MINIO

Performance comparison between MinIO and HDFS for MapReduce Workloads

AUGUST 2019

MapReduce Benchmark - HDFS vs MinIO

MinIO is a high-performance object storage server designed for disaggregated architectures. It is fully compatible with the Amazon S3 API.

HDFS is a high-throughput, fault-tolerant distributed file system designed for data locality. It is the default filesystem bundled with Apache Hadoop.

HDFS achieves high throughput by co-locating compute and data on the same nodes. This design overcomes the limitations of slow network access to data. However, it leads to significant operational challenges as the storage needs grow significantly faster than the compute needs.

Hadoop vendors limit the capacity per data node to a maximum of 100 TB and only support 4 TB or 8 TB capacity drives. For instance, in order to store 10 petabytes of data, 30 petabytes of physical storage is needed (3x replication). This would require a minimum of 300 nodes. This overprovisioning of compute resources results in wastage and operational overhead.

The solution to this problem is to disaggregate storage and compute, so that they can be scaled independently. Object storage utilizes denser storage servers such as the Cisco UCS S3260 Storage Server or the Seagate Exos AP 4U100 that can host more than a petabyte of usable capacity per server and 100 GbE network cards. Compute nodes, on the other hand, are optimized for MEM and GPU intensive workloads. This architecture is a natural fit for cloud-native infrastructure where the software stack and the data pipelines are managed elastically via Kubernetes.

While cloud-native infrastructure is scalable and easier to manage, it is important to understand the performance difference between the two architectures. This document compares the performance of Hadoop HDFS and MinIO using the most proven Hadoop benchmarks: Terasort, Sort and Wordcount. The results demonstrate that object storage is on par with HDFS in terms of performance - and makes a clear case for disaggregated Hadoop architecture.

Benchmark	HDFS	MinIO	MinIO is X % faster
Terasort	1005s	820s	22.5%
Sort	1573s	793s	98.3%
Wordcount	1100s	787s	39.7%

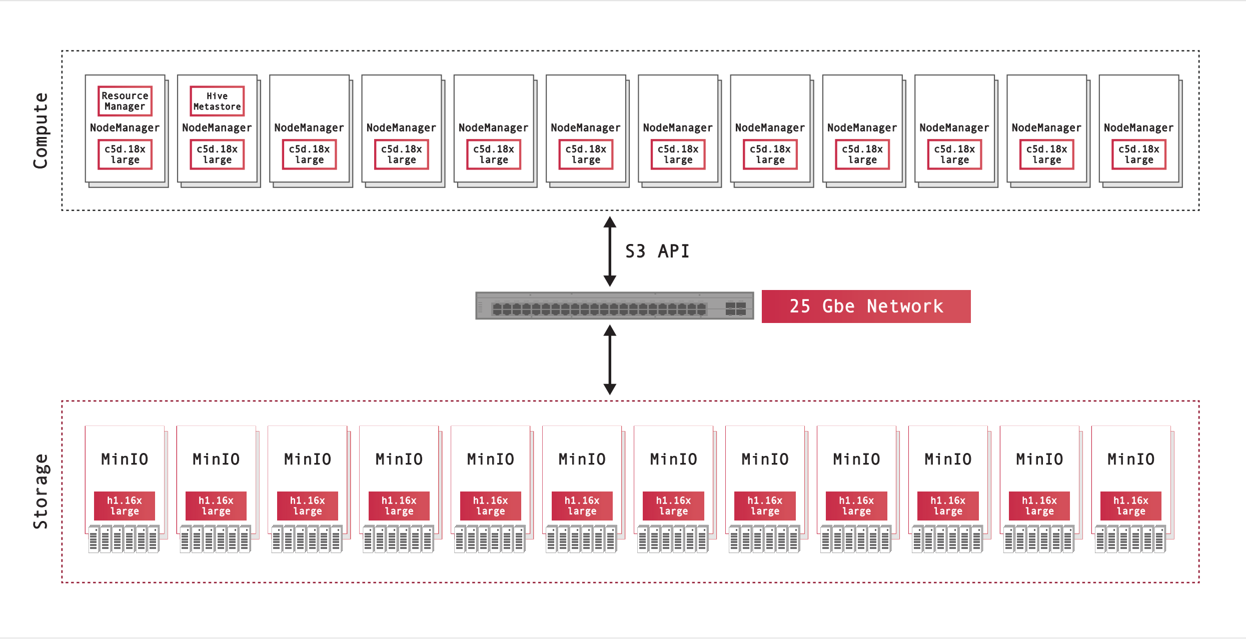
1. Benchmark Environment

1.1 Hardware for MinIO - Disaggregated compute and storage

These benchmarks were performed on AWS bare-metal storage-optimized instances (h1.16xlarge) with local hard disk drives and 25 GbE networking. These nodes were chosen for their drive capabilities, and its MEM and CPU resources were underutilized by MinIO.

The compute jobs ran on compute-optimized instances (c5d.18xlarge) connected to storage by 25GbE networking.

Instance	# Nodes	AWS Instance type	CPU	MEM	Storage	Network
Compute Nodes	12	c5d.18xlarge	72	144 GB	2 x 900 GB	25 Gbps
Storage Nodes	12	h1.16xlarge	64	256 GB	8 x 2 TB	25 Gbps

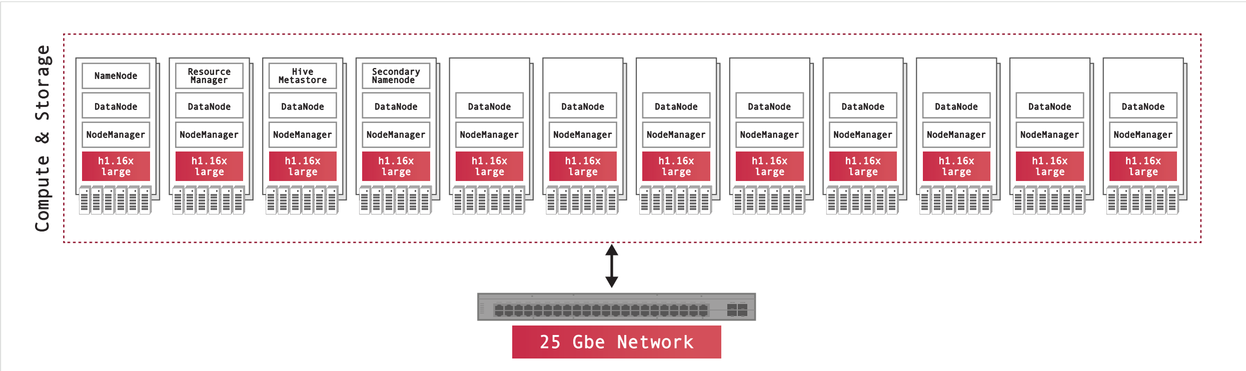


Disaggregated storage and compute architecture for MinIO

1.2 Hardware for Hadoop HDFS - Colocated compute and storage

These benchmarks were performed on AWS bare-metal instances (h1.16xlarge) with local hard disk drives and 25 GbE networking. MapReduce on HDFS has the advantage of data locality and 2x the amount of memory (2.4 TB).

Instance	# Nodes	AWS Instance type	CPU	MEM	Storage	Network
Hadoop Nodes	12	h1.16xlarge	64	256 GB	8 x 2 TB	25 Gbps



Colocated storage and compute architecture for Hadoop HDFS

1.3 Measuring Raw Hardware Performance

Hard Drive Performance

The performance of each drive was measured using the command dd.

Below is the output of a single HDD drive's **write** performance with 16MB block-size using the O_DIRECT option and a total count of 64:

```
$ dd if=/dev/zero of=/mnt/drive/test bs=16M count=64 oflag=direct
64+0 records in
64+0 records out
1073741824 bytes (1.1 GB, 1.0 GiB) copied, 7.83377 s, 137 MB/s
```

Below is the output of a single HDD drive's **read** performance with 16MB block-size using the O_DIRECT option and a total count of 64:

```
$ dd of=/dev/null if=/mnt/drive/test bs=16M count=64 iflag=direct
64+0 records in
64+0 records out
1073741824 bytes (1.1 GB, 1.0 GiB) copied, 5.23199 s, 205 MB/s
```

JBOD Performance

JBOD performance with O_DIRECT was measured using iozone. Iozone is a filesystem benchmark tool that generates and measures filesystem performance for read, write and other operations.

Here is the iozone command to test multi-drive performance with 32 parallel threads, 32KB block-size and the O_DIRECT option.

```
$ iozone -t 32 -I -r 32K -s 256M -F /mnt/drive{1..8}/tmp{1..4}
Num drives: 8
Write: 655 MB/sec
Read: 1.26 GB/sec
```

Network Performance

All of the servers used 25GbE networking for both client and internode communications. This network hardware provides a maximum of 25 Gbit/sec, which equates to 3.125 GB/sec.

```
$ iperf3 -c minio-2 -P 10
[SUM]   0.00-10.00  sec  28.9 GBytes  24.8 Gbits/sec         sender
[SUM]   0.00-10.00  sec  28.8 GBytes  24.7 Gbits/sec         receiver
```

Estimating Server Performance

The maximum server performance is limited by the network or drives, whichever is slower. The network performance sustains at 3.125GB/sec, but the JBOD sustained throughput is only 1.26 GB/sec. Therefore, the projected maximum server performance is 15.12 GB/sec (1.26 GB/sec x 12 servers).

1.4. Software

Property	Value
Hadoop	Hortonworks HDP: 3.1.0-78-4
MinIO	Minio-RELEASE.2019-07-24T02-02-23Z
Benchmark	Terasort | Sort | Wordcount | 1 TB
Server OS	Amazon Linux 2
S3 Connector	S3A
S3 Output Committer	Directory Staging Committer

1.4 Performance Tuning

The following `sysctl` values were set on all nodes in the environment for both MinIO and Hadoop HDFS nodes:

```
# maximum number of open files/file descriptors
fs.file-max = 4194303

# use as little swap space as possible
vm.swappiness = 1

# prioritize application RAM against disk/swap cache
vm.vfs_cache_pressure = 10

# minimum free memory
vm.min_free_kbytes = 1000000

# maximum receive socket buffer (bytes)
net.core.rmem_max = 268435456

# maximum send buffer socket buffer (bytes)
net.core.wmem_max = 268435456

# default receive buffer socket size (bytes)
net.core.rmem_default = 67108864
```

```
# default send buffer socket size (bytes)
net.core.wmem_default = 67108864

# maximum number of packets in one poll cycle
net.core.netdev_budget = 1200
# maximum ancillary buffer size per socket
net.core.optmem_max = 134217728

# maximum number of incoming connections
net.core.somaxconn = 65535

# maximum number of packets queued
net.core.netdev_max_backlog = 250000

# maximum read buffer space
net.ipv4.tcp_rmem = 67108864 134217728 268435456

# maximum write buffer space
net.ipv4.tcp_wmem = 67108864 134217728 268435456

# enable low latency mode
net.ipv4.tcp_low_latency = 1

# socket buffer portion used for TCP window
net.ipv4.tcp_adv_win_scale = 1

# queue length of completely established sockets waiting for accept
net.ipv4.tcp_max_syn_backlog = 30000

# maximum number of sockets in TIME_WAIT state
net.ipv4.tcp_max_tw_buckets = 2000000

# reuse sockets in TIME_WAIT state when safe
net.ipv4.tcp_tw_reuse = 1

# time to wait (seconds) for FIN packet
net.ipv4.tcp_fin_timeout = 5

# disable icmp send redirects
net.ipv4.conf.all.send_redirects = 0

# disable icmp accept redirect
net.ipv4.conf.all.accept_redirects = 0

# drop packets with LSR or SSR
net.ipv4.conf.all.accept_source_route = 0

# MTU discovery, only enable when ICMP blackhole detected
net.ipv4.tcp_mtu_probing = 1
```

In addition, the hard and soft limits on the maximum number of open files were set to 65,535.

```
$ echo "* hard nofile 65535" >>  /etc/security/limits.conf
$ echo "* soft nofile 65535" >>  /etc/security/limits.conf
```

MinIO binary was downloaded onto each server node, and started using the following commands:

```
$ export MINIO_STORAGE_CLASS_STANDARD=EC:4 # 4 parity 12 data
$ export MINIO_ACCESS_KEY=minio
$ export MINIO_SECRET_KEY=minio123
$ minio server http://minio-{1...12}/mnt/drive{1...8}/minio
```

MapReduce Performance Tuning for MinIO

MapReduce was configured to utilize 1.2 TB of aggregate memory across 12 compute nodes using the following settings:

```
$ cat $HADOOP_CONF_DIR/core-site.xml | kv-pairify | grep ‘java.opts’
mapreduce.map.java.opts=-Xmx104000m    # 104 GB per node for mappers
mapreduce.reduce.java.opts=-Xmx104000m # 104 GB per node for reducers
```

KV-parify is a simple script to allow easy grepping of Hadoop xml files using `xq` (XML query) and `jq` (JSON query).

```
$ alias kv-pairify = ‘xq “.configuration[]” |
                      jq “.[]” |
                      jq -r “.name + \"=\" + .value”’
```

In addition to the above parameters, the system was tuned to ensure that the MapReduce jobs could utilize the entire capacity of CPU and memory provided by the compute nodes.

The following parameters were tuned until a point was reached where the entirety of the 144GB of RAM was being utilized on all compute nodes, while also ensuring that this did not lead to swapping.

```
$ cat $HADOOP_CONF_DIR/core-site.xml | kv-pairify | grep 'mapred'
mapred.job.reuse.jvm.num.tasks=15       # reuse upto 15 JVMs
mapred.maxthreads.generate.mapoutput=2  # num threads to write map outputs
mapred.maxthreads.partition.closer=0    # asynchronous map flushers
mapred.min.split.size=2560000           # minimum split size
mapreduce.fileoutputcommitter.algorithm.version=2
mapreduce.job.reduce.slowstart.completedmaps=0.99  # 99% map, then reduce
mapreduce.map.java.opts=-Xmx104000m     # 104 GB per node for mappers
mapreduce.map.speculative=false         # disable speculation for mapping
mapreduce.reduce.java.opts=-Xmx104000m  # 104 GB per node for reducers
mapreduce.reduce.maxattempts=1          # do not retry on failure
mapreduce.reduce.merge.inmem.threshold=10000      # minimum # merges in RAM
mapreduce.reduce.shuffle.input.buffer.percent=0.9 # min % buffer in RAM
mapreduce.reduce.shuffle.merge.percent=0.9        # minimum % merges in RAM
mapreduce.reduce.shuffle.parallelcopies=3840      # num of parition copies
mapreduce.reduce.speculative=false     # disable speculation for reducing
mapreduce.task.io.sort.factor=999      # threshold before writing to disk
mapreduce.task.io.sort.mb=1000         # memory for sorting
mapreduce.task.sort.spill.percent=0.9  # minimum % before spilling to disk
mapreduce.task.timeout=1200000         # map/reduce task timeout
mapreduce.tasktracker.reserved.physicalmemory.mb.low=0.95
```

The master and slave components of YARN shared the hardware resources on the 12 compute nodes (c5d.18xlarge).

MapReduce Performance Tuning for HDFS

MapReduce was configured to utilize 2.4 TB of aggregate memory across 12 storage nodes using the following settings:

```
$ cat $HADOOP_CONF_DIR/core-site.xml | kv-pairify | grep 'java.opts'
mapreduce.map.java.opts=-Xmx204000m    # 204 GB per node for mappers
mapreduce.reduce.java.opts=-Xmx204000m # 204 GB per node for reducers
```

In addition to the above parameters, the system was tuned to ensure that the MapReduce jobs could utilize the entire capacity of CPU and memory provided by the compute nodes.

The following parameters were tuned until a point where the entirety of the 256GB of RAM was being utilized on all compute nodes, while also ensuring that this did not lead to swapping.

```
$ cat $HADOOP_CONF_DIR/core-site.xml | kv-pairify | grep 'mapred'
mapred.job.reuse.jvm.num.tasks=15       # reuse upto 15 JVMs
mapred.maxthreads.generate.mapoutput=2  # num threads to write map outputs
mapred.maxthreads.partition.closer=0    # asynchronous map flushers
mapred.min.split.size=2560000           # minimum split size
mapreduce.fileoutputcommitter.algorithm.version=2
mapreduce.job.reduce.slowstart.completedmaps=0.99  # 99% map, then reduce
mapreduce.map.java.opts=-Xmx204000m     # 104 GB per node for mappers
mapreduce.map.speculative=false         # disable speculation for mapping
mapreduce.reduce.java.opts=-Xmx204000m  # 104 GB per node for reducers
mapreduce.reduce.maxattempts=1          # do not retry on failure
mapreduce.reduce.merge.inmem.threshold=10000      # minimum # merges in RAM
mapreduce.reduce.shuffle.input.buffer.percent=0.9 # min % buffer in RAM
mapreduce.reduce.shuffle.merge.percent=0.9        # minimum % merges in RAM
mapreduce.reduce.shuffle.parallelcopies=3840       # num of parition copies
mapreduce.reduce.speculative=false     # disable speculation for reducing
mapreduce.task.io.sort.factor=999      # threshold before writing to disk
mapreduce.task.io.sort.mb=1000         # memory for sorting
mapreduce.task.sort.spill.percent=0.9  # minimum % before spilling to disk
mapreduce.task.timeout=1200000         # map/reduce task timeout
mapreduce.tasktracker.reserved.physicalmemory.mb.low=0.95
```

The master and slave components of Hadoop, HDFS and YARN shared the hardware resources on the 12 storage nodes (h1.16xlarge).

HDFS Performance Tuning

HDFS was configured to replicate data with replication factor set to 3:

```
$ cat $HADOOP_CONF_DIR/hdfs-site.xml | kv-pairify | grep 'replication'
dfs.namenode.replication.min=3     # minimum replicas before write success
dfs.namenode.maintenance.replication.min=3 # same as above, for maintenance
dfs.replication=3                          # replication factor
```

The master and slave components of Hadoop, HDFS and YARN shared the hardware resources on the 12 storage nodes (h1.16xlarge).

S3A Performance Tuning

S3A is the connector to use S3 and other S3-compatible object stores such as MinIO. MapReduce workloads typically interact with object stores in the same way they do with HDFS.

These workloads rely on HDFS's atomic rename functionality to complete writing data to the

datastore. Object storage operations are atomic by nature and they do not require/implement rename API. The default S3A committer emulates renames through copy and delete APIs. This interaction pattern, however, causes significant loss of performance because of the write amplification.

Netflix, for example, developed two new staging committers - the **Directory** staging committer and the **Partitioned** staging committer - to take full advantage of native object storage operations. These committers do not require rename operation.

The two staging committers were evaluated, along with another new addition called the **Magic** committer for benchmarking. It was found that the directory staging committer was the fastest among the three.

The S3A connector was configured with the following parameters:

```
$ cat $HADOOP_CONF_DIR/core-site.xml | kv-pairify | grep 's3a'
fs.s3a.access.key=minio
fs.s3a.secret.key=minio123
fs.s3a.path.style.access=true
fs.s3a.block.size=512M
fs.s3a.buffer.dir=${hadoop.tmp.dir}/s3a
fs.s3a.committer.magic.enabled=false
fs.s3a.committer.name=directory
fs.s3a.committer.staging.abort.pending.uploads=true
fs.s3a.committer.staging.conflict-mode=append
fs.s3a.committer.staging.tmp.path=/tmp/staging
fs.s3a.committer.staging.unique-filenames=true
fs.s3a.committer.threads=2048  # number of threads writing to MinIO
fs.s3a.connection.establish.timeout=5000
fs.s3a.connection.maximum=8192 # maximum number of concurrent conns
fs.s3a.connection.ssl.enabled=false
fs.s3a.connection.timeout=200000
fs.s3a.endpoint=http://minio:9000
fs.s3a.fast.upload.active.blocks=2048 # number of parallel uploads
fs.s3a.fast.upload.buffer=disk        # use disk as the buffer for uploads
fs.s3a.fast.upload=true               # turn on fast upload mode
fs.s3a.impl=org.apache.hadoop.fs.s3a.S3AFileSystem
fs.s3a.max.total.tasks=2048      # maximum number of parallel tasks
fs.s3a.multipart.size=512M       # size of each multipart chunk
fs.s3a.multipart.threshold=512M  # size before using multipart uploads
fs.s3a.socket.recv.buffer=65536  # read socket buffer hint
fs.s3a.socket.send.buffer=65536  # write socket buffer hint
fs.s3a.threads.max=2048          # maximum number of threads for S3A
```

Additional Considerations

Terasort distributed with all the major Hadoop distros as of August 2, 2019 and is hardcoded to only support FileOutputCommitter. It was made configurable to use S3A committers for our benchmarking purposes.

It was found that this fix was already contributed upstream on March 21st 2019. The details of this fix are captured in this JIRA issue (https://issues.apache.org/jira/browse/MAPREDUCE-7091).

Sort and Wordcount benchmarks did not require any changes.

2. Benchmark Results

Benchmarking was divided into two phases: data generation and benchmarking tests.

Data Generation

In this phase, the data for the appropriate benchmarks were generated. Even though this step is not performance-critical, it was still evaluated to assess the differences between MinIO and HDFS.

Benchmark	HDFS	MinIO
Terasort	294s	620s
Sort/Wordcount	365s	680s

Note that the data generated for the Sort benchmark can be used for Wordcount and vice-versa.

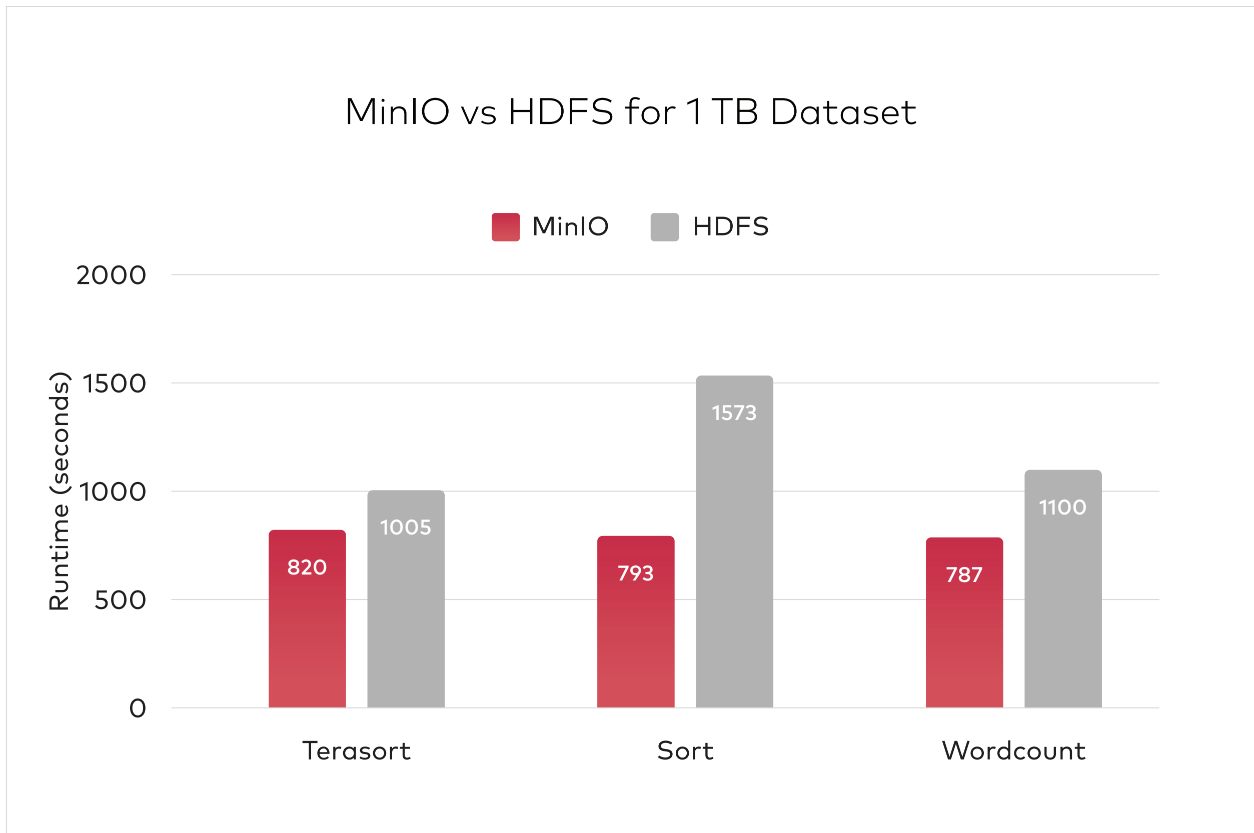
In the case of Terasort, the HDFS generation step performed 2.1x faster than MinIO. In the case of Sort and Wordcount, the HDFS generation step performed 1.9x faster than MinIO.

During the generation phase, the S3 staging committers were at a disadvantage, as the committers stage the data in RAM or disk and then upload to MinIO. In the case of HDFS and the S3A magic committer, the staging penalty does not exist.

Despite the disadvantage during the generation phase, the Directory staging committer showed significantly better for both read- and write-intensive workloads because the staging phase is significantly smaller compared to the overall CRUD phase.

Benchmarking Tests

The results of the benchmarking phase is presented below:



smaller values indicate higher performance

Benchmark	HDFS	MinIO	MinIO is X % faster
Terasort	1005s	820s	22.5%
Sort	1573s	793s	98.3%
Wordcount	1100s	787s	39.7%

In this phase, Terasort ran faster with MinIO than HDFS. The difference between their runtimes is 22.5%. In case of Sort and WordCount, the percentage difference between HDFS and MinIO was 98.3% and 39.7% respectively. In all cases, MinIO in disaggregated architecture was observed to be more performant.

Staging committer is recommended for all real world workloads in disaggregated mode.

3. Conclusion

The results conclude that colocating compute and storage is no longer an advantage from a performance or operational point of view. Given that today's drives are denser and networking is orders of magnitude faster, this makes a clear case for Hadoop stack to shift from HDFS to MinIO.