H

HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases

Salman Niazi, Mahmoud Ismail, Seif Haridi, and Jim Dowling
KTH – Royal Institute of Technology, Stockholm, Sweden

Definition

Modern NewSQL database systems can be used to store fully normalized metadata for distributed hierarchical file systems, and provide high throughput and low operational latencies for the file system operations.

Introduction

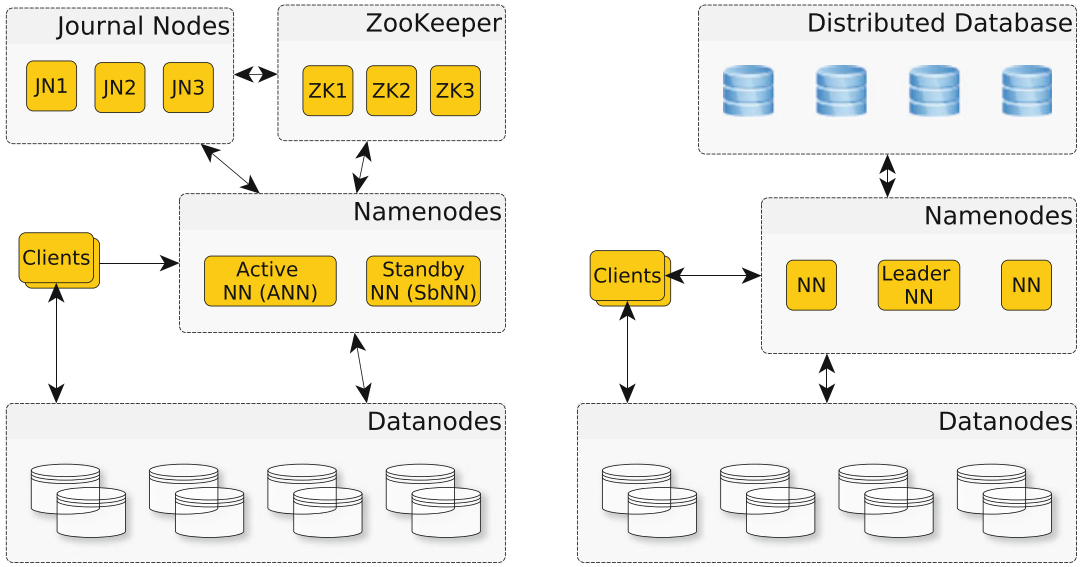
For many years, researchers have investigated the use of database technology to manage file system metadata, with the goal of providing extensible typed metadata and support for fast, rich metadata search. However, previous attempts failed mainly due to the reduced performance introduced by adding database operations to the file system's critical path. However, recent improvements in the performance of distributed in-memory online transaction processing databases (NewSQL databases) led us to reinvestigate the possibility of using a database to manage file system metadata, but this time for a distributed, hierarchical file system, the Hadoop file system (HDFS). The single-host metadata service of HDFS is a well-known bottleneck for both the size of HDFS clusters and their throughput. In this entry, we characterize the performance of different NewSQL database operations and design the metadata architecture of a new drop-in replacement for HDFS using, as far as possible, only those high-performance NewSQL database access patterns. Our approach enabled us to scale the throughput of all HDFS operations by an order of magnitude.

HopsFS vs. HDFS

In both systems, namenodes provide an API for metadata operations to the file system clients, see Fig. 1. In HDFS, an active namenode (ANN) handles client requests, manages the metadata in memory (on the heap of a single JVM), logs changes to the metadata to a quorum of journal servers, and coordinates repair actions in the event of failures or data corruption. A standly namenode asynchronously pull the metadata changes from the journal nodes and applies the changes to its in memeory metadata. A ZooKeeper service is used to signal namenode failures, enabling both agreement on which namenode is active as well as failover from active to standby.

In HopsFS, namenodes are stateless servers that handle client requests and process the metadata that is stored in an external distributed database, NDB in our case. The internal

S. Sakr, A. Zomaya (eds.), *Encyclopedia of Big Data Technologies*,
https://doi.org/10.1007/978-3-319-63962-8_146-1



HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases, Fig. 1 In HDFS, a single namenode manages the namespace metadata. For high availability, a log of metadata changes is stored on a set of journal nodes using quorum-based replication. The log is subsequently replicated asynchronously to a standby namenode. In contrast, HopsFS supports multiple stateless namenodes and a single leader namenode that all access metadata stored in transactional shared memory (NDB). In HopsFS, leader election is coordinated using NDB, while ZooKeeper coordinates leader election for HDFS

management (housekeeping) operations must be coordinated among the namenodes. HopsFS solves this problem by electing a leader namenode that is responsible for the housekeeping. We use the database as a shared memory to implement a leader election and membership management service (Salman Niazi et al. 2015; Guerraoui and Raynal 2006).

In both HDFS and HopsFS, datanodes are connected to all the namenodes. Datanodes periodically send a heartbeat message to all the namenodes to notify them that they are still alive. The heartbeat also contains information such as the datanode capacity, its available space, and its number of active connections. Heartbeats update the list of datanode descriptors stored at namenodes. The datanode descriptor list is used by namenodes for future block allocations, and it is not persisted in either HDFS or HopsFS, as it is rebuilt on system restart using heartbeats from datanodes.

HopsFS clients support random, round-robin, and sticky policies to distribute the file system operations among the namenodes. If a file system operation fails, due to namenode failure or overloading, the HopsFS client transparently retries the operation on another namenode (after backing off, if necessary). HopsFS clients refresh the namenode list every few seconds, 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.

MySQL's NDB Distributed Relational Database

HopsFS uses MySQL's Network Database (NDB) to store the file system metadata. NDB is an open-source, real-time, in-memory, shared nothing, distributed relational database management system.

NDB cluster consists of three types of nodes: NDB datanodes that store the tables, management nodes, and database clients; see Fig. 3.

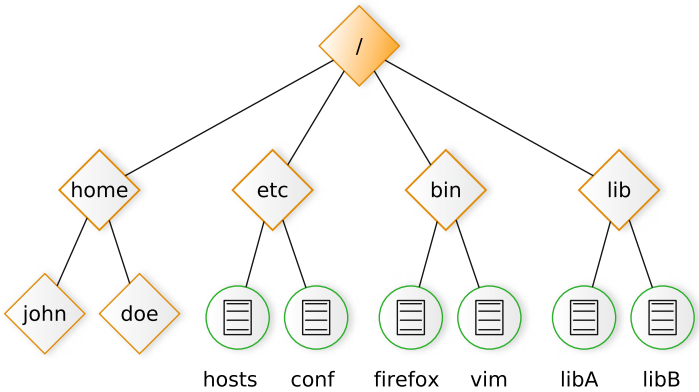
The management nodes are only used to disseminate the configuration information and to detect network partitions. The client nodes access the database using the SQL interface via MySQL Server or using the native APIs implemented in C++, Java, JPA, and JavaScript/Node.js.

Figure 3 shows an NDB cluster setup consisting of four NDB datanodes. Each NDB datanode consists of multiple transaction coordinator (TC) threads, which are responsible for performing two-phase commit transactions; local data management threads (LDM), which are responsible for storing and replicating the data partitions assigned to the NDB datanode; send and receive threads, which are used to exchange the data between the NDB datanodes and the clients; and IO threads which are responsible for performing disk IO operations.

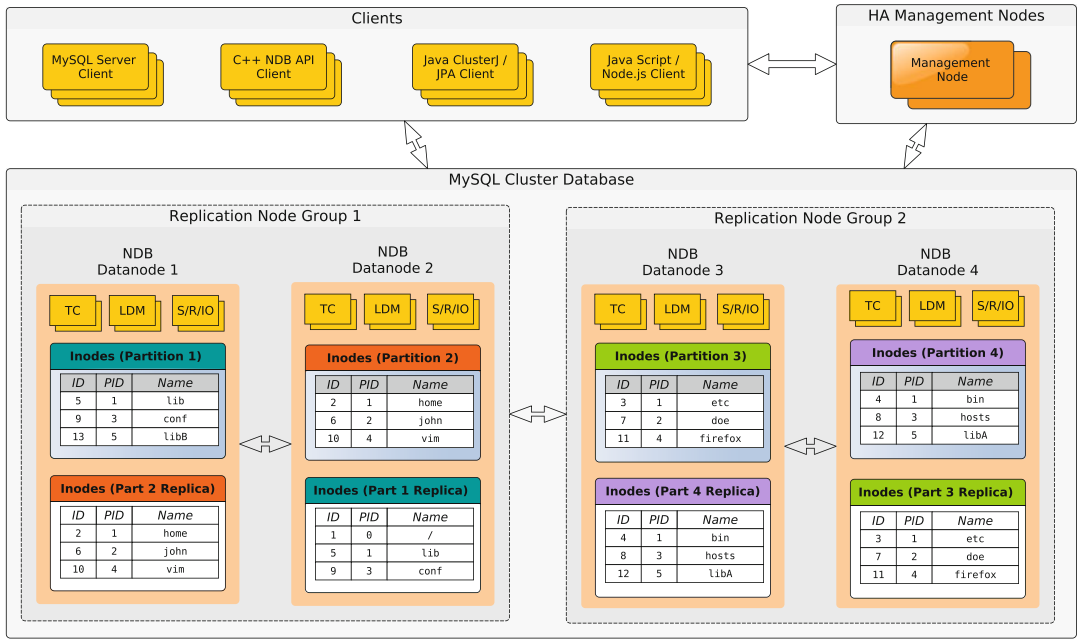
MySQL Cluster horizontally partitions the tables, that is, the rows of the tables are uniformly distributed among the database partitions stored on the NDB datanodes. The NDB datanodes are organized into node replication groups of equal sizes to manage and replicate the data partitions. The size of the node replication group is the replication degree of the database. In the example setup, the NDB replication degree is set to two (default value); therefore, each node replication group contains exactly two NDB datanodes. The first replication node group consists of NDB datanodes 1 and 2, and the second replication node group consists of NDB datanodes 3 and 4. Each node group is responsible for storing and replicating all the data partitions assigned to the NDB datanodes in the replication node group.

By default, NDB hashes the primary key column of the tables to distribute the table's rows among the different database partitions. Figure 3 shows how the inodes table for the namespace shown in Fig. 2 is partitioned and stored in the NDB cluster database. In production deployments each NDB datanode may store multiple data partitions, but, for simplicity, the datanodes shown in Fig. 3 store only one data partition. The replication node group 1 is responsible for storing and replicating partitions 1 and 2 of the inodes' table. The primary replicas of partition 1 is stored on the NDB datanode 1, and the replica of the partition 1 is stored on the other datanode of the same replication node group, that is, NDB datanode 2. For example, the NDB datanode 1 is responsible for storing a data partition that stores the *lib*, *conf*, and *libB* inodes, and the replica of this data partition is stored on NDB datanode 2. NDB also supports user-defined partitioning of the stored tables, that is, it can partition the data based on a user-specified table column. User-defined partitioning provides greater control over how the data is distributed among different database partitions, which helps in implementing very efficient database operations.

NDB only supports $read-committed$ transaction isolation level which is not sufficient for implementing practical applications. However, NDB supports row-level locking which can be used to isolate transactions operating on the same



HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases, Fig. 2 A sample file system namespace. The diamond shapes represent directories and the circles represent files



HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases, Fig. 3 System architecture diagram of MySQL's Network Database (NDB) cluster. The NDB cluster consists of three types of nodes, that is, database clients, NDB management nodes, and NDB database datanodes

datasets. NDB supports $exclusive$ (write) locks, $shared$ (read) locks, and $read-committed$ (read the last committed value) locks. Using these locking primitives, HopsFS isolates different file system operations trying to mutate the same subset of inodes.

Types of Database Operations

A transactional file system operation consists of three distinct phases. In the first phase, all the metadata that is required for the file system operation is read from the database; in the second phase, the operation is performed on the metadata; and in the last phase, all the modified metadata (if any) is stored back in the database. As the latency of the file system operation greatly depends on the time spent in reading and writing the data, therefore, it is imperative to understand the latency and computational cost of different types of database operations, used to read and update the stored data, in order to understand how HopsFS implements low-latency scalable file system operations. The data can be read using four different types of database operations, such as *primary key*, *partition-pruned index scan*, *distributed index scan*, and *distributed full table scan* operations. NDB also supports *user-defined data partitioning* and *data distribution-aware transactions* to reduce the latency of the distributed transactions. A brief description of these operations is as follows:

Application-Defined Partitioning (ADP)

NDB allows developers to override the default partitioning scheme for the tables, enabling a fine-grained control on how the tables' data is distributed across the data partitions. HopsFS leverage this feature, for example, the inodes table is partitioned by the parent inode ID, that is, all immediate children of a directory reside on the same database partition, enabling an efficient directory listing operation.

Distribution-Aware Transactions (DAT)

NDB supports *distribution-aware transactions* by specifying a *transaction hint* at the start of the transaction. This enlists the transaction with a transaction coordinator on the database node that

holds the data required for the transaction, which reduces the latency of the database operation. The choice of the transaction hint is based on the *application-defined partitioning* scheme. Incorrect hints do not affect the correctness of the operation, since the transaction coordinator (TC) will route the requests to different NDB datanode that holds the data; however, it will incur additional network traffic.

Primary Key (PK) Operation

PK operation is the most efficient operation in NDB that reads/writes/updates a single row stored in a database shard.

Batched Primary Key Operations

Batch operations use batching of PK operations to enable higher throughput while efficiently using the network bandwidth.

Partition-Pruned Index Scan (PPIS)

$PPIS$ is a distribution-aware operation that exploits the *distribution-aware transactions* and *application-defined partitioning* features of NDB, to implement a scalable index scan operation such that the scan operation is performed on a single database partition.

Distributed Index Scan (DIS)

DIS is an index scan operation that executes on all database shards. It is not a distribution-aware operation, causing it to become increasingly slow for increasingly larger clusters.

Distributed Full Table Scan (DFTS)

$DFTS$ operations are not distribution aware and do not use any index; thus, they read all the rows for a table stored in all database shards. DFTS operations incur high cpu and network costs; therefore, these operations should be avoided in implementing any database application.

Comparing Different Database Operations

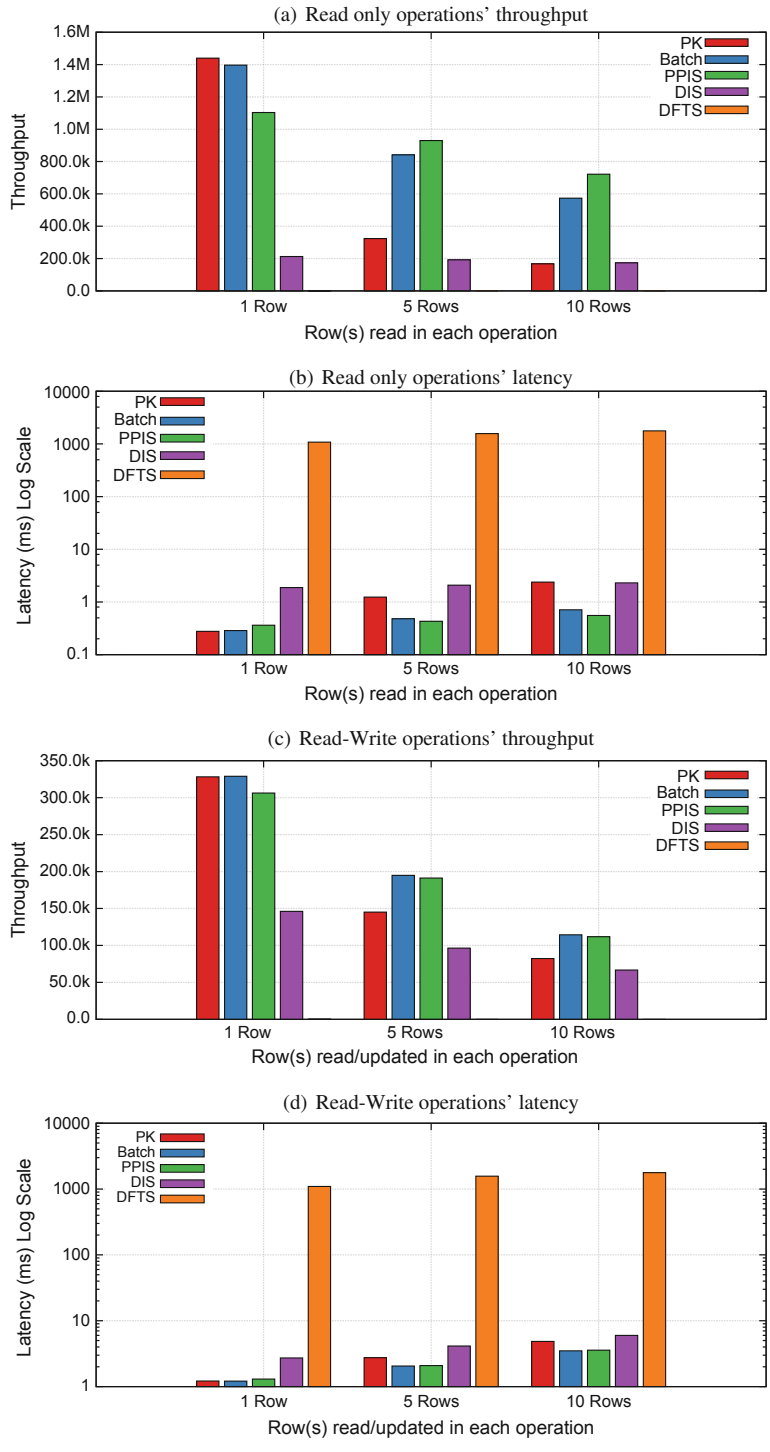
Figure 4 shows the comparison of the throughput and latency of different database operations. These micro benchmarks were performed on a four-node NDB cluster setup running NDB version 7.5.6. All the experiments were run on premise using Dell PowerEdge R730xd servers (Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40 GHz, 256 GB RAM, 4 TB 7200 RPM HDD) connected using a single 10 GbE network adapter. We ran 400 concurrent database clients, which were uniformly distributed across 20 machines. The experiments are repeated twice. The first set of experiments consist of read-only operations, and the second set of experiments consist of read-write operations where all the data that is read is updated and stored back in the database.

Distributed full table scan operations and distributed index scan operations do not scale, as these operations have the lowest throughput and highest end-to-end latency among all the database operations; therefore, there operations must be avoided in implementing file system operations. Primary key, batched primary key operations, and partition-pruned index scan operations are scalable database operations that can deliver very high throughput and have low end-to-end operational latency.

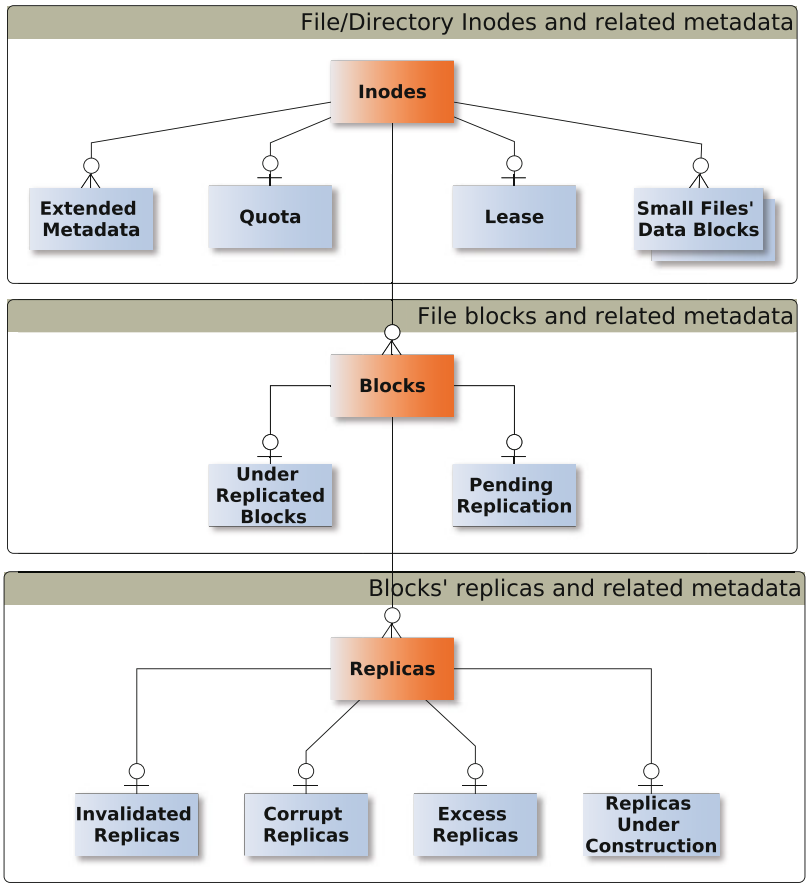
The design of the database schema, that is, how the data is laid out in different tables, the design of the primary keys of the tables, types/number of indexes for different columns of the tables, and data partitioning scheme for the tables, plays a significant role in choosing an appropriate (efficient) database operation to read-/update the data, which is discussed in detail in the following sections.

HopsFS Distributed Metadata

In HopsFS the metadata is stored in the database in normalized form, that is, instead of storing the full file paths with each inode as in Thomson and Abadi (2015), HopsFS stores individual file path components. Storing the normalized data in the database has many advantages, such as $rename$ file system operation can be efficiently implemented by updating a single row in the database that represents the inode. If complete file paths are stored for each inode, then it would not only consume significant amount of precious database storage, but also, a simple directory



HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases, Fig. 4 The comparison of the throughput and latency of different database operations. (**a**) Read-only operations' throughput. (**b**) Read-only operations' latency. (**c**) Read-write operations' throughput. (**d**) Read-write operations' latency



HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases, Fig. 5 Entity relational diagram for HopsFS schema

rename operation would require updating the file paths for all the children of that directory subtree.

Key tables in HopsFS metadata schema are shown in the entity relational (ER) diagram in Fig. 5. The inodes for files and directories are stored in the *inodes* table. Each inode is represented by a single row in the table, which also stores information such as inode ID, name, parent inode ID, permission attributes, user ID, and size. As HopsFS does not store complete file paths with each inode, the inode's parent ID is used to traverse the inodes table to discover the complete path of the inode. The extended metadata for the inodes is stored in the *extended metadata* entity. The *quota* table stores information about disk space consumed by the directories with quota restrictions enabled. When a client wants to update a file, then it obtains a lease on the file, and the file lease information is stored in the *lease* table. Small files that are few kilobytes in size are stored in the database for low-latency access. Such files are stored in the *small files' data blocks* tables.

Non-empty files may contain multiple blocks stored in the *blocks* table. The location of each replica of the block is stored in the *replicas* table. 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. HopsFS periodically checks the health of all the data blocks. HopsFS ensures that the number of replicas of each block does not fall below the threshold required to provide high

availability of the data. The replication monitor sends commands to datanodes to create more replicas of the under-replicated blocks. The blocks undergoing replication are stored in the *pending replication blocks* table. Similarly, a replica of a block has various states during its life cycle. When a replica gets corrupted due to disk errors, it is moved to the *corrupted replicas* table. The replication monitor will create additional copies of the corrupt replicas by copying healthy replicas of the corresponding blocks, and the corrupt replicas are moved to *invalidated replicas* table for deletion. Similarly, when a file is deleted, the replicas of the blocks belonging to the file are moved to the *invalidated replicas* table. Periodically the namenodes read the invalided replicas tables and send commands to the datanodes to permanently remove the replicas from the datanodes' disk. Whenever a client writes to a new block's replica, this replica is moved to the *replica under construction* table. If too many replicas of a block exist (e.g., due to recovery of a datanode that contains blocks that were re-replicated), the extra copies are stored in the *excess replicas* table.

Metadata Partitioning

The choice of partitioning scheme for the hierarchical namespace is a key design decision for distributed metadata architectures. HopsFS uses user-defined partitioning for all the tables stored in NDB. 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. Read-only metadata operations, such as *list*, *stat*, and *file read* operations, alone account for ≈95% of the operations in Spotify's HDFS cluster (Niazi et al. 2017). Based on the micro benchmarks of different database operations, we have designed the metadata partitioning scheme such that frequent file system operations are implemented using the scalable database operations such as primary key, batched primary key operations, and partition-pruned index scan operations. Index scans and full table scans are avoided, where possible, as they touch all database partitions and scale poorly.

HopsFS partitions *inodes* by their *parents' inode IDs*, resulting in inodes with the same parent inode being stored on the same database partitions. That enables 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 partitions 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 and replicas and their states, is partitioned using the file's *inode ID*. This results in metadata for a given file all being stored in a single database partition, again enabling efficient file operations. Note that the file inode-related entities also contain the inode's foreign key (not shown in the entity relational diagram Fig. 5) that is also the partition key, enabling HopsFS to read the file inode-related metadata using partition-pruned index scans.

It is important that the partitioning scheme distribute the metadata evenly across all the database partitions. Uneven distribution of the metadata across all the database partitions leads to poor performance of the database. It is not uncommon for big data applications to create tens of millions of files in a single directory (Ren et al. 2013; Patil et al. 2007). HopsFS metadata partitioning scheme does not lead to hot spots as the database partition that contains the immediate children (files and directories) of a parent directory only stores the names and access attributes, while the inode-related metadata (such as blocks, replicas, and associated states) that comprises the majority of the metadata is spread across all the database partitions because it is partitioned by the inode IDs of the files. For hot files/directories, that is, file/directories that are frequently accessed by many of concurrent clients, the performance of the file system will be limited by the performance of the database partitions that hold the required data for the hot files/directories.

HopsFS Transactional Operations

File system operations in HopsFS are divided into two main categories, that is, non-recursive file system operations that operate on a single file or directory, also called **inode operations**, and recursive file system operations that operate on directory subtrees which may contain large number of descendants, also called **subtree operations**. The file system operations are divided into these two categories because databases, for practical reasons, put a limit on the number of read and write operations that can be performed in a single database transaction, that is, a large file system operation may take a long time to complete and the database may timeout the transaction before the operation successfully completes. For example, creating a file, mkdir, or deleting file are *inode operations* that read and update limited amount of metadata in each transactional file system operation. However, recursive file system operations such as delete, move, chmod on large directories may read and update millions of rows. As subtree operations cannot be performed in a single database operation, HopsFS breaks the large file system operation into small transactions. Using application-controlled locking for the subtree operations, HopsFS ensures that the semantics and consistency of the file system operation remain the same as in HDFS.

Inode Hint Cache

In HopsFS all file system operations are path based, that is, all file system operations operate on file/directory paths. When a namenode receives a file system operation, it traverses the file path to ensure that the file path is valid and the client is authorized to perform the given file system operation. In order to recursively resolve the file path, HopsFS uses primary key operations to read each constituent inode one after the other. HopsFS partitions the inodes' table using the parent ID column, that is, the constituent inodes of the file path may reside on different database partitions. In production deployments, such as in case of Spotify, the average file system path length is 7, that is, on average it would take 7 round trips to the database to resolve the file path, which would increase the end-to-end latency of the file system operation. However, if the primary keys of all the constituent inodes for the path are known to the namenode in advance, then all the inodes can be read from the database in a single-batched primary key operation. For reading multiple rows batched primary key operations scale better than iterative primary key operations and have lower end-to-end latency. Recent studies have shown that in production deployments, such as in Yahoo, the file system access patterns follow a heavy-tailed distribution, that is, only 3% of the files account for 80% of file system operations (Abad 2014); therefore, it is feasible for the namenodes to cache the primary keys of the recently accessed inodes. The namenodes store the primary keys of the inodes table in a local least recently used (LRU) cache called the inodes hints cache. We use the inode hint cache to resolve the frequently used file paths using single-batch query. The inodes cache does not require cache coherency mechanisms to keep the inode cache consistent across the namenodes because if the inodes hints cache contains invalid primary keys, then the batch operation would fail and then the namenode will fall back to recursively resolving the file path using primary key operations. After recursively reading the path components from the database, the namenodes also update the inode cache. Additionally, only the move operation changes the primary key of an inode. Move operations are very tiny fraction of the production file system workloads, such as at Spotify less than 2% of the file system operations are file move operations.

Inode Operations

Inode operations go through the following phases.

Pre-transaction Phase

- The namenode receives a file system operation from the client.
- The namenode checks the local inode cache, and if the file path components are found in the cache, then it creates a batch operation to read all file components.

Transactional File System Operation

- The inodes cache is also used to set the partition key hint for the transactional file system operation. The inode ID of the last valid path component is used as partition key hint.
- The transaction is enlisted on the desired NDB datanode according to the partition hint. If the namenode is unable to determine the partition key hint from the inode cache, then the transaction is enlisted on a random NDB datanode.
- The batched primary key operation is sent to the database to read all the file path inodes. If the batched primary key operation fail due to invalid primary key(s), then the file path is recursively resolved. After resolving the file path components, the namenode also updates the inode cache for future operations.
- The last inode in the file path is read using appropriate lock, that is, read-only file system operation takes shared lock on the inode, while file system operations that need to update the inode take exclusive lock on the inode.
- Depending on the file system operation, some or all of the secondary metadata for the inode is read using partition-pruned index scan operations. The secondary metadata for an inode includes quota, extended metadata, leases, small files data blocks, blocks, replicas, etc., as shown in the entity relational diagram of the file system; see Fig. 5.
- After all the data has been read from the database, the metadata is stored in a per-transaction cache. The file system operation is performed, which reads and updates the data stored in the per-transaction cache. All the changes to the metadata are also stored in the per-transaction cache.
- After the file system operations successfully finish, all the metadata updates are sent to the database to persist the changes for later operations.
- If there are no errors, then the transaction is committed; otherwise the transaction is rolled back.

Post Transaction

- After the transactional file system operation finishes, the results are sent back to the client.
- Additionally, the namenode also updates it matrices about different statistics of the file system operations, such as frequency of different types of file system operations and duration of the operations.

Subtree Operations

Recursive file system operations that operate on large directories are too large to fit in a single database transaction, for example, it is not possible to delete hundreds of millions of files in a single database transaction. HopsFS uses subtree operations protocol to perform such large file system operations. The subtree operations protocol uses application-level locking to mark and isolate inodes. HopsFS subtree operations are designed in such a way that the consistency and operational semantics of the file system operations remain compatible with HDFS. The subtree operations execute in three phases as described below.

Phase 1 – Locking the Subtree: In the first phase, the directory is isolated using application-level locks. Using an inode operation, an exclusive lock is obtained for the directory, and the subtree lock flag is set for the inode. The subtree lock flag also contains other information, such as the ID of the namenode that holds the lock, the type of the file system operation, and the full path of the directory. The flag is an indication that all the descendants of the subtree are locked with exclusive (write) lock. 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.

Phase 2 – Quiescing the Subtree: Before the subtree operation updates the descendants of the subtree, it is imperative to ensure that none of the descendants are being locked by any other concurrent subtree or inode operation. Setting the subtree lock before checking if any other descendant directory is also locked for a subtree operation can result in multiple active subtree operations on the same inodes, which will

compromise the consistency of the namespace. 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. Additionally, 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 inode. 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. The tree is later used by some subtree operations, such as *move* and *delete* operations, to process the inodes. 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.

Phase 3 – Executing the FS Operation: Finally, after locking the subtree and ensuring that no other file system operation is operating on the same subtree, it is safe to perform the requested file system operation. In the last phase, the subtree operation is broken into smaller operations, which are run in parallel to reduce the latency of the subtree operation. For example, when deleting a large directory, starting from the leaves of the directory subtree, the contents of the directory is deleted in batches of multiple files.

Handling Failed Subtree Operation

As the locks for subtree operations are controlled by the application (i.e., the namenodes), therefore, it is important that the subtree locks are cleared when a namenode holding the subtree lock fails. HopsFS uses lazy approach for cleaning the subtree locks left by the failed namenodes. The subtree lock flag also contains information about the namenode that holds the subtree lock. Using the group membership service, the namenode maintains a list of all the active namenodes in the system. During the execution of the file system operations, when a namenode encounters an inode with a subtree lock flag set that does not belong to any live namenode, then the subtree operation flag is reset.

It is imperative that the failed subtree operations do not leave the file system metadata in an inconsistent state. For example, the deletion of the subtree progresses from the leaves to the root of the subtree. If the recursive delete operation fails, then the inodes of the partially deleted subtree remain connected to the namespace. When a subtree operation fails due to namenode failure, then the client resubmits the operation to another alive namenode to complete the file system operation.

H

Storing Small Files in the Database

HopsFS supports two file storage layers, in contrast to the single file storage service in HDFS, that is, HopsFS can store data blocks on HopsFS datanodes as well as in the distributed database. Large files blocks are stored on the HopsFS datanodes, while small files (<= 64 KB) are stored in the distributed database. The technique of storing the data blocks with the metadata is also called inode stuffing. In HopsFS, an average file requires 1.5 KB of metadata. As a rule of thumb, if the size of a file is less than the size of the metadata (in our case 1 KB or less), then the data block is stored in memory with the metadata. Other small files are stored in on-disk data tables. The latest solid-state drives (SSDs) are recommended for storing small files data blocks as typical workloads produce large number of random reads/writes on disk for small amounts of data.

Inode stuffing has two main advantages. First, it simplifies the file system operations protocol for reading/writing small files, that is, many network round trips between the client and datanodes are avoided, significantly reducing the expected latency for operations on small files. Second, it reduces the number of blocks that are stored on the datanode and reduces the block reporting traffic on the namenode. For example, when a client sends a request to the namenode to read a file, the namenode retrieves the file's metadata from the database. In case of a small file, the namenode also fetches the data block from the database. The namenode then returns

the file's metadata along with the data block to the client. Compared to HDFS, this removes the additional step of establishing a validated, secure communication channel with the datanodes (Kerberos, TLS/SSL sockets, and a block token are all required for secure client-datanode communication), resulting in lower latencies for file read operations (Salman Niazi et al. 2017).

Extended Metadata

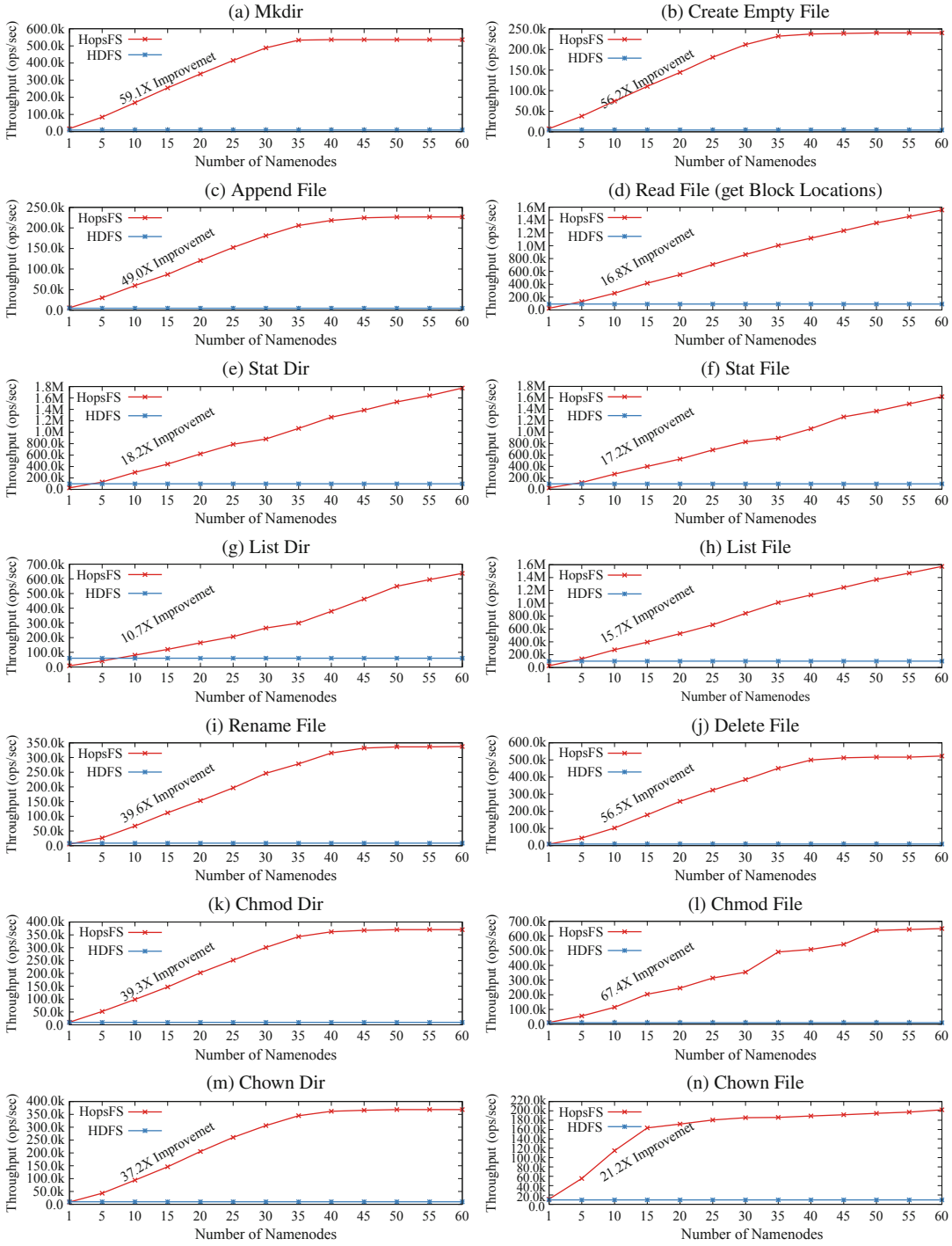
File systems provide basic attributes for files/directories such as name, size, modification time, etc. However, in many cases, users and administrators require the ability to tag files/directories with extended attributes for later use. Moreover, rich metadata has become an invaluable feature especially with the continuous growth of data volumes stored in scale-out file systems, such as HopsFS. As discussed earlier HopsFS store all the file system metadata in a database, where each file/directory (inode) is identified by a primary key. Therefore, creating an extended metadata table with a foreign key to the inodes table should be sufficient to ensure consistency and integrity of the metadata and the extended metadata.

HopsFS offers two modes to store extended metadata, that is, *schema-based* and *schemaless* extended metadata. In the schema-based approach, users have to create a predefined schema for their extended metadata, similar to information schema in database systems, and then associate their data according to that schema. The predefined schema enables validation of the extended metadata before attaching them to the inodes. On the other hand, in the schemaless approach, users don't have to create a predefined schema beforehand; instead, they can attach any arbitrary extended metadata that is stored in a self-contained manner such as a JSON file.

Using either approach schema-based or schemaless, users and administrators can tag their files/directories with extended metadata and then later query for the files/directories based on the tags provided. For example, users can tag files with a description field and later search for the files that match (or partially match) a given free-text query. Also, administrators could use the basic attributes, for example, file size to list all the big files in the cluster, or modification time to list all files that have not been modified within the last month. Depending on the queries pattern, it might be efficient for some queries to run directly on the metadata database, and for others especially the free-text queries, to run on a specialized free-text search engines. Also, the conventional wisdom in database community has been that there is no-size-fits-all solution. This encourages the use of different storage engines according to the query pattern needed that is known as a polyglot persistence of the data. Therefore, the metadata of the namespace should be replicated into different engines to satisfy different query requirements such as free-text, point, range, and sophisticated join queries. We have developed a databus system, called *ePipe* (Ismail et al. 2017), that provides a strongly consistent replicated metadata service as an extension to HopsFS and that in real-time delivers file system metadata and extended metadata to different storage engines. Users and administrators can later query the different storage engines according to their query requirements.

Results

As HopsFS addresses how to scale out the metadata layer of HDFS, all our experiments are designed to comparatively test the performance and scalability of the namenode(s) in HDFS and HopsFS. Most of the available benchmarking utilities for Hadoop are MapReduce-based, primarily designed to determine the performance of a Hadoop cluster. These benchmarks are very sensitive to the MapReduce subsystem and therefore are not suitable for testing the performance and scalability of namenodes in isolation (Noll 2015). Inspired by the NNThroughputBenchmark for HDFS and the benchmark utility designed to test QFS (Ovsiannikov et al. 2013), we developed a distributed benchmark that spawns tens of thousands of file system clients, distributed across many machines, which concurrently execute file system (metadata) operations on the



HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases, Fig. 6 Throughput of different file system operations in HopsFS and HDFS. As HDFS only supports only one active namenode, the throughput for HDFS file system operations are represented by the horizontal (blue) lines. (**a**) Mkdir. (**b**) Create empty file. (**c**) Append file. (**d**) Read file (getBlockLocations). (**e**) Stat dir. (**f**) Stat file. (**g**) List dir. (**h**) List file. (**i**) Rename file. (**j**) Delete file. (**k**) Chmod dir. (**l**) Chmod file. (**m**) Chown dir. (**n**) Chown file

namenode(s). The benchmark utility can test the performance of both individual file system operations and file system workloads based on industrial workload traces. The benchmark utility is open source (Hammer-Bench 2016) and the experiments described here can be fully reproduced on AWS using Karamel and the cluster definitions found in Hammer-Bench (2016).

We ran all the experiments on premise using Dell PowerEdge R730xd servers (Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40 GHz, 256 GB RAM, 4 TB 7200 RPM HDDs) connected using a single 10 GbE network adapter. Unless stated otherwise, NDB, version 7.5.3, was deployed on 12 nodes configured to run using 22 threads each, and the data replication degree was 2. All other configuration parameters were set to default values. Moreover, in our experiments Apache HDFS, version 2.7.2, was deployed on 5 servers. We did not colocate the file system clients with the namenodes or with the database nodes. As we are only evaluating metadata performance, all the tests created files of zero length (similar to the NNThroughputBenchmark Shvachko 2010). Testing with non-empty files requires an order of magnitude more HDFS/HopsFS datanodes, and provides no further insight.

Figure 6 shows the throughput of different file system operations as a function of number of namenodes in the system. HDFS supports only one active namenode; therefore, the throughput of the file system is represented by (blue) horizontal line in the graphs. From the graphs it is quite clear that HopsFS outperforms HDFS for all file system operations and has significantly better performance than HDFS for the most common file system operations. For example, HopsFS improves the throughput of create file and read file operations by $56.2X$ and $16.8X$, respectively.

Additional experiments for industrial workloads, write intensive synthetic workloads, metadata scalability, and performance under failures are available in HopsFS papers (Niazi et al. 2017; Ismail et al. 2017).

Conclusions

In this entry, we introduced HopsFS, an open-source, highly available, drop-in alternative for HDFS that provides highly available metadata that scales out in both capacity and throughput by adding new namenodes and database nodes. We designed HopsFS file system operations to be deadlock-free by meeting the two preconditions for deadlock freedom: all operations follow the same total order when taking locks, and locks are never upgraded. Our architecture supports a pluggable database storage engine, and other NewSQL databases could be used, providing they have good support for cross-partition transactions and techniques such as partition-pruned index scans. Most importantly, our architecture now opens up metadata in HDFS for tinkering: custom metadata only involves adding new tables and cleaner logic at the namenodes.

Cross-References

- Distributed File Systems
- In-memory Transactions
- Hadoop

References

Abad CL (2014) Big data storage workload characterization, modeling and synthetic generation. PhD thesis, University of Illinois at Urbana-Champaign

Guerraoui R, Raynal M (2006) A leader election protocol for eventually synchronous shared memory systems. In: The fourth IEEE workshop on software technologies for future embedded and ubiquitous systems, 2006 and the 2006 second international workshop on collaborative computing, integration, and assurance, SEUS 2006/WCCIA, pp 6–

Hammer-Bench (2016) Distributed metadata benchmark to HDFS. https://github.com/smkniazi/hammer-bench. [Online; Accessed 1 Jan 2016]

Ismail M, Gebremeskel E, Kakantousis T, Berthou G, Dowling J (2017) Hopsworks: improving user experience and development on hadoop with scalable, strongly consistent metadata. In: 2017 IEEE 37th international conference on distributed computing systems (ICDCS), pp 2525–2528

Ismail M, Niazi S, Ronström M, Haridi S, Dowling J (2017) Scaling HDFS to more than 1 million operations per second with HopsFS. In: Proceedings of the 17th IEEE/ACM international symposium on cluster, cloud and grid computing, CCGrid '17. IEEE Press, Piscataway, pp 683–688

Niazi S, Haridi S, Dowling J (2017) Size matters: improving the performance of small files in HDF. https://eurosys2017.github.io/assets/data/posters/poster09-Niazi.pdfl. [Online; Accessed 30 June 2017]

Niazi S, Ismail M, Haridi S, Dowling J, Grohsschmiedt S, Ronström M (2017) Hopsfs: scaling hierarchical file system metadata using newsql databases. In: 15th USENIX conference on file and storage technologies (FAST'17). USENIX Association, Santa Clara, pp 89–104

Noll MG (2015) Benchmarking and stress testing an hadoop cluster with TeraSort. TestDFSIO & Co. http://www.michael-noll.com/blog/2011/04/09/benchmarking-and-stress-testing-an-hadoop-cluster-with-terasort-testdfsio-nnbench-mrbench/. [Online; Accessed 3 Sept 2015]

Ovsiannikov M, Rus S, Reeves D, Sutter P, Rao S, Kelly J (2013) The quantcast file system. Proc VLDB Endow 6(11):1092–1101

Patil SV Gibson GA Lang S, Polte M (2007) GIGA+: scalable directories for shared file systems. In: Proceedings of the 2nd international workshop on petascale data storage: held in conjunction with supercomputing '07, PDSW '07. ACM, New York, pp 26–29

Ren K, Kwon Y, Balazinska M, Howe B (2013) Hadoop's adolescence: an analysis of hadoop usage in scientific workloads. Proc VLDB Endow 6(10):853–864

Salman Niazi GB, Ismail M, Dowling J (2015) Leader election using NewSQL systems. In: Proceeding of DAIS 2015. Springer, pp 158–172

Shvachko KV (2010) HDFS scalability: the limits to growth. Login Mag USENIX 35(2):6–16

Thomson A, Abadi DJ (2015) CalvinFS: consistent WAN replication and scalable metadata management for distributed file systems. In: 13th USENIX conference on file and storage technologies (FAST 15). USENIX Association, Santa Clara, pp 1–14