Understanding Memory Management in Spark For Fun And Profit

Shivnath Babu (Duke University, Unravel Data Systems)
Mayuresh Kunjir (Duke University)

SPARK SUMMIT 2016
DATA SCIENCE AND ENGINEERING AT SCALE
JUNE 6-8, 2016 SAN FRANCISCO

We are

- Shivnath Babu
  - Associate Professor @ Duke University
  - CTO, Unravel Data Systems
- Mayuresh Kunjir
  - PhD Student @ Duke University

A Day in the Life of a Spark Application Developer

Container [pid=28352,containerID=container_1464692140815_0006_01_000004] is running beyond physical memory limits. Current usage: 5 GB of 5 GB physical memory used; 6.8 GB of 10.5 GB virtual memory used. **Killing container.**

spark-submit --class SortByKey --num-executors 10 --executor-memory 4G --executor-cores 16

Searches on StackOverflow

Container is running beyond memory limits

Using all resources in Apache Spark with Yarn

Spark - Container is running beyond physical memory limits

Spark streaming on yarn - Container running beyond physical memory limits

I am getting the executor running beyond memory limits when running big join in spark

How to avoid Spark executor from getting lost and yarn container killing it due to memory limit?

Fix #1: Turn off Yarn's Memory Policing

yarn.nodemanager.pmem-check-enabled=false

Application Succeeds!

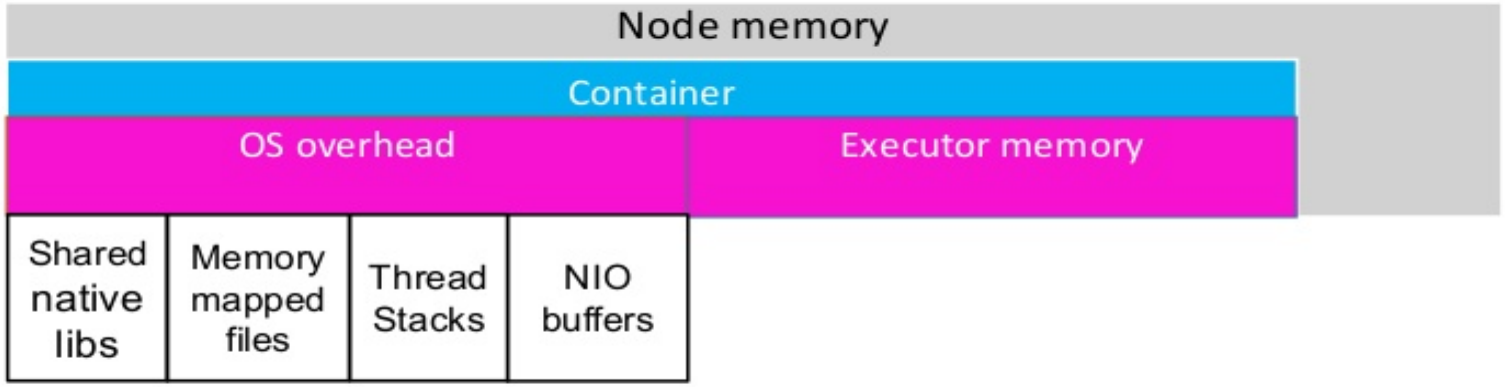
But, wait a minute

This fix is not multi-tenant friendly!
-- Ops will not be happy

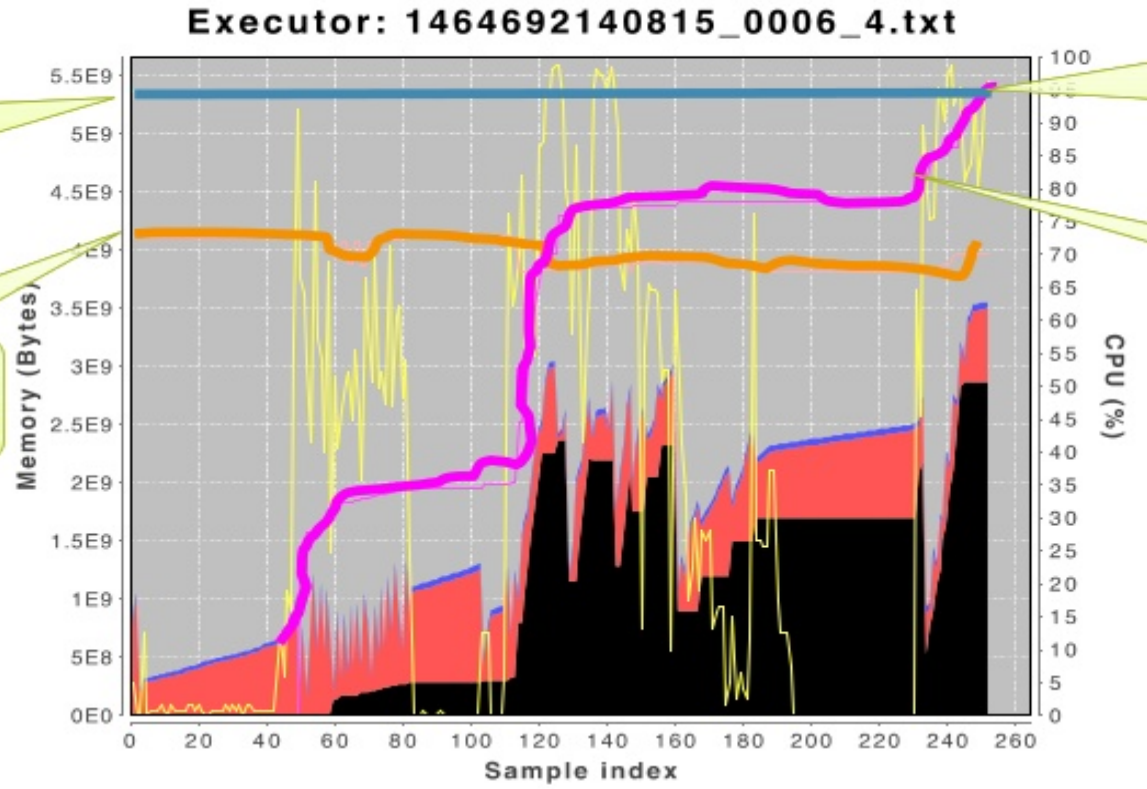
Fix #2: Use a Hint from Spark

WARN yarn.YarnAllocator: Container killed by YARN for exceeding memory limits. 5 GB of 5 GB physical memory used. ***Consider boosting spark.yarn.executor.memoryOverhead***

What is this Memory Overhead?



A Peek at the Memory Usage Timeline



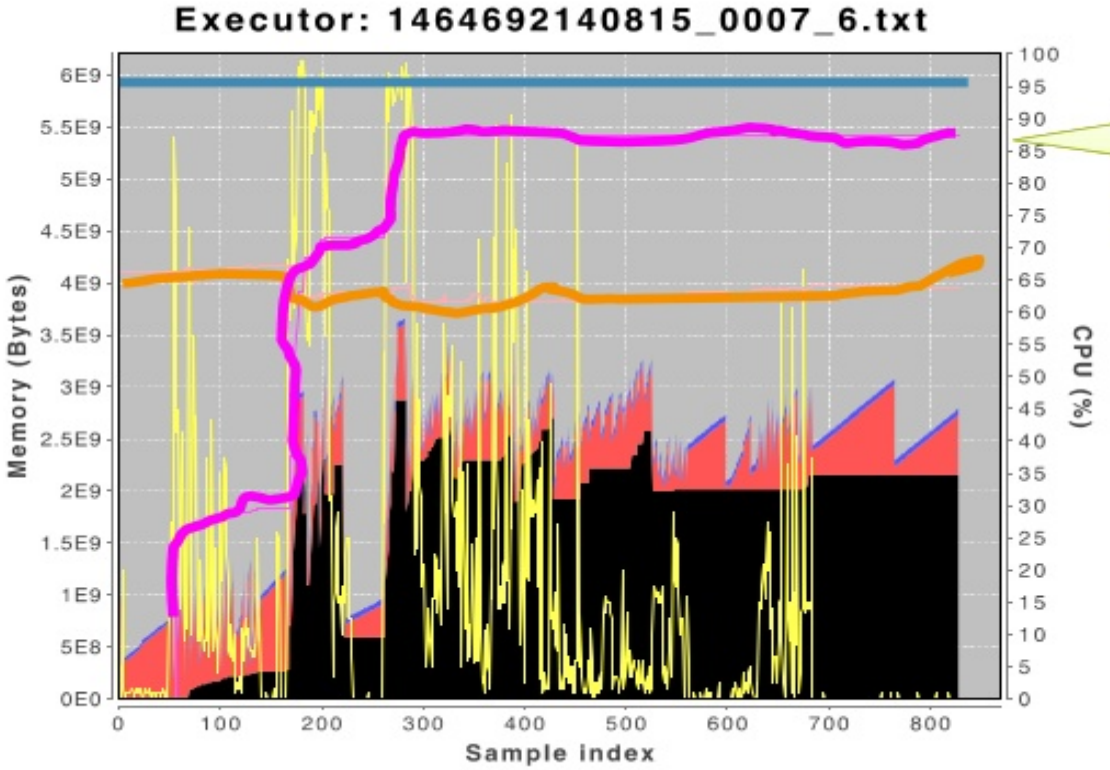
Container memory

Executor JVM max heap

Container is killed by Yarn

Physical memory used by Container as seen by OS

After Applying Fix #2



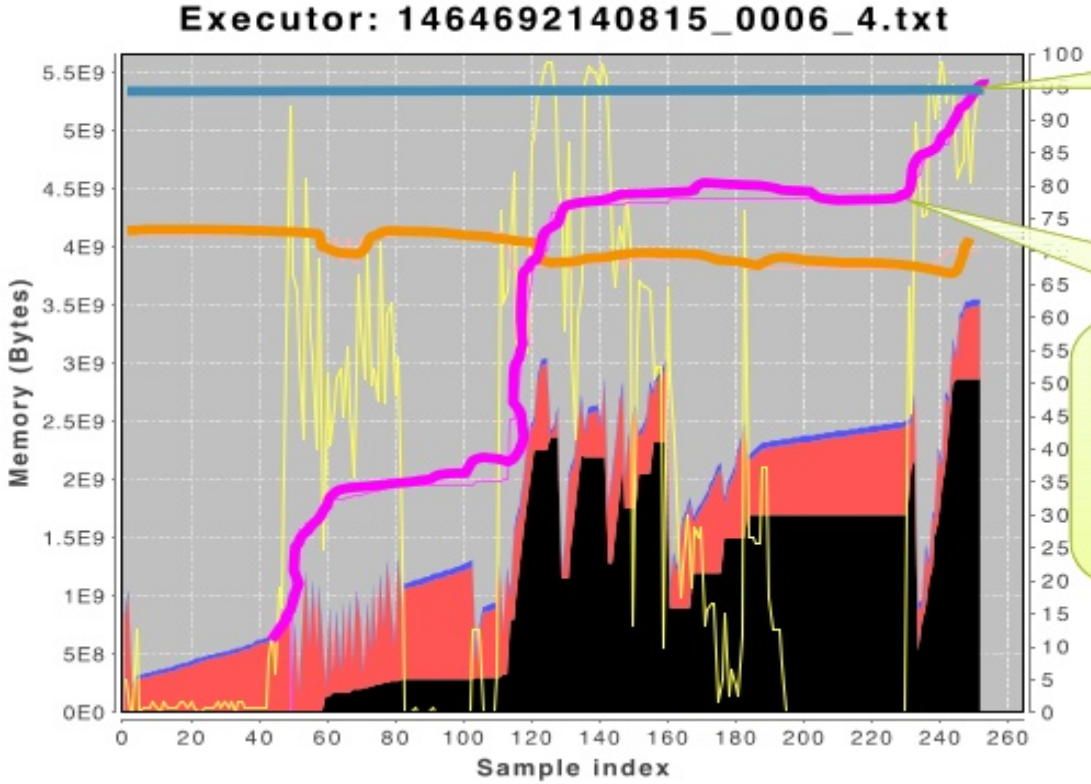
Leaving more room for overheads

spark-submit --class SortByKey --num-executors 10 --executor-memory 4G --executor-cores 16 **--conf spark.yarn.executor.memoryOverhead=1536m**

But, what did we do here?

We traded off memory efficiency for reliability

What was the Root Cause?



Container is killed by Yarn

Each task is fetching shuffle files over NIO channel. The buffers required are allocated from OS overheads

Fix #3: Reduce Executor Cores

Less Concurrent Tasks → Less Overhead Space

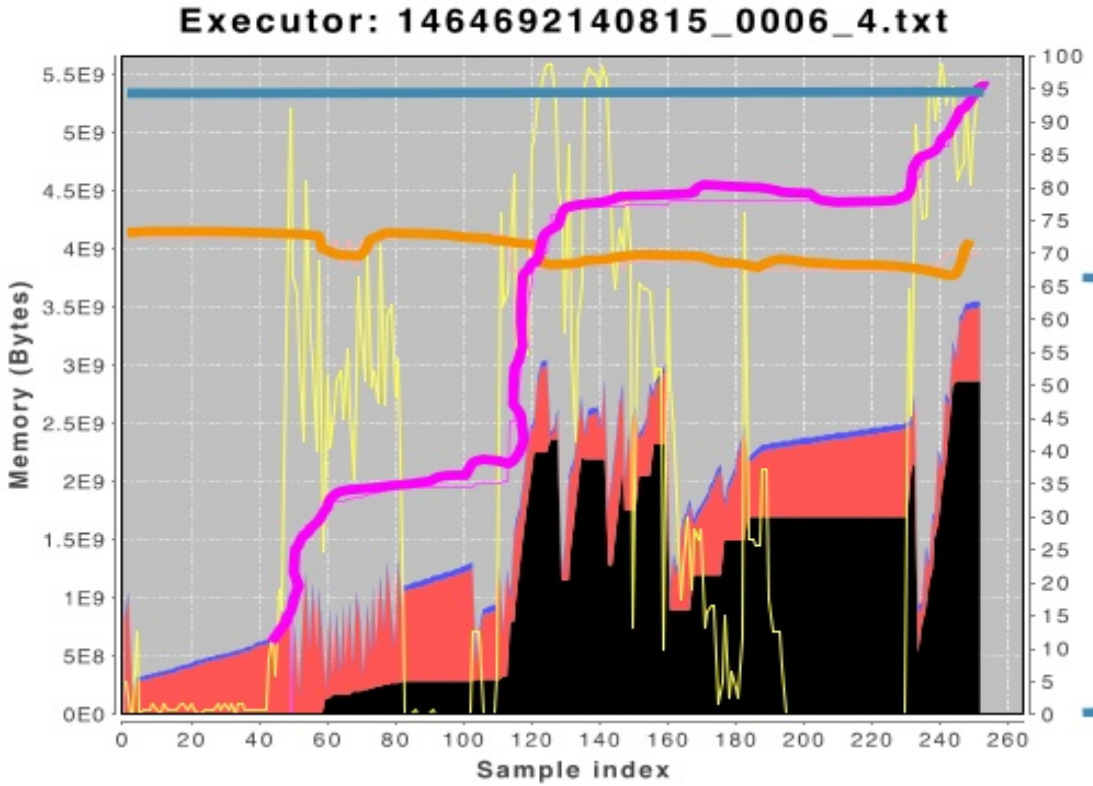
Application Succeeds!

spark-submit --class SortByKey --num-executors 10 --executor-memory 4G **--executor-cores 8**

But, what did we really do here?

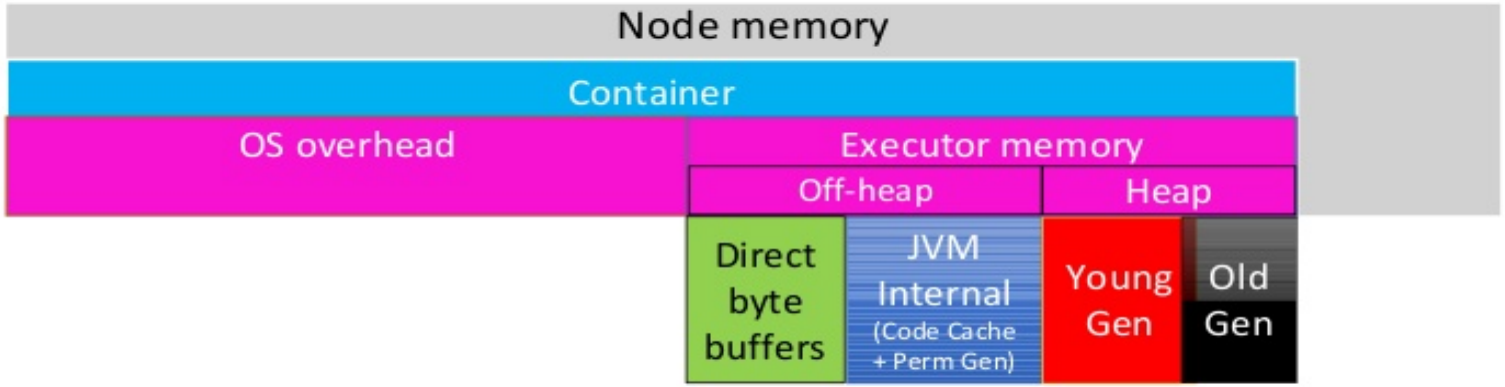
We traded off performance and CPU efficiency for reliability

Let's Dig Deeper

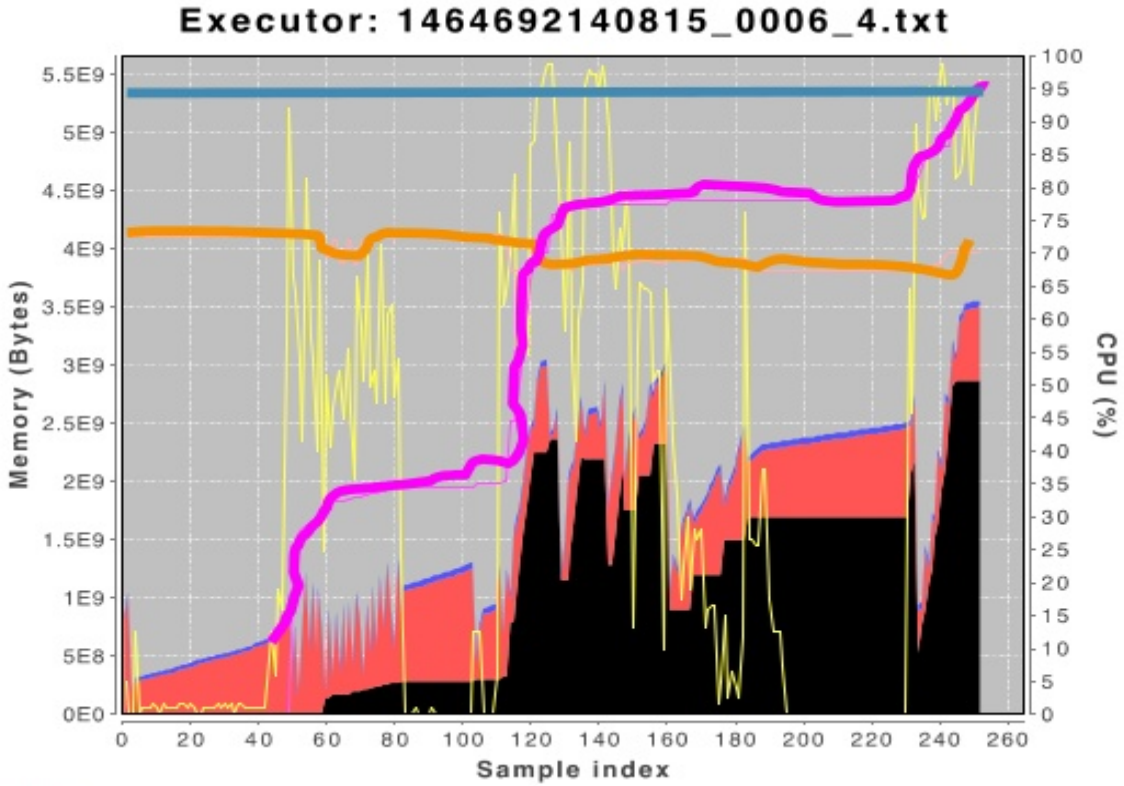


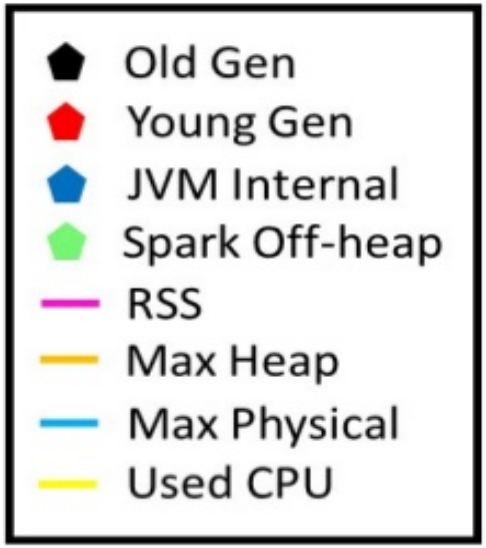
Why is so much memory consumed in Executor heap?

JVM's View of Executor Memory

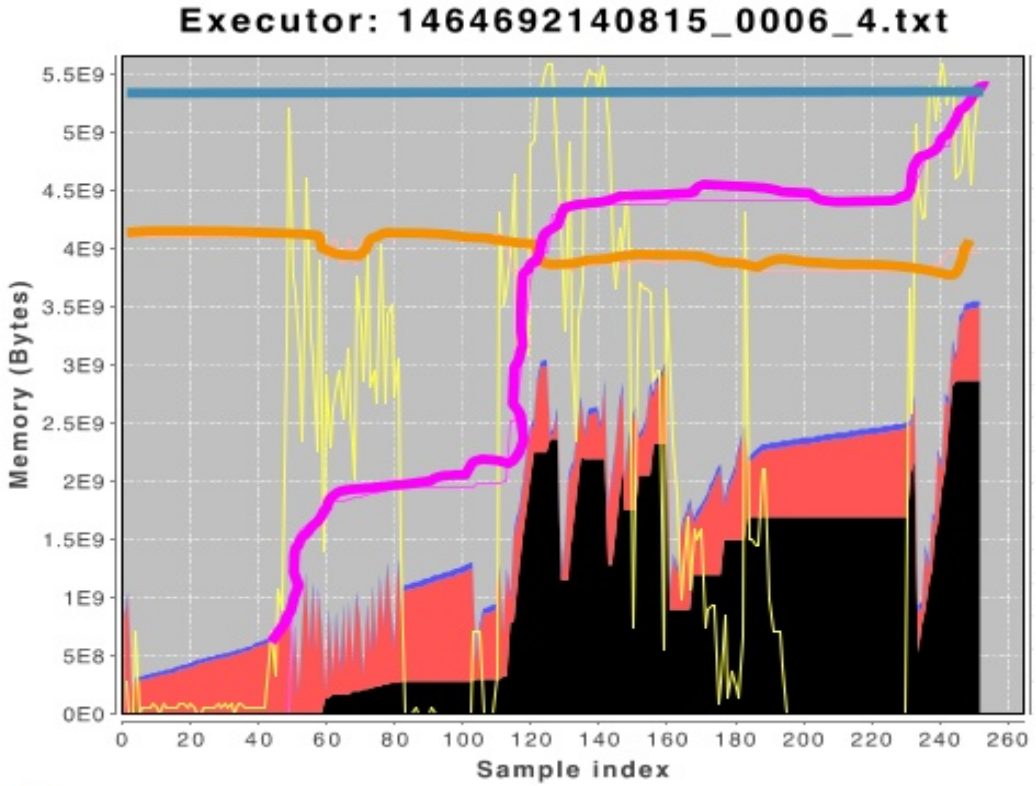


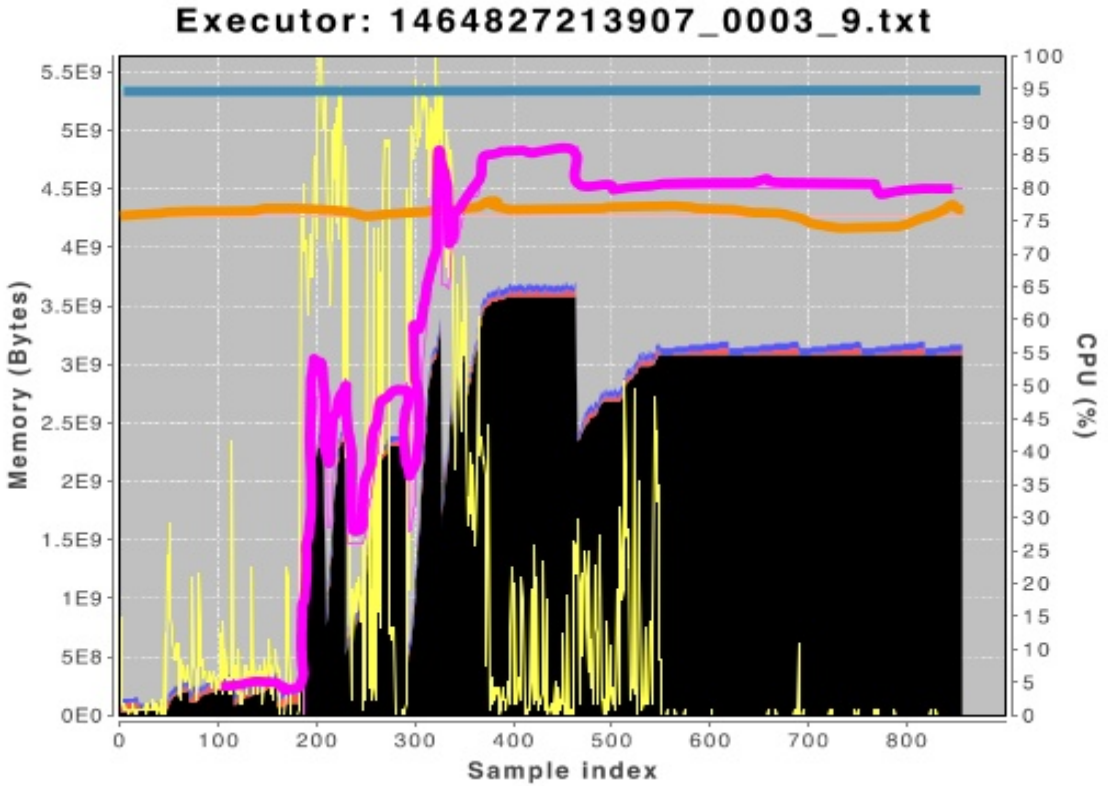
JVM's View of Executor Memory





Fix #4: Frequent GC for Smaller Heap





spark-submit --class SortByKey --num-executors 10 --executor-memory 4G --executor-cores 16

--conf "spark.executor.extraJavaOptions=-XX:OldSize=100m -XX:MaxNewSize=100m"

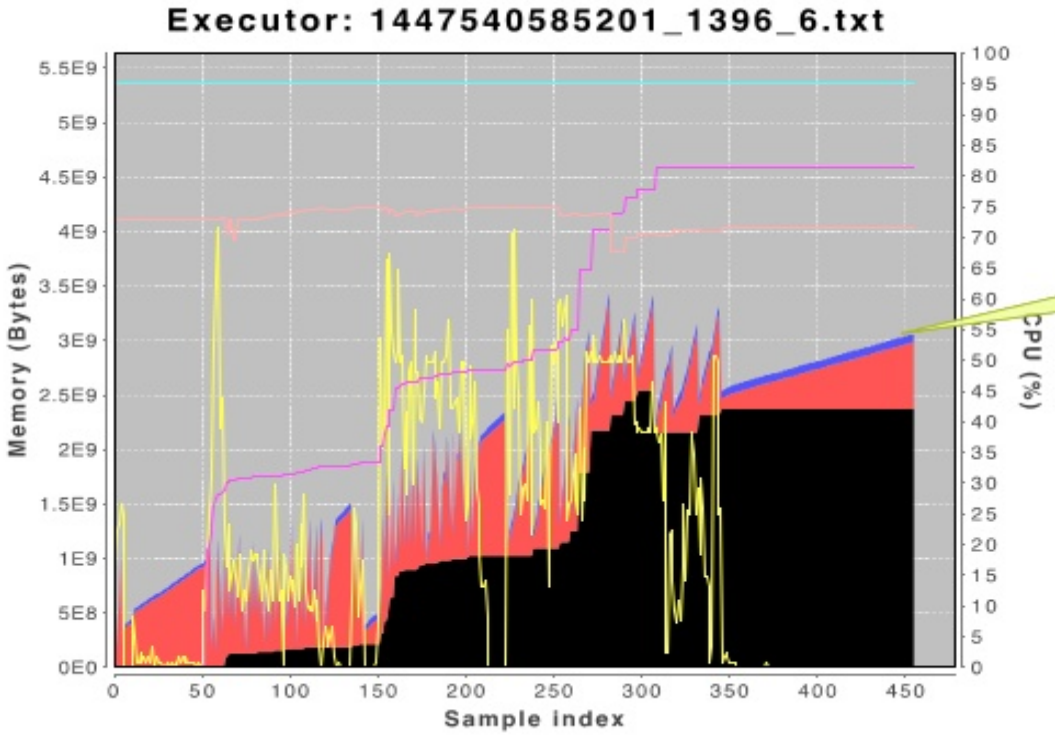
But, what did we do now?

Reliability is achieved at the cost of extra CPU cycles spent in GC, degrading performance by 15%

So far, we have sacrificed either performance or efficiency for reliability

Can we do better?

Fix #5: Spark can Exploit Structure in Data

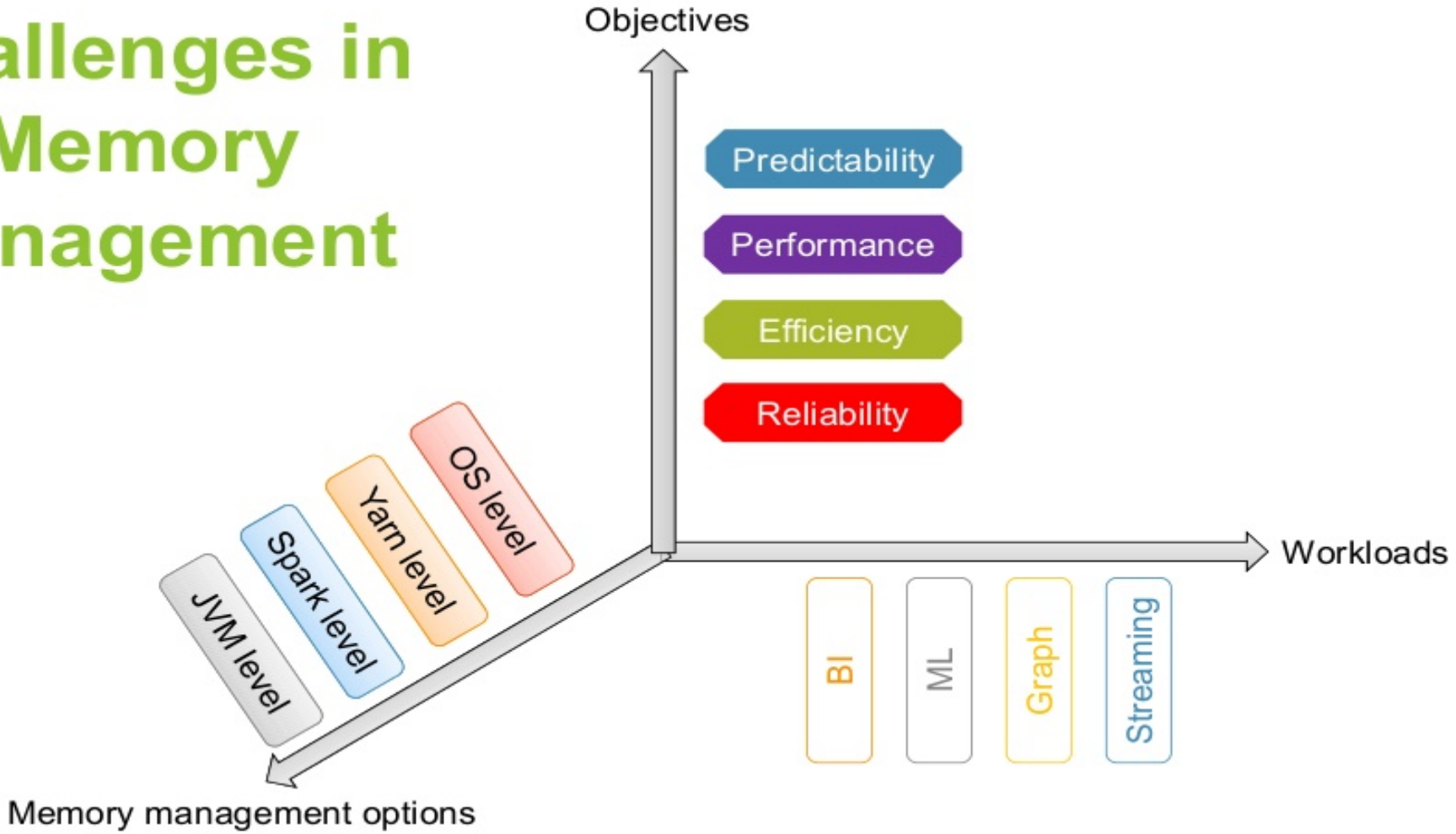


Tungsten's custom serialization reduces memory footprint while also reducing processing time

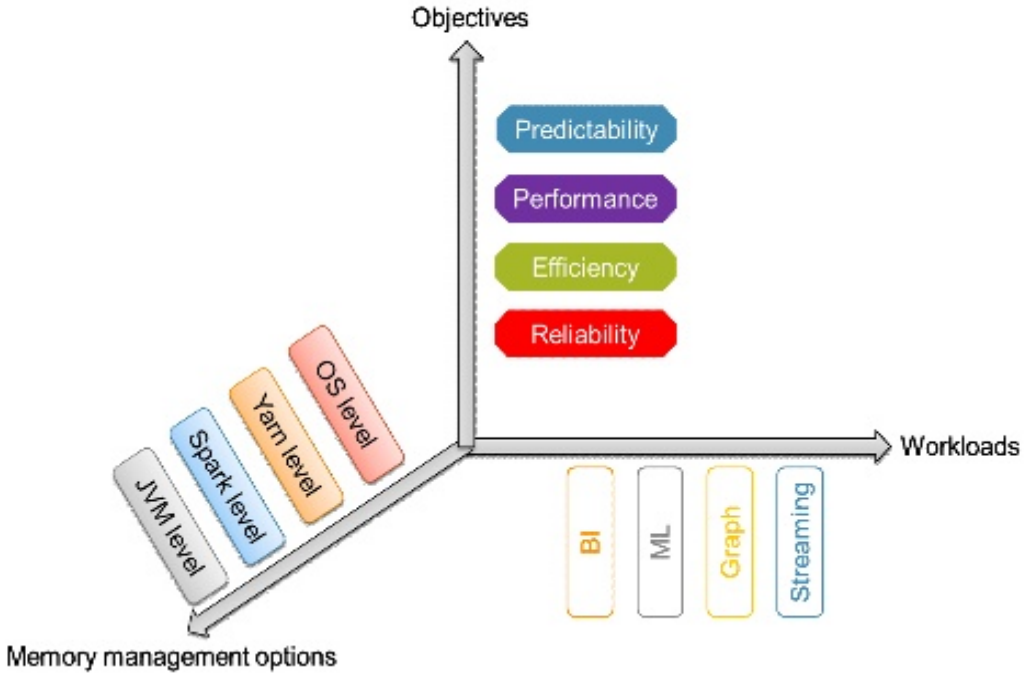
Application succeeds and runs 2x faster compared to Fix #2!

spark-submit **--class SortByKeyDF** --num-executors 10 --executor-memory 4G --executor-cores 16

Challenges in Memory Management



Next



- Key insights from experimental analysis
- Current work

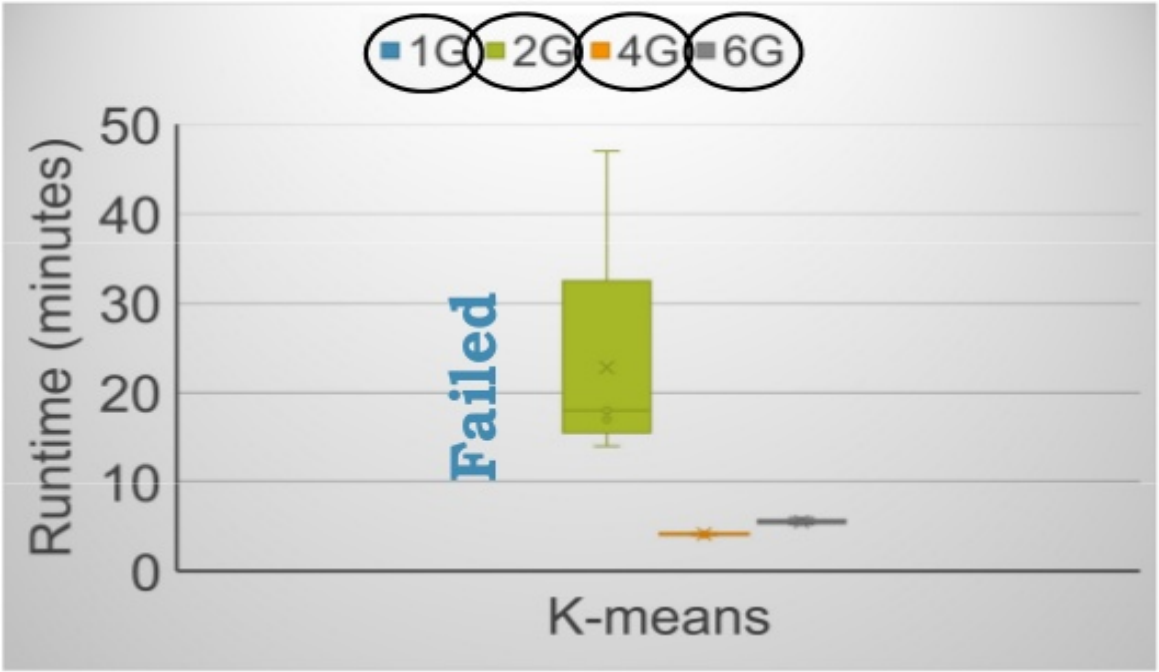
Yarn-level Memory Management

Yarn-level Memory Management

- Executor memory
- OS memory overhead per executor
- Cores per executor
- Number of executors

Impact of Changing Executor Memory

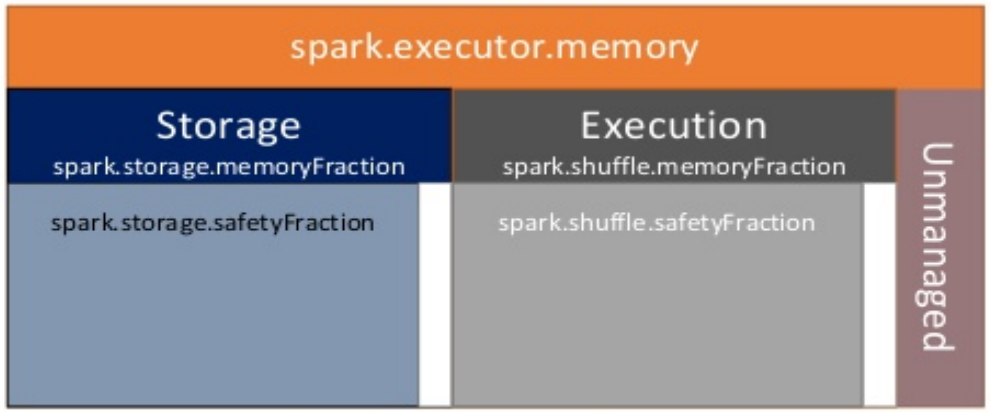
Reliability | Predictability | Performance | Efficiency



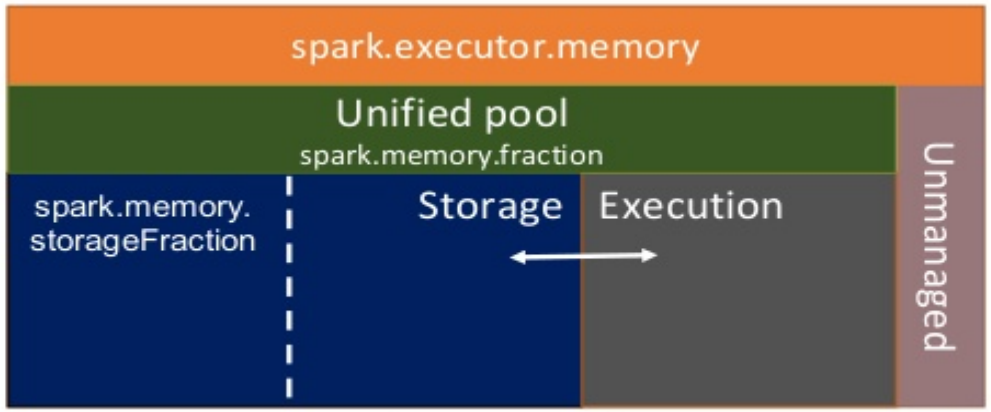

```
java.lang.OutOfMemoryError: Java heap space
        at java.util.Arrays.copyOf(Arrays.java:2271)
        at
java.io.ByteArrayOutputStream.grow(ByteArrayO
utputStream.java:118)
...
        at
org.apache.spark.storage.BlockManager.dataSeri
alize(BlockManager.scala:1202)
...
        at
org.apache.spark.CacheManager.putInBlockMan
ager(CacheManager.scala:175)
```

Spark-level Memory Management

Spark-level Memory Management



Legacy



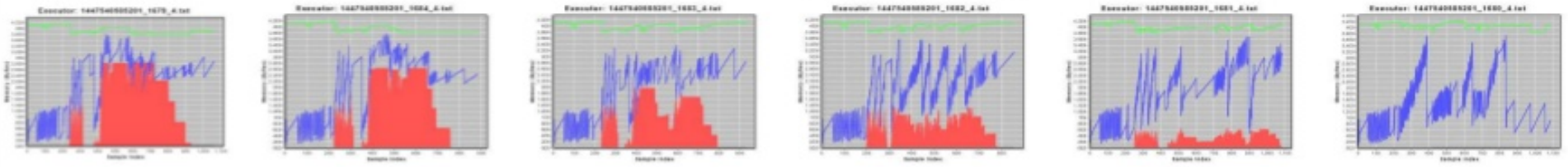
Unified

Spark-level Memory Management

- Legacy or unified?
  - If legacy, what is size of storage pool Vs. execution pool?
- Caching
  - On heap or off-heap (e.g., Tachyon)?
  - Data format (deserialized or serialized)
  - Provision for data unrolling
- Execution data
  - Java-managed or Tungsten-managed

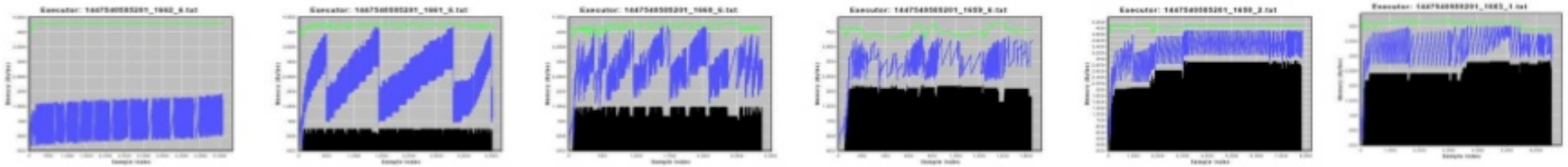
Comparing Legacy and Unified

SortByKey



Unified

Increasing storage pool size,
Decreasing execution pool size



K-Means

Unified

Spark Storage Spark Execution Used Heap Max Heap

Unified does as Expected, But…

Performance

Predictability

Executors fail due to OOM errors while receiving shuffle blocks

```
java.lang.OutOfMemoryError: Java heap space
        at java.util.Arrays.copyOf(Arrays.java:2271)
        at
java.io.ByteArrayOutputStream.grow(ByteArrayO
utputStream.java:118)
...
        at
org.apache.spark.storage.BlockManager.dataSeri
alize(BlockManager.scala:1202)
...
        at
org.apache.spark.network.netty.NettyBlockRpcSe
rver.receive(NettyBlockRpcServer.scala:58)
```

Unified Memory Manager is:

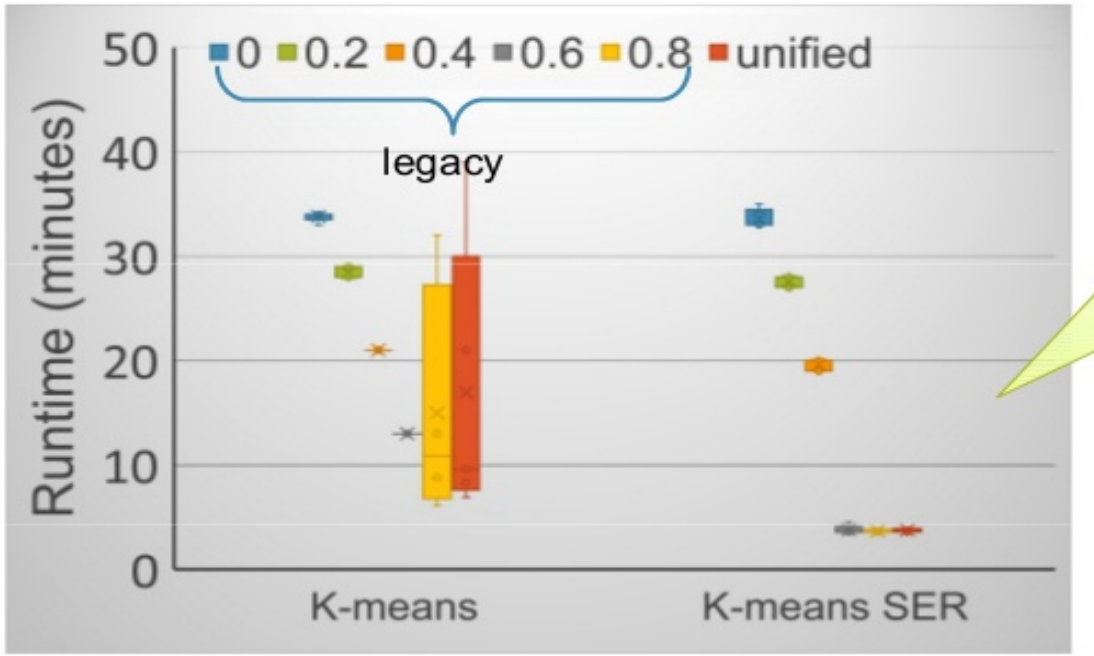
- A step in the right direction
- Not *unified* enough

Spark-level Memory Management

- Legacy or unified?
  - If legacy, what is size of storage pool Vs. execution pool?
- Caching
  - On heap or off-heap (e.g., Tachyon)?
  - Data format (deserialized or serialized)
  - Provision for data unrolling
- Execution data
  - Java-managed or Tungsten-managed

Deserialized Vs. Serialized cache

Performance | Predictability | Efficiency



Memory footprint of data in cache goes down by ~20% making more partitions fit in the storage pool

Another Application, Another Story!

Reliability

Predictability

Executors fail due to OOM errors while serializing data

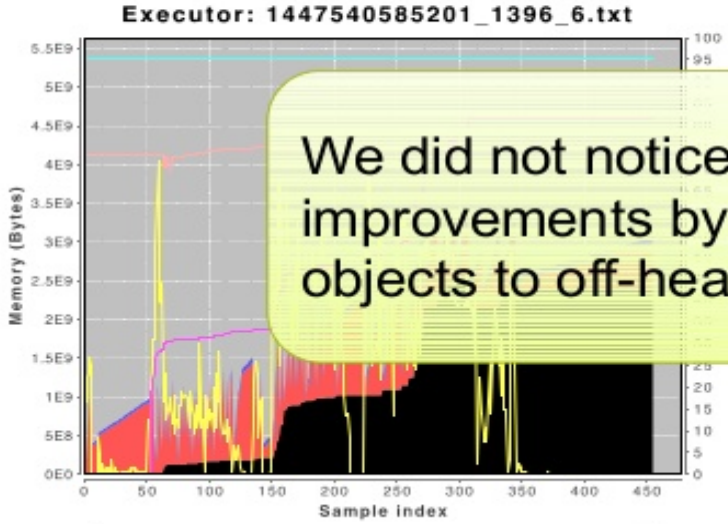
```
java.lang.OutOfMemoryError: Java heap space
        at java.util.Arrays.copyOf(Arrays.java:2271)
        at
java.io.ByteArrayOutputStream.grow(ByteArrayO
utputStream.java:118)
...
        at
org.apache.spark.storage.BlockManager.dataSeri
alize(BlockManager.scala:1202)
...
        at
org.apache.spark.CacheManager.putInBlockMan
ager(CacheManager.scala:175)
```

Spark-level memory management

- Legacy or unified?
  - If legacy, what is size of storage pool Vs. execution pool?
- Caching
  - On heap or off-heap (e.g., Tachyon)?
  - Data format (deserialized or serialized)
  - Provision for data unrolling
- Execution data
  - Java-managed or Tungsten-managed

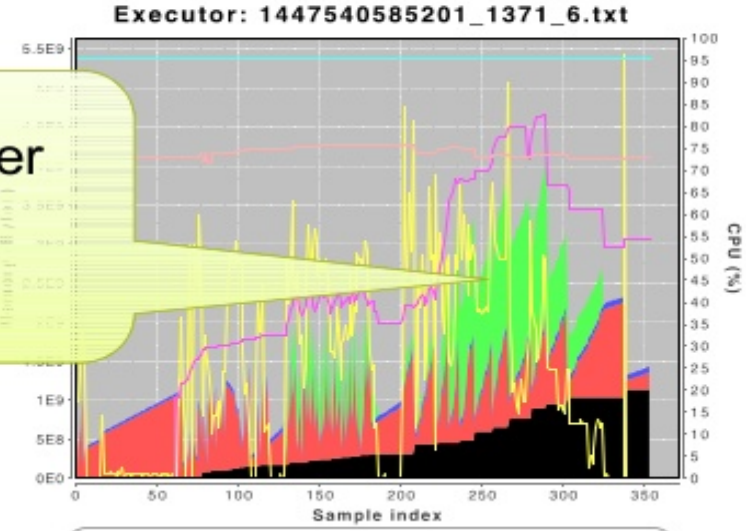
Execution Data Management

We have seen that Tungsten-managed heap improves the performance significantly. (Fix #5)



All objects in Heap

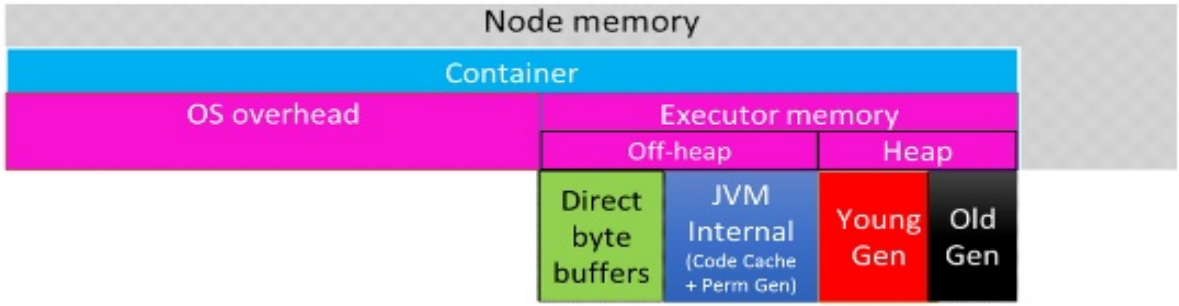
We did not notice much further improvements by pushing objects to off-heap



Up to 2GB objects in off-heap at any time

JVM-level Memory Management

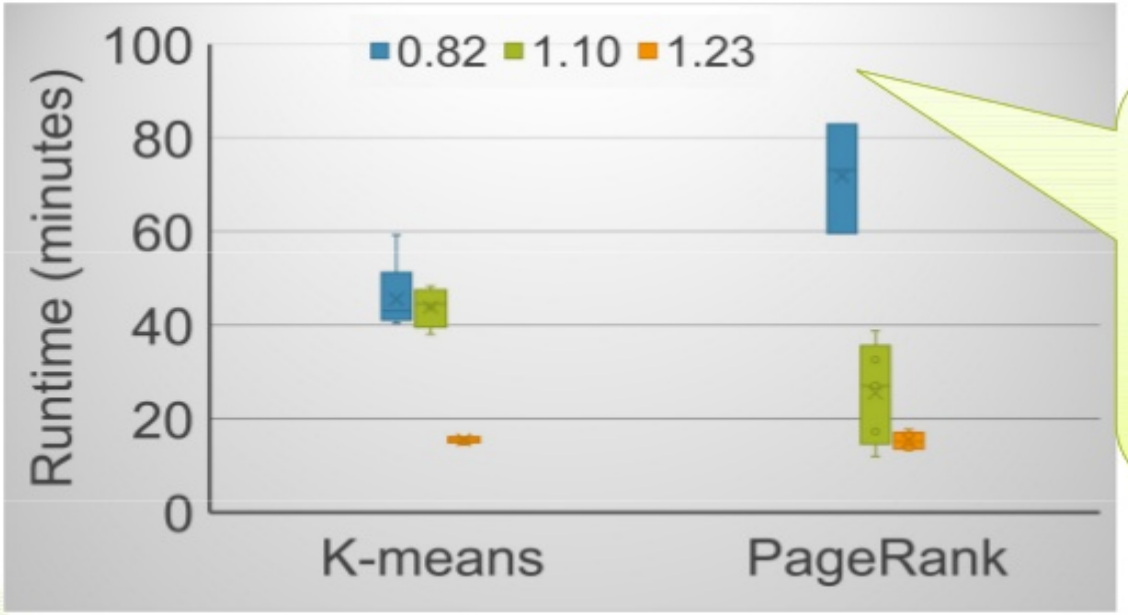
JVM-level Memory Management



- Which GC algorithm? (Parallel GC, G1 GC, …)
- Size cap for a GC pool
- Frequency of collections
- Number of parallel GC threads

Spark-JVM Interactions

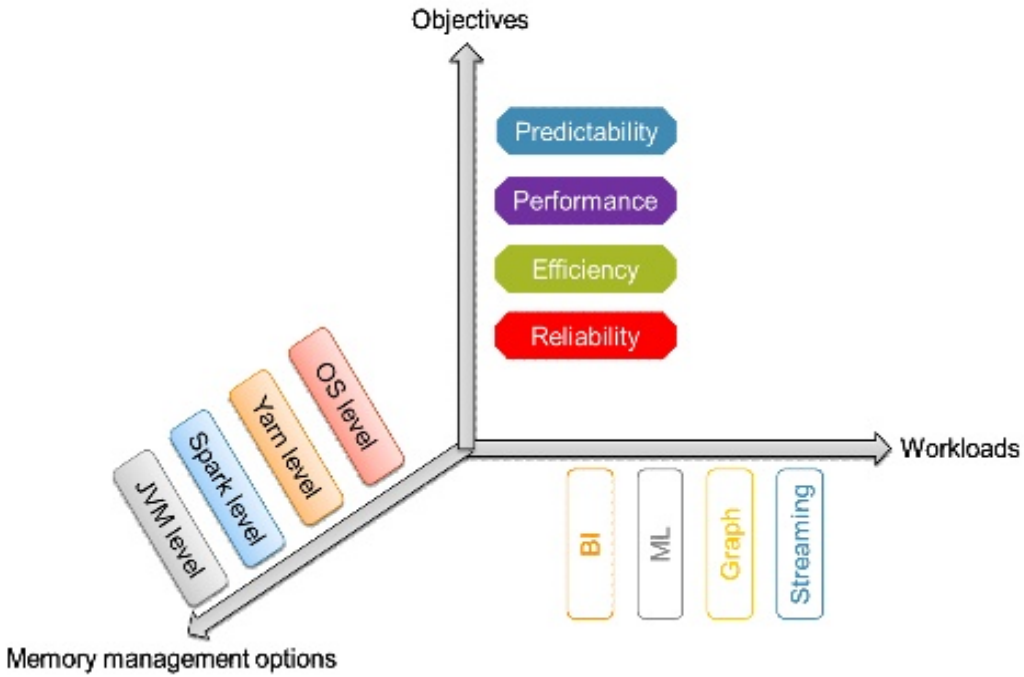
Keep JVM OldGen size at least as big as RDD cache



Keeping Spark storage pool size constant, the size of OldGen pool is increased from left to right

K-means executors display more skew in data compared to PageRank

Current Work

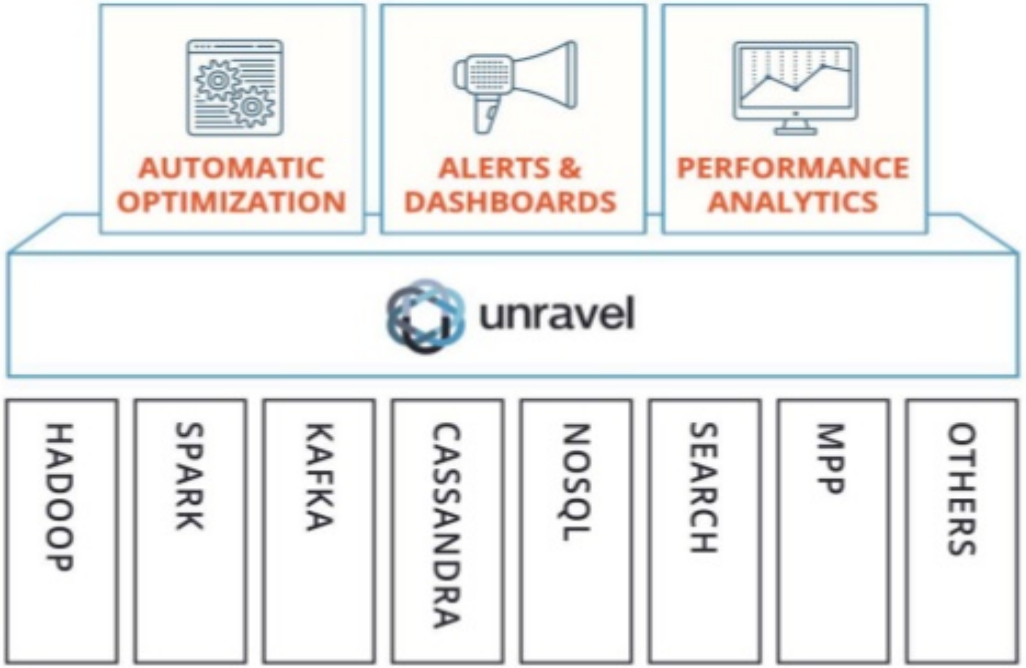


- Automatic root-cause analysis of memory-related issues
- Auto-tuning algorithms for memory allocation in multi-tenant clusters

unravel

Get Free Trial Edition:

bit.ly/getunravel



UNCOVER ISSUES
UNLEASH RESOURCES
UNRAVEL PERFORMANCE