SPARK OPTIMIZATION TECHNIQUES FOR HEAVY-WORKLOADS

Spark Tuning Techniques for Large-Scale Workloads:

- As we have different use case scenarios where we deal with different workloads and applications we need to make sure to tune the spark jobs according to the requirements rather than having a generic approach.
- **Spark Properties** controls most application parameters and can be set using a SparkConf object
- Below we will be discussing in specific to large pipelines the following topics

1. Scaling spark executor
2. Scaling spark executor memory
3. Scaling external shuffle service

1. Scaling Spark Executor:

1.1 This feature is used to add/remove executors dynamically to match with the work loads.

1.2 It can scale the executors based on the work load

Properties to set:

--> To enable dynamic allocation

-->To remove the unused executors after the idle time

-->To set min and max no. of executors

```
spark.dynamicAllocation.enable = true
spark.dynamicAllocation.executorIdleTimeout = 2m
spark.dynamicAllocation.minExecutors = 1
spark.dynamicAllocation.maxExecutors = 2000
```

Better Fetch Failure handling

The number of consecutive stage attempts allowed before a stage is aborted (by default is 4).

`spark.stage.maxConsecutiveAttempts = 10`

Tune RPC Server threads:
-->running threads in a program enables you to run processes in parallel to complete the task more efficiently. *Apache Spark*processes large amounts of data efficiently. One way it does this is by threading the processes.
-->**Threading** enables Spark to systematically utilize available resources to get better performance

Property Name	Default	Meaning
`spark.{driver or executor}.rpc.io.serverThreads`	Fall back on `spark.rpc.io.serverThreads`	Number of threads used in the server thread pool.
`spark.{driver or executor}.rpc.io.clientThreads`	Fall back on `spark.rpc.io.clientThreads`	Number of threads used in the client thread pool.
`spark.{driver or executor}.rpc.netty.dispatcher.numThreads`	Fall back on `spark.rpc.netty.dispatcher.numThreads`	Number of threads used in RPC message dispatcher thread pool.

2. Spark Executor Memory:

2.1 The executor memory is divided into different layers which can be tuned as well to improvise on the performance.

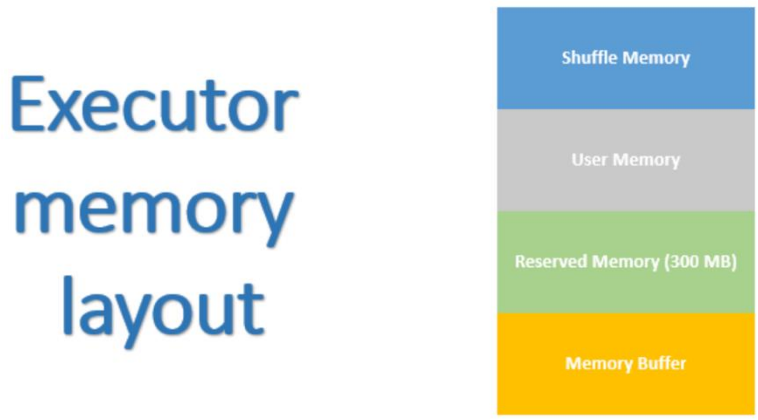

Fig. 1 Executor memory layout

2.2 Shuffle Memory:

-->It is the buffer memory space reserved to store the shuffled data while processing

-->The lower it is the more frequent spills and block eviction happens

-->It is mainly intended to set aside memory for metadata, shuffled data

```
spark.memory.fraction * (spark.executor.memory - 300 MB)
```

2.3 User Memory:

-->It is used to store the user defined data structures, variables

```
(1 - spark.memory.fraction) * (spark.executor.memory - 300 MB)
```

2.4 Reserved Memory:

-->This memory is reserved by the system and doesn't included in spark memory size.

-->Its default size is 300MB

2.5 Memory Buffer:

-->The amount of off heap memory to be allocated per executor

-->It includes JVM over heads etc

2.6 Enable off-heap memory:

```
#Shuffle Memory

spark.memory.offHeap.enable = true
spark.memory.ofHeap.size = 3g

#User Memory

spark.executor.memory = 3g

#Memory Buffer

spark.yarn.executor.memoryOverhead = 0.1 * (spark.executor.memory +
spark.memory.offHeap.size)
```

Garbage Collection Tuning:

-->Garbage collection can be a problem if we have huge amount of data where tracing the old and unused objects can be tricky
-->If the Rdd/data frames created once just to be read but never used again they can be removed entirely.
-->GC is a process which consumes more time if proper tuning is not done in this area

```
spark.executor.extraJavaOptions = -XX:ParallelGCThreads=4 -
XX:+UseParallelGC
```

Tune Shuffle File Buffer:

-->Disk access is slower when compared to in-memory data access as it involves serialization process that takes up time and resources
-->Due to this we tend to reduce disk I/O cost by introducing shuffle read/write file buffer in the memory

```
#Size of the in-memory buffer for each shuffle file output stream.
#These buffers reduce the number of disk seeks and system calls made
#in creating intermediate shuffle files. [Shuffle behavior]
spark.shuffle.file.buffer = 1 MB
```

Tune Compression block size:
-->We can change the default compressed block size especially for large data sets.
-->These data blocks can be compressed through either storage or speed based like Lzo, snappy, gzip

```
#Block size used in LZ4 compression, in the case when LZ4 #compression
codec is used. Lowering this block size will also lower #shuffle
memory usage when LZ4 is used. [Compression and Serialization]
spark.io.compression.lz4.blockSize = 512KB

#Note that tha default compression code is LZ4 you could change #using
spark.io.compression.codec
```

3. Scaling External Shuffle Service

3.1 Cache index files on shuffle server
-->For each shuffle fetch chances are that same index file can be read multiple times
-->To make the process more efficient we can use LRU(least recently used) to cache the indexes of the files
-->So that repetitive reads of the same index files wont happen

```
#Cache entries limited to the specified memory footprint.
spark.shuffle.service.index.cache.size = 2048
```

3.2 Configurable shuffle registration timeout and retry
-->It helps to set the idle time after which registration gets timed out
-->This is mainly useful for a large node cluster where there are chances of a node failure

```
spark.shuffle.registration.timeout = 2m
spark.shuffle.registration.maxAttempst = 5
```

Sources:
https://towardsdatascience.com/how-does-facebook-tune-apache-spark-for-large-scale-workloads-3238ddda0830
https://www.educative.io/edpresso/how-to-perform-thread-configuration-in-spark-3