Carnegie Mellon University
Software Engineering Institute

Automated Cluster Testing and Optimization

Brad Powell
Sr. Security Engineer | CERT

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Document Markings

This material is based upon work funded and supported by the Department of Homeland Security under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center sponsored by the United States Department of Defense.

DM18-1377

Introduction

- How to setup an automated testing framework to get benchmarks and results that will help determine tuning parameters and improve the performance of your Spark cluster

Development and Test Environment (DTE)

- Support the architecture, design, and test processes of the lifecycle
- Provide a baseline of technologies for prototyping and testing capabilities supporting cybersecurity use cases
- Manage a shared and collaborative environment
- Evaluate relevant technology and conduct demonstrations as appropriate to inform engineering efforts and lessen risk
- Prototype data analysis techniques using the variety of available data types and tools
- Deliver Trend Reports to capture changes in the industry/community for relevant technology spaces

Automated Testing Tools

HiBench (Intel) - Measure speed, throughput, and system resource utilization

Supported releases:
Hadoop: Apache Hadoop 2.x, CDH5, HDP
Spark: Spark 1.6.x, Spark 2.0.x, Spark 2.1.x, Spark 2.2.x
Flink: 1.0.3
Storm: 1.0.1
Gearpump: 0.8.1
Kafka: 0.8.2.2

- Micro benchmark workloads:
  - Sort, WordCount, TeraSort, Sleep, Enhanced DFSIO
- SQL workloads:
  - Scan, Join, Aggregate
- Machine Learning workloads:
  - Bayesian Classification, K-means clustering, Logistic Regression, Alternating Least Squares, Gradient Boosting Trees, Linear Regression, Latent Dirichlet Allocation, Principal Components Analysis, Random Forest, Support Vector Machine, Singular Value Decomposition
- Websearch benchmark workloads:
  - PageRank, Nutch indexing
- Graph benchmark workloads:
  - NWeight
- Streaming workloads:
  - Identity, Repartition, Stateful Wordcount, Fixwindow

Sample Output:

Type	Date	Time	Input_data_size	Duration(s)	Throughput(bytes/s)	Throughput/node
ScalaSparkWordcount	7/19/2018	15:45:00	36790	29.818	1233	30
ScalaSparkWordcount	7/19/2018	16:02:00	36790	28.841	1275	31
ScalaSparkWordcount	7/19/2018	16:16:58	36790	27.225	1351	34
ScalaSparkTerasort	7/19/2018	16:27:26	3200000	36.693	87210	2180
HadoopWordcount	7/19/2018	17:08:34	36790	28.971	1269	31

SparkBench (IBM) – Benchmarking and simulating Spark jobs

- Spark-Submit-Config:
  - SparkBench converts config files into spark-submit scripts
  - Allows multiple spark-submits in series or parallel
- Workloads:
  - Standalone Spark jobs with input/output
  - Data Generators: Graph, Kmeans, Linear Regression
  - Kmeans, Logistic Regression, Sleep, SparkPi, SQL
- Workload Suites:
  - Collections of one or more workloads
  - Control benchmark output and parallelism
- Custom Workloads:
  - Use Scala and SBT to build onto SparkBench
  - Test custom Spark libraries by including JAR

```
spark-bench = {
  spark-submit-config = [{
    workload-suites = [
      {
        descr = "One run of SparkPi and that's it!"
        benchmark-output = "console"
        workloads = [
          {
            name = "sparkpi"
            slices = 10
          }
        ]
      }
    ]
  }]
}
```

```
One run of SparkPi and that's it!
+-------+-------------+-------------+-----------------+-----+-------------------------+------+---+-----------------+-----------------+----
|   name|    timestamp|total_runtime|   pi_approximate|input|workloadResultsOutputDir|slices|run|spark.driver.host|spark.driver.port|hive
+-------+-------------+-------------+-----------------+-----+-------------------------+------+---+-----------------+-----------------+----
|sparkpi|1498683099328|   1032871662|3.141851141851142|     |                         |    10|  0|      10.200.22.54|            61657|
+-------+-------------+-------------+-----------------+-----+-------------------------+------+---+-----------------+-----------------+----
```

Sample Output:

	A	B	C	D	E	F	G	H	I	J	K	L
1	name	timestamp	total_Runtime	run	cache	saveTime	queryStr	loadTime	saveMode	output	queryTime	input
2	sql	1532027918175.00	8783122380.00	0	FALSE	0	select * from input	8763790695	error		19331685	hdfs:///tmp/csv-vs-parquet/kmeans-data.csv
3	sql	1532027935960.00	8938864314.00	0	FALSE	0	select c0, c22 from input where c0 < -0.9	8854089340	error		84774974	hdfs:///tmp/csv-vs-parquet/kmeans-data.csv
4	sql	1532027945400.00	294150093.00	0	FALSE	0	select * from input	286799781	error		7350312	hdfs:///tmp/csv-vs-parquet/kmeans-data.parquet
5	sql	1532027949991.00	210775226.00	0	FALSE	0	select c0, c22 from input where c0 < -0.9	201379267	error		9395959	hdfs:///tmp/csv-vs-parquet/kmeans-data.parquet
6	sql	1532027954978.00	9170048329.00	1	FALSE	0	select * from input	9153201838	error		16846491	hdfs:///tmp/csv-vs-parquet/kmeans-data.csv
7	sql	1532027972754.00	9009234399.00	1	FALSE	0	select c0, c22 from input where c0 < -0.9	8990084746	error		19149653	hdfs:///tmp/csv-vs-parquet/kmeans-data.csv
8	sql	1532027982164.00	328433835.00	1	FALSE	0	select * from input	316105852	error		12327983	hdfs:///tmp/csv-vs-parquet/kmeans-data.parquet
9	sql	1532027982883.00	170355412.00	1	FALSE	0	select c0, c22 from input where c0 < -0.9	162862259	error		7493153	hdfs:///tmp/csv-vs-parquet/kmeans-data.parquet
10	sql	1532027991881.00	9212337818.00	2	FALSE	0	select * from input	9205135808	error		7202011	hdfs:///tmp/csv-vs-parquet/kmeans-data.csv
11	sql	1532028000737.00	8148945348.00	2	FALSE	0	select c0, c22 from input where c0 < -0.9	8141262036	error		7683312	hdfs:///tmp/csv-vs-parquet/kmeans-data.csv
12	sql	1532028018147.00	209987449.00	2	FALSE	0	select * from input	203314672	error		6672777	hdfs:///tmp/csv-vs-parquet/kmeans-data.parquet

SparkBench Config

```
spark-bench = {
  spark-submit-parallel = false
  spark-submit-config = [{
    spark-home = "/opt/spark"
    spark-args = {
      master = "yarn"
      driver-class-path = "/nfs/home/bmpowell/spark-bench_2.3.0_0.4.0-RELEASE/lib/*"
      driver-memory = 128g
      driver-cores = 8
    }
    conf = [
      {
        "spark.dynamicAllocation.executorIdleTimeout" = "120s"
        "spark.executor.cores" = "5"
        "spark.executor.memory" = "80g"
      },
      {
        "spark.dynamicAllocation.executorIdleTimeout" = "120s"
        "spark.executor.cores" = "4"
        "spark.executor.memory" = "53g"
      }
    ]
```

https://codait.github.io/spark-bench/

Workload Definition

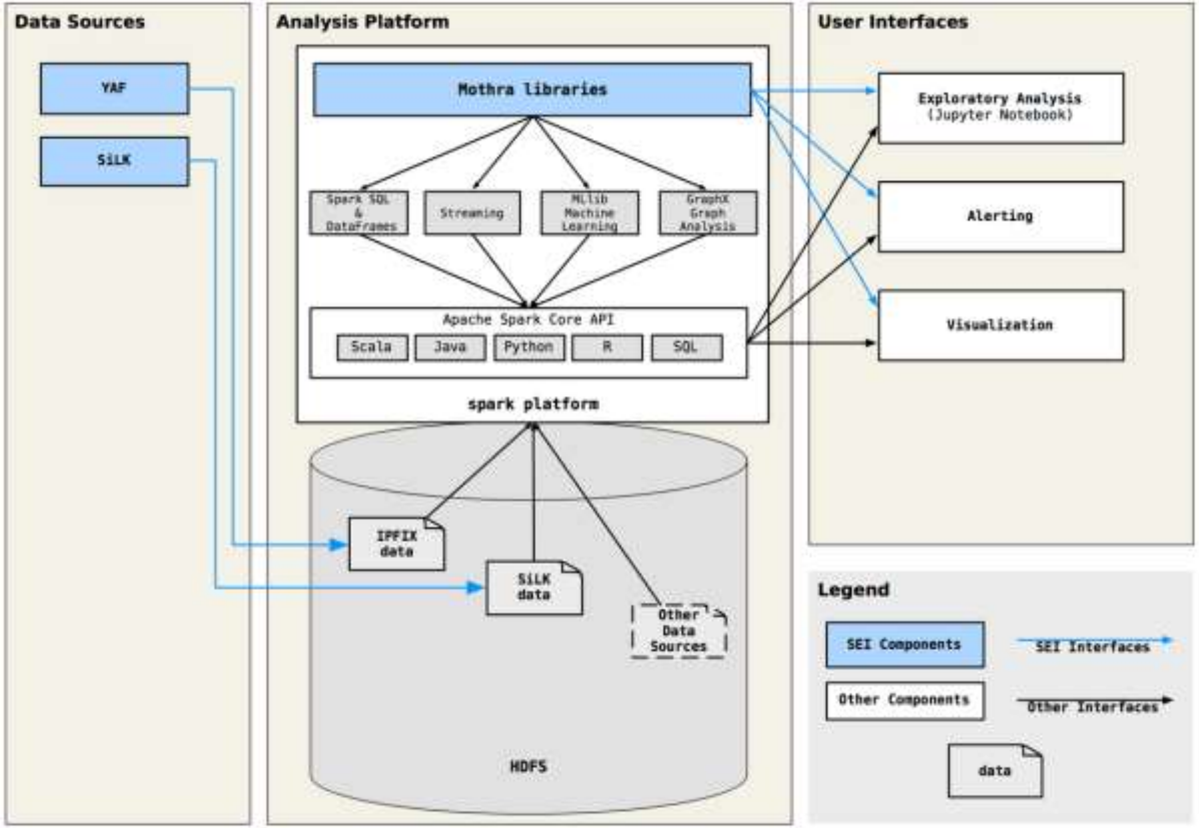
```
    suites-parallel = false
    workload-suites = [
      {
        descr = "Packed Count with filter on appLabel"
        benchmark-output = "hdfs:///tmp/mothra-test/mothra-appLabel-results.csv"
        parallel = false
        repeat = 2
        save-mode = "append"
        workloads = [
          {
            name = "custom"
            class = "com.example.MothraFilter"
            output = "hdfs:///tmp/mothra-test/mothra-results.csv"
            cache = "no"
            inputdata = "/data/mothra-ipfix/ixia-packed-applabel-test/"
          }
        ]
      }
      {
        descr = "Unpacked Count with filter on appLabel"
        benchmark-output = "hdfs:///tmp/mothra-test/mothra-appLabel-results.csv"
        parallel = false
        repeat = 1
        save-mode = "append"
        workloads = [
          {
            name = "custom"
            class = "com.example.MothraLoad"
            output = "hdfs:///tmp/mothra-test/mothra-results.csv"
            cache = "no"
            inputdata = "/data/mothra-ipfix/ixia/yaf-ixia*2018112*"
          }
        ]
      }
    ]
  }]
}
```

Automated Cluster Testing and Optimization

Mothra Refresher

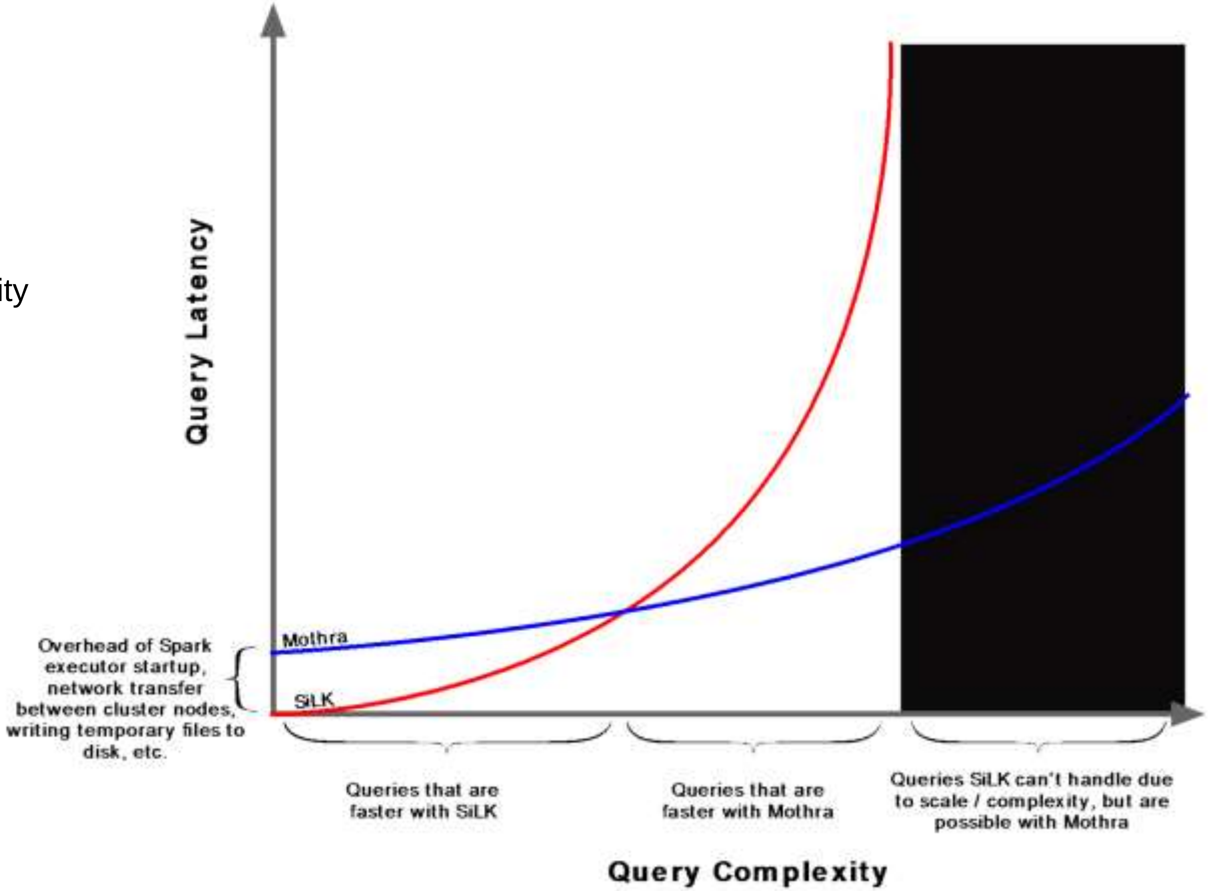
Mothra Architecture

- Facilitate bulk storage and analysis of cybersecurity data with high levels of flexibility, performance, and interoperability
- Reduce the engineering effort involved in developing, transitioning, and operationalizing new analytics
- Serve all major constituencies within the network security community, including data scientists, first-tier incident responders, system admins, and hobbyists



SiLK vs. Mothra Scalability

- Mothra enables more complex analyses at a scale beyond the capability of SiLK's single-node architecture



Automated Cluster Testing and Optimization

Test Plan

The goal of our testing was to identify the performance and benchmarks for the DTE cluster in the following areas:

1. Cluster Operations using pre-built, Micro and Machine Learning Workloads.
2. Mothra Dataframe Creation and Spark Query Performance.
3. Mothra Ingest Process Performance running Collector and Packer processes on a 16 core physical edge node.

Test Environment Details

- Number of Nodes: 40+10 Virtual nodes for NameNode, YARN Resource Manager, Zookeeper, and Edge Nodes
- RAM: 256GB, Disks: 4 Disks - 600 GB, CPU: 2x8 cores
- Network: Intel Corporation 82599ES 10-Gigabit dual port
- HDP Version: HDP 2.6.4 YARN
- Spark Version: Spark 2.2.1

Test parameters	Values
spark.submit.deployMode	client
spark.shuffle.service.enabled	true
spark.scheduler.mode	FIFO
spark.master	yarn
spark.executor.memory	4g
spark.dynamicAllocation.minExecutors	4
spark.dynamicAllocation.initialExecutors	4
spark.dynamicAllocation.enabled	true
spark.driver.port	36562
spark.driver.memory	8g

HiBench Generated Data

- Four groups: Large, Huge, Gigantic, and BigData

	Large	Huge	Gigantic	BigData
Naïve Bayes	hibench.bayes.large.pages 100000 hibench.bayes.large.classes 100 hibench.bayes.large.ngrams 2	hibench.bayes.huge.pages 500000 hibench.bayes.huge.classes 100 hibench.bayes.huge.ngrams 2	hibench.bayes.gigantic.pages 1000000 hibench.bayes.gigantic.classes 100 hibench.bayes.gigantic.ngrams 2	hibench.bayes.bigdata.pages 20000000 hibench.bayes.bigdata.classes 20000 hibench.bayes.bigdata.ngrams 2
Linear Regression	hibench.linear.large.examples 200000 hibench.linear.large.features 30000	hibench.linear.huge.examples 300000 hibench.linear.huge.features 50000	hibench.linear.gigantic.examples 500000 hibench.linear.gigantic.features 80000	hibench.linear.bigdata.examples 1000000 hibench.linear.bigdata.features 100000
Random Forest	hibench.rf.large.examples 1000 hibench.rf.large.features 1000	hibench.rf.huge.examples 10000 hibench.rf.huge.features 200000	hibench.rf.gigantic.examples 10000 hibench.rf.gigantic.features 300000	hibench.rf.bigdata.examples 20000 hibench.rf.bigdata.features 220000
K-means	hibench.kmeans.large.num_of_clusters 5 hibench.kmeans.large.dimensions 20 hibench.kmeans.large.num_of_samples 20000000 hibench.kmeans.large.samples_per_inputfile 4000000 hibench.kmeans.large.max_iteration 5 hibench.kmeans.large.k 10 hibench.kmeans.large.convergedist 0.5	hibench.kmeans.huge.num_of_clusters 5 hibench.kmeans.huge.dimensions 20 hibench.kmeans.huge.num_of_samples 100000000 hibench.kmeans.huge.samples_per_inputfile 20000000 hibench.kmeans.huge.max_iteration 5 hibench.kmeans.huge.k 10 hibench.kmeans.huge.convergedist 0.5	hibench.kmeans.gigantic.num_of_clusters 5 hibench.kmeans.gigantic.dimensions 20 hibench.kmeans.gigantic.num_of_samples 200000000 hibench.kmeans.gigantic.samples_per_inputfile 40000000 hibench.kmeans.gigantic.max_iteration 5 hibench.kmeans.gigantic.k 10 hibench.kmeans.gigantic.convergedist 0.5	hibench.kmeans.bigdata.num_of_clusters 5 hibench.kmeans.bigdata.dimensions 20 hibench.kmeans.bigdata.num_of_samples 24000000000 hibench.kmeans.bigdata.samples_per_inputfile 40000000 hibench.kmeans.bigdata.max_iteration 10 hibench.kmeans.bigdata.k 10 hibench.kmeans.bigdata.convergedist 0.5

Ixia Simulated IPFIX datasets

Filename(s)	Size	Record Count	Bytes per Flow	Description
/data/mothra-ipfix/live/ipfix-live-s1-20180820000030-00268.yaf	1.66 MB	8,746	189.80	1 hour of live ipfix data from DTE YAF sensor on 8/20
/data/mothra-ipfix/ixia/yaf-ixia-napa_lb0-20180819000021-00110.yaf	3.17 GB	21,997,654	144.11	1 hour of Ixia generated ipfix data for 1 YAF sensor on 8/19
/data/mothra-ipfix/ixia/yaf-ixia-napa_lb*-20180819000021-00110.yaf	22.19 GB	154,038,796	144.05	1 hour of Ixia generated ipfix data for 8 YAF sensors on 8/19
/data/mothra-ipfix/ixia/yaf-ixia-napa_lb*-20180819*.yaf	597.61 GB	4,159,659,052	143.67	24 hours of Ixia generated ipfix data for 8 YAF sensors on 8/19
/data/mothra-ipfix/ixia/yaf-ixia-napa_lb*-2018081*.yaf	3.08 TB	21,772,346,751	141.46	5.5 Days of Ixia generated ipfix data for 8 YAF sensors on 8/14-8/19

Filename(s)	Size	Record Count	Bytes per Flow	Description
/data/mothra-ipfix/ixia/yaf-ixia-napa_lb7-20180922110219-00274.yaf	263.78 MB	1,088,294	254.15	5 minutes of Ixia generated ipfix data for 1 YAF sensor on 9/22
/data/mothra-ipfix/ixia/yaf-ixia-napa_lb*-20180922110219-00274.yaf	2.06 GB	8,693,166	254.22	5 minutes of Ixia generated ipfix data for 8 YAF sensors on 9/22
/data/mothra-ipfix/ixia/yaf-ixia-napa_lb7-2018092211*	3.11 GB	13,199,568	252.92	1 hour of Ixia generated ipfix data for 1 YAF sensor on 9/22
/data/mothra-ipfix/ixia/yaf-ixia-napa_lb*-2018092211*	24.89 GB	105,639,034	253.00	1 hour of Ixia generated ipfix data for 8 YAF sensors on 9/22
/data/mothra-ipfix/ixia/yaf-ixia-napa_lb*-2018092*	605.74 GB	2,628,246,577	247.47	24 hours of Ixia generated ipfix data for 8 YAF sensors on 9/22

Automated Custom SparkBench Testing

Operation	Query
Build IPFIX DataFrame (mothra), Count	val input_data_ixia = "/data/mothra-ipfix/ixia/yaf-ixia-napa_lb*-20180815*.yaf" val input_df = (spark.read. fields("sIP", "dIP", "sPort", "dPort", "protocol", "packets", "bytes", "startTime", "endTime", "dnsQName" -> "ipfix:yaf_dns/yaf_dns_qr/dnsQName", "dnsQAddr" -> "ipfix:yaf_dns/yaf_dns_qr/yaf_dns_a/sourceIPv4Address") .ipfix(input_data_ixia) input_df.count()
Simple Filter (Spark), Count	var dns_flows = input_df.filter($"dport" === 53) dns_flows .count()
Column Selection & Display (Spark)	https_flows = input_df.filter($"dport" === 443).select("sip", "dip", "sport", "dport", "protocol", "packets", "bytes", "sslCertificateHash") https_flows.count()
Sorting (Spark)	https_flows.sort($"bytes".desc).show()
Aggregation (Spark)	https_flows. groupBy($"dip"). avg("packets", "bytes"). sort($"avg(bytes)".desc).count()
SQL Query (SparkSQL)	input_df.registerTempTable("df") spark.sql("""SELECT dnsQName, AVG(packets) AS avg_packets, SUM(packets) AS sum_packets, AVG(bytes) AS avg_bytes, SUM(bytes) AS sum_bytes FROM df WHERE dnsQName IS NOT NULL GROUP BY dnsQName ORDER BY sum_bytes DESC""").count()
Compound Query w/ Join, Filter, & Select (SparkSQL)	val bad_names = spark.read.parquet("/user/tonyc/data/sample/bad_dns_names.parquet") var bad_addrs = (dns_flows .join(bad_names, $"dnsQName" === $"name") .select("dnsQAddr") .distinct).toDF("addr") val pwned = input_df.join(bad_addrs, $"dIP" === $"addr").drop("addr") pwned.count()

Automated Cluster Testing and Optimization

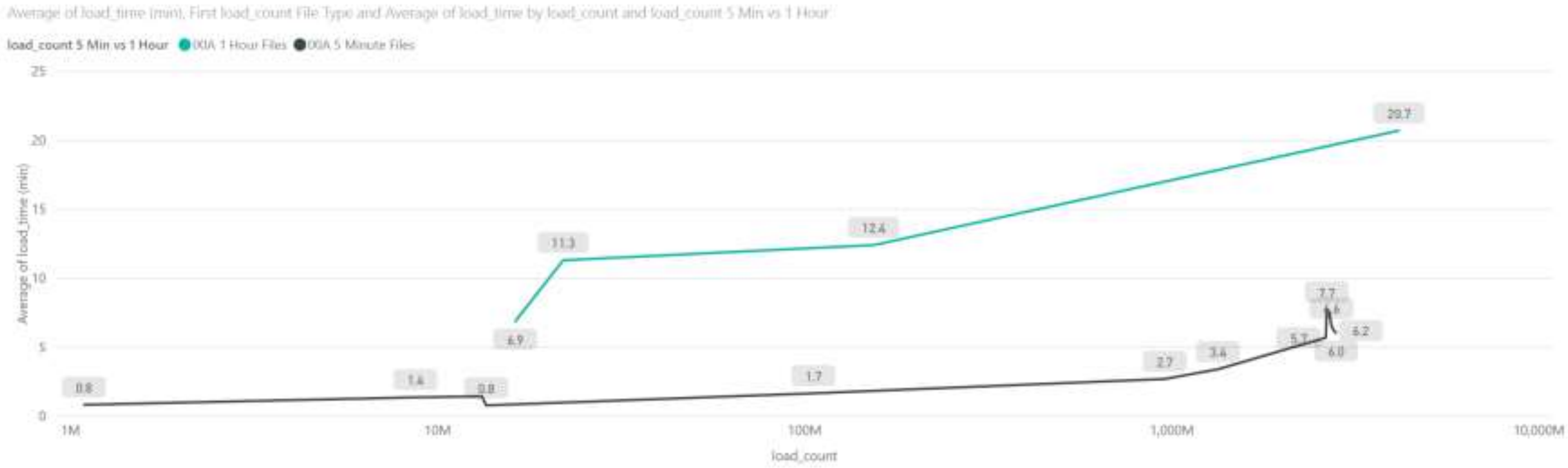
Results and Tuning

Operational ML Workloads

- Machine Learning workloads benchmark average throughput in GB/s over one month

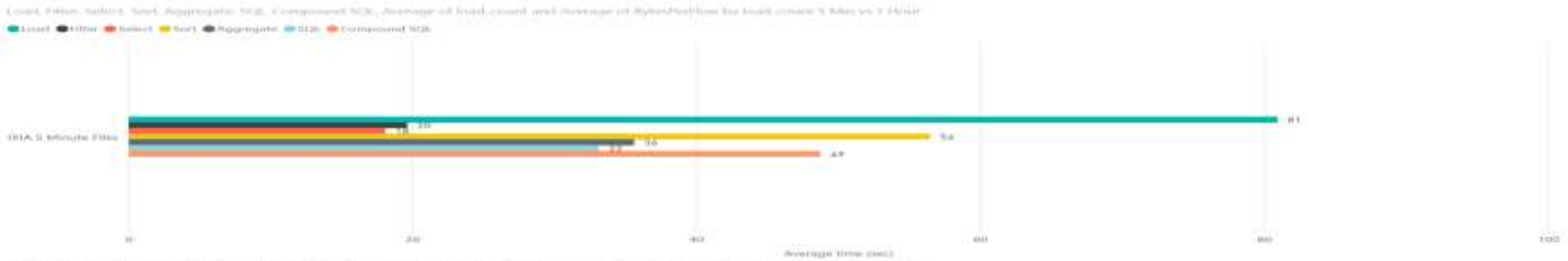
SparkBench Custom Mothra Workloads

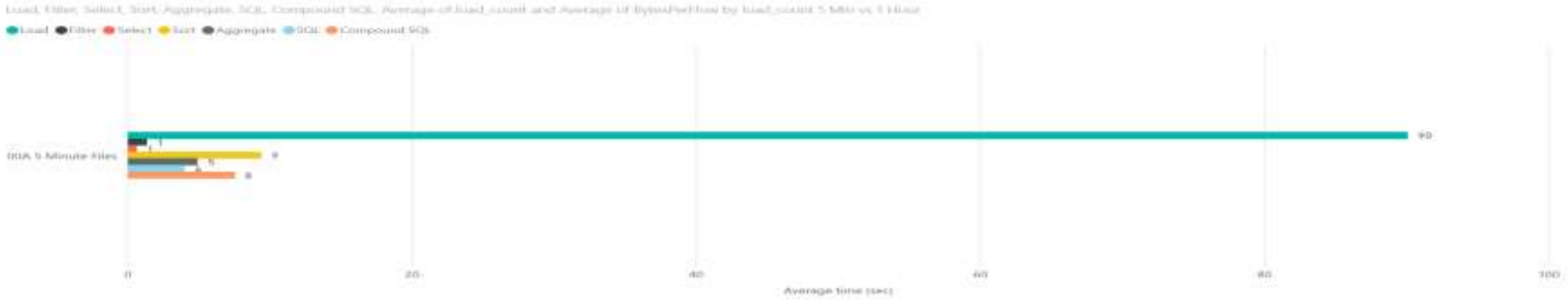
- Mothra Dataframe load time in minutes by input file record count and file size below. Graphs shows two different raw, unpartitioned file schemes. Green is one file per hour of data and black is twelve files per hour of data. There is a significant performance improvement when files are collected every five minutes vs. one hour.



SparkBench Custom Mothra Workloads

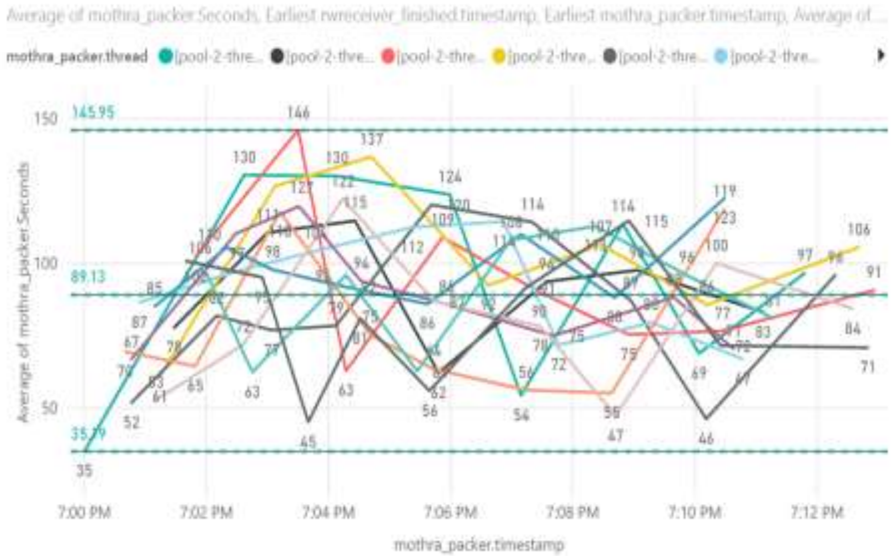
- Spark Submit completion times in seconds for Mothra and Spark queries. Graph is comparing equivalent data sets with one file per hour vs twelve files per hour. Caching in the second chart adds some overhead during load, but there is significant improvement in subsequent tasks reducing average processing time for all workloads from 293 seconds to 118 seconds a 60% improvement.

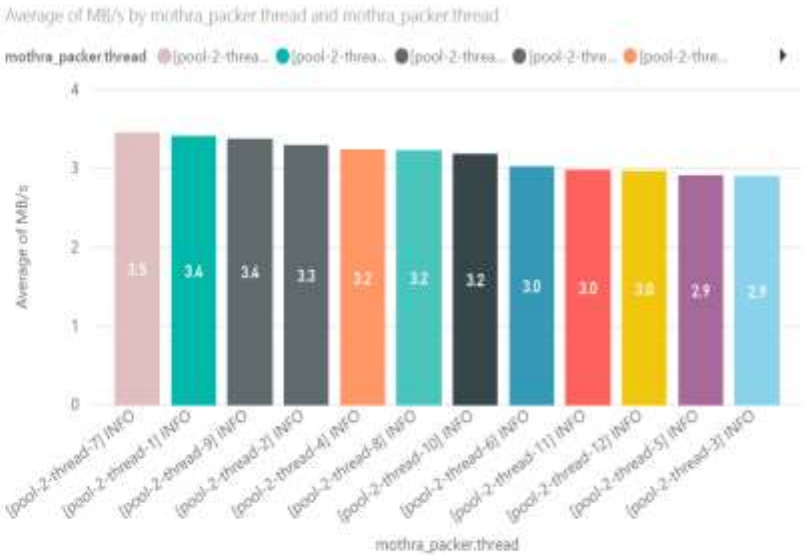




Mothra Packer Testing

- Load times and throughput for Mothra Packer. Two sample runs of 12 max pack jobs on a 16 core physical edge node. Rwsender landed 96 files (1 hour) at once with an average of ~271 MB per file (91,346,434 records). The average throughput per process is ~3MB/s. Adding polling and flush overhead, the average of total throughput is ~27MB/s



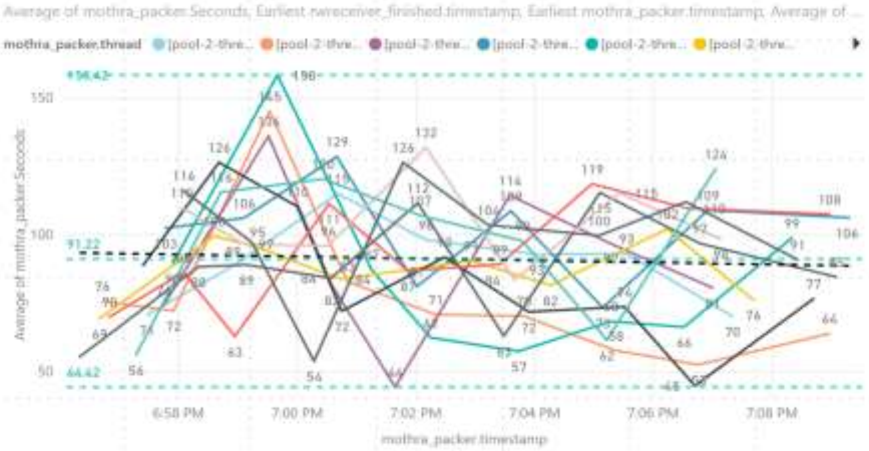


Mothra Packer Testing

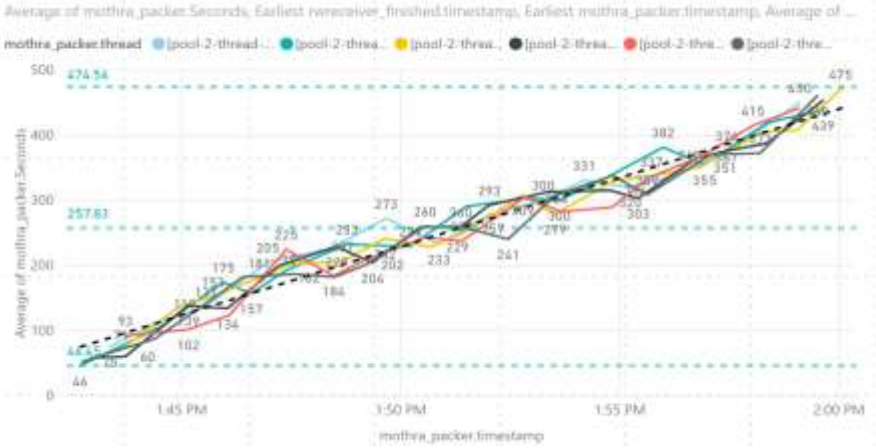
- Two sample runs 16 core edge node. Rwsender landed 96 files (1 hour) at once with an average of ~271 MB per file (91,346,434 records). The black dashed line shows the trend of completion time of each pack job.

- Test (a) shows a flat trend line which means that the jobs are keeping up with the files landing from rwsender while test (b) shows an incline trend which means that jobs are slowing over time and not able to keep up with the file ingestion. In both cases, one hour of our test data was packed in under 20 minutes, but test (a) should maintain this speed with more load, while test (b) would continue to slow as more files are landed.

(a) 12 Max Pack Jobs



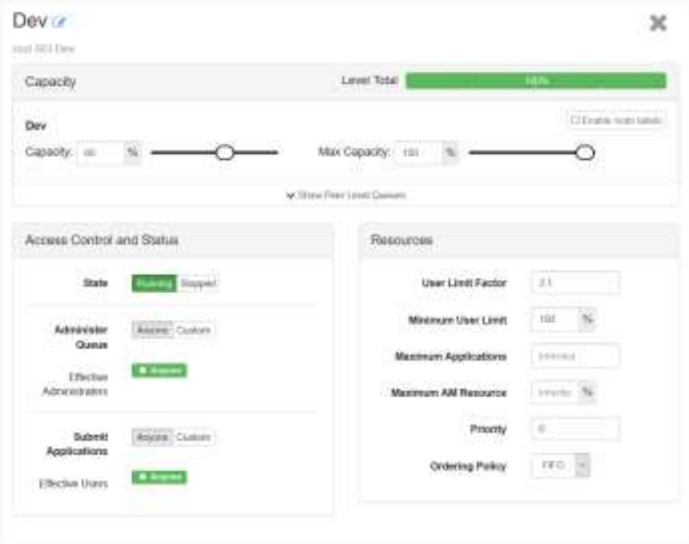
(b) 6 Max Pack Jobs



YARN Queue Manager / Capacity Scheduler

- Certain settings needed to be changed to take full advantage of the cluster resources and utilize dynamic allocation in Spark. Capacity and Max Capacity are not intuitive and only relate to the queue, not the whole cluster. In order to use resources beyond the queue (80% * 60% = 48%) , User Limit Factor needs to be set above 1.

- Depending on the number of users, Minimum User Limit and Ordering Policy can be used to avoid conflicts among analysts for cluster resources.



Spark Tuning

- Executor Cores
  - Typically no more than 5 cores can achieve full write throughput to HDFS
  - Setting cores too low (tiny executors) for large jobs on large clusters will cause garbage collection and out of memory errors
  - With executor-cores > 1, the DominantResourceCalculator must be selected for YARN

- Executor Memory
  - Calculated based on cluster size and executor-cores (Example = 6 nodes, 16 cores/node, 64gb memory/node, 5 executor-cores)

yarn.nodemanager.resource.cpu-vcores * total cluster nodes = total available cores	15 * 6 = 90 total available cores
total available cores / executor-cores = total available executors	90 / 5 = 18 total available executors
total available executors / total cluster nodes = number of executors per node	18 / 6 = 3 number of executors per node
yarn.nodemanager.resource.memory-mb / number of executors per node = memory per executor	63 / 3 = 21 memory per executor
memory per executor * (1 - spark.yarn.executor.memoryOverhead) = roundDown(executor-memory)	21 * (1 - .07) = roundDown(19.53) = 19 GB = executor-memory

Spark Tuning

- Dynamic Allocation Executor Idle Timeout
  - This option controls when executors are removed once idle.
  - Losing an executor due to a timeout and starting a new one adds additional overhead to a spark job.
  - For some use cases, such as exploratory analysis in Jupyter or Zeppelin, the default timeout of 60s might be too short.
  - Finding the ideal value for this, per use case, will require an iterative process between system administrators, cluster developers, and analysts.

Automated Cluster Testing and Optimization

Questions?

Contact

bmpowell@cert.org