Understanding Spark Tuning

(Magical spells to stop your pager going off at 2:00am)

Holden Karau, Rachel Warren

Rachel

- Rachel Warren → She/ Her
- Data Scientist / Software engineer at Salesforce Einstein
- Formerly at Alpine Data (with Holden)
- Lots of experience scaling Spark in different production environments
- The other half of the High Performance Spark team :)
- @warre_n_peace
- Linked in: https://www.linkedin.com/in/rachelbwarren/
- Slideshare: https://www.slideshare.net/RachelWarren4/
- Github: https://github.com/rachelwarren

Holden:

- My name is Holden Karau
- Prefered pronouns are she/her
- Developer Advocate at Google
- Apache Spark PMC :)
- previously IBM, Alpine, Databricks, Google, Foursquare & Amazon
- co-author of Learning Spark & High Performance Spark
- @holdenkarau
- Slide share http://www.slideshare.net/hkarau
- Code review livestreams & live coding: https://www.twitch.tv/holdenkarau / https://www.youtube.com/user/holdenkarau
- Github https://github.com/holdenk
- Spark Videos http://bit.ly/holdenSparkVideos
- Talk feedback: http://bit.ly/holdenTalkFeedback

Who we think you wonderful humans are?

- Nice enough people
- I’m sure you love pictures of cats
- Might know something about using Spark, or are using it in production
- Maybe sys-admin or developer
- Are tired of spending so much time fussing with Spark Settings to get jobs to run

The goal of this talk is to give you the resources to **programatically** tune your Spark jobs so that they run **consistently** and **efficiently**

In terms of  **and $$$$$**

What we will cover?

- A run down of the most important settings
- Getting the most out of Spark's built-in Auto Tuner Options
- A few examples of errors and performance problems that can be addressed by tuning
  - A job can go out of tune over time as the world and it changes, much like Holden's Vespa.
- How to tune jobs "statically" e.g. without historical data
- How to collect historical data (meet Robin Sparkles: https://github.com/high-performance-spark/robin-sparkles)
- An example of using static and historical information to programmatically configure Spark jobs
- The latest and greatest auto tuner tools

I can haz application :p


```
val conf = new SparkConf()
 .setMaster("local")
.setAppName("my_awesome_app")
val sc = SparkContext.getOrCreate(newConf)

val rdd = sc.textFile(inputFile)
val words: RDD[String] = rdd.flatMap(_.split(“ “).
 map(_.trim.toLowerCase))
val wordPairs = words.map((_, 1))

val wordCounts = wordPairs.reduceByKey(_ + _)

wordCount.saveAsTextFile(outputFile)
```

Settings go here

This is a shuffle

I can haz application :p


```
val conf = new SparkConf()
 .setMaster("local")
.setAppName("my_awesome_app")
val sc = SparkContext.getOrCreate(newConf)

val rdd = sc.textFile(inputFile)
val words: RDD[String] = rdd.flatMap(_.split(“ “).
 map(_.trim.toLowerCase))
val wordPairs = words.map((_, 1))

val wordCounts = wordPairs.reduceByKey(_ + _)
```

Start of application

End Stage 1

```
wordCount.saveAsTextFile(outputFile)
```

Action, Launches Job

Stage 2

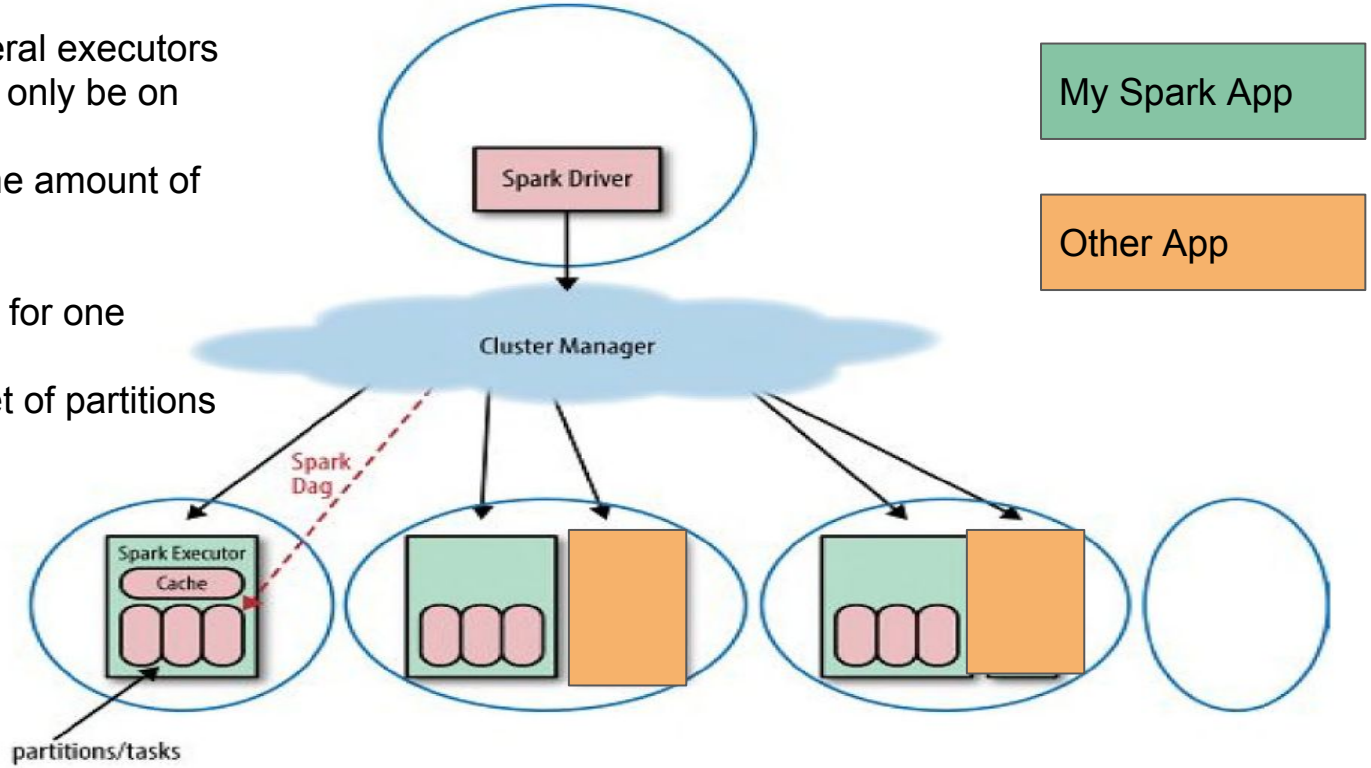
How many resources to give my application?

- Spark.executor.memory
- Spark.driver.memory
- spark.executor.vcores
- Enable dynamic allocation
  - (or set # number of executors)

```
val conf = new SparkConf()
 .setMaster("local")
 .setAppName("my_awesome_app")
 .set("spark.executor.memory", ???)
 .set("spark.driver.memory", ???)
 .set("spark.executor.vcores", ???)
```

Spark Execution Environment

- Node can have several executors
- But on executor can only be on one node
- Executors have same amount of memory and cores
- One task per core
- Task is the compute for one partition
- Rdd is distributed set of partitions



Executor and Driver Memory

- Driver Memory
  - As small as it can be without failing (but that can be pretty big)
  - Will have to be bigger is collecting data to the driver, or if there are many partitions
- Executor memory + overhead < less than the size of the container
- Think about binning
  - if you have 12 gig nodes making an 8 gig executor is maybe silly
- Pros Of Fewer Larger Executors Per Node
  - Maybe less likely to oom
  - Some tasks can take a long time
- Cons of Fewer Large Executors (Pros of More Small Executors)
  - Some people report slow down with more than 5ish cores … (more on that later)
  - If using dynamic allocation may be harder to “scale up” on a busy cluster

Vcores

- Remember 1 core = 1 task. So number of concurrent tasks is limited by total cores
  - Sort of, unless you change it. Terms and conditions apply to Python users.
- In HDFS too many cores per executor may cause issue with too many concurrent hdfs threads
  - maybe?
- 1 core per executor takes away some benefit of things like broadcast variables
- Think about “burning” cpu and memory equally
  - If you have 60Gb ram & 10 core nodes, making default executor size 30 gb but with ten cores maybe not so great

How To Enable Dynamic Allocation

Dynamic Allocation allows Spark to add and subtract executors between Jobs over the course of an application

- To configure
  - spark.dynamicAllocation.enabled=true
  - spark.shuffle.service.enabled=true (you have to configure external shuffle service on each worker)
  - spark.dynamicAllocation.minExecutors
  - spark.dynamicAllocation.maxExecutors
  - spark.dynamicAllocation.initialExecutors
- To Adjust
  - Spark will add executors when there are pending tasks (spark.dynamicAllocation.schedulerBacklogTimeout)
  - and exponentially increase them as long as tasks in the backlog persist (spark...sustainedSchedulerBacklogTimeout)
  - Executors are decommissioned when they have been idle for spark.dynamicAllocation.executorIdleTimeout

Why To Enable Dynamic Allocation

When

- Most important for shared or cost sensitive environments
- Good when an application contains several jobs of differing sizes
- The only real way to adjust resources throughout an application

Improvements

- If jobs are very short adjust the timeouts to be shorter
- For jobs that you know are large start with a higher number of initial executors to avoid slow spin up
- If you are sharing a cluster, setting max executors can prevent you from hogging it

Run it!

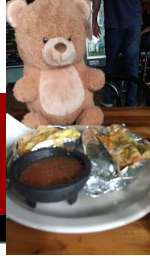
Oh no! It failed :(How could we adjust it?

Suppose that in the driver log, we see a “container lost exception” and on the executor logs we see:

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

This points to an out of memory error on the executors

Addressing Executor OOM



- If we have more executor memory to give it, try that!
- Lets try increasing the number of partitions so that each executor will process smaller pieces of the data at once
  - Spark.default.parallelism = 10
  - Or by adding the number of partitions to the code e.g. reduceByKey(numPartitions = 10)

- Many more things you can do to improve the code

Low Cluster Utilization: Idle Executors

Executors

▸ Show Additional Metrics

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Blacklisted
Active(80)	0	0.0 B / 487.1 GB	0.0 B	316	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	0
Total(80)	0	0.0 B / 487.1 GB	0.0 B	316	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	0

Executors

Show 20 ▾ entries Search:

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump
driver	10.138.0.5:58041	Active	0	0.0 B / 8.4 GB	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B		Thread Dump
1	holden-magic-sw-nwwj.c.boos-demo-projects-are-rad.internal:60507	Active	0	0.0 B / 6.1 GB	0.0 B	4	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
2	holden-magic-sw-4c5h.c.boos-demo-projects-are-rad.internal:52355	Active	0	0.0 B / 6.1 GB	0.0 B	4	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
3	holden-magic-w-14.c.boos-demo-projects-are-rad.internal:34594	Active	0	0.0 B / 6.1 GB	0.0 B	4	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump

What to do about it?

- If we see idle executors but the total size of our job is small, we may just be requesting too many executors
- If all executors are idle it maybe because we are doing a large computation in the driver
- If the computation is very large, and we see idle executors, this maybe because the executors are waiting for a “large” task → so we can increase partitions
  - At some point adding partitions will slow the job down
  - But only if not too much skew

Shuffle Spill to Disk in the Web UI

Details for Stage 9

Total task time across all tasks: 11 min
Output: 957.7 MB / 3804129
Shuffle read: 543.3 MB / 3804129
Shuffle spill (memory): 23.4 GB
Shuffle spill (disk): 634.3 MB

▸ **Show additional metrics**

Summary Metrics for 6 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	1.6 min	1.6 min	2.1 min	2.1 min	2.2 min
GC Time	26 s	26 s	37 s	37 s	37 s
Output Size / Records	158.1 MB / 626695	158.3 MB / 626839	160.5 MB / 639536	161.3 MB / 641730	161.3 MB / 641816
Shuffle Read Size / Records	89.9 MB / 626695	90.0 MB / 626839	90.6 MB / 639536	91.2 MB / 641730	91.3 MB / 641816
Shuffle spill (memory)	3.5 GB	3.8 GB	3.9 GB	4.1 GB	4.1 GB
Shuffle spill (disk)	104.5 MB	104.8 MB	105.6 MB	106.0 MB	107.8 MB

Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Output Size / Records	Shuffle Read Size / Records	Shuffle Spill (Memory)	Shuffle Spill (Disk)
0	hadoopvm2 :45179	6.4 min	3	0	3	483.1 MB / 1923082	273.2 MB / 1923082	12.1 GB	319.0 MB
1	hadoopvm3 ... :50249	5.0 min	3	0	3	474.6 MB / 1881047	270.1 MB / 1881047	11.3 GB	315.3 MB

Preventing Shuffle Spill to Disk

- Larger executors
- Configure off heap storage
- More partitions can help (ideally the labor of all the partitions on one executor can “fit” in that executor’s memory)
- We can adjust shuffle settings
  - Increase shuffle memory fraction (spark.shuffle.memory.fraction)
  - Try increasing:
    - spark.shuffle.file.buffer
  - Configure an external shuffle service, so that the shuffle files will not need to be stored in the spark executors
    - spark.shuffle.io.serverThreads
    - spark.shuffle.io.backLog

Signs of Too Many Partitions

Number of partitions is the size of the data each core is computing … smaller pieces are easier to process only up to a point

- Spark needs to keep metadata about each partition on the driver
- Driver memory errors & Driver overhead errors
- Very long task “spin up” time
  - Too many partitions at read usually caused by small part files
- Lots of pending tasks & Low memory utilization
- Long file write time for relatively small I/O “size” (especially with blockstores)

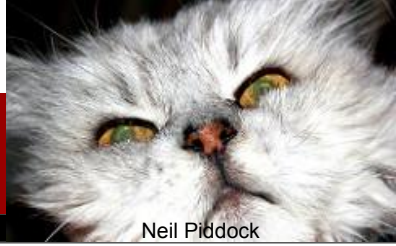
PYTHON SETTINGS

- Application memory overhead
  - We can tune this based on if an app is PySpark or not
  - Infact in the proposed PySpark on K8s PR this is done for us
  - More tuning may still be required
- Buffers & batch sizes oh my
  - spark.sql.execution.arrow.maxRecordsPerBatch
  - spark.python.worker.memory - default to 512 but default mem for Python can be lower :(
    - Set based on amount memory assigned to Python to reduce OOMs
  - Normal: automatic, sometimes set wrong - code change required :(

Tuning Can Help With

- Low cluster utilization ($$$$)
- Out of memory errors
- Spill to Disk / Slow shuffles
- GC errors / High GC overhead
- Driver / Executor overhead exceptions
- Reliable deployment

Things you can't "tune away"



- Silly Shuffles
  - You can make each shuffle faster through tuning but you cannot change the fundamental size or number of shuffles without adjusting code
  - Think about how much you are moving data, and how to do it more efficiently
  - Be very careful with column based operations on Spark SQL
- Unbalanced shuffles (caused by unbalanced keys)
  - if one or two tasks are much larger than the others
  - #Partitions is bounded by #distinct keys
- Bad Object management/ Data structures choices
  - Can lead to memory exceptions / memory overhead exceptions / gc errors
- Serialization exceptions*
  - *Exceptions due to cost of serialization can be tuned
- Python
  - This makes Holden sad, buuuut. Arrow?

But enough doom, lets watch software fail.

But what if we run a billion Spark jobs per day?

Tuning Could Depend on 4 Factors:

1. The execution environment
2. The size of the input data
3. The kind of computation (ML, python, streaming)

---

4. The historical runs of that job

We can get all this information programmatically!

Execution Environment

What we need to know about where the job will run

- How much memory do I have available to me: on a single node/ on the cluster
  - In my queue
- How much CPU: on a single node, on the cluster → corresponds to total number of concurrent tasks
- We can get all this information from the yarn api
  (https://dzone.com/articles/how-to-use-the-yarn-api-to-determine-resources-ava-1)
- Can I configure dynamic allocation?
- Cluster Health

Size of the input data

- How big is the input data on Disk
- How big will it be in memory?
  - “Coefficient of In Memory Expansion: shuffle Spill Memory / shuffle Spill disk
- <- can get historically or guess
- How many part files? (the default number of partitions at read)
- Cardinality and type?

Setting # partitions on the first try

$$\frac{\text{Size of input data in memory}}{\text{Amount of memory available per task on the executors}}$$

Assuming you have many distinct keys, you want to try to make partitions small enough that each partition fits in the memory “available to each task” to avoid spilling to disk or failing (the “Sandy Ryza formula”)

https://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/

How much memory should each task use?

```
def availableTaskMemoryMB(executorMemory : Long): Double = {
  val memFraction = sparkConf.getDouble("spark.memory.fraction", 0.6)
  val storageFraction = sparkConf.getDouble("spark.memory.storageFraction", 0.5)
  val nonStorage = 1-storageFraction
  val cores = sparkConf.getInt("spark.executor.cores", 1)
  Math.ceil((executorMemory * memFraction * nonStorage )/ cores)
}
```

Partitions *could* be inputDataSize/ memory per task

```
def determinePartitionsFromInputDataSize(inputDataSize: Double) : Int = {

 Math.round(inputDataSize/availableTaskMemoryMB()).toInt

}
```

Historical Data

What kinds of historical data could we use?

- Historical cluster info
  - (was one node sad? How many executors were used?)
- Stage Metrics
  - duration (only on an isolated cluster)
  - executor CPU
  - vcores/Second
- Task Metrics
  - Shuffle read & shuffle write → how much data is being moved & how big is input data
  - “Total task time” = time for each task * number of tasks
  - Shuffle read spill memory / shuffle spill disk = input data expansion in memory
- What kind of computation was run (python/streaming)
- Who was running the computation

How do we save the metrics?

- For a human: Spark history server can be persisted even if the cluster is terminated
- For a program: Can use listeners to save everything from web UI
  - (See Spark Measure and read these at the start of job)
- Can also get significantly more by capturing the full event stream
- We can save information at task/ stage/ job level.

Meet Robin Sparkles!!!!!

- Saves historical data using listeners from Spark Measure
  - (https://github.com/LucaCanali/sparkMeasure)
- Create directory scheme to save task and stage data for successive runs
- Read in n previous runs before creating conf
- Use the result of the previous runs in tandem with current Spark Conf to Optimize some Spark Settings. (WIP)
- Note: not to be used in production, simply the start of a spell book.

Start a Spark listener and save metrics

Extend Spark Measure flight recorder which automatically saves metrics:

```
class RobinStageListener(sc: SparkContext, override val metricsFileName: String)
  extends ch.cern.sparkmeasure.FlightRecorderStageMetrics(sc.getConf) {
}
```

Start Spark Listener

```
  val myStageListener = new RobinStageListener(sc,
    stageMetricsPath(runNumber))

  sc.addSparkListener(myStageListener)
```

Add listener to program code

```
def run(sc: SparkContext, id: Int, metricsDir: String,
        inputFile: String, outputFile: String): Unit = {

    val metricsCollector = new MetricsCollector(sc, metricsDir)
    metricsCollector.startSparkJobWithRecording(id)
    //some app code
}
```

Read in the metrics

```
val STAGE_METRICS_SUBDIR = "stage_metrics"

val metricsDir = s"$metricsRootDir/${appName}"

val stageMetricsDir = s"$metricsDir/$STAGE_METRICS_SUBDIR"

def stageMetricsPath(n: Int): String = s"$metricsDir/run=$n"
```

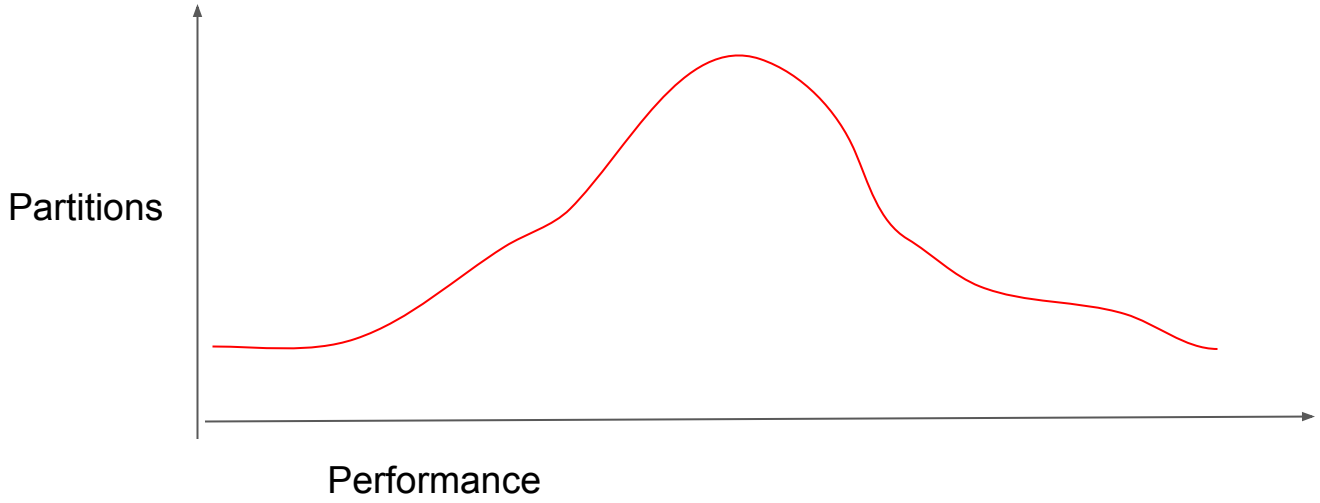
```
def readStageInfo(n : Int) =
ch.cern.sparkmeasure.Utils.readSerializedStageMetrics(stageMetricsPath(n))
```

To use: read in metrics, then create optimized conf

```
val conf = new SparkConf() ….

val (newConf: SparkConf, id: Int) = Runner.getOptimizedConf(metricsDir, conf)

val sc = SparkContext.getOrCreate(newConf)

Runner.run(sc, id, metricsDir, inputFile, outputFile)
```

Put it all together!

A programmatic way of defining the number of partitions …



Parallelism / number of partitions

Keep increasing the number of partitions until the metric we care about stops improving

- Spark.default.parallelism is only default if not in code
- Different stages could / should have different partitions
- can also compute for each stage and then set in shuffle phase:
  - rdd.reduceByKey(numPartitions = x)
  - You can design your job to get number of partitions from a variable in the conf
- Which stage to tune if we can only do one:
  - We can search for the longest stage
  - If we are tuning for parallelism, we might want to capture the stage “with the biggest shuffle”
  - We use “total shuffle write” which is the number of bytes written in shuffle files during the stage (this should be setting agnostic)

What metric defines better?

- Depends on your business … often a trade off between speed and efficiency
  - If you just want speed Holden has several cloud solutions to sell you (or would if she knew anyone in the sales department).
- cluster is “fully utilized”
  - Stage duration * executors = sum(task time)/ executors) + fuge factor
  - WARNING: Stage duration is dependent on instance & exec spin up, network delays excetera
- Stage is “Fastest”-> Executor Cpu Time (total cpu time for all the tasks)
- Could use anything that Spark measures:
  - Executor Serialization time
  - Deserialization time
  - Executor Idle time

Compute the number of partitions given a list of web UI input for each stage

```
def fromStageMetricSharedCluster(previousRuns: List[StageInfo]): Int = {

 previousRuns match {
   case Nil =>
//If this is the first run and parallelism is not provided, use the number of concurrent tasks
//We could also look at the file on disk
        possibleConcurrentTasks()

   case first :: Nil =>
     val fromInputSize = determinePartitionsFromInputDataSize(first.totalInputSize)
     Math.max(first.numPartitionsUsed + math.max(first.numExecutors,1), fromInputSize)

}
```

If we have several runs of the same computation

```
  case _  =>  val first = previousRuns(previousRuns.length - 2)

   val second = previousRuns(previousRuns.length - 1)

if(morePartitionsIsBetter(first,second)) {   //Increase the number of partitions from everything that we have tried
    Seq(first.numPartitionsUsed, second.numPartitionsUsed).max + second.numExecutors

 } else{ //If we overshot the number of partitions, use whichever run had the best executor cpu time
      previousRuns.sortBy(_.executorCPUTime).head.numPartitionsUsed
  } }
}
```

How to use Robin-Sparkles

- Assume user constructs the Spark Context
- Configure listener to write to a persistent location
  - Maybe need to be in an external location if you are using a web based system
- Modify the application code that creates the Spark context to use Robin Sparkles to set the settings

Note: not applicable for use where Spark context is created for you except for tuning number of partitions or variables that can change at the job (rather than application level)

She could do so much more!

- Additional settings to tune
  - Set executor memory and driver memory based on system
  - Executor health and system monitoring
  - Could surface warnings about data skew
  - Shuffle settings could be adjusted
- Currently we don’t get info if the stage fails
  - Our reader would actually fail if stage data is malformed, we should read partial stage
  - We could use the firehost listener that records every event from the event stream, to monitor failures
  - These other listeners would contain additional information that is not in the web ui

But first a nap

Other Tools

Sometimes the right answer isn’t tuning, it’s telling the user to change the code (see Sparklens) or telling the administrator to look at their cluster

Sparklens - Tells us what to tune or refactor

- AppTimelineAnalyzer
- EfficiencyStatisticsAnalyzer
- ExecutorTimelineAnalyzer
- ExecutorWallclockAnalyzer
- HostTimelineAnalyzer
- JobOverlapAnalyzer
- SimpleAppAnalyzer
- StageOverlapAnalyzer
- StageSkewAnalyzer

Dr Elephant

- Spark ConfigurationHeurestic
  - val SPARK_DRIVER_MEMORY_KEY = "spark.driver.memory"
  - val SPARK_EXECUTOR_MEMORY_KEY = "spark.executor.memory"
  - val SPARK_EXECUTOR_INSTANCES_KEY = "spark.executor.instances"
  - val SPARK_EXECUTOR_CORES_KEY = "spark.executor.cores"
  - val SPARK_SERIALIZER_KEY = "spark.serializer"
  - val SPARK_APPLICATION_DURATION = "spark.application.duration"
  - val SPARK_SHUFFLE_SERVICE_ENABLED = "spark.shuffle.service.enabled"
  - val SPARK_DYNAMIC_ALLOCATION_ENABLED = "spark.dynamicAllocation.enabled"
  - val SPARK_DRIVER_CORES_KEY = "spark.driver.cores"
  - val SPARK_DYNAMIC_ALLOCATION_MIN_EXECUTORS = "spark.dynamicAllocation.minExecutors"
  - etc.
- Among other things

Other monitoring/linting/tuning tools

- Kamon
- Sparklint
- Sparkoscope
- Graphite + Spark

Some upcoming talks

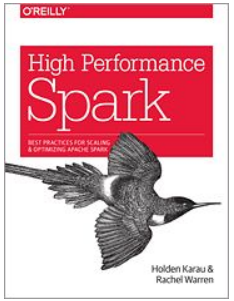
- May
  - JOnTheBeach (Spain) Friday - General Purpose Big Data Systems are eating the world - Is Tool Consolidation inevitable?
- June
  - Spark Summit SF (Accelerating Tensorflow & Accelerating Python + Dependencies)
  - Scala Days NYC
  - FOSS Back Stage & BBuzz
- July
  - Curry On Amsterdam
  - OSCON Portland
- August
  - JupyterCon (NYC)

High Performance Spark!

You can buy it today! Hopefully you got it signed earlier today if not … buy it and come see us again!

The settings didn't get its own chapter is in the appendix, (doing things on time is hard)

Cats love it*



*Or at least the box it comes in. If buying for a cat, get print rather than e-book.

Thanks and keep in touch