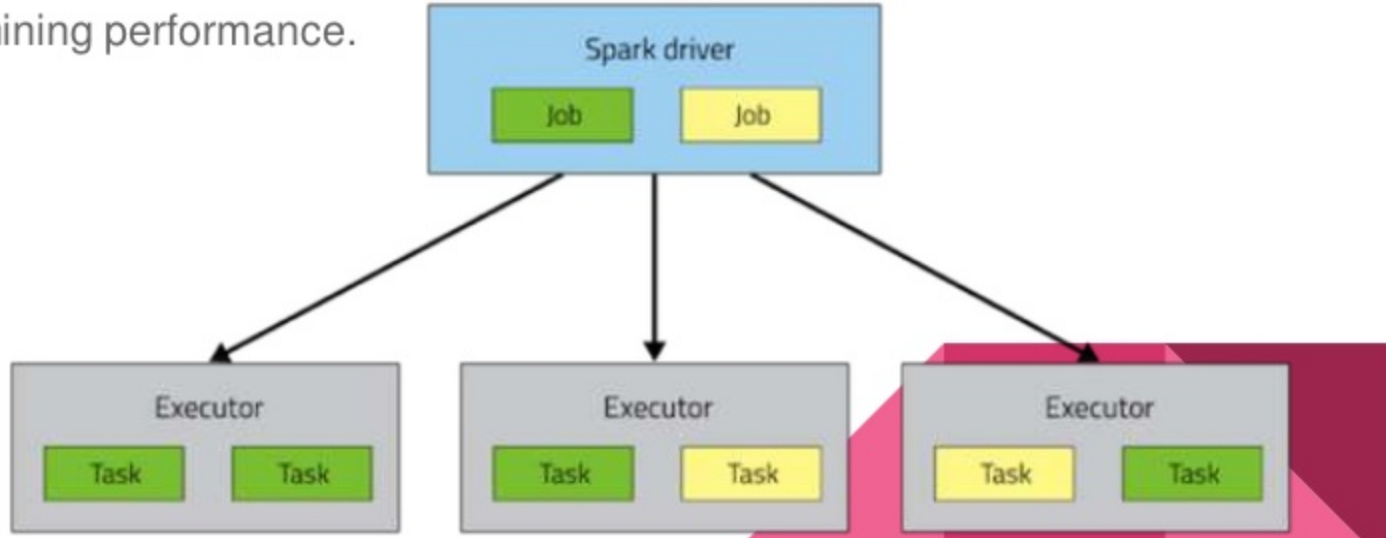
EMR Spark Tuning Demystified

Omid Vahdaty, Big Data Ninja

Tuning?

Spark, as you have likely figured out by this point, is a parallel processing engine. What is maybe less obvious is that Spark is not a “magic” parallel processing engine, and is limited in its ability to figure out the optimal amount of parallelism. Every Spark stage has a number of tasks, each of which processes data sequentially. In tuning Spark jobs, this number is probably the single most important parameter in determining performance.



What are the defaults?

- Thrift JDBC Server (Spark SQL)
- Zeppelin (Spark SQL)
- Yarn

Spark + Thrift

- **sudo /usr/lib/spark/sbin/start-thriftserver.sh --help**

config	Description	Default
--deploy-mode	Whether to launch the driver program locally ("client") or on one of the worker machines inside the cluster ("cluster")	Client
--driver-memory	Memory for driver (e.g. 1000M, 2G)	(Default: 1024M)
--driver-cores	Number of cores used by the driver, **only in cluster mode**	(Default: 1)
--executor-memory	Memory per executor (e.g. 1000M, 2G)	(Default: 1G).
--num-executors	**Number of executors to launch If dynamic allocation is enabled**, the initial number of executors will be at least default value .	(Default: 2)

EMR Spark

- https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-spark-configure.html
- In EMR cluster aws own logic and according to Slave instance type
- **Check If set maximizeResourceAllocation is enabled**
- Confirm setup via: nano /etc/spark/conf.dist/spark-defaults.conf

EMR Spark defaults

config	Description	Default for r4.4xl instance	Default for c4.8xl instance
Deploy mode	Client/Cluster		client
spark.driver.memory	this is set based on the smaller of the instance types in these two instance groups.		
spark.executor.cores	Setting is configured based on the slave instance types in the cluster.	32	4
spark.executor.memory	Setting is configured based on the slave instance types in the cluster.	218880M	5120M

Zeppelin

- Zeppelin interpreter overrides the spark cluster defaults
- https://spark.apache.org/docs/latest/configuration.html
- https://zeppelin.apache.org/docs/latest/interpreter/spark.html
- sqlContext.getConf("spark.executor.memory")
- sqlContext.getConf("spark.executor.cores")

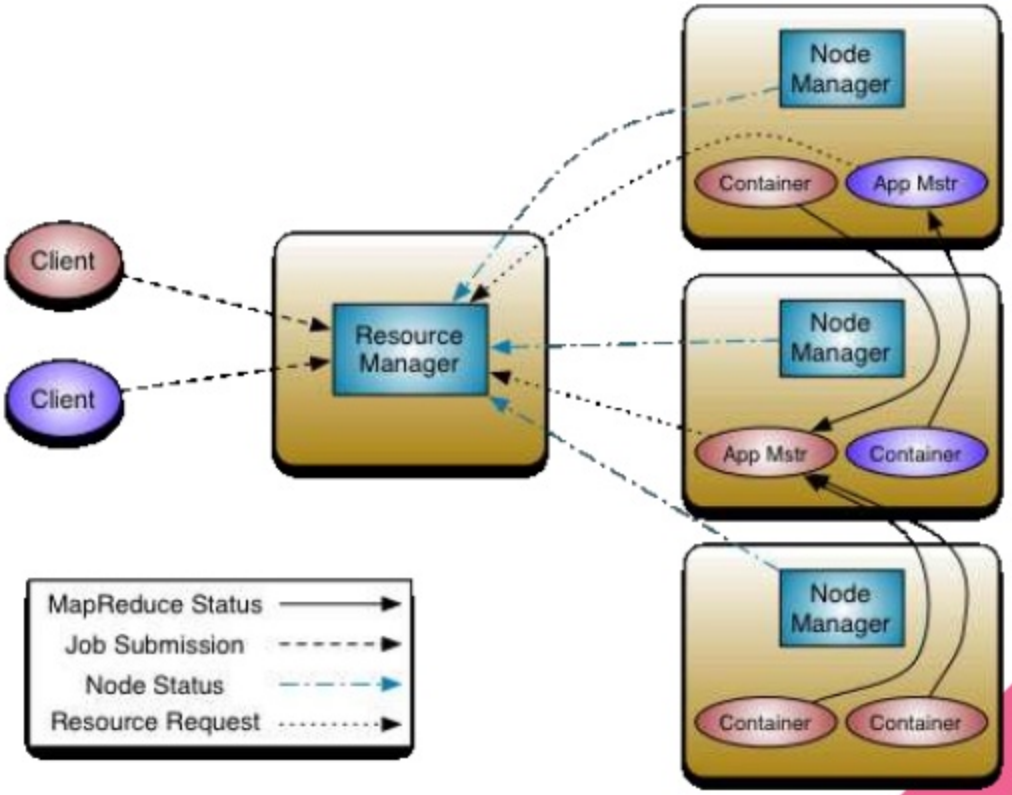
Zeppelin

config	description	default
zeppelin.spark.concurrentSQL	Running in parallel notebooks and paragraph	false
spark.app.name		zeppelin
spark.cores.max	Total number of cores to use.	Default: Empty , which means **uses all available core.**
spark.executor.instances	**(--num-executors in spark submit)**	
spark.executor.cores	**(--executor-cores)**	
spark.executor.memory	**(--executor-memory) (set using a G at the end for GB, like 16G)**	default 1G

Default Config Comparison Client mode (c4.8xl)

config	description	thrift	Zeppelin	Spark EMR
spark.executor.instances	**(--num-executors in spark submit) Number of executors to launch If dynamic allocation is enabled**, the initial number of executors will be at least default value .	2		
spark.executor.cores	**(--executor-cores)**	1	4	4
spark.executor.memory	**(--executor-memory)**	1G	5120M	5120M

What is yarn?


Client
Client
Resource Manager
Node Manager
Container
App Mstr
Node Manager
App Mstr
Container
Node Manager
Container
Container
MapReduce Status
Job Submission
Node Status
Resource Request

Yarn

- What is a container
  - A container is a YARN JVM process. In MapReduce the application master service, mapper and reducer tasks are all containers that execute inside the YARN framework. You can view running container stats by accessing Resource Managers web interface "http://<resource_manager_host>:8088/cluster/scheduler"
- When running Spark on YARN, each Spark executor runs as a YARN container. [...]
- YARN Resource Manager (RM) allocates resources to the application through logical queues which include memory, CPU, and disks resources.
  - By default, the RM will allow up to 8192MB *("**yarn.scheduler.maximum-allocation-mb**")* to an **Application Master (AM)** container allocation request.
  - The default minimum allocation is *("**yarn.scheduler.minimum-allocation-mb**")*.
  - The AM can only request resources from the RM that are in increments of ("**yarn.scheduler.minimum-allocation-mb**") and do not exceed *("**yarn.scheduler.maximum-allocation-mb**")*.
  - The AM is responsible for rounding off *("**mapreduce.map.memory.mb**") and ("**mapreduce.reduce.memory.mb**")* to a value divisible by the *("**yarn.scheduler.minimum-allocation-mb**")*.
  - RM will deny an allocation greater than 8192MB and a value not divisible by 1024MB in the following example.

Yarn Deploy mode: Cluster/Client

There are two deploy modes that can be used to launch Spark applications on YARN.

- **In cluster mode**, the Spark driver runs inside an application master process which is managed by YARN on the cluster, and the client can go away after initiating the application.
- **In client mode, the driver runs in the client process, and the application master is only used for requesting resources**
- **Confirm via:** http://ec2-34-245-34-125.eu-west-1.compute.amazonaws.com:18080/history/application_1519552338222_0001/environment/

Yarn Scheduler

- FAIR
- FIFO
- What is the default?
- default is FIFO in spark,
- FAIR in EMR : http://ec2-34-245-34-125.eu-west-1.compute.amazonaws.com:18080/history/application_1519552338222_0001/environment/)
- sqlContext.getConf("spark.scheduler.mode")
- Scheduling Queues
  - **Scheduling pools:**
  - https://spark.apache.org/docs/latest/job-scheduling.html#fair-scheduler-pools
  - **each pool has it own sub priorities/configuration:**
  - https://spark.apache.org/docs/latest/job-scheduling.html#configuring-pool-properties

Debugging an App

Debugging an app:

In YARN terminology, executors and application masters run inside "containers". YARN has two modes for handling container logs after an application has completed. If log aggregation is turned on (with the yarn.log-aggregation-enable config), container logs are copied to HDFS and deleted on the local machine. These logs can be viewed from anywhere on the cluster with the yarn logs command.

Current setup can be confirmed at:

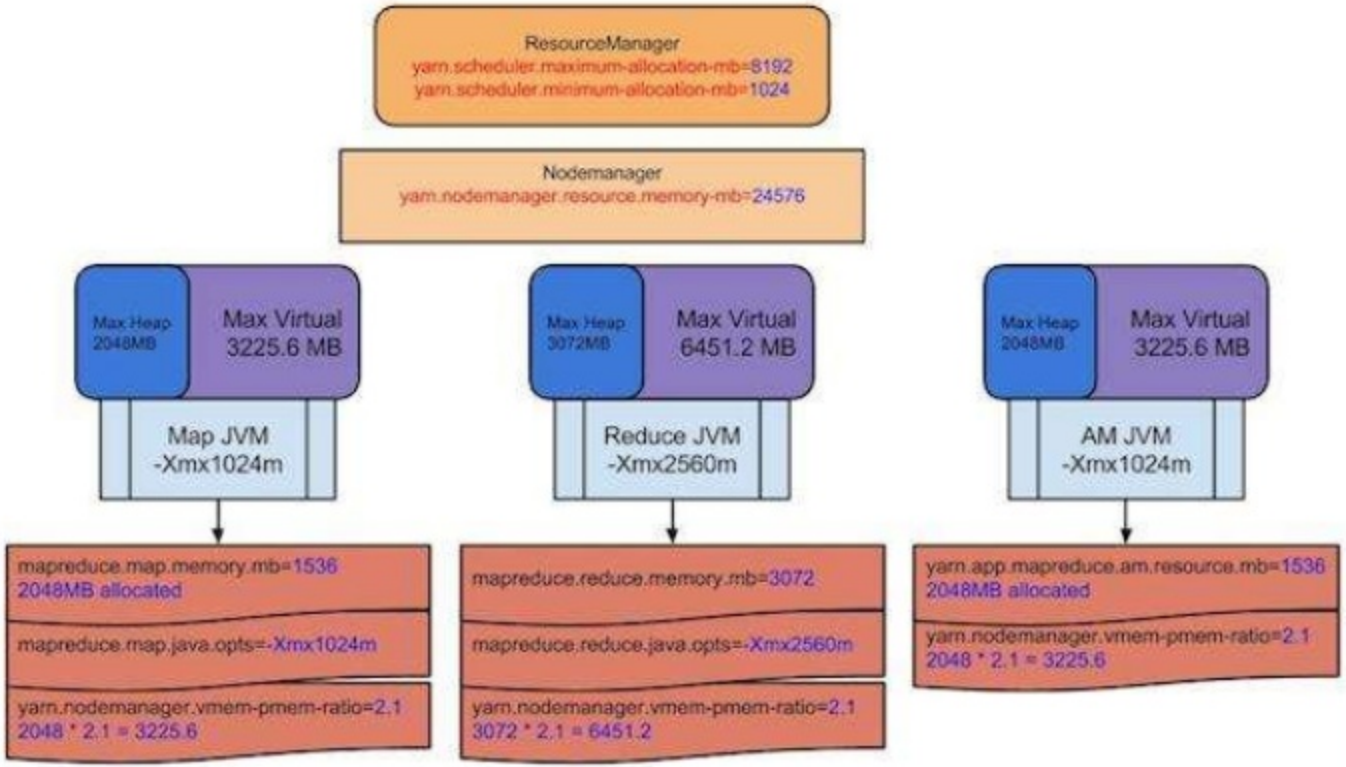
http://ec2-34-245-34-125.eu-west-1.compute.amazonaws.com:18080/history/application_1519552338222_0001/environment/

Yarn -default values per instance

Configuration Option		Default Value r 4.8xlarge	Default Value r 4.4xlarge	Default value c4.8xl
mapreduce.map.java.opts	Each mapper and reducer is a java process and we need some reserve memory to store the java code. Hence the Map/Reducer memory should be greater than the JVM heap size.	-Xmx6042m	-Xmx5837m	-Xmx1183m
mapreduce.reduce.java.opts		-Xmx12084m	-Xmx11674m	-Xmx2366m
mapreduce.map.memory.mb	is the upper memory limit that Hadoop allows to be allocated to a mapper	7552	7296	1479
mapreduce.reduce.memory.mb	is the upper memory limit that Hadoop allows to be allocated to a reducer	15104	14592	2958
yarn.app.mapreduce.am.resource.mb	criteria to select resource for particular job. Here is given 7552 Means any nodemanager which has equal or more memory available will get selected for executing job.	7552	14592	2958

Yarn -default values per instance

Configuration Option		r 4.8xlarge	Default Value r 4.4xlarge	Default value c4.8xl
yarn.scheduler.minimum-allocation-mb	Min ram allocation.	32	32	32
yarn.scheduler.maximum-allocation-mb	It defines the maximum memory allocation available for a container in MB it means RM can only allocate memory to containers in increments of `"yarn.scheduler.minimum-allocation-mb"` and not exceed `"yarn.scheduler.maximum-allocation-mb"` and It should not be more then total allocated memory of the Node.	241664	116736	53248
yarn.nodemanager.resource.memory-mb	Amount of physical memory, in MB, that can be allocated for containers. It means the amount of memory YARN can utilize on this node and therefore this property should be lower then the total memory of that machine.	241664	116736	53248


ResourceManager
yarn.scheduler.maximum-allocation-mb=8192
yarn.scheduler.minimum-allocation-mb=1024
Nodemanager
yarn.nodemanager.resource.memory-mb=24576
Max Heap 2048MB
Max Virtual 3225.6 MB
Map JVM -Xmx1024m
mapreduce.map.memory.mb=1536
2048MB allocated
mapreduce.map.java.opts=-Xmx1024m
yarn.nodemanager.vmem-pmem-ratio=2.1
2048 * 2.1 = 3225.6
Max Heap 3072MB
Max Virtual 6451.2 MB
Reduce JVM -Xmx2560m
mapreduce.reduce.memory.mb=3072
mapreduce.reduce.java.opts=-Xmx2560m
yarn.nodemanager.vmem-pmem-ratio=2.1
3072 * 2.1 = 6451.2
Max Heap 2048MB
Max Virtual 3225.6 MB
AM JVM -Xmx1024m
yarn.app.mapreduce.am.resource.mb=1536
2048MB allocated
yarn.nodemanager.vmem-pmem-ratio=2.1
2048 * 2.1 = 3225.6

Yarn Tuning

- 1 c4.8xlarge master

- 4 c4.8xlarge instances

- each with 36 CPU cores

- and each with 60 GB RAM.

First, The master will not count towards the calculation since it should be managing and not running jobs like a worker would.

Before doing anything else, we need to look at YARN, which allocates resources to the cluster. There are some things to consider:

Yarn Tuning

1) To figure out what resources YARN has available, there is a chart for emr-5.11.0 showing that

2) By default (for c4.8xlarge in the chart), YARN has 53248 MB (52 GB) max and 32 MB min available to it.

3) The max is supposed to be cleanly divisible by the minimum (1664 in this case), since memory incremented by the minimum number.

4) YARN is basically set for each instance it is running on (Total RAM to allocate to YARN to use).

5) Be sure to leave at least 1-2GB RAM and 1 vCPU for each instance's O/S and other applications to run too. The default amount of RAM (52 GB out of 60 GB RAM on the instance) seems to cover this, but this will lave us with 35 (36-1) vCPUs per instance off the top

6) If a Spark container runs over the amount of memory YARN has allocated, YARN will kill the container and retry to run it again.

7) **Also, if this works and you want to tune further, you can see about increasing resources available to YARN (Like using perhaps 58 GB of RAM instead of 52 GB) and test those too.**

When it comes to executors

1) This is set for the entire cluster. it doesn't have to use all resources

2) Likely, you are going to want more than 1 executor per node, since parallelism is usually the goal. This can be experimented with to manually size a job, again it doesn't even have to use all resources.

3) **If we were to use the article [1] as an example, we may want to try an do 3 executors per node to start with (Not too few or too many), so --num-executors 11 [3 executors per node x 4 nodes = 12 executors, -1 for application master running = 11].**

And Executors cores...

1) Although it doesn't explicitly say it, I have noticed problems working with other customers, when total executors exceeds the instance's vCPU / core count.

2) It is suggested to put aside at least 1 executor core per node to ensure there are resources left over to run O/S, App Master and the like.

3) **Executor cores are being assigned per executor** (discussed just above). So this isn't total per node or for the cluster, but rather depending on the resources available divided by the number of executors wanted per node.

4) In the article's example this works out to --executor-cores= 11 [36 cores/ vCPU per instance - 1 for AM/overhead = 35 cores. 35 cores/ 3 executors per node = 11 executor cores]

And Executors memory...

1) This is assigned like the executor cores (finally, something sized similarly). This is per executor and thus is not the total amount of memory on a node, or set in YARN, but rather depending on the resources available divided by the number of executors again (Like executor-cores)

2) Sometimes, some off-heap memory and other things can make an executor/container larger than the memory set. Leave room so YARN doesn't kill it due to using more memory than is provided.

3) **In the article's example this works out to --executor-memory 16G (16384 MB) [52 GB provided to YARN / 3 executors = 17 GB RAM max. 17 GB-1 GB for extra overhead = 16GB (16384 MB)]**

So, putting the above together as an example, we could use 3 executors per node with: --num-executors 11 --executor-cores 11 --executor-memory 16G

AND EXECUTOR.INSTANCES

If **Spark dynamic allocation is enabled**, the initial number of executor instances is based on the Spark default value for **spark.dynamicAllocation.initialExecutors** which in turn depends on the **spark.dynamicAllocation.minExecutors** which is **0**.

If you set --num-executors in the command line then the initial executors started will be equal to that value.

if **MaximizeResourceAllocation** is set then EMR service will set the spark.executor.instances to the number of core nodes in the cluster.

If Dynamic allocation is also set then the initial number of executors started will be the number of core nodes in the cluster.

Yarn Deploy mode tuning

1) **When we use --deploy-mode client, we tell YARN to deploy to an 'external node', but since this is being run from the master, it is straining the master for the entire cluster. Client mode is best if you have spark-shell on another instance in the same network and can submit a job from there to the cluster but use the instance's resources for the driver (taking the load off the cluster) or when using it interactively to test since the application master is on the master instance and its logs will be there).**

2) We probably want to use YARN as our deployment mode [5]. In particular, we should be looking at yarn-cluster. This means each container's App master is deployed in the cluster and not just on one device (That is also managing the whole cluster). The resources running are spread out, but the logs are too for the application masters). There's another good explanation on this available [6].

3) Remember, if you over-subscribe the driver memory and cores, there's less for the rest of the cluster to use for actual work. Most often, the driver memory is set to 1-2GB; 20GB would take more than half the RAM and 4 cores would take most of the cores in our earlier example. I t may be worth leaving these as their defaults (not using the options) for now and then conservatively raising them if the logs inform you to do so.

EMR and s3

* Use S3DistCp to move objects in and out of Amazon S3. S3DistCp implements error handling, retries and back-offs to match the requirements of Amazon S3. For more information, **see Distributed Copy Using S3DistCp**.
* Design your application with eventual consistency in mind. **Use HDFS for intermediate data storage while the cluster is running and Amazon S3 only to input the initial data and output the final results.**
* **If your clusters will commit 200 or more transactions per second to Amazon S3,** contact support to prepare your bucket for greater transactions per second and consider using the key partition strategies described in Amazon S3 Performance Tips & Tricks.
* **Set the Hadoop configuration setting io.file.buffer.size to 65536. This causes Hadoop to spend less time seeking through Amazon S3 objects.**
* Consider disabling Hadoop's speculative execution feature if your cluster is experiencing Amazon S3 concurrency issues. You do this through the mapred.map.tasks.speculative.execution and mapred.reduce.tasks.speculative.execution configuration settings. This is also useful when you are troubleshooting a slow cluster.
* If you are running a Hive cluster, see Are you having trouble loading data to or from Amazon S3 into Hive?.
* **Pre-cache the results of an Amazon S3 list operation locally on the cluster. You do this in HiveQL with the following command:** `set hive.optimize.s3.query=true;`.

- **Apply compression testing**…
- **Use s3a**

References:

[1] https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-troubleshoot-error-hive.html#emr-troubleshoot-error-hive-3
[2] https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-troubleshoot-errors-io.html#emr-troubleshoot-errors-io-1

how can i undo MaximizeResourceAllocation ?

When MaximizeResourceAllocation is enabled, 5 parameters value are set under spark-defaults.conf. To undo MaximizeResourceAllocation, we need to adjust the spark.executor.cores and spark.executor.memory parameters etc according to your use case.

https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-spark-configure.html

what would be a recommend setting for working a cluster in parallel for 2 and more users. should we turn off MaximizeResourceAllocation? turn on spark.dynamicAllocation ?

it depends on the use-case such as

- how many applications you want to run in parallel
- how many tasks their application is expected to launch.
- Disabling MaximizeResourceAllocation helps to run applications in parallel. But disabling the parameter will affect the application performance.

Example of tuning

Spark resource tuning is essentially a case of fitting the number of executors we want to assign per job to the available resources in the cluster. If we assume your **Core and Task nodes** are 40 (**20+20)** instances of type **m3.xlarge.**

each m3.xlarge instance has **4 vCPUs and 15GiB** of RAM available for use [1]. Since we are going to leave 1 vCPU per node free as per the initial issue, this gives us a total cluster vCPU resources of:

40 x 3 vCPUs = 120vCPUs

For m3.xlarge instance types, t**he maximum amount of RAM available for each node is controlled by the property "yarn.nodemanager.resource.memory-mb" located in the /etc/hadoop/conf/yarn-site.xml file**, located on the master instance node. It is **set by default to 11520** (**even with the maximumResourcesAllocation enabled**). This allows for some RAM to be left free on each node for other tasks such as system procedures, etc.

(40 x 11520MB RAM) / 1024 (to convert to GiB) = 450GiB RAM*

* Note: this does not mean that it is OK to divide this 450GiB by 40 and to set each executor memory to this value. This is because this maximum is a hard maximum and **each executor requires some memoryOverhead to account for things like VM overheads**, interned strings, other native overheads, etc.[2]. Typically, **the actual maximum memory set by --executor-memory is 80% of this value**.

Example of tuning

	Cpu	memory	comments
Per Instnace	4	15	**M3.xlarge., arn.nodemanager.resource .memory-mb = 11520MB, each executor requires some memoryOverhead to account for things like VM overheads. the actual maximum memory set by --executor-memory is 80% of this value.**
Per Cluster (20 tasks+ 20 core=40)	40 * (4-1)= **120**	**(40 x 11520MB RAM) / 1024 (to convert to GiB) =** 450GiB **RAM***	**-1 core for the OS and daemon**

Example of tuning

	Executor cores	Executor memory	comments
Per executor	3	80% of yarn (11.5GB)= 9GB Or 60% of machine total (15B)= 9GB.	**M3.xlarge., arn.nodemanager.resource .memory-mb = 11520MB, each executor requires some memoryOverhead to account for things like VM overheads. the actual maximum memory set by --executor-memory is 80% of this value.**
Per Cluster (20 tasks+ 20 core=40)	40 * (4-1)= 120	**40*9GB = 360GB.**	**-1 core for the OS and daemon**

Example of tuning

Using this as a simple formula, we could have many examples such as (using your previously provided settings as a basis):

1) 1 Spark-submit Job -> --executor-cores 3 --executor-memory 9g --conf spark.dynamicAllocation.maxExecutors=40 --conf spark.yarn.executor.memoryOverhead=2048

2) 2 Spark-submit Jobs -> --executor-cores 1 --executor-memory 4500m --conf spark.dynamicAllocation.maxExecutors=80 --conf spark.yarn.executor.memoryOverhead=1024

(--executor-cores are 1 here as 3/2 = 1.5 and decimals are rounded down)

3) 3 Spark-submit Jobs -> --executor-cores 1 --executor-memory 3000m --conf spark.dynamicAllocation.maxExecutors=120 --conf spark.yarn.executor.memoryOverhead=768

Example of tuning

Example	Executors Cores	Executors-mem	spark.dynamicAllocation.maxExecutors	spark.yarn.executor.memoryOverhead	comments
job1	3	9GB	40	2048	20% OVERHEAD
Job2 (half)	1	4.5GB	80	1024	-executor-cores are 1 here as 3/2 = 1.5 and decimals are rounded down
Job3 (third)	1	3GB	120	768	

Example of tuning - double the instance size...

Lets double the instance size and see what happens...

On a m3.2xlarge with double the resources per node we would have 40 x 8 vCPUs and 40 x 30GiB RAM. Again, leaving 1 vCPU free per node (8-1=7 vCPUs per node) and trusting that yarn limits the max used RAM to 23GiB (23552MiB) or so, our new values would look like:

1) 1 Spark-submit Job -> --executor-cores 7 --executor-memory 19g --conf spark.dynamicAllocation.maxExecutors=40 --conf spark.yarn.executor.memoryOverhead=4096

(19456MiB + 4096MiB) * 1 overall Job = 23552MiB => A perfect fit!

23552 * 0.8 = 18841.6MiB so we can probably round this up to 19GB of RAM or so without issue. The majority of the memory discrepency between 23-19 (4GiB) can be set in the memoryOverhead parameter)

2) 2 Spark-submit Jobs -> --executor-cores 3 --executor-memory 9g --conf spark.dynamicAllocation.maxExecutors=80 --conf spark.yarn.executor.memoryOverhead=2560

(9216 + 2560) *2 Jobs to run = 23552MiB => A perfect fit!

(--executor-cores are 3 here as 7/2 = 3.5 and decimals are rounded down)

So for the example:

"3) 3 Spark-submit Jobs -> --executor-cores 2 --executor-memory 6g --conf spark.dynamicAllocation.maxExecutors=120 --conf spark.yarn.executor.memoryOverhead=1536"

We have 3 jobs with a maximum executor memory size of 6GiB. This is 18GiB that fits into the 23GiB. Of the 5GiB remaining, we allocate that as memoryOverhead but with an equal share per job. This means that we set this value to 1536MB RAM to use the remaining RAM, except for 512MiB.

As you can see above, the calculation is that the per job is:

(executor-memory + memoryOverhead) * number of concurrent jobs = value that must be <= Yarn threshold.

(6144MB + 1536MB)*3 = 23040MiB which is 512MiB less than the Yarn threshold value of 23552MiB (23GiB).

Example of tuning - double the instance size...

	Cpu	memory	comments
Per Instance	8	30	**M3.xlarge., arn.nodemanager.resource .memory-mb = 23522MB, each executor requires some memoryOverhead to account for things like VM overheads. the actual maximum memory set by --executor-memory is 80% of this value.**
Per Cluster (20 tasks+ 20 core=40)	40 * (8-1)= **280**	**(40 x 23.5GB) = 940GiB RAM***	**-1 core for the OS and daemon**

Example of tuning- double the instance size...

	Executor cores	Executor memory	comments
Per executor	7	80% of yarn (23.5GB)= 18GB Or 60% of machine total (30GB)= 18GB.	**M3.xlarge., arn.nodemanager.resource .memory-mb = 11520MB, each executor requires some memoryOverhead to account for things like VM overheads. the actual maximum memory set by --executor-memory is 80% of this value.**
Per Cluster (20 tasks+ 20 core=40)	40 * (4-1)= 280	**40*18GB = 720GB.**	**-1 core for the OS and daemon**

Example of tuning - double the instance size...

Example	Executors Cores	Executors-mem	spark.dynamicAllocation.maxExecutors	spark.yarn.executor.memoryOverhead (in MB)	comments
job4	7	19GB	40	4096	(19456MiB + 4096MiB) * 1 overall Job = 23552MiB => A perfect fit! 23552 * 0.8 = 18841.6MiB so we can probably round this up to 19GB of RAM or so without issue. The majority of the memory discrepency between 23-19 (4GiB) can be set in the memoryOverhead parameter)
Job5 (half)	3	9GB	80	2560	(9216 + 2560) *2 Jobs to run = 23552MiB => A perfect fit! (--executor-cores are 3 here as 7/2 = 3.5 and decimals are rounded down)
Job6 (third)	2	6GB	120	1536	(--executor-cores are 2 here as 7/3 =

How much memory per executor? Alot? alittle?

- Running executors with too much memory often results in **excessive garbage collection delays**. **64GB is a good upper limit for a single executor.**
- I've noticed that the **HDFS client** has trouble with tons of **concurrent threads**. A rough guess is that at most **five tasks per executor** can achieve **full write throughput**, so it's good to **keep the number of cores per executor** below that number.
- Running **tiny executors** (with a single core and just enough memory needed to run a single task, for example) **throws away the benefits that come from running multiple tasks in a single JVM**. For example, broadcast variables need to be replicated once on each executor, so **many small executors will result in many more copies of the data.**

Is it right to say the setting maximizeResourceAllocation should be used in all cases, or if not, what are the reasons not to use it?"

Using the dynamicAllocation settings . we can better determine the number of overall resources that we have and how many executors that we want to give to each job.

For example, leaving the cluster as-is using only the original job per cluster, we would naturally want to allocate all of our resources to this one job. If the job runs better using large executors, we would then allocate all of our resources to this job. We could do this by creating the cluster using the maximizeResourceAllocation property set to true which would automatically tune our clusters to use the most resources available [8]. **Your existing settings of "--executor-memory 9g --executor-cores 3 --conf spark.yarn.executor.memoryOverhead=2048" would be sufficient for this task, though not actually necessary if the maximizeResourceallocation option is set. You can see the default settings that the maximizeResourceAllocation options sets in the /etc/spark/conf/spark-defaults.conf file on your master node.**

1 Job, smaller number of larger executor

spark-submit --master yarn --executor-cores 3

--conf maximizeResourceAllocation=true

--conf spark.dynamicAllocation.minExecutors=1

--conf spark.dynamicAllocation.maxExecutors=40

--conf spark.dynamicAllocation.initialExecutors=40

--conf spark.dynamicAllocation.enabled=true

--conf spark.driver.maxResultSize=10G

--conf spark.akka.frameSize=1000

Here, Spark dynamicAllocation will allocate the maximum available RAM for you for your executor. We will be running 40 individual executors that have 3 vCPUs each. However, say your job runs better with a smaller number of executors?

1 Job, greater number of smaller executors:

In this case you would simply set the dynamicAllocation settings in a way similar to the following, but adjust your memory and vCPU options in a way that allows for more executors to be launched on each node. So for twice the executors on our 40 nodes, we need to half the memory and the vCPUs in order to make the 2 executors "fit" onto each of our nodes.

spark-submit --master yarn --executor-cores 1 --executor-memory=4500m

--conf spark.yarn.executor.memoryOverhead=1024

--conf spark.shuffle.service.enabled=true

--conf spark.dynamicAllocation.minExecutors=1

--conf spark.dynamicAllocation.maxExecutors=80

--conf spark.dynamicAllocation.initialExecutors=20**

--conf spark.dynamicAllocation.enabled=true

--conf spark.driver.maxResultSize=10G

2 Jobs, equal number of 1 larger executor per node per job

spark-submit --master yarn --executor-cores 1 --executor-memory=4500m

--conf spark.yarn.executor.memoryOverhead=1024

--conf spark.shuffle.service.enabled=true

--conf spark.dynamicAllocation.minExecutors=1

--conf spark.dynamicAllocation.maxExecutors=40

--conf spark.dynamicAllocation.initialExecutors=20

--conf spark.dynamicAllocation.enabled=true

--conf spark.driver.maxResultSize=10G

2 Jobs, equal number of 1 larger executor per node per job

The main difference here is the setting of the value "--conf spark.dynamicAllocation.maxExecutors=40". We need to set this value to 40 as our memory limit is essentially halved as we are running 2 separate jobs on each node. Hence, our total resources on our cluster effectively halves to 80vCPUs and 225GiB RAM per Spark-submit job.So to fit in our 2 jobs into these newer limits, it gives us again 1vCPU per executor:

80 vCPUs / 40 nodes = 2 vCPUs per job. Since we need to leave 1 of the 4 cores free per node, that gives us 4-1=3 vCPUs per node divided by 2 jobs is 1.5. We round down then to 1 vCPU per job.

225 * 0.8 (to get 80% of settable memory) = 180GiB per job. 180 / 40 nodes = 4.5GiB per job per node.

By setting the value of spark.dynamicAllocation.maxExecutors to 40 for each job, it ensures that neither job encroaches into the total resources allocated for each job. By limiting the maxExecutors to 40 for each job, the maximum resources available per jobs is:

40 X 4500MiB RAM (divided by 1000 to convert to GiB) = 180GiB per job. 40 x 1 vCPU = 40vCPUs per job.

2 Jobs means that in total 360GiB RAM is used (80% of total RAM as expected) and 40vCPUs * 2 jobs is 80 vCPUs which again, fits in with the maximum vCPUs for the cluster.

The difference between Example 3 and Example 2 is that in Example 2, all of the resources are being used by 2 executors on each node for the 1 job. In Example 2, your job would have a maximum of 80 executors (40 nodes * 2 executors per node) running on the same 1 job. In Example 3, your 2 jobs would have 40 executors maximum (40 nodes * 1 executor per node) running on each job concurrently.

NOTE 1

The --executor-cores value was only set to 1. This is because since we are only using 3 out of the 4 vCPUs per node, and 3/2 = 1.5, decimal values are not allowed in this parameter. This means that a whole vCPU core is going unused.

In Example 2 above, you could simply lower the --executor-memory value to 3GiB and increase the --conf spark.dynamicAllocation.maxExecutors value to 120 to use this core.

"Please note that with the reduced CPU power per node, it may be worthwhile either increasing your instance count or instance type in order to maintain the speeds that you want. Increasing the instance type though (say for example, to a m3.2xlarge type) also has the benefit of though of being able to use more of the nodes CPU for the executors. The OS only needs a small bit of CPU free, but the spark tunings only let us specify the CPU to use in terms of cores.

Hence, on a m3.xlarge instance, 1 core left free from the 4 cores is 25% of your CPU resources left free but on a m3.2xlarge, leaving 1 core free from the 8 cores is only 12.5% of your CPU resources left free, and so on."

You may need to balance the instance types to the instance count in order to get a "best case" usage for your resource requirements.

NOTE 2

You could consider **trying to amalgamate both of your jobs into the one** .jar file that is being called. While this would likely require more coding and tweaking of your application, doing so would allow you to allow the Spark dynamicAllocation process to effectively "load balance" the number of executors across your jobs.

Instead of running your 2 jobs concurrently in two separate spark-submit calls, using the 1 spark-submit call using settings similar to Example 1 above, would allow the resources from one job to be used on the other job if one job was to finish earlier than the other.

Of course you have less control over which job gets more exclusive resources than the other, but it may mean that if using Example 3 above (as an example) resources left over from the end of Job 1 could be used to help speed up the remainder of Job 2, and vice-versa.

NOTE 3

If running the jobs on the same cluster, please note that the job files will be grouped together. This can make debugging issues in your jobs more difficult as there will be less separation of the job logs. If you want a greater level of job log separation, then I would suggest that you perhaps consider running each job on a separate EMR cluster. This will also give you an extra cluster to monitor and will also give you separate CloudWatch metrics per job, instead of having the 1 set of metrics for both jobs. This will make monitoring your jobs a lot easier as the metrics per cluster will be related to only one of your jobs. If there is an issue with either job, it will be a lot easier to debug the issue as it would be a lot harder to see from the CloudWatch metrics which specific job was causing the issue.

Links:

[1] Amazon EC2 Instance Types: https://aws.amazon.com/ec2/instance-types/

[2] Running Spark on yarn: http://spark.apache.org/docs/1.6.1/running-on-yarn.html

[3] https://spark.apache.org/docs/1.6.1/configuration.html#dynamic-allocation

[4] How-to: Tune Your Apache Spark Jobs (Part 1): http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-1/

[5] How-to: Tune Your Apache Spark Jobs (Part 2): http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/

[6] Tuning Spark: http://spark.apache.org/docs/1.6.1/tuning.html

[7] Submitting User Applications with spark-submit: https://blogs.aws.amazon.com/bigdata/post/Tx578UTQUV7LRP/Submitting-User-Applications-with-spark-submit

[8] Configure Spark (EMR 4.x Releases): http://docs.aws.amazon.com/ElasticMapReduce/latest/ReleaseGuide/emr-spark-configure.html#spark-dynamic-allocation

[9] http://c2fo.io/c2fo/spark/aws/emr/2016/07/06/apache-spark-config-cheatsheet/ - **Sam ,it would be important for your use case

---

==========

Resource

[1] http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/

[2] https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-hadoop-task-config.html#emr-hadoop-task-jvm

[3] http://spark.apache.org/docs/latest/submitting-applications.html

[4] http://spark.apache.org/docs/latest/cluster-overview.html

[5] http://spark.apache.org/docs/latest/running-on-yarn.html

[6] https://www.cloudera.com/documentation/enterprise/5-5-x/topics/cdh_ig_running_spark_on_yarn.html

[7] http://docs.aws.amazon.com/emr/latest/ReleaseGuide/latest/emr-spark-configure.html

[8] https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-zeppelin.html#zeppelin-considerations

[9] http://agrajmangal.in/blog/big-data/spark-parquet-s3/

[10] https://docs.aws.amazon.com/emr/latest/ReleaseGuide/UsingEMR_s3distcp.html