replica) of the partition assigned to its node group. In NDB, the default replication degree is two, which means that each node group can tolerate one NDB datanode failure as the other NDB datanode in the node group contains a full copy of the data. So, a twelve node NDB cluster has six node groups can tolerate six NDB datanode failures as long as there is one surviving NDB datanode in each of the node groups. To tolerate multiple failures within a node group, the replication degree can be increased at the cost of lower throughput.

NDB数据节点被组织成节点组，其中数据复制因子R确定节点组中的数据节点的数量。给定簇N，有N / R个节点组。 NDB将表（默认为哈希分区）分区为分布在节点组中的一组固定的分区。可以线上添加新节点组，并将现有数据自动重新平衡到新节点组。分区是由节点组存储和复制的数据的片段。每个datanode存储分配给其节点组的分区的副本（副本）。在NDB中，默认的副本为2，这意味着每个节点组可以容忍一个NDB datanode故障，因为节点组中的其他NDB datanode包含数据的完整副本。因此，只要在每个节点组中存在一个幸存的NDB datanode，一个十二节点NDB集群有六个节点组可以容忍六个NDB数据节点故障。为了容忍节点组中的多个故障，可以以较低吞吐量为代价来增加副本的数量。

**2.2.2 Transaction Isolation**

NDB only supports read-committed transaction isolation, which guarantees that any data read is committed at the moment it is read. The read-committed isolation level does not allow dirty reads but phantom and fuzzy (non- repeatable) reads can happen in a transaction [7]. However, NDB supports row level locks, such as, exclusive (write) locks, shared (read) locks, and read-committed locks that can be used to isolate conflicting transactions.

NDB只支持读取提交的事务隔离，这样可以保证在读取时提交任何数据。 读取提交的隔离级别不允许脏读取，但是在事务中可能会发生幻影和模糊（不可重复）读取[7]。 但是，NDB支持行级锁，例如独占（写）锁，共享（读取）锁和可用于隔离冲突事务的读取提交的锁。

**3 HopsFS Overview**

HopsFS is a fork of HDFS v2.0.4. Unlike HDFS, HopsFS provides a scale-out metadata layer by decoupling the metadata storage and manipulation services. HopsFS supports multiple stateless namenodes, written in Java, to handle clients’ requests and process the metadata stored in an external distributed database, see Figure 1. Each namenode has a Data Access Layer (DAL) driver that, similar to JDBC, encapsulates all database operations allowing HopsFS to store the metadata in a variety of NewSQL databases. The internal management (house-keeping) operations, such as datanode failure handling, must be coordinated amongst the namenodes. HopsFS solves this problem by electing a leader namenode that is responsible for the housekeeping. HopsFS uses the database as shared memory to implement a leader election and membership management service. The leader election protocol assigns a unique ID to each namenode, and the ID of the namenode changes when the namenode restarts. The leader election protocol defines an alive namenode as one that can write to the database in bounded time, details for which can be found in [56].

HopsFS是HDFS v2.0.4的分支。与HDFS不同，HopsFS通过解耦元数据存储和操作服务来提供横向扩展的元数据层。 HopsFS支持使用Java编写的多个无状态Namenode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，请参见图1.每个namenode都有一个数据访问层（DAL）驱动程序，与JDBC类似，封装了所有数据库允许HopsFS将元数据存储在各种NewSQL数据库中的操作。内部管理（保管）操作，如datanode故障处理，必须在namenode之间进行协调。 HopsFS通过选择一个负责管理节点的领导者namenode来解决这个问题。 HopsFS使用数据库作为共享内存来实现领导选举和会员管理服务。领导选举协议为每个namenode分配一个唯一的ID，当namenode重新启动时，namenode的ID会更改。领导选举协议将一个活着的namenode定义为可以在有限时间内写入数据库，详细信息可以在[56]中找到。

Clients can choose between random, round-robin, and sticky policies for selecting a namenode on which to execute file system operations. HopsFS clients periodically refresh the namenode list, enabling new namenodes to join an operational cluster. HDFS v2.x clients are fully compatible with HopsFS, although they do not distribute operations over namenodes, as they assume there is a single active namenode. Like HDFS, the datanodes are connected to all the namenodes, however, the datanodes send the block reports to only one namenode. The leader namenode load balances block reports over all alive name- nodes.

In section 4, we discuss how HopsFS’ auto sharding scheme enables common file system operations to read metadata using low cost database access queries. Section 5 discusses how the consistency of the file system metadata is maintained by converting file system operations into distributed transactions, and how the latency of the distributed transactions is reduced using per-transaction and namenode level caches. Then, in section 6, a protocol is introduced to handle file system operations that are too large to fit in a single database transaction.

客户可以选择随机，循环和粘性策略来选择执行文件系统操作的namenode。 HopsFS客户端定期刷新namenode列表，使新的节点能够加入一个可操作的集群。 HDFS v2.x客户端与HopsFS完全兼容，尽管它们不会在Namenode上分发操作，因为它们假设只有一个活动的namenode。像HDFS一样，datanodes连接到所有的namenode，但是datanodes将块报告只发送到一个namenode。领导者namenode负载平衡所有活着的名称 - 节点上的块报告。

在第4节中，我们将讨论HopsFS的自动分片方案如何使常规文件系统操作能够使用低成本数据库访问查询来读取元数据。第5节讨论如何通过将文件系统操作转换为分布式事务来维护文件系统元数据的一致性，以及如何使用每个事务和namenode级别缓存来减少分布式事务的延迟。然后，在第6节中，引入一个协议来处理太大的文件系统操作，以适应单个数据库事务。

## 4 HopsFS Distributed Metadata

Metadata for hierarchical distributed file systems typically contains information on inodes, blocks, replicas, quotas, leases and mappings (directories to files, files to blocks, and blocks to replicas). When metadata is distributed, an application defined partitioning scheme is needed to shard the metadata and a consensus protocol is required to ensure metadata integrity for operations that cross shards. Quorum-based consensus protocols, such as Paxos, provide high performance within a single shard, but are typically combined with transactions, implemented using the two-phase commit protocol, for operations that cross shards, as in Megastore [6] and Spanner [10]. File system operations in HopsFS are implemented primarily using multi-partition transactions and row-level locks in MySQL Cluster to provide serializability [23] for metadata operations.operations.

分层分布式文件系统的元数据通常包含有关inode，块，副本，配额，租约和映射（目录到文件的映射，文件到块和块到副本的映射）的信息。 当分发元数据时，需要应用程序定义的分区方案来分割元数据，并且需要协商一致的协议来确保跨分片的操作的元数据完整性。 基于Quorum的协商协议（如Paxos）在单个分片中提供高性能，但通常与使用两阶段提交协议实现的事务相结合，用于交叉分片的操作，如Megastore [6]和Spanner [10]。 HopsFS中的文件系统操作主要在MySQL Cluster中使用多分区事务和行级锁实现，为元数据操作提供可串行化[23]。

The choice of partitioning scheme for the hierarchical namespace is a key design decision for distributed metadata architectures. We base our partitioning scheme on the expected relative frequency of HDFS operations in production deployments and the cost of different database operations that can be used to implement the file system operations. Table 1 shows the relative frequency of selected HDFS operations in a workload generated by Hadoop applications, such as, Pig, Hive, HBase, MapReduce, Tez, Spark, and Giraph at Spotify. List, stat and file read operations alone account for ≈ 95% of the operations in the HDFS cluster. These statistics are similar to the published workloads for Hadoop clusters at Yahoo [1], LinkedIn [52], and Facebook [65]. Figure 2a shows the relative cost of different database operations. We can see that the cost of a full table scan or an index scan, in which all database shards participate, is much higher than a partition pruned index scan in which only a single database shard participates. HopsFS metadata design and metadata partitioning enables implementations of common file system operations using only the low cost database operations, that is, primary key operations, batched primary key operations and partition pruned index scans. For example, the read and directory listing operations, are implemented using only (batched) primary key lookups and partition pruned index scans. Index scans and full table scans were avoided, where possible, as they touch all database shards and scale poorly.

分层命名空间的分区方案的选择是分布式元数据架构的关键设计决策。我们的分区方案是基于生产部署中HDFS操作的预期相对频率以及可用于实现文件系统操作的不同数据库操作的成本而产生的。表1显示了由Hadoop应用程序（如Pigify，Hive，HBase，MapReduce，Tez，Spark和Giraph）在Spotify生成的工作负载中所选HDFS操作的相对频率。单独的列表，统计和文件读取操作占HDFS集群中≈95％的操作。这些统计数据接近于Yahoo [1]，LinkedIn [52]和Facebook [65]的Hadoop集群发布的工作负载。图2a显示了不同数据库操作的相对成本。我们可以看到，所有数据库分片都参与的全表扫描或索引扫描的成本远高于只有单个数据库分片参与的分区修剪的索引扫描。 HopsFS元数据设计和元数据分区允许仅使用低成本数据库操作（即主键操作，批量主键操作和分区修剪的索引扫描）实现常见文件系统操作。例如，读取和目录列表操作仅使用（批量）主键查找和分区修剪的索引扫描来实现。在可能的情况下，避免了索引扫描和全表扫描，因为它们缩小了查找所有数据库分片的比例。

## 4.1 Entity Relation Model实体关系模型

In HopsFS, the file system metadata is stored in tables where a directory inode is represented by a single row in the *Inode* table. File inodes, however, have more associated metadata, such as a set of blocks, block locations, and checksums that are stored in separate tables.

在HopsFS中，文件系统元数据存储在表中，其中目录inode由Inode表中的单个行表示。 然而，文件inode具有更多关联的元数据，例如存储在单独表中的一组块，块位置和校验和。

Figure 3 shows the Entity Relational model depicting key entities in the HopsFS metadata model. Files and directories are represented by the Inode entity that contains a reference to its parent inode (parent inode ID) in the file system hierarchy. We store path individual components, not full paths, in inode entries. Each file contains multiple blocks stored in the Block entity. The location of each block replica is stored in the Replica entity. During its life-cycle a block goes through various phases. Blocks may be under-replicated if a datanode fails and such blocks are stored in the under-replicated blocks table (URB). The replication manager, located on the leader namenode, sends commands to datanodes to create more replicas of under-replicated blocks. Blocks undergoing replication are stored in the pending replication blocks table (PRB). Similarly, a replica of a block has various states during its life-cycle. When a replica gets corrupted, it is moved to the corrupted replicas (CR) table. Whenever a client writes to a new block’s replica, this replica is moved to the replica under construction (RUC) table. If too many replicas of a block exist (for example, due to recovery of a datanode that contains blocks that were re-replicated), the extra copies are stored in the excess replicas (ER) table and replicas that are scheduled for deletion are stored in the invalidation (Inv) table. Note that the file inode related entities also contain the inode’s foreign key (not shown in Figure 3) that is also the partition key, enabling HopsFS to read the file inode related metadata using partition pruned index scans.

图3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Inode实体表示，它包含对文件系统层次结构中其父inode（父inode ID）的引用。我们在路径条目中存储路径单个组件，而不是完整路径。每个文件包含存储在块实体中的多个块。每个块副本的位置存储在副本实体中。在其生命周期中，一个块经历了各个阶段。如果datanode故障，并且这样的块存储在未被复制块表（URB,Under replicated blocks:指的是副本数少于指定副本数的block数量）中，则这个块就是属于副本数量少于指定副本数的。位于leader namenode上的副本管理器向datanodes发送命令，以创建更多不满足最低副本数量的副本。正在复制的块存储在挂起的复制块表（PRB）中。类似地，块的复制品在其生命周期中具有各种状态。当副本被破坏时，它被移动到损坏的副本（CR）表。每当客户端写入新块的副本时，该副本将被移动到正在构建的副本（RUC）表。如果存在块的太多副本（例如，由于恢复包含重新复制的块的数据库），则额外的副本将存储在多余的副本（ER）表中，并且计划删除的副本存储在无效（Inv）表中。请注意，文件inode相关实体还包含inode的外键（未在图3中显示），也是分区键，使HopsFS能够使用分区修剪的索引扫描读取与文件inode相关的元数据。

## 4.2 Metadata Partitioning

With the exception of hotspots (see the following subsection), HopsFS partitions inodes by their parents’ inode IDs, resulting in inodes with the same parent inode being stored on the same database shard. This has the effect of uniformly partitioning the metadata among all database shards and it enables the efficient implementation of the directory listing operation. When listing files in a directory, we use a hinting mechanism to start the transaction on a transaction coordinator located on the database shard that holds the child inodes for that directory. We can then use a pruned index scan to retrieve the contents of the directory locally. File inode related metadata, that is, blocks, replica mappings and checksums, is partitioned using the file’s inode ID. This results in metadata for a given file all being stored in a single database shard, again enabling efficient file operations, see Figure 3.

除了热点（见下面的小节）之外，HopsFS会通过父节点的inode IDs分区inode，从而导致同一个父节点的inode存储在同一个数据库分片上。 这具有在所有数据库分片之间均匀分割元数据的作用，并且能够有效地实现目录列表操作。 当进行在目录中列出文件的操作时，我们使用提示机制在位于数据库分片上的事务协调器上启动该事务，该数据库分片包含该目录的子索引节点。 然后，我们可以使用修剪的索引扫描来在本地检索目录的内容。 文件inode相关的元数据，即块，副本映射和校验和，使用文件的inode ID进行分区。 这导致给定文件的元数据全部存储在单个数据库分片中，并再次启用高效的文件操作，参见图3。

##### Hotspots（ppt没有这部分）

A hotspot is an inode that receives a high proportion of file system operations. The maximum number of file system operations that can be performed on a ’hot’ inode is limited by the throughput of the database shard that stores the inode. Currently, HopsFS does not have any built in mechanisms for identifying hotspots at run time.

All file system operations involve resolving the path components to check for user permissions and validity of the path. The root inode is shared among all file system valid paths. Naturally the root inode is read by all file system path resolution operations. The database shard that stores the root inode becomes a bottleneck as all file system operations will retrieve the root inode from the same database shard. HopsFS caches the root inode at all the namenodes. In HopsFS, the root inode is immutable, that is, we do not allow operations, such as, renaming, deleting or changing the permissions of the root inode. Making the root inode immutable prevents any inconsistencies that could result from its caching.

热点是一个接收高比例的文件系统操作的inode。 “hot”inode上可以执行的文件系统操作的最大数量受存储inode的数据库分片的吞吐量的限制。目前，HopsFS没有任何内置的机制来在运行时识别热点。

所有文件系统操作包括解析路径组件以检查路径的用户权限和有效性。Root inode在所有文件系统有效路径之间共享。当然，所有文件系统路径解析操作都会读取根节点。存储根节点（root inode）的数据库碎片成为瓶颈，因为所有文件系统操作都将从同一数据库分片中检索根节点。 HopsFS将根节点缓存在所有的namenode节点上。在HopsFS中，根inode是不可变的，也就是说，我们不允许操作，如重命名，删除或更改root inode的权限。使root inode不可变是为了防止可能由其缓存引起的任何不一致。

In HopsFS, all path resolution operations start from the second path component (that is, the top level directories). For the top-level directories, our partitioning scheme in-advertently introduced a hotspot – all top-level directories and files are children of the root directory, and, therefore, resided on the same database shard. Operations on those inodes were handled by a single shard in the database. To overcome this bottleneck, HopsFS uses a configurable directory partitioning scheme where the immediate children of the top level directories are pseudo-randomly partitioned by hashing the names of the children. By default, HopsFS pseudo-randomly partitions only the first two levels of the file system hierarchy, that is, the root directory and its immediate descendants. However, depending on the file system workloads it can be configured to pseudo-randomly partition additional levels at the cost of slowing down move and Ls operations at the top levels of the file system hierarchy.

在HopsFS中，所有路径解析操作都从第二个路径组件（即顶层目录）开始。对于顶级目录，我们的分区方案通常引入了热点 - 所有顶级目录和文件都是根目录的子目录，因此驻留在同一数据库分片上。这些inode的操作由数据库中的单个分片处理。为了克服这个瓶颈，HopsFS使用一个可配置的目录分区方案，其中顶级目录的直接的孩子目录（或文件）通过散列子代的名称进行伪随机分区。默认情况下，HopsFS仅对文件系统层次结构的前两个级别（即根目录及其即时后代）进行伪随机分区。然而，根据文件系统工作负载，可以将其配置为伪随机分区附加级别，代价是减慢文件系统层次结构的顶层移动速度和*ls*操作。

**5 HopsFS Transactional Operations**

Transactional metadata operations in HopsFS belong to one of the two categories: **Inode** operations that oper- ate on single file, directory or block (for example, cre- ate/read file, mkdir, and block state change operations), and **subtree** operations that operate on an unknown num- ber of inodes, potentially millions, (for example, recursive *delete*, *move*, *chmod*, and *chown* on non-empty directo- ries).

This section describes how HopsFS efficiently encap- sulates inode operations in transactions in NDB. The strongest transaction isolation level provided by NDB is *read-committed*, which is not strong enough to provide at least as strong consistency semantics as HDFS which uses single global lock to serialize all HDFS operations. To this end, we use row-level locking to serialize con- flicting inode operations. That is, the operations execute in parallel as long as they do not take conflicting locks on the same inodes. However, taking multiple locks in a transaction could lead to extensive deadlocking and trans- action timeouts. The reasons are:

HopsFS中的事务元数据操作属于两个类别之一：对单个文件，目录或块进行操作的Inode操作（例如，创建/读取文件，mkdir和块状态更改操作）以及对子树的操作，可能是操作未知数量的inodes，可能是数百万个（例如，递归删除，移动，chmod(用来修改某个目录或文件的访问权限:可读可写可执行)和chown(用来更改某个目录或文件的用户名和用户组的)非空目录）。

本节介绍HopsFS如何在NDB的事务中高效地封装inode操作。 NDB提供的最强的事务隔离级别是读取提交的，它不够强大，不能提供至少与使用单个全局锁序列化所有HDFS操作的HDFS一样强大的一致性语义。为此，我们使用行级锁来序列化有冲突的inode操作。也就是说，只要在同一个inode上没有冲突的锁，操作就会并行执行。但是，在事务中占用多个锁可能导致大量的死锁和事务超时。原因是：

Cyclic Deadlocks: In HDFS, not all inode operations follow the same order in locking the metadata which would lead to cyclic deadlocks in our case. To solve this problem, we have reimplemented all inode operations so that they acquire locks on the metadata in the same total order, traversing the file system tree from the root down to leave nodes using left-ordered depth-first search.

Lock Upgrades: In HDFS, many inode operations contain read operations followed by write operations on the same metadata. When translated into database operations within the same transaction, this results in deadlocking due to lock upgrades from read to exclusive locks. We have examined all locks acquired by the inode operations, and re-implemented them so that all data needed in a transaction is read only once at the start of the transaction (see Lock Phase, section 5.2.1) at the strongest lock level that could be needed during the transaction, thus preventing lock upgrades.

循环死锁：在HDFS中，并非所有inode操作都遵循相同的顺序锁定元数据，在这种情况下这将导致循环死锁的发生。 为了解决这个问题，我们重新实现了所有inode操作，以便以相同的顺序获取元数据上的锁定，从根节点遍历文件系统树，使用左次深度优先搜索来释放节点。

锁定升级：在HDFS中，许多inode操作包含读取操作，然后对相同的元数据进行写入操作。 当在同一个事务中转换成数据库操作时，这会导致由于从读取锁定升级到专用(排他)锁的锁定而导致死锁。 我们检查了由inode操作获取的所有锁，并重新实现它们，以便在事务进行期间可能需要最强的锁级的情况下，事务中所需的所有数据在事务开始时只读取一次。从而防止锁升级。

5.1 Inode Hint Cache

Resolving paths and checking permissions is by far the most common operation in most HDFS workloads, see Table 1. In HDFS, the full path is recursively resolved into individual components. In HopsFS for a path of depth N, it would require N roundtrips to the database to retrieve file path components, resulting in high latency for file system operations.

Similar to AFS [36] and Sprite [40], we use hints [30] to speed up the path lookups. Hints are mechanisms to quickly retrieve file path components in parallel (batched operations). In our partitioning scheme, inodes have a composite primary key consisting of the parent inode’s ID and the name of the inode (that is, file or directory name), with the parent inode’s ID acting as the partition key. Each namenode caches only the primary keys of the inodes. Given a pathname and a hit for all path components directories, we can discover the primary keys for all the path components which are used to read the path components in parallel using a single database batch query containing only primary key lookups.

解析路径和检查权限是迄今为止大部分HDFS负载最常见的操作，见表1。在HDFS，完整的路径是递归地分解成各个组成部分。在HopsFS的一个深度为n的路径，它将需要N次往返数据库来检索文件路径，导致文件系统操作的高延迟。

类似于AFS [ 36 ]和Sprite[ 40 ]，我们使用提示加快路径查找。提示机制快速并行检索文件路径组件（批量操作）。在我们的分区方案中，inode节点有一个复合主键由父inode节点的ID和inode节点的名称（即文件或目录的名称）组成，父节点的ID作为分区键。每个namenode节点的高速缓存的inode的主键。给定的路径名和所有路径组成的目录的命中，我们可以使用仅包含主键查找的单个数据库批处理查询，发现用于并行读取路径组件的所有路径组件的主键。

##### Cache Consistency

We use the inode hint cache entries to read the whole inodes in a single batch query at the start of a transaction for a file system operation. If a hint entry is invalid, a primary key read operation fails and path resolution falls back to recursive method for resolving file path components, followed by repairing the cache. Cache entries infrequently become stale as move operations, that update the primary key for an inode, are less than 2% of operations in typical Hadoop workloads, see Table 1. Moreover, typical file access patterns follow a heavy-tailed distribution (in Yahoo 3% of files account for 80% of ac- cesses [1]), and using a sticky policy for HopsFS clients improves temporal locality and cache hit rates.

我们使用inode提示来缓存条目在文件系统操作的事务开始时读取单个批处理查询中的整个inode。 如果提示项无效，则主键读取操作失败，路径解析落后于解析文件路径组件的递归方法，然后修复高速缓存。 缓存条目不需要经常更新，因为一个inode节点的更新逐渐的移动操作在典型的Hadoop工作负载中小于2％，参见表1.另外，典型的文件访问模式遵循重尾分布（在雅虎，3％的文件占访问量的80％[1]），并且为HopsFS客户端使用粘性策略提高了时间局部性和缓存命中率。

## Inode Operations

HopsFS implements a pessimistic concurrency model that supports parallel read and write operations on the namespace, serializing conflicting inode and subtree operations. We chose a pessimistic scheme as, in contrast to opti- mistic concurrency control, it delivers good performance for medium to high levels of resource utilization [4], and many HDFS clusters, such as Spotify’s, run at high load. Inode operations are encapsulated in a single transaction that consists of three distinct phases, which are, **lock**, **execute**, and **update** phases.

HopsFS实现了一种悲观的并发模型，支持对命名空间进行并行读写操作，序列化冲突的inode和subtree操作。 我们选择了一种悲观的方案，与乐观并发控制相反，它为中高资源利用率（）提供了良好的性能[4]，许多HDFS集群（如Spotify）运行在高负载下。 Inode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。

##### Lock Phase

In the lock phase, metadata is locked and read from the database with the strongest lock that will be required for the duration of the transaction. Locks are taken in the *total order*, defined earlier. Inode operations are path-based and if they are not read-only operations, they only modify the last component(s) of the path, for example, *rm* */etc/conf* and *chmod +x /bin/script*. Thus, only the last component(s) of the file paths are locked for file system operations.

Figure 4 shows a transaction template for HopsFS inode operations. Using the inode hint cache the primary keys for the file path components are discovered, line 1. The transaction is started on the database shard that holds all or most of the desired data, line 2. A batched operation reads all the file path components up to the penultimate path component without locking (*read-committed*) the metadata, line 3. For a path of depth *N*, this removes *N-1* round trips to the database. If the inode hints are invalid then the file path is recursively resolved and the inode hint cache is updated, line 4.

在锁定阶段，元数据被锁定并从整个事务期间加了强锁的数据库中读取。锁定按照先前定义的总顺序进行。 Inode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，例如rm /etc/conf和chmod + x /BIN /SCRIPT。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4显示了HopsFS inode操作的事务模板。使用inode提示缓存，发现文件路径组件的主键，第1行。事务在数据库分片上启动，该数据库分片包含所有或大部分所需的数据，第2行。在没有锁定（读取-提交）元数据的情况下，批量操作会将所有文件路径组件读取到倒数第二个路径组件，第3条。对于深度N的路径，这将减少N-1次往返数据库的次数。如果inode提示无效，则递归解析文件路径，并更新inode提示缓存，第4行。

After the path is resolved, either a shared or an exclusive lock is taken on the last inode component in the path, line 5. Shared locks are taken for read-only inode opera- tions, while exclusive locks are taken for inode operations that modify the namespace. Additionally, depending on the operation type and supplied operation parameters, inode related data, such as block, replica, and PRB, are read from the database in a predefined total order using partition pruned scans operations, line 6.

HopsFS uses *hierarchical locking* [19] for inode operations, that is, if data is arranged in tree like hierarchy and all data manipulation operations traverse the hierarchy from top to bottom, then taking a lock on the root of the tree/subtree implicitly locks the children of the tree/subtree. The entity relation diagram for file inode related data, see Figure 3, shows that the entities are arranged in a tree with an inode entity（元组，条目） at the root. That is, taking a lock on an inode implicitly locks the tree of file inode related data. As in all operations, inodes are read first, followed by its related metadata. For some operations, such as creating files/directories and listing operations, the parent directory is also locked to prevent ***phantom* and *fuzzy* reads** for file system operations.

路径解析后，在路径中的最后一个inode组件上采用共享或排他（独占）锁，第5点。共享锁用于只读inode操作，而针对inode操作进行独占锁修改命名空间。另外，根据操作类型和提供的操作参数，使用分区修剪的扫描操作以预定义的总顺序从数据库读取诸如块，副本和PRB之类的inode相关数据，第六点

HopsFS使用分层锁定[19]进行inode节点操作，也就是说，如果数据以树状层次结构排列，并且所有数据处理操作从上到下遍历层次结构，则在树/子树的根目录上放一个锁隐式的锁定树/子树的孩子（我认为这句话意思是：锁定了一个结点，就相当于锁定了其子树的所有结点，因为父节点不释放锁，子树上的数据是根据父节点才能访问到，如果这样就没法访问到）。文件inode相关数据的实体关系图，参见图3，显示了实体被排列在一个以innode实体为根的树中。也就是说，对inode进行锁定就相当于隐式锁定文件inode相关数据的及其子树上的所有数据。在所有操作中，首先读取inode，然后读取其相关元数据。对于一些操作，如创建文件/目录和列表操作，父目录也被锁定，以防止文件系统操作的幻像和模糊读取。

##### Per-Transaction Cache 每个事务缓存

All data that is read from the database is stored in a per-transaction cache (a snapshot) that withholds the propagation of the updated cache records to the database until the end of the transaction. The cache saves many round trips to the database as the metadata is often read and updated multiple times within the same transaction. Row-level locking of the metadata ensures the consistency of the cache, that is, no other transaction can update the metadata. Moreover, when the locks are released upon the completion of the transaction the cache is cleared.

从数据库读取的所有数据都存储在每个事务高速缓存（快照）中，该缓存（快照）将更新的高速缓存记录传送到数据库，直到事务结束为止。 缓存通过保存在同一事务中经常读取和更新多次元数据，从而节省了数据库的多次往返次数。 元数据的行级锁定确保缓存的一致性，也就是说，没有其他事务可以更新元数据。 此外，当事务完成后释放锁时，缓存被清除。

**5.2.3 Execute and Update Phases**

The inode operation is performed by processing the metadata in the per-transaction cache. Updated and new metadata generated during the second phase is stored in the cache which is sent to the database in batches in the final update phase, after which the transaction is either committed or rolled back.

通过处理每个事务高速缓存中的元数据来执行inode操作。 在第二阶段生成的更新的和新的元数据存储在缓存中，该缓存在最终更新阶段分批发送到数据库，之后事务被提交或回滚。

## Handling Large Operations

Recursive operations on large directories, containing millions of inodes, are too large to fit in a single transaction, that is, locking millions of rows in a transaction is not sup-ported in existing online transaction processing systems. These operations include *move*, *delete*, *change owner*, *change permissions*, and *set quota* operations. *Move* op- eration changes the absolute paths of all the descendant inodes, while *delete* removes all the descendant inodes, and the *set quota* operation affects how all the descendant inodes consume disk space or how many files/directories they can create. Similarly changing the permissions or owner of a directory may invalidate operations executing at the lower subtrees.

大型目录的递归操作（包含数百万个索引节点）太大，无法适应单个事务，即在现有的在线事务处理系统中不支持在事务中锁定数百万行。 这些操作包括移动，删除，更改所有者，更改权限和设置配额操作。 move操作更改所有后代inode的绝对路径，而delete则删除所有后代inode，并且设置的配额操作会影响所有后代inode如何消耗磁盘空间或者可以创建多少个文件/目录。 类似地，更改目录的权限或拥有者可能会使在较低子树下执行的操作无效。

## Subtree Operations Protocol

Our solution is a protocol that implements subtree operations incrementally in batches of transactions. Instead of row level database locks, our subtree operations protocol uses an application-level distributed locking mechanism to mark and isolate the subtrees. We serialize subtree operations by ensuring that all ongoing inode and subtree operations in a subtree complete before a newly requested subtree operation is executed. We implement this serialization property by enforcing the following invariants: (1) no new operations access the subtree until the operation completes, (2) the subtree is quiesced before the subtree operation starts, (3) no orphaned inodes or inconsistencies arise if failures occur.

Our subtree operations protocol provides the same consistency semantics as subtree operations in HDFS. For delete subtree operation HopsFS provides even stronger consistency semantics. Failed delete operations in HDFS can result in orphaned blocks that are eventually reclaimed by the block reporting subsystem (hours later). HopsFS improves the semantics of delete operation as failed operations does not cause any metadata inconsistencies, see section 6.2. Subtree operations have the fol- lowing phases.

我们的解决方案是一种协议，可以批量地实现子树操作(我们的解决方案是在批处理事务中逐步实现子树操作的协议。)。我们的子树操作协议不是行级数据库锁，而是使用应用程序级分布式锁定机制来标记和隔离子树。我们通过确保子树中所有正在进行的inode和子树操作在新请求的子树操作执行之前完成，从而序列化子树操作。我们通过执行以下不变量来实现这个序列化属性：（1）操作完成之前没有新的操作访问子树，（2）子树在子操作开始之前停顿，（3）如果发生故障，则不会出现孤立的inode或不一致。

我们的子树操作协议提供与HDFS中的子树操作相同的一致性语义。对于删除子树操作，HopsFS提供了更强的一致性语义。 HDFS中的失败删除操作可能导致孤立快的产生，最终由块报告子系统回收这些孤立块（几小时后）。 HopsFS改进了删除操作的语义，因为失败的操作不会导致任何元数据不一致，请参见第6.2节。子树操作有以下阶段。

**Phase 1:** In the first phase, an exclusive lock is ac- quired on the root of the subtree and a subtree lock flag (which also contains the ID of the namenode that owns the lock) is set and persisted in the database. The flag is an indication that all the descendants of the subtree are locked with exclusive (write) lock.

Before setting the lock it is essential that there are no other active subtree operations at any lower level of the subtree. Setting the *subtree lock* could fail active subtree operations executing on a subset of the subtree. We store all active subtree operations in a table and query it to ensure that no subtree operations are executing at lower levels of the subtree. In a typical workload, this table does not grow too large as subtree operations are usually only a tiny fraction of all file system operations. It is important to note that during path resolution, inode and subtree operations that encounter an inode with a subtree lock turned on voluntarily abort the transaction and wait until the *subtree lock* is removed.

阶段1：在第一阶段，在子树的根处获取排他锁，并且在数据库中设置并保留子树锁定标志（其中还包含拥有锁的namenode的ID）。该标志表示子树的所有后代都被锁定为独占（写入）锁。

在设置锁之前，必须确保在子树的任何较低级别没有其他活动的子树操作。设置子树锁可能会导致子树子集上执行的活动子树操作失败。我们将所有活动的子树操作存储在表中并进行查询，以确保在子树的较低级别没有正在执行的子树操作。在典型的工作负载中，由于子树操作通常只是所有文件系统操作的一小部分，因此该表不会增长太多。重要的是要注意，在路径解析时，遇到具有子树锁的inode的inode和subtree操作，他们会自动中止（暂停）事务并等待直到子树锁被删除。

**Phase 2:** To quiesce the subtree we wait for all ongoing inode operations to complete by taking and releasing database write locks on all inodes in the subtree in the same total order used to lock inodes. To do this efficiently, a pool of threads in parallel execute partition pruned index scans that write-lock child inodes. This is repeated down the subtree to the leaves, and, a tree data structure containing the inodes in the subtree is built in memory at the namenode, see Figure 5. The tree is later used by some subtree operations, such as, move and delete operations, to process the inodes. We reduce the overhead of reading all inodes in the subtree by using projections to only read the inode IDs. If the subtree operations protocol fails to quiesce the subtree due to concurrent file system operations on the subtree, it is retried with exponential backoff.

*Figure 5: Execution of a delete subtree operation. Parallel transactions progress down (shown left) the subtree waiting for active operations to finish by taking and releasing write locks on all the descendant inodes. In the next phase (shown right), the delete operation is executed in batches using parallel transactions upwards from the leaf nodes.*

**Phase 3:** In the last phase the file system operation is broken down into smaller operations that execute in parallel. For improved performance, large batches of inodes are manipulated in each transaction.

阶段2：要停止子树，我们等待所有正在进行的inode操作完成，方法是以与锁定inode相同的总顺序，在子树中的所有inode上采集和释放数据库写入锁。为了有效地执行此操作，并行的线程池执行分区修剪的索引扫描，扫描有写入锁的子innode。这是从树的子树重复到叶子，并且包含子树中的inode的树数据结构被构建在到内存中的namenode中，参见图5.该树稍后被某些子树操作使用，例如移动和删除操作，以处理inode。我们通过使用仅读取inode ID的投影来减少读取子树中所有inode的开销。如果子树操作协议由于子树上的并发文件系统操作而无法停止子树，则会以指数回退重试。

图5：执行删除子树操作。 通过在所有后代的inode上取消和释放写锁，并行事务进行下去（左图）等待主动操作完成的子树。 在下一阶段（右图）中，使用从叶节点向上的并行事务分批执行删除操作。

阶段3：在最后阶段，文件系统操作被分解成并行执行的较小的操作。为了提高性能，在每个事务中都会处理大批索引节点。

**6.2 Handling Failed Subtree Operations**

HopsFS takes lazy approach to cleanup subtree locks left by the failed namenodes [45]. Each namenode maintains a list of the active namenodes provided by the leader election service. If an operation encounters an inode with a subtree lock set and the namenode ID of the subtree lock belongs to a dead namenode then the subtree lock is cleared. However, it is important that when a namenode that is executing a subtree operation fails then it should not leave the subtree in an inconsistent state. The in-memory tree built during the second phase plays an important role in keeping the namespace consistent if the namenode fails. For example, in case of delete operations the subtree is deleted incrementally in post-order tree traversal manner using transactions. If half way through the operation the namenode fails then the inodes that were not deleted remain connected to the namespace tree. HopsFS clients will transparently resubmit the file system operation to another namenode to delete the remainder of the subtree.

Other subtree operations (move, set quota, chmod and chown) do not cause any inconsistencies as the actual operation where the metadata is modified is done in the third phase using a single transaction that only updates the root inodes of the subtrees and the inner inodes are left intact. In the case of a failure, the namenode might fail to unset the subtree lock, however, this is not a problem as other namenodes can easily remove the subtree lock when they find out that the subtree lock belongs to a dead namenode.

HopsFS采取懒惰的方法来清理失败的namenodes留下的子树锁[45]。每个namenode维护由领导选举服务选举出来的的Active NameNodes列表。如果操作遇到具有子树锁的inode，并且子树锁的namenode ID属于死亡的namenode，则子树锁将被清除。但是，重要的是，当执行子树操作的namenode失败时，不应该将子树留在不一致的状态。如果namenode发生故障，在第二阶段构建的内存中的树，在命名空间保持不变时，起着重要的作用。例如，在删除操作的情况下，使用事务以后序树遍历方式逐步删除子树。如果操作进行到一半namenode发生故障，那么未删除的inodes仍然连接到命名空间树。 HopsFS客户端将透明地将文件系统操作重新提交给另一个namenode，以删除子树的其余部分。

其他子树操作（移动，设置配额，chmod和chown）不会导致任何不一致，因为元数据被修改的实际操作是在第三阶段中完成的，只使用仅更新子树的根节点和内部索引节点的单个事务保持原样。在发生故障的情况下，namenode可能无法取消设置的子树锁，但是，这并不是问题，因为其他namenode可以轻松地删除子树锁，当它们发现子树锁属于死亡的namenode。

* 1. Inode and Subtree Lock Compatibility

Similar to the inode operation’s locking mechanism (see section 5.2.1), subtree operations also implement hierarchical locking, that is, setting a subtree flag on a di- rectory implicitly locks the contents of the directory. Both inode and subtree locking mechanisms are compatible with each other, respecting both of their corresponding locks. That is, a subtree flag cannot be set on a direc- tory locked by an inode operation and an inode operation voluntarily aborts the transaction when it encounters a directory with a subtree lock set.

Inode和Subtree Lock兼容性

类似于inode操作的锁定机制（参见5.2.1节），子树操作也实现分层锁定，即在目录上设置子树标志隐含地锁定目录的内容。 inode和子树锁定机制都相互兼容，这两个锁定机制都相互锁定。 也就是说，不能在由inode操作锁定的目录中设置子树标志，并且当遇到具有子树锁定的目录时，inode操作会自动中止事务。

## Metadata (Namespace) Scalability

In HDFS, as the entire namespace metadata must fit on the heap of single JVM, the data structures are highly optimized to reduce the memory footprint [60]. In HDFS, a file with two blocks that are replicated three ways re- quires 448 + L bytes of metadata1 where L represents the filename length. If the file names are 10 characters long, then a 1 GB JVM heap can store 2.3 million files. In reality the JVM heap size has to be significantly larger to accommodate secondary metadata, thousands of con- current RPC requests, block reports that can each be tens of megabytes in size, as well as other temporary objects.

在HDFS中，由于整个命名空间元数据必须适合单个JVM的堆，所以数据结构被高度优化以减少内存占用[60]。 在HDFS中，具有三个复制方式的两个块的文件需要448 + L个字节的元数据1，其中L表示文件名长度。 如果文件名长度为10个字符，则1 GB JVM堆可以存储230万个文件。 实际上，JVM堆大小必须要大得多以适应次要元数据，数千个RIO请求，阻塞报告，每个都可能是数十兆字节的大小以及其他临时对象。

Migrating the metadata to a database causes an expan- sion in the amount of memory required to accommodate indexes, primary/foreign keys and padding. In HopsFS the same file described above takes 1552 bytes if the metadata is replicated twice. For a highly available deployment with an active and standby namenodes for HDFS, you will need twice the amount of memory, thus, HopsFS requires≈ 1.5 times more memory than HDFS to store metadata that is highly available. Table 3 shows the metadata scalability of HDFS and HopsFS.

将元数据迁移到数据库会导致扩展容纳索引，主键/外键和填充所需的内存量。 在HopsFS中，如果元数据被复制两次，则上述相同的文件将占用1552字节。 对于具有HDFS主动和备用节点的高可用性部署，您将需要两倍的内存量，因此，HopsFS需要比HDFS约1.5倍的内存来存储高可用性的元数据。 表3显示了HDFS和HopsFS的元数据可扩展性。

**9 External Metadata Implications**

Administrators often resort to writing their own tools to analyze the HDFS namespace. HopsFS enables online ad hoc analytics on the metadata. With a NDB backend, HopsFS metadata can be selectively and asynchronously replicated to either a backup cluster or a MySQL slave server, enabling complex analytics without affecting the performance of the active cluster. HopsFS metadata is also easy to export to external systems and it is easy to safely extend the metadata. That is, additional tables can be created that contain a foreign key to the associated inode, thus ensuring the integrity of the extended metadata. Using this approach, we have already added new features to HopsFS, including extended attributes for inodes and erasure coding. Moreover, following similar ideas to [28], we developed an eventually consistent replication protocol that replicates (extended) HopsFS metadata to Elasticsearch [15] for free-text search. This enables us to search the entire namespace with sub-second latency. We believe that distributed metadata in a commodity database is a significant new enabling technology and it can become a reliable source of ground truth for metadata applications built on top of distributed file systems.

管理员经常诉诸编写自己的工具来分析HDFS命名空间。 HopsFS可以对元数据进行在线特别分析。使用NDB后端，HopsFS元数据可以有选择地和异步地复制到备份集群或MySQL从属服务器，实现复杂的分析而不影响主集群的性能。 HopsFS元数据也易于导出到外部系统，并且易于安全地扩展元数据。也就是说，可以创建包含相关联的inode的外键的附加表，从而确保扩展元数据的完整性。使用这种方法，我们已经向HopsFS添加了新功能，包括inode和erasure编码的扩展属性。此外，按照与[28]类似的想法，我们开发了一个最终一致的复制协议，将HopsFS元数据复制到Elasticsearch [15]进行自由文本搜索。这使我们能够以次秒延迟搜索整个命名空间。我们认为商品数据库中的分布式元数据是一种重要的新型支持技术，它可以成为建立在分布式文件系统之上的元数据应用程序的可靠依据。

## 10. Summary

In this paper, we introduced HopsFS, that is, to the best of our knowledge, the first production-grade distributed hierarchical file system that stores its metadata in an external NewSQL database. HopsFS is an open-source, highly available file system that scales out in both capacity and throughput by adding new namenodes and database nodes. HopsFS can store 37 times more metadata than HDFS and for a workload from Spotify, HopsFS scales to handle 16 times the throughput of HDFS. HopsFS also has lower average latency for large number of concurrent clients, and no downtime during failover. Our architecture supports a pluggable database storage engine, and other NewSQL databases could be used. Finally, HopsFS makes metadata tinker friendly, opening it up for users and applications to extend and analyze in new and creative ways.

在本文中，我们引入了HopsFS，就我们所知，是第一个生产级分布式分层文件系统将其元数据存储在外部NewSQL数据库中。 HopsFS是一个开放源码，高可用性的文件系统，通过添加新的节点和数据库节点，扩展了容量和吞吐量。 HopsFS可以存储比HDFS多37倍的元数据，对于来自Spotify的工作负载，HopsFS可以扩展到处理HDFS吞吐量的16倍。 HopsFS对大量并发客户端的平均延迟更低，在故障转移期间也不会有停机时间。 我们的架构支持可插拔数据库存储引擎，可以使用其他NewSQL数据库。 最后，HopsFS使元数据修补程序变得友好，为用户和应用程序打开新的和创造性的方式来扩展和分析。