SPARK SUMMIT

Apache Spark Performance Troubleshooting at Scale: Challenges, Tools and Methods

Luca Canali, CERN

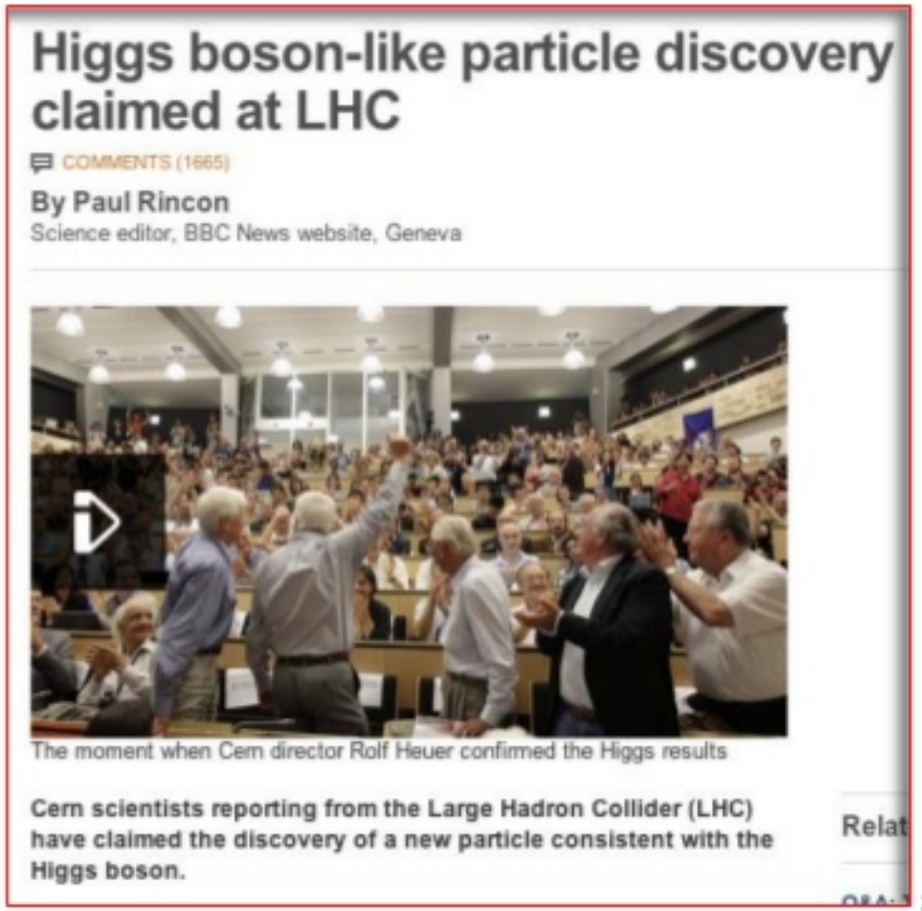
#EUdev2

About Luca

- Computing engineer and team lead at CERN IT
  - Hadoop and Spark service, database services
  - Joined CERN in 2005
- 17+ years of experience with database services
  - Performance, architecture, tools, internals
  - Sharing information: blog, notes, code
-  @LucaCanaliDB – http://cern.ch/canali

CERN and the Large Hadron Collider

- Largest and most powerful particle accelerator



Higgs boson-like particle discovery claimed at LHC

COMMENTS (1665)

By Paul Rincon

Science editor, BBC News website, Geneva

The moment when Cern director Rolf Heuer confirmed the Higgs results

Cern scientists reporting from the Large Hadron Collider (LHC) have claimed the discovery of a new particle consistent with the Higgs boson.

Relat

Apache Spark @ 

- Spark is a popular component for data processing
  - Deployed on four production Hadoop/YARN clusters
    - Aggregated capacity (2017): ~1500 physical cores, 11 PB
  - Adoption is growing. Key projects involving Spark:
    - Analytics for accelerator controls and logging
    - Monitoring use cases, this includes use of Spark streaming
    - Analytics on aggregated logs
    - Explorations on the use of Spark for high energy physics

Link: http://cern.ch/canali/docs/BigData_Solutions_at_CERN_KT_Forum_20170929.pdf

Motivations for This Work

- Understanding Spark workloads
  - Understanding technology (where are the bottlenecks, how much do Spark jobs scale, etc?)
  - Capacity planning: benchmark platforms
- Provide our users with a range of monitoring tools
- Measurements and troubleshooting Spark SQL
  - Structured data in Parquet for data analytics
  - Spark-ROOT (project on using Spark for physics data)

Outlook of This Talk

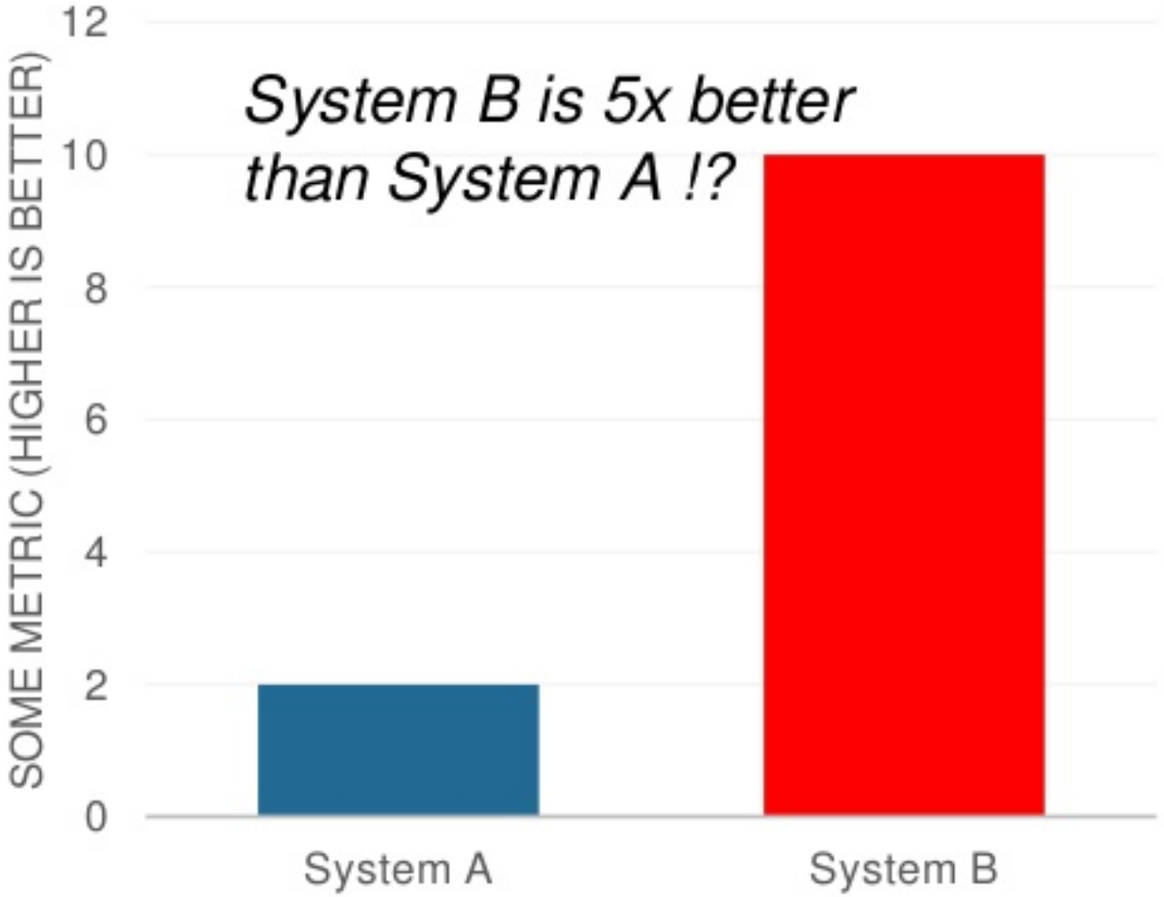
- Topic is vast, I will just share some ideas and lessons learned
- How to approach performance troubleshooting, benchmarking and relevant methods
- Data sources and tools to measure Spark workloads, challenges at scale
- Examples and lessons learned with some key tools

Challenges

- Just measuring performance metrics is easy
- Producing actionable insights requires effort and preparation
  - Methods on how to approach troubleshooting performance
  - How to gather relevant data
    - Need to use the right tools, possibly many tools
    - Be aware of the limitations of your tools
  - Know your product internals: there are many “moving parts”
  - Model and understand root causes from effects

Anti-Pattern: The Marketing Benchmark

- The over-simplified benchmark graph
- Does not tell you why B is better than A
- To understand, you need more context and root cause analysis



Benchmark for Speed

- Which one is faster?

- 20x
- 10x
- 1x

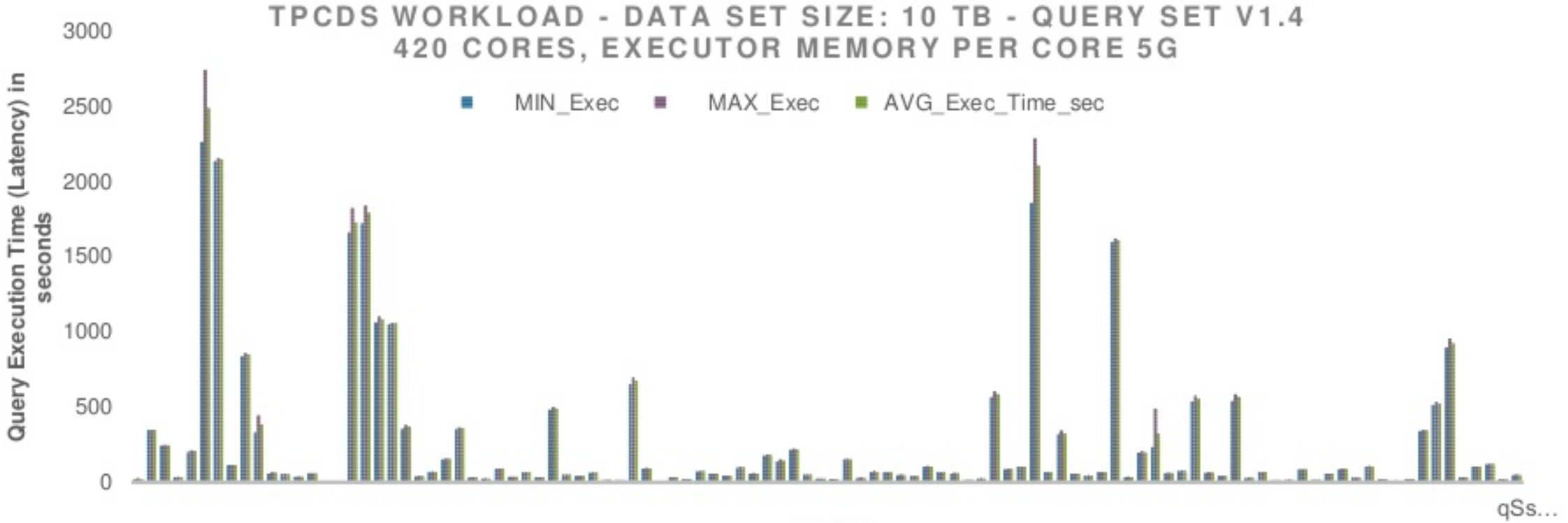
Adapt Answer to Circumstances

- Which one is faster?

- 20x
- 10x
- 1x
- Actually, it depends..

Active Benchmarking

- Example: use TPC-DS benchmark as workload generator
  - Understand and measure Spark SQL, optimizations, systems performance, etc



Troubleshooting by Understanding

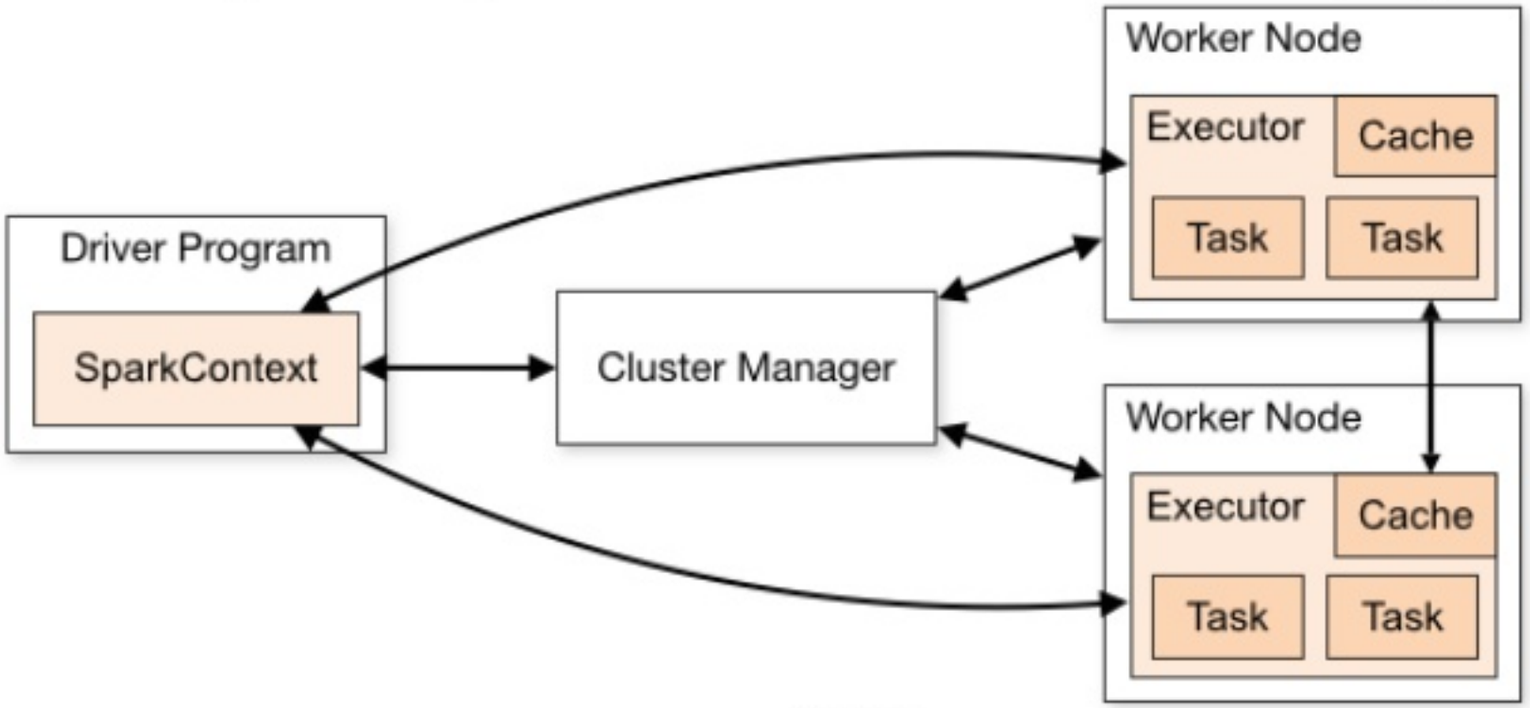
- Measure the workload
  - Use all relevant tools
  - Not a “black box”: instrument code where is needed
- Be aware of the blind spots
  - Missing tools, measurements hard to get, etc
- Make a mental model
  - Explain the observed performance and bottlenecks
  - Prove it or disprove it with experiment
- Summary:
  - Be data driven, no dogma, produce insights

Actionable Measurement Data

- You want to find answers to questions like
  - What is my workload doing?
  - Where is it spending time?
  - What are the bottlenecks (CPU, I/O)?
  - Why do I measure the {latency/throughput} that I measure?
    - Why not 10x better?

Measuring Spark

- Distributed system, parallel architecture
  - Many components, complexity increases when running at scale
  - Optimizing a component does not necessarily optimize the whole

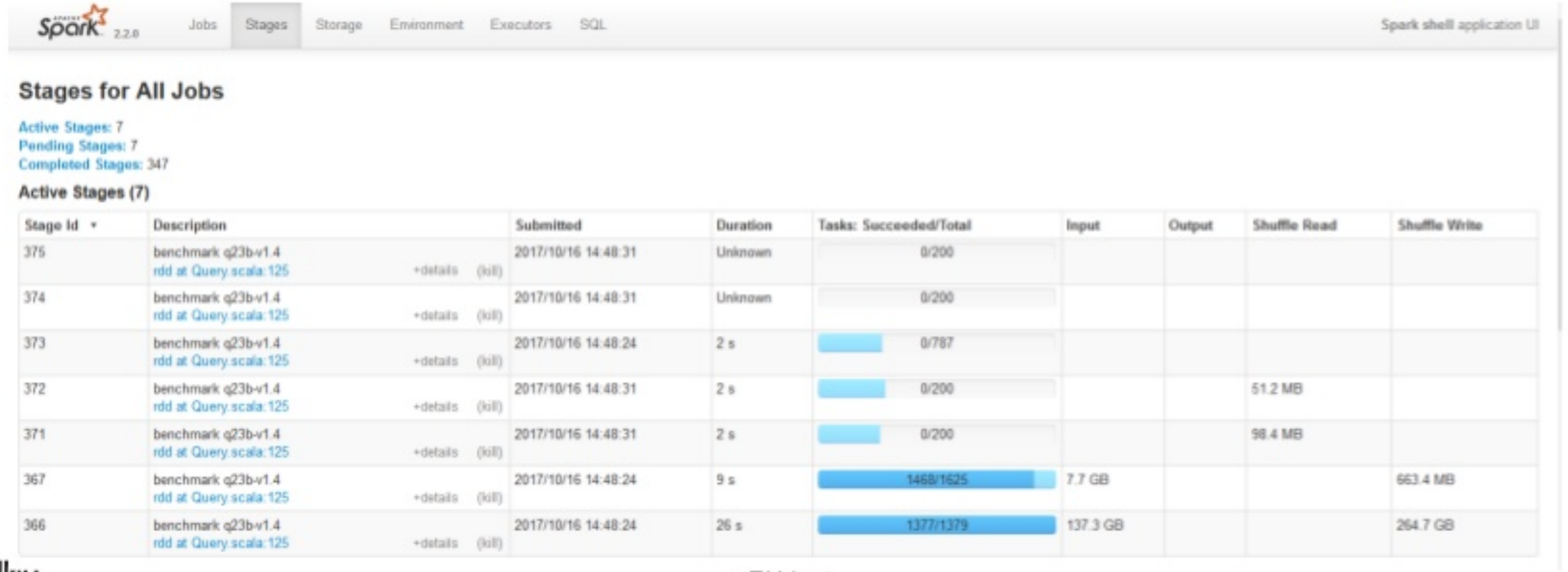


Spark and Monitoring Tools

- Spark instrumentation
  - Web UI
  - REST API
  - Eventlog
  - Executor/Task Metrics
  - Dropwizard metrics library
- Complement with
  - OS tools
  - For large clusters, deploy tools that ease working at cluster-level
- https://spark.apache.org/docs/latest/monitoring.html

Web UI

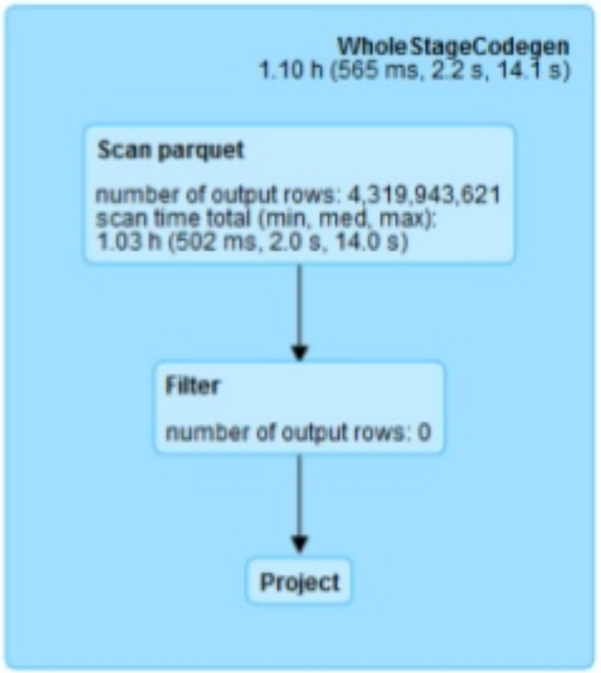
- Info on Jobs, Stages, Executors, Metrics, SQL,..
  - Start with: point web browser driver_host, port 4040

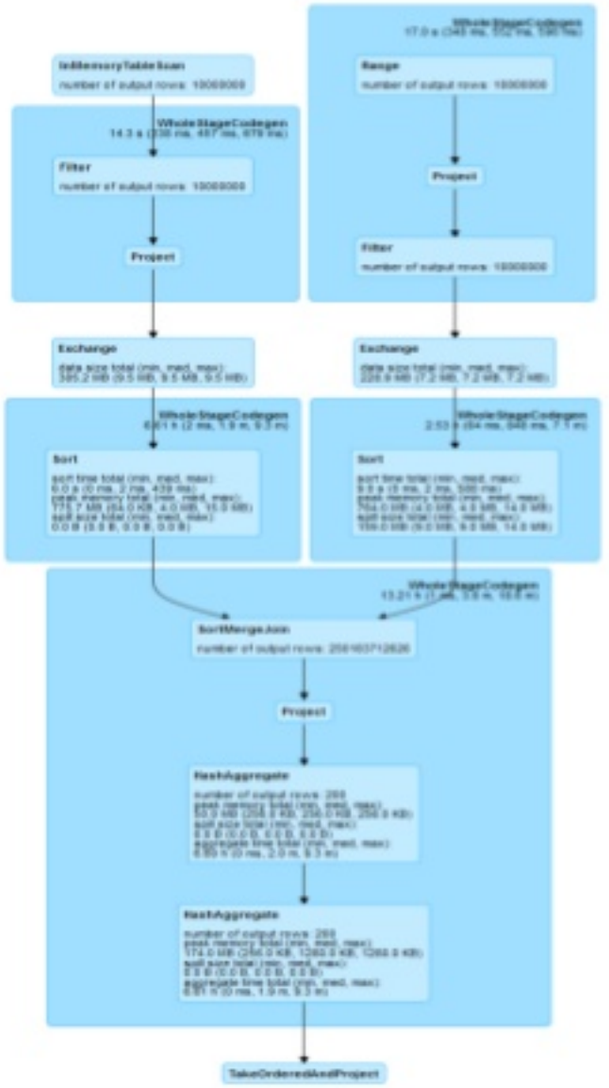


Execution Plans and DAGs

Details for Query 0

Submitted Time: 2017/06/06 11:19:49
Duration: 1.4 min
Succeeded Jobs: 3

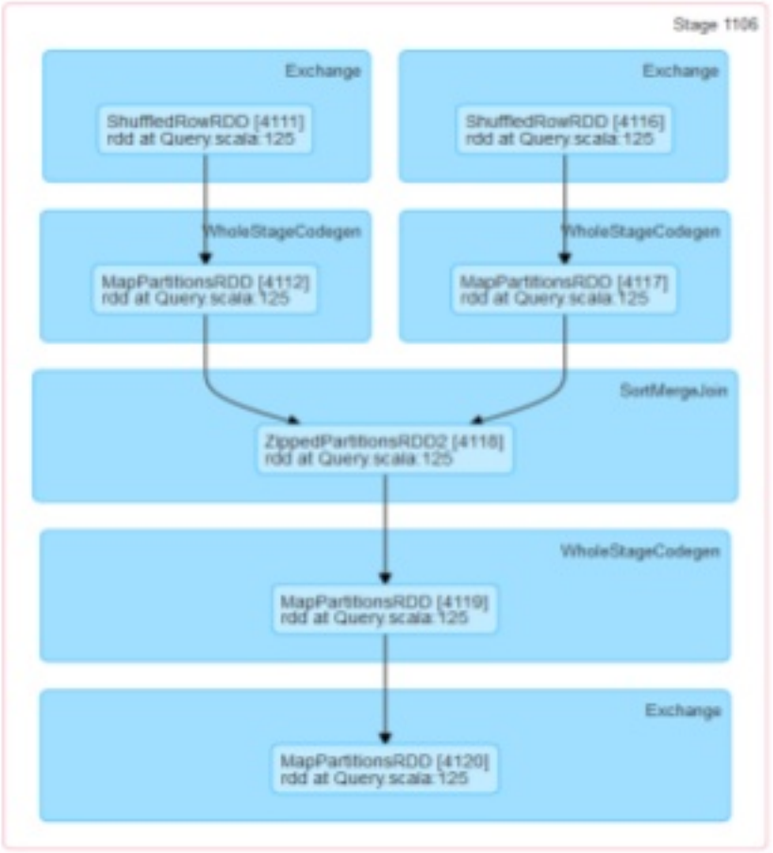




Details for Stage 1106 (Attempt 0)

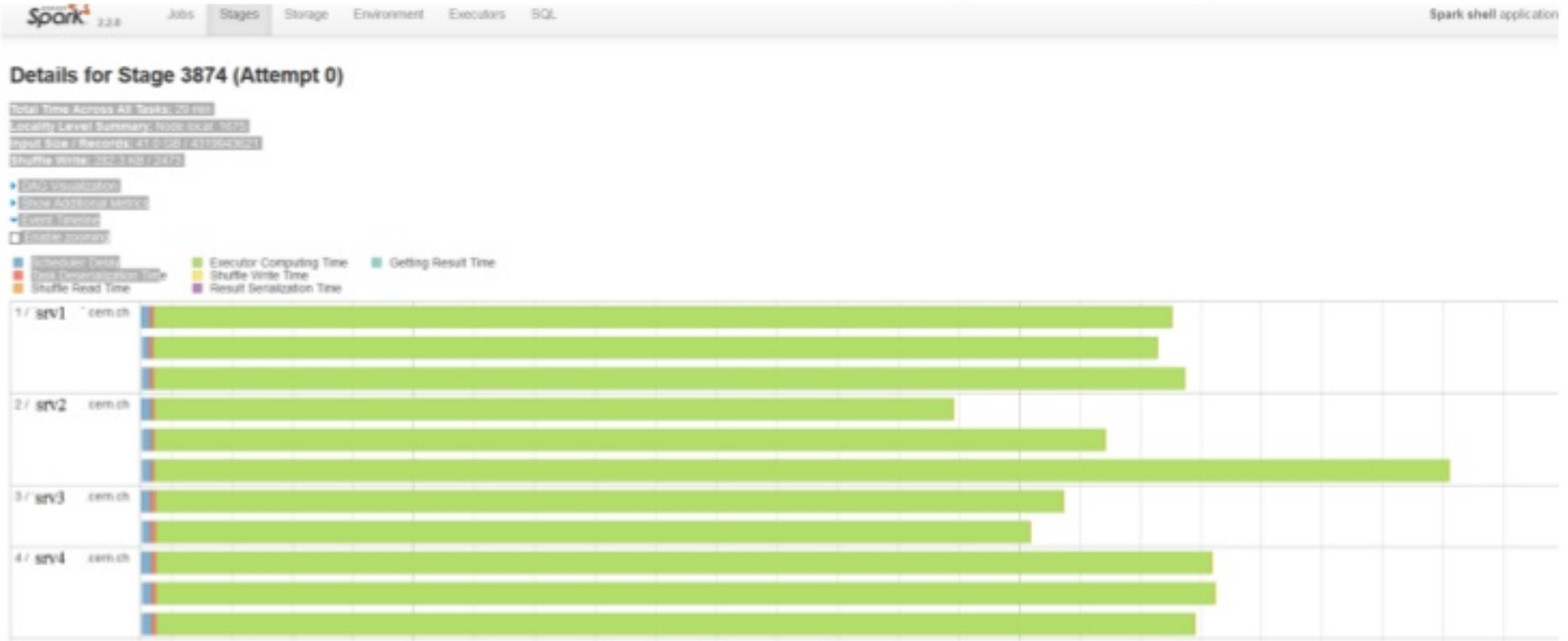
Total Time Across All Tasks: 1.2 h
Locality Level Summary: Process local: 200
Shuffle Read: 60.2 GB / 2357107158
Shuffle Write: 3.8 MB / 60480

▾ DAG Visualization



Web UI Event Timeline

- Event Timeline
  - show task execution details by activity and time



REST API – Spark Metrics

- History server URL + /api/v1/applications
- http://historyserver:18080/api/v1/applications/application_1507881680808_0002/stages

EventLog – Stores Web UI History

- Config:
  - `spark.eventLog.enabled=true`
  - `spark.eventLog.dir = <path>`

- JSON files store info displayed by Spark History server
  - You can read the JSON files with Spark task metrics and history with custom applications. For example sparklint.
  - You can read and analyze event log files using the Dataframe API with the Spark SQL JSON reader. More details at: https://github.com/LucaCanali/Miscellaneous/tree/master/Spark_Notes

Spark Executor Task Metrics

```
val df = spark.read.json("/user/spark/applicationHistory/application_...")
df.filter("Event='SparkListenerTaskEnd'").select("Task Metrics.*").printSchema
```

```
Task ID: long (nullable = true)
 |-- Disk Bytes Spilled: long (nullable = true)
 |-- Executor CPU Time: long (nullable = true)
 |-- Executor Deserialize CPU Time: long (nullable = true)
 |-- Executor Deserialize Time: long (nullable = true)
 |-- Executor Run Time: long (nullable = true)
 |-- Input Metrics: struct (nullable = true)
 |     |-- Bytes Read: long (nullable = true)
 |     |-- Records Read: long (nullable = true)
 |-- JVM GC Time: long (nullable = true)
 |-- Memory Bytes Spilled: long (nullable = true)
 |-- Output Metrics: struct (nullable = true)
 |     |-- Bytes Written: long (nullable = true)
 |     |-- Records Written: long (nullable = true)
 |-- Result Serialization Time: long (nullable = true)
 |-- Result Size: long (nullable = true)
 |-- Shuffle Read Metrics: struct (nullable = true)
 |     |-- Fetch Wait Time: long (nullable = true)
 |     |-- Local Blocks Fetched: long (nullable = true)
 |     |-- Local Bytes Read: long (nullable = true)
 |     |-- Remote Blocks Fetched: long (nullable = true)
 |     |-- Remote Bytes Read: long (nullable = true)
 |     |-- Total Records Read: long (nullable = true)
 |-- Shuffle Write Metrics: struct (nullable = true)
 |     |-- Shuffle Bytes Written: long (nullable = true)
 |     |-- Shuffle Records Written: long (nullable = true)
 |     |-- Shuffle Write Time: long (nullable = true)
 |-- Updated Blocks: array (nullable = true)
```

Spark Internal Task metrics:
Provide info on executors' activity:
Run time, CPU time used, I/O metrics, JVM Garbage Collection, Shuffle activity, etc.

Task Info, Accumulables, SQL Metrics

```
df.filter("Event='SparkListenerTaskEnd'").select("Task Info.*").printSchema
```

```
root
 |-- Accumulables: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- ID: long (nullable = true)
 |    |    |-- Name: string (nullable = true)
 |    |    |-- Value: string (nullable = true)
 |    |    | . . .
 |-- Attempt: long (nullable = true)
 |-- Executor ID: string (nullable = true)
 |-- Failed: boolean (nullable = true)
 |-- Finish Time: long (nullable = true)
 |-- Getting Result Time: long (nullable = true)
 |-- Host: string (nullable = true)
 |-- Index: long (nullable = true)
 |-- Killed: boolean (nullable = true)
 |-- Launch Time: long (nullable = true)
 |-- Locality: string (nullable = true)
 |-- Speculative: boolean (nullable = true)
 |-- Task ID: long (nullable = true)
```

Accumulables are used to keep accounting of metrics updates, including SQL metrics

Details about the Task: Launch Time, Finish Time, Host, Locality, etc

EventLog Analytics Using Spark SQL

Aggregate stage info metrics by name and display sum(values):

```
scala> spark.sql("select Name, sum(Value) as value from
aggregatedStageMetrics group by Name order by Name").show(40,false)
```

```
+----------------------------------------------------+---------------+
|Name                                                |value          |
+----------------------------------------------------+---------------+
|aggregate time total (min, med, max)                |1230038.0      |
|data size total (min, med, max)                     |5.6000205E7    |
|duration total (min, med, max)                      |3202872.0      |
|number of output rows                               |2.504759806E9  |
|internal.metrics.executorRunTime                    |857185.0       |
|internal.metrics.executorCpuTime                    |1.46231111372E11|
|...                                                 |...            |
```

Drill Down Into Executor Task Metrics

Relevant code in Apache Spark - Core

- Example snippets, show instrumentation in Executor.scala
- Note, for SQL metrics, see instrumentation with code-generation

```
// Run the actual task and measure its runtime.
taskStart = System.currentTimeMillis()
taskStartCpu = if (threadMXBean.isCurrentThreadCpuTimeSupported) {
  threadMXBean.getCurrentThreadCpuTime
} else 0L
```

```
task.metrics.setExecutorRunTime((taskFinish - taskStart) - task.executorDeserializeTime)
task.metrics.setExecutorCpuTime(
  (taskFinishCpu - taskStartCpu) - task.executorDeserializeCpuTime)
task.metrics.setJvmGCTime(computeTotalGcTime() - startGCTime)
task.metrics.setResultSerializationTime(afterSerialization - beforeSerialization)
```

Read Metrics with sparkMeasure

sparkMeasure is a tool for performance investigations of Apache Spark workloads https://github.com/LucaCanali/sparkMeasure

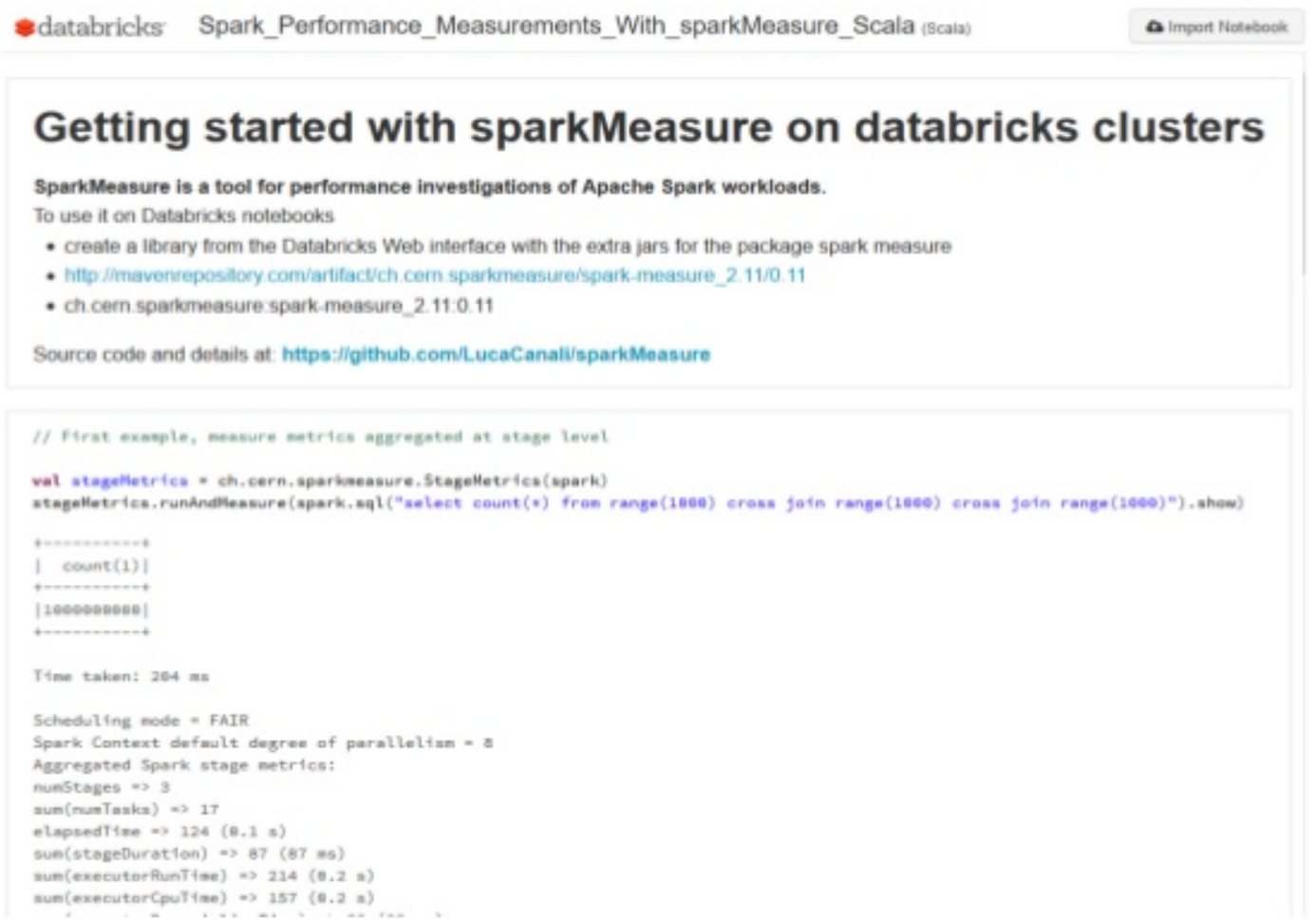
```
$ bin/spark-shell --packages ch.cern.sparkmeasure:spark-measure_2.11:0.11

scala> val stageMetrics = ch.cern.sparkmeasure.StageMetrics(spark)
scala> stageMetrics.runAndMeasure(spark.sql("select count(*) from
range(1000) cross join range(1000) cross join range(1000)").show)

Scheduling mode = FIFO
Spark Context default degree of parallelism = 8
Aggregated Spark stage metrics:
numStages => 3
sum(numTasks) => 17
elapsedTime => 9103 (9 s)
sum(stageDuration) => 9027 (9 s)
sum(executorRunTime) => 69238 (1.2 min)
sum(executorCpuTime) => 68004 (1.1 min)
. . . <more metrics>
```

Notebooks and sparkMeasure

- Interactive use: suitable for notebooks and REPL
- Offline use: save metrics for later analysis
- Metrics granularity: collected per stage or record all tasks
- Metrics aggregation: user-defined, e.g. per SQL statement
- Works with Scala and Python



Collecting Info Using Spark Listener

- Spark Listeners are used to send task metrics from executors to driver

- Underlying data transport used by WebUI, sparkMeasure, etc

- Spark Listeners for your custom monitoring code

```
// Proof-of-concept code of how to extend Spark listeners for custom monitoring of Spark metrics
// When using this from the spark-shell, use the REPL command :paste and copy-paste the following code
// Tested on Spark 2.1.0, March 2017

import org.apache.spark.scheduler._
import org.apache.log4j.LogManager
val logger = LogManager.getLogger("CustomListener")

class CustomListener extends SparkListener {
  override def onStageCompleted(stageCompleted: SparkListenerStageCompleted): Unit = {
    logger.warn(s"Stage completed, runTime: ${stageCompleted.stageInfo.taskMetrics.executorRunTime}, " +
                s"cpuTime: ${stageCompleted.stageInfo.taskMetrics.executorCpuTime}")
  }
}

val myListener=new CustomListener
//sc is the active Spark Context
sc.addSparkListener(myListener)

// run a simple Spark job and note the additional warning messages emitted by the CustomLister with
// Spark execution metrics, for exmaple run
spark.time(sql("select count(*) from range(1e4) cross join range(1e4)").show)
```

ProofOfConceptSparkCuctomListener.scala.txt hosted with ❤ by GitHub view raw

Examples – Parquet I/O

- An example of how to measure I/O, Spark reading Apache Parquet files
- This causes a full scan of the table store_sales

```
spark.sql("select * from store_sales where ss_sales_price=-1.0")
.collect()
```

- Test run on a cluster of 12 nodes, with 12 executors, 4 cores each
- Total Time Across All Tasks: **59 min**
- Locality Level Summary: Node local: 1675
- Input Size / Records: **185.3 GB** / 4319943621
- Duration: **1.3 min**

Parquet I/O – Filter Push Down

- Parquet filter push down in action
- This causes a full scan of the table store_sales with a filter condition pushed down

```
spark.sql("select * from store_sales where ss_quantity=-1.0")
.collect()
```

- Test run on a cluster of 12 nodes, with 12 executors, 4 cores each
- Total Time Across All Tasks: **1.0 min**
- Locality Level Summary: Node local: 1675
- Input Size / Records: **16.2 MB** / 0
- Duration: **3 s**

Parquet I/O – Drill Down

- Parquet filter push down
  - I/O reduction when Parquet pushed down a filter condition and using stats on data (min, max, num values, num nulls)
  - Filter push down not available for decimal data type (`ss sales price`)

```
scala> sql("select * from store_sales where ss_quantity=-1").explain
== Physical Plan ==
*Project [ss_sold_time_sk#0, ss_item_sk#1, ss_customer_sk#2, ss_cdemo_sk#3, ss_hdemo_sk#4, ss_addr_sk#5, ss_store_sk#6, ss_promo_sk#7, ss_ticket_number#8, ss_quantity#9,
ss_wholesale_cost#10, ss_list_price#11, ss_sales_price#12, ss_ext_discount_amt#13, ss_ext_sales_price#14, ss_ext_wholesale_cost#15, ss_ext_list_price#16, ss_ext_tax#17, ss_coupon_amt#18,
ss_net_paid#19, ss_net_paid_inc_tax#20, ss_net_profit#21, ss_sold_date_sk#22]
+- *Filter (isnotnull(ss_quantity#9) && (ss_quantity#9 = -1))
   +- *FileScan parquet
[ss_sold_time_sk#0,ss_item_sk#1,ss_customer_sk#2,ss_cdemo_sk#3,ss_hdemo_sk#4,ss_addr_sk#5,ss_store_sk#6,ss_promo_sk#7,ss_ticket_number#8,ss_quantity#9,ss_wholesale_cost#10,ss_list_price#1
1,ss_sales_price#12,ss_ext_discount_amt#13,ss_ext_sales_price#14,ss_ext_wholesale_cost#15,ss_ext_list_price#16,ss_ext_tax#17,ss_coupon_amt#18,ss_net_paid#19,ss_net_paid_inc_tax#20,ss_net_
profit#21,ss_sold_date_sk#22] Batched: true, Format: Parquet, Location: InMemoryFileIndex[hdfs://XXX.XXX.XXX/user/XXX/TPCDS/tpcds_1500/store_sales], PartitionCount: 1824,
PartitionFilters: [], PushedFilters: [IsNotNull(ss_quantity), EqualTo(ss_quantity,-1)], ReadSchema:
struct<ss_sold_time_sk:int,ss_item_sk:int,ss_customer_sk:int,ss_cdemo_sk:int,ss_hdemo_sk:int,ss_a...
```

Filter/predicate push down

https://db-blog.web.cern.ch/blog/luca-canali/2017-06-diving-spark-and-parquet-workloads-example

CPU and I/O Reading Parquet Files

```
# echo 3 > /proc/sys/vm/drop_caches # drop the filesystem cache
$ bin/spark-shell --master local[1] --packages ch.cern.sparkmeasure:spark-measure_2.11:0.11 --driver-memory 16g
val stageMetrics = ch.cern.sparkmeasure.StageMetrics(spark)
stageMetrics.runAndMeasure(spark.sql("select * from web_sales where ws_sales_price=-1").collect())

Spark Context default degree of parallelism = 1
Aggregated Spark stage metrics:
numStages => 1
sum(numTasks) => 787
elapsedTime => 465430 (7.8 min)
sum(stageDuration) => 465430 (7.8 min)
sum(executorRunTime) => 463966 (7.7 min)
sum(executorCpuTime) => 325077 (5.4 min)
sum(jvmGCTime) => 3220 (3 s)
```

CPU time is 70% of run time

Note: OS tools confirm that the difference “Run”- “CPU” time is spent in read calls (used a SystemTap script)

Stack Profiling and Flame Graphs

- Use stack profiling to investigate CPU usage
- Flame graph visualization to help identify “hot methods” and context (parent stack)
- Use profilers that don’t suffer from Java Safepoint bias, e.g. **async-profiler**

https://github.com/LucaCanali/Miscellaneous/blob/master/Spark_Notes/Tools_Spark_Linux_FlameGraph.md

How Does Your Workload Scale?

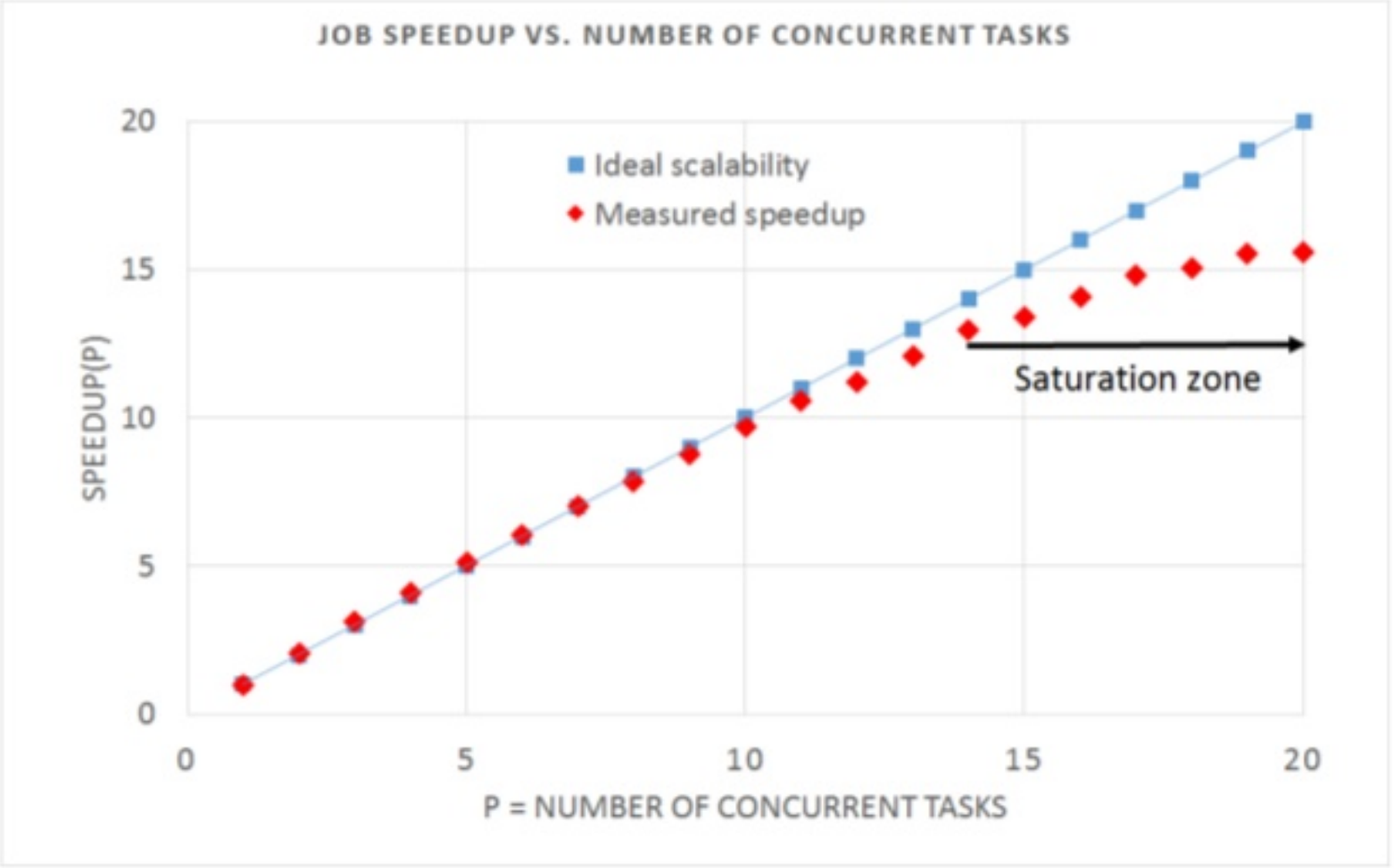
Measure latency as function of N# of concurrent tasks

Example workload: Spark reading Parquet files from memory

Speedup(p) = R(1)/R(p)

Speedup grows linearly in ideal case. Saturation effects and serialization reduce scalability

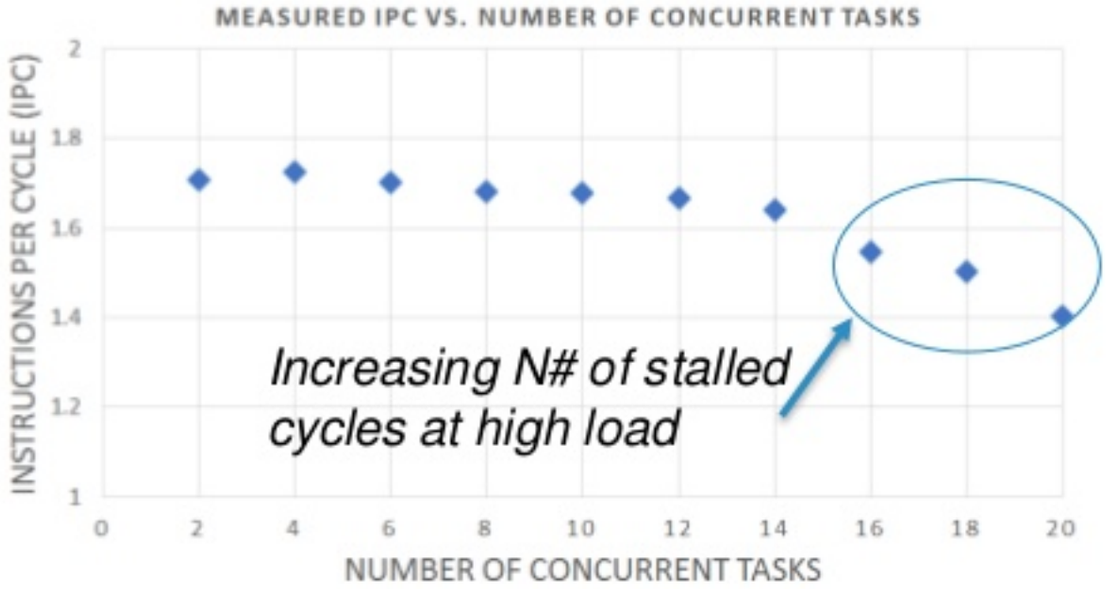
(see also Amdhal's law)

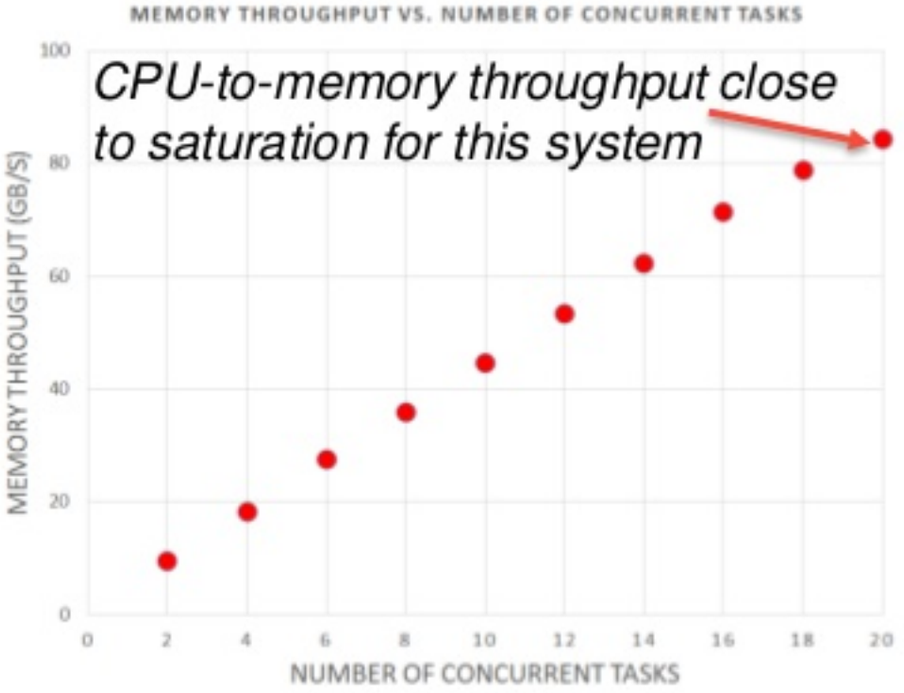


Are CPUs Processing Instructions or Stalling for Memory?

- Measure Instructions per Cycle (IPC) and CPU-to-Memory throughput
- Minimizing CPU stalled cycles is key on modern platforms
- Tools to read CPU HW counters: perf and more

https://github.com/LucaCanali/Miscellaneous/blob/master/Spark_Notes/Tools_Linux_Memory_Perf_Measure.md





Lessons Learned – Measuring CPU

- Reading Parquet data is CPU-intensive
  - Measured throughput for the test system at high load (using all 20 cores)
    - about 3 GB/s – max read throughput with lightweight processing of parquet files
  - Measured CPU-to-memory traffic at high load ~80 GB/s

- Comments:
  - CPU utilization and memory throughput are the bottleneck in this test
    - Other systems could have I/O or network bottlenecks at lower throughput
  - Room for optimizations in the Parquet reader code?

https://db-blog.web.cern.ch/blog/luca-canali/2017-09-performance-analysis-cpu-intensive-workload-apache-spark

Pitfalls: CPU Utilization at High Load

- Physical cores vs. threads
  - CPU utilization grows up to the number of available threads
  - Throughput at scale mostly limited by number of available cores
  - Pitfall: understanding Hyper-threading on multitenant systems

Example data: CPU-bound workload (reading Parquet files from memory)
Test system has 20 physical cores

Metric	20 concurrent tasks	40 concurrent tasks	60 concurrent tasks
Elapsed time	20 s	23 s	23 s
Executor run time	392 s	892 s	1354 s
Executor CPU Time	376 s	849 s	872 s
CPU-memory data volume	1.6 TB	2.2 TB	2.2 TB
CPU-memory throughput	85 GB/s	90 GB/s	90 GB/s
IPC	1.42	0.66	0.63

Job latency is roughly constant

Extra time from CPU runqueue wait

20 tasks -> each task gets a core
40 tasks -> they share CPU cores
It is as if CPU speed has become 2 times slower

Lessons Learned on Garbage Collection and CPU Usage

Measure: reading Parquet Table with "--driver-memory 1g" (default)

```
sum(executorRunTime) => 468480 (7.8 min)
sum(executorCpuTime) => 304396 (5.1 min)
sum(jvmGCTime) => 163641 (2.7 min)
```

Run Time = CPU Time (executor) + JVM GC

OS tools: (ps -efo cputime -p <pid_of_SparkSubmit>)

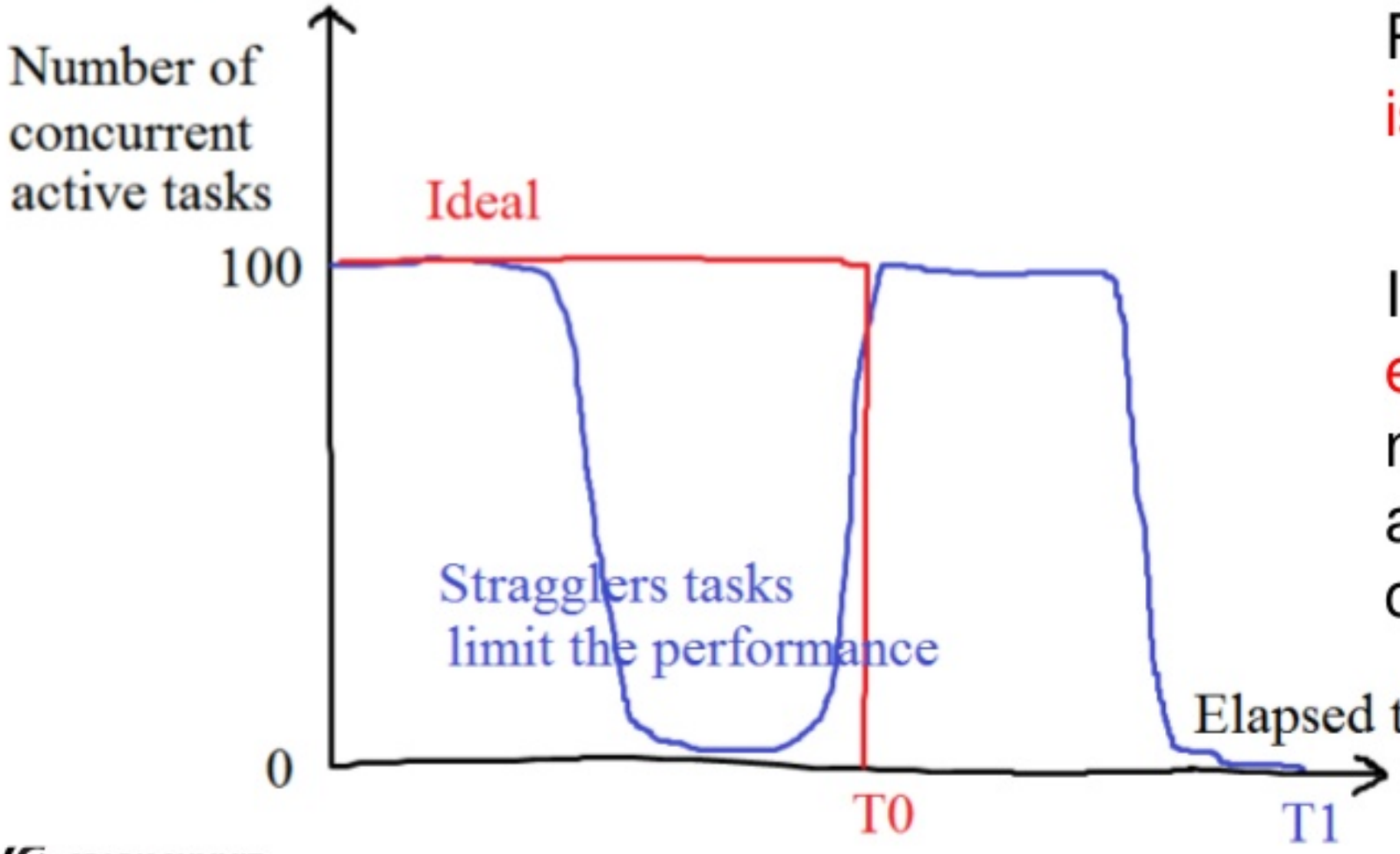
CPU time = 2306 sec

Many CPU cycles used by JVM, extra CPU time not accounted in Spark metrics due to GC

Lessons learned:

- Use OS tools to measure CPU used by JVM
- Garbage Collection is memory hungry (size your executors accordingly)

Performance at Scale: Keep Systems Resources Busy



Running tasks in parallel is key for performance

Important loss of efficiency when the number of concurrent active tasks << available cores

Issues With Stragglers

- Slow running tasks - stragglers
  - Many causes possible, including
  - Tasks running on slow/busy nodes
  - Nodes with HW problems
  - Skew in data and/or partitioning

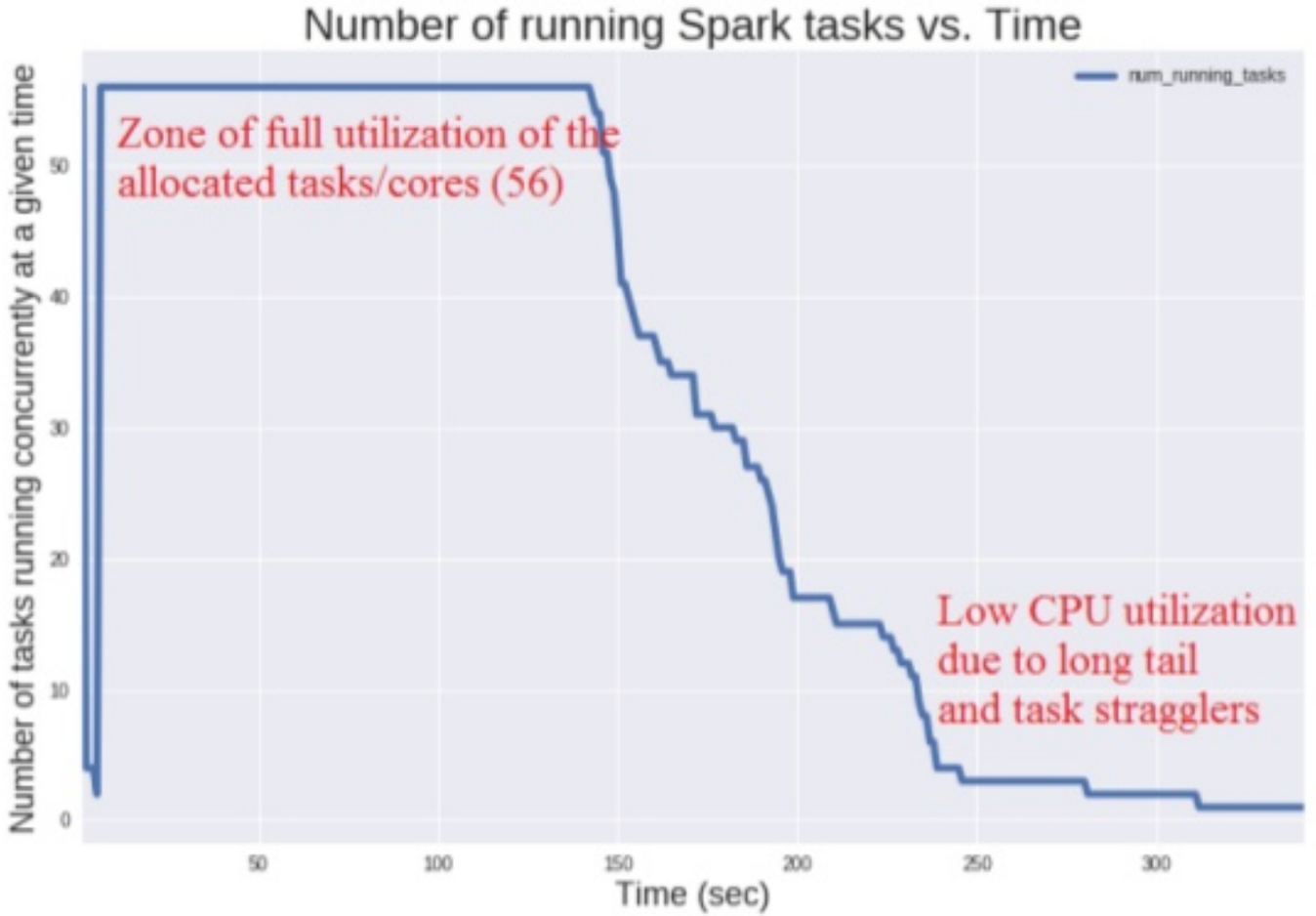
- A few "local" slow tasks can wreck havoc in global perf
  - It is often the case that one stage needs to finish before the next one can start
  - See also discussion in SPARK-2387 on stage barriers
  - Just a few slow tasks can slow everything down

Investigate Stragglers With Analytics on "Task Info" Data

Example of performance limited by long tail and stragglers

Data source: EventLog or sparkMeasure (from task info: task launch and finish time)

Data analyzed using Spark SQL and notebooks



From https://db-blog.web.cern.ch/blog/luca-canali/2017-03-measuring-apache-spark-workload-metrics-performance-troubleshooting

Task Stragglers – Drill Down

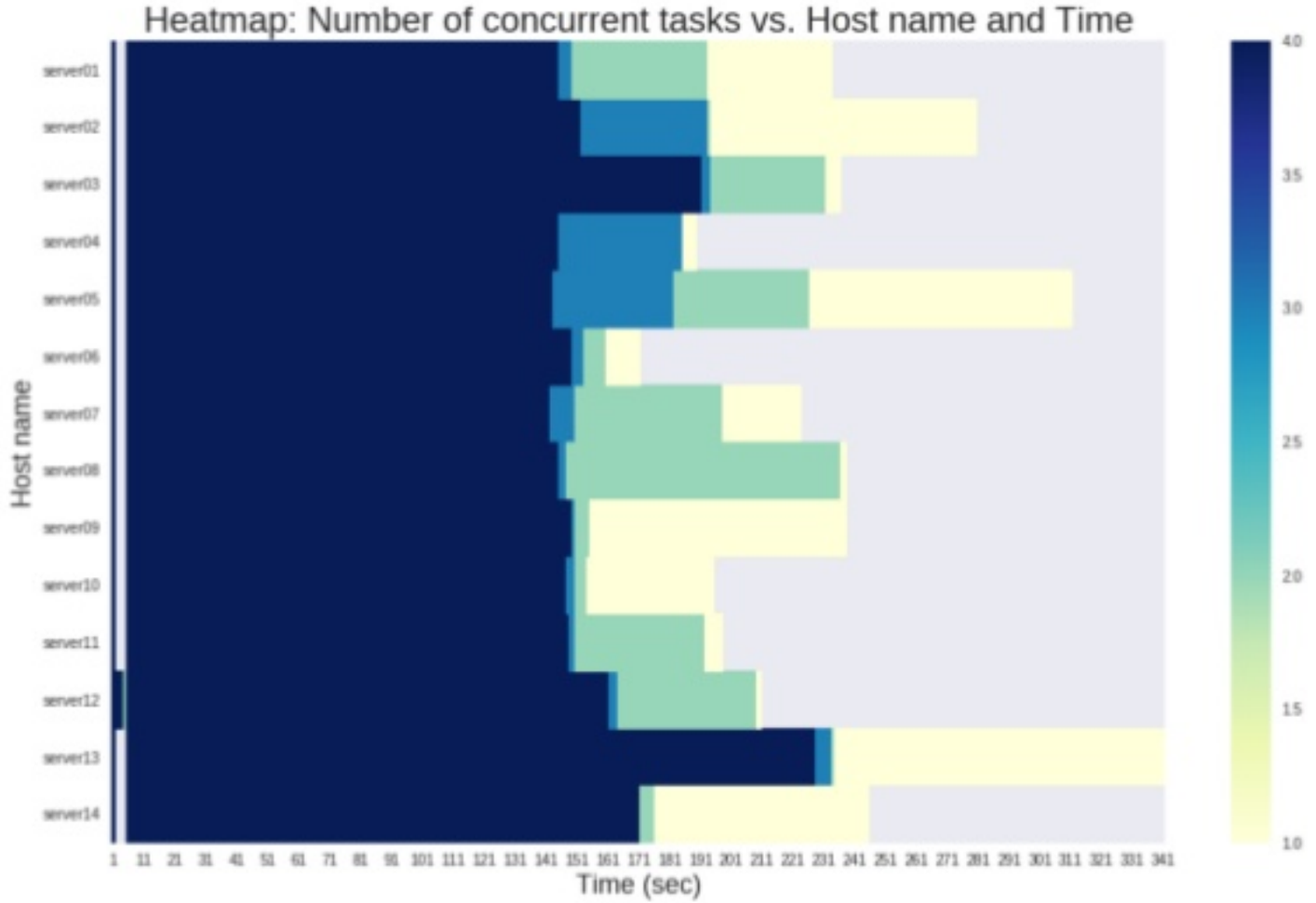
Drill down on task latency per executor:

it's a plot with 3 dimensions

Stragglers due to a few machines in the cluster:

later identified as slow HW

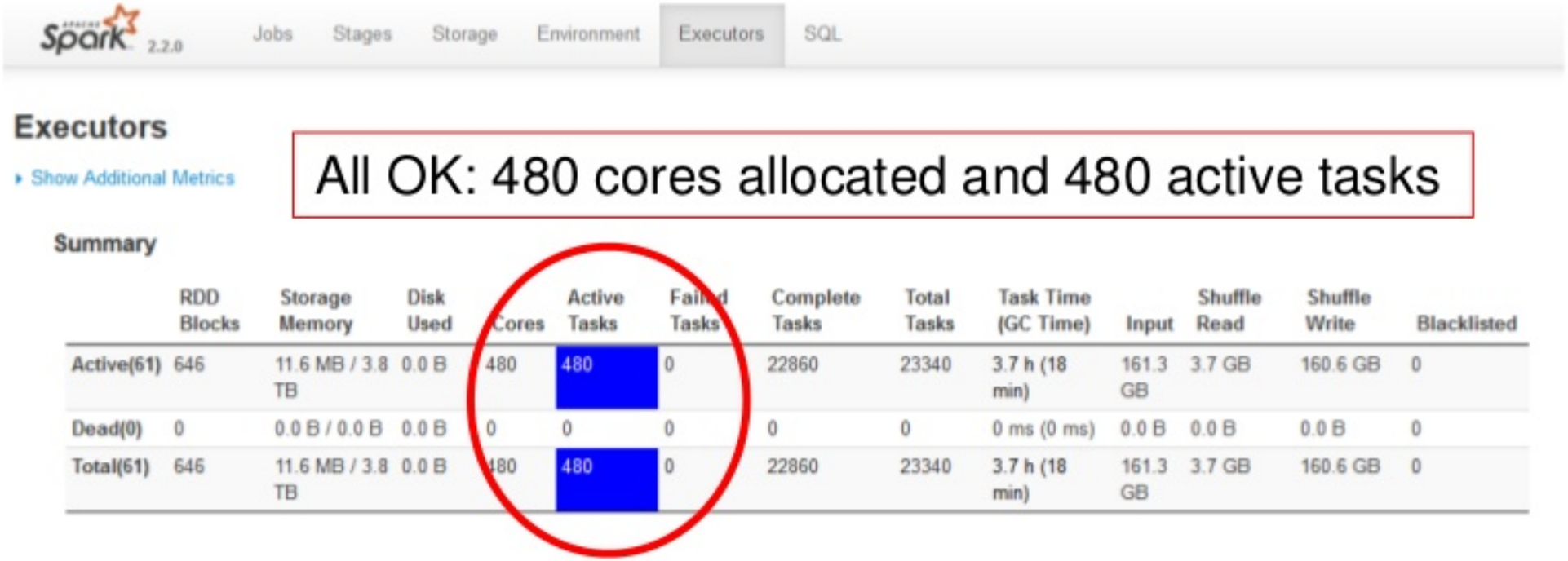
Lessons learned: identify and remove/repair non-performing hardware from the cluster



From https://github.com/LucaCanali/sparkMeasure/blob/master/examples/SparkTaskMetricsAnalysisExample.ipynb

Web UI – Monitor Executors

- The Web UI shows details of executors
  - Including number of active tasks (+ per-node info)



	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(61)	646	11.6 MB / 3.8 TB	0.0 B	480	480	0	22860	23340	3.7 h (18 min)	161.3 GB	3.7 GB	160.6 GB	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(61)	646	11.6 MB / 3.8 TB	0.0 B	480	480	0	22860	23340	3.7 h (18 min)	161.3 GB	3.7 GB	160.6 GB	0

Example of Underutilization

- Monitor active tasks with Web UI

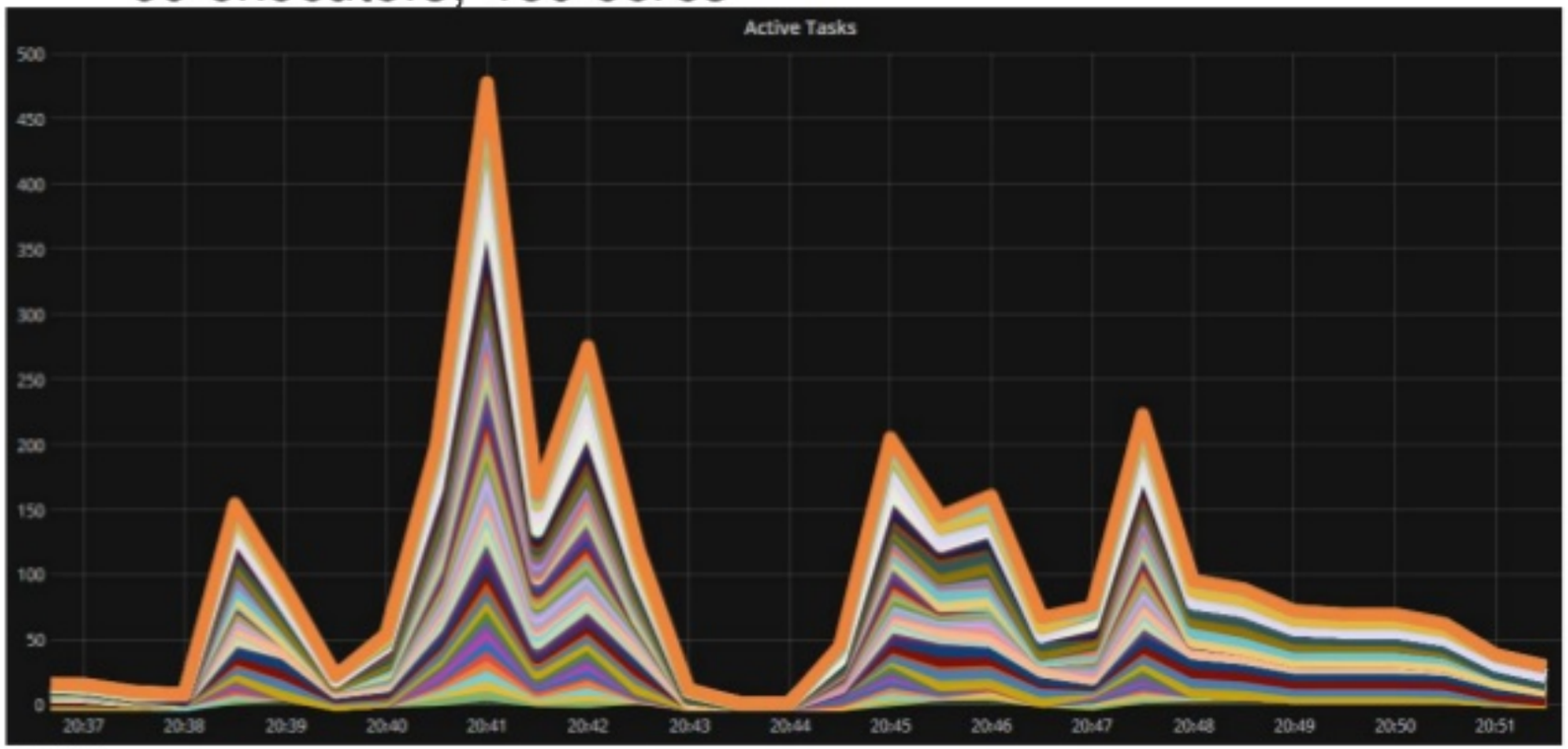
Utilization is low at this snapshot:
480 cores allocated and 48 active tasks

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(61)	582	403.6 MB / 3.8 TB	0.0 B	480	48	0	68950	68998	22.2 h (2.4 h)	893.9 GB	387.8 GB	242.8 GB	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(61)	582	403.6 MB / 3.8 TB	0.0 B	480	48	0	68950	68998	22.2 h (2.4 h)	893.9 GB	387.8 GB	242.8 GB	0

Visualize the Number of Active Tasks

- Plot as function of time to identify possible under-utilization
  - Grafana visualization of number of active tasks for a benchmark job running on 60 executors, 480 cores



Data source:
/executor/threadpool/
activeTasks
Transport: Dropwizard
metrics to Graphite sink

Measure the Number of Active Tasks With Dropwizard Metrics Library

- The Dropwizard metrics library is integrated with Spark
  - Provides configurable data sources and sinks. Details in doc and config file "metrics.properties"

    --conf spark.metrics.conf=metrics.properties
- Spark data sources:
  - Can be optional, as the JvmSource or "on by default", as the executor source
  - Notably the gauge: **/executor/threadpool/activeTasks**
  - Note: executor source also has info on I/O
- Architecture
  - Metrics are sent directly by each executor -> no need to pass via the driver.
  - More details: see source code "ExecutorSource.scala"

Limitations and Future Work

- Many important topics not covered here
  - Such as investigations and optimization of shuffle operations, SQL plans, etc
  - Understanding root causes of stragglers, long tails and issues related to efficient utilization of available cores/resources can be hard
- Current tools to measure Spark performance are very useful.. but:
  - Instrumentation does not yet provide a way to directly find bottlenecks
    - Identify where time is spent and critical resources for job latency
    - See Kay Ousterhout on "Re-Architecting Spark For Performance Understandability"
  - Currently difficult to link measurements of OS metrics and Spark metrics
    - Difficult to understand time spent for HDFS I/O (see HADOOP-11873)
  - Improvements on user-facing tools
    - Currently investigating linking Spark executor metrics sources and Dropwizard sink/Grafana visualization (see SPARK-22190)

Conclusions

- Think clearly about performance
  - Approach it as a problem in experimental science
  - Measure – build models – test – produce actionable results
- Know your tools
  - Experiment with the toolset – active benchmarking to understand how your application works – know the tools' limitations
- Measure, build tools and share results!
  - Spark performance is a field of great interest
  - Many gains to be made + a rapidly developing topic

Acknowledgements and References

- CERN
  - Members of Hadoop and Spark service and CERN+HEP users community
  - Special thanks to Zbigniew Baranowski, Prasanth Kothuri, Viktor Khristenko, Kacper Surdy
  - Many lessons learned over the years from the RDBMS community, notably www.oaktable.net
- Relevant links
  - Material by Brendan Gregg (www.brendangregg.com)
  - More info: links to blog and notes at http://cern.ch/canali