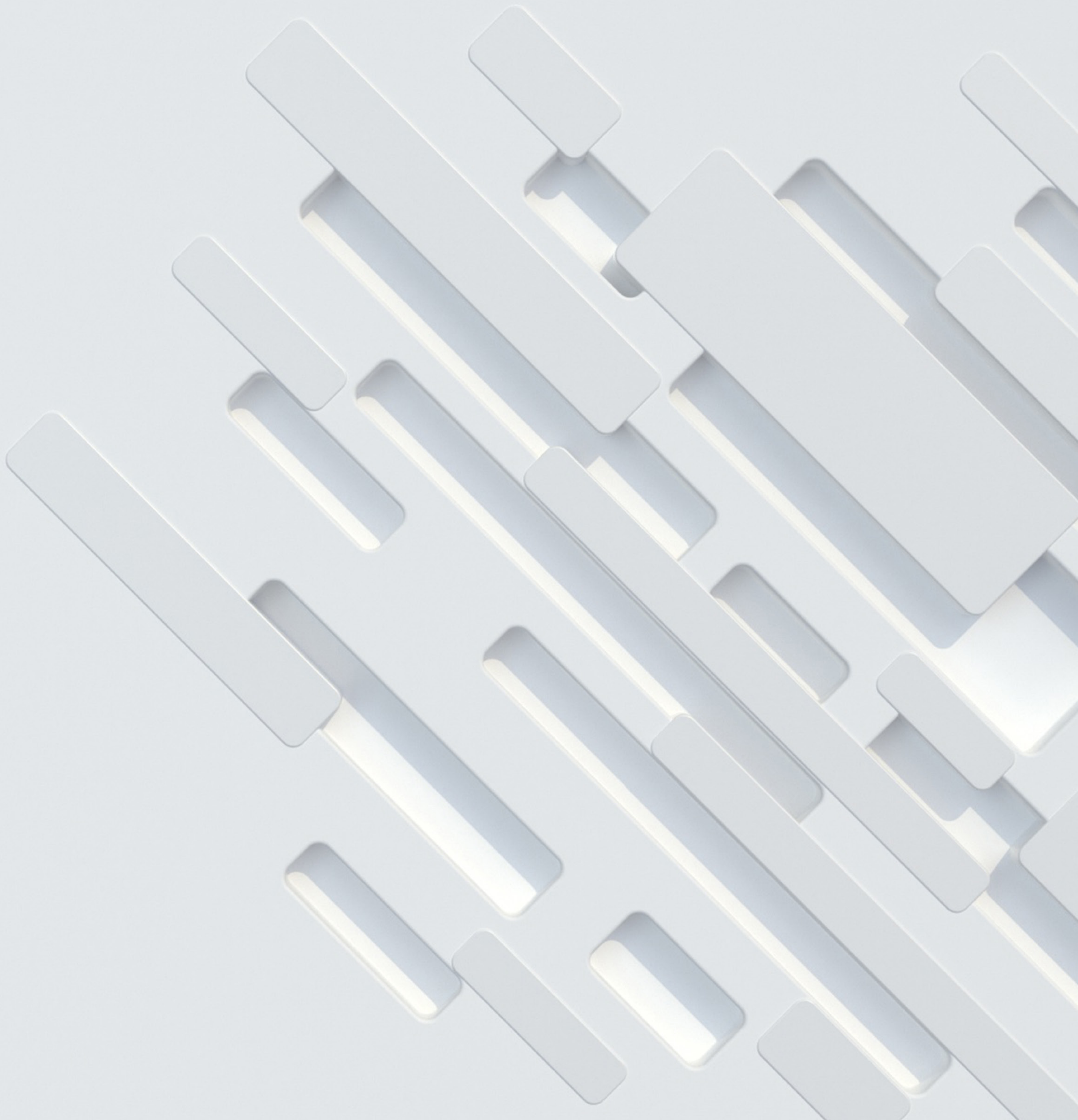
Hadoop Performance on WekaFS



SUMMARY

Hadoop and Spark are frameworks that are widely used for processing Big Data in a distributed environment. The Hadoop framework was built as a shared-nothing standalone architecture that relies heavily on data locality to improve performance. Data locality is the notion that data is processed on the local HDD/SSD inside the server to avoid/alleviate network traffic. While this may work well for small environments, it is cumbersome to manage at scale because complete compute nodes have to be added in order to increase available capacity. In addition, the data cannot be shared with other applications and analytics tools; so Hadoop becomes an island of storage that requires back and forth copying of data to and from other application data repositories.

WekaIO™ can remove this constraint by allowing all of the user applications to access a single namespace. This is done by using the Weka file system (WekaFS™) POSIX client to access the data store instead of relying on dedicated Hadoop nodes utilizing HDFS. The result is a single data set that improves performance, eliminates copying of data, and optimizes cost, as customers no longer have to manage multiple storage clusters.

WHAT IS HDFS?

HDFS (Hadoop Distributed File System) is a storage system used by several applications, including Hadoop, to implement a distributed file system (FS) for parallel data access across scalable clusters. HDFS was originally architected to leverage low-cost hardware that frequently crashed, and to minimize network traffic by performing all tasks locally on the node that possesses the data. HDFS is not a Portable Operating System Interface (POSIX) compliant file system, which means that data stored in the HDFS format cannot be shared with other applications that expect POSIX file semantics. With HDFS, data cannot be accessed by simply mounting the file system as a POSIX file system; instead, data is accessed via an API or via CLI tool.

HADOOP SUPPORT FOR POSIX FILE SYSTEMS

Hadoop has the ability to mount directly to a POSIX file system "out-of-the-box". To access data in this manner, use Hadoop's built-in connector to POSIX file systems called LocalFileSystem.

Any Hadoop application can use LocalFileSystem to access WekaIO's global namespace, removing the need for a dedicated Hadoop/ HDFS cluster. LocalFileSystem can be mounted as a POSIX file system (FS) that will be recognized by the applications

Enabling LocalFileSystem support requires a code change to use a local path:
Replace hdfs path (starting with hdfs://) with local path (starting with file://)

BENEFITS OF WEKAFS

NO TRIPLE REPLICATION REQUIRED

HDFS utilizes triple replication for data protection, which becomes prohibitively expensive when high performance storage media such as NVME or Flash is required to drive performance and reduce latency. WekaFS eliminates the need for three data copies as it employs a form of erasure coding that delivers significantly better capacity utilization. The WekaFS mount not only replaces HDFS as the shared FS, but also provides efficient distributed data coding. Capacity utilization jumps from 33.3% to over 70%.

DATA SHARING ACROSS APPLICATIONS

Hadoop clusters typically operate as stand-alone application clusters that perform business critical tasks required for further workflow needs. Other applications may need this data, requiring a full data copy and doubling storage costs. Weka's POSIX file system allows multiple applications to access a single copy of data reducing storage resource needs. This reduces cost and saves time as data does not need to be replicated across islands of storage.

HUGE PERFORMANCE GAINS

The Weka file system running on NVMe accelerated flash delivers significant performance gains for Hadoop environments. As the following benchmark results show, WekaFS running on virtualized environment in AWS on a minimal 8 node cluster achieved 28GB/sec read performance and 12GB/sec write performance.

WekaFS OPERATIONS (OPS) TESTING ENVIRONMENT

The following section provides insight into WekaFS performance for a Hadoop cluster mounted presented as a local mount.
For testing Hadoop performance on WekaIO, the following environment was created in Amazon AWS:

WekaIO Storage Cluster: x8 i3en.12xlarge instances running WekaFS version 3.6.2
Hadoop cluster: x16 c5n.18xlarge instances
Hadoop: Ambari version 2.7.0.0, HDFS 3.1.1, YARN 3.1.1, MapReduce2 3.1.1, Tez 0.9.1, ZooKeeper 3.4.0

ENVIRONMENT SETUP

Benchmarks were run in two scenarios with Hadoop on weka:

1. Benchmark is run against HDFS which is configured to run from the Weka file system, WekaFS
2. Benchmark is run directly against WekaFS

For each of the 16 hadoop members, a 4TB file system was created in Weka. Each file system was then mounted to /mnt/storage on each host. Additionally, a 10TB, shared Weka file system was mounted to /mnt/weka of each host.

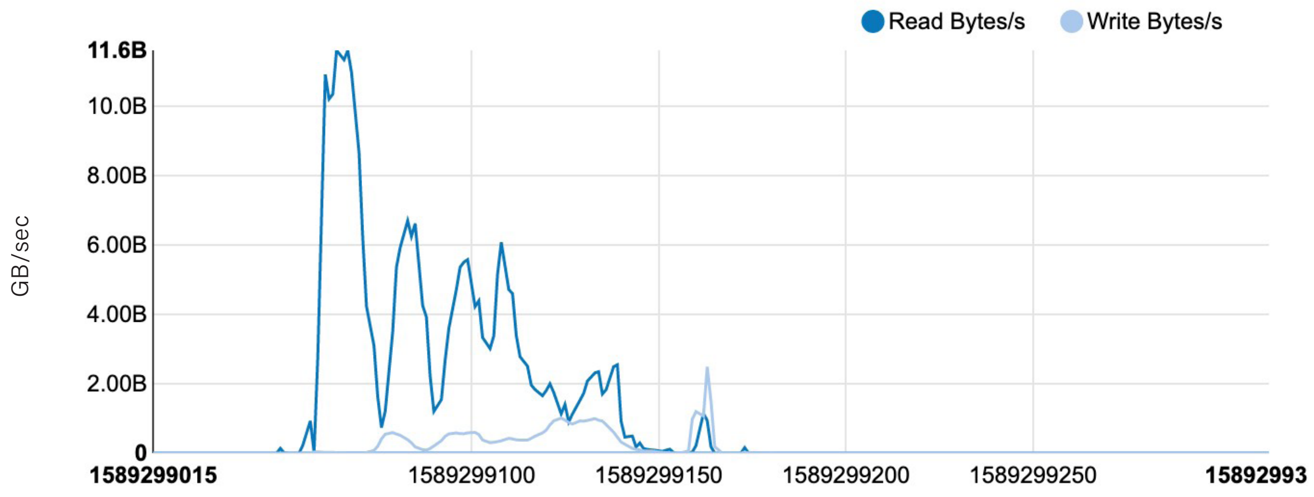
BENCHMARKS

WORDCOUNT

The following example wordcount was run from the Weka file system on a 300GB Wikipedia archive (https://engineering.purdue.edu/~puma/datasets.htm) which completed in 122 seconds:

```
time hadoop jar /home/hdfs/hadoop-mapreduce-examples-3.1.1.3.0.1.0-187.jar wordcount \
-Dmapreduce.map.log.level=INFO \
-Dmapreduce.reduce.log.level=INFO \
-Dyarn.app.mapreduce.am.log.level=INFO \
-Dio.file.buffer.size=131072 \
-Dmapreduce.map.cpu.vcores=4 \
-Dmapreduce.map.java.opts=-Xmx3276m \
-Dmapreduce.map.maxattempts=1 \
-Dmapreduce.map.memory.mb=4096 \
-Dmapreduce.map.output.compress=true \
-Dmapreduce.map.output.compress.codec=org.apache.hadoop.io.compress.Lz4Codec \
-Dmapreduce.reduce.cpu.vcores=2 \
-Dmapreduce.reduce.java.opts=-Xmx6553m \
-Dmapreduce.reduce.maxattempts=1 \
-Dmapreduce.reduce.memory.mb=8192 \
-Dmapreduce.task.io.sort.factor=300 \
-Dmapreduce.task.io.sort.mb=384 \
-Dyarn.app.mapreduce.am.command.opts=-Xmx6553m \
-Dmapreduce.input.fileinputformat.split.minsize=536870912 \
-Dyarn.app.mapreduce.am.resource.mb=4096 \
-Dmapred.reduce.tasks=200 \
 file:///mnt/weka/hdp/wiki/wikipedia_300GB/
 OUTPUT=file:///mnt/weka/hdp/wiki/ wikipedia_300GB_$RANDOM >> $RESULTSFILE 2>&1
```

The job completed in 126 seconds during which we saw a peak of 11.6GB/sec reads and a peak of just over 2GB/sec writes:

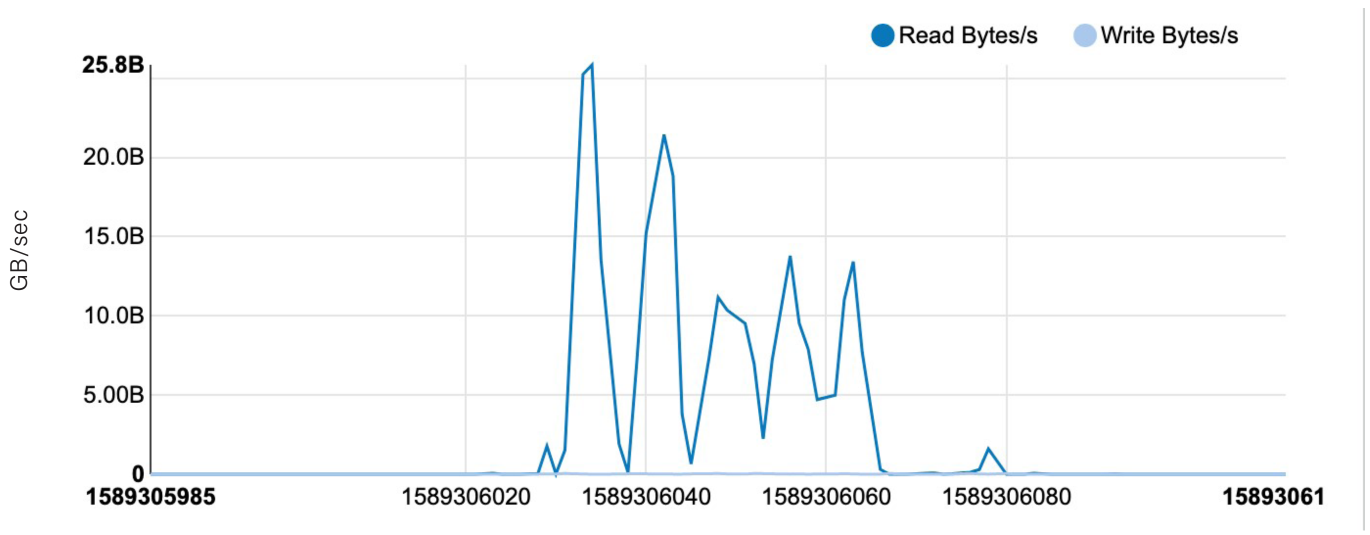


GREP MR

Using the same 300GB Wiki archive, the following MR grep was run against a non-existent string "f2fa8586f061b9fc72dc6c05b0b7d5cf6eb5b78b":

```
hadoop jar /home/hdfs/hadoop-mapreduce-examples-3.1.1.3.0.1.0-187.jar grep \
-Dmapreduce.map.log.level=INFO \
-Dmapreduce.reduce.log.level=INFO \
-Dyarn.app.mapreduce.am.log.level=INFO \
-Dio.file.buffer.size=131072 \
-Dmapreduce.map.cpu.vcores=4 \
-Dmapreduce.map.java.opts=-Xmx3276m \
-Dmapreduce.map.maxattempts=1 \
-Dmapreduce.map.memory.mb=8192 \
-Dmapreduce.map.output.compress=true \
-Dmapreduce.map.output.compress.codec=org.apache.hadoop.io.compress.Lz4Codec \
-Dmapreduce.reduce.cpu.vcores=2 \
-Dmapreduce.reduce.java.opts=-Xmx6553m \
-Dmapreduce.reduce.maxattempts=1 \
-Dmapreduce.reduce.memory.mb=4096 \
-Dmapreduce.task.io.sort.factor=300 \
-Dmapreduce.task.io.sort.mb=300 \
-Dyarn.app.mapreduce.am.command.opts=-Xmx6553m \
-Dmapreduce.input.fileinputformat.split.minsize=268435456 \
-Dyarn.app.mapreduce.am.resource.mb=4096 \
-Dmapred.reduce.tasks=200 \
file:///mnt/weka/hdp/wiki/wikipedia_300GB/
file:///mnt/weka/hdp/wiki/wikipedia_300GB_$RANDOM/
f2fa8586f061b9fc72dc6c05b0b7d5cf6eb5b78b >> $RESULTSFILE 2>&1
```

This job completed in 68 seconds and saw peak reads of 25GB/sec from the Weka storage:

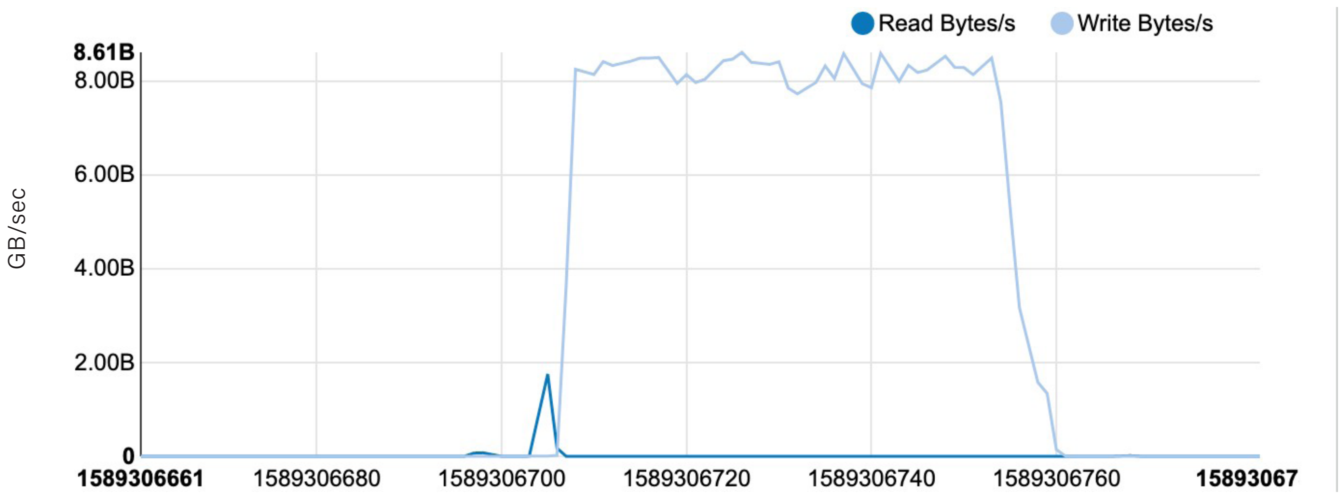


TERAGEN

Teragen was used to write 400GB of data to the Weka file system:

```
hadoop jar /home/hdfs/hadoop-mapreduce-examples-3.1.1.3.0.1.0-187.jar teragen \
-Dmapreduce.map.log.level=INFO \
-Dmapreduce.reduce.log.level=INFO \
-Dyarn.app.mapreduce.am.log.level=INFO \
-Dio.file.buffer.size=131072 \
-Dmapreduce.map.cpu.vcores=2 \
-Dmapreduce.map.java.opts=-Xmx3276m \
-Dmapreduce.map.maxattempts=1 \
-Dmapreduce.map.memory.mb=4096 \
-Dmapreduce.map.output.compress=true \
-Dmapreduce.map.output.compress.codec=org.apache.hadoop.io.compress.Lz4Codec \
-Dmapreduce.reduce.cpu.vcores=2 \
-Dmapreduce.reduce.java.opts=-Xmx6553m \
-Dmapreduce.reduce.maxattempts=1 \
-Dmapreduce.reduce.memory.mb=8192 \
-Dmapreduce.task.io.sort.factor=300 \
-Dmapreduce.task.io.sort.mb=384 \
-Dyarn.app.mapreduce.am.command.opts=-Xmx6553m \
-Dyarn.app.mapreduce.am.resource.mb=8192 \
-Dmapred.map.tasks=200 \
4000000000 400G >> $RESULTSFILE 2>&1
```

This job completed in 68 seconds and held a write throughput greater than 8GB/sec throughout:

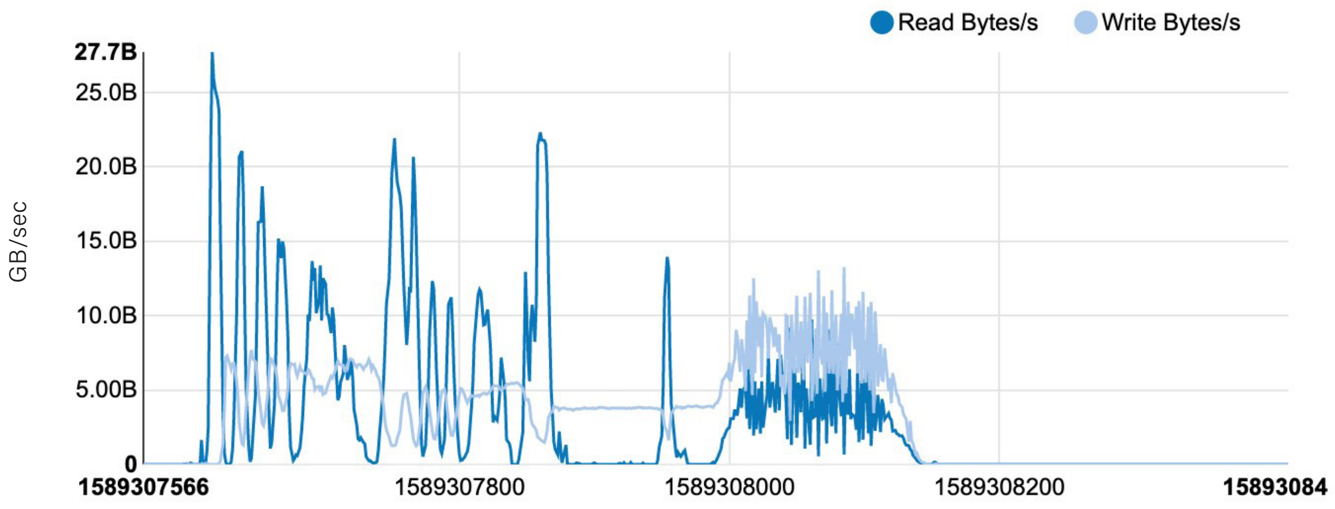


TERASORT

Terasort looks to be tightly coupled with HDFS (https://issues.apache.org/jira/browse/MAPREDUCE-5528) and we were unable to run it directly on Weka. However, we did run Terasort on HDFS running from Weka file systems:

```
hadoop jar /home/hdfs/hadoop-mapreduce-examples-3.1.1.3.0.1.0-187.jar terasort \
-Dmapreduce.map.log.level=INFO \
-Dmapreduce.reduce.log.level=INFO \
-Dyarn.app.mapreduce.am.log.level=INFO \
-Dio.file.buffer.size=131072 \
-Dmapreduce.map.cpu.vcores=4 \
-Dmapreduce.map.java.opts=-Xmx3276m \
-Dmapreduce.map.maxattempts=1 \
-Dmapreduce.map.memory.mb=4096 \
-Dmapreduce.map.output.compress=true \
-Dmapreduce.map.output.compress.codec=org.apache.hadoop.io.compress.Lz4Codec \
-Dmapreduce.reduce.cpu.vcores=2 \
-Dmapreduce.reduce.java.opts=-Xmx6553m \
-Dmapreduce.reduce.maxattempts=1 \
-Dmapreduce.reduce.memory.mb=8192 \
-Dmapreduce.task.io.sort.factor=300 \
-Dmapreduce.task.io.sort.mb=384 \
-Dyarn.app.mapreduce.am.command.opts=-Xmx6553m \
-Dmapreduce.input.fileinputformat.split.minsize=1073741824 \
-Dyarn.app.mapreduce.am.resource.mb=8192 \
-Dmapred.reduce.tasks=230 \
-Dmapreduce.terasort.output.replication=1 \
/home/hdfs/teragen/1T-terasort-input  /home/hdfs/teragen/1T-terasort-output  >>  $RESULTSFILE
2>&1
```

Terasort ran with 1TB of data and completed in 9 minutes and 8 seconds. This workload hit a peak of 27GB/sec reads and a peak of 12GB/sec writes.



SUMMARY

This document demonstrates that WekaFS provides very high throughput per nodes from a minimal 8 node cluster running on a shared virtualized network in AWS across a range of the Hadoop. Typically Weka sees an order of 2x performance improvement running on bare metal instances on industry standard hardware from vendors such as Supermicro, Dell or HPE.

- For Wordcount, WekaFS delivered 725MB/second per Hadoop node from a minimum cluster size of Weka on 8 nodes across AWS virtualized environment, for a peak of 11GB/sec read performance.
- For MapReduce GREP, WekaFS delivered in excess of 25GB/sec across the 16 Hadoop nodes, completing in 68 seconds.
- For TeraGen, WekaFS delivered 8GB/sec write performance across the 16 Hadoop nodes, also completing in 68 second.
- For TeraSort, due to tight coupling with HDFS, the benchmark was run on HDFS running on the Weka file system. The peak performance was ~28GB/second reads and 12GB/sec writes.

The Weka file system will deliver very efficient performance per storage node, ensuring that clients can complete tasks quickly. Leveraging a shared file system vs. typical Hadoop nodes, performance scaling can occur independently for the Hadoop nodes and the storage system without the need for tight coupling of the two.

Weka offers the additional ability to scale the data lake by extending the name space from the flash tier to object tier, allowing the data lake to scale to Petabytes, without the need to scale out the Hadoop cluster.

WEKA

910 E Hamilton Avenue, Suite 430, Campbell, CA 95008   T: 408.335.0085   E: info@weka.io   www.weka.io