Access Pattern of HDFS on Linux Local File System

Zhichao Yang, 20661179
Liwen Dai, 20552153

Abstract

We studied the access pattern of HDFS to Linux local file system. We analyzed the logs of all file system calls generated by the tool strace when running different tests, including reading, writing, creating files, making directories and modifying the metadata on HDFS in pseudo-distributed mode. We observed the behaviour of the NameNode and DataNodes during these operations. Our important discoveries include that a DataNode performs write operations by writing to the block file and its metadata file in turn for 64KBytes at a time. We also summarized the behaviour of the NameNode when accessing a large number of files, where the synchronization of small metadata files to disk becomes a heavy cost. According to our findings we proposed that buffering written data eagerly and performing batch writes in the local file system would improve the throughput of HDFS.

1 Introduction

The Hadoop Distributed File System (HDFS) is designed to run on commodity hardware[1] and its usage has been increasing greatly in recent years. HDFS is built using the Java language, and can be deployed on a wide range of machines. A common way of deployment is to install hadoop on a Linux machine, which means HDFS relies on Linux local file system to access the local storage. Although this feature is user-friendly, we know that modern cloud applications, such as HDFS, have very different workloads from personal Linux machines: HDFS is tuned to support large files, and a typical file in HDFS can be gigabytes to terabytes in size[1], which is rarely the case for personal computers. In this project, we study the access pattern of HDFS to Linux local file system, and analyze how the Linux file system APIs are used. Based on the access pattern we propose ways of tuning Linux local file system in order to improve the performance of HDFS.

1.1 HDFS Architecture

An HDFS cluster consists of a single NameNode and a number of DataNodes. The NameNode manages the file system namespace can controls file access by clients[1]. A file is split into blocks, each of typically 64 or 128 MBytes, which are stored in the DataNodes. The blocks of a file are replicated for fault tolerance.

1.2 Methodology and Setup

We performed our tests on pseudo-distributed mode, where a NameNode or a DataNode runs its own daemon and has its own root directory for storage, but everything happens on a single machine. This method has saved us from the complex deployment of a real cluster, while still providing results close to those from the real-distributed mode. When testing large read and write operations, we deployed 3 DataNodes, with 2 replicas for each block, and block size of 64 Mbytes.

We used the *strace* command on Linux to print the logs of all file system calls, based on the process IDs of NameNode and DataNodes. Note that a daemon can create hundreds of child processes, it is necessary to trace them all, to figure out what has happened on the nodes.

2 Trace Access Pattern to Local File System

We analyzed the access patterns of HDFS to local file system under several workflows: idle state, short appending writes, large writes, large reads, creating small files, creating directories and modifying the metadata.

2.1 When HDFS Is Idle

We observed some looped communications, or heartbeats, were happening between the NameNode and DataNodes. In each iteration, the NameNode started with calling statfs() to get the file system information on its mount point, the root directory of its storage. Then it called epoll_wait() on a file descriptor, which could be a duplicate of some TCP socket, where the DataNode then read twice, 4 and 394 bytes in sequence, and replied 40 bytes. Such an iteration on the DataNode occurred once in about every 5 seconds.

At the same time, we observed similar loop pattern on all three DataNodes, with about 3 seconds for each iteration. For each iteration, a DataNode called statfs(), then it tried to read from a TCP socket but ended with a failure. So it wrote back 398 bytes, which led to the next successful read of 40 bytes. Note that the number of bytes here matched our observation on the NameNode, which implied they were talking. Our guess that this communication was about heartbeats was verified by setting the heartbeat interval to 30 seconds long. This change on configuration allowed us to better observe the “real work” of HDFS without interruption of the noise of heartbeats.

It became obvious that when idle, DataNodes were sending heartbeats to the NameNode via TCP. They used file system calls, but these read and write operations didn’t touch the disk, hence probably not a real concern for the design of local file system. It is worth mentioning that, we could verify that some file descriptors were indeed TCP sockets, by *readlink* for example, because they were still open when the trace ended. For the later part of this report, many file descriptors were closed after use, which made it harder to figure out what they really were.

2.2 Appending Text Repeatedly

In this part we kept appending a text file of 5.1 MB in size, from local file system to the same target file in HDFS. We observed the “data flows” on the DataNodes. First, the DataNode called stat() *to* check the two files under the current/rbw directory of a DataNode -- a block file (blk_xxx) and its metadata file (blk_xxx.meta), in the “Replica Being Written” state, representing the last block of an open file[2], and created them if not existed. Next, 64 KBytes at a time, the DataNode read input data from a TCP socket, wrote it to the block file, and appended 504 bytes to the metadata file. When the append command is done, or when the block file reached its size limit, 64 MB as we set, the two files were closed, and renamed to the current/finalized directory. If more blocks were needed, a new pair of block/metadata files were created the process repeated as descripted above. Also, if the append command was completed, the DataNode performed more than 300 calls to close() in a row, to a list of file descriptors. When the next append command came in to the same target, the last finalized block and metadata files were renamed back to rbw/ directory, and repeated the process.

The interesting part is, every time before the data was written to the block file, it was first transferred to another socket, going to another DataNode. This indicated that for each block, both replicas were being written at the same time, yet asynchronously, since the acknowledgement of 17 bytes was not observed until the whole block was written. Text log writes of 200 to 300 bytes in size each time were observed at the beginning as well as the end of writing every block.

We also observed the activities on the NameNode, which had a relatively short strace history in this case. Besides talking through sockets, few interactions were observed to the local file system, besides writing to its own logs, with 200-300 bytes each time in size. We believed that a better way to observe the NameNode would be through interactions with a large number of HDFS files.

2.3 Large Writes

In this part we performed the test by putting local a text file of 452 MB to HDFS. We discovered that access pattern was very similar to that of short writes, except that reads and writes are much more intensive, without much noise (like the spanned close() calls).

2.4 Large Reads

We traced what happened when reading from the text file of 452 MB which we had just appended to HDFS. Perhaps not surprisingly, no writes to local file system were observed on the NameNode, not even log writes, though it did some short reads and writes to the socket.

On the DataNodes, we could verify by counting that for each block, only 1 replica was read. The DataNode checked stat of the block and its metadata files, opened them, and transferred 64 KBytes at a time, as we had expected. Something different could be that each time the 543 bytes of metadata was transferred though write() to the socket, while the file 64 KBytes of file content was transferred using sendfile(), which removed the overheads of copying the chunk to user space.

2.5 Creating small files

This test try to create 100 small files. On the NameNode, it will first create the file with a suffix “COPYING”, indicating the file is being creating. Then it will save the file to block metadata. It will then write the metadata indicates that content has written to the file. At last it will save metadata indicating the renaming of the file without suffix “COPYING”. The important finding here is, each creation involve four fdatasync(), which means four disk I/O, which can make the process very slow.

On the DataNode, datanode will first create the block file “blk_xxx” and metadata “blk_xxx.meta” for the file. Then write content to the block. Finally it will move the block file from /rbw to /finalized.

2.6 Creating directories

This test try to create 100 folders. On the NameNode, it will append the metadata to metadata log file for each mkdir operation. On the datanode, it does not have much file system calls.

2.7 Changing metadata

This test try to change the access permission of 100 files. On the NameNode, it appends the new metadata to the “edits_inprogress_xxx” file, and flush the change to disk. The important finding here is, the metadata change is appended to the metadata file, and it is overwritten. This just confirm that the metadata on the NameNode is a log based approach. On the DataNode, no operation was done, when we change the metadata of files.

2.8 Checkpointing

During the test, the checkpointing process was observed. This results can help us understand how does the NameNode handle the metadata. On the NameNode, the file system metadata are stored in two structure: fsimage and edits. Fsimage contains complete state of the file system at a point, and edits are log file that list each file system change. Checkpointing is the process of merging the content of fsimage with all the changes in edits log to create a new fsimage file. Detailed process can be found in our appendix. Instead of using unbounded edits log, checkpointing can help reduce the Namenode startup time. Since it can load the state from fsimage rather than apply all edits log.

3 Summary of API Usage

For each test case we also short-listed the local file system APIs called by HDFS, how many times each was called, and how much time each API had consumed. The complete lists can be found in our appendix.

3.1 API Usage in Reads and Writes

From tests in sections 2.1 - 2.4, only 13 local file system APIs were observed on the NameNode, and 24 on the DataNodes. Large writes and reads involved even fewer kinds of API calls. To our surprise, the nodes spent most of the time in epoll_wait(). A DataNode often spent over 99% of its time in epoll_wait(), while read() and write() took about 0.5% of the time. In the NameNode, fdatasync() also took significant amount of time, with 13.6% in our test of large writes(section 2.3).

Further analysis showed that during write tests (section 2.2, 2.3), epoll_wait() happened when the input socket became temporarily unavailable for a read operation, and during read test (section 2.4) it happened when the output socket became unavailable for a write. A reasonable explanation could be that write operations were not synchronized. Perhaps more importantly, this high resource contention implied a bad idea to deploy multiple virtual HDFS nodes on a single machine.

4 Discussion

Since HDFS is designed to process large files, some researchers had focused on improving its performance on large number of small files [3]. We discovered that although typical workloads mainly concern big files, it is inevitable to create lots of small files on the NameNode, a single bottleneck of the cluster. According to our observation, one small file creation involves four fdatasync() system call, which means 4 disk I/O operations. Such behavior significantly influence the performance. One optimization could be batch some writes within one single disk I/O, since all these metadata change are continuous in the file.

We have also discovered that, when writing data to HDFS, a DataNode would write to the block file and its metadata file by turns. We believe this pattern could potentially cause a heavy overhead of disk seek latency, since it would take 1000 turns to write a block of 64 MBytes. An optimization of aggressive batched writes on the DataNodes could improve this situation. This could be implemented by tuning HDFS or deploying a local file system by FUSE. Since a DataNode has its specific root directory for HDFS storage, such an optimization can be limited to this place and doesn’t have to affect the reliability of the whole machine.

References:

[1] Borthakur, Dhruba. "HDFS architecture guide." Hadoop Apache Project 53 (2008). APA
[2] Yongjun Zhang, “Understanding HDFS Recovery Processes (Part 1)”, http://blog.cloudera.com/blog/2015/02/understanding-hdfs-recovery-processes-part-1/
[3] Bende, Sachin, and Rajashree Shedge. "Dealing with Small Files Problem in Hadoop Distributed File System." *Procedia Computer Science* 79 (2016): 1001-1012.
[4] Chris Nauroth, “HDFS Metadata Directories Explained”, https://hortonworks.com/blog/hdfs-metadata-directories-explained/
[5] Andrew Wang, “A Guide to Checkpointing in Hadoop”, http://blog.cloudera.com/blog/2014/03/a-guide-to-checkpointing-in-hadoop/