Scaling HDFS with a Strongly Consistent Relational Model for Metadata

Kamal Hakimzadeh, Hooman Peiro Sajjad, Jim Dowling

Scaling HDFS with a Strongly Consistent Relational Model for Metadata

Kamal Hakimzadeh, Hooman Peiro Sajjad, Jim Dowling

KTH - Royal Institute of Technology
Swedish Institute of Computer Science (SICS)
{mahh, shps, jdowling}@kth.se

Abstract. The *Hadoop Distributed File System* (HDFS) scales to store tens of petabytes of data despite the fact that the entire file system's metadata must fit on the heap of a single Java virtual machine. The size of HDFS' metadata is limited to under 100 GB in production, as garbage collection events in bigger clusters result in heartbeats timing out to the metadata server (*NameNode*).
In this paper, we address the problem of how to migrate the HDFS' metadata to a relational model, so that we can support larger amounts of storage on a shared-nothing, in-memory, distributed database. Our main contribution is that we show how to provide at least as strong consistency semantics as HDFS while adding support for a multiple-writer, multiple-reader concurrency model. We guarantee freedom from deadlocks by logically organizing inodes (and their constituent blocks and replicas) into a hierarchy and having all metadata operations agree on a global order for acquiring both explicit locks and implicit locks on subtrees in the hierarchy. We use transactions with pessimistic concurrency control to ensure the safety and progress of metadata operations. Finally, we show how to improve performance of our solution by introducing a snapshotting mechanism at NameNodes that minimizes the number of roundtrips to the database.

1 Introduction

Distributed file systems, such as the *Hadoop Distributed File System (HDFS)*, have enabled the open-source Big Data revolution, by providing a highly available (HA) storage service that enables petabytes of data to be stored on commodity hardware, at relatively low cost [2]. HDFS' architecture is based on earlier work on the Google Distributed File System (GFS) [4] that decoupled metadata, stored on a single node, from block data, stored across potentially thousands of nodes. In HDFS, metadata is kept in-memory on a single NameNode server, and a system-level lock is used to implement a multiple-reader, single writer concurrency model. That is, HDFS ensures the consistency of metadata by only allowing a single client at a time to mutate its metadata. The metadata must fit on the heap of a single *Java virtual machine* (JVM) [10] running on the NameNode.

The current implementation of HDFS does, however, support highly available metadata through an eventually consistent replication protocol, based on the

Active/Standby replication pattern, but limited to having a single standby node. All read and write requests are handled by the Active node, as reads at the Standby node could return stale data. The replication protocol is based on the Active node making quorum-based updates to a recovery log, called the *edit log*, persistently stored on a set of journal nodes. The Standby node periodically pulls updates to the edit log and applies it to its in-memory copy of the metadata. The quorum-based replication protocol requires at least three journal nodes for high availability. Failover from the Active to the Standby can, however, take several tens of seconds, as the Standby first has to apply the set of outstanding edit log entries and all nodes need to reach agreement on who the current Active node is. They solve the latter problem by using a Zookeeper coordination service that also needs to run on at least three nodes to provide a high availability [7].

The challenge we address in this paper is how to migrate HDFS' metadata from highly optimized data structures stored in memory to a distributed relational database. Using a relational database to store file system metadata is not a novel idea. WinFs [13], a core part of the failed Windows Longhorn, was supposed to use Microsoft SQL Server to store its file system metadata, but the idea was abandoned due to poor performance. However, with the advent of *New SQL* systems [17], we believe this is an idea whose time has now come. Recent performance improvements for distributed in-memory databases make it now feasible. Version 7.2 of MySQL Cluster, an open-source new SQL database by Oracle, supports up to 17.6 million transactional 100-byte reads/second on 8 nodes using commodity hardware over an infiniband interconnect [17]. In addition to this, recent work on using relational databases to store file system metadata has shown that relational databases can outperform traditional inode data structures when querying metadata [5].

Our implementation of HDFS replaces the Active-Standby and eventually consistent replication scheme for metadata with a transactional shared memory abstraction. Our prototype is implemented using MySQL Cluster [14]. In our model, the size of HDFS' metadata is no longer limited to the amount of memory that can be managed on the JVM of a single node [10], as metadata can now be partitioned across up to 48 nodes. By applying fine-grained pessimistic locking, our solution allows multiple compatible write operations [8] to progress simultaneously. Even though our prototype is built using MySQL Cluster, our solution can be generalized to support any transactional data store that either supports transactions with at least read-committed isolation level and row-level locking. Our concurrency model also requires implicit locking, and is motivated by Jim Gray's early work on hierarchical locking [8]. We model all HDFS metadata objects as a directed acyclic graph of resources and then with a row-level locking mechanism we define the compatibility of metadata operations so as to isolate transactions for the fewest possible resources allowing a maximum number of concurrent operations on metadata. We show how serializable transactions are required to ensure the strong consistency of HDFS' metadata, by showing how anomalies that can arise in transaction isolation levels lower than serializable [1] can produce inconsistencies in HDFS metadata operations. As our solution

produces a high level of load on the database, we also introduce a snapshot layer (per-transaction cache) to reduce the number of roundtrips to the database.

2 HDFS Background and Concurrency Semantics

Distributed file systems have typically attempted to provide filesystem semantics that are as close as possible to the POSIX strong model of consistency [19]. However, for some operations, HDFS provides a consistency level weaker than POSIX. For example, because of the requirement to be able to process large files that are still being written, clients can read files that are opened for writing. In addition, files can be appended to, but existing blocks cannot be modified. At the file block level, HDFS can be considered to have sequential consistency semantics for read and write operations [19], since after a client has successfully written a block, it may not be immediately visible to other clients. However, when a file has been closed successfully, it becomes immediately visible to other clients, that is, HDFS supports linearizability [6] at the file read and write level.

Metadata in HDFS. Similar to POSIX file systems, HDFS represents both directories and files as inodes (*INode*) in metadata. Directory inodes contain a list of file inodes, and files inodes are made up a number of blocks, stored in a *BlockInfo* object. A block, in its turn, is replicated on a number of different data nodes in the system (default 3). Each replica is a *Replica* object in metadata. As blocks in HDFS are large, typically 64-512 MB in size, and stored on remote DataNodes, metadata is used to keep track of the state of blocks. A block being written is a *PendingBlock*, while a block can be under-replicated if a DataNode fails (*UnderReplicatedBlock*) or over-replicated (*ExcessReplica*) if that DataNode recovers after the block has been re-replicated. Blocks can also be in an *InvalidatedBlock* state. Similarly, replicas (of blocks) can be in *ReplicaUnderConstruction* and *CorruptedReplica* states. Finally, a *Lease* is a mutual grant for a number of files being mutated by a single client while *LeasePath* is an exclusive lock regarding a single file and a single client.

Tying together the NameNode and DataNodes. Filesystem operations in HDFS, such as file open/close/read/write/append, are blocking operations that are implemented internally as a sequence of *metadata operations* and *block operations* orchestrated by the client. First, the client queries or mutates metadata at the NameNode, then blocks are queried or mutated at DataNodes, and this process may repeat until the filesystem operation returns control to the client. The consistency of filesystem operations is maintained across metadata and block operations using *leases* stored in the NameNode. If there is a failure at the client and the client doesn't recover, any leases held by the client will eventually expire and their resources will be freed. If there is a failure in the NameNode or a DataNode during a filesystem operation, the client may be able to retry the operation to make progress or if it cannot make progress it will try to release the leases and return an error (the NameNode needs to be contactable to release leases).

HDFS' Single-Writer Concurrency Model for Metadata Operations. The NameNode is the bottleneck preventing increased write scalability for HDFS applications [16], and this bottleneck is the result of its multiple-reader, single-writer concurrency model [20]. Firstly, metadata is not partitioned across nodes. Secondly, within the NameNode, a global read/write lock (*FSNamesystem lock* - a Java language ReentrantReadWriteLock) protects the namespace by grouping the NameNode's operations into read or write operations. The NameNode uses optimized data structures like multi-dimensional linked-lists for accessing blocks, replicas and DataNode information on which it is almost impossible to use fine-grained concurrency control techniques. The data structures are tightly coupled, and generally not indexed as memory access is fast and indexing would increase metadata storage requirements.

As the FSNamesystem lock is only acquired while updating metadata in memory, the lock is only held for a short duration. However, write operations also incur at least one network round-trip as they have to be persisted at a quorum of journal nodes. If writes were to hold FSNamesystem lock while waiting for the network round-trip to complete, it would introduce intolerable lock contention. So, writes release the FSNamesystem lock after applying updates in memory, while waiting for the updates to be persisted to the journal nodes. In addition to this, to improve network throughput to the journal nodes, writes are sent in batches [11] to journal nodes. When batched writes return, the thread waiting for the write operation returns to the client. However, thread scheduling anomalies at the NameNode can result in writes returning out-of-order, thus violating linearizability of metadata. As threads don't hold the FSNamesystem lock while waiting for edits to complete, it is even possible that thread scheduling anomalies could break sequential consistency semantics by returning a client's writes out-of-order. However, metadata operations also acquire leases while holding the FSNamesystem lock, thus making individual filesystem operations linearizable.

3 Problem Definition

We are addressing the problem of how to migrate HDFS' metadata to a relational model, while maintaining consistency semantics at least as strong as HDFS' NameNode currently provides. We assume that the database supporting the relational model provides support for transactions. While metadata consistency could be ensured by requiring that transactions' execute at a serializable isolation level, distributed relational databases typically demonstrate poor throughput when serializing all updates across partitions [18]. In HDFS, filesystem operations typically traverse the root directory, and the root inode is, therefore, a record that is frequently involved in many different transactions. As the root directory can only be located on one partition in a distributed database, transactions that take a lock on the root (and other popular directories) will frequently cross partitions. Thus, the root directory and popular directories become a synchronization bottleneck for transactions. A challenge is safely removing them from transactions' contention footprint, without having to give up on strong consistency for metadata. If we are to implement a consistency model at least as

strong consistency as that provided for HDFS' metadata, we need transactions, and they need to support at least read-committed isolation level and row-level locks, so that we can implement stronger isolation levels when needed.

Another challenge we have is that, in the original HDFS, metadata operations, each executed in their own thread, do not read and write metadata objects in the same order. Some operations may first access blocks and then inodes, while other operations first access inodes and then blocks. If we encapsulate metadata operations in transactions, locking resources as we access them, cycles will be introduced resulting in deadlocks. Another problem, that is also an artifact of HDFS's NameNode design, is that many operations read objects first, and then update them later within the same metadata operation. When these operations are naively implemented as transactions, deadlock occurs due to transactions upgrading read locks to exclusive locks.

Finally, as we are moving the metadata to remote hosts, an excessive number of roundtrips from a NameNode to the database increases the latency of filesystem operation latencies and reduces throughput. Although we cannot completely avoid network roundtrips, we should avoid redundant fetching of the same metadata object during the execution of a transaction.

4 Hierarchical Concurrency Model

The goal of our concurrency model is to support as a high a degree of concurrent access to HDFS' metadata as possible, while preserving freedom from deadlock and livelock. Our solution is based on modelling the filesystem hierarchy as a *directed acyclic graph (DAG)*, and metadata operations that mutate the DAG are in a single transaction or that either commits or, in the event of partial failures in the distributed database, aborts. Aborted operations are transparently retried at NameNodes unless the error is not recoverable. Transactions pessimistically acquire locks on directory/file subtrees and file/block/replica subtrees, and these locks may be either explicit or implicit depending on the metadata operation. *Explicit locking* requires a transaction to take locks on all resources in the subtree. *Implicit locking*, on the other hand, only requires a transaction to take one explicit lock on the root of a subtree and it then implicitly acquires locks on all descendants in the subtree. There is a trade-off between overhead of taking too many locks with explicit locking over lower level of concurrency with implicit locking [8]. However, metadata operations not sharing any subtrees can be safely executed concurrently.

4.1 Building a DAG of Metadata Operations

After careful analysis of all metadata operations in the NameNode, we have classified them into three different categories based on the primary metadata object used to start the operation:

1. *path operations*,
2. *block operations,*
3. *lease operations.*

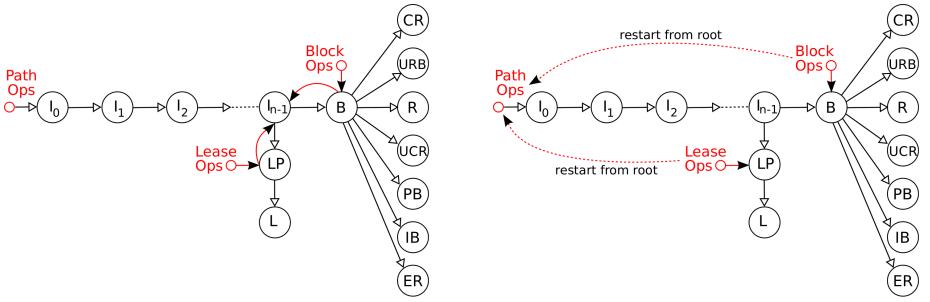
The majority of HDFS' metadata operations are path operations that take an absolute filesystem path to either a file or directory as their primary parameter. Path operations typically lock one or more inodes, and often lock block objects, lease paths and lease objects. Block operations, on the other hand, take a block identifier as their primary parameter and contain no inode information. An example block operation is *AddBlock*: when a block has been successfully added to a DataNode, the DataNode acknowledges that the block has been added to the NameNode that then updates the block's inode. Blocks are unique to inodes, as HDFS does not support block sharing between files. Lease operations also provide a filesystem path, but it is just a subpath that is used to find all the lease-paths for the files containing that subpath. In figure 1a, we can see how block and lease operations can mutate inodes, introducing cycles into the metadata hierarchy and, thus, deadlock.

Our solution to this problem, in figure (1b), is to break up both block operations and lease operations into two phases. In the first phase, we start a transaction that executes only read operations, resolving the inodes used by the operations at a read committed isolation level. This transaction does not introduce deadlock. In the second phase, we start a new transaction that acquires locks in a total order, starting from the root directory. This second transaction needs to validate data acquired in the first phase (such as inode id(s)). Now path, block and lease operations all start acquiring locks starting from the root inode.

We need to ensure that metadata operations do not take locks on inodes in a conflicting order. For example, if a metadata operation $operation(x, y)$ that take two inodes as parameters always takes a lock on the first inode x then on the second inode y, then the concurrent execution of $operation(a, b)$ and $operation(b, a)$ can cause deadlock. The solution to this problem is to define a total ordering on inodes, a *total order rule*, and ensure all transactions acquire locks on inodes using this global ordering. The total order follows the traversal of the file system hierarchy that depth-first search would follow, traversing first towards the leftmost child and terminating at the rightmost child. The first inode is the root inode, followed by directory inodes until the leftmost child is reached, then all nodes in that directory, then going up and down the hierarchy until the last inode is reached.

More formally, we use the hierarchy of the file system to map inodes to a partially ordered set of IDs. A transaction that already holds a lock for an inode with ID m can only request a lock on an inode with ID n if $n > m$. This mechanism also implements implicit locking, as directory inodes always have a lower ID than all inodes in its subtree.

Our total ordering is impossible for range queries (with or without indexes), because not all databases support ordered queries. We fix this issue by also taking implicit locks in such cases. As paths are parsed in a consistent order from the root to leaf inodes in the path, when we take an exclusive lock on a directory inode, we implicitly lock its subtree. This prevents concurrent access to the subtree, and thus reduces parallelism, but solves our problem. Fortunately,



(a) HDFS' Directed Graph has cycles.

(b) Acyclic DAG. Ops start from root, locks taken in order from leftmost child.

I: *INode*, ***B***: *BlockInfo*, ***L***: *Lease*, ***LP***: *LeasePath*, ***CR***: *CorruptedReplica*, ***URB***: *UnderRepliatedBlock*, ***R***: *Replica*, ***UCR***: *UnderConstructionReplica*, ***PB***: *PendingBlock*, ***IB***: *InvalidatedBlock*, ***ER***: *ExcessReplica*

Fig. 1: Access graph of HDFS metadata

typical operations, such as getting blocks for a file and writing to a file do not require implicit locks at the directory level. However, we do take implicit locks at the file inode level, so when a node is writing to a file, by locking the inode, we implicitly lock all block and replica objects within that file.

4.2 Preventing Lock Upgrades

A naive implementation of our relational model would translate read and write operations on metadata in the existing NameNode to read and write operations directly on the database. However, assuming each metadata operation is encapsulated inside a single transaction, such an approach results in locks being upgraded, potentially causing deadlock. Our solution is to only acquire a lock once on each data item within a metadata operation, and we take the lock with the highest strength lock that will be required for the duration of that transaction.

4.3 Snapshotting

As we only want to acquire locks once for each data item, and we are assuming an architecture where the NameNode accesses a distributed database, it makes no sense for the NameNode to read or write the same data item more than once from the database within the context of a single transaction. For any given transaction, data items can be cached and mutated at a NameNode and only updated in the database when the transaction commits. We introduce a snapshotting mechanism for transactions that, at the beginning of each transaction, reads all the resources a transaction will need, taking locks at the highest strength that will be required. On transaction commit or abort, the resources are freed. This solution enables NameNodes to perform operations on the *per-transaction cache* (or snapshot) of the database state during the transaction, thus reducing the number of roundtrips required to the database. Note, this technique is not implementing snapshot isolation [1], we actually support serializable transactions.

Algorithm 1 Snapshotting taking locks in a total order.

```
1: snapshot.clear

2: operation doOperation
3:     tx.begin
4:     create-snapshot()
5:     performTask()
6:     tx.commit

7: operation create-snapshot
8:     S = total_order_sort(op.X)
9:     foreach x in S do
10:        if x is a parent then level = x.parent_level_lock
11:        else level = x.strongest_lock_type
12:        tx.lockLevel(level)
13:        snapshot += tx.find(x.query)
14:    end for

15: operation performTask
16:    //Operation Body, referring to transaction cache for data
```

An outline of our concurrency model for transactions, including total order locks and snapshotting, is given in algorithm 1.

5 Correctness Discussion

In our solution, transactions are serializable, meaning that transactions are sortable in the history of operations. Therefore, it is always true that at any moment in time, all readers get the final and unique view of the mutated data which is strongly consistent. We ensure that transactions that contain both a read and a modify filesystem operation for the same shared metadata object should be serialized based on the *serialization rule:*

- $\forall (w_i, w_j)\, if\, X_{wi} \cap X_{wj} \neq \emptyset$ then transactions of (w_i, w_j) must be serialized;
- $\forall (r_i, w_j)\, if\, X_{ri} \cap X_{wj} \neq \emptyset$ then transactions of (r_i, w_j)must be serialized.

First, we use the hierarchy of the file system to define a partial ordering over inodes. Transactions follow this partial ordering when taking locks, ensuring that the circular wait condition for deadlock never holds. Similarly, the partial ordering ensures that if a transaction takes an exclusive lock on a directory inode, subsequent transactions will be prevented from accessing the directory's subtree until the lock on the directory's lock is released. Implicit locks are required for operations such as creating files, where concurrent metadata operations could return success even though only one of them actually succeeded. For operations such as deleting a directory, explicit locks on all child nodes are required.

To show that our solution is serializable, we use an anomalies-based definition of isolation levels, and then we justify why none of these anomalies happen in our solution [1]. The list of anomalies that can arise in transactions are namely *Dirty Write*, *Dirty Read*, *Fuzzy Read*, *Lost Update*, *Read Skew, Write Skew,* and

Phantom Reads [1]. Assuming well-formed locking [1], that is, we have no bugs in our locking code, then the system guarantees that it is never possible that two concurrent transactions could mutate the same data item. This prevents *Dirty Reads and Write*, as well as *Fuzzy Reads* and *Read Skew*. Similarly, *Lost Updates* only occur if we do not have well-formed locking. Similarly, *Write Skew* is impossible, as a reader and writer transactions require concurrent access to the same data item. Likewise for a single data item, predicates are also taken into account in our solution in the form of implicit locks. All predicates are also locked even if the metadata operation does not intend to change them directly, thus making *Phantom Reads* impossible. Finally, we only execute index scans when we have an implicit lock preventing the insertion of new rows that could be returned by that index scan. This means that, for example, when listing files in a directory we take an implicit lock on the directory so that no new files can be inserted in the directory while the implicit lock is held. Similarly, list all blocks for an inode only happens when we have an implicit lock on the inode.

6 Experiments

We used MySQL Cluster as the distributed relational database for metadata. In experiments the MySQL Cluster nodes and the NameNode run on machines each with 2 AMD 6 core CPUs (2.5 GHz clock speed, 512 KB cache size) connected with 1 GB Ethernet. The versions of our software were: MySQL Cluster 7.2.8, Java virtual machine 1.6 and ClusterJ 7.1.15a as the connector.

6.1 Capacity

Based on Shvachko in [16], HDFS files on average contain 1.5 blocks and, assuming a replication factor of 3, then 600 bytes of memory is required per file. Due to garbage collection effects, the upper limit on the size of the JVM heap for the NameNode is around 60GB, enabling the NameNode to store roughly 100 million files [16]. Existing clusters at Facebook have larger block sizes, up to 1 GB, and carefully configure and manage the JVM to scale the heap up to 100 GB, leading to larger clusters but not to significantly more files. For our NameNode, we estimate the amount of metadata consumed per file by taking into account that each INode, BlockInfo and Replica row in database require 148, 64 and 20 bytes, respectively. Per file, our system creates 1 INode, 2 BlockInfo and 6 Replica rows, which is 396 bytes. MySQL Cluster supports up to 48 data-nodes and, in practice, each node can have up to 256GB of memory for storage. So in principle, a MySQL Cluster implementation can scale up to 12 TB in size, although the largest cluster we are aware of is only 4 TBs. If we conservatively assume that MySQL Cluster can support up to 3.072 TB for metadata, then with a replication factor of 2 for the metadata in MySQL cluster, our file system can store up to 4.1 billion files. This is a factor of 40 increase over Shvachko's estimate for HDFS from 2010.

6.2 Snapshots reduce the Number of Roundtrips to the Database

Our snapshotting layer, or *Transaction Cache,* caches data items retrieved from the database in the local memory of the NameNode. This minimizes the num-

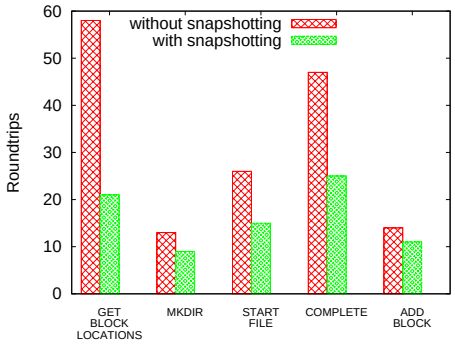


Fig. 2: Impact of snapshotting on database roundtrips

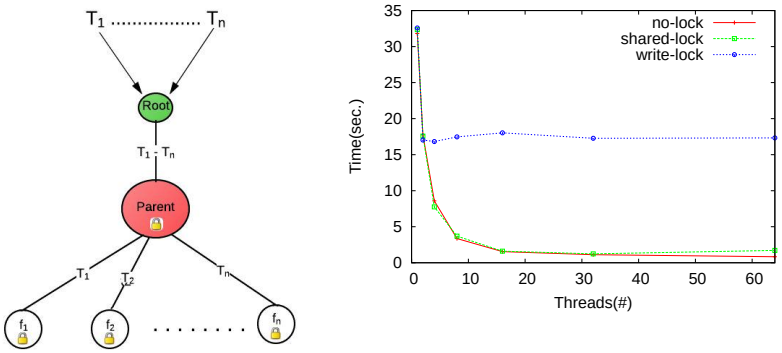
ber of roundtrips to the database and consequently the overall latency for the metadata operation. We wrote an experiment to analyze a number of popular metadata operations, counting the number of roundtrips to the database that our *Transaction Cache* saves for each metadata operation. GET_BLK_LOC is a metadata operation that returns the addresses of the DataNodes storing a replica of a given block. MKDIR creates directories recursively. START_FILE creates INodes for all the non-existent inodes, writes the owner of the lease and creates a new lease-path entry. COMPLETE, sent by the client after having successfully written the last block of a file, removes all the under-construction-replicas and marks the corresponding BlockInfo as complete. ADD_BLOCK adds a new BlockInfo and returns a list containing the location for its replicas. As can be seen in figure 2, GET_BLK_LOC, START_FILE, and COMPLETE reduce the number of roundtrips to the database by 60%, 40% and 50%.

6.3 Row-level Lock in MySQL Cluster

To demonstrate the feasibility of our approach on a real database, we present a micro-benchmark on the performance of row-level locking in MySQL Cluster. In this setup, MySQL Cluster has 4 DataNodes, each running on a different host. In this experiment, we vary the number of threads and the lock type taken, while we measure the total time for threads to read a set of rows of data in a pre-existing namespace. This experiment simulates the cost of a taking a lock on a parent directory and then reading the rows required to read in a file (inode, blocks, and replicas).

In the namespace structure in figure 3a, the root and a parent directory are shared between all threads while each thread is assigned just one file to read. All threads read the root directory without a lock (at read committed isolation level), but they each acquire a different type of lock on the parent directory. Threads that take write locks on the parent directory must be executed serially, while threads that take either a shared lock (read lock) or no lock can execute in parallel.

The results for 10,000 transactions are shown in the figure 3b. As the number of threads is increased, the time to perform 10,000 transactions decreases



(a) Benchmark Namespace (b) MySQL Cluster throughput benchmark

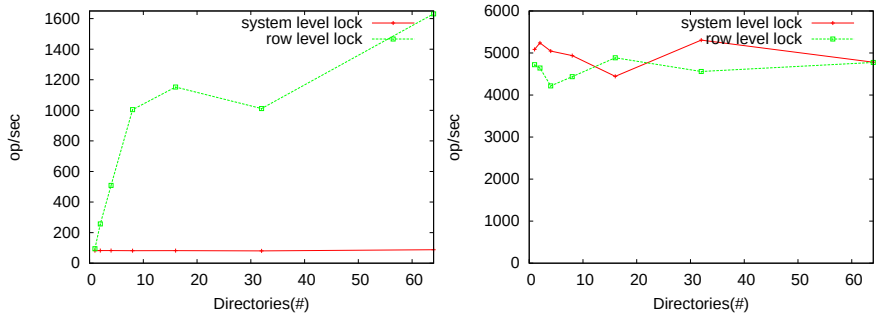
Fig. 3: Influence of locking on MySQL Cluster throughput

almost linearly for reading with shared lock until about 30 threads are run in parallel, then the time taken levels out, finally increasing slightly, starting from 50+ threads. We believe this increase is because of the extra overhead of acquiring/releasing locks at the data nodes in MySQL Cluster. For transactions that do not take any locks, the time taken decreases continually up to the 60 threads used in our experiments. However, for the write lock, we can see that the total time is halved for more than one thread but it doesn't decrease after that. This is because only one thread can acquire the write lock on the parent at a time, and the threads must wait until the lock is released before they can read the data.

6.4 System-level vs Row-level Locking

In order to compare the performance of Apache HDFS' NameNode using a system-level lock (*FSNamesystem lock*) with our NameNode that uses row-level locks, we implemented a NameNode benchmark as an extension of NNThroughputBenchmark [16]. In this benchmark, we measure the throughput of open and create operations on two different locking mechanisms with a constant 64 number of threads while increasing the number of directories (decreasing number of files per directory). The number of concurrent threads on each parent directory is a function of the number of directories. The modification we made to the NNThroughputBenchmark is that we allocated one directory per thread.

The result of a create operation for 16384 (2^{14}) files is depicted in figure 4a. As the diagram shows, the throughput for creating files under a single directory is the same for both the system-level lock in Apache's HDFS and our row-level lock. This is because all 64 threads try to acquire a write lock on the same directory in the NameNode. The advantage of row-level locking is seen when we increase the number of directories. Increasing the number of directories, we see that the throughput of our NameNode with row-level locking increases while for Apache's NameNode using a system-level lock, the throughput remains almost



(a) Create Operation (b) Open Operation

Fig. 4: Impact of the row-level lock on throughput

constant. For the *open* operation in figure 4b, row-level locking and system-level locking perform almost the same, since both row-level and system-level locks allow multiple read locks to be acquired simultaneously.

7 Related Work

Microsoft's WinFs [13] is an early attempt at building a distributed filesystem for a networked desktop environment with the goal of centrally managing updates to the distributed filesystem's metadata in a relational database. The database supported efficient searching and filtering of metadata. However, in contrast to our model clients mutated local copies of the filesystem and then used peer-to-peer synchronization with conflict resolution, resulting in a model where the metadata was eventually consistent.

HDFS v1 [2] follows the *single metadata server* model. Google's GFS [4], HDFS v2 [20] and TidyFs [3] have a highly available *master/slave replication model* for metadata, while Ceph [21] and Lustre [15] partition metadata across *distributed metadata servers*. However, in contrast to HDFS, Ceph and Lustre are object-based file systems, somewhat simplifying the partitioning of metadata. Google's successor to GFS, Collosus, also partitions metadata across many machines, but the mechanisms of how they maintain consistency across partitions are not public knowledge [12]. Similarly, MapR has built a new proprietary version of HDFS with partitioned metadata, although MapR have stated at the Hadoop Summit 2011 that the metadata is not replicated using a strongly consistent replication protocol.

GFS's replication scheme is based on batching operation log updates to the local disk of the primary server and replicating them to a remote server to handle failures by reconstructing state of the file system. TidyFs [3] supports flat URL-based files and replicates metadata using Paxos [9]. In Ceph [21], the main data structure is the Directory that stores information about its child inodes. By using a two phase commit algorithm, Ceph dynamically partitions the namespace tree and evenly distributes it over a cluster of metadata servers (MDS). Lustre's

general object model stores both metadata and block data in a cluster of Object Storage Devices (OSD) [15], so for replication Lustre relies on the promises of the underlying OSD cluster, while its Distributed Lock Manager (DLM) library assures the consistency of the file system.

It is also worth mentioning that our approach is similar to multi-version concurrency control (MVCC), in that we take a copy of the data items, however, in contrast to MVCC, we only work on a single copy of the data items and we take pessimistic locks.

8 Conclusion and Future Work

In this paper, we introduced a new relational model for HDFS' metadata. We also showed how to migrate metadata from highly performant and in-memory data structures in Apache's HDFS to a relational representation that guarantees strong consistency for metadata. In particular, we demonstrated that how metadata operations could be made serializable and deadlock-free using pessimistic-locking concurrency control, requiring only locking and transactions that support a read-committed isolation level. The mechanisms we introduced to ensure freedom from deadlock were the representation of a logical DAG for the metadata, specifying a global order for acquiring locks on metadata objects, preventing lock upgrades, and using of implicit locks to lock subtrees. We showed that the performance of the database underlying our system is competitive, and that our NameNode architecture can potentially scale to store 40 times more metadata than Apache's NameNode.

In future work, we will extend our system to support a multiple NameNode architecture. Certain cluster-wide NameNode tasks, such as replication management, should only be executed by a single NameNode at a time. We are also working on optimizing our schemas for representing inodes, blocks, and replicas to reduce their footprint, and to support distribution aware transactions - we want to reduce the number of partitions that transactions for common operations must cross to help improve throughput.

References

1. Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A Critique of ANSI SQL Isolation Levels. *ACM SIGMOD Record*, 24(2):1–10, 1995.
2. Dhruba Borthakur. The Hadoop Distributed File System: Architecture and Design, 2007. http://hadoop.apache.org/docs/r0.18.0/hdfs_design.pdf, [Online; accessed 20-Nov-2011].
3. Dennis Fetterly, Maya Haridasan, Michael Isard, and Swaminathan Sundararaman. TidyFS: A Simple and Small Distributed File System. In *Proc. of USENIXATC'11*, page 34. USENIX Association, 2011.
4. Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proc. of SOSP'03*, pages 29–43. ACM, 2003.
5. Haryadi S Gunawi, Abhishek Rajimwale, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. SQCK: A Declarative File System Checker. In *Proc. of OSDI'08*, pages 131–146. USENIX Association, 2008.

6. Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM TPL*, 12(3):463–492, 1990.
7. Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proc. of USENIXATC'10*, pages 11–11. USENIX Association, 2010.
8. G. Putzolu J. Gray, R. Lorie and I. Traiger. Granularity of Locks and Degrees of Consistency in a Shared Database. In *IFIP Working Conference on Modelling in Data Base Management Systems*, pages 365–394. IFIP, 1976.
9. Leslie Lamport. The Part-Time Parliament. *ACM TOCS'98*, 16(2):133–169, 1998.
10. Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification*. Addison-Wesley, 2013.
11. Todd Lipcon. Quorum-Journal Design, 2012. https://issues.apache.org/jira/browse/HDFS-3077, [Online; accessed 11-Dec-2012].
12. Marshall Kirk McKusick and Sean Quinlan. GFS: Evolution on Fast-forward. *ACM Queue*, 7(7):10, 2009.
13. Lev Novik, Irena Hudis, Douglas B Terry, Sanjay Anand, Vivek Jhaveri, Ashish Shah, and Yunxin Wu. Peer-to-Peer Replication in WinFS. *Technical ReportMSR-TR-2006-78, Microsoft Research*, 2006.
14. Mikael Ronström and Jonas Oreland. Recovery Principles of MySQL Cluster 5.1. In *Proc. of VLDB'05*, pages 1108–1115. VLDB Endowment, 2005.
15. Philip Schwan. Lustre: Building a File System for 1000-node Clusters. In *Proc. of OLS'03*, 2003.
16. Konstantin V Shvachko. HDFS Scalability: The limits to growth. *login*, 35(2):6–16, 2010.
17. Michael Stonebraker. New Opportunities for New SQL. *CACM*, 55(11):10–11, 2012.
18. Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proc. of SIGMOD'12*, pages 1–12. ACM, 2012.
19. José Valerio, Pierre Sutra, Étienne Rivière, and Pascal Felber. Evaluating the Price of Consistency in Distributed File Storage Services. In *Proc. of DAIS'2013*, pages 141–154. Springer, 2013.
20. Feng Wang, Jie Qiu, Jie Yang, Bo Dong, Xinhui Li, and Ying Li. Hadoop High Availability Through Metadata Replication. In *Proc. CloudDB'09*, pages 37–44. ACM, 2009.
21. Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proc. of OSDI'06*, pages 307–320. USENIX Association, 2006.