Tuning Topologies

Date of publish: 2017-11-06

CLOUDERA

Legal Notice

Contents

Performance Tuning Overview

You can use these very high level steps to tune your CCP topologies. For more detailed performance tuning information, see the instructions for tuning a parser, enrichment topology, and indexing topology.

Procedure

1. Start the tuning process with a single worker.

   After tuning the bolts within a single worker, scale out with additional worker processes.

2. Initially set the thread pool size to 1.

   Increase this value slowly only after tuning the other parameters first. Consider that each worker has its own thread pool and the total size of this thread pool should be far less than the total number of cores available in the cluster.

3. Initially set each bolt parallelism hint to the number of partitions on the input Kafka topic.

   Monitor bolt capacity and increase the parallelism hint for any bolt whose capacity is close to or exceeds 1.

4. If the topology is not able to keep-up with a given input, then increasing the parallelism is the primary means to scale up.

5. Parallelism units can be used for determining how to distribute processing tasks across the topology.

   The sum of parallelism can be close to, but should not far exceed this value.

   ```
   (number of worker nodes in cluster * number cores per worker node) -
     (number of acker tasks)
   ```

6. The throughput that the topology is able to sustain should be relatively consistent.

   If the throughput fluctuates greatly, increase back pressure using topology.max.spout.pending.

   When you restart the topologies, ensure that the Kafka offset strategy is set to LATEST.

Tuning a Parser

When tuning CCP, it is best to start with Parsers before moving on to Enrichments and Indexing topologies. A new parser should be tuned to efficiently handle the estimated throughput.

Tune Parser Kafka Partitions

When you tune a new parser, the first variable that you should determine is the minimum number of Kafka partitions required.

Procedure

1. Create a Kafka topic with a single partition.

2. Run the Kafka producer for a set amount of time.

   For example, 10 minutes.

3. Calculate the approximate number of events per second based on the total size of the Kafka partition.

4. Launch the parser topology with the following:

   - 1 spout
   - 1 worker
   - Several parser executers (10 or more)

5. Let the parser run for a set amount of time.
6. If the parser executors reach capacity, increase the number of executors and restart.

   When you restart the topologies, ensure that the Kafka offset strategy is set to "LATEST".
7. Calculate the approximate number of events per second from the statistics in the Storm user interface.
8. If the events in the Kafka topic are fully processed by the parser topology before the set amount of time is complete, you can omit the events per second calculation and instead use the first result.

   For example:

```
Num partitions = t/p
```

The number of partitions should be proportional to the number of Storm nodes. Because Kafka partitions are tied to the number of Kafka spouts, which need to be evenly distributed between Storm workers, the number of partitions should be divisible by the number of Storm workers.

Storm Parser Parameters

You can modify certain parser properties to tune your CCP architecture using the Management user interface. Modifying properties using the Management UI is simple and can be performed by any user.

Parsers tend to vary a lot. Some will be very high volume receiving thousands of messages per second and others will be much lower. Rather than using a standard setting for the number of partitions and parallelism, you should base your settings on the expected data volume. That said, use the following guidelines:

• The spout parallelism should be roughly the same as your Kafka partitions.
• Consider data flow when assigning Kafka partitions to parsers.
• Keep in mind the aggregate number of partitions when assigning them to partitions. You do not want to assign the maximum number of partitions to each parser because that can overload your system.

The parser topologies are deployed by a builder pattern that takes parameters from the CLI as set by the Management UI. The parser properties materialize as follows:

```
Management UI -> parser json config and CLI -> Storm
```

The following table lists the parser properties you can modify in the Management UI:

Category	Management UI Property Name	CLI Option
Storm topology config	Num Workers	-nw,--num_workers <NUM_WORKERS>
	Num Ackers	--na,--num_ackers <NUM_ACKERS>
	Storm Config	<JSON_FILE>, e.g., { "topology.max.spout.pending" : NUM }
Kafka	Spout Parallelism	-sp,--spout_p <SPOUT_PARALLELISM_HINT>
	Spout Num Tasks	-snt,--spout_num_tasks <NUM_TASKS>
	Spout Config	<JSON_FILE>, e.g., { "spout.pollTimeoutMs" : 200 }
	Spout Config	<JSON_FILE>, e.g., { "spout.maxUncommittedOffsets" : 10000000 }
	Spout Config	<JSON_FILE>, e.g., { "spout.offsetCommitPeriodMs" : 30000 }
Parser bolt	Parser Num Tasks	-pnt,--parser_num_tasks <NUM_TASKS>
	Parser Parallelism	-pp,--parser_p <PARALLELISM_HINT>

Category	Management UI Property Name	CLI Option
	Parser Parallelism	-pp,--parser_p <PARALLELISM_HINT>

All of the Storm parameters are available in the STORM SETTINGS section of the Management UI.

For the Storm config and Spout config properties, you enter the JSON_FILE information in the appropriate field using the JSON format supplied in the following table.

For more detail on starting parsers, see Starting and Stopping Parsers.

Tune Parser Core Storm Settings

You can set the number of Kafka spouts to match the number of Kafka partitions. You can also increase the number of workers and ackers to match the Storm nodes, unless the estimated throughput for the parser is very low.

Procedure

1. Set the parser Storm settings in the Management user interface.

2. You can add the following command to the Storm settings to test the parser:

```
spout.firstPollOffsetStrategy": "LATEST"
```

The command allows the Kafka topic to be written to continuously during testing so when the parser is restarted, the topology will not be flooded with events.

3. Increase the **Parser Parallelism** and **Num Tasks** values in increments based on the number of workers.

For example, in the previous example, the parameters could be incremented by 3.

4. As you increase the **Parser Parallelism** and **Num Tasks** values, check two Storm statistics: Parser Capacity and the number of tuples acked in a 10-minute window.

Topology stats

Window	Emitted	Transferred	Complete latency (ms)	Acked	Failed
10m 0s	0	0	0		
3h 0m 0s	0	0	0		
1d 0h 0m 0s	0	0	0		
All time	0	0	0		

Bolts (All time)

Search: [   ]

Id	Executors	Tasks	Emitted	Transferred	Capacity (last 10m)	Execute latency (ms)	Executed	Process latency (ms)	Acked	Failed	Error Host	Error Port	Last error	Error Time
errorMessageWriter	5	5	0	0	0.000	0.000	0	0.000	0	0				
parserBolt	18	18	0	0	0.000	0.000	0	0.000	0	0				

For a given estimated throughput, the capacity should be no greater than ~0.800. This will allow for ~20% overhead should the number of incoming events spike above the estimated average. If the capacity is above this level, **Parallelism** and **Num Tasks** should be incremented and the topology restarted.

The number of acked tuples should be approximately equal to (Desired Throughput ×600) assuming the topology has been active for at least 11 - 12 minutes. If the number of acked tuples and the capacity of the topology are both low, there may not be enough Kafka partitions.

If the Storm UI is showing a capacity of ~0.800 or less, the Kafka consumer should be monitored to ensure that there is no significant lag or buildup of messages for the parser. The following command shows an example of how this can be monitored via the command line on a Kafka node:

```
cd /usr/hdp/current/kafka-broker/bin/

watch -n 2 ./kafka-run-class.sh kafka.tools.ConsumerOffsetChecker --
zookeeper
master01:2181 --topic asa --group asa_parser
```

```
Group         Topic              Pid Offset         logSize         Lag           Owner
asa_parser    asa                0   233            234             1             none
asa_parser    asa                1   231            232             1             none
asa_parser    asa                2   234            235             1             none
asa_parser    asa                3   232            233             1             none
asa_parser    asa                4   232            233             1             none
asa_parser    asa                5   233            234             1             none
asa_parser    asa                6   231            232             1             none
asa_parser    asa                7   231            232             1             none
asa_parser    asa                8   234            235             1             none
```

Tune Additional Parser Storm Settings

After the number of parser executors has been determined and thoroughly tested, you can set or modify the last remaining Storm parameters.
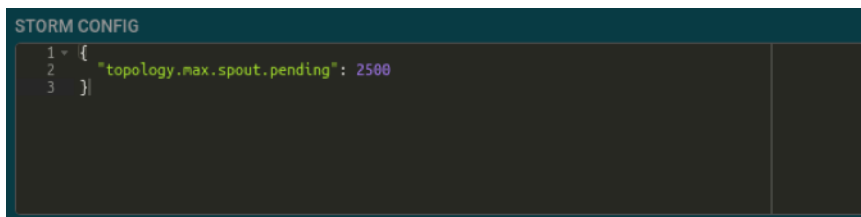
Procedure

1. Based on the capacity you've seen during testing, reduce the overall number of ackers.

Alternatively, you can leave a single acker per worker as it will ensure that there are no messages sent between Storm workers over the network interface.

2. Set the **Max Spout Pending** parameter such that the maximum number of unacked tuples in the topology is close to the **Parser Executor** capacity (for example, ~0.950).

Setting the maximum number of unacked tupples to the **Parser Executor** capacity ensures that if there is a large spike in incoming events, the topology will not become overloaded. For example, to determine this value you can increase the producer events per second by a large amount and test various values for **Max Spout Pending**. The value can be set under the Storm settings of the relevant parser.

```
STORM CONFIG
1 ▾ {
2       "topology.max.spout.pending": 2500
3 }
```
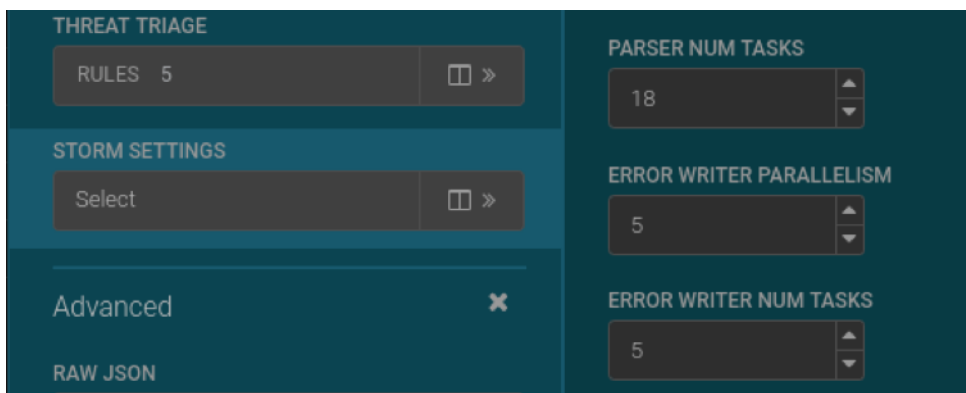
3. Check the **Parser Executor** capacity.

The **Parser Executor** capacity should not exceed ~0.950. Assuming the number of events generated by the producer is far greater than the capacity of the Parser topology, capacity is the only value that needs to be monitored in the Storm UI.

4. Set the **Error Writer Parallelism** and **Num Tasks** values.

Generally, since a small number of errors is expected, it can be set quite low. There should be more than 3 for redundancy but going as high as 1 per worker may also be of benefit for even distribution.

THREAT TRIAGE	PARSER NUM TASKS
RULES 5	18
STORM SETTINGS	ERROR WRITER PARALLELISM
Select	5
Advanced ✕	ERROR WRITER NUM TASKS
RAW JSON	5

Tuning an Enrichment Topology

Enrichment settings focus more on the compute workload than on the mapping workload in parsers or the IO driven workload in indexing. Enrichments make significant use of caching for performance. Because all of the data is coming together in enrichments, you will probably need larger enrichments settings than your parallelism settings. You can modify many performance tuning properties for enrichment using Ambari or Storm Flux. Modifying properties using Ambari is simple and can be performed by any user. However, you should have knowledge of Storm Flux usage and formatting before attempting to modify any Flux files.

Test Enrichment Topology Settings

While the parameters for the Enrichments topology should be modified via Ambari for persistence, there is a method by which the topologies can be started via the command line on the Metron node and parameters easily modified for testing. The commands below demonstrate how to create a copy of the Metron files for making changes quickly during testing.

Procedure

1. From the user's home directory, execute the following commands:

```
sudo cp /usr/hcp/current/metron/bin/start_enrichment_topology.sh ~
   sudo cp /usr/hcp/current/metron/config/enrichment.properties ~
   sed -i 's+$METRON_HOME/config/+/home/<user>/+g' ./
start_enrichment_topology.sh
```

While the product name has been rebranded to Cloudera Cybersecurity Platform (CCP), the product directories are still hcp.

2. Now, the variables can be edited outside of Ambari via the following command:

```
vi ~/enrichment.properties
```

3. To start the topology with the new variables, you must execute the following command:

```
~/start_enrichment_topology.sh
```

Tune Enrichment Kafka Partitions

The first enrichment variable that should be determined is the minimum number of Kafka partitions required.

Procedure

Use the following formula to determine the minimum number of Kafka partitions for the enrichment topology

```
Num Partitions=Max( t/p,t/c)
```

where

- t is the desired throughput
- p is the maximum throughout using a single producer
- c is the maximum throughput using a single consumer

> **Note:** You can estimate the minimum number of partitions requires based on the original calculations performed for parser topologies. However, this may or may not be suitable as the message size increases as it progresses through Metron topologies. This increased message size can affect throughput. For this reason, we recommend that you perform the same steps to calculate the value for the enrichment topic.

Storm Enrichment Parameters

You can modify various Storm enrichment properties for the unified topology using Ambari.

The following list provides tuning guidelines for the enrichment properties you can modify in Ambari:

enrichment.workers	The number of worker processes for the enrichment topology. Increase parallelism before attempting to increase the number of workers.
	Start by tuning only a single worker. Maximize throughput for that worker, then increase the number of workers.

The throughput should scale relatively linearly as workers are added. This reaches a limit as the number of workers running on a single node saturate the resources available.

When this happens, adding workers, but on additional nodes should allow further scaling.

enrichment.acker.executors

The number of ackers within the topology.

This should most often be equal to the number of workers defined in enrichment.workers.

Within the Storm UI, click the "Show System Stats" button. This will display a bolt named __acker. If the capacity of this bolt is too high, then increase the number of ackers.

topology.worker.childopts

This parameter accepts arguments that will be passed to the JVM created for each Storm worker. This allows for control over the heap size, garbage collection, and any other JVM-specific parameter.

Start with a 2G heap and increase as needed. Running with 8G was found to be beneficial, but will vary depending on caching needs.

-Xms8g -Xmx8g

The Garbage First Garbage Collector (G1GC) is recommended along with a cap on the amount of time spent in garbage collection. This is intended to help address small object allocation issues due to our extensive use of caches.

-XX:+UseG1GC -XX:MaxGCPauseMillis=100

If the caches in use are very large (as defined by either enrichment.join.cache.size or threat.intel.join.cache.size) and performance is poor, turning on garbage collection logging might be helpful.

topology.max.spout.pending

This limits the number of unacked tuples that the spout can introduce into the topology.

Decreasing this value will increase back pressure and allow the topology to consume messages at a pace that is maintainable.

If the spout throws 'Commit Failed Exceptions' then the topology is not keeping up. Decreasing this value is one way to ensure that messages can be processed before they time out.

If the topology's throughput is unsteady and inconsistent, decrease this value. This should help the topology consume messages at a maintainable pace.

If the bolt capacity is low, the topology can handle additional load. Increase this value so that more tuples are introduced into the topology which should increase the bolt capacity.

kafka.spout.parallelism

The parallelism of the Kafka spout within the topology. Defines the maximum number of

executors for each worker dedicated to running the spout.

The spout parallelism should most often be set to the number of partitions of the input Kafka topic.dd

If the enrichment bolt capacity is low, increasing the parallelism of the spout can introduce additional load on the topology.

enrichment.parallelism

The parallelism hint for the enrichment bolt. Defines the maximum number of executors within each worker dedicated to running the enrichment bolt.

If the capacity of the enrichment bolt is high, increasing the parallelism will introduce additional executors to bring the bolt capacity down.

If the throughput of the topology is too low, increase this value. This allows additional tuples to be enriched in parallel.

Increasing parallelism on the enrichment bolt will at some point put pressure on the downstream threat intel and output bolts. As this value is increased, monitor the capacity of the downstream bolts to ensure that they do not become a bottleneck.

threat.intel.parallelism

The parallelism hint for the threat intel bolt. Defines the maximum number of executors within each worker dedicated to running the threat intel bolt.

If the capacity of the threat intel bolt is high, increasing the parallelism will introduce additional executors to bring the bolt capacity down.

If the throughput of the topology is too low, increase this value. This allows additional tuples to be enriched in parallel.

Increasing parallelism on this bolt will at some point put pressure on the downstream output bolt. As this value is increased, monitor the capacity of the output bolt to ensure that it does not become a bottleneck.

kafka.writer.parallelism

The parallelism hint for the output bolt which writes to the output Kafka topic. Defines the maximum number of executors within each worker dedicated to running the output bolt.

If the capacity of the output bolt is high, increasing the parallelism will introduce additional executors to bring the bolt capacity down.

enrichment.cache.size

The Enrichment bolt maintains a cache so that if the same enrichment occurs repetitively, the value can be retrieved from the cache instead of it being recomputed. Increase the size of the cache to improve the rate of cache hits.

There is a great deal of repetition in network telemetry, which leads to a great deal of repetition for the enrichments that operate on that telemetry.

Having a highly performant cache is one of the most critical factors driving performance.

Increasing the size of the cache may require that you increase the worker heap size using `topology.worker.childopts'.

threat.intel.cache.size

The Threat Intel bolt maintains a cache so that if the same enrichment occurs repetitively, the value can be retrieved from the cache instead of it being recomputed.

There is a great deal of repetition in network telemetry, which leads to a great deal of repetition for the enrichments that operate on that telemetry. Having a highly performant cache is one of the most critical factors driving performance.

Increase the size of the cache to improve the rate of cache hits.

Increasing the size of the cache may require that you increase the worker heap size using `topology.worker.childopts'.

enrichment.threadpool.size

This value defines the number of threads maintained within a pool to execute each enrichment. This value can either be a fixed number or it can be a multiple of the number of cores (5C = 5 times the number of cores).

The enrichment bolt maintains a static thread pool that is used to execute each enrichment. This thread pool is shared by all of the executors running within the same worker.

Start with a thread pool size of 1. Adjust this value after tuning all other parameters first. Only increase this value if testing shows performance improvements in your environment given your workload.

If the thread pool size is too large this will cause the work to be shuffled amongst multiple CPU cores, which significantly decreases performance. Using a smaller thread pool helps pin work to a single core.

If the thread pool size is too small this can negatively impact IO-intensive workloads. Increasing the thread pool size, helps when using IO-intensive workloads with a significant cache miss rate. A thread pool size of 3-5 can help in these cases.

Most workloads will make significant use of the cache and so 1-2 threads will most likely be optimal.

The bolt uses a static thread pool. To scale out, but keep the work mostly pinned to a CPU core, add more Storm workers while keeping the thread pool size low.

If a larger thread pool increases load on the system, but decreases the throughput, then it is likely that

the system is thrashing. In this case the thread pool size should be decreased.

enrichment.threadpool.type

The enrichment bolt maintains a static thread pool that is used to execute each enrichment. This thread pool is shared by all of the executors running within the same worker.

Defines the type of thread pool used. This value can be either "FIXED" or "WORK_STEALING".

Currently, this value must be manually defined within the flux file at $METRON_HOME/flux/enrichment/remote-unified.yaml. This value cannot be altered within Ambari.

Tune Enrichment Core Storm Settings

You can set the number of Kafka spouts to match the number of Kafka partitions. You can also increase the number of workers and ackers to match the Storm nodes, unless the estimated throughput for the parser is very low.

Procedure

1. Set the parser Storm settings using the enrichment.properties file.

```
vi ~/enrichment.properties
```

```
##### Storm #####
enrichment.workers=3
enrichment.acker.executors=3
topology.worker.childopts=
topology.auto-credentials=[]
topology.max-spout.pending=

   ...

kafka.start=LATEST

   ...

##### Parallelism #####
kafka.spout.parallelism=9
```

2. Set the Kafka Offset Strategy to LATEST to allow the Kafka topic to be written to continuously during testing so when the parser is restarted, the topology will not be flooded with events.

```
kafka.start-LATEST
```

3. Alternatively, you can set the Kafka Offset Strategy to EARLIEST to determine the maximum throughput of the topology though you should set Max Spout Pending to avoid errors..

```
kafka.start-EARLIEST
```

4. Increase the enrichment.join.parallelism, threat.intel.join.parallelism, and kafka.writer.parallelism values in increments based on the number of workers.

For example, in the previous example, the parameters could be incremented by 3.

```
##### Parallelism #####
kafka.spout.parallelism=9
```

```
enrichment.split.parallelism=
enrichment.stellar.parallelism=
enrichment.join.parallelism=18
threat.intel.split.parallelism=
threat.intel.stellar.parallelism=
threat-intel.join-parallelism-18
kafka.writer.parallelism=9
```

5. As you increase the enrichment.join.parallelism, threat.intel.join.parallelism, and kafka.writer.parallelism values, check the two Storm statistics, Parser Capacity and the number of tuples acked in a 10-minute window.

Topology stats

Window	Emitted	Transferred	Complete latency (ms)	Acked	Failed
10m 0s	0	0	0		
3h 0m 0s	0	0	0		
1d 0h 0m 0s	0	0	0		
All time	0	0	0		

Bolts (All time)

Search:

Id	Executors	Tasks	Emitted	Transferred	Capacity (last 10m)	Execute latency (ms)	Executed	Process latency (ms)	Acked	Failed	Error Host	Error Port	Last error	Error Time
enrichmentBolt	18	18	0	0	0.000	0.000	0	0.000	0	0				
enrichmentErrorOutputBolt	1	1	0	0	0.000	0.021	1900	0.011	1900	0				
outputBolt	9	9	0	0	0.000	0.008	17040	0.004	17060	0				
threatIntelBolt	18	18	0	0	0.000	0.000	0	0.000	0	0				
threatIntelErrorOutputBolt	1	1	0	0	0.000	0.000	1880	0.000	1900	0				

For a given estimated throughput, the capacity should be no greater than ~0.800. This will allow for ~20% overhead should the number of incoming events spike above the estimated average. If the capacity is above this level, **Parallelism** and **Num Tasks** should be incremented and the topology restarted.

The number of acked tuples should be approximately equal to (Desired Throughput ×600) assuming the topology has been active for at least 11 - 12 minutes. If the number of acked tuples and the capacity of the topology are both low, there may not be enough Kafka partitions.

If the Storm UI is showing a capacity of ~0.800 or less, the Kafka consumer should be monitored to ensure that there is no significant lag or buildup of messages for the parser. The command below shows an example of how this can be monitored via the command line on a Kafka node:

```
cd /usr/hdp/current/kafka-broker/bin/

watch -n 2 ./kafka-run-class.sh kafka.tools.ConsumerOffsetChecker --
zookeeper
master01:2181 --topic enrichments --group enrichments
```

```
Group        Topic                         Pid Offset      logSize      Lag       Owner
enrichments  enrichments                   0   346         347          1         none
enrichments  enrichments                   1   346         347          1         none
enrichments  enrichments                   2   339         340          1         none
enrichments  enrichments                   3   340         341          1         none
enrichments  enrichments                   4   345         346          1         none
enrichments  enrichments                   5   342         343          1         none
enrichments  enrichments                   6   349         350          1         none
enrichments  enrichments                   7   349         350          1         none
enrichments  enrichments                   8   344         345          1         none
```

Tune Additional Enrichment Storm Settings

After the number of enrichment executors has been determined and thoroughly tested, you can set or modify the last remaining Storm parameters.

1. Based on the capacity you've seen during testing, reduce the overall number of ackers.

   Alternatively, you an leave a single acker per worker as it will ensure that there are no messages sent between Storm workers over the network interface.

2. Set the **Max Spout Pending** parameter such that the maximum number of unacked tuples in the topology is close to the **Parser Executor** capacity (for example, ~0.950).

   If this is the case then it can be assured that if there is a large spike in incoming events, the topology will not become overloaded. An example approach to determine this value would be to increase the producer events per second by a large amount and test various values for **Max Spout Pending**. The value can be set under the Storm settings of the relevant Parser.

   ```
   vi ~/enrichment.properties
   ```

   ```
   ##### Storm #####
   enrichment.workers=3
   enrichment.acker.executors=3
   topology.worker.childopts=
   topology.auto-credentials=[]
   topology.max-spout.pending=
   ```

3. Check the **Executor** capacity.

   The executor capacity should not exceed ~0.950. Assuming the number of events generated by the producer is far greater than the capacity of the Parser topology, capacity is the only value that needs to be monitored in the Storm UI.

4. If you need to increase the **Error Writer Num Executors** value, you can directly modify the Flux file and include the "parallelism" parameter under the appropriate Storm Bolt declarations.

   While the product name has been rebranded to Cloudera Cybersecurity Platform (CCP), the directory name currently remains hcp.

   ```
   sudo vi /usr/hcp/current/metron/flux/enrichment/remote-unified.yaml
   ```

   ```
   id: "enrichmentErrorOutputBolt"
   className: "org.apache.metron.writer.bolt.BulkMesageWriterBolt"
   constructorArgs:
       - "${kafka.zk}"
   configMethods:
       - name: "withMessageWriter"
                 args:
                       - ref: "enrichmentErrorKafkaWriter"
   parallelism: 3
   ```

   Generally, since a small number of errors is expected, the **Error Writer Num Executors** value does not need to be increased.

Modifying Enrichment Properties Using Flux (Advanced)

Some of the tuning enrichment properties can only be modified using Flux. However, if you manually change your Flux file, if you perform an upgrade, Ambari will overwrite all of your changes. Be sure to save your Flux changes prior to performing an upgrade.

⚠️ **Important:** You should be familiar with Storm Flux before you adjust the values in this section. Changes to Flux file properties that are managed by Ambari will render Ambari unable to further manage the property.

You can find the enrichment Flux file at $METRON_HOME/flux/enrichment/remote.yaml.

The following table lists the enrichment properties you can modify in the flux file:

Category	Flux Property or Function	Flux Section Location
Kafka spout	session.timeout.ms	line 201, id: kafkaProps
	enable.auto.commit	line 201, id: kafkaProps
	setPollTimeoutMs	line 230, id: kafkaConfig
	setMaxUncommittedOffsets	line 230, id: kafkaConfig
	setOffsetCommitPeriodMs	line 230, id: kafkaConfig

You can add Kafka spout properties or functions using two methods:

Flux properties - Flux # kafkaProps

Add a new key/value to the kafkaProps section HashMap on line 201. For example, if you want to set the Kafka Spout consumer'ssession.timeout.ms to 30 seconds, add the following:

```
        -   name: "put"
            args:
                -
   "session.timeout.ms"
                - 30000
```

Flux functions - Flux # kafkaConfig

Add a new setter to the kafkaConfig object section on line 230. For example, if you want to set the Kafka Spout consumer's poll timeout to 200 milliseconds, add the following under configMethods:

```
        -   name:
   "setPollTimeoutMs"
            args:
                - 200
```

Tuning a Batch Indexing Topology

Indexing is primarily IO driven. Tuning indexing tends to focus on the search index (Solr or Elasticsearch). Problems with indexing running too slow will often manifest as Kafka not committing in time. This results from the indexing backing up so that it fails batches and the poll interval in Kafka is exceeded. The issue is actually with the index rather than Kafka.

Test Batch Indexing Topology Settings

While the parameters for the Batch Indexing topology should be modified via Ambari for persistence, there is a method by which the topologies can be started via the command line on the Metron node and parameters easily modified for testing. The commands below demonstrate how to create a copy of the Metron files for making changes quickly during testing.

Procedure

1. From the user's home directory, execute the following commands:

```
sudo cp /usr/hcp/current/metron/bin/start_hdfs_topology.sh ~
   sudo cp /usr/hcp/current/metron/config/hdfs.properties ~
   sed -i 's+$METRON_HOME/config/+/home/<user>/+g' ./start_hdfs_topology.sh
```

2. Now, the variables can be edited outside of Ambari via the following command:

```
vi ~/hdfs.properties
```

3. To start the topology with the new variables, you must execute the following command:

```
~/start_hdfs_topology.sh
```

Tune Batch Indexing Kafka Partitions

The first batch indexing variable that should be determined is the minimum number of Kafka partitions required.

Procedure

Use the following formula to determine the minimum number of Kafka partitions for the enrichment topology

```
Num Partitions=Max( t/p,t/c)
```

where

- t is the desired throughput
- p is the maximum throughout using a single producer
- c is the maximum throughput using a single consumer

Note: You can estimate the minimum number of partitions requires based on the original calculations performed for enrichment topologies. However, this may or may not be suitable as the message size increases as it progresses through Metron topologies. This increased message size can affect throughput. For this reason, we recommend that you perform the same steps to calculate the value for the batch indexing topic.

Storm Index Parameters

You can modify various Storm indexing properties using Ambari. The HDFS sync policy is not currently managed by Ambari. To accommodate the HDFS sync policy setting, modify the Flux file directly.

The following table lists the indexing properties you can modify in Ambari:

Category	Ambari Property Name	Storm Property Name
Storm topology config	enrichment_workers	topology.workers

Category	Ambari Property Name	Storm Property Name
	enrichment_acker_executors	topology.acker.executors
	enrichment_topology_max_spout_pending	topology.max.spout.pending
Kafka spout	batch_indexing_kafka_spout_parallelism	n/a
Output bolt	hdfs_writer_parallelism	n/a
	bolt_hdfs_rotation_policy_units	n/a
	bolt_hdfs_rotation_policy_count	n/a

Tune Batch Indexing Core Storm Settings

You can set the number of Kafka spouts to match the number of Kafka partitions. You can also increase the number of workers and ackers to match the Storm nodes, unless the estimated throughput for the parser is very low.

Procedure

1. Set the parser Storm settings using the enrichment.properties file.

```
vi ~/hdfs.properties
```

```
##### Storm #####
enrichment.workers=3
enrichment.acker.executors=3
topology.worker.childopts=
topology.auto-credentials=[]
topology.max-spout.pending=

   ...

kafka.start=LATEST

   ...

##### Parallelism #####
kafka.spout.parallelism=9
```

2. Set the Kafka Offset Strategy to LATEST to allow the Kafka topic to be written to continuously during testing so when the parser is restarted, the topology will not be flooded with events.

```
kafka.start-LATEST
```

3. Alternatively, you can set the **Kafka Offset Strategy** to EARLIEST to determine the maximum throughput of the topology though you should set **Max Spout Pending** to avoid errors..

```
kafka.start-EARLIEST
```

4. Increase the hdfs.writer.parallelism values in increments based on the number of workers.

For example, in the previous example, the parameters could be incremented by 3.

```
##### Parallelism #####
kafka.spout.parallelism=9
enrichment.split.parallelism=
enrichment.stellar.parallelism=
enrichment.join.parallelism=18
threat.intel.split.parallelism=
threat.intel.stellar.parallelism=
```

```
threat-intel.join-parallelism-18
kafka.writer.parallelism=9
```

5. As you increase the enrichment.join.parallelism, threat.intel.join.parallelism, and kafka.writer.parallelism values, check the two Storm statistics, Capacity of the HDFS indexing component, the number of tuples acked in a 10-minute window, and the complete latency.

Topology stats

Window	Emitted	Transferred	Complete latency (ms)	Acked	Failed
10m 0s	0	0	0		
3h 0m 0s	0	0	0		
1d 0h 0m 0s	0	0	0		
All time	0	0	0		

Bolts (All time)

Search:

Id	Executors	Tasks	Emitted	Transferred	Capacity (last 10m)	Execute latency (ms)	Executed	Process latency (ms)	Acked	Failed	Error Host	Error Port	Last error	Error Time
hdfsIndexingBolt	18	18	0	0	0.000	0.035	14460	0.011	14480	0				
indexingErrorBolt	1	1	0	0	0.000	0.050	800	0.025	800	0				

For a given estimated throughput, the capacity should be no greater than ~0.800. This will allow for ~20% overhead should the number of incoming events spike above the estimated average. If the capacity is above this level, **Parallelism** and **Num Tasks** should be incremented and the topology restarted.

The number of acked tuples should be approximately equal to (Desired Throughput ×600) assuming the topology has been active for at least 11 - 12 minutes. If the number of acked tuples and the capacity of the topology are both low, there may not be enough Kafka partitions.

If the Storm UI is showing a capacity of ~0.800 or less, the Kafka consumer should be monitored to ensure that there is no significant lag or buildup of messages for the parser. The command below shows an example of how this can be monitored via the command line on a Kafka node:

```
cd /usr/hdp/current/kafka-broker/bin/

watch -n 2 ./kafka-run-class.sh kafka.tools.ConsumerOffsetChecker --
zookeeper
master01:2181 --topic indexing --group index-batch
```

```
Group          Topic      Pid Offset      logSize      Lag      Owner
indexing-batch indexing   0   560         561          1        none
indexing-batch indexing   1   605         606          1        none
indexing-batch indexing   2   609         610          1        none
indexing-batch indexing   3   606         607          1        none
indexing-batch indexing   4   606         607          1        none
indexing-batch indexing   5   608         609          1        none
indexing-batch indexing   6   605         606          1        none
indexing-batch indexing   7   604         605          1        none
indexing-batch indexing   8   608         609          1        none
```

Tune Additional Batch Indexing Storm Settings

After the number of executors has been determined and thoroughly tested, you can set or modify the last remaining Storm parameters.

Procedure

1. Based on the capacity you've seen during testing, reduce the overall number of ackers.

Alternatively, you an leave a single acker per worker as it will ensure that there are no messages sent between Storm workers over the network interface.

2. Set the **Max Spout Pending** parameter such that the maximum number of unacked tuples in the topology is close to the Parser Executor capacity (for example, ~0.950).

   If this is the case then it can be assured that if there is a large spike in incoming events, the topology will not become overloaded. An example approach to determine this value would be to increase the producer events per second by a large amount and test various values for **Max Spout Pending**. The value can be set under the Storm settings of the relevant Parser.

   ```
   vi ~/enrichment.properties
   ```

   ```
   ##### Storm #####
   indexing.workers=3
   indexing.acker.executors=3
   topology.worker.childopts=
   topology.auto-credentials=[]
   topology.max-spout.pending=
   ```

3. Check the Executor capacity.

   The executor capacity should not exceed ~0.950. Assuming the number of events generated by the producer is far greater than the capacity of the Parser topology, capacity is the only value that needs to be monitored in the Storm UI.

4. If you need to increase the **Error Writer Num Executors** value, you can directly modify the Flux file and include the "parallelism" parameter under the appropriate Storm Bolt declarations.

   While the product has been rebranded to Cloudera Cybersecurity Platform (CCP), the directory currently is hcp.

   ```
   sudo vi /usr/hcp/current/metron/flux/indexing/batch/remote.yaml
   ```

   ```
   id: "indexingErrorOutputBolt"
   className: "org.apache.metron.writer.bolt.BulkMesageWriterBolt"
   constructorArgs:
       - "${kafka.zk}"
   configMethods:
       - name: "withMessageWriter"
                 args:
                       - ref: "KafkaWriter"
   parallelism: 3
   ```

   Generally, since a small number of errors is expected, the Error Writer Num Executors value does not need to be increased.

Modifying Index Parameters Using Flux (Advanced)

Some of the tuning indexing properties, for example the HDFS sync policy setting, can only be modified using Flux. However, if you manually change your Flux file, if you perform an upgrade, Ambari will overwrite all of your changes. Be sure to back up your Flux changes prior to performing an upgrade.

⚠️ **Important:** You should be familiar with Storm Flux before you adjust the values in this section. Changes to Flux file properties that are managed by Ambari will render Ambari unable to further manage the property.

You can find the indexing Flux file at $METRON_HOME/flux/indexing/batch/remote.yaml.

Category	Flux Property	Flux Section Location	Suggested Value
Kafka spout	session.timeout.ms	line 80, id: kafkaProps	Kafka consumer client property

Category	Flux Property	Flux Section Location	Suggested Value
	enable.auto.commit	line 80, id: kafkaProps	Kafka consumer client property
	setPollTimeoutMs	line 108, id: kafkaConfig	Kafka consumer client property
	setMaxUncommittedOffsets	line 108, id: kafkaConfig	Kafka consumer client property
	setOffsetCommitPeriodMs	line 108, id: kafkaConfig	Kafka consumer client property
Output bolt	hdfsSyncPolicy	line 47, id: hdfsWriter	See notes below about adding this prop

To modify index tuning properties, complete the following steps:

1. Add a new setter to the hdfsWriter around line 56.

```
53                  -    name: "withRotationPolicy"
54                       args:
55                          - ref: "hdfsRotationPolicy
56                  -    name: "withSyncPolicy"
57                       args:
58                          - ref: "hdfsSyncPolicy
```

Lines are 53-55 provided for context.

2. Add an hdfsSyncPolicy after the hdfsRotationPolicy that appears on line 41:

```
41        -    id: "hdfsRotationPolicy"
...
45              - "${bolt.hdfs.rotation.policy.units}"
46
47        -    id: "hdfsSyncPolicy"
48             className: "org.apache.storm.hdfs.bolt.sync.CountSyncPolicy"
49             constructorArgs:
50               -  100000
```

Tuning a Random Access Indexing Topology

Indexing is primarily IO driven. Tuning indexing tends to focus on the search index (Solr or Elasticsearch). Problems with indexing running too slow will often manifest as Kafka not committing in time. This results from the indexing backing up so that it fails batches and the poll interval in Kafka is exceeded. The issue is actually with the index rather than Kafka.

Test Random Access Indexing Topology Settings

While the parameters for the Indexing topology should be modified via Ambari for persistence, there is a method by which the topologies can be started via the command line on the Metron node and parameters easily modified for testing. The commands below demonstrate how to create a copy of the Metron files for making changes quickly during testing.

Procedure

1. From the user's home directory, execute the following commands:

```
sudo cp /usr/hcp/current/metron/bin/start_elasticsearch_topology.sh ~
  sudo cp /usr/hcp/current/metron/config/elasticsearch.properties ~
  sed –i 's+$METRON_HOME/config/+/home/<user>/+g' ./
start_elasticsearch_topology.sh
```

While the product has been rebranded to Cloudera Cybersecurity Platform (CCP), the directory name currently remains hcp.

2. Now, the variables can be edited outside of Ambari via the following command:

```
vi ~/elasticsearch.properties
```

3. To start the topology with the new variables, you must execute the following command:

```
~/start_elasticsearch_topology.sh
```

Tune Random Access Indexing Kafka Partitions

If the number of Kafka partitions was correctly calculated for the Indexing topic, then no modifications should be required to the Kafka topic.

Tune Random Access Indexing Elasticsearch Templates

Before tuning the Elasticsearch indexing topology, the Elasticsearch templates for the appropriate sensors should be created and uploaded to Elasticsearch.

Procedure

1. Use a curl request to upload the Elasticsearch templates:

```
curl -X POST \
  http://<ES Master IP>:9200/_template/<sensor>_index \
  -H 'content-type: application/json' \
  -d <Template JSON>
```

```
{
  "template": "<sensor>_index*",
  "mappings": {
        "_default_": {
      "_all": {
          "enabled": "false"
      }
    },
    },
    "<sensor>_doc": {
      "dynamic_templates": [
        {
          "geo_location_point": {
            "match": "enrichments:geo:*:location_point",
            "match_mapping_type": "*",
            "mapping": {
              "type": "geo_point"
            }
          }
        },
        {
          "geo_country": {
            "match": "enrichments:geo:*:country",
            "match_mapping_type": "*",
            "mapping": {
              "type": "keyword"
            }
          }
```

```
            },
            {
            "geo_city": {
            "match": "enrichments:geo:*:city",
            "match_mapping_type": "*",
            "mapping": {
            "type": "keyword"
            }
            }
            },
            {
            "geo_location_id": {
            "match": "enrichments:geo:*:locID",
            "match_mapping_type": "*",
            "mapping": {
            "type": "keyword"
            }
            }
            },
            {
            "geo_dma_code": {
            "match": "enrichments:geo:*:dmaCode",
            "match_mapping_type": "*",
            "mapping": {
            "type": "keyword"
            }
            }
            },
            {
            "geo_postal_code": {
            "match": "enrichments:geo:*:postalCode",
            "match_mapping_type": "*",
            "mapping": {
            "type": "keyword"
            }
            }
            },
            {
            "geo_latitude": {
            "match": "enrichments:geo:*:latitude",
            "match_mapping_type": "*",
            "mapping": {
            "type": "float"
            }
            }
            },
            {
            "geo_longitude": {
            "match": "enrichments:geo:*:longitude",
            "match_mapping_type": "*",
            "mapping": {
            "type": "float"
            }
            }
            },
            {
            "timestamps": {
            "match": "*:ts",
            "match_mapping_type": "*",
            "mapping": {
            "type": "date",
            "format": "epoch_millis"
            }
            }
```

```
        },
        {
        "threat_triage_score": {
        "mapping": {
        "type": "float"
        },
        "match": "threat:triage:*score",
        "match_mapping_type": "*"
        }
        },
        {
        "threat_triage_reason": {
        "mapping": {
        "type": "text",
        "fielddata": "true"
        },
        "match": "threat:triage:rules:*:reason",
        "match_mapping_type": "*"
        }
        },
        {
        "threat_triage_name": {
        "mapping": {
        "type": "text",
        "fielddata": "true"
        },
        "match": "threat:triage:rules:*:name",
        "match_mapping_type": "*"
        }
        }
        ],
        "properties": {
        "timestamp": {
        "type": "date",
        "format": "epoch_millis"
        },
        "source:type": {
        "type": "text",
        "fielddata": "true"
        }
        "is_alert": {
        "type": "boolean"
        },
        "alert": {
        "type": "nested"
        }
        }
        }
        },
        "aliases": {},
        "settings": {
        "number_of_shards": 16,
        "number_of_replicas": 2
        }
        }
```

2. Modify the template to specify all other fields that can appear in an HCP event under the properties section:

```
…
        "ip_src_addr": {
            "type": "ip"
        },
```

```
 "ip_src_port": {
        "type": "integer"
 },
 "action": {
        "type": "keyword"
 },
 "ciscotag": {
        "type": "keyword"
 }
 …
```

> **Note:** It is very important for the specified type to be the minimum size required for the field. For example, if you do not specify "int", the value would be auto-detected as "long" which will consume more system resources. If there is a string field, it is advised to specify it as "keyword" only and not "text". See the following link for full list of data types: https://www.elastic.co/guide/en/elasticsearch/reference/5.2/mapping-types.html. Duplication of fields should be avoided as it can lead to large performance impacts when indexing. Some consideration should also be given to the index.refresh_interval parameter in Ambari. This specifies the interval at which Elasticsearch creates a new segment. Increasing this value can improve indexing performance by allowing larger segments to flush and decreasing merge pressure.

Tune Random Access Indexing Core Storm Settings

You can set the number of Kafka spouts to match the number of Kafka partitions. You can also increase the number of workers and ackers to match the Storm nodes, unless the estimated throughput for the parser is very low.

Procedure

1. Set the parser Storm settings using the enrichment.properties file.

```
vi ~/elasticsearch.properties
```

```
##### Storm #####
indexing.workers=3
indexing.acker.executors=3
topology.worker.childopts=
topology.auto-credentials=[]
topology.max-spout.pending=

  ...

kafka.start=LATEST

  ...

##### Parallelism #####
kafka.spout.parallelism=9
```

2. Set the Kafka Offset Strategy to LATEST to allow the Kafka topic to be written to continuously during testing so when the parser is restarted, the topology will not be flooded with events.

```
kafka.start-LATEST
```

3. Alternatively, you can set the Kafka Offset Strategy to EARLIEST to determine the maximum throughput of the topology though you should set Max Spout Pending to avoid errors..

```
kafka.start-EARLIEST
```

4. Increase the hdfs.writer.parallelism values in increments based on the number of workers.

For example, in the previous example, the parameters could be incremented by 3.

```
##### Parallelism #####
kafka.spout.parallelism=9
enrichment.split.parallelism=
enrichment.stellar.parallelism=
enrichment.join.parallelism=18
threat.intel.split.parallelism=
threat.intel.stellar.parallelism=
threat-intel.join-parallelism-18
kafka.writer.parallelism=9
```

5. As you increase the enrichment.join.parallelism, threat.intel.join.parallelism, and kafka.writer.parallelism values, check the two Storm statistics, Capacity of the HDFS indexing component, the number of tuples acked in a 10-minute window, and the complete latency.

Topology stats

Window	Emitted	Transferred	Complete latency (ms)	Acked	Failed
10m 0s	0	0	0		
3h 0m 0s	0	0	0		
1d 0h 0m 0s	0	0	0		
All time	0	0	0		

Bolts (All time)

Search:

Id	Executors	Tasks	Emitted	Transferred	Capacity (last 10m)	Execute latency (ms)	Executed	Process latency (ms)	Acked	Failed	Error Host	Error Port	Last error	Error Time
hdfsIndexingBolt	18	18	0	0	0.000	0.035	14460	0.011	14480	0				
indexingErrorBolt	1	1	0	0	0.000	0.050	800	0.025	800	0				

For a given estimated throughput, the capacity should be no greater than ~0.800. This will allow for ~20% overhead should the number of incoming events spike above the estimated average. If the capacity is above this level, Parallelism and Num Tasks should be incremented and the topology restarted.

The number of acked tuples should be approximately equal to (Desired Throughput ×600) assuming the topology has been active for at least 11 - 12 minutes. If the number of acked tuples and the capacity of the topology are both low, there may not be enough Kafka partitions.

If the Storm UI is showing a capacity of ~0.800 or less, the Kafka consumer should be monitored to ensure that there is no significant lag or buildup of messages for the parser. The command below shows an example of how this can be monitored via the command line on a Kafka node:

```
cd /usr/hdp/current/kafka-broker/bin/

watch -n 2 ./kafka-run-class.sh kafka.tools.ConsumerOffsetChecker --
zookeeper
master01:2181 --topic indexing --group index-batch
```

```
Group           Topic            Pid Offset      logSize      Lag       Owner
indexing-batch  indexing         0   560         561          1         none
indexing-batch  indexing         1   605         606          1         none
indexing-batch  indexing         2   609         610          1         none
indexing-batch  indexing         3   606         607          1         none
indexing-batch  indexing         4   606         607          1         none
indexing-batch  indexing         5   608         609          1         none
indexing-batch  indexing         6   605         606          1         none
indexing-batch  indexing         7   604         605          1         none
indexing-batch  indexing         8   608         609          1         none
```

Tune Additional Random Access Indexing Storm Settings

After the number of executors has been determined and thoroughly tested, you can set or modify the last remaining Storm parameters.

Procedure

1. Based on the capacity you've seen during testing, reduce the overall number of ackers.

   Alternatively, you an leave a single acker per worker as it will ensure that there are no messages sent between Storm workers over the network interface.

2. Set the **Max Spout Pending** parameter such that the maximum number of unacked tuples in the topology is close to the Parser Executor capacity (for example, ~0.950).

   If this is the case then it can be assured that if there is a large spike in incoming events, the topology will not become overloaded. An example approach to determine this value would be to increase the producer events per second by a large amount and test various values for **Max Spout Pending**. The value can be set under the Storm settings of the relevant Parser.

   ```
   vi ~/enrichment.properties
   ```

   ```
   ##### Storm #####
   indexing.workers=3
   indexing.acker.executors=3
   topology.worker.childopts=
   topology.auto-credentials=[]
   topology.max-spout.pending=
   ```

3. Check the **Executor** capacity.

   The executor capacity should not exceed ~0.950. Assuming the number of events generated by the producer is far greater than the capacity of the Parser topology, capacity is the only value that needs to be monitored in the Storm UI.

4. If you need to increase the **Error Writer Num Executors** value, you can directly modify the Flux file and include the "parallelism" parameter under the appropriate Storm Bolt declarations.

   While the product has been rebranded to Cloudera Cybersecurity Platform (CCP), the directory name currently remains hcp.

   ```
   sudo vi /usr/hcp/current/metron/flux/indexing/batch/remote.yaml
   ```

   ```
   id: "indexingErrorOutputBolt"
   className: "org.apache.metron.writer.bolt.BulkMesageWriterBolt"
   constructorArgs:
       - "${kafka.zk}"
   configMethods:
       - name: "withMessageWriter"
                 args:
                       - ref: "KafkaWriter"
   parallelism: 3
   ```

   Generally, since a small number of errors is expected, the **Error Writer Num Executors** value does not need to be increased.