**Spark Performance Tuning Techniques**

**>>How to choose No of Executors and Memory :**

6 Nodes

16 Cores (16 \* 6 = 96 cores)

64 BG per node

**Smallest possible executor :**

1 core / 4 GB RAM

So 16 executors on each machine

In above approach we are not utilizing multi processing capability of JVM.

only with limited resources each executor will perform worse

**Largest possible Executor :**

16 cores/64 GB RAM for executor

1 executor per node

Here problem is any single executor in any node will consume all memory and all cores.

There is lot of HDFS IO contention : Snce all cores will compete each other to get the resources for getting the task done. Read and write will become very slow.

No resources for OS

No Memory overhead for Yarn and spark (off heap memory)

**Optimal way :**

Executor and Executor Core :

No Of Cores : 96 - No Of Machines(nodes) : 6 = 90

No of Cores per machine : 90/6 = 15 cores

No of executor cores : 5

So no of executor per machine : 15/5 = 3

**Memory :**

Available per machine = 64 -1 = 63 GB

Available per executor = 63/3 = 21 GB

Yarn Overhead = 2 GB (7% of executor memory)

Per Executor m0emory = 19 GB

*Full memory requested to yarn per executor =*

*spark-executor-memory + spark.yarn.executor.memoryOverhead.*

*spark.yarn.executor.memoryOverhead =*

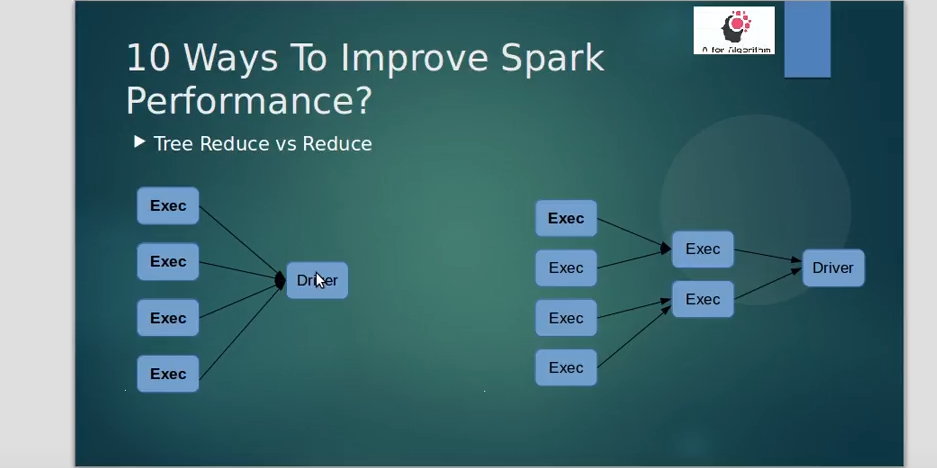
*Max(384MB, 7% of spark.executor-memory)*

*So, if we request 20GB per executor, AM will actually get 21GB + memoryOverhead = 20 + 7% of 20GB = ~23GB memory for us.*

**Reference:**

<https://spoddutur.github.io/spark-notes/distribution_of_executors_cores_and_memory_for_spark_application.html>

**>>Tree Reduce v/s ReduceByKey**



In ReduceByKey results are collected by driver from all executors. But if we a huge number of executors and partitions involved , we will have a huge load on driver since driver has to take care of combining the final output . It is observed that , time taken by driver increases linearly as the number of partitions increases.

So Tree reduce instead partially combine the output by passing the outputs from all executors to intermediate executors

We can define the reduce tree depth

**>>Use Broadcast Join whenever required**

Define spark.sql.autobroadcast.threshold=500M (or 1G)

So that by default smaller look up tables will be automatically selected for broadcast join

**>>Use Spark Encoders for better shuffling**

**>>Use Right File Formats with Right Compression**

For read heavy analytical workloads use parquet+snappy compression

For appending new columns as a part of schema evolution, parquet will support that too

But if we needs it to be ACID compliance , then ORC format is preferred.

**>>Handling the Skewed data**

1. If we are doing a join operation on a skewed dataset one of the tricks is to increase the spark.sql.autoBroadcastJoinThreshold value so that smaller tables get broadcasted. This should be done to ensure sufficient driver and executor memory.
2. If there are too many null values in a join or group-by key they would skew the operation. Try to preprocess the null values with some random ids and handle them in the application.
3. In a SQL join operation, the join key is changed to redistribute data in an even manner so that processing for a partition does not take more time. This technique is called salting

**>>QUEUES**

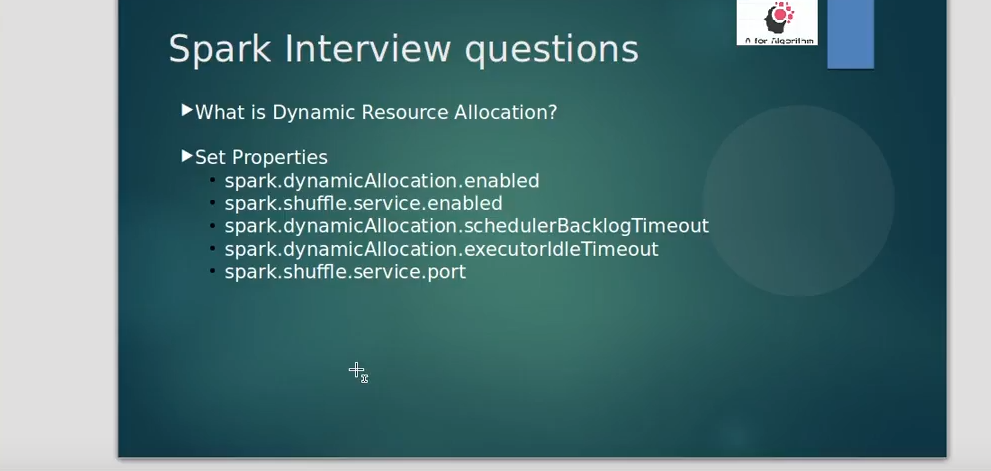
1. In a shared cluster , its better to create separate queues for your own jobs, o that for such a queue you can assign the priority , min/max resource limits etc. So the jobs submitted to this queue will be executed accordingly

**>>Spark Dynamic Allocation**

1. Use spark dynamic allocation with shuffle services enabled
2. If you enable shuffle service , sparks takes out the responsibility of shuffling the data from executors and is given to the shuffle services.

Yarn will kill the idle executors only with respect to the timeout specified in configuration

3.







**>>Dynamic Partition Pruning**

Dynamic partition pruning improves job performance by more accurately selecting the specific partitions within a table that need to be read and processed for a specific query. By reducing the amount of data read and processed, significant time is saved in job execution. With Amazon EMR 5.26.0, this feature is enabled by default. With Amazon EMR 5.24.0 and 5.25.0, you can enable this feature by setting the Spark property spark.sql.dynamicPartitionPruning.enabled from within Spark or when creating clusters.

Dynamic partition pruning allows the Spark engine to dynamically infer at runtime which partitions need to be read and which can be safely eliminated. For example, the following query involves two tables: the store\_sales table that contains all of the total sales for all stores and is partitioned by region, and the store\_regions table that contains a mapping of regions for each country. The tables contain data about stores distributed around the globe, but we are only querying data for North America.