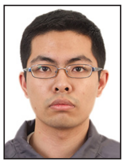
Scaling Namespace Operations with Giraffa File System

KONSTANTIN V. SHVACHKO AND YUXIANG (CHRIS) CHEN

Konstantin V. Shvachko is an expert in big data technologies, file systems, and storage solutions. He specializes in efficient data structures and algorithms for large-scale distributed storage systems. Konstantin is known as an open source software developer, author, inventor, and entrepreneur. He is currently a part of the Hadoop team at LinkedIn.
kshvachko@linkedin.com



Yuxiang Chen is a graduate student in the School of Computer Science, Carnegie Mellon University. In summer 2016 he worked as an intern with the Hadoop Development team at LinkedIn. His research interests include cloud computing and distributed systems.
yuxiang1@andrew.cmu.edu

HDFS clusters rely on a single NameNode, the master, as its metadata service. Single master design of HDFS is known to be a limiting factor for potential growth of the file system in its size and performance. Project Giraffa replaces the single master of HDFS with a dynamically distributed namespace service, thus overcoming scalability limits of HDFS while remaining fully compatible with it. We focus on the performance of namespace operations and present a benchmark that demonstrates that Giraffa can linearly scale the throughput of metadata operation by simply adding more servers to store the file-system namespace.

Apache Hadoop is a system for distributed storage and computation for big data problems. As members of the Hadoop Development team at LinkedIn, it is our daily job to monitor the condition of our clusters, fix problems, and optimize their performance. The most troubling problems are those that result in a cluster-wide crisis.

One day, a user complained that his job was running unusually slowly and not progressing. We thought it could be a problem of the particular job. But with more similar reports coming in, we realized that the cluster became stagnant for most of the jobs assigned to it. Eventually we noticed that the NameNode was unresponsive, running at 100% CPU. Further drilling into HDFS audit logs, we identified one job that was producing hundreds of thousands of namespace operations per second, saturating the NameNode and degrading its performance. The majority of these operations were read requests such as `listStatus`, `getFileInfo`, `getBlockLocations`.

We call the above scenario the “bad client” problem, which means a single “bad” job can make the whole cluster unavailable for everybody. The root cause of this problem is the single master architecture of HDFS, where the performance of a single NameNode, the single master, can constrain the performance of the entire cluster.

Scaling file system metadata along with its data is our primary motivation for building the Giraffa file system. We show that Giraffa metadata operations scale linearly and thus can prevent the bad client problem. See [4] for different aspects of scalability limitations of HDFS architecture [6].

Giraffa Overview

Giraffa [5] is a distributed, highly scalable file system that aims to:

1. Support millions of concurrent clients
2. Store trillions of objects
3. Maintain exabyte total storage capacity

Giraffa is intended to scale both the data storage and its metadata. Giraffa keeps its metadata—directories, files, and blocks—in a distributed key-value store, currently Apache HBase, as a single table distributed across multiple servers, while file data are stored in block

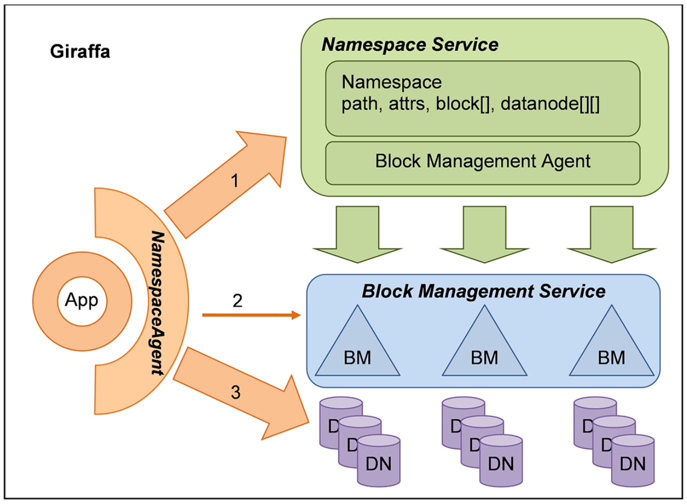


Figure 1: Giraffa Namespace Agent obtains metadata from Giraffa Namespace Service and streams data to or from HDFS DataNodes, while Giraffa Block Manager maintains all blocks.

files located on HDFS DataNodes. In other words, we still store all the data in DataNodes as Hadoop does. However, we save all the information that is stored in the NameNode in Hadoop to an HBase table in Giraffa. This architecture makes Giraffa a drop-in (no data copy) replacement for HDFS. Figure 1 shows the high-level architecture of Giraffa.

In Giraffa the file system metadata is served by the Namespace Service, which is composed of a single HBase table called Namespace. The Namespace table stores records corresponding to files and directories. Each record has a unique key, identifying the file or the directory, and contains the following attributes: local name, owner, group, permissions, access time, modification time, block size, replication, length, and a directory flag. When you need to read a file, you get the file's list of blocks and their locations, so your application can read the data from the respective DataNodes. When you write to a file, Giraffa allocates a block using its BlockManager. The client then writes data to the designated DataNodes.

BlockManager is another service that is used to maintain the flat namespace of blocks. The BlockManager is responsible for:

1. New block allocation
2. Scheduling block replication and deletion
3. DataNode management: process DataNode block reports, heartbeats, detect lost nodes

HBase automatically partitions its tables, and this allows Giraffa to dynamically partition its Namespace. That is, file and directory metadata—table rows—can automatically migrate between nodes based on nodes' utilization and load-balancing requirements. Since metadata is distributed across multiple nodes, this allows the number of files in the file system to increase and ensures that Giraffa is able to deal with trillions of files representing as much as 1000 PB of data on a single cluster.

Row keys identify files and directories as rows in the Namespace table, and they also define the sorting of the rows in the table. Thus, keys play an important role in Namespace partitioning. Row-key definition is based on the locality requirement and is chosen during file-system formatting.

Currently the row key is implemented as a byte array representing the full path to a file in the namespace tree. For example, file `/user/jsmith/job.xml` is identified by the row key, which is a byte representation of the string "`/user/jsmith/job.xml`". Lexicographic ordering of such keys guarantees locality of reference—that is, the children of the same directory fall into the same table partition, a region, most of the time. In the future we plan to define the row keys based on unique immutable INode IDs, which include selfID and two nearest parent IDs. This way, we still guarantee the locality of reference but also allow in-place renames—that is, if a file name changes, it remains in the same region because name changes do not affect row key values.

Giraffa is still in an experimental phase. The problems remaining to be addressed include:

1. Full set of HDFS functionality
2. INode ID-based keys to allow in-place atomic rename
3. Distributed block management
4. Short-circuit HBase metadata into itself
5. HBase scalability: single HMaster, region redundancy

Setting Up a Giraffa Cluster

We've used Giraffa on Java 8 without issues, but it also works with Java 7. We need Gradle 2.5 to build Giraffa sources. Similar to Hadoop, Giraffa uses Google Protocol Buffers version 2.5.0. Giraffa currently depends on hbase-1.0.1 and hadoop-2.5.1.

Although the Giraffa Wiki page on GitHub has instructions for setting up Giraffa in standalone mode, we will show you how to install Giraffa on a real cluster. Our cluster consisted of 11 physical servers (node-001 to node-011). Below are the step-by-step instructions on how to set up the cluster. One may consider writing a batch of scripts to automate the installation process.

Hadoop 2.5.1 Setup

Set up Hadoop normally if you haven't already, following Cluster Setup instructions [1]. HDFS cluster status can be checked via the NameNode Web UI at http://node-001:50070. In our case, node-001 runs the NameNode process, while the other 10 servers node-002–node-011 run DataNodes.

HBase 1.0.1 Setup

1. Follow the official Apache HBase Reference guide [2] to configure and set up HBase cluster.
2. Start HBase. In our cluster, node-001 hosts HMaster and HQuorumPeer processes, and the remaining machines host HRegionServer process. The status of the HBase cluster can be checked on the HMaster Web UI at http://node-001:16010.
3. Stop HDFS and HBase after testing.

Giraffa Setup

1. Download and build Giraffa according to [3].
2. Copy giraffa-standalone-0.4-SNAPSHOT.tgz to all nodes, and change the configuration according to [3].
3. Start and format Giraffa using `giraffa format` command. The script that starts Giraffa will also bring up Hadoop and HBase.

After completing these steps, you should be able to run file system operations on Giraffa. Here are some examples of Giraffa CLI commands.

Get listing of the Giraffa root directory:

```
bin/giraffa fs –ls /
```

Create a new directory:

```
bin/giraffa fs -mkdir testdir
```

YARN Setup

1. Configure YARN according to the official Apache Hadoop tutorial [1].
2. Use Giraffa commands to start YARN daemons: the ResourceManager on node-001, and NodeManager processes on the rest of the nodes:
   ```
   bin/yarn-giraffa-daemon.sh start resourcemanager
   bin/yarn-giraffa-daemon.sh start nodemanager
   ```

The cluster setup is now complete.

TeraSort is an example of a YARN application. By default it starts small MapReduce jobs, which will test the entire setup. Note that in this case all data is stored and processed on the Giraffa file system rather than on HDFS.

1. Run TeraGen:
   ```
   bin/yarn-giraffa jar $HADOOP_HOME/share/hadoop
   /mapreduce/hadoop-mapreduce-examples-2.5.1.jar
   teragen 10000000 /teragen
   ```
2. Run TeraSort:
   ```
   bin/yarn-giraffa jar $HADOOP_HOME/share/hadoop
   /mapreduce/hadoop-mapreduce-examples-2.5.1.jar
   terasort /teragen /terasort
   ```
3. Run TeraValidate:
   ```
   bin/yarn-giraffa jar $HADOOP_HOME/share/hadoop
   /mapreduce/hadoop-mapreduce-examples-2.5.1.jar
   teravalidate /terasort /teravalidate
   ```

The Benchmarks

In order to show that Giraffa scales linearly with the number of region servers, we built a benchmark. In this benchmark, we first create a number of files, and then run a MapReduce job, where each mapper calls `listStatus` for those files.

Suppose we have *m* map tasks running in parallel, and each map task performs `listStatus` for *n* files. Then the result we want to output is `(m * n / t)`, where `t` is the time of the mapping phase. YARN does not guarantee that all tasks start at the same time. In order to synchronize our *m* map tasks running in parallel, we set a start time `t1`. All map tasks will wait until time point `t1` before running the `listStatus` operations. That way we can guarantee that the mappers hit the Namespace Service all at once, providing maximum workload on the service. Finally, we record time `t2` when the last map task stops, and measure the running time for all mappers as `t = t2 – t1`.

This benchmark gives us the number of read operations that Giraffa can handle per second, which is an important metric of the cluster performance.

The configuration of the experiment is as follows:

We set up a cluster with 11 nodes. node-001 hosts master processes: NameNode, HMaster, ResourceManager. node-002–node-011 host the slave processes: DataNode, HRegion, Node-Manager. We managed to run 220 map tasks simultaneously on our cluster, and required each of them to perform listStatus for 10,000 files. We collected the running time and repeated this experiment several times to get rid of the soft bias.

We chose the number of map tasks to run (220) based on the capacity of the cluster. YARN as a resource manager allocates containers, each of which runs a single task and defines how much of execution resources, RAM and CPU (vCores), to be dedicated to a specific task. Thus, the cluster capacity is determined by the total amount of RAM and the total number of vCores. Our goal was to fully utilize the cluster without overutilizing it, so that all mappers ran simultaneously rather than in "waves."

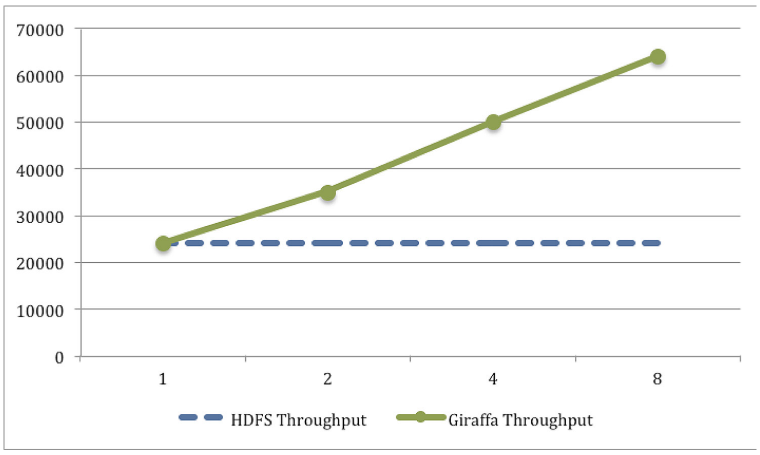


Figure 2: Giraffa read performance scales linearly with number of servers compared to the single NameNode.

In our cluster, we had a total of 220 GB of RAM and 320 vCores available for containers. Each task requires at least 1 GB of memory and 1 vCore. We therefore decided to set the number of map tasks to be 220, which satisfies the single wave requirement without affecting the performance of the cluster.

We started the Giraffa benchmark with a single region server serving the entire Namespace table. Then we used the HBase `split` command to dynamically partition the table into two regions served by two different region servers. *Dynamically* here means that we did not need to copy file data or restart the cluster for repartitioning. Then we similarly split the table into four and eight regions and made sure that each of them was assigned to a different region server.

In order to compare the performance of Giraffa and HDFS, we ran the same benchmark on an HDFS cluster using the same hardware. The main difference is that the Hadoop cluster does not need HMaster and HRegion processes. We stopped the Giraffa cluster, set up HDFS, and configured and started YARN with HDFS according to [1].

For Hadoop we also ran 220 parallel mappers with each of them performing `listStatus` for 10,000 files. Figure 2 shows the benchmark results.

The x-axis represents the number of region servers serving Giraffa namespace, and the y-axis represents the number of read operations per second that the file system processed. Since in our HDFS cluster we had only one NameNode, the number of read operations per second does not change, and the dashed line serves as the baseline. The solid line represents the throughput of Giraffa. It shows linear growth of read operations per second with the number of region servers. The benchmark is limited to eight region servers because of the cluster size limitations.

From these tests, we can see that the read performance of Giraffa scales linearly with the number of region servers. The write performance was partly addressed in [7]. It shows that the `mkdir` operation scales linearly. We expect that some operations like file `create` or `delete` will scale linearly as well, but some like `addBlock` will not due to limitations of the current Giraffa implementation, something yet to be fixed.

Conclusion

We showed that the Giraffa file system could linearly scale metadata operation for read requests by simply adding more servers to store the file-system namespace.

Authors of [7] came to the same conclusion as they benchmarked Giraffa along with two other systems, ShardFS and IndexFS, on a variety of metadata workloads. It shows that Giraffa scales linearly in throughput as more servers are dynamically added to the system for most of the workloads.

References

[1] Apache Hadoop Cluster Setup: https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/ClusterSetup.html.

[2] Apache HBase Configuration: https://hbase.apache.org/0.94/book/configuration.html.

[3] How to Set up, Build, and Use Giraffa: https://github.com/GiraffaFS/giraffa/wiki/How-to-Setup,-Build,-and-Use-Giraffa.

[4] K. V. Shvachko, "HDFS Scalability: The Limits to Growth," *;login:*, vol. 35, no. 2 (April 2010): https://www.usenix.org/legacy/publications/login/2010-04/openpdfs/shvachko.pdf.

[5] K. V. Shvachko, P. Jeliazkov, "Dynamic Namespace Partitioning with Giraffa File System," Hadoop Summit 2012: http://lanyrd.com/2012/hadoop-summit/stttw/.

[6] K. Shvachko, H. Kuang, S. Radia, R. Chansler, "The Hadoop Distributed File System," 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), 2010.

[7] L. Xiao, K. Ren, Q. Zheng, G. A. Gibson, "ShardFS vs. IndexFS: Replication vs. Caching Strategies for Distributed Metadata Management in Cloud Storage Systems," Sixth ACM Symposium on Cloud Computing, 2015.