Affects of Small Files on Read Requests in the Hadoop Distributed File System

Aaron Kuehler, West Chester University of Pennsylvania

Abstract – The Hadoop Distributed File System is a high throughput distributed file system designed to accommodate large data sets; average file sizes in the gigabyte-terabyte range. However, when a data set is composed of large amounts of small files, say in the kilobyte range, the storage system's semantics introduce high amounts of overhead in terms of file system block storage and read latency. This paper explains the architectural attributes which cause these problems and examines techniques to mitigate their impact when working with data sets comprised of large numbers of small files.

1. Introduction

Large scale networked, internet based, cloud enabled systems are an increasingly important means of storing and processing user data. Hadoop Distributed File System (HDFS) is an open distributed file system capable of handling such large scale computing platforms and is the backbone of both major academic research as well as industrial production environments. [1]

File system access latency composes much of the system overhead of Hadoop MapReduce[2]; an open implementation of the MapReduce computation model originally proposed by Google which is built on top of HDFS.[3] This bottleneck is generally compounded by the excessive presence of small files occupying HDFS blocks. This paper first explains the general architecture and design concepts employed in HDFS in section 2; it next examines several scenarios under which the performance of HDFS is adversely affected by large volumes of small files and provides summary discussion of research into the mitigation of such bottlenecks in section 3; finally, in section 4, general conclusions are made about the approaches presented by the work in section 3 and thoughts on future research are proposed.

2. Background

HDFS is a distributed file system designed to run on commodity hardware. HDFS is designed to provides high throughput for applications that have large data sets. [6] HDFS accommodates applications with large data sets; generally those types of data in which a single file's size is in the range of gigabytes or terabytes. As such, HDFS is intended to process large files by horizontally scaling to hundreds of nodes to support such storage loads.

2.1 Architecture

Each HDFS cluster consists of a single Master node; called the NameNode. The NameNode manages its cluster's metadata and controls client access to block data. The NameNode manages metadata such as the namespace of the cluster, block replica locations within the cluster, block lengths, etc. The NameNode is also responsible for triggering File System operations like open, close, read, write, rename on its file system objects: files and directories.

A HDFS cluster is also composed of a number of Slave nodes; called DataNodes. DataNodes are generally run on a per-cluster-node instance basis, though multiple DataNodes can be run on a single cluster node instance, and have the responsibility of

managing the block level storage of a cluster

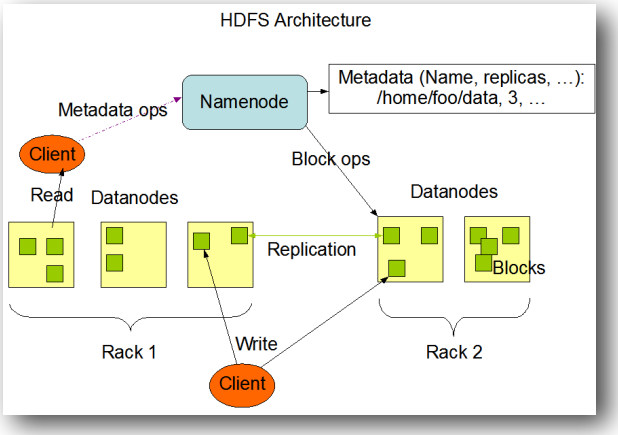

Figure 1 [6]

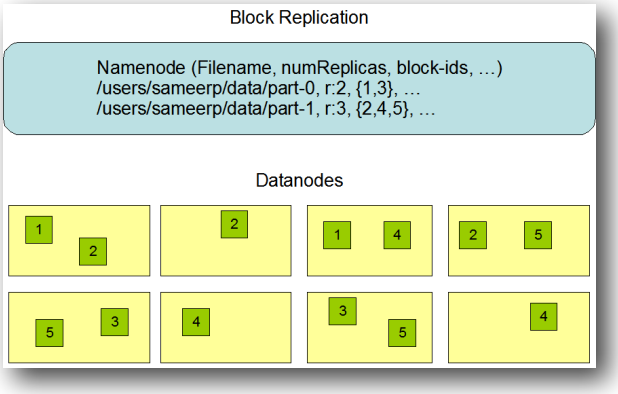

Figure 2 [6]

node. A DataNode responds to block requests of the NameNode: read, write, replicate, etc.

When HDFS receives a store request from a client, it receives the client's data as a file. The file is split into one or more blocks. The NameNode maps and distributes the blocks to its DataNodes.

2.2 Fault Tolerance and Error Handling

HDFS is designed such that very large files are stored across nodes in its DataNode cluster. A file is stored as a sequence of blocks; the block size of DataNodes in a HDFS cluster is configurable, however a typical block size is 64 megabytes. Block level replication is used to ensure that the probability of data loss is minimized. Based on replication factor configuration, the NameNode decides the quantity of replica of each block that should be distributed among the DataNodes of the cluster. The NameNode receives periodic heartbeat and block reports from the DataNodes of a cluster and uses this information to detect a failed DataNode and recover lost blocks.

In large implementations of HDFS, DataNodes may be connected through complex network architectures. In this case, a performance hotspot in data access can be the flow of traffic through network infrastructure as network bandwidth has traditionally increased orders of magnitudes slower than CPU throughput[11]. The NameNode is aware of the DataNode cluster's network topology, estimates transmission costs based on locality, and attempts to position replicas of related blocks in close proximity on network segments; it is said that the NameNode is Rack-Aware[10]. This is an attempt to reduce the usage of network bandwidth when retrieving related blocks across complex arrangements of DataNodes; IE when the blocks of a file are split across several data nodes. The reader is directed to the nearest DataNode which contains a replica of the requested block.

3. Small File Issues

The Small Files Problem consists of three fundamental problems. The first involves penalties in disk space when severely under-utilizing blocks on DataNodes of the cluster. The second incorporates the over-utilization of NameNode main memory. The third issue involves an overwhelming amount of client requests made on the NameNode.

3.1 DataNode Block Consumption

Typically HDFS stores data in 64 megabyte blocks. Larger block sizes are used to accommodate large files; as fewer block operations will be serviced by the file system mechanisms. A problem arises when large amounts of files which are significantly smaller in size than the standard block size are introduced into the file system. For

example, consider a single file of 300 kilobytes is introduced into a HDFS cluster; the file will occupy a single 64 megabyte (65536 kilobytes) block, rendering the remaining 65,336 kilobytes of the block as unusable by the file system. This is a loss of 99.5% of the available block space. As the number of small files increases on the file system, such losses are compounded.

3.2 NameNode Memory Consumption

The NameNode stores the state of its cluster in a file named FsImage on its node's local file system. The FsImage contains a persistent representation of the cluster's metadata, file-to-block mappings; BlockMap, and file system configuration. While the NameNode is running the FsImage is loaded into main memory for use by the NameNode. The BlockMap is meant to be compact such that the NameNode should not require more resources than available in commodity hardware.[6] Periodically DataNodes send their block reports back to the NameNode. The NameNode collates the block reports and updates its BlockMap. Regardless of the block size of a file its metadata remains relatively constant.[2] An increased number of allocated blocks on DataNodes causes an increase in block level metadata which needs to be maintained by the NameNode. It is easy to see how an increase in the number of small files may overwhelm the NameNode's main memory with an extremely large BlockMap.

3.3 NameNode Read Load

HDFS is designed to handle large data sets. As such it provides a stream based access strategy. For each file read, a HDFS client contacts the NameNode to retrieve the target file's metadata. For large files, this one-time-per-file metadata access introduces minor overhead as many blocks are read as part of the same file.[9] For example, imagine a read operation for a single large file across several blocks of the file system. In this case the client would request the file's metadata from the NameNode once and subsequently read the appropriate blocks. Now consider the same number of blocks occupied by small files; such that each block contains one small file. In this scenario the client performs N metadata requests to the NameNode.[9] Since HDFS uses a single NameNode per cluster, when large numbers of read requests for small, but related, files are received the name node may be overwhelmed servicing metadata requests.

4. Solutions

4.1 Small File Archiving

A profound solution in managing large amounts of small files in an HSFS cluster is to simply avoid treating small files as small files. A simple way to achieve this is to use an archiving mechanism. Archiving, or file merging, is the act of collecting two or more files together into a single logical structure; in the case of Hadoop Archives, or HARs, a single HAR maps to a file system directory[8]. A HAR contains an index, additional metadata, of the content size and offset of the individual, smaller, data files. The content of the HAR is transparently available to HDFS clients, however the difference in storage semantics offers significant benefits to HDFS in the problem areas outlined in section 3.

4.1.1 Mitigating DataNode Block Consumption

Archiving solves the problem of excessive DataNode block consumption of small files in a very simple way. Several small files are packed together into a single HAR. Optimally, the HAR is constructed in such a way that the sum of the content sizes of all of its partials and metadata are significantly close to the configured DataNode block size. Consequently, rather than several small files occupying blocks on the data node, a single file is created which consumes roughly the disk space allowed by a single block.

4.1.2 Mitigating NameNode Memory Consumption

Archiving makes many small files appear as one large logical file to NameNode, greatly reducing the burden on the NameNode's BlockMap. Because HARs are atomic in this sense, they can be

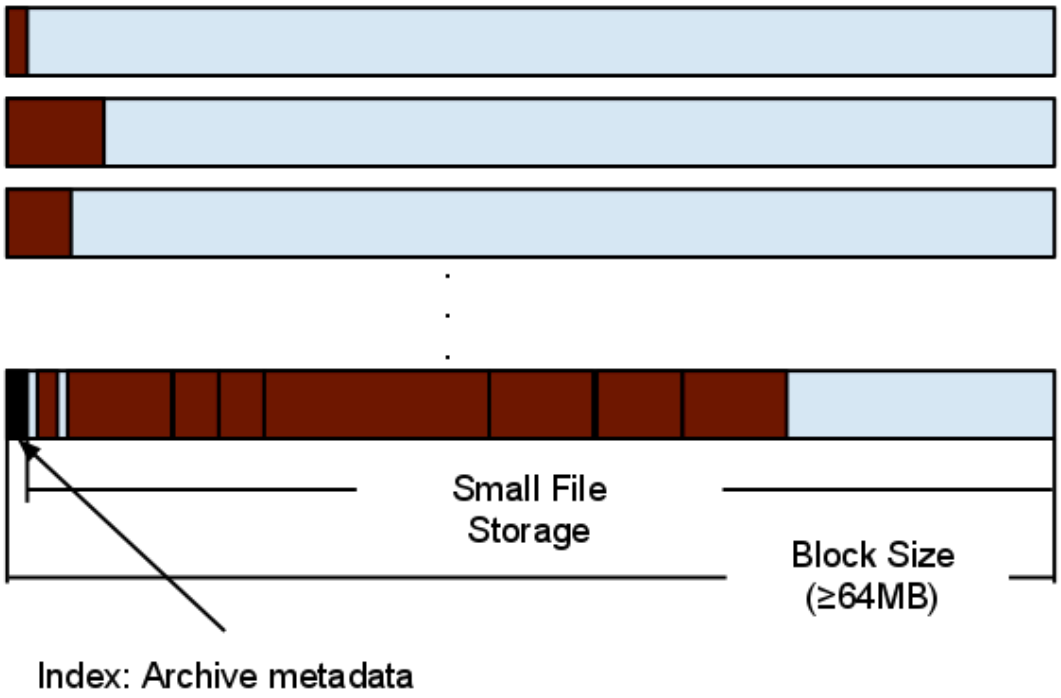

Figure 3: Archiving Overview

managed by the same semantics as other files system structures within HDFS; in replication and block reporting for example. Instead of reporting each of a HAR's content files, the DataNode needs only to report the HAR when sending its heartbeat report. Requests to the HAR's content are mapped by using the logical file's index.

5. Evaluation

While archiving seems to solve many problems in dealing with small files at the physical level, some severe logical problems remain.

5.1 NameNode Read Load

The main purpose of archiving is to reduce the namespace of the NameNode.[8] Though archiving reduces main memory consumption on the NameNode, the issue of request load cannot be mitigated by the proposed archiving mechanism. Each HDFS client request for the content of a HAR must still be serviced by the NameNode to resolve the HAR's location on the file system; therefore, the NameNode's metadata read

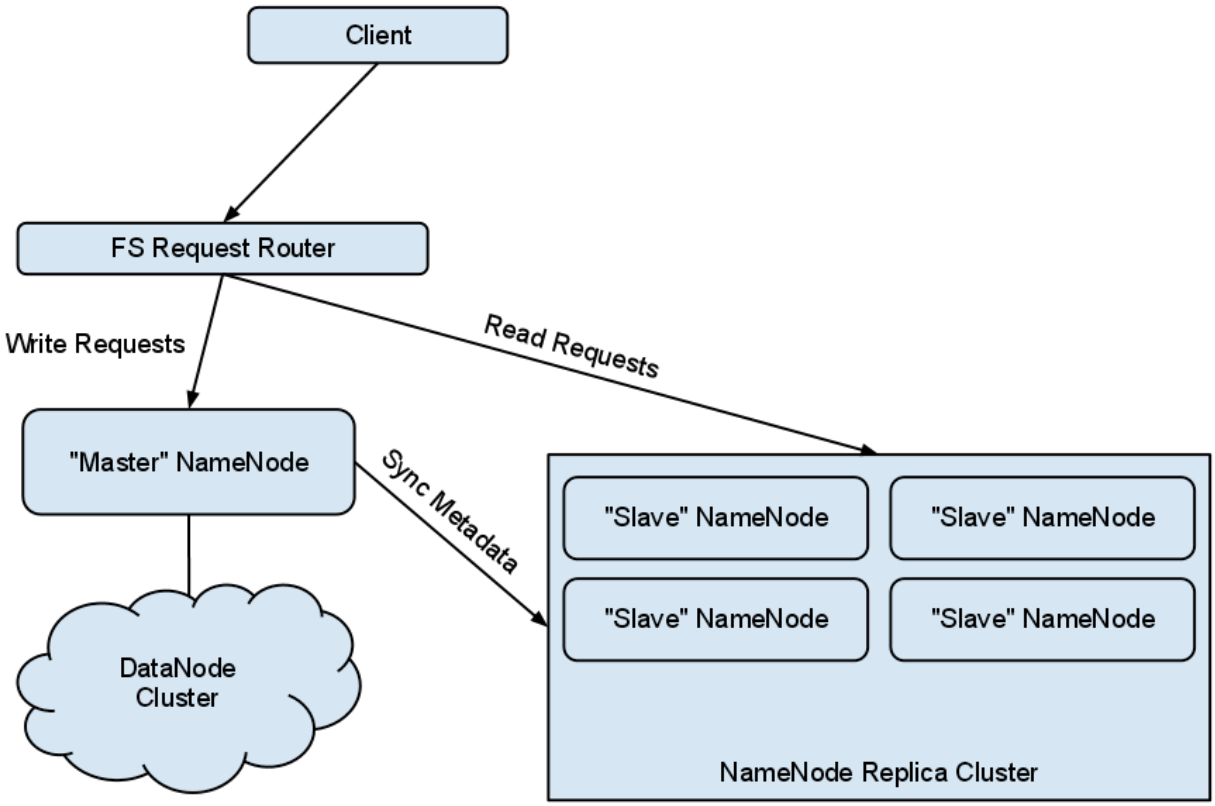

Figure 4: NameNode Replication Architecture

load is unaffected by the reduction in namespace.

5.2 DataNode Read Latency

The client request for a file contained within a HAR must perform an additional read step of the HAR's metadata. When a client requests a file contained within a HAR, the client request must trigger a read of the HAR's metadata in order to locate the requested content.

6. Suggestions

6.1 NameNode Replication

HDFS deliberately employs a single NameNode. While this design choice simplifies the implementation of the metadata management of the HDFS cluster[12], it is also an obvious area of exploration in mitigating the NodeName specific affects of small files in HDFS. Section 5.1 speculates that NameNode read load is unaffected by file archiving. Through more difficult to coordinate, it may be possible to mitigate the burden of metadata reads on the NameNode by adding a NameNode replication layer; as well as allowing failover of the NameNode. It should be noted that this method of mitigation is only optimal, or perhaps even feasible, in situations where the content of DataNodes changes infrequently in relation to read load. Multiple NameNodes could exists to increase parallelism of metadata reads and failover in the event of NameNode failure. Two problems exist with this approach: load balancing, and metadata coordination on write and updates.

6.1.1 Metadata Read Load Balancing

The former problem is well defined and could be mitigated much in the same way that common HTTP server clusters are load balanced. The cluster of NameNodes

would be placed behind a Load Balancing Node. The LoadNode, as it were, is responsible for receiving client requests, evaluating the current workload of the NameNodes it manages, then assigning the read requests based on the result of a node selection algorithm. Additionally, an Elastic NodeManager, such as Elastic Hadoop Auto-Deployer (EHAD)[13], may be employed to dynamically provision additional NameNode replicas in response to system load.

6.1.2 Metadata Coordination

Replicating the NameNode introduces the problem of coordinating metadata when the state of the cluster changes. To achieve this, a simple Master-Slave architecture is proposed for the NodeName replication process. A single Master NameNode would be used to service all mutating requests. When the mutation operation is complete, the Master NameNode will Push its updated state of metadata out to its Slave NameNodes; preferably using a low latency, publish/subscribe messaging interface like MPI[14]. It is the responsibility of Slave NameNodes to service only client read requests.

6.1.3 Automated NameNode Failover

An interesting side effect of NameNode replication is that the NameNode is no longer the single point of failure in the cluster. When a cluster's Master NameNode fails a Slave can be promoted to Master status. DataNodes would receive a message indicating that Master rights have been given to the new Master NameNode and heartbeat reports will be rerouted accordingly. The load balancing system will begin to send mutation requests to the new Master NameNode, and the new NameNode will begin publishing its metadata mutations to the rest of the Slave NameNodes.

7. Conclusion

This paper outlined the architecture of HDFS to ensure the reader understands the concepts which are core to understanding the Small Files Problem in HDFS. The Small Files Problem was defined, its causes and major affects were outlined. Archiving was presented as a possible solution to some of the negative affects of small files on components of HDFS clusters. It was shown that Archiving alone cannot mitigate all of the performance bottlenecks caused by the Small Files Problem. Suggestions were then made to relieve the burden of the remaining problems.

8. References

[1] B. Dong, X. Zhong, Q. Zheng, L. Jian, J. Liu, J. Qiu, and Y. Li, "Correlation Based File Prefetching Approach for Hadoop," *IEEE International Conference on Cloud Computing Technology and Science*, 2010, pp. 41-48

[2] B. Dong, J. Qiu, Q. Zheng, X. Zhong, J. Li, and Y. Li, "A Novel Approach to Improving the Efficiency of Storing and Accessing Small Files on Hadoop: A Case Study by PowerPoint Files", *IEEE International Conference on Services Computing*, 2010, pp. 65-72

[3] J. Dean, and S. GHEMAWAT, "Mapreduce: simplified data processing on large clusters," *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, 2004, pp. 10–10.

[4] Go. Soundararajan, M. Mihailescu, and C. Amza, "Context-aware prefetching at the storage server", *Proc. of the 2008 USENIX Annual Tech. Conf.*, 2008, pp. 377-390

[5] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," *Mass Storage Systems and Technologies, IEEE / NASA Goddard Conference*, 2010, pp. 1-10

[6] Hadoop Distributed File System Documentation, http://hadoop.apache.org/common/docs/current/HDFS_design.html, v0.20

[7]B. Jia, T. Wlodarczyk, and C. Rong, "Performance Considerations of Data Acquisition in Hadoop System," *IEEE Second International*

Conference on Cloud Computing Technology and Science, 2010, pp. 545-549

[8] Hadoop Archives Documentation, http://hadoop.apache.org/common/docs/current/hadoop_archives.html, v0.20

[9] J. Hendricks, R.Sambasivan, S. Sinnamohideenand, and G. Ganger, "Improving small file performance in object-based storage," Technical Report, Tech. Report CMU-PDL-06-104, May 2006

[10]Hadoop Rack Awareness, http://hadoop.apache.org/common/docs/r0.19.1/cluster_setup.html#Hadoop+Rack+Awareness.

[11] B. Welton, D. Kimpe, J. Cope, C. Patrick, K. Iskra, and R. Ross, "Improving I/O Forading Throughput with Data Compression," *IEEE International Conference on Cluster Computing*, 2011, pp. 438-445

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

[13] H. Mao, Z. Zhang, B, Zhao, L. Xiao, and L. Ruan, "Towards Deploying Elastic Hadoop in the Cloud," *International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, 2011, pp. 476-482

[14] X. Lu, B. Wang, L. Zha, Z. Xu, "Can MPI Benefit Hadoop and MapReduce Applications?," *International Conference on Parallel Processing Workshops*, 2011 pp. 371-379