Automated Cluster Testing and Optimization

Brad Powell
Sr. Security Engineer | CERT
Automated Cluster Testing and Optimization

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
Introduction

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
Automated Testing Tools

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

- Micro benchmark workloads:
  - Sort, WordCount, TeraSort, Sleep, Enhanced DFSIO

- SQL workloads:
  - Scan, Join, Aggregate

- Machine Learning workloads:

- Websearch benchmark workloads:
  - PageRank, Nutch indexing

- Graph benchmark workloads:
  - NWeight

- Streaming workloads:
  - Identity, Repartition, Stateful Wordcount, Fixwindow

Sample Output:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Date</td>
<td>Time</td>
<td>Input_data_size</td>
<td>Duration(s)</td>
<td>Throughput(bytes/s)</td>
<td>Throughput/node</td>
<td></td>
</tr>
<tr>
<td>ScalaSparkWordcount</td>
<td>7/19/2018 15:45:00</td>
<td>36790</td>
<td>29.818</td>
<td>1233</td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>ScalaSparkWordcount</td>
<td>7/19/2018 16:02:00</td>
<td>36790</td>
<td>28.841</td>
<td>1275</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>ScalaSparkWordcount</td>
<td>7/19/2018 16:16:58</td>
<td>36790</td>
<td>27.225</td>
<td>1351</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>ScalaSparkTerasort</td>
<td>7/19/2018 16:27:26</td>
<td>3200000</td>
<td>36.693</td>
<td>87210</td>
<td>2180</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>HadoopWordcount</td>
<td>7/19/2018 17:08:34</td>
<td>36790</td>
<td>28.971</td>
<td>1269</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

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
Automated Testing Tools

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

Sample Output:
Automated Testing Tools

**SparkBench Config**

```
spark-bench = {
  spark-submit-parallel = false
  spark-submit-config = (
    master = "yarn"
    driver-class-path = "/nfs/home/bnpowell/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-iplfix/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-iplfix/ixia-packed-applabel-test/"
      }
    ]
  }
]
```
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
Mothra Refresher

**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
Test Plan

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


Test Plan

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

<table>
<thead>
<tr>
<th>Test parameters</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>spark.submit.deployMode</td>
<td>client</td>
</tr>
<tr>
<td>spark.shuffle.service.enabled</td>
<td>true</td>
</tr>
<tr>
<td>spark.scheduler.mode</td>
<td>FIFO</td>
</tr>
<tr>
<td>spark.master</td>
<td>yarn</td>
</tr>
<tr>
<td>spark.executor.memory</td>
<td>4g</td>
</tr>
<tr>
<td>spark.dynamicAllocation.minExecutors</td>
<td>4</td>
</tr>
<tr>
<td>spark.dynamicAllocation.initialExecutors</td>
<td>4</td>
</tr>
<tr>
<td>spark.dynamicAllocation.enabled</td>
<td>true</td>
</tr>
<tr>
<td>spark.driver.port</td>
<td>36562</td>
</tr>
<tr>
<td>spark.driver.memory</td>
<td>8g</td>
</tr>
</tbody>
</table>
## Test Plan

### HiBench Generated Data

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

<table>
<thead>
<tr>
<th></th>
<th>Large</th>
<th>Huge</th>
<th>Gigantic</th>
<th>BigData</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Naïve Bayes</strong></td>
<td>hibench.bayes.large.pages 1000000</td>
<td>hibench.bayes.huge.pages 5000000</td>
<td>hibench.bayes.gigantic.pages 1000000</td>
<td>hibench.bayes.bigdata.pages 20000000</td>
</tr>
<tr>
<td></td>
<td>hibench.bayes.large.classes 100</td>
<td>hibench.bayes.huge.classes 100</td>
<td>hibench.bayes.gigantic.classes 100</td>
<td>hibench.bayes.bigdata.classes 2000000</td>
</tr>
<tr>
<td></td>
<td>hibench.bayes.large.ngrams 2</td>
<td>hibench.bayes.huge.ngrams 2</td>
<td>hibench.bayes.gigantic.ngrams 2</td>
<td>hibench.bayes.bigdata.ngrams 2</td>
</tr>
<tr>
<td><strong>Linear Regression</strong></td>
<td>hibench.linear.large.examples 200000</td>
<td>hibench.linear.huge.examples 300000</td>
<td>hibench.linear.gigantic.examples 500000</td>
<td>hibench.linear.bigdata.examples 10000000</td>
</tr>
<tr>
<td></td>
<td>hibench.linear.large.features 30000</td>
<td>hibench.linear.huge.features 50000</td>
<td>hibench.linear.gigantic.features 80000</td>
<td>hibench.linear.bigdata.features 1000000</td>
</tr>
<tr>
<td><strong>Random Forest</strong></td>
<td>hibench.rf.large.examples 1000</td>
<td>hibench.rf.large.features 10000</td>
<td>hibench.rf.gigantic.examples 10000</td>
<td>hibench.rf.bigdata.examples 200000</td>
</tr>
<tr>
<td></td>
<td>hibench.rf.large.features 10000</td>
<td>hibench.rf.large.features 20000</td>
<td>hibench.rf.gigantic.features 30000</td>
<td>hibench.rf.bigdata.features 22000000</td>
</tr>
<tr>
<td></td>
<td>hibench.kmeans.large.dimensions 20</td>
<td>hibench.kmeans.large.num_of_samples 200000000</td>
<td>hibench.kmeans.large.num_of_clusters 5</td>
<td>hibench.kmeans.bigdata.dimensions 20</td>
</tr>
<tr>
<td></td>
<td>hibench.kmeans.large.samples_per_inputfile 40000000</td>
<td>hibench.kmeans.large.dimensions 20</td>
<td>hibench.kmeans.large.num_of_samples 200000000</td>
<td>hibench.kmeans.bigdata.samples_per_inputfile 400000000</td>
</tr>
<tr>
<td></td>
<td>hibench.kmeans.large.max_iteration 5</td>
<td>hibench.kmeans.large.max_iteration 5</td>
<td>hibench.kmeans.large.max_iteration 5</td>
<td>hibench.kmeans.bigdata.max_iteration 10</td>
</tr>
<tr>
<td></td>
<td>hibench.kmeans.large.k 10</td>
<td>hibench.kmeans.huge.k 10</td>
<td>hibench.kmeans.gigantic.k 10</td>
<td>hibench.kmeans.bigdata.k 10</td>
</tr>
<tr>
<td></td>
<td>hibench.kmeans.large.convergedist 0.5</td>
<td>hibench.kmeans.huge.convergedist 0.5</td>
<td>hibench.kmeans.gigantic.convergedist 0.5</td>
<td>hibench.kmeans.bigdata.convergedist 0.5</td>
</tr>
</tbody>
</table>
## Ixia Simulated IPFIX datasets

<table>
<thead>
<tr>
<th>Filename(s)</th>
<th>Size</th>
<th>Record Count</th>
<th>Bytes per Flow</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/data/mothra-ipfix/live/ipfix-live-s1-20180820000030-00268.yaf</td>
<td>1.66 MB</td>
<td>8,746</td>
<td>189.80</td>
<td>1 hour of live ipfix data from DTE YAF sensor on 8/20</td>
</tr>
<tr>
<td>/data/mothra-ipfix/ixia/yaf-ixia-napa_lb0-20180819000021-00110.yaf</td>
<td>3.17 GB</td>
<td>21,997,654</td>
<td>144.11</td>
<td>1 hour of Ixia generated ipfix data for 1 YAF sensor on 8/19</td>
</tr>
<tr>
<td>/data/mothra-ipfix/ixia/yaf-ixia-napa_lb*-20180819000021-00110.yaf</td>
<td>22.19 GB</td>
<td>154,038,796</td>
<td>144.05</td>
<td>1 hour of Ixia generated ipfix data for 8 YAF sensors on 8/19</td>
</tr>
<tr>
<td>/data/mothra-ipfix/ixia/yaf-ixia-napa_lb*-20180819*.yaf</td>
<td>597.61 GB</td>
<td>4,159,659,052</td>
<td>143.67</td>
<td>24 hours of Ixia generated ipfix data for 8 YAF sensors on 8/19</td>
</tr>
<tr>
<td>/data/mothra-ipfix/ixia/yaf-ixia-napa_lb*-2018081*.yaf</td>
<td>3.08 TB</td>
<td>21,772,346,751</td>
<td>141.46</td>
<td>5.5 Days of Ixia generated ipfix data for 8 YAF sensors on 8/14-8/19</td>
</tr>
<tr>
<td>/data/mothra-ipfix/ixia/yaf-ixia-napa_lb7-20180922110219-00274.yaf</td>
<td>263.78 MB</td>
<td>1,088,294</td>
<td>254.15</td>
<td>5 minutes of Ixia generated ipfix data for 1 YAF sensor on 9/22</td>
</tr>
<tr>
<td>/data/mothra-ipfix/ixia/yaf-ixia-napa_lb*-20180922110219-00274.yaf</td>
<td>2.06 GB</td>
<td>8,693,166</td>
<td>254.22</td>
<td>5 minutes of Ixia generated ipfix data for 8 YAF sensors on 9/22</td>
</tr>
<tr>
<td>/data/mothra-ipfix/ixia/yaf-ixia-napa_lb7-2018092211*</td>
<td>3.11 GB</td>
<td>13,199,568</td>
<td>252.92</td>
<td>1 hour of Ixia generated ipfix data for 1 YAF sensor on 9/22</td>
</tr>
<tr>
<td>/data/mothra-ipfix/ixia/yaf-ixia-napa_lb*-2018092211*</td>
<td>24.89 GB</td>
<td>105,639,034</td>
<td>253.00</td>
<td>1 hour of Ixia generated ipfix data for 8 YAF sensors on 9/22</td>
</tr>
<tr>
<td>/data/mothra-ipfix/ixia/yaf-ixia-napa_lb*-2018092*</td>
<td>605.74 GB</td>
<td>2,628,246,577</td>
<td>247.47</td>
<td>24 hours of Ixia generated ipfix data for 8 YAF sensors on 9/22</td>
</tr>
</tbody>
</table>
## Test Plan

### Automated Custom SparkBench Testing

<table>
<thead>
<tr>
<th>Operation</th>
<th>Query</th>
</tr>
</thead>
</table>
| Build IPFIX DataFrame (mothra), Count | `val input_data_ixia = "data/mothra-iplc/ixia/ixia-ixia-napa_b_*-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",  
"protocol", "packets", "bytes", "sslCertificateHash")` |
| Sorting (Spark)                     | `https_flows.sort("bytes".desc).show()` |
| Aggregation (Spark)                 | `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")  
pwned = input_df.join(bad_addrs, "dIP" === "addr")  
pwned.count())` |
Results and Tuning
Results and Tuning

Operational ML Workloads

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

SparkBench Custom Mothra Workloads

- Mothra Dataframe load time in minutes by input file record count and file size below. Graphs show 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.
Results and Tuning

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.
Results and Tuning

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.
Results and Tuning

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)

<table>
<thead>
<tr>
<th>Calculation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>yarn.nodemanager.resource.cpu-vcores * total cluster nodes = total available cores</td>
<td>15 * 6 = 90 total available cores</td>
</tr>
<tr>
<td>total available cores / executor-cores = total available executors</td>
<td>90 / 5 = 18 total available executors</td>
</tr>
<tr>
<td>total available executors / total cluster nodes = number of executors per node</td>
<td>18 / 6 = 3 number of executors per node</td>
</tr>
<tr>
<td>yarn.nodemanager.resource.memory-mb / number of executors per node = memory per executor</td>
<td>63 / 3 = 21 memory per executor</td>
</tr>
<tr>
<td>memory per executor * (1 - spark.yarn.executor.memoryOverhead) = roundDown(executor-memory)</td>
<td>21 * (1 - .07) = roundDown(19.53) = 19 GB = executor-memory</td>
</tr>
</tbody>
</table>
Results and Tuning

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.
Questions?

Contact
bmpowell@cert.org