Spark cluster hardware setup

0. General rules for hardware setup

The general rule of thumb for a balanced Spark cluster is 4GB and 1 disk per CPU core. For disk-bound tasks, add more disks per CPU; CPU-bound tasks can get by with fewer disks per CPU. For memory-intensive jobs add more memory. Profile your application so you know what the limiting factor is.

1. Data source

Spark often needs to read data from external sources(e.g HDFS, HBase, etc).

1.1 HDFS as data source

If your spark cluster reads data mainly from HDFS, then the best solution is to install your spark workers on the hdfs data node. If your mapreduce and spark use Mesos or yarn as resource manager, you don't need to worry about how mapreduce and spark share the cpu and memory.

But if your spark cluster uses standalone mode, you need to configure the cpu and memory usage for both Mapreduce and spark.

For mapreduce in hadoop:

- mapred.child.java.opts: defines how many memory each task will use
- mapreduce.tasktracker.map.tasks.maximum: defines the max number of mapper task
- mapreduce.tasktracker.reduce.tasks.maximum: defines the max number of reduce task

For spark in standalone mode:

- SPARK_WORKER_CORES: It defines total number of cores to allow Spark applications to use on the machine (default: all available cores).
- SPARK_WORKER_MEMORY: It defines total amount of memory to allow Spark applications to use on the machine, e.g. 1000m, 2g (default: total memory minus 1 GiB); note that each application's individual memory is configured using its spark.executor.memory property.

For full spark standalone cluster configuration, check this Install spark on multi node mode

Normally, we don't recommend spark worker use all cpu and memory of the server. We should leave 10~20 percent to system. This can avoid server lock down.

Note if you can't run spark on the same server of the HDFS node, the spark cluster should be at least on the same local network.

1.2 Other external data source

For other external data source, there is no better solution than increase your network bandwidth.

2 Network

Once the data have been loaded to spark, the bottleneck of spark performance is usually the network. Because spark needs to do distributed reduce(e.g groupBy, reduceBy, sortBy, etc). These operations need to shuffle data between executors which are in different workers(different severs). **To accelerate the data transmission, we should have 10 GB or more bandwidth.**

You can use spark ui to check how many data have been transmitted due to spark shuffle.

3 Disks

While Spark can perform a lot of its computation in memory, **spark still uses local disks to store data that doesn't fit in RAM, as well as to preserve intermediate output between stages.** We recommend having 4-8 disks per node, configured without RAID (just as separate mount points). In Linux, **mount the disks with the noatime option to reduce unnecessary writes**. In Spark, configure the spark.local.dir variable to be a comma-separated list of the local disks. If you are running HDFS, it's fine to use the same disks as HDFS.

For example, we mount a disk(located /dev/vg01/lvol0) with gfs to directory /gfs1. with noatime option. For more info of gfs, plz visit https://en.wikipedia.org/wiki/GFS2

```
mount -t gfs /dev/vg01/lvol0 /gfs1 -o noatime
```

4 Memory

In general, Spark can run well with anywhere from 8 GiB to hundreds of gigabytes of memory per machine(physical or VM). In all cases, **we recommend allocating only at most 75% of the memory for Spark**; leave the rest for the operating system and buffer cache.

How much memory you will need will depend on your application. To determine how much your application uses for a certain dataset size, load part of your dataset in a Spark RDD and use the Storage tab of Spark's monitoring UI to see its size in memory. Note that memory usage is greatly affected by storage level and serialization format – see the tuning guide(https://spark.apache.org/docs/latest/tuning.html) for tips on how to reduce it.

Spark worker divide memory into working memory and storage memory evenly. For example, if we set spark worker has 8GB memory, it will have 4GB for calculation and 4 GB for storage.

```
# You can access spark ui web interface via following url

```

```
http://<driver-node>:4040
```

4.1 JVM memory limitation

Finally, **note that the Java VM does not always behave well with more than 200 GiB of RAM**. Because the default jvm garbage collector can't handle more than 200GiB of RAM. For more details about jvm GC, please visit Understanding Java Garbage Collection.

If you purchase machines with more RAM than this, you can launch multiple executors and/or workers in a single node. **In Spark's standalone mode, a worker is responsible for launching multiple executors according to its available memory and cores, and each executor will be launched in a separate Java VM.**

Normally we recommend you run only one worker per machine. But if you see many unused RAM, you can run multiple worker on one machine(Double check before you do this).

Standalone mode

In Spark Standalone mode we can edit conf/spark-env.sh to configure

```
# for example if we want 4 workers on the machine
SPARK_WORKER_INSTANCES=4
# each worker has 8 cores
SPARK_WORKER_CORES=8
# each worker has 8GB memory
SPARK_WORKER_MEMORY=8GB
```

Yarn mode

In yarn mode, yarn will be responsible for distributing memory to each executor.

```
# each EXECUTOR has 8 cores
SPARK_EXECUTOR_CORES=8
# each EXECUTOR has 8GB memory
SPARK_EXECUTOR_MEMORY=8GB
```

Change the default JVM GC to support large memory

In Java 8, the default GC was the **Parallel Garbage Collector**, which does not work well with more than 200GB ram.

We can use **G1 Garbage Collector** instead. For more details, please read this article https://databricks.com/blog/2015/05/28/tuning-java-garbage-collection-for-spark-applications.html

There are other GC available, you need to choose the most appropriate one based on your requirements.

5. CPU Cores

Spark scales well to tens of CPU cores per machine because it performs minimal sharing between threads. You should likely provision at least 8-16 cores per machine. Depending on the CPU cost of your workload, you may also need more: once data is in memory, most applications are either CPU- or network-bound.