Spark Common Errors and their Solutions

1. Issue: Your application runs out of heap space on the executors.

Error messages:
– *java.lang.OutOfMemoryError: Java heap space* on the executors nodes
– *java.lang.OutOfMemoryError: GC overhead limit exceeded*
– *org.apache.spark.shuffle.FetchFailedException*

Possible Causes and Solutions

- An executor might have to deal with partitions requiring more memory than what is assigned. Consider increasing the *–executor memory* or the *executor memory overhead* to a suitable value for your application.
- Shuffles are expensive operations since they involve disk I/O, data serialization, and network I/O. Avoid shuffles whenever possible. Shuffle operations can result in large partitions where the data is not evenly distributed across them. Also, the shuffle block size is limited to two gigabytes.
- Very large partitions may end up with writing blocks greater than two gigabytes. In cases like these, consider increasing the number of partitions in your job. This ensures that there is lesser data per partition.
- If the data that you are dealing with has skewed partitions i.e. certain partitions having huge amount of data compared to the rest, then append some hash value to the end of your key. This will lead to better distribution of your data and you can have an additional aggregate step to remove the appended hash and get back all values for that key.
- While working with RDDs, avoid using *groupByKeys*. *GroupBy* keys tend to keep all values for a given key in memory. Keys having a very large value list that cannot be kept in memory will result in OOMs as they aren’t spilled to disk. One solution is to replace *groupByKeys* with *reduceByKeys* that does a map side combine and decreases the amount of data that is passed to the reducers.
- The *groupByKey* functionality works better with dataframes and datasets because of the query optimizer where it may switch to using reduceByKey instead or applies a filter before grouping based on what your application does with grouped data.
- While joining two datasets where one of them is considerably smaller in size, consider broadcasting the smaller dataset. Set *spark.sql.autoBroadcastJoinThreshold* to a value equal to or greater than the size of the smaller dataset or you could forcefully broadcast the right dataset by *left.join(broadcast(right), columns)*.
- Configure your application to run with more cores per executors. While this still maintains parallelism, it also allows for fewer shuffles than when the application runs with large number of executors and fewer cores.

- A lot of time spent on GC is an indication that data didn't fit into the heap space. One of the first and foremost things to do is to ensure there aren't any memory leaks in your code (Check for large number of temporary objects created by doing a heap dump).
- Allocate sufficient storage memory (increase `*spark.memory.storageFraction*`) for caching data and only cache them if they are being re-used elsewhere in the job.

2. Issue: Your application runs out of heap space on the driver node.

Error messages:

– *java.lang.OutOfMemoryError: Java heap space on the driver node*

Possible Causes and Solutions

- First and foremost, consider increasing *–driver-memory* or *driver memory overhead* if set to a very low value.
- When you perform actions on the data that ends up collecting the result on the driver end, for example, applying a collect() on the data is an unsafe operation that can lead to an OOM if the collected size is larger than what the driver can house in its memory. If the goal is to save the data, then instead of bringing back all the data to the driver, it is preferable to let the executors parallely write down each of the resultant partitions into separate files.
- When your application has a really large number of tasks (partitions), an OOM on the driver end can occur easily. Every task sends a mapStatus object back to the driver. When there is a shuffle (*re-partition, coalesce, reduceByKey, groupByKey, foldByKey, combineByKey, sortByKey, cogroup, join*) operation involved, data gets re-distributed across executors and leads to the generation of map and reduce tasks.
- Intermediate files are written to disk in the shuffle stage to avoid re-computation. The map status contains location information for the tasks which is sent back to the driver. A large number of tasks would lead to the driver receiving a lot of data and also having to deal with multiple Map output status requests in the reduce phase.

3. Issue: Cluster runs out of disk space.

Error messages:

– *java.io.IOException: No space left on device*

Possible Causes and Solutions

- Check the executors for logs like this `*UnsafeExternalSorter: Thread 75 spilling sort data of 141.0 MB to disk (90 times so far)*`. This is an indication that the storage data is continuously evicted to disk. Exceptions like this occur when data becomes larger than what is configured to be stored on the node

- Ensure that the `*spark.memory.fraction*` isn't too low. The default being 0.6 of the heap space, setting it to a higher value will give more memory for both execution and storage data and will cause lesser spills.
- Shuffles involve writing data to disk at the end of the shuffle stage. As a general practice avoid spilling to disk unless the cost of computation is much higher than reading from the disk. One such example to avoid shuffles is to broadcast the smaller table while joining or even partition both the datasets with the same hash partitioner so that keys with the same hash from both tables reside in the same partition.
- If you are running the job with minimal number of nodes, consider adding more nodes to increase the DFS for the cluster.
- By default, the spilled data and intermediate files are written to */tmp*. You can mount more disks and specify them in *spark.local.dirs* to provide more disk space for the cluster.

4. Issue: Application takes too long to complete or is indefinitely stuck and does not show progress.

Possible Causes and Solutions

- Long running tasks often referred to as *stragglers* are mostly a result of skewed data. This means there are a few partitions that have more data than the rest causing tasks that deal with them to run for a longer time. The right solution here is to re-partition your data to ensure even distribution among tasks.
- Tasks are scheduled to the executors based on the lowest locality level of data. However, a task that is set to run on a particular executor will be handed over to another executor if it is free. This brings in the possibility of the re-assigned task to read data from over the network and might change the locality level of the data for the task.

Tasks dealing with higher locality levels will face delays due to the increase in network I/O. If you encounter this scenario where most of your stragglers are because of the higher locality levels, consider increasing *spark.locality.wait* to let the task wait a little longer before it gets reassigned to a free executor.

Apart from the above errors that revolve under what operations you perform on your data and how efficiently you use the memory resources available, one might also encounter network issues where an executor's heartbeat times out. In such cases, consider increasing
your *spark.network.timeout* and *spark.executor.heartbeatInterval*.

5. Exceeding memory limits

Problem:
1) Container killed by YARN for exceeding memory limits. 2.4 GB of 2.4 GB physical memory used. Consider boosting spark.yarn.executor.memoryOverhead.

2) Application application_1472670787492_0003 failed 2 times due to AM Container for appattempt_1472670787492_0003_000002 exited with exitCode: -104
For more detailed output, check application tracking page:http://ip_address:8088/cluster/app/application_1472670787492_0003Then, click on links to logs of each attempt.
Diagnostics: Container [pid=7250,containerID=container_1472670787492_0003_02_000001] is running beyond physical memory limits. Current usage: 6.7 GB of 6.6 GB physical memory used; 9.6 GB of 33.1 GB virtual memory used. Killing container.

Solution: Increase spark.yarn.executor.memoryOverhead.

6. Executor heartbeat timed out after 177005 ms

Problem: Some tasks fails (executor goes to dead state) because of heartbeat timeout issue.

Solution: Increase the value of ***spark.network.timeout*** to avoid the issue.

7. Executor RECEIVED SIGNAL 15: SIGTERM

Problem: Task failed due to "executor.CoarseGrainedExecutorBackend: RECEIVED SIGNAL 15: SIGTERM" error.

Solution: Increase executor memory ***spark.executor.memory***.

8. Futures timed out after [300 seconds]

Problem: java.util.concurrent.TimeoutException: Futures timed out after [300 seconds]

Solution: To resolve this exception, we found the ***spark.sql.broadcastTimeout*** parameter that controlled the internal timeout value. We then set it to the longest query time when the query does work to completion with a single user, for example:

```
spark.sql.broadcastTimeout          1200
```