Scalability Limits of HDFS

Christopher Chute, David Brandfonbrener, Leo Shimonaka, Matthew Vasseur

May 3, 2016

Abstract

This paper explores the scalability limits of the Hadoop Distributed File System (HDFS). The single-machine, in-memory metadata store of HDFS is a known bottleneck for applications requiring storage of many small files. In addition, the singular NameNode presents possible CPU bottlenecks when scaling for storage of a large number of small files. For example, small files create more complex heartbeats and are associated with more frequent access request patterns, both of which increase the CPU load on the NameNode. In this paper we first model and provide experimental evidence for the NameNode memory bottleneck. We compare these results to proposed CPU bottlenecks, and we find that the memory bottleneck dominates in all reasonable usage patterns. We conclude with suggestions for future work on HDFS scalability limits.

1 Introduction

This paper examines the scalability limits of the Hadoop Distributed File System (HDFS), a module of the Apache Hadoop ecosystem. HDFS is a distributed file system focused on scalability and portability. HDFS is not a POSIX-compliant file system in that it trades some POSIX requirements such as mutable files to increase the throughput and concurrency of distributed reads.

The architecture of HDFS is designed with the following goals in mind [3]:

1. *Support regular hardware failures.* Data is replicated across multiple DataNodes for fault tolerance.

2. *Allow streaming data access.* Partitioning is transparent so that files appear to be stored sequentially. Reads are distributed so that streaming access does not block other transactions.

3. *Support very large data sets.* To increase the file system's storage capacity, more machines can be added to an HDFS cluster. Importantly, this assumes that the NameNode memory bottleneck (discussed in Section 3) has not been reached.

4. *Provide simple concurrency control.* In the write-once, read-many model of HDFS, only one appender/creator per file is allowed at a given time. Written data is immutable. This allows HDFS to provide highly available, concurrent reads.

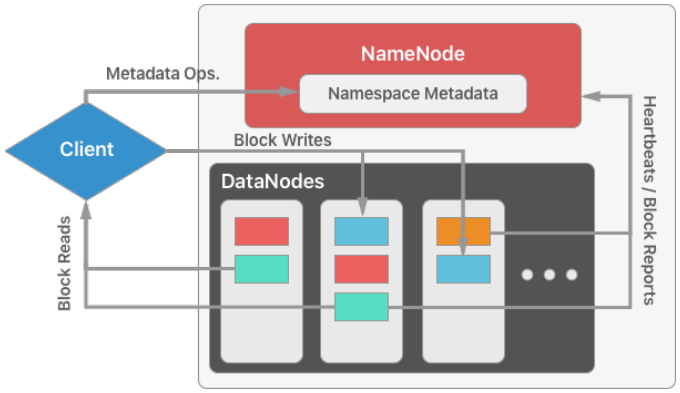
5. *Universal portability.* HDFS runs on top of the Java Virtual Machine, and thus can be run on any architecture and operating system that supports the JVM.

One key design choice of HDFS is the decoupling of metadata from file data. Although its centralized namespace storage offers several advantages for linear performance scaling, it also introduces hard bounds for growth. This paper explores how these limitations occur, analyzes its objective effects on performance, and discusses potential approaches to overcoming this limit.

The rest of this paper is organized as follows. In Section 2, we outline the architecture of an HDFS cluster. Section 3 provides a high-level overview of how this architecture interferes with an HDFS cluster's performance. Section 4 discusses the design and analysis of various experiments conducted on an HDFS cluster, and in Section 5 techniques are proposed for improving performance. Section 6 concludes.

2 HDFS Architecture

With the above principles in mind, the architecture of HDFS is designed as follows. Much of the information in this section is adapted from [1]. An HDFS cluster consists of a single *NameNode* and many *DataNodes*. The NameNode holds the metadata for every file in the file system. The metadata for each file consists of an (inode, mapping) pair. The inode is a UNIX-style inode and thus contains properties such as permissions and last modification time. Moreover, an HDFS inode stores HDFS-specific attributes such as quotas and replication factors.



Each file is partitioned into one or more *blocks*, and each block is replicated across multiple DataNodes. The mapping for a file is used to obtain the list of DataNodes on which each block of that file resides. As presented in Section 4.1.1, the size of metadata objects associated with each file scales linearly with the number of blocks in each file. That is, there is a fixed size mapping for each block in the file. The NameNode holds all of this metadata in memory to facilitate fast metadata operations.

A single NameNode is used primarily to simplify the architecture and metadata management of HDFS. For instance, having a single NameNode makes it simpler to maintain consistency in the file system. It also allows for simple, centrally organized replication of blocks across many DataNodes. If the metadata namespace were not entirely managed by NameNode, this replication operation and updating all data mappings would become complicated, and potentially overuse the network connecting the nodes.

Given a file of size F and a block size B, the file is partitioned into $n = \lceil F/B \rceil$ blocks. Upon creation of the file, the NameNode replicates each of the n blocks across R DataNodes, where R is the *replication factor* of the cluster. Once the file is created, the NameNode is responsible for pointing read or append requests to the appropriate blocks on the DataNodes. The NameNode keeps the block mappings up-to-date by communicating with the DataNodes via *block reports* carried in periodic *heartbeats* from each DataNode. A heartbeat also indicates that a DataNode is functioning correctly and has a working network connection. The NameNode uses this information to ensure each block is replicated over R DataNodes, issuing maintenance replication commands if needed.

Durability of the NameNode is critical to the basic functions of HDFS. There are various schemes to ensure a durable NameNode, including a write-ahead log that is maintained on stable storage. By default, the NameNode is a single point of failure for an HDFS cluster, and recovery of the NameNode after failure involves processing many block reports to reconstruct the file hierarchy.

Files in HDFS are append-only and can only be written to by a single writer at a time. That is, once data is written it cannot be overwritten. This greatly simplifies concurrency control and allows the system to guarantee that writes can be read as soon as a file is closed. On a write, a client must first query the NameNode to find which DataNodes and blocks to write. Then, the client will ensure a replicated write by sending packets of data through a pipeline of TCP connections. This pipeline includes all DataNodes over which the block will be replicated, and guarantees that the writes will be consistent across multiple DataNodes. On a read, a client queries the NameNode to find which DataNodes contain a file's blocks and block replicas. Then, the client reads from the closest available DataNodes directly.

3 Scalability Limits

Despite the benefits of having only a single NameNode, this architectural decision limits the scalability of HDFS. These problems are especially pronounced when HDFS is used to store many small files as opposed to few large files. The primary issue is that a single file must be partitioned into at least one block, even if its entire contents are less than the block size. Thus even small files produce a block mapping of metadata. As a result, small files produce more metadata relative to their content size. This exhausts the available memory of the NameNode even for moderately sized data sets when they consist of many small files. This memory bottleneck has been discussed in the literature, notably in [5]. For this same reason, small files result in larger block reports as each DataNode can hold a larger number of (virtual) blocks. Finally, the high-frequency access pattern associated with small files causes increased interaction with the NameNode, which creates another potential bottleneck.

3.1 Physical Memory Size

In this section we present an analysis of the memory bottleneck described above.

3.1.1 Theoretical Model

Let M be the size of memory on the NameNode in bytes, λ be the number of blocks allocated to each file, and F be the number of files on the system. Also, let c be the number of bytes to represent a single block mapping to all replicas of a block, and i the number of bytes to represent and inode. If we want to fit all metadata into memory, it is clear that we must have:

$$M > \lambda cF + iF$$

This limit is much easier to reach with small files since each file is allocated its own virtual block, no matter

how small the file is, *i.e.,* we have that $\lambda \geq 1$. This means that creating many small files should take up $c+i$ bytes for each additional file, while growing a file by a similar amount will use at most c bytes. Thus the amount of content stored is smaller for small files relative to the amount of metadata they produce. As a result, the memory limit is much more of a problem for many small files.

3.1.2 Simulation

Using this model, we can perform simulations of how much data can be stored in an HDFS cluster if each file is of a given size. The motivation for this is that with larger files, each file will use more blocks, and thus create more metadata. However, the cost of storing the inode relative to the amount of data in the file will decrease with larger files. It is interesting to see how both file size and block size affect the total amount of data that an HDFS cluster can store before the memory bottleneck is reached.

Below, we have graphs that represent the outcomes of these simulations. We assume that the NameNode has 2GB of RAM, and we perform the simulation for various block sizes (the size of each block that a file is divided into). We use the metadata sizes of $c \approx 175$ bytes and $i \approx 350$ bytes, which is empirically supported in Section 4.1.1. First, we ran the simulation with a block size of 128 MB. We have one plot of file sizes from 1 byte to 100 GB and another showing only file sizes larger than the block size of 128 MB, since this part is nonlinear and more interesting. Note the use of a logarithmic scale on each axis.

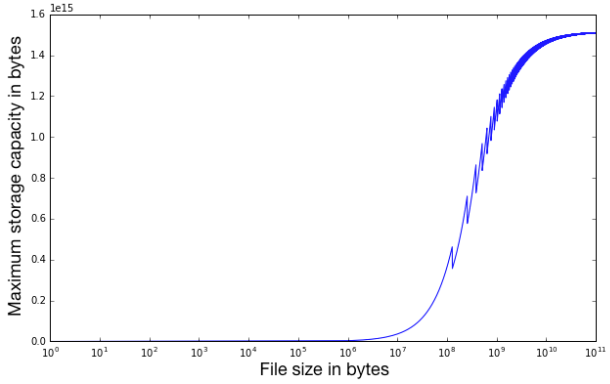


Fig. 2: Cluster Capacity vs. File Size

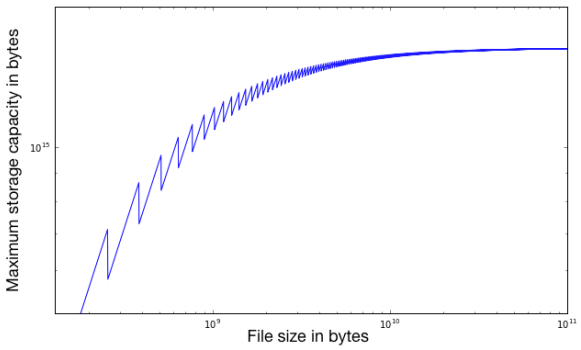


Fig. 3: Capacity vs. File Size (Zoomed Window)

While each file is smaller than one block, a file is always allocated its own block in the system and thus has a constant metadata size. Then, since the files are linearly increasing in the amount of data they hold, but metadata remains constant, the capacity of the system (as limited by the RAM on the NameNode) grows linearly until files are larger than blocks.

Once a file is larger than a block, the behavior becomes more interesting. At the threshold where the file size increases from n blocks to $n+1$ blocks, the cost of the extra metadata to record this block outweighs the increase in file size so that the overall capacity of the system decreases. This causes the zig-zag nature of the above graph. At the same time, the system also begins to reach the limit of the number of metadata mappings that can be stored in 2 GB, explaining why the graph flattens out with very large files.

Lastly, we ran the same simulation with many different block sizes. Since the block size determines the number of blocks per file, and thus the amount of metadata per file, a larger block size should create less metadata per file. This resulted in the following plot:

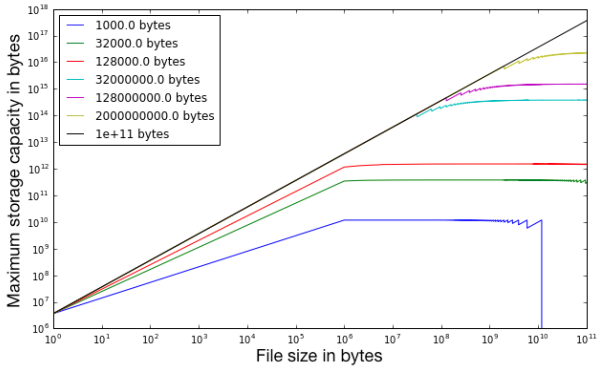


Fig. 4: Capacity vs. File Size (Varying Block Size)

This graph shows how larger block size does indeed lead to larger system capacity. In fact the system has the highest capacity if each block is as large as

memory of the NameNode, but such large blocks are impractical, as explained below. Another interesting point is that for the smallest block size, we see that at a certain point a single file that creates too many blocks will not fit in the system. Thus, the system has capacity 0. This happens since the metadata to hold all of the blocks for a single file of that size would not fit in memory on the NameNode.

Thus, we expect this memory limit to be affected in the following main ways. First, larger files will give the system a larger overall capacity. Second, larger blocks will give the system larger overall capacity, assuming the limiting factor is memory on the NameNode and not disk on the DataNodes (since we can always add more DataNode machines).

Although large blocks produce the smallest amount of metadata, increasing the block size poses a tradeoff. There are two primary concerns with a very large block size. First, a larger block size causes the contents of each file to be less equally distributed throughout the cluster. This potentially reduces the efficiency of streaming reads, as file operations are less distributed across the DataNodes. Second, during a MapReduce job, each map task is run on a single block (the FileSplit factor is set to the size of a block) [2]. As a result, larger blocks make MapReduce less distributed and hence less efficient. In practice HDFS has a large default block size of 128 MB [2].

3.2 NameNode CPU Bottleneck

The less obvious scalability problem with the HDFS architecture is the potential for a CPU bottleneck at the NameNode. The system is built on the assumption that metadata operations are inexpensive relative to network communication and file I/O. The HDFS architecture also assumes that there will be few enough large files that the number of such metadata operations (including requests for block locations) will be insignificant. However, with many small files, these assumptions are violated and the NameNode CPU could become a bottleneck.

The first potential source of CPU bottleneck is the internal load of communication with the DataNodes. This internal load can come about in two ways. First, internal load can be created when the system holds a large amount of small files. The size of the block report sent by each DataNode is directly proportional to the number of blocks that the DataNode holds. Since each file is given its own virtual block even if the entire file is much smaller than a block, small files allow each DataNode to hold a larger number of blocks. That is, small files increase the block density. As a result, each DataNode must send a larger block report to the NameNode with each heartbeat. The processing of these block reports could potentially overload the NameNode CPU and increase the latency of requests to the NameNode.

Another potential source of CPU bottleneck is external load from clients querying the NameNode. As explained above, for each read or write of a block, a client must query the NameNode to obtain the list of DataNodes where the block is replicated. The NameNode must then perform a metadata operation to determine what to return to the client. Therefore, the NameNode can become a bottleneck if there are many clients trying to perform operations concurrently since they all must access the metadata via the singular NameNode. This should be an especially prevalent problem when there are many small files because it would be likely that the use pattern of a system with many small files would be for there to be many clients each trying to access their files concurrently. Moreover, DataNode operations should return relatively quickly since each file operation is on a small amount of data. This I/O speedup may be mitigated by more disk seeks, due to the fact that I/O with small files is less sequential.

With these potential CPU bottlenecks in mind, we pose the question of whether it is reasonable to generate an appropriate load so as to reach these CPU bottlenecks before the NameNode runs out of memory. In our experiments we determine whether the CPU bottleneck is ever the dominant scalability limit for HDFS. We restrict our experiments to valid requests and simulate a plausible usage pattern for small files (*e.g.,* each file requested must exist and there are no "hot" files).

4 Performance Evaluation

In this section we present our experimental results on the memory bottleneck and the CPU bottlenecks of HDFS. We evaluate HDFS 2.7.1 on a cluster consisting of four identical Amazon EC2 instances. Each instance has 8 GB of disk space, 2 GB of RAM, and a 3.3 GHz Intel Xeon processor. Three of these instances are dedicated as DataNodes, and one instance is the NameNode. Although this is a small amount of memory for a typical HDFS NameNode, the goals

of our experiments are to analyze the performance of HDFS around the limits of memory. Thus we are concerned with the amount of metadata *relative* to the capacity of the NameNode's memory, rather than the absolute amount of metadata. Our experimental results can be normalized to account for the memory capacity of the NameNode and the disk capacity of the DataNodes in our cluster.

We query HDFS using the Hadoop Java API. On top of the file system class provided by the API, we implemented a class called HDFSClient. Each HDFSClient object is runnable, and represents a single client making file access or modification requests to the file system. On each machine querying the NameNode, we spin up a pool of threads, each of which wraps an HDFSClient and reads from a global thread-safe *request queue*. The master thread (different in each experiment) generates requests and places them on the request queue for processing by the HDFSClient worker threads. Each client repeatedly removes a request from the queue, issues that request to the NameNode and waits for the subsequent response. Examples of requests include writing a file from the local file system to HDFS, reading a file from HDFS, and deleting a file from HDFS.

4.1 Memory Bottleneck Experiments

In this section we explore the memory bottleneck of HDFS. First we provide empirical results on the metadata size, and then we precisely determine when the NameNode will run out of memory.

4.1.1 Experiment 1: Metadata Size

The first goal of our experiments was to determine precisely how much memory is occupied on the NameNode for each file added to the system. Analysis of the source code for HDFS and theoretical results in the literature suggest that each file should yield roughly 150 bytes of inode plus 150 bytes for each block to account for the mapping [1, 5]. However, the peculiarities of the NameNode storage for recovery suggest that these numbers are not precisely correct. First, these numbers are based on an analysis of the source code rather than empirical results. It is possible that the memory footprint of each file extends beyond explicit storage of the (inode, mapping) pair. Second, the NameNode stores the FSImage, a snapshot of the file system, as well as an Edits log that indicates the updates since the last snapshot. Backups of each file are also maintained. Hence it is possible that as more files are added to the system, each additional file will result in more occupied memory on the NameNode. Moreover we seek to verify that the mapping for each block is fixed-size, whether or not the entire block is occupied by file contents.

To test the size of metadata, we create a pool of HDFSClient threads and add large batches of files to HDFS (cf. MetadataSizeTest.java). We do this for files of varying size and measure the resulting increase in occupied memory on the NameNode. We find that each file accounts for a fixed 352 bytes of memory footprint, plus 175 bytes per block. In our analytical model above, this corresponds to values of $i = 352$ and $c = 175$. This is a significantly larger footprint than reported in [5]. However, our data does support the result that each block mapping is of fixed size and each additional block accounts for the same amount of additional metadata. That is, the metadata footprint of a file on HDFS is linear in the number of blocks occupied by that file.

4.1.2 Experiment 2: Memory Bottleneck

Next we pushed our HDFS cluster to the memory bottleneck. This was achieved by continually adding small files (10 bytes each) to the HDFS cluster until the write requests were refused. We found that our 2 GB NameNode could support roughly 234,000 10-byte files, accounting for approximately 123.3 MB of metadata. This discrepancy between 2 GB and 123.3 MB exists because the HDFS NameNode must allocate memory to the Java Virtual Machine and the operating system. Thus the metadata storage was restricted to 128 MB of heap memory. This limit can be changed in the HDFS configuration and was set to its maximum value in our experiments given the size of our NameNode memory.

We note that once the memory bottleneck is reached, HDFS does not refuse more files but rather the JVM fails on basic operations. For example, a query of the root directory (`hdfs dfs -ls /`) fails due to shortage of memory. So the file system is rendered inaccessible unless files are removed.

4.2 CPU Bottleneck Experiments

In this section we test the throughput of file operations as more files are added to HDFS. We analyze different request patterns to determine the performance bottleneck. We note that the small size of our cluster makes

our results on throughput less convincing. That is, because there are only three DataNodes for the NameNode to monitor, there is relatively little block report processing on the NameNode CPU. Nonetheless, we can still add a large number of blocks and analyze throughput of HDFS near the maximum capacity of our cluster.

4.2.1 Experiment 3: Read Throughput

We next tested the throughput of HDFS reads as more small files were added to the system. One might expect throughput to decrease as more files are added due to the increasing complexity of block mappings, as well as a wider range of disk seeks on the DataNodes. In this experiment (found in ThroughputTest.java), there is a pool of 64 HDFSClient workers on three different machines. The workers go through two stages: First we create a pool of worker threads and measure throughput while reading roughly 1000 random files from HDFS to the local file system. In the second stage, new workers add 25,000 files to the system. The worker threads are killed and the throughput is again measured with 25,000 additional files on the system. This was repeated at increments of 25,000 files until the memory bottleneck was reached at roughly 234,000 files.

We found that there was no decrease in throughput as more files were added. As shown in Figure 5, the read throughput hovered around 100 files per second. This supports the HDFS design assumption that metadata operations are inexpensive relative to other file system and network operations of HDFS. Moreover, it does not support the hypothesis that disk seeks on random file access are more expensive when more files were added.

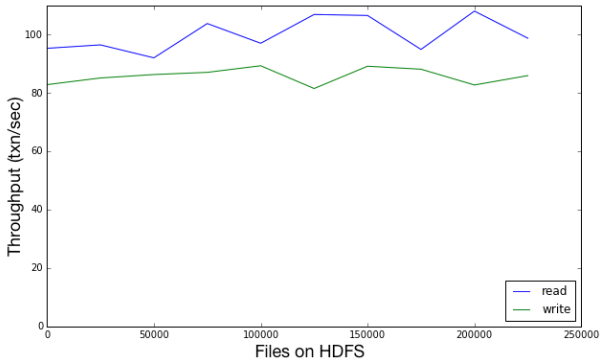


Fig. 5: Isolated Read/Isolated Write Throughput

4.2.2 Experiment 4: Write Throughput

Although read throughput was not affected by the number of files stored on the cluster, we posited that write throughput may still be affected. While reads need only query a single DataNode, writes must be replicated down a pipeline of R DataNodes, where R is the replication factor (here $R = 3$). Hence write operations involve all DataNodes in our cluster. Each DataNode must perform a disk seek to a free location in physical memory, and HDFS cannot select the least busy DataNode to service a request.

The design of the write throughput experiments mirrored the design of the read throughput test. As seen in Figure 5 above, our experiments showed no decrease in write throughput as more files were added to HDFS. The write throughput hovered around 80 files per second over the entire range of 0 to 225,000 files on the cluster. Even though batch writes may be sequential, one would expect writes to be slower than reads on HDFS. This is because writes must be replicated across R DataNodes, and the pipeline write process involves R TCP connections to be setup and torn down.

4.2.3 Experiment 5: Mixed Reads and Writes

A more typical usage pattern for HDFS would involve interleaved reads and writes. Therefore we ran experiments measuring throughput of HDFS mixed reads and writes as more files are added to the system. We note that it is possible that batch writes allow each DataNode in the pipeline to avoid disk seeks between writes. That is, assuming there is a large contiguous segment of free disk space available, the DataNode can write each new file sequentially without disk seeks. Thus Experiment 4 above would not show any decrease in throughput due to disk seeks. However, mixed (random) reads and writes would force each DataNode to seek on disk, so one might still expect to see performance degenerate as more small files are added.

The results of our experiments showed that there is no throughput degeneration as the number of files on HDFS is increased. As shown in the following figure, mixed reads and writes (at a read-to-write ratio of 4:1) produce approximately a weighted average of the throughout.

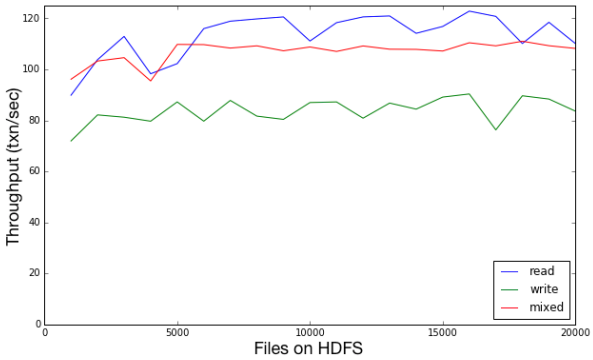


Fig. 6: Mixed Read/Write Throughput vs. Files

4.2.4 Experiment 6: Frequent Requests

A dataset consisting of a large number of small files would likely receive a higher frequency of queries than another dataset. A typical client will read a file, process that file's data, then request the next file. Small files will be processed at the client more quickly than large files, therefore one would expect each client to have a higher frequency of requests. Since each request must first query the NameNode for a block mapping, this will put a large strain on the NameNode. In this section we explore the possibility that this usage pattern associated with small files results in a bottleneck.

Our experimental setup was as follows. Again we used HDFSClient threads querying from multiple machines. We uploaded a large set of roughly 25,000 10-byte files to HDFS. The HDFSClients then randomly read these files to their local file system. We measured the throughput of these requests as the number of clients varied from a single connection thread and moving up to 64 HDFSClients. Each client made requests as quickly as possible, and we looked for a decrease in throughput as the NameNode received more concurrent read and write queries. This produced the following graph.

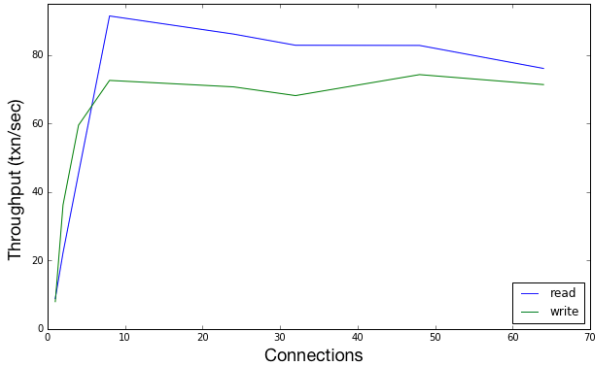


Fig. 7: Read Throughput vs. Request Frequency

We see that there is an initial spike in throughput as the utilization of the DataNodes increases to full capacity. In the case of reads, there is a slight decrease in the number of connections beyond 8. Importantly we do not observe a sharp decline in throughput. For writes, we see no decline whatsoever.

These results show only that the scalability of reads and writes is not linear on our cluster. That is, there is some bottleneck achieved around 8 connections. However, we expected that this bottleneck is not due to NameNode processing but rather in disk I/O and network communication between the DataNodes. We also proposed that this might be a limitation of the sub-optimal networking of our cluster. To test this hypothesis, we ran the same test but only requested the block *locations*.

By requesting block locations rather than reads/writes, we restricted our request to touching only the NameNode and clients. In these block location request trials, we observed the same throughput behavior as above. However, we were not convinced that this indicates the NameNode CPU is a bottleneck for requests. This is because we monitored the CPU load at the NameNode and saw very low (below 10% peak) CPU usage during the test. Therefore these results are likely attributed to confounding factors in our experimental setup.

4.2.5 Experiment 7: Internal Load

To see the internal load created by the heartbeats, we checked the CPU load on an idle system with different amounts of files on the system. For our very small system, we were not able to put enough files on the servers to notice any uptick in CPU load from the heartbeats. Essentially, the limit of the memory of the NameNode dominated this potential bottleneck so thoroughly that we were not able to even see any substantive change in CPU load when the memory limit was reached. This could be because our system had only three DataNodes, and thus received only three heartbeats. We saw about a .3% CPU usage from each node, so with thousands of nodes this could become a bottleneck, but not in our system.

5 Improvements to HDFS

Some improvements have been suggested to solve the scalability problems with HDFS. One attempt to address the memory bottleneck is Apache's HDFS Fed-

eration. This system modifies HDFS by creating multiple NameNodes each of which has its own separate namespace [4]. This is not a fundamental solution to the architectural problem, but rather merely adds more hardware to increase the memory bottleneck. Thus HDFS Federation is still a potentially inefficient use of memory on the NameNodes given a dataset of many small files. Additionally, with multiple independent namespaces the system cannot perform operations across the namespaces. Thus, the complexity of cross-partition transactions must be handled by the client application. The client can use symbolic links and client-side mount tables to make the multiple namespaces transparent to the user, so long as the client can tolerate higher request latency.

Some other distributed file systems like CalvinFS have been designed to more fundamentally address the memory bottleneck [6]. CalvinFS treats file metadata similar to any other data in a distributed database system. It transforms operations on file metadata into distributed transactions as outlined by the Calvin database system. In this way, CalvinFS supports scalable metadata management while ensuring durability and providing standard file system operations. However, many users would prefer to have an open-source solution.

We now offer a potential improvement for future exploration. We consider the case when a potential application will perform large batch reads and writes of many small files with infrequent updates to files. In this case, an intermediate overlay could be implemented between HDFS and the client application. To perform a batch write operation, the overlay would aggregate many small files into one large file and store a mapping of each small file into its constituent large block. This large file of files would then be passed to HDFS for writing. On batch reads, the overlay would map the requested files to the large file, request from HDFS, and then return the desired files. Such a scheme would only be appropriate in cases where random file access and file edits are infrequent.

Finally, there are multiple modifications that could be made to HDFS to reduce the metadata produced by each file. For example, one could remove all permissions from the inodes. We explored this possibility in the source code, but chose not to pursue this idea for two reasons. First, even very small changes to the HDFS source code invalidate many other source files. This is because HDFS checks permissions in a wide array of components, *i.e.,* permissions are a dependency of many files. Second, we determined that permissions (and many other UNIX-inspired features) have a very small footprint in HDFS (namely, each file stores an 8-byte long to represent permissions).

A more fruitful approach may be to simplify the scheme for mappings to reduce the amount of metadata needed to map each block to a list of DataNodes. However, this is least feasible for small files, since there is a larger potential domain of distinct blocks to map to DataNodes. That is, smaller and more numerous files imply that each mapping must potentially represent more distinct blocks and DataNode locations. Thus more bits will be needed even by an optimally efficient scheme.

6 Conclusion

This paper has presented and experimented on the scalability limits of HDFS for storing small files. We presented an analytical model of the NameNode memory bottleneck, and we provided a precise measurement of the memory footprint of each file on the NameNode. We confirmed that the single-NameNode architecture is indeed a bottleneck of HDFS. While this result had been previously presented in [5] and elsewhere, we showed that the NameNode's memory limitations dominate potential CPU bottlenecks in all reasonable usage patterns. We showed that read-only, write-only, and mixed read-write workloads are unaffected by the number of files stored on an HDFS cluster. This suggests that there is negligible CPU load on the NameNode caused by increased block report complexity and a greater metadata space. Finally we showed that even a high frequency of requests for small files does not cause the NameNode CPU to be a bottleneck. Rather, the NameNode memory bottleneck is the dominant bottleneck in all usage patterns.

References

[1] Robert Chansler, Hairong Kuang, Sanjay Radia, Konstantin Shvachko, and Suresh Srinivas. The hadoop distributed file system. 2016. [Online; accessed 3-May-2016].

[2] Apache Foundation. Hadoop distributed file system default settings. 2016. [Online; accessed 3-May-2016].

[3] Apache Foundation. HDFS architecture guide. 2016. [Online; accessed 3-May-2016].

[4] Sanjay Radia and Suresh Srinivas. Scaling HDFS cluster using namenode federation. 2010. [Online; accessed 3-May-2016].

[5] Konstantin Shvachko. HDFS scalability: The limits to growth. 2010. [Online; accessed 3-May-2016].

[6] Alexander Thomson and Daniel J. Abadi. CalvinFS: Consistent WAN replication and scalable metadata management for distributed file systems. 2015. [Online; accessed 3-May-2016].