# Additional Metrics for Memory Tuning Design Document

## Overview and Motivation

This document describes a proposal for adding new executor memory metrics, in order to give better visibility into executor memory usage.

At LinkedIn, we have multiple clusters, running thousands of Spark applications, and these numbers are growing rapidly. We need to ensure that these Spark applications are well tuned -- cluster resources, including memory, should be used efficiently so that the cluster can support running more applications concurrently, and applications should run quickly and reliably.

Currently there is limited visibility into how much memory executors are using, and users are guessing numbers for executor and driver memory sizing. These estimates are often much larger than needed, leading to memory wastage. Examining the metrics for one cluster for a month, the average percentage of used executor memory (max JVM used memory across executors /  spark.executor.memory) is 35%, leading to an average of 591GB unused memory per application (number of executors * (spark.executor.memory - max JVM used memory)). Spark has multiple memory regions (user memory, execution memory, storage memory, and overhead memory), and to understand how memory is being used and fine-tune allocation between regions, it would be useful to have information about how much memory is being used for the different regions.

Identifying memory related errors, such as tasks failing due to OutOfMemoryError or from executors being killed by YARN due to exceeding memory limits, also helps users determine if they need to allocate more memory.

## Additional Executor Memory Metrics

To improve visibility into memory usage for the driver and executors and different memory regions, the following additional memory metrics can be be tracked for each executor and driver:

- Heap and non-heap JVM used memory: the JVM heap size, and non-heap memory used by the JVM, for the executor/driver.
  - ManagementFactory.getMemoryMxBean.getHeapMemoryUsage().getUsed()
  - ManagementFactory.getMemoryMxBean.getNonHeapMemoryUsage().getUsed()
- On heap and off heap execution memory: memory used for computation in shuffles, joins, sorts and aggregations.
  - onHeapExecutionMemoryPool.memoryUsed
  - offHeapExecutionMemoryPool.memoryUsed

- On heap and off heap storage memory: memory used caching and propagating internal data across the cluster.
    - onHeapStorageMemoryPool.memoryUsed
    - offHeapStorageMemoryPool.memoryUsed
- Direct memory: amount of memory used by direct byte buffers
- Mapped memory: amount of memory used for memory mapped files

The peak values for each memory metric can be tracked for each executor over the lifetime of the Spark application, and also per stage. This information can be shown in the Spark UI and the REST APIs. Information for peak JVM used memory can help with determining appropriate values for spark.executor.memory and spark.driver.memory, and information about execution and storage memory can help with determining appropriate values for spark.memory.fraction and spark.memory.storageFraction. Stage memory information can help identify which stages are most memory intensive, and users can look into the relevant code to determine if it can be optimized.

## Heartbeat Changes

Executors send a heartbeat to the driver, by default every 10 seconds. A snapshot of the JVM used memory, executor memory and storage memory sizes can be taken and sent as part of the heartbeat. This may miss gathering information for very short stages or applications, but would provide useful metrics for longer running stages and applications, which are using more cluster resources.

The driver can also periodically (every 10 seconds) gather its own memory metrics.

At the driver, the executor memory metrics in the heartbeats can be processed as SparkListenerExecutorMetricsUpdate events by interested SparkListeners. The AppStatusListener can track the peak values of the different memory metrics for each executor, and for the executor summaries for each stage.

## Spark History Log Changes

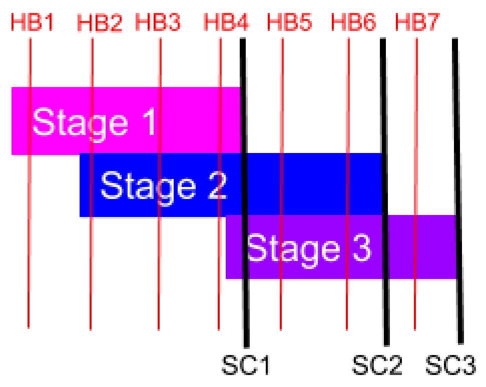
### Proposed Design

Currently, SparkListenerExecutorMetricsUpdate events are not added to the Spark history log by EventLoggingListener since these are very frequent and would generate too much logging. With the new executor level metrics, when a stage has completed, SparkListenerExecutorMetricsUpdate events can be added to the Spark history log for each executor that has sent metrics during the stage, with information about peak memory values for the executors during the stage.

EventLoggingListener will keep track of the current peak values for JVM used memory, execution memory and storage memory for each stage and executor. If it receives a SparkListenerExecutorMetricsUpdate event with a higher value for any of the metrics, then it will update the peak value for the metric(s).  When it receives a SparkListenerStageCompleted event, then it will iterate over the executors, and log a SparkListenerExecutorMetricsUpdate (executor metrics only, no task metrics) for each executor with metric values.

With this design, the additional logging is kept to a minimum, with an additional (numExecutors * numStages) events. There is enough information to determine the peak memory values for executors, and for each executor for each stage.

Some information could be lost for concurrently running stages. For example, if stage 1 has peak X, then stage 2 starts, and the peak value for the two stages is Y, with X<Y, we would record that the peak for stage 1 is Y, and lose the information that stage 1 running by itself had peak Y earlier.



**HB#**: Heartbeats, with current metric values

**SC#**: Stage # Completed, log ExecutorMetricsUpdate events, with peak metric values set to

- SC1: max(HB1, HB2, HB3, HB4)
- SC2: max(HB2, HB3, HB4, HB5, HB6)
- SC3: max(HB5, HB6, HB7)

## Alternative Design

Another option is to log a SparkListenerExecutorMetricsUpdate event for new peak value for any metric, for each stage. The EventLoggingListener would keep track of the current peak values for each metric for each executor. If it receives an event with a higher value for one of the metrics, then it would update the peak value for the metric, and write the event to the Spark history log, omitting the task metrics to minimize the size. If a new stage starts, then the tracking variables would be reset, so that it would start logging new values for the stage.

Analysis of some Spark applications on our clusters shows extra logging overhead averaging about 14% per application, and 4% overall (sum of logging for ExecutorMetricsUpdate / sum of Spark history logs), when 4 memory metrics are used. The overhead for larger Spark history logs is mostly pretty small (~1%), but increases significantly for smaller ones. With more metrics, the overhead would be higher, both in the size of each logged event, and the number of potential peaks, since a new peak for any metric would be logged. Due to the extra logging overhead, we are going with the first proposed design.

## Web UI Changes

For the Executors page, the summary can show the maximum peak JVM used  heap/non-heap memory, on-heap/off-heap execution memory and on-heap/off-heap storage memory across all executors. The table for individual executors can include columns for these memory metrics.

For the individual Stage page, a new summary metrics for executors table can be added, with quantile values for peak JVM used heap/non-heap memory, on-heap/off-heap execution memory and on-heap/off-heap storage memory for executors for the stage. The aggregated metrics by executor table can include  peak JVM used heap/non-heap memory, on-heap/off-heap execution memory and on-heap/off-heap storage memory for each executor for the stage.

## REST API Changes

LinkedIn uses Dr. Elephant to provide automated monitoring and tuning advice for Hadoop MapReduce jobs, and this functionality is being extended to Spark applications. Dr. Elephant will use the REST API to gather Spark metrics.

The executors REST API can include the peak JVM used heap/non-heap memory, on-heap/off-heap execution memory and on-heap/off-heap storage memory.

The stages REST API can include the peak JVM used heap/non-heap memory, on-heap/off-heap execution memory and on-heap/off-heap storage memory across executors for the stage. The executor summary list would also include these new metrics for the stage for each executor.

A new executor summary option can be added for the individual stage REST API, to return the quantile values for peak executor memory usage (JVM used, execution, storage, etc.) and other executor metrics across executors for a stage. It would give some idea of differences in memory usage for executors (for example if most executors are using 2G, but there are a couple using 10G). There is already information about skew at the task level with the taskSummary REST API (input, output and shuffle read/write), but an executor summary would show the effects of skew at the executor level.

```
http://<server-url>:18080/api/v1/applications/<application id>/<application
attempt>/stages/<stage id>/<stage attempt>/executorSummary
```

To simplify gathering information about any failures, a new failed tasks REST API can also be added, to return the stages with failed tasks, and detailed information for those failed tasks:

```
http://<server-url>:18080/api/v1/applications/<application id>/<application
attempt>/stages/failedTasks
```