Spark Troubleshooting

Date published: 2024-09-23
Date modified:

CLOUDERA

Legal Notice

Contents

Spark jobs failing with memory issues

This article describes troubleshooting tips for when Spark application fails with memory issues such as java.lang.Ou tOfMemoryError : *.

Condition

The general symptom for this issue is that Spark applications fail with one of the errors listed below.

This can either be a new application or an application that has been previously working, but is now failing after an increase in the data size being processed or after a configuration change. The specific symptoms for this issue, which are contained in the log output, are exposed in the Spark driver, Spark executor, YARN Application Master, and NodeManager logs.

Symptoms Error messages include:

```
java.lang.OutOfMemoryError: Java heap space
```

```
java.lang.OutOfMemoryError: GC overhead limit exceeded
```

```
java.lang.OutOfMemoryError: unable to create new native thread
```

```
java.lang.OutOfMemoryError: Not enough memory to build and broadcast the tab
le to all worker nodes. As a workaround, you can either disable broadcast by
 setting
            spark.sql.autoBroadcastJoinThreshold to -1 or increase the spark
 driver memory by setting spark.driver.memory to a higher value
```

```
java.lang.OutOfMemoryError: Unable to acquire * bytes of memory, got 0
```

```
java.lang.OutOfMemoryError: Requested array size exceeds VM limit
```

```
ERROR org.apache.spark.executor.Executor: Managed memory leak detected; size
 = * bytes, TID = *
```

```
containerID=container_*] is running [*** VALUE B***] beyond the 'PHYSICAL' m
emory limit. Current usage: * GB of * GB physical memory used; * GB of * GB
virtual memory used. Killing container.
```

Also, the job may report WARN messages such as

```
20/11/24 20:23:20 WARN memory.MemoryStore: Not enough space to cache rdd_343
3_344 in memory! (computed 276.3 MB so far)
```

Cause

When Spark runs on YARN, containers memory is allocated as per the Spark configurations set.

You can configure it using the spark.executor|driver property (the default value is the driver|executor with a minimum of 1024 MB).

When the configured memory is too small to accommodate the needs of a running application, the application containers exceed the memory limit set by YARN and are killed by the framework. When enough containers are killed, the Spark application fails.

Understanding Spark configurations related to memory

This section provides an overview of Spark configuration order of precedence rules and instructions for resolving issues with YARN killing containers for exceeding memory limits.

1. **Understanding the Spark Configuration Order of Precedence**

   When configuring Spark application, you need to understand the order of precedence for configuration values. Configuration values explicitly set in code(SparkConf) take the highest precedence, then flags passed to the spark-submit command, then values in the Spark defaults configuration file.

   The exact order of precedence is:

   - Spark configuration file: Parameters are passed to Spark Conf.
   - Command-line arguments are passed to spark-submit, spark-shell, or pyspark.
   - Cloudera Manager UI properties set in the spark-defaults.conf configuration file.

   a. Available Spark configuration properties

      - spark.executor.memory
      - spark.driver.memory
      - spark.yarn.am.memory
      - spark.yarn.executor.memoryOverhead
      - spark.yarn.driver.memoryOverhead

   b. Using Command-line arguments

      You can set the above configuration properties when launching Spark applications using the spark-submit command:

      ```
      ./bin/spark-submit --conf spark.executor.memory=[*** VALUE ***] --conf
      spark.driver.memory=[*** VALUE ***]
      ```

2. **Identify what type of containers are being killed**

   Review the logs and identify which container is being killed, that is: the driver, executor, or application master container.

   If the container being killed is the executor container, Use the container ID in the NodeManager logs in conjunction with the application logs, to identify the error if not logged in Executor container logs.

3. **Verifying the actual memory values used**

   Before increasing the Driver or executor memory, access the Spark application properties through the Spark UI:

   - For each application, run a job and open the **Spark UI**.
   - Click on the Environment tab.
   - Scroll to the Spark Properties section and view the **Memory** values.

   > **Note:** Refer to *Tuning Spark Applications* for more information about fine tuning Spark.

Error 1 : java.lang.OutOfMemoryError: Java heap space

Driver side causes

Possible causes when the driver can go OOM are:

- Running a Collect/Show/ operation on a large file
- Memory leak

Executor side causes

Possible causes when the executor can go OOM are:

- Possible causes data skewness

- Less partitions or the data size is larger than the memory allocated
- Memory leak - where the memory usage increases sharply leading to OOM, even while less or no data is being processed.

Possible solutions

- Increase the memory for the driver or executor depending on the error message. You need to try this until the job succeeds and we recommend using spark-submit command line arguments.
- Check the Spark event log , if the last operation that caused the failure is due to Collect/show , then check with the customer on the necessity to perform this case. This is generally done over testing propose for small data size and is not needed over PROD or larger dataset runs.

Error 2 : java.lang.OutOfMemoryError: GC overhead limit exceeded

Driver side causes

Possible causes when the driver can go OOM are:

- Running a Collect/Show/ operation on a large file
- Memory leak

Executor side causes

- Possible causes data skewness
- Less partitions or the data size is larger than the memory allocated
- Memory leak

Possible solutions

- Increase the memory for the driver or executor depending on the error message. You need to try this until the job succeeds and we recommend using spark-submit command line arguments.
- Check the Spark event log , if the last operation that caused the failure is due to Collect/show , then check with the customer on the necessity to perform this case. This is generally done over testing propose for small data size and is not needed over PROD or larger dataset runs.
- If even after multiple re-tires of increasing the driver or executor memory and job fails with processing small amount of data, then it can be a issues with Memory Leak. Take Heap Dump via steps to take heap dump on Spark application.
- For data skewness issue, please check this article on how to identify and the solutions to resolve the same.

Error 3 : java.lang.OutOfMemoryError: unable to create new native thread

Possible causes when the driver/executor can go OOM are

This indicates that that the application has hit the limit of how many threads it can launch, which is dictated by the OS (the number it will allow your JVM to use). This in turn usually either means that you need to make changes to the code or increase the ulimit (nproc) on all NodeManager hosts.

Possible solutions

In the /etc/security/limits.d/ directory, edit the file *[*** NUMBER ***]*-nproc.conf, and make sure the user who needs higher nproc settings has the appropriate limit configured as follows:

```
userid soft nproc 65536
```

The above is only an example, and should be modified as desired on all NodeManager hosts.

Error 4 : java.lang.OutOfMemoryError: Not enough memory to build and broadcast the table to all worker nodes.

Possible solutions

- Increase the memory for the driver depending on the error message.

- Check the table size being broadcasted. It should be noted that we should always broadcast a smaller table. Users can implicitly specify Broadcast Hint (see *Join Strategy Hints for SQL Queries* in the Apache Spark Documentation) in the query.
- Disable broadcast by setting spark.sql.autoBroadcastJoinThreshold to **-1**.

Error 5 : java.lang.OutOfMemoryError: Unable to acquire * bytes of memory, got 0

Possible causes when the driver can go OOM are:

- Possible causes data skewness
- Less partitions or the data size is larger than the memory allocated
- Memory leak
- System does not have enough memory to allocate resources

Possible solutions

- Increase Executor memory.
- Increase Executor memory overhead:

  ```
  --conf spark.yarn.executor.memoryOverhead=[*** VALUE ***]
  ```

  The *[*** VALUE ***]* is usually 10% of the value specified in --executor-memory.
- Check system setting on Free memory available.

Error 5 : java.lang.OutOfMemoryError: Requested array size exceeds VM limit

Possible causes when the driver can go OOM are:

Shuffle block size is greater than the 2GB Shuffle Size limit . The issue you're experiencing here has to do with the 2 GB block limit in Spark.

Possible solutions

Consider increasing the following:

- the number of partitions for your RDDs in the code by using either the RDD's coalesce() or repartition() method.
- >Executor memory

Error 5 : ERROR org.apache.spark.executor.Executor: Managed memory leak detected; size = * bytes, TID = *

Managed memory leak error in executors issue occurs when the shuffle read requires more memory than what is available for the task. Since the shuffle-read side doubles its memory request each time, it can easily end up acquiring all of the available memory, even if it does not use it.

Note: The job will report with one of the above java.lang.OutOfMemoryError: * reported in the executor logs along with the above Managed memory leak detected* error.

Possible solutions

- To increase Driver memory overhead:

  ```
  --conf spark.yarn.driver.memoryOverhead=[*** VALUE ***]
  ```

  The *[*** VALUE ***]* is usually 10% of the value specified in --executor-memory.
- To Increase Executor memory overhead:

  ```
  --conf spark.yarn.executor.memoryOverhead=[*** VALUE ***]
  ```

  The *[*** VALUE ***]* is usually 10% of the value specified in --executor-memory.

containerID=container_*] is running *[*** VALUE B***]* beyond the 'PHYSICAL' memory limit. Current usage: * GB of * GB physical memory used; * GB of * GB virtual memory used. Killing container.

Possible causes when the executor can go OOM are:

- Possible causes data skewness
- Less partitions or the data size is larger than the memory allocated
- Memory leak

> **Note:** The job may report with one of the above java.lang.OutOfMemoryError: * reported in the executor logs along with the above beyond the 'PHYSICAL' memory limit error.

Possible solutions

Consider increasing Executor memory:

- To Increase Executor memory overhead:

```
--conf spark.yarn.executor.memoryOverhead=[*** VALUE ***]
```

The *[*** VALUE ***]* is usually 10% of the value specified in --executor-memory.

WARN Message: WARN memory.MemoryStore: Not enough space to cache rdd*

Data spilling to disk and no memory to Cache RDD. Indicates memory being easily filled and it can cause performance issues and also application failure with OOM errors as above.

1. To avoid frequent spilling you would need to increase the heap size of Spark executor.
2. Increase the partition count to have more distribution of data. Review *Tuning Partitions in Spark*.
3. Memory leak

Possible solutions

- Increase the executor or driver memory Overhead (spark.executor.memoryOverhead and -spark.driver.memory Overhead).

  The memory overhead defaults to 10% of the memory specified (with a minimum of 384mb). Thus increasing the executor or driver memory can also increase the memory overhead if sufficiently large values are used.

- Decrease the number of executor cores (--executor-cores)

  By decreasing the number of parallel tasks being run in executors, each executor requires less memory overhead.

Related Information

Tuning Partitions in Spark

Join Strategy Hints for SQL Queries | Apache Spark Documentation

Tuning Spark Applications