KONSTANTIN V. SHVACHKO

HDFS scalability: the limits to growth

Konstantin V. Shvachko is a principal software engineer at Yahoo!, where he develops HDFS. He specializes in efficient data structures and algorithms for large-scale distributed storage systems. He discovered a new type of balanced trees, S-trees, for optimal indexing of unstructured data, and he was a primary developer of an S-tree-based Linux file system, treeFS, a prototype of reiserFS. Konstantin holds a Ph.D. in computer science from Moscow State University, Russia. He is also a member of the Project Management Committee for Apache Hadoop.

shv@yahoo-inc.com

THE HADOOP DISTRIBUTED FILE SYSTEM (HDFS) is an open source system currently being used in situations where massive amounts of data need to be processed. Based on experience with the largest deployment of HDFS, I provide an analysis of how the amount of RAM of a single namespace server correlates with the storage capacity of Hadoop clusters, outline the advantages of the single-node namespace server architecture for linear performance scaling, and establish practical limits of growth for this architecture. This study may be applicable to issues with other distributed file systems.

By software evolution standards Hadoop is a young project. In 2005, inspired by two Google papers, Doug Cutting and Mike Cafarella implemented the core of Hadoop. Its wide acceptance and growth started in 2006 when Yahoo! began investing in its development and committed to use Hadoop as its internal distributed platform. During the past several years Hadoop installations have grown from a handful of nodes to thousands. It is now used in many organizations around the world.

In 2006, when the buzzword for storage was Exabyte, the Hadoop group at Yahoo! formulated long-term target requirements [7] for the Hadoop Distributed File System and outlined a list of projects intended to bring the requirements to life. What was clear then has now become a reality: the need for large distributed storage systems backed by distributed computational frameworks like Hadoop MapReduce is imminent.

Today, when we are on the verge of the Zettabyte Era, it is time to take a retrospective view of the targets and analyze what has been achieved, how aggressive our views on the evolution and needs of the storage world have been, how the achievements compare to competing systems, and what our limits to growth may be.

The main four-dimensional *scale requirement targets for HDFS* were formulated [7] as follows:

```
10PB capacity x 10,000 nodes x
100,000,000 files x 100,000 clients
```

The biggest Hadoop clusters [8, 5], such as the one recently used at Yahoo! to set sorting records, consist of 4000 nodes and have a total space capacity of 14PB each. Many production clusters run on 3000 nodes with 9PB storage capacities.

Hadoop clusters have been observed handling more than 100 million objects maintained by a single namespace server with a total capacity of 100 million files.

Four thousand node clusters successfully ran jobs with a total of more than 14,000 tasks reading from or writing to HDFS simultaneously.

Table 1 compares the targets with the current achievements:

	Target	**Deployed**
Capacity	10PB	14PB
Nodes	10,000	4000
Clients	100,000	15,000
Files	100,000,000	60,000,000

TABLE 1: TARGETS FOR HDFS VS. ACTUALLY DEPLOYED VALUES AS OF 2009

The bottom line is that we achieved the target in petabytes and got close to the target in the number of files, but this is done with a smaller number of nodes, and the need to support a workload close to 100,000 clients has not yet materialized.

The question is now whether the goals are feasible with the current system architecture. And the main concern is the *single namespace server architecture*. This article studies scalability and performance limitations imposed on HDFS by this architecture.

The methods developed in this work could be useful or applicable to other distributed systems with similar architecture.

The study is based on experience with today's largest deployments of Hadoop. The performance benchmarks were run on real clusters, and the storage capacity estimates were verified by extrapolating measurements taken from production systems.

HDFS at a Glance

Being a part of Hadoop core and serving as a storage layer for the Hadoop MapReduce framework, HDFS is also a stand-alone distributed file system like Lustre, GFS, PVFS, Panasas, GPFS, Ceph, and others. HDFS is optimized for batch processing focusing on the overall system throughput rather than individual operation latency.

As with most contemporary distributed file systems, HDFS is based on an architecture with the namespace decoupled from the data. The namespace forms the file system metadata, which is maintained by a dedicated server called the *name-node*. The data itself resides on other servers called *data-nodes*.

The file system data is accessed via *HDFS clients*, which first contact the name-node for data location and then transfer data to (write) or from (read) the specified data-nodes (see Figure 1).

The main motivation for decoupling the namespace from the data is the scalability of the system. Metadata operations are usually fast, whereas data transfers can last a long time. If a combined operation is passed through a single server (as in NFS), the data transfer component dominates the response time of the server, making it a bottleneck in a highly distributed environment.

In the decoupled architecture, fast metadata operations from multiple clients are addressed to the (usually single) namespace server, and the data transfers are distributed among the data servers utilizing the throughput of the whole cluster.

The namespace consists of files and directories. Directories define the hierarchical structure of the namespace. Files—the data containers—are divided into large (128MB each) blocks.

The name-node's metadata consist of the hierarchical namespace and a block to data-node mapping, which determines physical block locations.

In order to keep the rate of metadata operations high, HDFS keeps the whole namespace in RAM. The name-node persistently stores the namespace *image* and its modification log (the *journal*) in external memory such as a local or a remote hard drive.

The namespace image and the journal contain the HDFS file and directory names and their attributes (modification and access times, permissions, quotas), including block IDs for files, but not the locations of the blocks. The locations are reported by the data-nodes via block reports during startup and then periodically updated once an hour by default.

If the name-node fails, its latest state can be restored by reading the namespace image and replaying the journal.

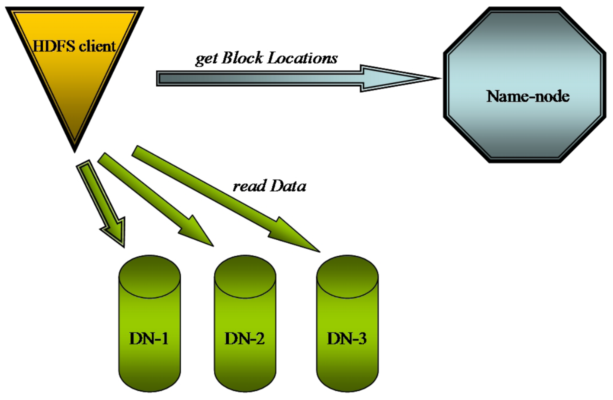

FIGURE 1: AN HDFS READ REQUEST STARTS WITH THE CLIENT MAKING A REQUEST TO THE NAME-NODE USING A FILE PATH, GETTING PHYSICAL BLOCK LOCATIONS, AND THEN ACCESSING DATA-NODES FOR THOSE BLOCKS.

Namespace Limitations

HDFS is built upon the single-node namespace server architecture.

Since the name-node is a single container of the file system metadata, it naturally becomes a limiting factor for file system growth. In order to make metadata operations fast, the name-node loads the whole namespace into its memory, and therefore the size of the namespace is limited by the amount of RAM available to the name-node.

Estimates show [12] that the name-node uses fewer than 200 bytes to store a single metadata object (a file inode or a block). According to statistics on our clusters, a file on average consists of 1.5 blocks, which means that it takes 600 bytes (1 file object + 2 block objects) to store an average file in name-node's RAM. This estimate does not include transient data structures, which the name-node creates for replicating or deleting blocks, etc., removing them when finished.

CONCLUSION 1

If

- $objSize$ is the size of a metadata object,
- λ is the average file to block ratio, and
- F is the total number of files,

then the memory footprint of the namespace server will be at least

$$RAM \geq F \cdot \lceil 1 + \lambda \rceil \cdot objSize$$

Particularly, in order to store 100 million files (referencing 200 million blocks) a name-node should have at least 60GB ($10^8 \cdot 600$) of RAM. This matches observations on deployed clusters.

Replication

HDFS is designed to run on highly unreliable hardware. On Yahoo's long-running clusters we observe a node failure rate of 2–3 per 1000 nodes a day. On new (recently out of the factory) nodes, the rate is three times higher.

In order to provide data reliability HDFS uses block replication. Initially, each block is replicated by the client to three data-nodes. The block copies are called *replicas*. A replication factor of three is the default system parameter, which can either be configured or specified per file at creation time.

Once the block is created, its replication is maintained by the system automatically. The name-node detects failed data-nodes, or missing or corrupted individual replicas, and restores their replication by directing the copying of the remaining replicas to other nodes.

Replication is the simplest of known data-recovery techniques. Other techniques, such as redundant block striping or erasure codes, are applicable and have been used in other distributed file systems such as GPFS, PVFS, Lustre, and Panasas [1, 3, 6, 10]. These approaches, although more space efficient, also involve performance tradeoffs for data recovery. With striping, depending on the redundancy requirements, the system may need to read two or more of the remaining data segments from the nodes it has been striped to in order to reconstruct the missing one. Replication always needs only one copy.

For HDFS, the most important advantage of the replication technique is that it provides high availability of data in high demand. This is actively exploited by the MapReduce framework, as it increases replications of configuration and job library files to avoid contention during the job startup, when multiple tasks access the same files simultaneously.

Each block replica on a data-node is represented by a local (native file system) file. The size of this file equals the actual length of the block and does not require extra space to round it up to the maximum block size, as traditional file systems do. Thus, if a block is half full it needs only half of the space of the full block on the local drive. A slight overhead is added, since HDFS also stores a second, smaller metadata file for each block replica, which contains the checksums for the block data.

Replication is important both from reliability and availability points of view, and the default replication value of 3 seem to be reasonable in most cases for large, busy clusters.

STORAGE CAPACITY VS. NAMESPACE SIZE

With 100 million files each having an average of 1.5 blocks, we will have 200 million blocks in the file system. If the maximal block size is 128MB and every block is replicated three times, then the total disk space required to store these blocks is close to 60PB.

CONCLUSION 2

If

- $blockSize$ is the maximal block size,
- r is the average block replication,
- λ is the average file-to-block ratio, and
- F is the total number of files,

then the storage capacity (SC) referenced by the namespace will not exceed

$$SC \leq F \cdot \lambda \cdot r \cdot blockSize$$

Comparison of Conclusions 1 and 2 leads us to the following rule.

RULE 1

As a rule of thumb, the correlation between the representation of the metadata in RAM and physical storage space required to store data referenced by this namespace is:

1GB metadata ≈ 1PB physical storage

The rule should not be treated the same as, say, the Pythagorean Theorem, because the correlation depends on cluster parameters, the block-to-file ratio, and the block size, but it can be used as a practical estimate for configuring cluster resources.

CLUSTER SIZE AND NODE RESOURCES

Using Conclusion 2, we can estimate the number of data-nodes the cluster should have in order to accommodate namespace of a certain size.

On Yahoo's clusters, data-nodes are usually equipped with four disk drives of size 0.75–1TB, and configured to use 2.5–3.5TB of that space per node. The remaining space is allocated for MapReduce transient data, system logs, and the OS.

If we assume that an average data-node capacity is 3TB, then we will need on the order of 20,000 nodes to store 60PB of data. To be consistent with the target requirement of 10,000 nodes, each data-node should be configured with eight hard drives.

CONCLUSION 3

In order to accommodate data referenced by a 100 million file namespace, an HDFS cluster needs 10,000 nodes equipped with eight 1TB hard drives. The total storage capacity of such a cluster is 60PB.

Note that these estimates are true under the assumption that the block-per-file ratio of 1.5 and the block size remain the same. If the ratio or the block size increases, a gigabyte of RAM will support more petabytes of physical storage, and vice versa.

Sadly, based on practical observations, the *block-to-file ratio tends to decrease* during the lifetime of a file system, meaning that the object count (and therefore the memory footprint) of a single namespace server grows faster than the physical data storage. That makes the *object-count problem*, which becomes a *file-count problem* when $\lambda \to 1$, the real bottleneck for cluster scalability.

BLOCK REPORTS, HEARTBEATS

The name-node maintains a list of registered data-nodes and blocks belonging to each data-node.

A data-node identifies block replicas in its possession to the name-node by sending a *block report*. A block report contains *block ID*, *length,* and the *generation stamp* for each block replica.

The first block report is sent immediately after the data-node registration. It reveals block locations, which are not maintained in the namespace image or in the journal on the name-node. Subsequently, block reports are sent periodically every hour by default and serve as a sanity check, providing that the name-node has an up-to-date view of block replica distribution on the cluster.

During normal operation, data-nodes periodically send *heartbeats* to the name-node to indicate that the data-node is alive. The default heartbeat interval is three seconds. If the name-node does not receive a heartbeat from a data-node in 10 minutes, it pronounces the data-node dead and schedules its blocks for replication on other nodes.

Heartbeats also carry information about total and used disk capacity and the number of data transfers currently performed by the node, which plays an important role in the name-node's space and load-balancing decisions.

The communication on HDFS clusters is organized in such a way that the name-node does not call data-nodes directly. It uses heartbeats to reply to the data-nodes with important instructions. The instructions include commands to:

- Replicate blocks to other nodes
- Remove local block replicas
- Re-register or shut down the node
- Send an urgent block report

These commands are important for maintaining the overall system integrity; it is therefore imperative to keep heartbeats frequent even on big clusters. The name-node is optimized to process thousands of heartbeats per second without affecting other name-node operations.

THE INTERNAL LOAD

The block reports and heartbeats form the *internal load of the cluster*. This load mostly depends on the number of data-nodes. If the internal load is too high, the cluster becomes dysfunctional, able to process only a few, if any, external client operations such as `ls`, `read`, or `write`.

This section analyzes what percentage of the total processing power of the name-node is dedicated to the internal load.

Let's assume the cluster is built of 10,000 data-nodes having eight hard drives with 6TB of effective storage capacity each. This is what it takes, as we learned in previous sections, to conform to the targeted requirements.

As usual, our analysis is based on the assumption that the block-to-file ratio is 1.5.

The ratio particularly means that every other block on a data-node is half full. If we group data-node blocks into pairs having one full block and one half-full block, then each pair will occupy approximately 200 MB ≈ 128 MB + 64 MB on a hard drive. This gives us an estimate that a 6 TB (8 HD x 0.75 TB) node will hold 60,000 blocks. This is the size of an average block report sent by a data-node to the name-node.

The sending of block reports is randomized so that they do not come to the name-node together or in large waves. Thus, *the average number of block reports the name-node receives is* 10,000/hour, which is *about three reports per second*.

The heartbeats are not explicitly randomized by the current implementation and, in theory, can hit the name-node together, although the likelihood of this is very low. Nevertheless, let's assume that the name-node should be able to handle 10,000 heartbeats per second on a 10,000 node cluster.

In order to measure the name-node performance, I implemented a benchmark called NNThroughputBenchmark, which now is a standard part of the HDFS code base.

NNThroughputBenchmark is a single-node benchmark, which starts a name-node and runs a series of client threads on the same node. Each client repetitively performs the same name-node operation by directly calling the name-node method implementing this operation. Then the benchmark measures the number of operations performed by the name-node per second.

The reason for running clients locally rather than remotely from different nodes is to avoid any communication overhead caused by RPC connections and serialization, and thus reveal the upper bound of pure name-node performance.

The following numbers were obtained by running NNThroughputBenchmark on a node with two quad-core Xeon CPUs, 32GB RAM, and four 1TB hard drives.

Table 2 summarizes the name-node throughput with respect to the two internal operations. Note that the block report throughput is measured in the number of blocks processed by the name-node per second.

	Throughput
Number of blocks processed in block reports per second	639,713
Number of heartbeats per second	300,000

TABLE 2: BLOCK REPORT AND HEARTBEAT THROUGHPUT

We see that the name-node is able to process more than 10 reports per second, each consisting of 60,000 blocks. As we need to process only three reports per second, we may conclude that less than 30% of the name-node's total processing capacity will be used for handling block reports.

The heartbeat load is 3.3%, so that the combined internal load of block reports and heartbeats is still less than 30%.

CONCLUSION 4

> The internal load for block reports and heartbeat processing on a 10,000-node HDFS cluster with a total storage capacity of 60 PB will consume 30% of the total name-node processing capacity.

Thus, the internal cluster load directly depends on the average block report size and the number of the reports. The impact of heartbeats is negligible.

Another way to say this is that the internal load is proportional to the number of nodes in the cluster and the average number of blocks on a node. Thus, if a node had only 30,000 blocks, half of the estimated amount, then the name-node would dedicate only 15% of its processing resources to the internal load, because the nodes would send the same number of block reports but the size of the block reports would be smaller by a half compared to the original estimate.

Conversely, if the average number of blocks per node grows, then the internal load will grow proportionally. In particular, it means the decrease in block-to-file ratio (more small files with the same file system size) increases the internal load and therefore negatively affects the performance of the system.

REASONABLE LOAD EXPECTATIONS

The good news from the previous section is that the name-node can still use 70% of its time to process *external client requests*. If all the clients started sending arbitrary requests to the name-node with very high frequency, the name-node most probably would have a hard time coping with the load and would become unresponsive, potentially sending the whole cluster into a tailspin, because internal load requests do not have priority over regular client requests. But this can happen even on smaller clusters with extreme load levels.

The goal of this section is to determine *reasonable load expectations* on a large cluster (10,000 nodes, 60PB of data) and estimate whether the name-node would be able to handle it.

Regular Hadoop clusters run MapReduce jobs. We first assume that all our 100,000 clients running different tasks provide *read-only load* on the HDFS cluster. This is typical for the map stage of a job execution.

Usually a map task produces map output, which is written to a local hard drive. Since MapReduce servers (task-trackers) share nodes with HDFS data-nodes, map output inevitably competes with HDFS reads. This reduces the HDFS read throughput, but also decreases the load on the name-node. Thus, for the sake of this analysis we may assume that our *tasks do not produce any output*, because otherwise the load on the name-node would be lower.

Typically, a map task reads one block of data. In our case, files consist of 1.5 blocks. Thus an average client reads a chunk of data of size 96MB (1.5 * 128MB/2) and we may assume that *the size of a read operation per client is 96MB*.

Figure 1 illustrates that client reads conceptually consist of two stages:

1. Get block locations from the name-node.
2. Pull data (block replica) from the nearest data-node.

We will estimate how much time it takes for a client to retrieve a block replica and, based on that, derive how many "get block location" requests the name-node should expect per second from 100,000 clients.

DFSIO was one of the first standard benchmarks for HDFS. The benchmark is a map-reduce job with multiple mappers and a single reducer. Each mapper writes (reads) bytes to (from) a distinct file. Mappers within the job either all write or all read, and each mapper transfers the same amount of data. The mappers collect the I/O stats and pass them to the reducer. The reducer averages them and summarizes the I/O throughput for the job. The key measurement here is the byte transfer rate of an average mapper.

The following numbers were obtained on a 4000-node cluster [8] where the name-node configuration is the same as in NNThroughputBenchmark and data-nodes differ from the name-node only in that they have 8GB RAM. The cluster consists of 100 racks with 1 gigabit Ethernet inside a rack and 4 gigabit uplink from rack.

Table 3 summarizes the average client read and write throughput provided by DFSIO benchmark.

	Throughput
Average read throughput	66 MB/s
Average write throughput	40 MB/s

TABLE 3: HDFS READ AND WRITE THROUGHPUT

We see that an average client will read 96MB in 1.45 seconds. According to our assumptions, it will then go to the name-node to get block locations for another chunk of data or a file. Thus, 100,000 clients will produce 68,750 get-block-location requests to the name-node per second.

Another series of throughput results [11] produced by NNThroughputBenchmark (Table 4) measures the number of "open" (the same as "get block location") and "create" operations processed by the name-node per second:

	Throughput
Get block locations	126,119 ops/s
Create new block	5,600 ops/s

TABLE 4: OPEN AND CREATE THROUGHPUT

This shows that with the internal load at 30% the name-node will be able to process more than 88,000 get-block-location operations, which is enough to handle the read load of 68,750 ops/sec.

CONCLUSION 5

A 10,000-node HDFS cluster with internal load at 30% will be able to handle an expected read-only load produced by 100,000 HDFS clients.

The write performance looks less optimistic. For writes we consider a different distcp-like job load, which produces a lot of writes. As above, we assume that an average write size per client is 96MB. According to Table 3, an average client will write 96MB in 2.4 seconds. This provides an average load of 41,667 create-block requests per second from 100,000 clients, and this is way above 3,920 creates per second—70% of the possible processing capacity of the name-node (see Table 4). Furthermore, this does not yet take into account the 125,000 confirmations (three per block-create) sent by data-nodes to the name-node for each successfully received block replica.

Although these confirmations are not as heavy as create-blocks, this is still a substantial additional load.

Even at 100% processing capacity dedicated to external tasks (no internal load), the clients will not be able to run at “full speed” with writes. They will experience substantial idle cycles waiting for replies from the name-node.

CONCLUSION 6

> A reasonably expected write-only load produced by 100,000 HDFS clients on a 10,000-node HDFS cluster will exceed the throughput capacity of a single name-node.

Distributed systems are designed with the expectation of linear performance scaling: more workers should be able to produce a proportionately larger amount of work. The estimates above (working the math backwards) show that 10,000 clients can saturate the name-node for write-dominated workloads. On a 10,000-node cluster this is only one client per node, while current Hadoop clusters are set up to run up to four clients per node. This makes the single name-node a bottleneck for linear performance scaling of the entire cluster. There is no benefit in increasing the number of writers. A smaller number of clients will be able to write the same amount of bytes in the same time.

Final Notes

We have seen that a 10,000 node HDFS cluster with a single name-node is expected to handle well a workload of 100,000 readers, but even 10,000 writers can produce enough workload to saturate the name-node, making it a bottleneck for linear scaling.

Such a large difference in performance is attributed to get block locations (read workload) being a memory-only operation, while creates (write workload) require journaling, which is bounded by the local hard drive performance.

There are ways to improve the single name-node performance, but any solution intended for single namespace server optimization lacks scalability.

Looking into the future, especially taking into account that the ratio of small files tends to grow, the most promising solutions seem to be based on distributing the namespace server itself both for workload balancing and for reducing the single server memory footprint. There are just a few distributed file systems that implement such an approach.

Ceph [9] has a cluster of namespace servers (MDS) and uses a dynamic subtree partitioning algorithm in order to map the namespace tree to MDSes evenly. [9] reports experiments with 128 MDS nodes in the entire cluster consisting of 430 nodes. Per-MDS throughput drops 50% as the MDS cluster grows to 128 nodes.

Google recently announced [4] that GFS [2] has evolved into a distributed namespace server system. The new GFS can have hundreds of namespace servers (masters) with 100 million files per master. Each file is split into much smaller size than before (1 vs. 64 MB) blocks. The details of the design, the scalability, and performance facts are not yet known to the wider community.

Lustre [3] has an implementation of clustered namespace on its roadmap for the Lustre 2.2 release. The intent is to stripe a directory over multiple metadata servers (MDS), each of which contains a disjoint portion of the namespace. A file is assigned to a particular MDS using a hash function on the file name.

ACKNOWLEDGMENTS

I would like to thank Jakob Homan and Rik Farrow for their help with the article.

REFERENCES

[1] P.H. Carns, W.B. Ligon III, R.B. Ross, and R. Thakur, "PVFS: A Parallel File System for Linux Clusters," *Proceedings of the 4th Annual Linux Showcase and Conference*, 2000, pp. 317–327.

[2] S. Ghemawat, H. Gobioff, and S. Leung, "The Google File System," *Proceedings of the ACM Symposium on Operating Systems Principles*, Lake George, NY, October 2003, pp. 29–43.

[3] Lustre: http://www.lustre.org.

[4] M.K. McKusick and S. Quinlan, "GFS: Evolution on Fast-forward," *ACM Queue*, vol. 7, no. 7, ACM, New York, NY. August 2009.

[5] O. O'Malley and A.C. Murthy, "Hadoop Sorts a Petabyte in 16.25 Hours and a Terabyte in 62 Seconds," Yahoo! Developer Network Blog, May 11, 2009: http://developer.yahoo.net/blogs/hadoop/2009/05/hadoop_sorts_a_petabyte_in_162.html.

[6] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," *Proceedings of FAST '02: 1st Conference on File and Storage Technologies* (USENIX Association, 2002), pp. 231–244.

[7] K.V. Shvachko, "The Hadoop Distributed File System Requirements," Hadoop Wiki, June 2006: http://wiki.apache.org/hadoop/DFS_requirements.

[8] K.V. Shvachko and A.C. Murthy, "Scaling Hadoop to 4000 Nodes at Yahoo!," Yahoo! Developer Network Blog, September 30, 2008: http://developer.yahoo.net/blogs/hadoop/2008/09/scaling_hadoop_to_4000_nodes_a.html.

[9] S. Weil, S. Brandt, E. Miller, D. Long, and C. Maltzahn, "Ceph: A Scalable, High-Performance Distributed File System," *Proceedings of OSDI '06: 7th Conference on Operating Systems Design and Implementation* (USENIX Association, 2006).

[10] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable Performance of the Panasas Parallel File System," *Proceedings of FAST '08: 6th Conference on File and Storage Technologies* (USENIX Association, 2008), pp. 17–33.

[11] "Compare Name-Node Performance When Journaling Is Performed into Local Hard-Drives or NFS," July 30, 2008: http://issues.apache.org/jira/browse/HADOOP-3860.

[12] "Name-Node Memory Size Estimates and Optimization Proposal," August 6, 2007: https://issues.apache.org/jira/browse/HADOOP-1687.