

Tuning Java for Big Bacon...er Data

Scott Seighman
Systems Engineer
Oracle

✉ scott.seighman@oracle.com

🐦 JavaCleveland

Session Agenda

- 1 Symptoms, Errors, Performance Bottlenecks
- 2 Java Tuning Considerations / Examples
- 3 Summary / Q & A
- 4 **Appendix: Identifying Performance Bottlenecks**

Symptoms, Errors & Bottlenecks

- Symptoms:
 - Jobs get stuck, stall or fail
- Errors:
 - ‘Out of Memory’ (OOM)
 - ‘Error: Java Heap Space’
 - ‘GC overhead limit exceeded’
- Bottlenecks:
 - [Appendix](#): Methods/tools to identify bottlenecks

Say
'It depends'
one more time.



Thank You!

 scott.seighman@oracle.com  [@JavaCleveland](https://twitter.com/JavaCleveland)



They're more
what you'd call ...
GUIDELINES.

Somewhat Obvious Observation #1

Memory usage depends on
the job.

Memory Errors

- Memory tuning issues may appear in the form of one or more symptoms, depending on the circumstances:
 - Job stuck in mapper or reducer phase
 - Job hangs for long periods of time or completes much later than expected
 - The job completely fails and or terminates
- These symptoms are typical when the master instance type or slave nodes run out of memory

Error: java.lang.RuntimeException: java.lang.OutOfMemoryError

Memory Considerations

- Memory available to each JVM task (Map or Reduce) is managed by the **mapreduce.child.java.opts** property specified in **mapred-site.xml**
- To specify different memory available to Map and Reduce JVM tasks use the following options:
 - **mapreduce.map.child.java.opts**
 - **mapreduce.reduce.child.java.opts**

mapreduce.child.java.opts



(map | reduce)

-Xmx512m



(default may
vary)

The memory available to each task (JVM). Other JVM options can also be provided in this property

Memory Considerations *(continued)*

- Any other VM options for map and reduce tasks can be specified via properties with space as delimiter:

<property>

<name>mapreduce.map.child.java.opts</name>

<value>-Xmx512M -Djava.net.preferIPv4Stack=true</value>

</property>

<property>

<name>mapreduce.reduce.child.java.opts</name>

<value>-Xmx1024M</value>

</property>

Memory Considerations

- Suggested formula to avoid Java Heap space error:

$$\begin{aligned} &(\text{num_of_maps} * \text{map_heap_size}) \\ &+ \\ &(\text{num_of_reducers} * \text{reduce_heap_size}) \end{aligned}$$

- Make certain result is not larger than memory available in the system otherwise you will get "Error: Java Heap space" error

Memory Considerations

- **mapreduce.map.memory.mb** is the upper memory limit that Hadoop allows to be allocated to a mapper, in megabytes
 - **Default is 512.** If this limit is exceeded, Hadoop will kill the mapper
 - If the mapper process runs out of heap memory, the mapper throws a Java 'Out of Memory Exception'
- The Java heap settings should be smaller than the Hadoop container memory limit because we need reserve memory for Java code
 - Typically, it is recommended you **reserve 20% memory** for code to avoid experiencing "Killing container" errors
- If you experience Java out of memory errors, consider increasing both memory settings

Tip: Configuration Files

- Property names in configuration files change from release to release
 - Make certain the property is applicable to your release
 - Ex: `mapred.map.child.java.opts` is deprecated
 - Not all properties are backwards compatible
- Some properties require Hadoop re-start to take effect
- Some properties will apply when new files are written in HDFS and will not apply to files already in HDFS

Memory Usage Summary

- Balance of memory properties may improve performance:
 - `mapreduce.map.child.java.opts`
 - `mapreduce.reduce.child.java.opts`
 - `mapreduce.map.memory.mb`
- Consider Java Heap formula
- Reserve 20% of memory for code



Your mileage may vary

Not Completely Obvious Observation #2

GC behavior can vary, based on code and workloads.

Garbage Collector Primer

- achieves automatic memory management through the following operations:
 - Allocating objects to a young generation and promoting aged objects into an old generation
 - Finding live objects in the old generation through a concurrent (parallel) marking phase.
 - The VM triggers the marking phase when the total Java heap occupancy exceeds the default threshold
 - Recovering free memory by compacting live objects through parallel copying

Garbage Collector Primer

- In typical applications, most objects "die young"
- **True for HBase**
- Most modern garbage collectors accommodate this by organizing the heap into generations
- Java HotSpot manages four generations:
 - **Eden** for all new objects
 - **Survivor I and II** where surviving objects are promoted when Eden is collected
 - **Tenured** space for objects surviving a few rounds (16 by default) of Eden/Survivor collection are promoted into the tenured space
 - **PermGen*** for classes, interned strings, and other more or less permanent objects

*PermGen removed in Java 8

Garbage Collector Primer

- The JDK provides **four different garbage collectors**, each with its own unique advantages and disadvantages
- Serial Collector
- Parallel Collector (*throughput collector*)
- Mostly Concurrent collectors

Garbage Collector Primer: **Serial Collector**

- The **serial collector** is the simplest , and the one you probably won't be using, mainly because it's designed for **single-threaded environments** (e.g. 32 bit or Windows) and for small heaps
- This collector freezes all application threads whenever it's working, which disqualifies it for all intents and purposes from being used in a server environment
- Usage: *-XX:+UseSerialGC*

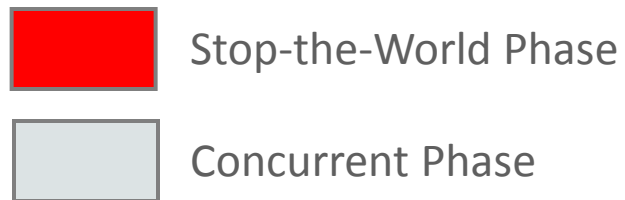
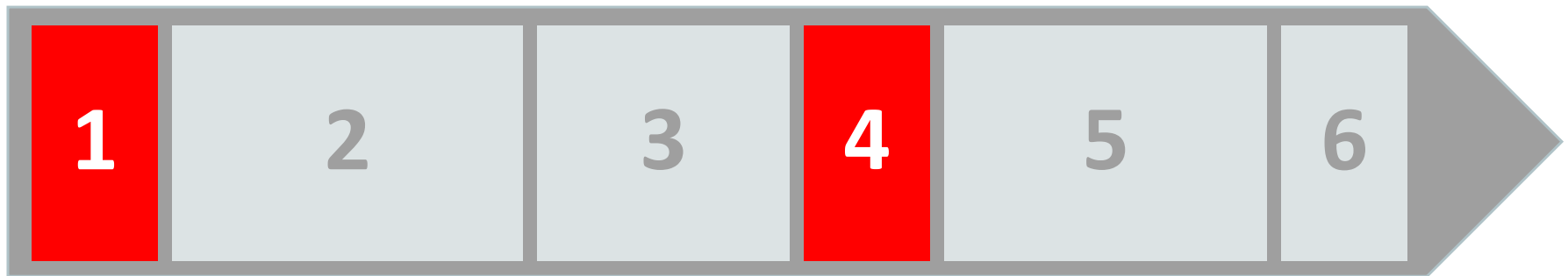
Garbage Collector Primer: **Parallel Collector**

- The parallel collector is the JVM's **default collector**
- Much like its name, its biggest advantage is that it uses multiple threads to scan through and compact the heap
- The downside to the parallel collector is that it will stop application threads when performing either a minor or full GC collection
- The parallel collector is best suited for apps that can tolerate application pauses and are trying to optimize for lower CPU overhead caused by the collector
- Usage: *-XX:+UseParallelGC*

Garbage Collector Primer: CMS Collector

- The CMS collector (“**concurrent-mark-sweep**”) uses multiple threads (“concurrent”) to scan through the heap (“mark”) for unused objects that can be recycled (“sweep”)
- This algorithm will enter “**stop the world**” (STW) mode in two cases:
 - When initializing the initial marking of roots (objects in the old generation that are reachable from thread entry points or static variables)
 - When the application has changed the state of the heap while the algorithm was running concurrently, forcing it to go back and do some final touches to make sure it has the right objects marked
- Usage: `-XX:+UseConcMarkSweepGC`
-XX:+UseParNewGC used in combination with CMS

CMS Collector Phases



Source: https://blogs.oracle.com/jonthecollector/entry/hey_joe_phases_of_cms

Garbage Collector Primer: **G1 Garbage Collector**

- The Garbage First Garbage Collector (G1 GC) is the low-pause, server-style generational garbage collector
- G1 GC uses concurrent and parallel phases to achieve its target pause time and to maintain good throughput
- When G1 GC determines that a garbage collection is necessary, it collects the regions with the least live data first (garbage first)

Garbage Collector Primer: **G1 Garbage Collector**

- The G1 GC achieves automatic memory management through the following operations:
 - Allocating objects to a young generation and promoting aged objects into an old generation.
 - Finding live objects in the old generation through a concurrent (parallel) marking phase. The Java HotSpot VM triggers the marking phase when the total Java heap occupancy exceeds the default threshold
 - Recovering free memory by compacting live objects through parallel copying

Garbage Collector Primer: **G1 Garbage Collector**

- The G1 GC is a regionalized and generational garbage collector, which means that the Java object heap (heap) is divided into a number of equally sized **regions**
- Upon startup, JVM sets the region size
- The region sizes can vary from 1 MB to 32 MB depending on the heap size
- The goal is to have no more than 2048 regions
- The eden, survivor, and old generations are logical sets of these regions and are not contiguous.
- More on the G1 GC:
 - <http://www.oracle.com/technetwork/articles/java/g1gc-1984535.html>

FYI: Java 8 and PermGen

- PermGen has been removed, successor is **Metaspace**
 - Classes metadata is now stored in the native heap
 - By default, class metadata allocation is only limited by the amount of available native memory
- Options **PermSize** and **MaxPermSize** have also been removed in JDK 8
- More info:
 - https://blogs.oracle.com/poonam/entry/about_g1_garbage_collector_permanent

FYI: Java 8 and the G1 Collector

- Beginning with **Java 8 update 20**, we introduced the G1 Collector **String deduplication**
 - Since strings (and their internal `char[]` arrays) fills much of our heap, a new optimization has been added
- Enables the G1 collector to identify strings which are duplicated more than once across your heap and correct them to point to the same internal `char[]` array
- Avoids multiple copies of the same string residing within the heap
- Usage: `-XX:+UseStringDeduplication`

Garbage Collector Primer: **Logging Options**

Option Flags	Description
-XX:+PrintGC	Print a short message after each GC
-verbose:gc	Same as "-XX:+PrintGC"
-XX:+PrintGCTimeStamps	Print a time stamp relative to JVM start when GC occurs
-XX:+PrintGCDateStamps	Print date/time stamp when GC occurs
-XX:+PrintGCDetails	Print a detailed message after each GC
-Xloggc:<file>	Force GC message to be logged to a file

GC Log Example

Reason for GC



54.145: [GC (Allocation Failure)



Time relative to JVM start

54.145: [DefNew: 17348K->15388K(19648K), 0.0000226 secs]

GC type (Young Gen)



Object size **before** GC



Object size **after** GC



Total area size



Time spent on GC



CPU time report of the entire GC operation



[Times: user=0.02 sys=0.00, real=0.01 secs]

JVM



OS Kernel



Elapsed time



Garbage Collector Primer

- When does the choice of a garbage collector matter?
- For some applications, the answer is never
- That is, the application can perform well in the presence of garbage collection with pauses of modest frequency and duration
- However, this is not the case for a large class of applications, particularly those with large amounts of data (multiple gigabytes), many threads, and high transaction rates

Excessive I/O Wait Due To Swapping

- If system memory is heavily overcommitted, the kernel may enter a vicious cycle, using up all of its resources swapping Java heap back and forth from disk to RAM as Java tries to run garbage collection
- Nothing good happens when swapping occurs

G1 GC Guidelines

- **Do not Set Young Generation Size**

- Explicitly setting young generation size via `-Xmn` meddles with the default behavior of the G1 collector.
 - G1 will no longer respect the pause time target for collections. So in essence, setting the young generation size disables the pause time goal
 - G1 is no longer able to expand and contract the young generation space as needed. Since the size is fixed, no changes can be made to the size

- **Response Time Metrics**

- Instead of using average response time (ART) as a metric to set the `XX:MaxGCPauseMillis=<N>`
- Consider setting value that will meet the goal 90% of the time or more
- This means 90% of users making a request will not experience a response time higher than the goal

G1 GC Guidelines

- A promotion failure (**Evacuation Failure**) occurs when a JVM runs out of heap regions during GC for either survivors and promoted objects
- The heap can't expand because it is already at max. Indicated by '**to-space overflow**' in GC log
- **This is expensive!**
 - GC still has to continue so space has to be freed up.
 - Unsuccessfully copied objects have to be tenured in place.
 - Any updates to RSets of regions in the CSet have to be regenerated

G1 GC Guidelines

- To avoid evacuation failure, consider the following options:
 - Increase heap size
 - Increase the **-XX:G1ReservePercent=n**, the default is 10.
 - G1 creates a false ceiling by trying to leave the reserve memory free in case more 'to-space' is desired
 - Start the marking cycle earlier
 - Increase the number of marking threads using the **-XX:ConcGCThreads=n**

G1 GC Guidelines

- Use **-XX:+ParallelRefProcEnabled**
 - When this flag is turned on, GC uses multiple threads to process the increasing references during Young and mixed GC
- **-XX:-ResizePLAB** and **-XX:ParallelGCThreads = 8 + ((N - 8) * 5 / 8)**
 - With both settings, we are able to see smoother GC pauses during the run
- Change **-XX:G1NewSizePercent** default from 5 to 1 for 100GB heap

*Source: <http://blog.cloudera.com/blog/2014/12/tuning-java-garbage-collection-for-hbase/>

G1 GC Guidelines

- Use JDK1.7_u60+ (better performance)

JVM Version	Avg. Pause Times*
JDK7_21	1.548 ms
JDK7_60	0.369 ms

- Consider G1 for heap sizes 16GB+
- The biggest benefit of G1 is that it doesn't compact like CMS (Full GC) does and your heap doesn't get fragmented like crazy over time and have crazy long pauses like you can see with CMS

*Source: <http://blog.cloudera.com/blog/2014/12/tuning-java-garbage-collection-for-hbase/>

HBase: MSLAB, CMS, ParallelGC

- Optimal configuration settings:
 - `HBASE_OPTS="-XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:CMSInitiatingOccupancyFraction=70 -XX:+CMSParallelRemarkEnabled"`
- Other option:
 - `HBASE_OPTS="-server -XX:+UseParallelGC XX:+UseParallelOldGC -XX:ParallelGCThreads=8"`
- MSLAB settings:
 - `hbase.hregion.memstore.mslab.enabled=true`
 - `hbase.hregion.memstore.mslab.chunksize=2MB` (default)
 - `hbase.hregion.memstore.mslab.max.allocation=256KB` (default)

HBase GC Tuning Guidelines

Parameter	Description
-Xmn256m	Small Eden space
-XX:+UseParNewGC	Collect Eden in parallel
-XX:+UseConcMarkSweepGC	Use the non-moving CMS collector
-XX:CMSInitiatingOccupancyFraction=70	Start collecting when 70% of tenured gen is full to avoid forced collection
-XX:+UseCMSInitiatingOccupancyOnly	Do not attempt to adjust CMS setting

- You should experience average GC times between 50ms - 150ms
- Considering hardware with heaps of 32GB or less
- If your application maintains a large cache of long-lived objects in the heap, consider increasing the threshold triggering setting:
-XX:CMSInitiatingOccupancyFraction=90

Java 8 Observations

- Java 8 allocates extra virtual memory (when compared to Java 7), may need to control the non-heap memory usage by limiting the max allowed values for some JVM parameters:

'-XX:ReservedCodeCacheSize=100M

-XX:MaxMetaspaceSize=128m

-XX:CompressedClassSpaceSize=128m'

- Java 8 requires additional heap
 - Consider increasing **mapreduce.reduce.memory.mb**
 - Consider increasing **Xmx** (**mapreduce.reduce.java.opts**)

GC Summary

- Make certain you're using JDK7_U60+
- Tuning the GC may result in negligible pause times
 - Test the guidelines
- Enable GC logging to determine where it's spending time
- G1 Collector may be appropriate for your workload



Your mileage may vary

Another not-so-obvious Observation #3

JVM reuse can add
efficiencies.

JVM Reuse

- Map/Reduce tasks are executed by JVM processes which are forked by the TaskTracker
- Creation of a JVM, which includes initialization of execution environments is costly, especially when the number of tasks is large
- JVM Reuse is an optimization of reusing JVMs for multiple tasks

JVM Reuse

- Hadoop by default launches a new JVM for each for map or reduce job and each run the map/reduce tasks in parallel and in isolation
- When we have long initialization process which takes significant time and our map/reduce method takes hardly a few seconds, then spawning a new JVM for each map/reduce is overkill
- With JVM reuse the tasks sharing the JVM will run serially instead of in parallel
- Note : This property specifies **tasks** and not map or reduce tasks
- Setting the value to -1 means all the tasks for a job will use the same JVM

JVM Reuse

- When reusing the same JVM, Hotspot will build those hotspots (mission critical sections) into native machine code which boosts performance optimization
- Can be useful for the long running tasks
 - Short-lived processes won't benefit

JVM Reuse Recommendations

- For large number of small tasks, set this property to -1
 - You should realize a significant performance improvement.
- With long running tasks, it's good to recreate the task process
 - Due to issues like heap fragmentation degrading your performance
- For middle-of-the-road jobs, you could reuse 2-3 tasks

mapreduce.job.reuse.jvm.num.tasks | 1 | Reuse single JVM

JVM Reuse Summary

- Can reduce JVM startup times
- Can reduce number of JVMs
- HBase would be a good candidate



Your mileage may vary

Really? Aren't you done

Observation #4

General Java tidbits.

Runtime Errors

- `java.io.IOException: Too many open files`
 - See [ulimit and nproc configuration](#)
- System instability, and the presence of `"java.lang.OutOfMemoryError: unable to create new native thread in exceptions"`
 - See [ulimit and nproc configuration](#)

JDK Issues

- **NoSuchMethodError:**
`java.util.concurrent.ConcurrentHashMap.keySet`
- Check if you compiled with JDK8 and tried to run it on JDK7
 - If so, this won't work. Run on JDK8 or recompile with JDK7

Java Coding Guidelines

- Avoid frequent JNI calls (~100 instructions overhead)
- Write better Java loops to make use of vectorization (AVX2)
- Avoid implicit Strings
- Possible self-memory management to avoid GC pitfalls
- Use default Java libraries methods DON'T write your own
- Use floats instead of doubles whenever possible
- Lambdas make code more efficient

Source: Accelerating Hadoop Performance on Intel® Architecture Based Platforms – IDF14

Java 8 Lambdas: Word Count Example

```
1. JavaRDD<String> lines = sc.textFile("hdfs://logfile.txt");
2. JavaRDD<String> words = lines.flatMap(
3.   new FlatMapFunction<String, String>() {
4.     public Iterable<String> call(String line) {
5.       return Arrays.asList(line.split(" "));
6.     }
7.   });
8. JavaPairRDD<String, Integer> ones = words.mapToPair(
9.   new PairFunction<String, String, Integer>() {
10.    public Tuple2<String, Integer> call(String w) {
11.      return new Tuple2<String, Integer>(w, 1);
12.    }
13.  });
14. JavaPairRDD<String, Integer> counts = ones.reduceByKey(
15.   new Function2<Integer, Integer, Integer>() {
16.     public Integer call(Integer i1, Integer i2) {
17.       return i1 + i2;
18.     }
19.  });
20. counts.saveAsTextFile("hdfs://counter.txt");
```



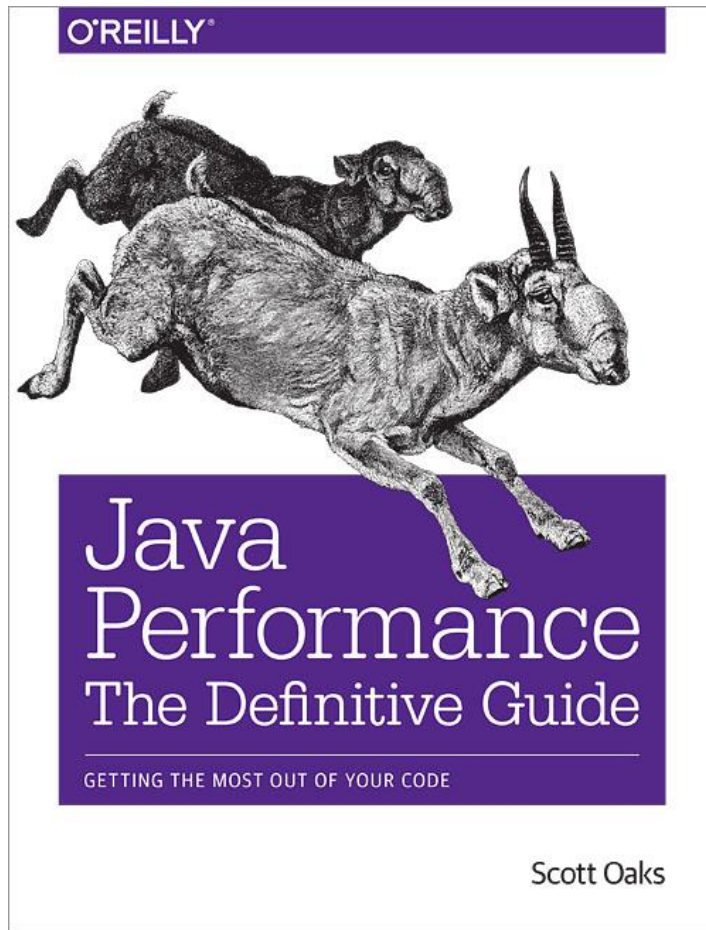
20 lines

```
1. JavaRDD<String> lines = sc.textFile("hdfs://logfile.txt");
2. JavaRDD<String> words =
3.   lines.flatMap(line -> Arrays.asList(line.split(" ")));
4. JavaPairRDD<String, Integer> counts =
5.   words.mapToPair(w -> new Tuple2<String, Integer>(w, 1))
6.   .reduceByKey((x, y) -> x + y);
7. counts.saveAsTextFile("hdfs://counter.txt");
```






7 lines

Resources



Summary

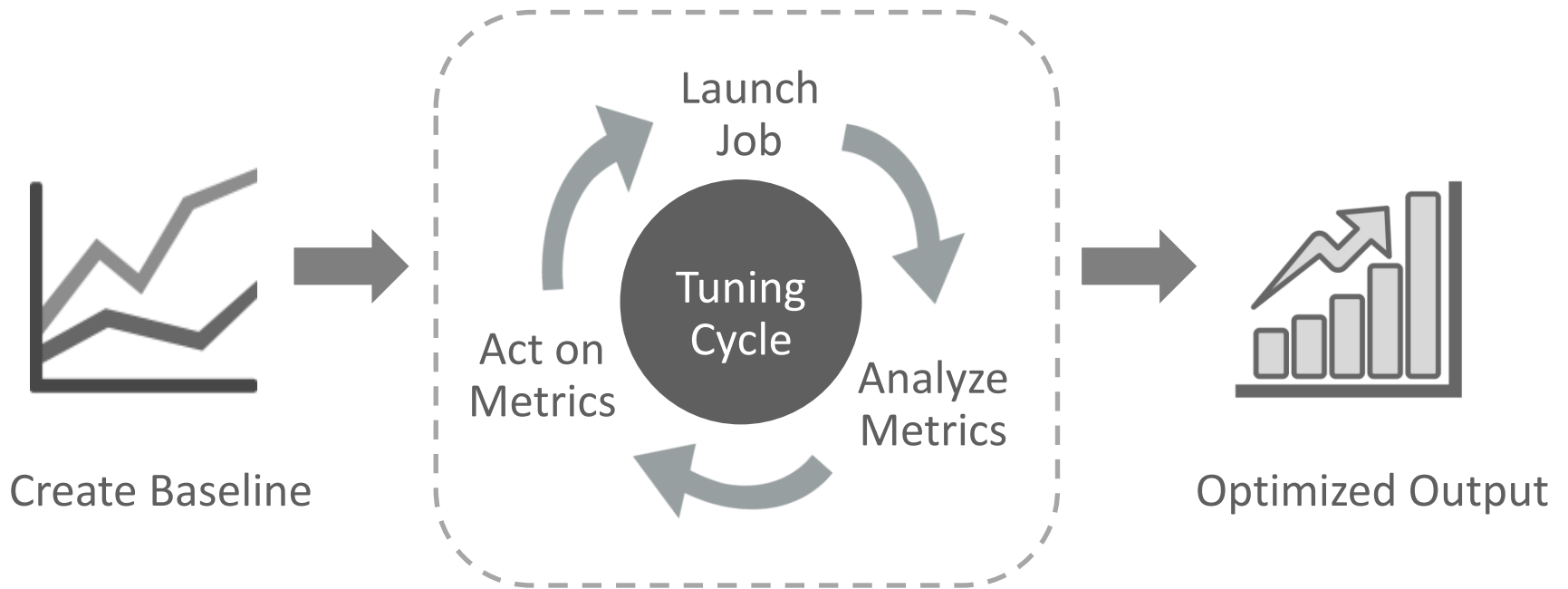
- No silver bullet for configuration parameters
 - Review hardware resources to properly allocate memory
 - Choose GC to fit your workload characteristics
 - Consider JVM reuse for appropriate workloads
- Launch, Analyze, Act
 - Use benchmark utilities (see Appendix), but the best benchmark is your app!
- Contact me:  scott.seighman@oracle.com  [@JavaCleveland](https://twitter.com/JavaCleveland)
- Cleveland Java Meetup:  <http://meetup.com/cleveland-java/>

Thank You!

 scott.seighman@oracle.com  [@JavaCleveland](https://twitter.com/JavaCleveland)

ORACLE®

Appendix: Identifying Performance Bottlenecks



Source: [Optimizing Hadoop for MapReduce](#)

Provided Benchmark Utilities

hadoop jar /usr/lib/hadoop-mapreduce/hadoop-*test*.jar

Utility	Description
DFSCIOTest	Distributed I/O benchmark of libhdfs
DistributedFSCheck	Distributed checkup of filesystem consistency
JHLogAnalyzer	Job history log analyzer
MRReliabilityTest	Tests reliability of MR framework by injecting faults/failures
SliveTest	HDFS stress test and live data verification
TestDFSIO	Distributed I/O benchmark
fail	A job that always fails
filebench	Benchmark SequenceFile(I O)Format (block,record compressed/uncompressed), Text(I O)Format (compressed/uncompressed)
largesorter	Large-sort tester
loadgen	Generic map/reduce load generator
mapredtest	A map/reduce test check

Provided Benchmark Utilities

hadoop jar /usr/lib/hadoop-mapreduce/hadoop-*test*.jar

Utility	Description
minicluster	Single process HDFS and MR cluster
mrbench	A map/reduce benchmark that can create many small jobs
nnbench	A benchmark that stresses the namenode
sleep	A job that sleeps at each map and reduce task
testbigmapoutput	MR program on very big non-splittable file, does identity MR
testfilesystem	A test for FileSystem read/write
testmapredsort	A MR program that validates the map-reduce framework's sort
testsequencefile	A test for flat files of binary key value pairs
testsequencefileinputformat	A test for sequence file input format
testtextinputformat	A test for text input format
threadedmapbench	A MR benchmark comparing performance of maps with multiple spills over maps with 1 spill

Benchmarks: **TestDFSIO**

- The **TestDFSIO** benchmark is a read and write test for HDFS
 - Stress testing HDFS
 - Discover performance bottlenecks in your network
 - Evaluate hardware, OS and Hadoop setup of your cluster machines (particularly the NameNode and the DataNodes)
 - Provides first impression of I/O speed
- Usage:

```
hadoop jar $HADOOP_HOME/hadoop-*test*.jar  
TestDFSIO -read | -write | -clean [-nrFiles N]  
[-fileSize MB] [-resFile resultFileName]  
[-bufferSize Bytes]
```

Benchmarks: TestDFSIO

----- TestDFSIO ----- : write

Date & time: Mon Dec 29 11:15:02 PST 2014

Number of files: 10

Total MBytes processed: 100.0

Throughput mb/sec: 2.999850007499625

Average IO rate mb/sec: 3.9975154399871826

IO rate std deviation: 2.3060773432541115

Test exec time sec: 136.892

Benchmarks: TestDFSIO

----- TestDFSIO ----- : write

Date & time: Mon Dec 29 11:47:56 PST 2014

Number of files: 4

Total MBytes processed: 100.0

Throughput mb/sec: 16.70285618840822

Average IO rate mb/sec: 17.350488662719727

IO rate std deviation: 3.004533919435319

Test exec time sec: 49.107

Benchmarks: **TestDFSIO**

- When interpreting **TestDFSIO** results, keep in mind:
 - The HDFS **replication factor** plays an important role
 - A **higher** replication factor leads to **slower** writes
 - If you compare two otherwise identical TestDFSIO write runs which use an HDFS replication factor of 2 and 3, respectively, you will see **higher throughput** and **higher average I/O numbers** for the run with the **lower replication factor**

Tip: Identifying Replication Factor

- The second column of the output will show the default replication factor:

```
[cloudera@quickstart]$ hdfs dfs -ls test/

Found 1 items

-rw-r--r--    1 cloudera cloudera  2506 2014-12-29 12:43 test/passwd
```

To change/modify the replication factor:

```
[cloudera@quickstart]$ hdfs dfs -setrep 2 test/passwd

Replication 2 set: test/passwd

[cloudera@quickstart]$ hdfs dfs -ls test/

Found 1 items

-rw-r--r--    2 cloudera cloudera  2506 2014-12-29 12:43 test/passwd
```


Benchmarks: **nnbench**

- **NameNode** benchmark
- Useful for load testing the NameNode hardware and configuration
- Generates a lot of HDFS-related requests with normally very small “payloads” for the sole purpose of putting a high HDFS management stress on the NameNode
- Can simulate requests for creating, reading, renaming and deleting files on HDFS
- Usage: **nnbench <options>**
- By default, the benchmark waits 2 minutes before it actually starts!

Benchmarks: **mrbench**

- **MapReduce** benchmark
- Loops a small job a number of times
- Complimentary benchmark to the “large-scale” TeraSort benchmark suite
 - **mrbench** checks whether *small* job runs are responsive and running efficiently on your cluster
- Puts focus on the MapReduce layer as its impact on the HDFS layer is very limited.

Benchmarks: **HiBench** (Intel)

- **HiBench** suite consists of a set of Hadoop programs including both synthetic micro-benchmarks and real-world applications
- Currently the benchmark suite contains eight workloads, classified into four categories: **Micro-benchmarks, Web Search, Machine Learning and HDFS benchmarks**
- The first seven are directly taken from their open source implementations
- The last one is an enhanced version of the DFSIO benchmark they extended to evaluate the aggregated bandwidth delivered by HDFS
- More info: <https://github.com/intel-hadoop/HiBench>

Benchmarks: **BigBench**

- **BigBench** is the first end-to-end big data analytics benchmark suite
- Due to the lack of standards and standard benchmarks, users have a hard time choosing the right systems for their requirements ... enter **BigBench**
- **BigBench** is an “**application-level**” benchmark
 - It captures operations performed at an application level via SQL queries and data mining operations, rather than low level operations such as, say, file I/O, or performance of specific function such as sorting or graph traversal

Benchmarks: **BigBench**

- **TPC-H** and **TPC-DS** benchmarks, developed by the Transaction Processing Performance Council, have been used for benchmarking big data systems
- The TPC-H benchmark has been implemented in Hadoop, Pig, and Hive
- A subset of TPC-DS has been used to compare query performance with implementations using Impala and Hive
 - While they have been used for measuring performance of big data systems, both TPC-H and TPC-DS are both “**pure SQL**” benchmarks and, thus, do not cover the new aspects and characteristics of big data and big data systems

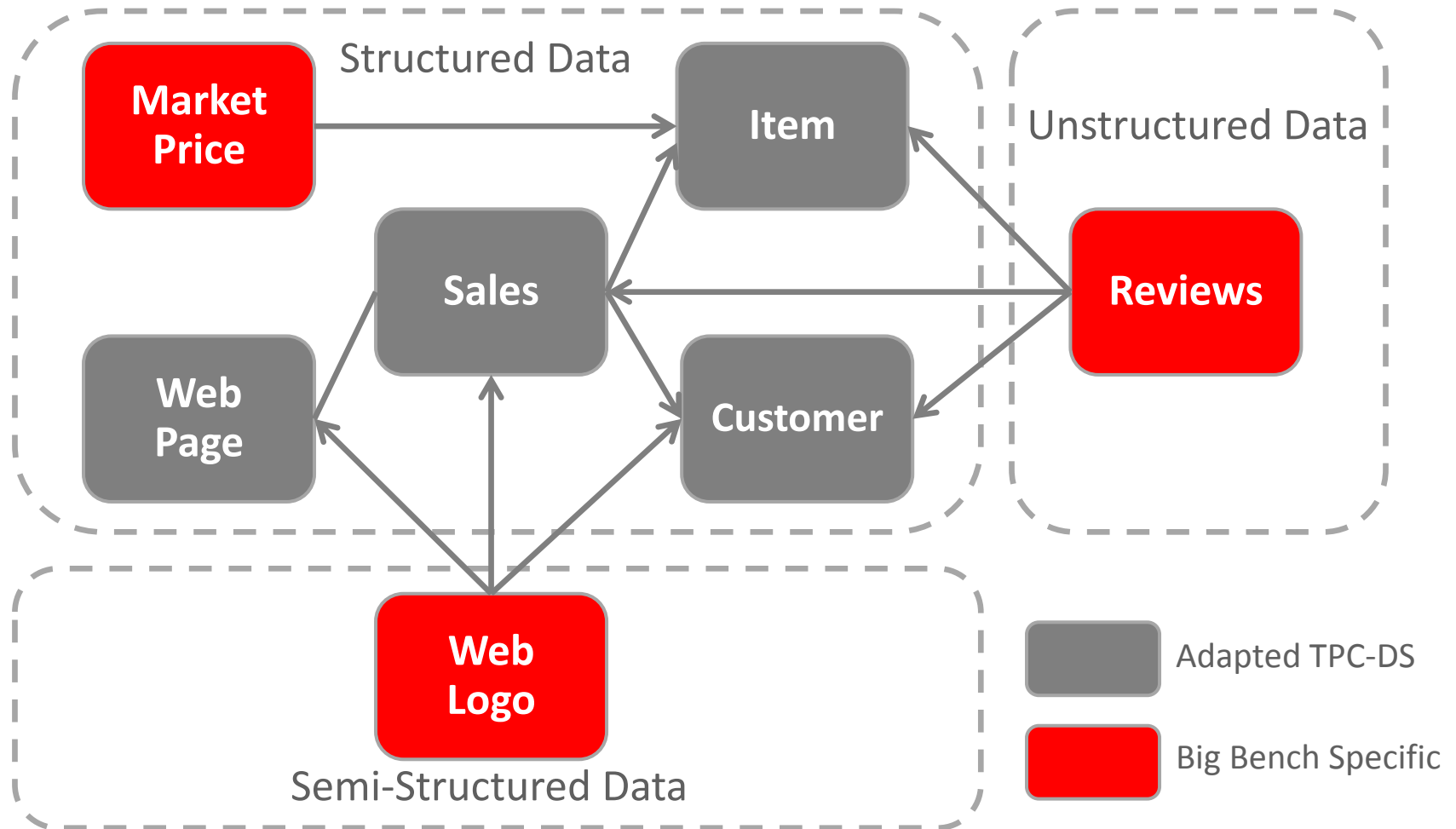
Benchmarks: **BigBench**

- The **BigBench** specification comprises two key components:
 - Data model specification
 - Workload/query specification

Benchmarks: **BigBench Data Model**

- The structured part of the **BigBench data model** is adopted from the **TPC-DS** data model **simulating a product retailer**
- The data model specification is implemented by a data generator, which is based on an extension of **PDGF** (Parallel Data Generation Framework)
- Plugins for PDGF enable data generation for an arbitrary schema
- Using the **BigBench plugin**, data can be generated for all three parts of the schema: **structured, semi-structured and unstructured**

Benchmarks: BigBench Data Model



Benchmarks: **BigBench Workload**

- **BigBench 1.0 workload** specification consists of **30 queries/workloads**
- **10** of these queries have been taken from the **TPC-DS** workload and run against the structured portion of the schema
- **20** were adapted from a McKinsey & Company report:
 - http://www.mckinsey.com/insights/business_technology/big_data_the_next_frontier_for_innovation
- **7** run against the semi-structured and **5** run against the unstructured
- Reference implementation:
 - <https://github.com/intel-hadoop/Big-Bench>

Benchmarks: **Terasort**

- Most well-known Hadoop benchmark
- The goal of TeraSort is to sort 1TB of data (or any other amount of data you want) as fast as possible
- It's a benchmark that combines **testing** the **HDFS** and **MapReduce** layers of a Hadoop cluster
- **TeraSort** benchmark suite is often used in practice to compare the results of our own cluster with the clusters of other people
- You can use the TeraSort benchmark to evaluate your Hadoop configuration after your cluster passed a convincing **TestDFSIO** benchmark

Benchmarks: **Terasort**

- A full **TeraSort** benchmark run consists of the following three steps:
 - Generating the input data via **TeraGen**
 - Running the actual **TeraSort** on the input data
 - Validating the sorted output data via **TeraValidate**
- You do not need to re-generate input data before every **TeraSort** run
- You can **TeraGen** for later **TeraSort** runs if you are satisfied with the generated data

Benchmarks: Terasort

- **TeraGen**

- Generates random data that can be conveniently used as input data for a subsequent TeraSort run
- Usage:

```
hadoop jar hadoop-examples.jar teragen <number of 100-byte rows> <output dir>
```

- **TeraGen** will run map tasks to generate the data and will not run any reduce tasks
- The default number of map task is defined by "**mapreduce.job.maps=2**" param
- Its only purpose is to generate the 1TB of random data in the following format:

"10 bytes key | 2 bytes break | 32 bytes ASCII/Hex | 4 bytes break | 48 bytes filler | 4 bytes break | \r\n"

Benchmarks: Terasort

- **TeraSort**

- Implemented as a MapReduce sort job with a custom partitioner that uses a sorted list of $n-1$ sampled keys that define the key range for each reduce
- Usage:

```
hadoop jar hadoop-examples.jar terasort <input dir> <output dir>
```

- This will spawn a series of map tasks that sort the ASCII key data
- There will be one map task for each HDFS block of data
- By default there will be one reduce task defined by `"mapreduce.job.reduces=1"`
- The data will be partitioned based on the number reduce tasks with a 1:1 ratio
- One partition for every reduce task

Benchmarks: Terasort

- **TeraValidate**

- Ensures that the output data of TeraSort is globally sorted
- Usage:

```
hadoop jar hadoop-examples.jar teravalidate <terasort  
output dir (= input data)> <teravalidate output dir>
```

- Reads the output data and ensures that each key is less than the next key in the entire dataset

Hadoop Vaidya Diagnostic Framework

- **Hadoop Vaidya** (means "one who knows", or "a physician")
- A rule based performance diagnostic tool for MapReduce jobs
- It performs a **post execution analysis** of map/reduce job by parsing and collecting execution statistics through job history and job configuration files
- It runs a set of **predefined tests/rules** against job execution statistics to diagnose various performance problems
- Each test rule **detects a specific performance problem** with the MapReduce job and provides a targeted advice to the user
- This tool **generates an XML report** based on the evaluation results of individual test rules.

Java Flight Recorder & Java Mission Control

- Together create a complete tool chain to continuously collect low level and detailed runtime information enabling after-the-fact incident analysis
- **Java Flight Recorder** is a profiling and event collection framework built into the Oracle JDK
 - Allows Java administrators and developers to gather detailed low level information about how the JVM and the Java application are behaving
- **Java Mission Control** is an advanced set of tools that enables efficient and detailed analysis of the extensive of data collected by Java Flight Recorder
- The tool chain enables developers and administrators to collect and analyze data from Java applications running locally or deployed in production environments
- More info:
 - <http://www.oracle.com/technetwork/java/javaseproducts/mission-control/index.html>

Appendix: Other Tuning Guides

HBase GC Tuning

- HBase is memory intensive, and using the default GC may cause long pauses in all threads including the *Juliet Pause* aka “STW”
- To debug (or confirm it’s happening), GC logging can be turned on in the JVM, in **hbase-env.sh** uncomment one of the lines below:

```
# This enables basic GC logging to the .out file  
# export SERVER_GC_OPTS="-verbose:gc -XX:+PrintGCDetails -  
XX:+PrintGCDateStamps"
```

```
# This enables basic GC logging to its own file  
# export SERVER_GC_OPTS="-verbose:gc -XX:+PrintGCDetails -  
XX:+PrintGCDateStamps -Xloggc:<FILE-PATH>"
```

HBase GC Tuning *(continued)*

This enables basic GC logging to its own file with automatic log rolling. Only applies to jdk 1.6.0_34+ and 1.7.0_2+.

```
# export SERVER_GC_OPTS="-verbose:gc -XX:+PrintGCDetails -  
XX:+PrintGCDateStamps -Xloggc:<FILE-PATH> -XX:+UseGCLogFileRotation -  
XX:NumberOfGCLogFiles=1 -XX:GCLogFileSize=512M"
```

If <FILE-PATH> is not replaced, the log file(.gc) would be generated in the HBASE_LOG_DIR

- At this point you should see logs like so:

```
64898.952: [GC [1 CMS-initial-mark: 2811538K(3055704K)] 2812179K(3061272K),  
0.0007360 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
```

```
64898.953: [CMS-concurrent-mark-start]
```

```
64898.971: [GC 64898.971: [ParNew: 5567K->576K(5568K), 0.0101110 secs] 2817105K-  
>2812715K(3061272K), 0.0102200 secs] [Times: user=0.07 sys=0.00, real=0.01 secs]
```

HBase GC Tuning *(continued)*

- Similarly, to enable GC logging for client processes, uncomment one of the below lines in **hbase-env.sh**:

This enables basic gc logging to the .out file.

```
# export CLIENT_GC_OPTS="-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCDateStamps"
```

This enables basic gc logging to its own file

```
# export CLIENT_GC_OPTS="-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCDateStamps  
-Xloggc:<FILE-PATH>"
```

This enables basic GC logging to its own file with automatic log rolling. Only applies to jdk 1.6.0_34+ and 1.7.0_2+.

```
# export CLIENT_GC_OPTS="-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCDateStamps  
-Xloggc:<FILE-PATH> -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=1 -  
XX:GCLogFileSize=512M"
```

If <FILE-PATH> is not replaced, the log file(.gc) would be generated in the HBASE_LOG_DIR .

Hadoop Configuration Options

- JVM options (hadoop-env.sh)
 - export HADOOP_NAMENODE_OPTS=""
 - Dcom.sun.management.jmxremote -Xms\${dfs.namenode.heapsize.mb}m
 - Xmx\${dfs.namenode.heapsize.mb}m
 - Dhadoop.security.logger=INFO,DRFAS
 - Dhdfs.audit.logger=INFO,RFAAUDIT
 - XX:ParallelGCThreads=8
 - XX:+UseParNewGC -XX:+UseConcMarkSweepGC
 - XX:+HeapDumpOnOutOfMemoryError -
 - XX:ErrorFile=\${HADOOP_LOG_DIR}/hs_err_pid%p.log \$HADOOP_NAMENODE_OPTS"
 - export HADOOP_DATANODE_OPTS=""
 - -Dcom.sun.management.jmxremote
 - -Xms\${dfs.datanode.heapsize.mb}m
 - -Xmx\${dfs.datanode.heapsize.mb}m
 - -Dhadoop.security.logger=ERROR,DRFAS \$HADOOP_DATANODE_OPTS"

Hadoop Configuration Options - YARN

- RM/NM JVM options (yarn-env.sh)

```
export YARN_RESOURCEMANAGER_HEAPSIZE=2GB
```

```
export YARN_NODEMANAGER_HEAPSIZE=2GB
```

```
YARN_OPTS="$YARN_OPTS -server
```

```
-Djava.net.preferIPv4Stack=true
```

```
-XX:+UseParNewGC -XX:+UseConcMarkSweepGC
```

```
-XX:+HeapDumpOnOutOfMemoryError
```

```
-XX:ErrorFile=${YARN_LOG_DIR}/hs_err_pid%p.log"
```