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 and memory, which is all that we really need at the moment.

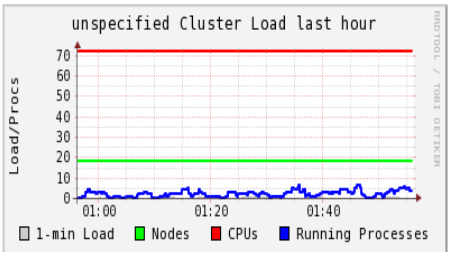
Initial metrics at rest:

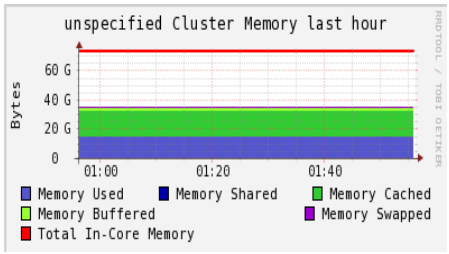
After installing the Ganglia monitoring software, the cluster used shows these metrics while at rest:

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

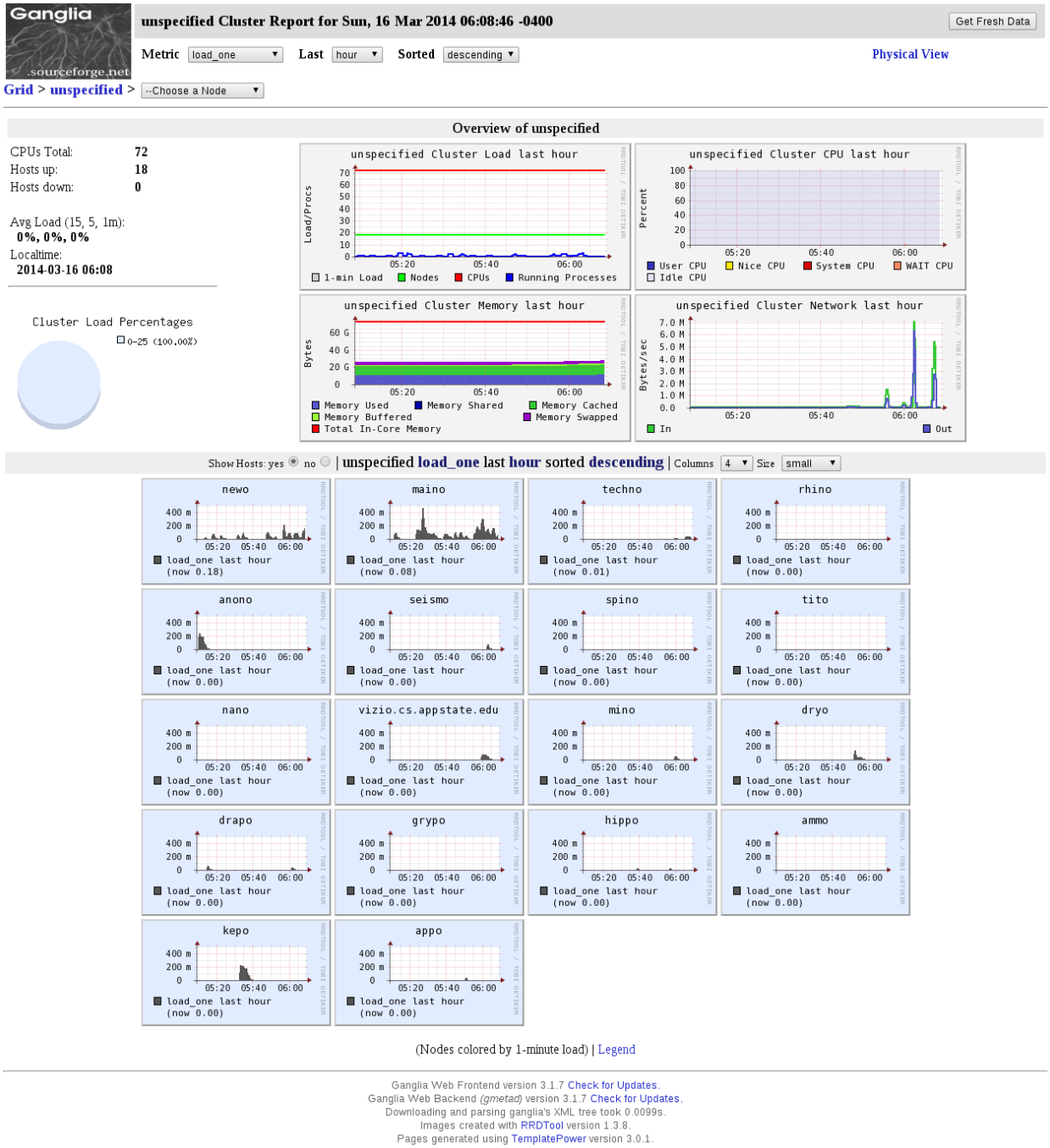
Avg Load (15, 5, 1m):
0%, 0%, 0%

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:

- 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 four classes in order to make configuration simpler:
ACER_CLASS: Spino, Nano, Ammo, Techno, Dryo, Grypo, Anono, Seismo
LOW_END: Aho, Rhino
MID_RANGE: Tito, Maino, Drapo, Hippo, Kepo
HIGH_END: Newo, Appo, 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).

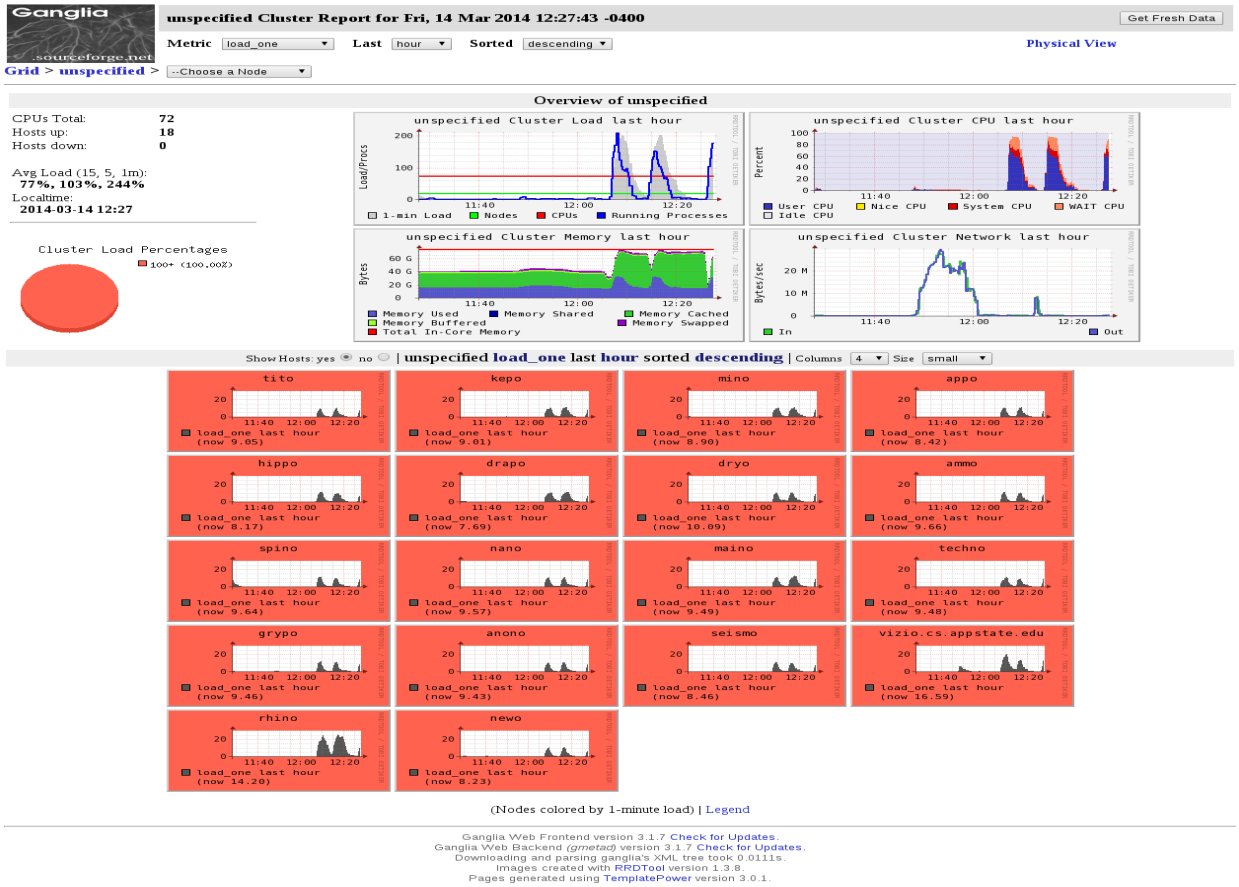
- For these initial read/write tests, all meaningful work is done during the map stage and thus this stage is the only important one. A small subsequent reduce step merely aggregates the results from the various map tasks.
- The number of maps that will launch corresponds to the number of files we want 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 ten gigabytes 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," (default 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.

Sort Results:

Run1	Run2	Run3
5:59.88	**6:21.76**	**6:21.88**

RandomWriter:Sort:

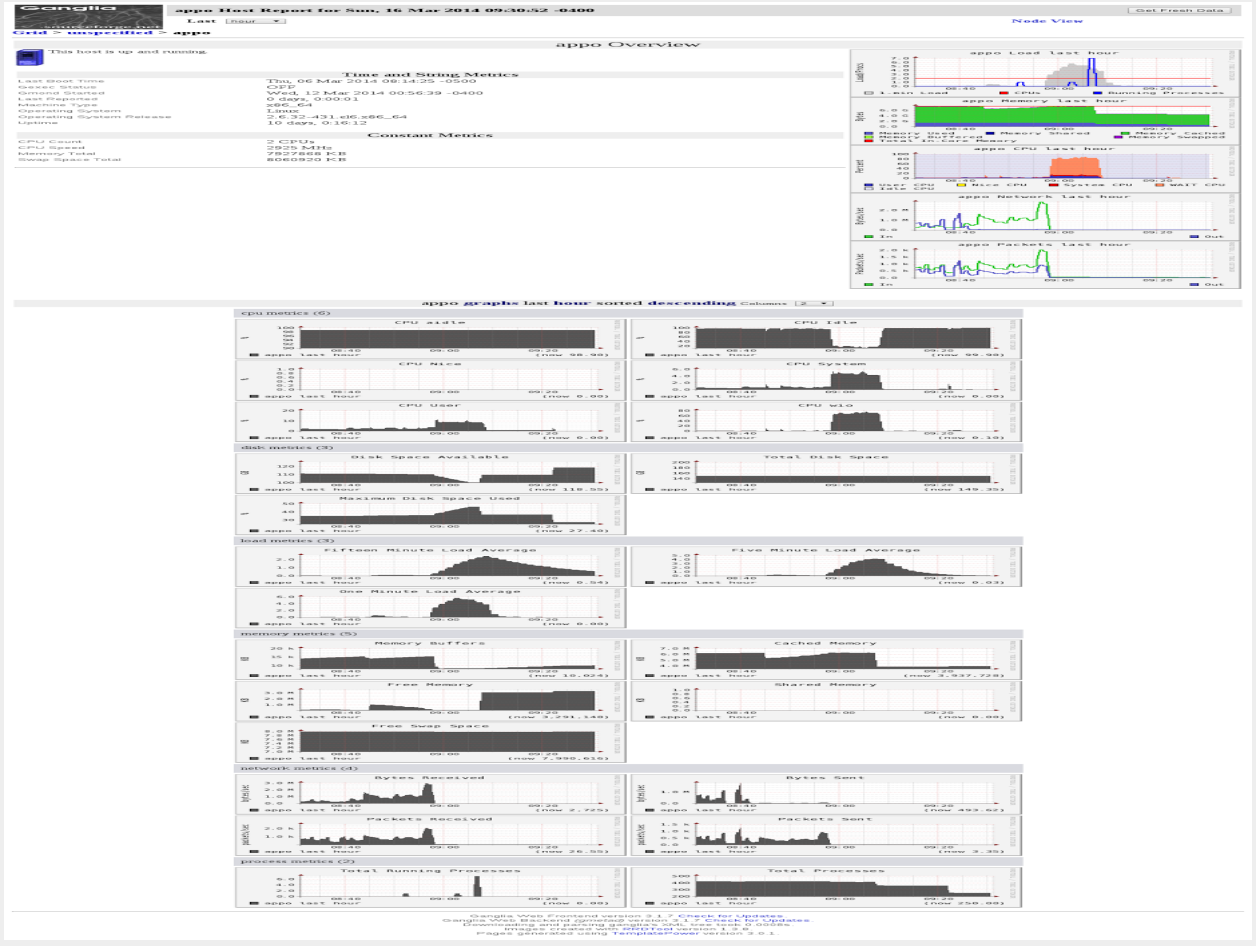
Run1	Run2	Run3
149:16.23	**215:20.21**	**185:24.91**

The wide disparity between results it likely due to the intermittent errors that occurred during the sort, most likely due to poor default configuration values (for our somewhat under-powered machines). To see what exactly went wrong, we head to `yarn.nodemanager.logdirs` (by default, /var/log/hadoop-yarn/ on default filesystem (HDFS)).

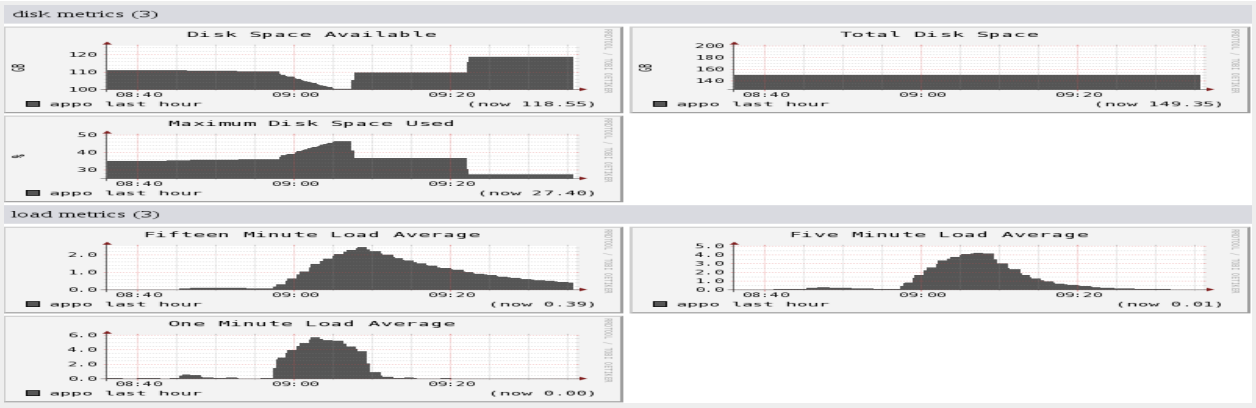
- copying the resulting log files to the local machine can make life easier, and make it simple to use Linux utils such as ‘grep’ to help identify the errors.
    - `hadoop fs -copyToLocal /var/log/hadoop-yarn/apps/hadoop/logs/application_1394842973416_0032 .`
    - `grep -R ERROR *`
- The first error we encountered was the ApplicationManager killing a task that failed to complete in 600 seconds, which is too little time for some of the weaker nodes. To fix this, alter mapred-site.xml:

```
<property>
    <name>mapred.task.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>
```

- By checking the ResourceManager Web UI, we can see that the ApplicationManager was given 143 containers to use during the completion of the sort task, with each one expecting access to 1GB of memory resources, which will leave most with dramatically fewer resources and cause many costly spills to disk during the various tasks. To fix this terrible misconfiguration, we need to assign proper values to the various classes of machines. In yarn-site.xml:

- ACER_CLASS:

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

- LOW_END:

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

- MID_RANGE:

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

- HIGH_END:

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

- 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 HIGH_END:HIGH_END

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. As we've seen, this defaults to 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.

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:

Cluster-wide properties to configure:

- **yarn.scheduler.minimum-allocation-mb** (default 1024) -> 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 with allow 512, we'll set this to 512 as well.
- **yarn.scheduler.maximum-allocation-mb** (default 8192) -> 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 out HIGH_END class at 4096.

Class-specific properties to configure:

- **mapreduce.map.memory.mb** -> This property sets the resource limit for locally allocated map tasks.
- **mapreduce/reduce.memory.mb** -> This property sets the resource limit for locally allocated reduce tasks.
- **mapreduce.map.java.opts** -> Specifies the maximum JVM heap-size for map tasks so as to avoid problems brushing up against container limits.
- **mapreduce.map.java.opts** -> Specifies the maximum JVM heap-size for reduce tasks so as to avoid problems brushing up against container limits.
- **yarn.app.mapreduce.am.resource.mb** -> This setting specifies the resource limit for ApplicationMasters running on these nodes. Generally speaking we don't want ApplicationMasters running on weak nodes. An appropriate value is roughtly twice the amount of RAM bundled by a typical class container.
- **yarn.app.mapreduce.am.command-opts** -> Specifies the maximum JVM heap-size for ApplicationMasters running on these machines.
- **yarn.nodemanager.vmem-pmem-ratio** -> This property specifies the ratio of virtual to physical memory for containers. Obviously keeping data in memory will increase efficiency but at the cost of having to deal with less data, etc. simultaneously. The ApplicationMaster will remotely kill any containers that violate this ratio.