Hadoop Performance Tuning - A Pragmatic & Iterative Approach

Dominique Heger
DH Technologies

Introduction

Hadoop represents a Java-based distributed computing framework that is designed to support applications that are implemented via the MapReduce programming model. In general, workload dependent Hadoop performance optimization efforts have to focus on 3 major categories: the systems HW, the systems SW, and the configuration and tuning/optimization of the Hadoop infrastructure components. From a systems HW perspective, it is paramount to balance the appropriate HW components in regards to performance, scalability, and cost. It has to be pointed out that Hadoop is classified as a highly-scalable, but not necessarily as a high-performance cluster solution. From a SW perspective, the choice of the OS, the JVM, the specific Hadoop version, as well as other SW components necessary to run the Hadoop setup do have a profound impact on performance and stability of the environment. The design, setup, configuration, and tuning phase of any Hadoop project is paramount to fully benefit from the distributed Hadoop HW and SW solution stack.

A typical Hadoop cluster consists of an *n*-level architecture that is comprised of rack-mounted server systems. Each rack is typically interconnected via some (as an example GbE) switch while channel bonding may have to be considered depending on the workload. Each rack-level switch may be connected to a cluster-level switch, which typically represents a larger port-density 10GbE switch. Those cluster-level switches may also interconnect with other cluster-level switches, or may be uplinked to another level of switching infrastructure. From a functional perspective, the Hadoop server systems are categorized as
(1) JobTracker system that performs task assignment,
(2) NameNode that maintains all file system metadata (if the Hadoop Distributed File System is used). Preferably (but not required), the NameNode should represent a separate physical server, and should not be bundled with the JobTracker.
(3) Secondary NameNode that periodically check-points the file system metadata on the NameNode.
(4) TaskTracker nodes that perform the MapReduce tasks.
(5) DataNode systems that store HDFS files and handle the HDFS read/write requests. It is suggested to co-located the DataNode systems with the TaskTracker nodes to assure optimal data locality. Please see [HEGE2012] for a more detailed introduction to the Hadoop architecture.

One of the most important design decisions to be made while planning for a Hadoop infrastructure deployment is the number, type, and setup/configuration of the server nodes in the cluster. As with any other IT configuration, the workload dependent Hadoop application threads may be CPU, memory, and/or IO bound. For a lot of

Hadoop installations, contemporary dual-socket server systems are used. These server systems are normally preferred to larger SMP nodes from a cost-benefit perspective, and do provide good load-balancing and thread concurrency perspectives. For most Hadoop installations (workload dependent), configuring a few hard drives per server node is sufficient. For Hadoop workloads that are highly IO intensive, it may be necessary to configure higher ratios of disks to available cores. Most Hadoop installation use >= 1TB drives. While it would be feasible to use RAID solutions, the usage of RAID systems with Hadoop servers is generally not recommended, as redundancy is build into the HDFS framework (via replicating blocks across multiple nodes). Sufficient memory capacity per server node is paramount, as executing concurrent MapReduce tasks at high throughput rates is required to achieve good aggregate cluster performance behavior. From an OS perspective, it is suggested to run Hadoop on a contemporary Linux kernel. Newer Linux kernels provide a much improved threading behavior and are more energy efficient. With large Hadoop clusters, any power inefficiency amounts to significant (unnecessary) energy costs.

Opportunities and Challenges

Hadoop is considered a large and complex SW framework that incorporates a number of components that interact among each other across multiple HW systems. Bottlenecks in a subset of the HW can cause overall performance issues for any Hadoop workload. Hadoop performance is sensitive to every component of the stack, including Hadoop/HDFS, JVM, OS, NW, the underlying HW, as well as possibly the BIOS settings. Every Hadoop version is distributed with a VERY large set of configuration parameters, and a rather large subset of these parameters can potentially impact performance. It has to be pointed out though that one cannot optimize a HW/SW infrastructure if one does not understand the internals and interrelationships of the HW/SW components. Or in other words, one cannot tune what one does not understand, and one cannot improve what one cannot measure. Hence, adjusting these configuration parameters to optimize performance requires the knowledge of the internal working of the Hadoop framework. As with any other SW system, some parameters impact other parameter values. Hence, it is paramount to use a pragmatic/iterative process, as well as several measurement tools, to tune a Hadoop environment. The approach outlined in this paper is based on an actual tuning cycle where Hadoop and Linux workload generators are used, data is collected and analyzed, and potential bottlenecks are identified and addressed until the Hadoop cluster meets/exceeds expectations. It is paramount to not only address the HW or the OS, but also to scrutinize the actual MapReduce code and potentially re-write some of the procedures to improve the aggregate performance behavior.

Monitoring and Profiling Tools

Some of the tools used to monitor Hadoop jobs include:

- Ganglia and Nagios represent distributed monitoring systems that capture/report various system performance statistics such as CPU utilization, NW utilization,

memory usage, or load on the cluster. These tools are effective in monitoring the overall health of the cluster

- The Hadoop task and job logs capture very useful systems counters and debug information that aid in understanding and diagnosing job level performance bottlenecks

- Linux OS utilities such as *dstat*, *top*, *htop, iotop*, *vmstat*, *iostat*, *sar*, or *netstat* aid in capturing system-level performance statistics. This data is used to study how different resources of the cluster are being utilized by the Hadoop jobs, and which resources may be under contention

- Java Profilers (such as *Hprof*) should be used to identify and analyze Java hot-spots

- System level profilers such as Linux *perf* or *strace* should be used to conduct a deep-dive analysis to identify potential performance bottlenecks induced by SW/HW events.

Methodology

The tuning methodologies and recommendations presented in this paper are based on experience made by designing and tuning Hadoop systems for varying workload conditions. The tuning recommendations made here are based on optimizing the Hadoop benchmark *TeraSort* workload. Other Hadoop workloads may require different/additional tuning. Nevertheless, the methodology presented here should be universally applicable to any Hadoop performance project. The base configuration used for this study consisted of a Hadoop (1.0.2) cluster consisting of 8 Ubuntu 12.10 server nodes that were all equipped with 8 cores (Nehalem 2.67GHz), 24GB RAM (1333 MHz), and hence provided 3GB of memory per core. The setup consisted of 7 slave nodes (DataNodes, TaskTrackers) and 1 master node (NameNode, SecondaryNameNode, JobTracker). Each data node was configured with 6 1TB hard disks (10,000RPM) in a JBOD (Just a Bunch Of Disks) setup. The interconnect consisted of a GbE (switched) network.

It is always paramount to assure that performance oriented SW engineering practices are applied while implementing the Hadoop jobs. Some initial capacity planning has to be done to determine the base HW requirements. After the cluster is designed and setup, it is necessary to test if the Hadoop jobs are *functional*. The following sanity/stability related steps are suggested prior to any deep-dive action into the actual tuning phase of any Hadoop cluster:

- Verify that the HW components of the cluster are configured correctly
- If necessary, upgrade any SW components of the Hadoop stack to the latest stable version
- Perform burn-in/stress test simulations at the Linux level

- Tweak OS and Hadoop configuration parameters that may hamper Hadoop jobs *to complete*

Correctness of the HW Setup

Apart from the configuration of the HW systems, the BIOS, firmware and device drivers, as well as the memory DIMM configuration may have a noticeable performance impact on the Hadoop cluster. To verify the correctness of the HW setup, the suggestion made is to follow the guidelines provided by the HW manufacturers. This is a very important first step for configuring an optimized Hadoop cluster. The default settings for certain performance related architectural features are controlled by the BIOS. Bug fixes and improvements are made available via new BIOS releases. Hence, it is recommended to always upgrade the systems BIOS to the latest stable version. Systems log messages have to be monitored during the cluster burn-in and stress-testing stage to rule out any potential HW issues. Upgrading to the latest (stable) firmware and device driver levels addresses performance as well as stability issues. Optimal IO subsystem performance is paramount to any Hadoop workload. Faulty hard drives identified in this process have to be replaced. Appropriate systems memory configuration is necessary (check the DIMM setup/speed). Confirming the baseline performance of the CPU, memory, IO, and NW subsystems (via benchmarks) is paramount prior to conducting any actual Hadoop performance analysis. To summarize, the following steps are required to correctly setup the Hadoop cluster hardware infrastructure:

- Follow manufacturer's guidelines on installing and configuring HW components of the cluster
- Establish detailed HW profiles (describe the performance potential of the Hadoop cluster)
- Upgrade to the latest (stable) BIOS, firmware, and device driver levels
- Perform benchmark tests to verify baseline performance of all the systems subsystems (see below)

Upgrade SW Components

The Linux OS distribution level, the Hadoop distribution level, the JDK version, and the version of any other 3d-party libraries that comprise the Hadoop framework impact the performance of the Hadoop cluster. Hence, installing the latest (stable) SW components of the Hadoop stack (that allow the execution of the Hadoop jobs) is recommended. Performance and stability enhancements, as well as bug fixes released with newer Linux distributions may improve Hadoop performance. Linux kernel functionality is improved on an on-going basis, and components such as the file systems or the networking stack, which play an important role in Hadoop performance, are continuously improved in this process. The Hadoop environment is evolving all the time as well. A number of bug fixes and performance enhancements are integrated into basically every new release. So the suggestion made is to upgrade to the latest stable

versions of Hadoop and the JVM that allows the correct functioning of the Hadoop jobs. To summarize:

- Use the latest (stable) Linux distribution that allows for the correct functioning of the Hadoop jobs
- Use the latest (stable) Hadoop distribution for the Hadoop workload at hand
- Use the latest (stable) JVM and 3d-party libraries that the underlying Hadoop workload depends on

Baseline Performance Stress-Test

Stress testing the different subsystems of a Hadoop cluster prior to moving any Hadoop jobs into production is paramount to help uncover any potential performance bottlenecks. These tests/benchmarks are also used to establish/verify the baseline performance of the different subsystems of the cluster. In general, in a 1st phase, non-Hadoop benchmarks (Linux CPU, memory, IO, and NW benchmarks) are used, while in a 2nd phase, actual Hadoop benchmarks are executed. While running these benchmarks, the log files are monitored to identify any potential cluster level performance bottlenecks. Following is a list of some of the benchmarks that aid in performing these tests:

- Linux micro and macro benchmarks such *DHTUX*, *FFSB*, *STREAM*, *IOzone*, or *Netperf* can be used to establish the cluster node and interconnect performance baseline
- Hadoop micro-benchmarks such as *TestDFSIO*, *NNBench*, *TeraSort*, or *MRBench* can be used to stress-test the setup of the Hadoop framework (these micro-benchmarks are part of the Hadoop distributions).

Depending on the nature of the Hadoop workload, some default values of certain OS and Hadoop parameters may cause MapReduce task and/or Hadoop job failures, or may contribute to noticeable performance regressions. Hence, from a baseline perspective, for most Hadoop workloads, the following configuration parameters may be of interest:

Initial OS Parameters:

- The default maximum number of open file descriptors (FD) (as configured by using *ulimit*) may cause the FD's to be exhausted depending on the nature of the Hadoop workload. This may trigger exceptions that lead to job failures. Increasing the FD value as a baseline configuration is normally suggested.
- Fetch failure scenarios may occur while running Hadoop jobs with default settings. In some cases, these failures are due to a low value of the *net.core.somaxconn* Linux kernel parameter. The default value equals to 128, and may have to be increased to 512 or 1,024

Initial Hadoop Parameters:

- Depending on the nature of the Hadoop workload, the MapReduce tasks may not indicate any progress for a period of time that can exceed *mapred.task.timeout* (set in *mapred-site.xml*). In some Hadoop distributions, this value is set to 600 seconds by default. If necessary, the value may have to be increased based on the workload requirements. It has to be pointed out though that if there are actual task hang-ups due to HW, cluster setup, or workload implementation issues, increasing the value may mask some actual Hadoop cluster issue
- If *java.net.SocketTimeoutException* exceptions are encountered (check error logs), the *dfs.socket.timeout* and *dfs.datanode.socket.write.timeout* values (in *hdfs-site.xml*) have to be increased. As above, it is paramount to determine though if the change is necessary due to the actual workload, or if some underlying HW issue is causing the exceptions
- The replication factor for each block of an HDFS file (as specified by *dfs.replication*) is typically set to 3 (for fault tolerance). It is normally not suggested to set the parameter to a smaller value. To reiterate, RAID systems are normally not deployed with Hadoop clusters.

Performance Tuning

After the initial HW and SW (setup) components of the cluster are verified and determined to be operating at the currently best possible performance level, a deep dive into fine-tuning the actual workload onto the logical and physical resources can be commenced. As already discussed, parameters at all levels of the Hadoop stack do impact aggregate Hadoop application performance. The next few paragraphs discuss how to establish a performance baseline for a particular Hadoop workload, and how to utilize tuning to achieve a maximum level of resource utilization and performance. For this study, after executing *DHTUX* and *FFSB* on the Hadoop cluster nodes and hence, assuring that no faulty HW components or logical Linux OS resources are hampering performance, the Hadoop *TeraSort* benchmark was used as the workload generator. Without compression, the Map phase of the TeraSort workload processes 1TB of read and 1TB of write IO operations, respectively (excluding any spill related IO operations). The Reduce phase also performs 1TB of read and 1TB of write IO operations (excluding any spill related IO operations). Overall, the TeraSort benchmark is considered as being IO intensive.

Performance Baseline Setup

The default number of MapReduce slots, as well as the default Java heap size configuration, is normally not sufficient for most Hadoop workloads. Therefore, the first step while establishing the performance baseline is to scrutinize and potentially adjust these configuration parameters. The goal is to maximize the HW resource utilization (including the CPU's) of the cluster. The Java heap size requirements, the number of MapReduce task, the number of HW cores, the IO bandwidth, as well as the amount of

available RAM impact the baseline setup. A methodology that aids in the setup is focused on:

1. Configure enough disk space to accommodate the anticipated storage requirements
2. Configure sufficient Map and Reduce slots to maximize CPU utilization
3. Configure the Java heap size (for Map and Reduce JVM processes) so that ample memory is still available for the OS kernel, buffers, and cache subsystems

The corresponding Hadoop configuration parameters are:

- *mapred.map.tasks,*
- *mapred.tasktracker.map.tasks.maximum,*
- *mapred.reduce.tasks,*
- *mapred.tasktracker.reduce.tasks.maximum,*
- *mapred.map.child.java.opts,*
- *mapred.reduce.child.java.opts*

and are located in *mapred-site.xml*. For larger Hadoop clusters, the rule of thumb is to set the maximum number of Map and Reduce tasks that execute simultaneously on a TaskTracker in the range between [cores per node / 2] and [cores per node * 2]. Further, it is suggested to assure that the number of input streams (files) to be merged at once via the MapReduce tasks is appropriately configured for the workload at hand. The corresponding parameter is labeled *io.sort.factor*, and depending on the Hadoop workload, the value may have to be increased to a sufficiently large number. Another rule of thumb is that each Map task should run for a few minutes. Short running Map tasks encounter too much startup overhead and are not efficient in the shuffle phase (during the shuffle phase, MapReduce partitions data among the various Reducers). On the other hand, very long running Map tasks limit/hamper cluster parallelism and cluster sharing.

Utilizing the above discussed approach, the following baseline configuration was setup:

- Configuring 4 data disk drives per DataNode
- Allocating 1 Map slot and 1 Reduce slot per CPU core
- Allocating 1GB of initial and max Java heap size for Map and Reduce JVM processes

Hence, the 2 JVM processes per core may allocate up to 2GB of heap space per core (3GB of RAM per core is available). So the remaining 1GB of memory per core can be used by the OS kernel, application executables, and cache subsystems. This baseline configuration was chosen based on the TeraSort workload behavior. Based on this setup, a first set of 10 benchmark runs were executed, the collected performance data was post-processed and analyzed (the CV revealed a less than 4% fluctuation among the runs), and a first baseline performance document was compiled.

Sensitivity Study 1 - Data Disk Scaling

One of the first speedup questions to be answered is how well the workload scales while additional disks are made available. In a Hadoop environment, that requires adjusting *mapred.local.dir* (in *mapred-site.xml*) as well as *dfs.name.dir* and *dfs.data.dir* (in *hdfs-site.xml*) to reflect the number of data disks utilized by the Hadoop framework. For this study, the TeraSort benchmark was executed (10 times each) with 5 and 6 disk drives available per data node. Based on the rather high IO demand of the TeraSort workload, the benchmark runs revealed that performance scales rather well while adding additional data disks. Using the 4 disk setup as the normalized performance baseline (aka 100%), scaling the number of disks to 5 and 6, resulted in lowering the normalized execution time by 19% and 34%, respectively. Hence, the decision was made to execute the other sensitivity studies with 6 data disks per data node.

Sensitivity Study 2 - Data Compression

Hadoop supports compression at the input data, intermediate Map output data, and Reduce output data stages, respectively. Hadoop further supports multiple codecs to perform the compression/decompression tasks (see Table 1). Some codecs provide better compression factors, but take longer to compress/decompress the data, while others strive to balance the compression factor with the compression/decompression related overhead. The TeraSort workload does not support input data compression or Reduce output data compression. Hence, only compression at the intermediate Map output stage can be benchmarked. Enabling Map output compression reduces the disk and the network IO overhead at the expense of utilizing additional CPU cycles to compress/decompress the data. Therefore, compression in Hadoop may reflect an exercise in compromises where based on the workload and the HW setup, the decision has to be made to either or not utilize a codec (and if yes, at what stage). The configuration files for data compression in Hadoop are:

- *mapred.compress.map.output,*
- *mapred.map.output.compression.codec,*
- *mapred.output.compress,*
- *mapred.output.compression.type,*
- *mapred.output.compression.codec*

all located in *mapred-site.xml*.

To reiterate, Hadoop clusters that are already CPU bound (scenarios where the cores do not stall much for IO activities and hence not many CPU cycles are available for compression/decompression tasks), may not benefit from using a codec. For this study, benchmarks with *Snappy*, *LZO*, and *gzip* where conducted. Based on the TeraSort workload, the *LZO* codec provided the best performance behavior. Compared to the normalized execution time baseline, using LZO compression reduced the execution time by approximately *32%*. While the other 2 codecs provided improved

performance behaviors, neither one was close to *LZO*. Hence, the decision was made to use the *LZO* codec for the next rounds of sensitivity studies.

Table 1: Some Hadoop Compression Formats

Compression Format	*Hadoop Compression Codec*
DEFLATE	*org.apache.hadoop.io.compress.DefaultCodec*
gzip	*org.apache.hadoop.io.compress.GzipCodec*
bzip2	*org.apache.hadoop.io.compress.BZip2Codec*
LZO	*com.hadoop.compression.lzo.LzopCodec*
LZ4	*org.apache.hadoop.io.compress.Lz4Codec*
Snappy	*org.apache.hadoop.io.compress.SnappyCodec*

Note: The LZO libraries are GPL-licensed and may not be included in some distributions. Depending on the Hadoop environment, next to compression, it may also be beneficial to implement a *combiner* to reduce the amount of data to be transferred. The combiner function is used as an optimization for the MapReduce job. The combiner function runs on the output of the Map phase, and is used as a filtering or aggregating step to lessen the number of intermediate keys that are being passed to the Reducers.

Sensitivity Study 3 - JVM Reuse Policy

The Hadoop parameter *mapred.job.reuse.jvm.num.tasks* determines whether or not the spawned MapReduce JVM threads can be reused (aka execute more than 1 task). The parameter is defined in *mapred-site.xml* with a default value of 1, which implies that the JVM is not being reused. Adjusting the value to -1 changes the behavior in a way that allows an unlimited number of tasks can be scheduled per JVM instance. Enabling the JVM reuse feature may reduce the JVM startup and shutdown overhead and improve performance, as the JVM spends less time interpreting Java bytecode (JIT compilation). The JVM reuse feature is normally beneficial in cases where the workload consists of a large number of very short running tasks. Having said that, enabling JVM reuse did not have a significant impact on the TeraSort workload.

Sensitivity Study 4 - HDFS Block Size

Each Map task operates on what is labeled an input split [2]. The *mapred.min.split.size*, *mapred.max.split.size* (in *mapred-site.xml*), and *dfs.block.size* (in *hdfs-site.xml*) configuration values determine the size of the input split. The input split size and the total input data size of the Hadoop workload govern the total number of Map tasks spawned by the Hadoop framework. For a workload such as TeraSort, the best way to adjust the input split size is by adjusting the HDFS block size via

dfs.block.size. If a Hadoop thread is spawning a large number of Map tasks, it is suggested to evaluate the performance behavior with larger HDFS block sizes. Reducing the number of Map tasks by using larger block sizes may decrease the Map JVM start and shutdown overhead. Further, it may also reduce the cost incurred while merging Map output segments during the Reduce phase. Larger block sizes normally also aid in prolonging the execution time for each Map task.

With Hadoop, it is beneficial to execute a small number of long(er) running Map tasks compared to a setup with a large number of (very) short running Map tasks. It has to be pointed out though that that if the Map output size is proportional to the HDFS block size, bigger block sizes may trigger additional Map-side spills if the spill related properties are not adjusted accordingly (see below). For this study, HDFS block sizes of 64MB (default), 128MB, and 256MB were benchmarked. The conducted benchmarks disclosed that for the TeraSort workload, a 256MB HDFS block size provides the most effective setup. To illustrate, compared to the 64MB baseline, with a 256MB HDFS block size, the normalized execution time was reduced by approximately 18%.

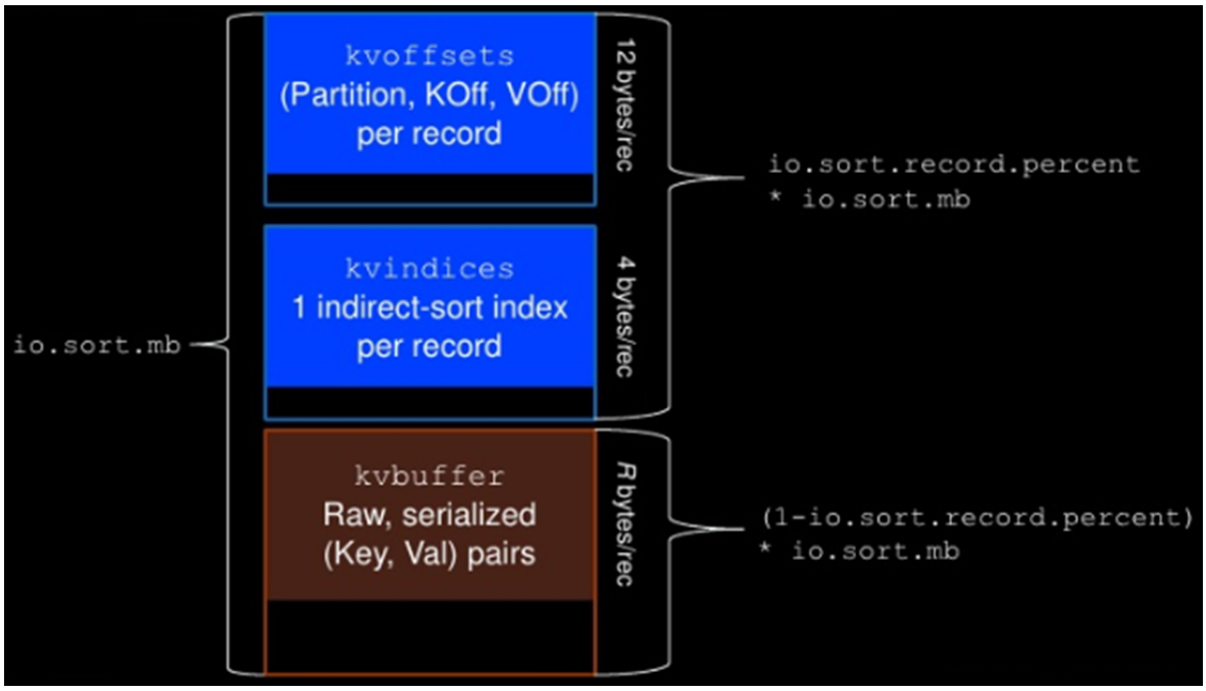
Sensitivity Study 5 - Map Side Spills

The number 1 goal of the Map task optimization phase is to assure that the workload only spills once (during the final spill). While the Map tasks are being executed, the (intermediate) output is stored in a buffer. This buffer basically reflects a chunk of reserved memory that is part of the Map JVM heap space. The default buffer size equals to 100 MB (total buffer space governed by *io.sort.mb* in *mapred-site.xml*). Further, a portion of that buffer is reserved for metadata (for the spilled records). The default value for *io.sort.record.percent* equals to 0.5 (5%) (in *mapred-site.xml*) and hence, for the default 100MB buffer size, the metadata buffer size equals to 5MB (see Figure 1). As each metadata record equals to 16 bytes, a total of 327,680 metadata records can be stored in the buffer. The actual spills occur as soon as a certain threshold is reached, either by the data or the metadata portion of the buffer. The default threshold value *io.sort.spill.percent* (in *mapred-site.xml*) equals to 0.8 (80%). In a lot of Hadoop installations, the crux of the issue is that the *metadata* buffers are saturated much faster than the *data* buffers.

The process of spilling Map outputs *n* times to disk prior to the final spill generates additional overhead due to reading/merging the spilled records. If sufficient Java heap memory is available for the Map JVM's, the goal has to be to eliminate all the intermediate spills. If the available heap space is limited, the focus has to be on minimizing the number of spills by adjusting the *io.sort.record.percent* parameter. To determine if a Hadoop setup is encountering intermediate spills requires monitoring the *Map Output Records* and the *Spilled Records* counters via the JobTracker Web interface. This has to be done immediately after the Map phase for a job is completed. If the number of *Spilled Records* is greater than *Map Output Records*, additional spilling is occurring in the Hadoop setup. For the TeraSort benchmark and a HDFS block size of 256MB (each record is 100 bytes long), *io.sort.mb* was set to 300MB,

io.sort.record.percent to 0.15 (15% of 300MB) and *io.sort.spill.percent* to 0.95 (95%) to eliminate Map-side spills. The Map side tuning resulted into an approximately 2% performance improvement (normalized execution time). While tuning a Hadoop setup, it is imperative to understand that the value for *io.sort.mb* has to fit into the Java heap (plus whatever memory the Mapper requires plus an additional memory overhead that is normally approximated at 30%). *Note*: As of Hadoop 2.0, it is no longer necessary to adjust *io.sort.record.percent*.

Figure 1: MapOutputBuffer (Hadoop 1.x)



Note: Picture courtesy of Hadoop/Apache

Sensitivity Study 6 - Copy Phase Tuning

In Hadoop setups where the Reduce phase does not complete copying Map outputs soon after all the Map tasks are processed, additional tuning may be necessary. A slow copy phase may be due to several possible scenarios:

- The default maximum number of parallel MapOutput copy threads (*mapred.reduce.parallel.copies* in *mapred-site.xml*) equals to 5, and hence could be the limiting factor. It is suggested to increase that value (especially on larger Hadoop clusters). It has to be pointed out though that higher values may lead to IO contention and hence, benchmarking to detect an optimal balance for the workload at hand is necessary
- The default maximum number of worker threads (at the TaskTracker level) used to serve Map outputs to the Reducers (*tasktracker.http.threads*) may be

configured too small. This is a TaskTracker level property, and adjusting the value (in small increments) may improve performance in the copy phase
- Parameters such as *dfs.namenode.handler.count* (in *hdfs-site.xml*) or *mapred.job.tracker.handler.count* (in *mapred-site.xml*) may have to be adjusted. These parameters govern the number of NameNode and JobTracker server threads that handle remote procedure calls (RPC's) and for some installations, the default values are too small
- The *dfs.datanode.handler.count* (in *hdfs-site.xml*) parameter may have to be adjusted, as the default value may be too small, especially if a large number of HDFS clients are active in the cluster. It has to be pointed out though that each additional thread increases the demand on the memory subsystem
- Reduce side spill scenarios could be the culprit (see below)
- Network related constraints may be contributing to the behavior. Using Linux NW benchmarks allow quantifying the throughput potential of the network subsystem

Sensitivity Study 7 - Reduce Side Spills

For most Hadoop workloads, from a performance perspective, the Reduce phase is paramount in regards to the total execution time. In general, the Reduce phase is network and IO intensive, as all the output data generated by (potentially) a large number of Map tasks has to be copied, aggregated/merged, processed, and written back into HDFS. Hence, depending on the total number of allocated Reduce slots, the Java heap requirements for the Reduce JVM's may (greatly) exceed the Map JVM's demand. As the Map tasks start completing their work, the Map output is sorted, partitioned (per Reducer), and written to the TaskTracker disks. Next, these Map partitions are copied to the appropriate Reduce TaskTrackers. A buffer governed by *mapred.job.shuffle.input.buffer.percent* (in mapred-*site.xml*) is used to store the Map output data. If the buffer is too small to store all the data, the Map output has to be spilled to disk. The default value for *mapred.job.shuffle.input.buffer.percent* equals to 0.70, which implies that 70% of the Reduce JVM heap space is reserved for storing the copied Map output data. If the buffer reaches a certain threshold (controlled by *mapred.job.shuffle.merge.percent* in *mapredsite.xml* and set to 0.66 (66%) by default), the accumulated Map output data is merged and spilled to disk. As the Reduce-side sort phase completes, part of the Reduce JVM heap (restricted by *mapred.job.reduce.input.buffer.percent* in *mapred-site.xml*) can be used to retain the Map outputs prior to feeding it into the final reduce function of the Reduce phase. By default, the *mapred.job.reduce.input.buffer.percent* parameter is set to 0, which implies that all of the Reduce JVM heap is allotted to the final reduce function. For some Hadoop environments, benchmarks have show that setting *mapred.job.reduce.input.buffer.percent* to 0.7 or 0.8 is sufficient to keep all of the reducer input data in memory.

Configuring (via *mapred.job.shuffle.input.buffer.percent* and *mapred.job.reduce.input.buffer.percent*) large buffers normally aids in avoiding unnecessary IO operations caused by Reduce-side spills. In cases where Identity

Reducers (an *identity reducer* simply outputs each value) are used (such as by the TeraSort workload), the reduce function normally does not require a large Java heap. In such a scenario, performance may be improved by increasing the *mapred.job.reduce.input.buffer.percen* value. For the TeraSort benchmark, increasing *mapred.job.shuffle.input.buffer.percent* to 0.85 and *mapred.job.reduce.input.buffer.percent* to 0.75 resulted into a nice performance gain of approximately 8% (based on the normalized execution time). Depending on the workload, minimizing the number of Map slots, and hence allocating the freed up memory resources to the Reduce JVM's may substantially boost performance. In other words, large heap space setups for the Reduce side may noticeably improve aggregate performance. Another parameter to scrutinize is *mapred.inmem.merge.threshold*, the threshold number of map outputs for starting the process of merging the outputs and spilling to disk. A value of 0 or less implies no threshold, and hence the spill behavior is governed solely by *mapred.job.shuffle.merge.percent*.

Sensitivity Study 8 - JVM Configuration Tuning

The next step in the tuning process focuses on the JVM. The JVM vendors normally strive to optimize/improve performance with any new release. Nevertheless, there is always ample opportunity to improve performance via a *flag* tuning exercise. The suggestion made is to study the available flags for the particular JVM version being used in the Hadoop setup. Some of the JVM flags (the Oracle JDK was used for this TeraSort study) that may have a profound impact on performance are:

- *AggressiveOpts* – An umbrella flag that governs whether certain optimizations are enabled or not by the JVM. The optimizations vary depending on the JVM version, and hence the suggestion made is to experiment with this flag, especially when the systems are upgraded to a newer JVM version
- *UseCompressedOops* – Compressed Ordinary Object Pointers represent compressed pointers that aid in reducing the memory footprint of 64-bit JVM's (at the expense of reduced addressable Java heap space)
- *UseBiasedLocking* - The biased locking feature may improve performance in scenarios where locks are generally not contended

It is always suggested to conduct a garbage collection (GC) performance analysis to potentially further fine-tune the Map and Reduce JVM's (check for any potential GC overhead via *+PrintGCDetails –verbose:gc*). For profiling purposes, add *–Xprof* to *mapred.child.java.opts* and scrutinize the stdout task log. In addition, depending on the JVM being used, the parameter *java.net.preferIPv4Stack* should be set to true (to avoid timeouts in scenarios where the OS/JVM is presented with an IPv6 address and has to resolve the hostname). As a best practice systems implementation task, for the conducted TeraSort benchmarks, the JVM configuration was optimized prior to executing any actual benchmarks on the Hadoop cluster.

Sensitivity Study 9 - OS (Linux) Tuning

The Transparent Huge Pages (THP) feature focuses on simplifying large page management. Depending on the application and the workload, the feature may significantly improve the overall performance behavior of Linux installations. However, some Hadoop workloads (executing *RedHat* Linux with the THP feature enabled) revealed high system CPU utilization (due to the THP compaction process) that let to rather large Hadoop performance degradations. In such a scenario, it is suggested to set the THP parameter to *never* (disabled).

The local Linux filesystem, as well as the IO scheduler chosen for the Hadoop setup have a profound impact on aggregate Hadoop cluster performance. Contemporary Linux kernels support the CFQ, the deadline, as well as the noop IO scheduler (the anticipatory IO scheduler has been retired as of 2.6.37). Depending on the actual workload and the physical setup of the IO subsystem, any of the 3 IO schedulers may perform best in any given Hadoop environment. The same holds true for the local file systems. Some of the more popular Linux file systems are ext4, XFS, JFS, and Btrfs. For most Hadoop installations, the ext4 file system reflects the preferred choice. For this study, the ext4 filesystem was used in conjunction with the CFQ IO scheduler. The normalized execution time delta between using the CFQ and the deadline IO scheduler, respectively, was approximately 3% (in favor of CFQ). As the noop scheduler reflects an IO scheduler solution that normally works well for SAN subsystems, and as the benchmarked Hadoop cluster does not use any SAN solution, the noop option was not further pursued.

With Hadoop, it is paramount to mount the data disk file systems with the *noatime* attribute. Omitting the mount option results in triggering for each file read operation a disk write call to maintain the last access time stamp for the file. For the TeraSort benchmark, mounting the data disk file systems with *noatime* (on Linux, *noatime* includes *nodiratime*) resulted into a significant performance gain (approximately 21%). For some Hadoop workloads, it is beneficial to tune the file system read-ahead buffer size (especially for sequential read operations of large files) by prefetching additional blocks into memory. The read-ahead tuning on Linux systems is conducted via *blockdev --setra*. It has to be pointed out that additional Linux IO tuning may be necessary to optimize the workload onto the logical and physical systems resources. Please see [HEGE2010] for more information on optimizing Linux IO performance.

Summary & Conclusion

Compared to the initial baseline configuration with 4 disks per data node, the configuration and tuning adjustments discussed in this report resulted into a cumulative performance improvement factor of 4.2 (for the TeraSort benchmark). A big portion of the performance gain for the TeraSort benchmark was due to utilizing 6 instead of 4 disks per data node. Nevertheless, comparing the 6 disk setup tuned verses the 6 disk

non-tuned TeraSort benchmark runs still revealed an improvement factor (normalized execution time) of 2.6.

It has to be pointed out though that designing and configuring a Hadoop cluster for *optimal performance* is considered a moving target that is heavily workload dependent. To illustrate, while this study did incorporate an evaluation of the Linux IO schedulers impact on the TeraSort benchmark, the study did not address the impact of running the TeraSort workload against either the Fair or the Capacity Hadoop scheduler, respectively. Due to the nature of the TeraSort workload, the behavior of the Fair and the Capacity Hadoop scheduler, as well as the setup of the Hadoop cluster used in this study, the decision made was to only run the benchmark with the Hadoop default scheduler (FIFO).

In general, achieving optimal performance behavior from any Hadoop setup requires choosing the appropriate HW and SW stack. Fine-tuning the Hadoop environment necessitates a fairly in-depth analysis of the code path, and ultimately the physical and logical resources utilized by the application workload. Depending on the Hadoop environment, this may be a rather time-consuming (but necessary) process. As with most IT projects, the due diligence in the design and planning phase normally pays off in regards to the performance, stability, and the TCO aspects while the Hadoop cluster is ultimately moved into production.

References

INTE2010 Intel, "Optimizing Hadoop Deployments", Intel White Paper, 2010

HEGE2012 Heger, D. “Hadoop Design, Architecture & MapReduce Performance", CMG Journal, 2012

JOSH2012 Shrinivas Joshi, "Hadoop Performance Tuning Guide", AMD White Paper, 2012

AFRA2010 F. N. Afrati, J. D. Ullman, "Optimizing joins in a map-reduce environment", EDBT, 2010

VERN2010 R. Vernica, M. J. Carey, C. Li, "Efficient parallel set-similarity joins using MapReduce", SIGMOD, 2010

NAYL2010 Bruce F. Naylor, "A Tutorial on Binary Space Partitioning Trees", Spatial Labs, 2010

WHIT2012 Tom White, "Hadoop: The Definitive Guide", 3rd Edition, O’Reilly Media, May 2012

HEGE2010 Heger, D., "Quantifying IT Stability - 2nd Edition, Fundcraft Publication, 2010

SHVA2010 K. Shvachko, H. Kuang, S. Radia, R. Chansler, "The Hadoop distributed file system", In Proc. of the 26th IEEE Symposium on Massive Storage Systems and Technologies (MSST), 2010.

GHEM2003 Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung, "The Google File System", Proc. of SOSP 2003

DEAN2004 Jeffrey Dean, Sanjay Ghemawat, "MapReduce: Implied Data Processing on Large Clusters", Proc of OSDI'04, 2004

VAQU2009 Luis Vaquero, Luis Rodero, Juan Caceres, et al., "A Break in the Clouds: Towards a Cloud Definition" ACM SIG-COMM Computer Communication Review, 2009

ZAHA2008 Matei Zaharia, Andy Konwinski, Anthony Joseph, "Improving MapReduce Performance in Heterogeneous Environments", Proc of the 8th Usenix Symp on Operating Systems Design and Implementation, 2008

Apache, "Fair Scheduler", *http://Hadoop.apache.org/common/docs/current/Fair_scheduler.html,2010*

Apache, "Capacity Scheduler", *http://Hadoop.apache.org/common/docs/current/Capacity_scheduler.html, 2010*

Hadoop: *http://hadoop.apache.org/*

HDFS: *http://hadoop.apache.org/hdfs/*

Hbase: *http://hbase.apache.org/*

Oracle, "Hadoop Sizing", *https://blogs.oracle.com/datawarehousing/entry/sizing_for_data_volume_or, 2012*