Appendix B
Cluster Benchmark Document

As a Hadoop cluster is brought online and optimized for peak performance, it is important to periodically benchmark the performance of the cluster to ensure any changes made were beneficial to the performance of the cluster.

Install distributed monitoring software:
To assist in this task, we'll use the Ganglia distributed system monitoring software. We'll use the one node we installed the GUI version of CentOS to be our main monitoring node. To easily install the software we need, it is necessary to add the RPMForge repository:

- `wget .` http://apt.sw.be/redhat/el6/en/x86_64/rpmforge/RPMS/rpmforge-release-0.5.3-1.el6.rf.x86_64.rpm
- `rpm -ivh` rpmforge-release-0.5.3-1.el6.rf.x86_64.rpm

Now that we have access to the RPMForge repository, lets install the necessary packages on the main monitoring node:

- `sudo yum install rrdtool ganglia ganglia-metad ganglia-gmond ganglia-web httpd`
- `sudo service gmond start`
- `sudo service gmetad start`
- `sudo service httpd start`

If everything went successfully, a Ganglia monitor should be visible by going to http://localhost/ganglia.

To get Ganglia up and running on our other nodes, SCP the RPMForge rpm to those machines and install it as we did above. Then:

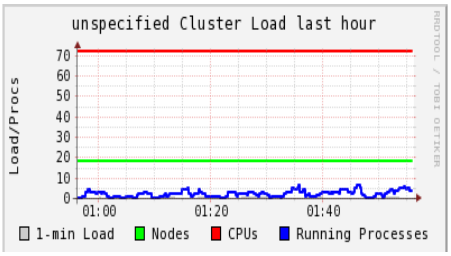
- `sudo yum install ganglia ganglia-gmond`
- `sudo service gmond start`

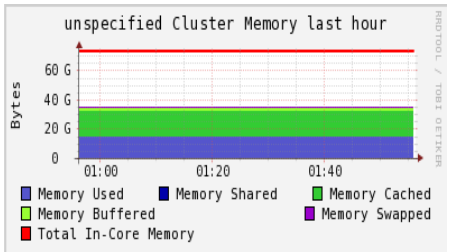
By default, Ganglia will monitor processor load, memory, network bandwidth and hard disk space, which is all that we really need at the moment.

Initial metrics at rest:
After installing the Ganglia monitoring software, the cluster used shows these basic metrics while at rest:

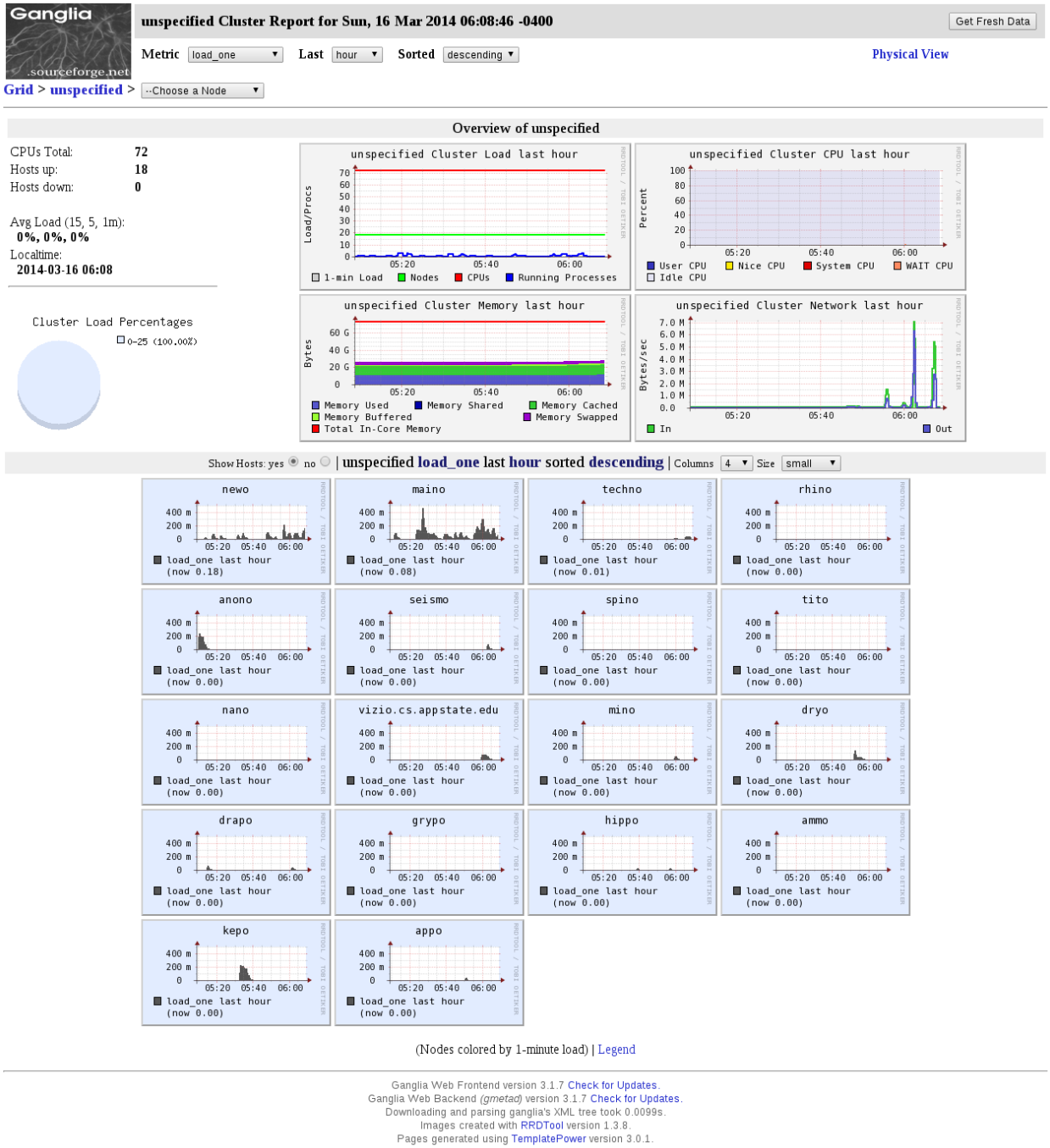
CPUs Total:	72
Hosts up:	18
Hosts down:	0

Avg Load (15, 5, 1m):
0%, 0%, 0%
Localtime:
2014-03-14 01:55





Clicking on the graphs will allow for expanded visualization and customization options, appearing initially like this:



Machine 'classes':

Proper configuration of a Hadoop cluster is essential to it's achieving optimal performance. In many/most Hadoop clusters, relatively homogenous hardware is used, making it relatively easy to assign configuration specifications across the cluster. In this case, the cluster is made up of very different hardware specs. To gain these specs, we'll use the following Linux commands (dmidecode and hdparm may have to be installed: `sudo yum install hdparm`, `sudo yum install dmidecode`):

- free (see total and unused memory) or 'vmstat -s -S -M | grep mem' for more

detailed breakdown.
- dmidecode --type 17 (determine memory speed)
- df (see size of disk/partitions)
- sudo hdparm -tT <mounted hard disk> (test speed of cached reads/buffered disk reads).
- nproc (see number of processors - includes SMT/Hyperthreaded cores).
- cat /proc/cpuinfo (see speed of processors)

1) Aho (Dual-NIC, Namenode)
- Memory: 4.1 GB @ 667 MHz (622 MB free)
- Disk: 520GB @ 3192.25 / 83.32
- Processors: 8 Xeon @ 1.86 GHz

2) Tito
- Memory: 4.0 GB @ 1066 MHz (3.1 GB free)
- Disk: 184GB @ 7557.01 / 88.88
- Processors: 2 Core 2 Duo @ 3 GHz

3) Spino
- Memory: 2.6 GB @ 800 MHz (2.15 GB free)
- Disk: 98GB @ 2597.25 / 70.52
- Processors: 4 AMD Phenom Quad-Core 9350e @ 2 GHz

4) Nano
- Memory: 2.6 GB @ 800 MHz (2.12 GB free)
- Disk: 98GB @ 2282.11 / 71.18
- Processors: 4 AMD Phenom Quad-Core 9350e @ 2 GHz

5) Ammo
- Memory: 2.6 GB @ 800 MHz (2.14 GB free)
- Disk: 98GB @ 2432.68 / 72.19
- Processors: 4 AMD Phenom Quad-Core 9350e @ 2 GHz

6) Techno
- Memory: 2.6 GB @ 800 MHz (2.13 GB free)
- Disk: 98GB @ 2418.30 / 71.93
- Processors: 4 AMD Phenom Quad-Core 9350e @ 2 GHz

7) Dryo
- Memory: 2.6 GB @ 800 MHz (2.12 GB free)
- Disk: 98GB @ 2679.21 / 71.88
- Processors: 4 AMD Phenom Quad-Core 9350e @ 2 GHz

8) Grypo
- Memory: 2.6 GB @ 800 MHz (2.08 GB free)
- Disk: 98GB @ 2437.58 / 72.23
- Processors: 4 AMD Phenom Quad-Core 9350e @ 2 GHz

9) Anono
- Memory: 2.6 GB @ 800 MHz (2.12 GB free)
- Disk: 98GB @ 2487.57 / 71.31
- Processors: 4 AMD Phenom Quad-Core 9350e @ 2 GHz

10) Seismo

- Memory: 2.6 GB @ 800 MHz (2.10 GB free)
- Disk: 98GB @ 2423.35 / 75.01
- Processors: 4 AMD Phenom Quad-Core 9350e @ 2 GHz

11) Rhino
- Memory: 1.9 GB @ 667 MHz (1.17 GB free)
- Disk: 97 GB @ 3292.20 / 42.23
- Processors: 8 Intel Xeon @ 2 GHz

12) Maino
- Memory: 3.84 GB @ 1333 MHz (3.05 GB free)
- Disk: 184 GB @ 6155.0 / 95.95
- Processors: 4 Intel Core i5 @ 3.2 GHz

13) Newo
- Memory: 16.27 GB @ 1600 MHz (14.25 GB free)
- Disk: 901 GB @ 16258.23 / 182.52
- Processors: 8 Intel Core i7 @ 3.4 GHz

14) Appo
- Memory: 7.92 GB @ 1066 MHz (7.17 GB free)
- Disk: 93 GB @ 2201.58 / 70.69
- Processors: 2 Intel Core 2 Duo @ 2.9 GHz

15) Drapo
- Memory: 3.84 GB @ 667 MHz (3.1 GB free)
- Disk: 97 GB @ 1283.59 / 70.08
- Processors: 2 Intel Core 2 @ 2.13 GHz

16) Mino
- Memory: 5.85 GB @ 667 MHz (5.12 GB free)
- Disk: 95 GB @ 4181.61 / 74.89
- Processors: 2 Intel Core 2 Duo @ 2.33 GHz

17) Hippo
- Memory: 3.79 GB @ 667 MHz (3.01 GB free)
- Disk: 338 GB @ 3974.09 / 75.88
- Processors: 2 Intel Core 2 Duo @ 2.33 GHz

18) Kepo
- Memory: 3.79 GB @ 667 MHz (3.02 GB free)
- Disk: 97 GB @ 3988.78 / 74.36
- Processors: 2 Intel Core 2 Duo @ 2.33 GHz

Based upon these hardware specifications, the machines in the cluster can generally be divided up into six classes in order to make configuration simpler:
ACER_CLASS: Spino, Nano, Ammo, Techno, Dryo, Grypo, Anono, Seismo, Maino
LOW_END: Aho, Rhino
MID_RANGE: Tito, Drapo, Hippo, Kepo
NEWO: Newo
APPO: Appo
MINO: Mino

The clusterAdmin.sh script will make it easy to update and push out configurations to the various classes of machines, but first we will test the cluster without making any configuration changes (other then those absolutely necessary in order for the examples to run).

HDFS Testing:
The most straightforward initial test for our cluster is to measure the performance of the HDFS filesystem on reads and writes. To accomplish this, we'll use the TestDFSIO program that comes bundled with Hadoop (/usr/lib/hadoop-mapreduce/hadoop-mapreduce-*test*.jar).

- For these initial read/write tests, all meaningful work is done during the map stage and thus this stage is the only important one. A small subsequent reduce step merely aggregates the results from the various map tasks.
- The number of maps that will launch corresponds to the number of files we want to write/read (indicated by the '-nrFiles' flag).
- `time hadoop jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-*test*.jar TestDFSIO -write -nrFiles 10 -fileSize 1000`
- `time hadoop jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-*test*.jar TestDFSIO -read -nrFiles 10 -fileSize 1000`
- `hadoop jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-*test*.jar TestDFSIO -clean`

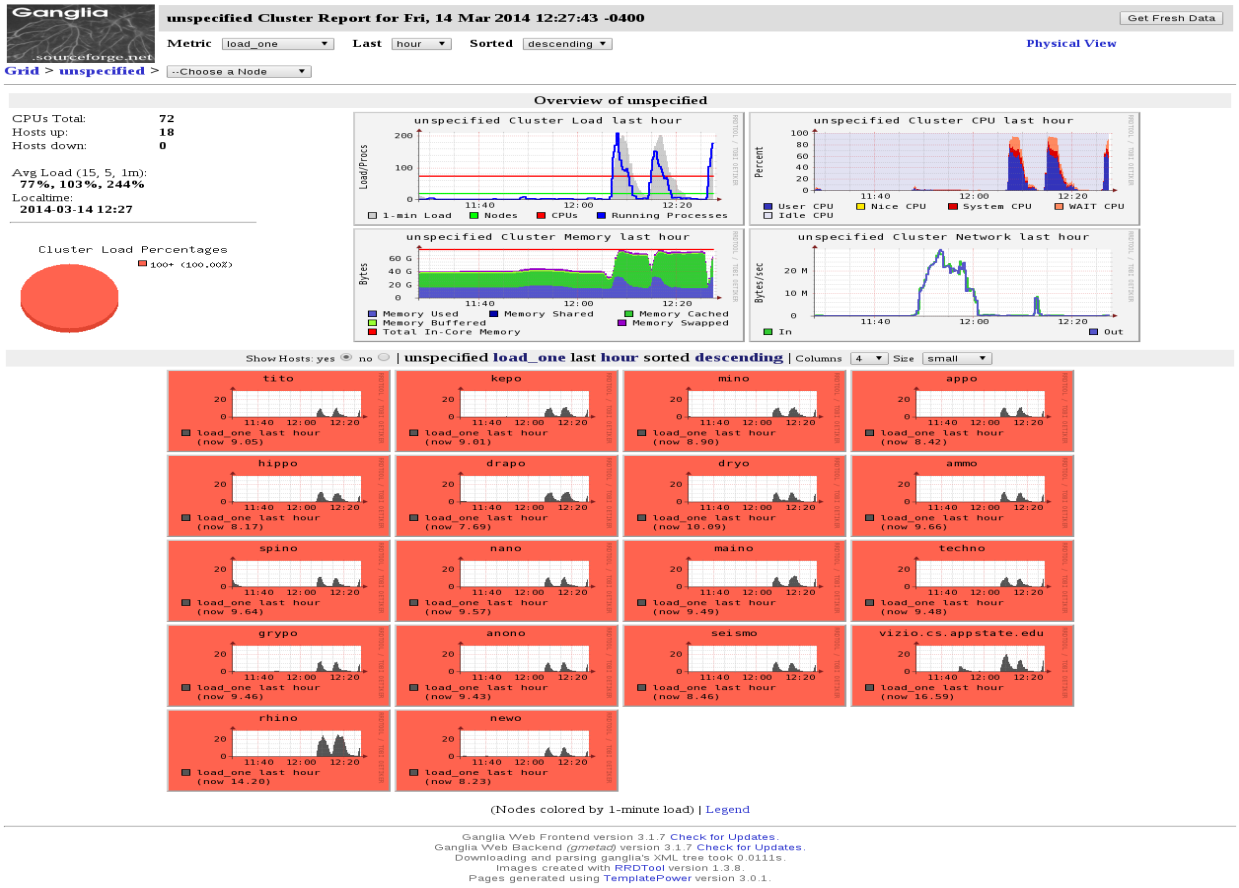
HDFS Initial Results:

- Test Write: `2:53.27`
- Test Read: `12:35.38`

The next logical step is testing the performance of MapReduce on YARN. To do this, we'll generate 180GB+ of random data and then sort it. To generate the random data, we'll use the randomwriter utility found in /usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples-2.0.6-alpha.jar.

- To set the number of maps to be used, pass argument for test.randomwriter.maps_per_host:
  - `hadoop jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples-2.0.6-alpha.jar randomwriter -Dtest.randomwriter.maps_per_host=10 randomdata`
  - To see how large the generated randomdata directory is:
    - `hadoop fs -du -s /user/hadoop/randomdata`
- When the MapReduce application deploys, it will indicate how many maps it will attempt to launch, "Running 180 maps," (10 per node), which should reflect the argument set above. You can also use the -conf flag to specify an xml configuration file to pass to randomwriter.
- This test presents the first opportunity thus far to really see the cluster under some

stress, as indicated by the Ganglia monitoring:



Now to sort the data we just generated we can use another bundled sort program found in the examples JAR used above. The number of reducers by default is 1.0 x configured capacity and may run faster with more.

- `time hadoop jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples-2.0.6-alpha.jar sort randomdata sorteddata`
- Since this sorting is by lengths the longest test we've performed so far, it is usually easier to deploy the three instances through a BASH script.

```
#!/bin/bash
hadoop fs -rm -r /user/hadoop/sorteddata > sortOutput
time hadoop jar
/usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples-2.0.6-alph
a.jar sort randomdata sorteddata >> sortOutput
hadoop fs -rm -r /user/hadoop/sorteddata >> sortOutput
time hadoop jar
/usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples-2.0.6-alph
a.jar sort randomdata sorteddata >> sortOutput
hadoop fs -rm -r /user/hadoop/sorteddata >> sortOutput
time hadoop jar
/usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples-2.0.6-alph
```

```
a.jar sort randomdata sorteddata >> sortOutput
hadoop fs -rm -r /user/hadoop/sorteddata >> sortOutput
```

- Log4j sends most logging info to the console by default, leaving STDOUT with just the value run/time data:

```
Running on 36 nodes to sort from
hdfs://aho:8020/user/hadoop/randomdata into
hdfs://aho:8020/user/hadoop/sorteddata with 64 reduces.
Job started: Tue Mar 25 00:27:18 EDT 2014
Job ended: Tue Mar 25 02:16:18 EDT 2014
The job took 6539 seconds.
```

At this point, sorts are very unlikely to run without error. Many of the default configuration values do not suit our cluster well at all and others may prevent some nodes from running certain tasks entirely. To see what went wrong with running/failed containers, it is helpful to enable `yarn.log-aggregation-enable` (it's on by default) and look in the directory specified `by yarn.nodemanager.remote-app-log-dir` (/var/log/hadoop-yarn/apps on HDFS by default). If remote logging is not enabled or HDFS is not functioning properly, check `yarn.nodemanager.log-dirs` (default to /var/lib/hadoop-yarn/containers on the local filesystem).

- copying the resulting log files to the local machine can make life easier, and make it simple to use Linux utils such as 'grep' to help identify the errors.
  - `hadoop fs -copyToLocal /var/log/hadoop-yarn/apps/hadoop/logs/application_1394842973416_0032 .`
  - `grep -R ERROR *`
- alternatively, we can use Hadoop's own HDFS cat program:
  - hadoop fs -cat /var/log/hadoop-yarn/apps/hadoop/logs/application_`1394842973416_0032/* | grep "60000" | grep -v "600000"`
- The first error we encountered was the ApplicationManager killing a task that failed to complete in 60 seconds, which is too little time for some of the weaker nodes. To fix this, alter mapred-site.xml:

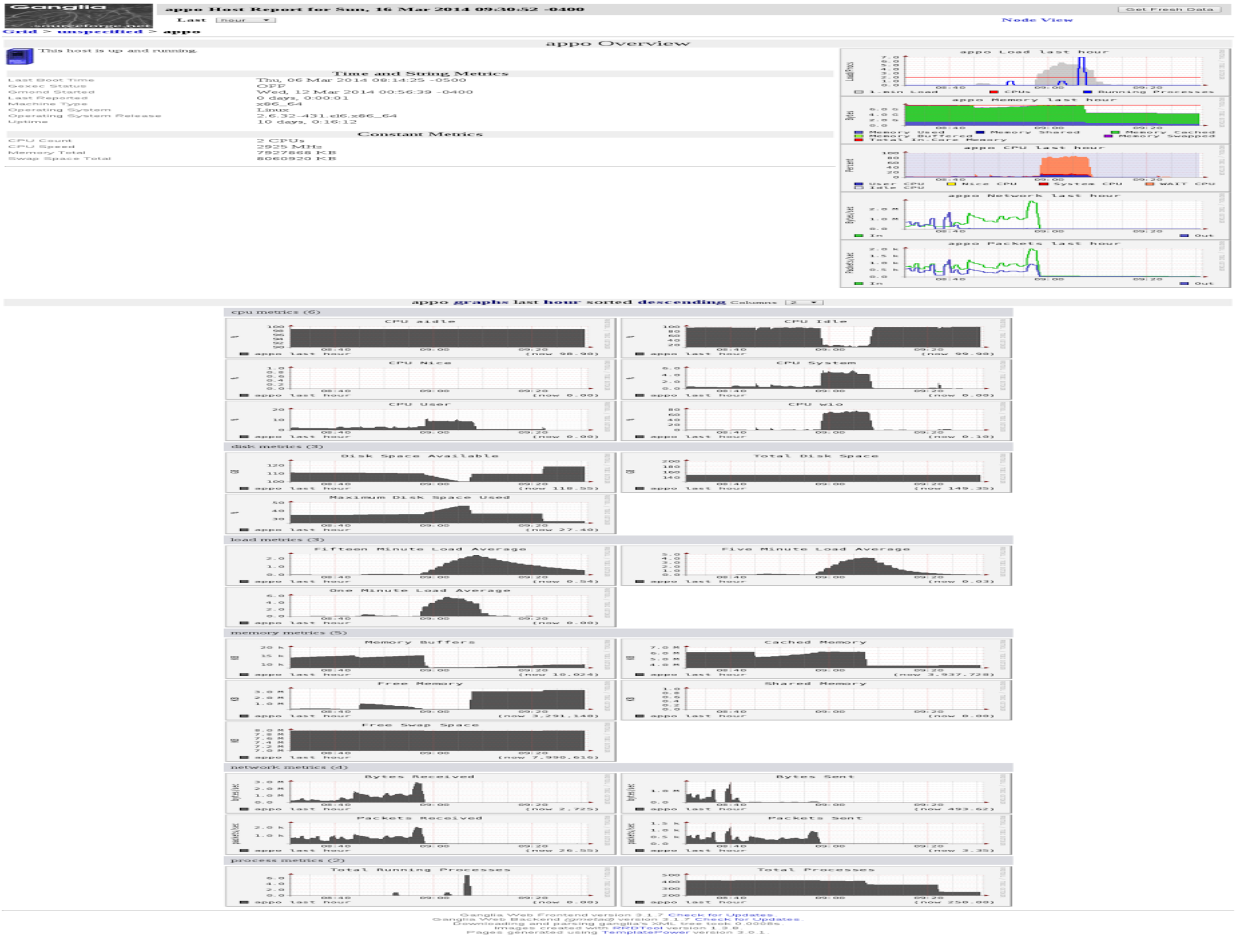
```
<property>
    <name>mapreduce.task.timeout</name>
    <value>1800000</value> <!-- 30 minutes -->
</property>
```

- Also encountered are errors related to cleanup when a task is killed on a node (likelly due to speculative execution on another faster node concluding before the node in question could finish). Setting this property will fix these:

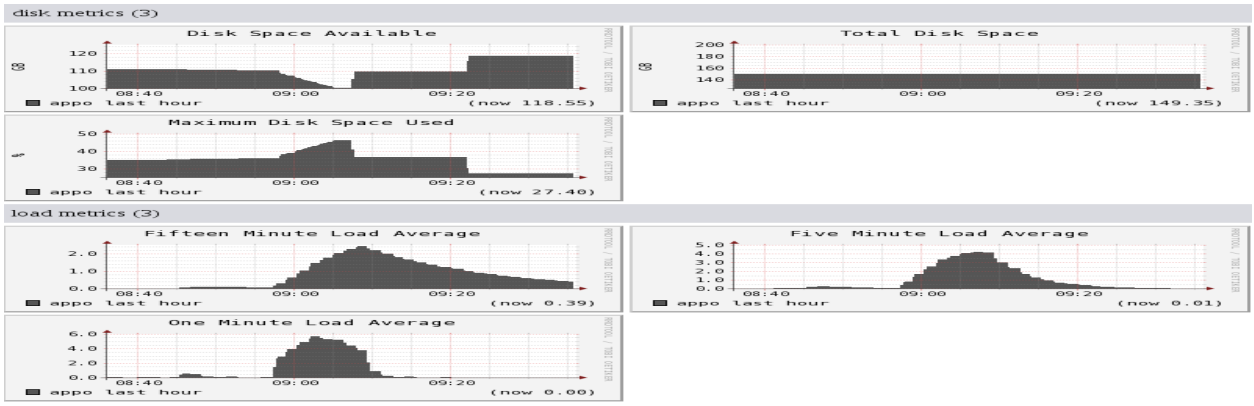
```
<property>
```

```
        <name>yarn.app.mapreduce.am.job.committer.cancel-timeout</
name>

        <value>1800000</value> <!-- 30 minutes -->

    </property>
```

- We also encountered some on disk merge errors, likely due to running out of disk space. By examining the logs as above, we determine that the merge error occurred on node drapo, which we can pull up individually in Ganglia to see what transpired:



- The Ganglia data indicates that the node ran out of disk space while under heavy load:



- The ACER class of machines have the least amount of hard disk and are thus the most likely to encounter such errors during a long task. To combat this, we'll give the ACER nodes more space on another partition and also compress the output of Map tasks written to disk.
    - ./clusterAdmin.sh - e "mkdir -p /tmp/hdfs/dfs/data"
    - ./clusterAdmin.sh -e "sudo chown -R hdfs /tmp/hdfs"
    - In hdfs-site:

```
<property>
    <name>dfs.datanode.data.dir</name>
    <value>/home/hdfs/dfs/data,/tmp/hdfs/dfs/data</value>
</property>
```

    - `./clusterAdmin -h CLASS_ACER:CLASS_ACER`
    - `./clusterAdmin -r`
- In addition to increasing the default space on HDFS, we can specify that ouput of map jobs should be compressed (mapred-site.xml):

```
<property>
    <name>mapreduce.map.output.compress</name>
    <value>true</value>
</property>
```

With these changes, we now should have a cluster capable of handling the sort task without failing multiple task attempts/failing completely. This configuration will serve as our BASIC configuration that we will now further tune to increase performance.

Optimizing:

In many ways, configuring a YARN cluster is easier than the old MR1 clusters. Of primary concern is specifying a proper upper limit for the amount of memory the individual nodes

should make available to the ResourceManager and configuring many other buffer sizes and values. As we've seen, the default memory per node is 8GB, a setting vastly too small for most industry clusters yet vastly too large for our purposes. This misconfiguration can cause a large amount of spillage and thrashing, greatly retarding overall cluster performance.

By checking the ResourceManager Web UI, we can see that the ApplicationMaster was given 143 containers to use during the completion of the sort task, with each one expecting access to 1GB of memory resources, leaving most with dramatically fewer resources and cause many costly spills to disk during the various tasks.

Configuring a Hadoop cluster is not an exact science and can depend to a great extent upon the type of operations one is seeking to use it for. That being said, there are intelligent guidelines that can be followed which I will expound upon here:

As a general rule of thumb for deciding how many YARN containers to run on a specific machine, this approximation is often useful:

- # of containers = min (2*CORES, 1.8*DISKS, (Total available RAM) / MIN_CONTAINER_SIZE)

Before specifying the various configuration values we'll use and repeatedly benchmarking the cluster, it is necessary to understand what the various values actually affect during the execution of tasks:

Round 1 (yarn-site.xml):

- **yarn.scheduler.minimum-allocation-mb** -> this property specifies the smallest container that will be granted by the ApplicationMaster. Any requests for lower amount will be bumped up to this amount. It makes sense for this value to correspond to the smallest amount of memory any class gives map tasks, to which it should give twice as much to reduce tasks. Since our LOW_END class will allow 1024, we'll set this to 1024 as well.
- **yarn.scheduler.maximum-allocation-mb** -> this property specifies the largest container that will be granted by the ApplicationMaster. Any requests for a greater amount of RAM will be capped at this value. It makes sense for this value to correspond to the largest amount of memory any class gives reduce tasks, which will be for NEWO at 14336.
- **yarn.nodemanager.resource.cpu-cores** -> properties specifically tied to the local nodemanager are among the only node-specific configurations we can set on the cluster. This property corresponds to the number of CPU cores available to YARN on this node.
- **yarn.nodemanager.resource.memory-mb** -> the amount of memory made available to YARN on this specific node.
- **yarn.nodemanager.vmem-pmem-ratio** -> This property specifies the ratio of virtual to physical memory for containers. Raising it will allow weaker nodes to handle larger splits, but at a cost of increased spills and decreased overall performance. Checks are performed (yarn.nodemanager.vmem-check-enabled,

yarn.nodemanager.pmem-check-enabled) and the ApplicationMaster will remotely kill any containers that are in violation of this ratio.
- **yarn.app.mapreduce.am.resource.mb** -> Amount of memory resources to allocate to the ApplicationMaster. The default 1.5GB is small and will likely result in unused leftover (512mb) on most nodes in the cluster.
- **yarn.app.mapreduce.am.command-opts** -> Can specify JVM options such as maximum heap size for the ApplicationMaster.
- **yarn.app.mapreduce.am.resource.cpu-vcores** -> The number of virtual CPU cores allocated to the ApplicationMaster process.

Round 2 (mapred-site.xml):

- **mapreduce.[map/reduce].log.level** (mapred-site.xml) -> Hadoop generates an extremely large amount of logging information to assist in diagnosing problems or performance bottlenecks. Much of this logging information is held in memory until the job is completed, so as not to throw off the HD head. Setting these levels to warn (but not final) reduces the amount of memory that will be held for logging while still allowing individual jobs to override if they're experiencing difficulties, etc. Hadoop's system-wide logging level can also be altered in it's log4j.properties file, but this requires more work to change by jobs at runtime.
- **mapreduce.task.userlog.limit.kb** (mapred-site.xml) -> By default, Hadoop will cache (in memory) logs until there is physically no more space to do so. By setting this limit, we ensure that some of our lower-performance nodes don't get overwhelmed simply keeping track of log files. It's likely the log is repeating itself after 4096 kilobytes of warnings.
- **mapreduce.map.memory.mb** -> Size of Map containers running on YARN. Should correspond to our yarn.scheduler.minimum-allocation-mb value so that all nodes can run map tasks. Note that this and all "mapreduce" settings are job-specific and cannot be custom configured for individual nodes in the cluster.
- **mapreduce.reduce.memory.mb** -> Size of Reduce containers running on YARN. Generally the reduce phase is more resource intensive and can require merging large map outputs with significant spill costs. By setting this value to large enough to exclude our weakest nodes we can further modify reduce buffers and other configs that will streamline this phase and use the larger memory resources of our strong nodes to their fullest potentials.
- **mapreduce.map.output.compress** -> This property determine whether the output from all map tasks should be compressed. While obviously adding additional computation, in general the cost savings in network bandwidth make compressing map output a wise choice.
- **mapreduce.map.output.compress.codec** -> Hadoop's default DEFLATE compression algorithm uses a combination of the LZ77 algorithm and Huffman coding. Widely used in gzip, the DEFLATE algorithm is great at reducing size and performs moderately fast. One great issue with DEFLATE in a Hadoop environment however is its inability to split files at boundaries (such as HDFS blocksizes). This means that one node must store the whole thing and that mappers processing that file will run entirely on that node (or other replicants). For this reason I chose the

BZip2 algorithm for compression. While slower, it compresses more effectively than DEFLATE and is fully splittable. LZO is another alternative that works nicely with Hadoop, with the ability to be pre-processed with an indexer tool to provide splittable functionality.

- **mapreduce.[map/reduce].speculative** (hdfs-site.xml) -> this property defines whether the cluster will begin speculatively executing tasks (up to mapreduce.job.speculative.speculativecap) when it determines that a certain task or node is running slowly. By default, once a task has taken more than one standard deviation (34%) longer than the average, it is eligible to be launched speculatively on other nodes that have generally performed well in the past (above mapreduce.job.speculative.slownodethreshold). Since our cluster is so heterogeneous with respect to hardware and with the average time each class of node takes to finish tasks likely to differ greatly, it is better for throughput in the long run to disable speculative execution entirely.
- **mapreduce.reduce.shuffle.input.buffer.percent** -> the amount of JVM heap space reserved in Reduce tasks for Mapper output. The main cause of Reducers being more resource-intensive than Mappers.
- **mapreduce.reduce.shuffle.merge.percent** -> when the buffer corresponding to mapreduce.reduce.shuffle.input.buffer.percent gets filled to this percentage, the accumulated outputs are merged and spilled onto disk.
- **mapreduce.map.sort.spill.percent** -> In a MapReduce program, the output key-value pairs are sorted in the Map phase prior to being shuffled into Reducers. This configuration specifies the maximum amount of the heap this sorting can occupy in memory before it will begin spilling to disk.
- **mapreduce.task.io.sort.mb** -> when map functions begin producing output, this output is not immediately written to disk. Each map task has a circular memory buffer that can temporarily store these results. This size of this buffer is determined by this property, which defaults to 100MB. Once this buffer fills up a certain percentage of the buffer (mapreduce.map.sort.spill.percent), it is spilled to disk.
    - In the results printed after a MR run, if “Map output records” is less than “Spilled Records”, the buffer was not large enough and spilling occurred.

Round 3 (hdfs-site.xml):

- **dfs.replication** (hdfs-site.xml) -> the replication factor for files placed in HDFS. Increasing this value will increase network traffic but also increase performance on pre-written data due to increased locality/availability of local containers.
- **dfs.client.read.shortcircuit** -> When executing anything on Hadoop HDFS, reads normally must go through the local datanode. When a client wants to read a file, they contact the datanode that possesses it and that datanode subsequently reads the file from disk and sends it over a TCP socket. Short-circuit reads bypass the datanode entirely and allow the client to read the file directly from the local filesystem. This direct access can provide large performance benefits for certain access patterns.
    - Note: Native Libraries must be present and property configured for short-circuit reads to work properly. It is simple to compile these from the

Hadoop source package:
    - install a C compiler (e.g. GNU C compiler)
    - install the GNU autotools chain: autoconf, automake, libtool
    - install the latest stable zlib-development package
    - once the source package is downloaded and extracted, invoke Maven from the main pom.xml: `mvn package -Pdist,native -Dskiptests -Dtar`
    - resulting native libraries should be found in hadoop-dist/target/hadoop-2.3.0/lib/native.
    - copy native libraries into $HADOOP_HOME/lib/native
    - in core-default.xml, set 'io.native.lib.available' to 'true' & 'mapreduce.admin.user.env' to 'LD_LIBRARY_PATH=$HADOOP_COMMON_HOME/lib/native'
- **dfs.domain.socket.path** -> short-circuit reads used standard UNIX domain sockets. This path must be accessible to the datanode process but secure to all other users (except hdfs and root).
- **dfs.block.local-path-access.user** -> user allowed to use the shortcut domain socket for reading. Should correspond to our hadoop user (and hbase if running). To add the ability for more users to perform short-circuit reads, the dfs.datanode.data.dir.perm configuration should be edited to 740 and the intended user added to hdfs supergroup.
- **dfs.datanode.data.dir.perm** -> the permissions expected on the HDF data directory. If actual permissions differ from this, the system will automatically reset them to this value.

Once finsished, the clusterAdmin script can be very helpful in speading the new configurations throughout the cluster.

- `./clusterAdmin.sh -h ACER_CLASS:ACER_CLASS`
- `./clusterAdmin.sh -h LOW_END:LOW_END`
- `./clusterAdmin.sh -h MID_RANGE:MID_RANGE`
- `./clusterAdmin.sh -h NEWO:NEWO`
- `./clusterAdmin.sh -h MINO:MINO`
- `./clusterAdmin.sh -h APPO:APPO`

##

- **dfs.block.size** -> Most clusters can benefit from increasing the default HDFS block size from it's default 64MB to 128MB or 256MB. Because some nodes in our cluster are extremely memory-limited however, this increase in block size would

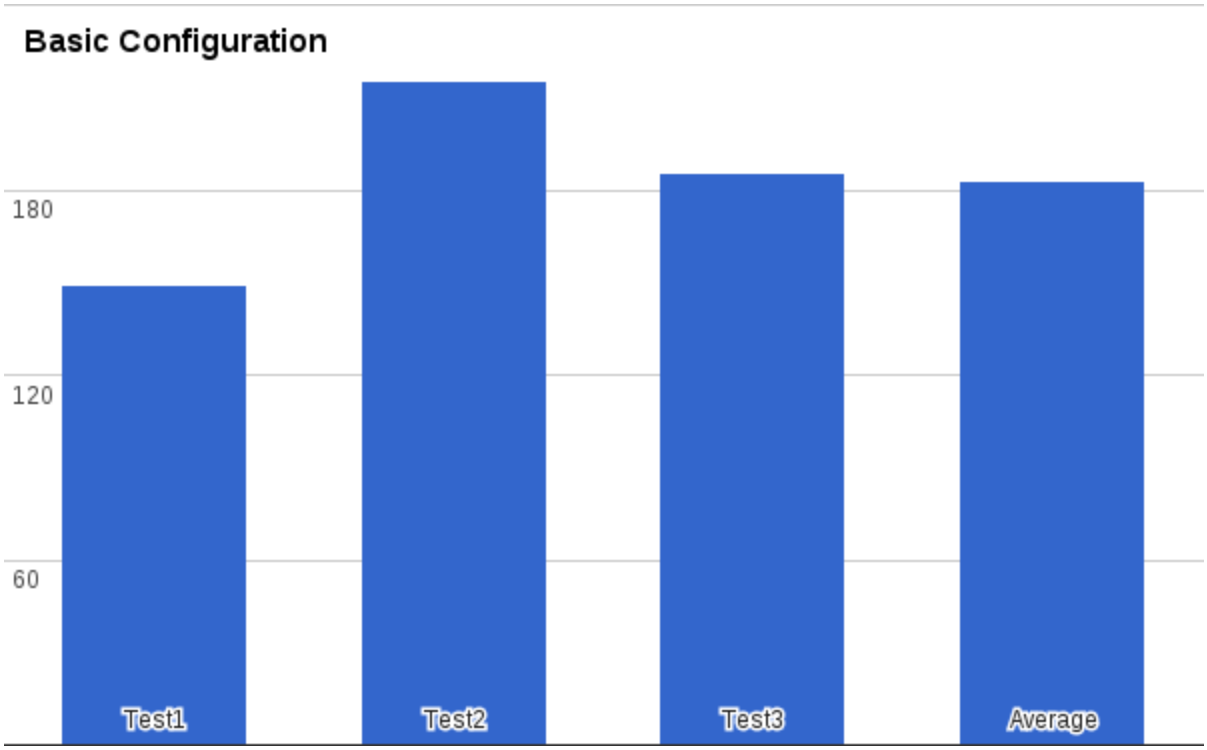
cause much larger merges that would overwhelm their ability to sort and reduce. We'll leave it at the default 67108864 so as not to make the splits too large for our nodes.

Leaving out for now:

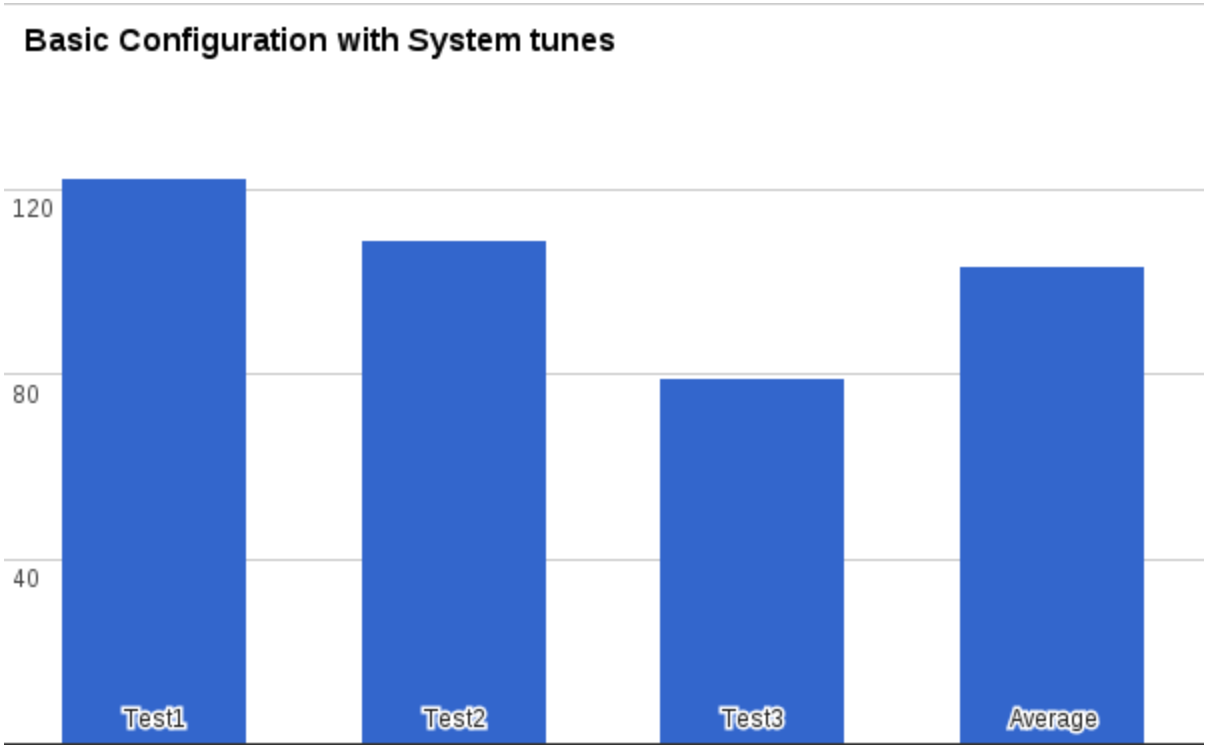
-
- **mapreduce.map.java.opts** -> Specifies the maximum JVM heap-size for map tasks so as to avoid problems brushing up against container limits. It is set on a per-*JOB* basis and can result in many headaches on a hardware heterogenous cluster similar to what we're running. The default value is -Xmx1000m, and this particular property is best set on a per-job basis, depending on the type of work being done.
- **mapreduce.map.java.opts** -> Specifies the maximum JVM heap-size for reduce tasks so as to avoid problems brushing up against container limits. It is set on a per-*JOB* basis and can result in many headaches on a hardware heterogenous cluster similar to what we're running. The default value is -Xmx1000m, and this particular property is best set on a per-job basis, depending on the type of work being done.

Results

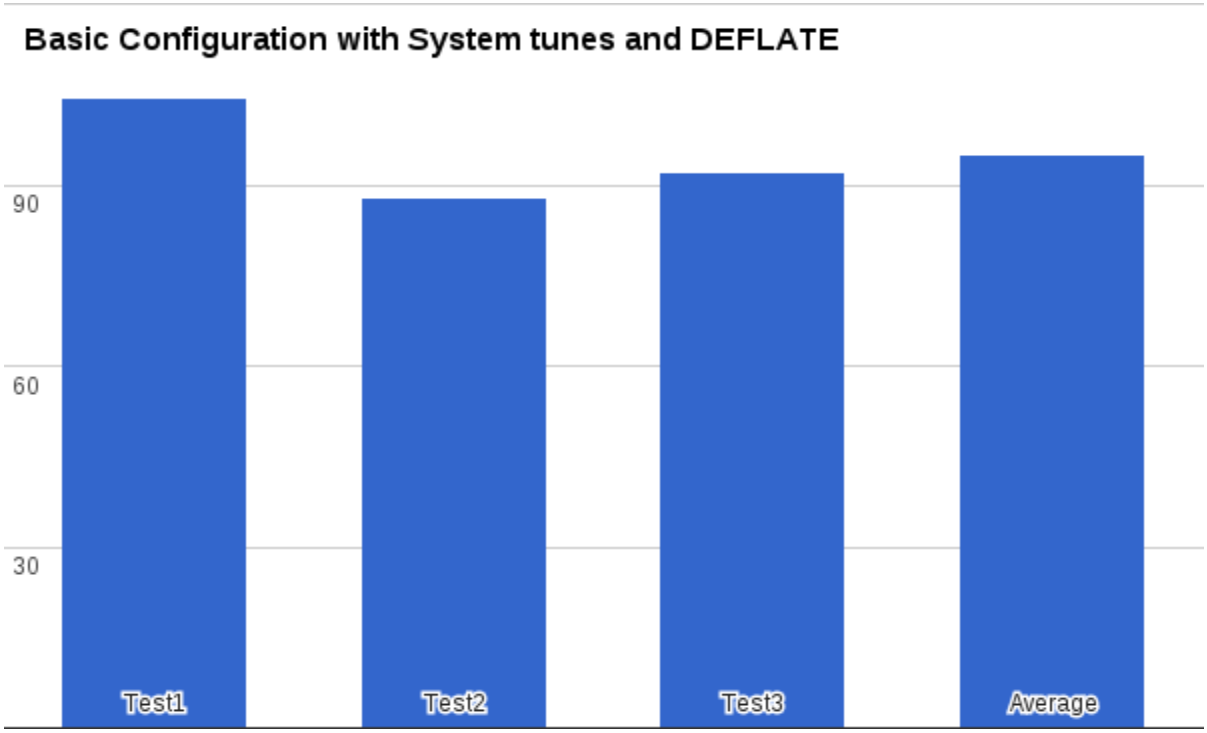
Basic configuration:



Basic configuration with system optimizations:



and default DEFLATE compression:



Yarn settings:

asdf