**EMR Cluster Capacity Planning**

This document describes the strategy for sizing the cluster based on input data volume and spark workload types.

**Instance Type Selection:**

We will use the latest version of EMR and start with a general purpose **m5.8xlarge** instance type. Identify the system bottlenecks(CPU, memory, storage and I/O) from OS and YARN metrics(Ganglia and CloudWatch) and then select the appropriate instance type. Use storage optimized instances or SSD based storage(**r5d’s and r6gd’s**) for spark jobs which are **I/O and or shuffle intensive.**

If theintermediate data from transformations is much larger than the final output, we can leverage HDFS as temporary storage and then use S3DistCp to copy the data into the final location once your application is finished.

**Maximize Cluster Resource Utilization:**

**Job Specific Transient Cluster:**

We will set **maximizeResourceAllocation=true** in **spark** configuration classification with EMR Auto Scaling enabled. We will also set **spark.dynamicAllocation.enabled=false** in **spark-defaults** configuration classification. EMR will calculate the maximum compute and memory resources available for an executor on an instance in the core instance group. It then sets the following spark-defaults settings based on the calculated maximum values.

1. spark.default.parallelism
2. spark.driver.memory
3. spark.executor.memory
4. spark.executor.cores

**Shared Cluster:**

We will set **maximizeResourceAllocation=false** in **spark** configuration classification. We already observed input data volume changes every day. We will use EMR Auto Scaling and Spark Dynamic Resource Allocation to handle unpredictable workloads and maximize cluster resource utilization.

**Executor Memory Calculation:**

We will start with **spark.executor.core=4** (can also be tried with 2,3,5) and calculate the **spark.executor.memory** based on the value of **yarn.nodemanager.resource.memory-mb** for the selected instance type.

Cluster Info for example

|  | **Primary Node** | **Core Node** |
| --- | --- | --- |
| **No of Node** | 1 | 10 |
| **Instance Type** | m5.8xlarge | m5.8xlarge |
| **vCPU per node** | 32 | 32 |
| **Memory per node(GB)** | 128 | 128 |
| **yarn.nodemanager.resource.memory-mb** | 122880 | 122880 |
| **spark.yarn.executor.memoryOverheadFactor** | 0.1875 | 0.1875 |

We have to reserve 1 core and 1 GB of memory on each node for Hadoop yarn daemon processes.

**No of executor per node** = (vCPU per node - 1) / (spark.executor.cores) = (32-1)/4 = 7

spark.executor.memory + (spark.executor.memory \* spark.yarn.executor.memoryOverhaedFactor) = (12880 MB / 7) = 17554.28 MB

**spark.executor.memory** = 17554.28 - (17554.28 \* 0.1875)=17554.28 - 3291.42 = 14262.85=**14GB**

**spark.executor.instances =** (No of node \* No of executor per node) - (1 driver)=10 \* 7 - 1 = 69

**Configure** **spark.sql.shuffle.partitions:**

Target size of each shuffle partition should be **100 MB - 200 MB.**

We will assume the following values to calculate the **spark.sql.shuffle.partitions**

| **Target shuffle partition size** | **Incoming data size to the shuffle stage** |
| --- | --- |
| 160MB | 100GB |

Estimated Shuffle Partition Size = Incoming data size to shuffle stage / Target shuffle partition size

Estimated Shuffle Partition Size = 100 GB / 160 MB = **640**

Now set the **spark.sql.shuffle.partitions** as

| **If Estimated Shuffle Partition Size** | **spark.sql.shuffle.partitions** |
| --- | --- |
| **<** total no of executor core in the cluster | total no of executor core in the cluster |
| **>** total no of executor core in the cluster | Multiple (1,2) of total no of executor core in the cluster. |

Now if the Estimated Shuffle Partition Size > 3 \* (the total no of executor core in the cluster), then add more nodes to the cluster to increase parallelism.

In this example we have **69 executors with 4 core** each, so total number of core to execute all the tasks is = 69 \* 4 = **276**

640 > 276, so 640 / 276 = 2.31

So we can set the **spark.sql.shuffle.partitions** = 276 \* 2 = **552**

**Spark Dynamic Allocation in Shared Cluster:**

Enable spark dynamic allocation with the following configurations when the input data volume is unpredictable and EMR Auto-Scaling is turned on.

| spark.executor.cores | 4 |
| --- | --- |
| spark.executor.memory | 14GB |
| spark.dynamicAllocation.enabled | True |
| spark.dynamicAllocation.minExecutors | 0 |
| spark.dynamicAllocation.maxExecutors | 69 |
| spark.dynamicAllocation.initialExecutors | 1 |
| spark.dynamicAllocation.shuffleTracking.enabled | True |
| spark.dynamicAllocation.executorIdleTimeout | 60s |
| spark.dynamicAllocation.cachedExecutorIdleTimeout | 60s |

**References**

1. [**https://aws.github.io/aws-emr-best-practices/applications/spark/best\_practices/#bp-516-tune-driverexecutor-memory-cores-and-sparksqlshufflepartitions-to-fully-utilize-cluster-resources**](https://aws.github.io/aws-emr-best-practices/applications/spark/best_practices/#bp-516-tune-driverexecutor-memory-cores-and-sparksqlshufflepartitions-to-fully-utilize-cluster-resources)
2. [**https://spoddutur.github.io/spark-notes/distribution\_of\_executors\_cores\_and\_memory\_for\_spark\_application.html**](https://spoddutur.github.io/spark-notes/distribution_of_executors_cores_and_memory_for_spark_application.html)
3. [**https://www.youtube.com/watch?v=\_ArCesElWp8&t=4512s**](https://www.youtube.com/watch?v=_ArCesElWp8&t=4512s)
4. [**https://granulate.io/blog/aws-emr-cluster-viewing-managing-scaling-your-clusters/**](https://granulate.io/blog/aws-emr-cluster-viewing-managing-scaling-your-clusters/)