Scalability Limits of HDFS

Christopher Chute, David Brandfonbrener, Leo Shimonaka, Matthew Vasseur

May 2, 2016

Abstract

1 Introduction

2 HDFS Architecture

Hadoop claims to be built on five core principles that govern the architectural decisions. These are (1) hardware failure is the norm. As a result, HDFS wants to be highly reliable in the face of hardware failure so the architecture incorporates replication of data across multiple nodes. Then HDFS wants to (2) allow streaming data access and (3) support very large data sets. With these goals in mind, HDFS wants to use a distributed system to accommodate large files and implement distributed reads to prevent streaming access from clogging the system. Next HDFA wants to (4) provide simple concurrency control with a write-once read-many model. This is less a feature and more a decision to weigh reads over writes by providing highly available and concurrent reads at the expense of allowing concurrent writes. And lastly, HDFS wants (5) portability, which makes sense as HDFS must work on different computer architectures

To satisfy these principles, the HDFS architecture was designed as follows. An HDFS cluster will have one unique NameNode and many DataNodes. Briefly, the NameNode holds the metadata of each file in the system. This metadata consists of (inode, mapping) pairs where the inode is a UNIX-style inode representing the file and the mapping holds the information about where the file resides in disk on the DataNodes. The NameNode holds all of this metadata in memory to facilitate fast metadata operations.

This single NameNode architecture is beneficial in a few ways. First, this facilitates large reads since the reads are distributed among the DataNodes and the relatively inexpensive metadata operations all happen on the NameNode and never block each other. If each node had to handle its own metadata operations, the metadata operation to indicate which node to read from could be slowed by threads performing large data operations at the node being queried. Second, a single NameNode architecture allows for a simple durability scheme via replication of each block of data across many DataNodes and organized by the central NameNode. If the metadata was not centralized, this replication operation and updating all data mappings would become complicated, and potentially overuse the network connecting the nodes.

Now we will examine certain aspects of the specific HDFS architecture in more detail.

2.1 NameNode and DataNodes

All metadata is kept in memory on the NameNode. As explained above, the metadata holds a UNIX-style inode and location on DataNodes for each file. These locations are represented as block locations in virtual memory of each of the DataNodes. So, upon creation of a file, the NameNode will allocate a block for the file in virtual memory on however many DataNodes the file will be replicated across (the default is 3 replicas). Then, the NameNode is responsible to pointing clients to the appropriate blocks to write or read files, and also to maintain the namespace. The NameNode keeps these maps to data blocks up to date by communicating with the DataNodes via heartbeats. The DataNodes are responsible for sending periodic heartbeats to indicate that they are functioning correctly that contain block reports. The block report gives the block locations of all files in the virtual memory of the DataNode. The NameNode uses this information to ensure each block is replicated enough times and to keep the locations of the blocks up to date.

Clearly durability of the NameNode is very important to the system. There are various schemes to ensure a durable NameNode. A write ahead log is maintained on stable storage and this can be used to build a BackupNode or CheckpointNode that essentially maintain a copy of the NameNode.

2.2 Reads and Writes in HDFS

Files in HDFS are append-only and can only be written to by a single writer at a time so that once data is written it cannot be overwritten. This simplifies con-

currency 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 to write to and which blocks to write to on those DataNodes. Then, the client will ensure a replicated write by sending data in TCP-style packets through a pipeline including all DataNodes over which the file will be replicated. This guarantees that the writes will be consistent across multiple DataNodes. On a read, a client must first query the NameNode to find which DataNodes contain replicas of the desired file and which blocks hold the file. Then, the client reads from the closest available DataNode directly.

3 Limitations of Single Namenode Architecture

Despite the benefits of the single NameNode architecture listed above, this architectural decision causes some potentially major scalability problems. These problems will be especially pronounced when HDFS is used to store many small files as opposed to few large files.

3.1 Physical Memory Size

The obvious problem with forcing all metadata to be in memory on the NameNode is that the NameNode will run out of available memory.

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. Then, it is clear that we must have

$$M > \lambda cF + iF$$

if we want to fit all metadata in memory. 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, ie we have that $\lambda \geq 1$. So, 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 memory limit is much more of a problem for many small files.

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 the NameNode operations are small enough and there will be few enough large files that they 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 be created 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 file is much smaller than a block, the DataNode can potentially hold a great number of small files and have to send a large block report containing the metadata of each of these files. This could potentially overload the NameNode CPU. Second, the number of block reports that a NameNode receives is directly proportional to the number of DataNodes in the system. So, if the system has a large number of DataNodes, it is possible to generate a large amount of internal load by having many DataNodes.

Another potential source of CPU bottleneck is external-load from clients querying the NameNode. As explained above, on both reads and writes a client must first contact the NameNode to find the appropriate block locations and DataNodes for the read or write. The NameNode then has to perform a metadata operation to determine what to return to the client. So, 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 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 cncurrently.

So, with these potential CPU bottlenecks in mind, we pose the question of whether it is possible to generate an appropriate load so as to reach these CPU bottlenecks before the NameNode runs out of memory. This will be addressed by out experiments.