

Component Tuning Levers

Date of Publish: 2018-12-21



Contents

Component Tuning Levers Overview.....	3
Kafka Tuning.....	3
Storm Tuning.....	3
Parser Tuning.....	5
Enrichment Tuning.....	6
Modifying Split-Join Topology Enrichment Properties Using Ambari.....	6
Modifying Enrichment Properties Using Flux (Advanced).....	6
Modifying Unified Topology Enrichment Properties Using Ambari.....	7
Index Tuning.....	10
Modifying Index Parameters Using Ambari.....	11
Modifying Index Parameters Using Flux (Advanced).....	11

Component Tuning Levers Overview

There are a number of services that you can use to tune the performance of your Metron cluster. These services include Kafka, Storm, and HDFS. Within these services, you can modify parsers, enrichment, and indexing (Elasticsearch or Solr).

When you consider tuning your HCP architecture, it is important to note where you can modify settings. For example, Storm gives you the ability to independently set tasks in executors for parser topologies. This is important if you want to set the number of tasks higher than the number of executors to accommodate for future performance tuning and rebalancing without the need to bring down your topologies. However, for enrichment and indexing topologies, HCP uses Flux, and there is no method for specifying the number of tasks from the number of executors in Flux. By default, the number of tasks equals the number of executors.

The following lists the major properties for each service that you can modify to tune your cluster:

- Kafka
 - Number partitions
- Storm
 - Kafka spout
 - Polling frequency
 - Polling timeouts
 - Offset commit period
 - Max uncommitted offsets
 - Number workers (OS processes)
 - Number executors (threads in a process)
 - Number ackers
 - Max spout pending
 - Spout and bolt parallelism
- HDFS
 - Replication factor
- Indexing
 - Elasticsearch
 - Solr

Kafka Tuning

The main lever you can adjust to tune Kafka throughput is the number of partitions.

To determine the number of partitions required to attain your desired throughput, calculate the throughput for a single producer (p) and a single consumer (c), and then use that with the desired throughput (t) to roughly estimate the number of partitions to use. You would want at least $\max(t/p, t/c)$ partitions to attain the desired throughput.

Storm Tuning

There are several Storm properties you can adjust to tune your Storm topologies. Achieving the desired performance can be iterative and will take some trial and error.

Hortonworks recommends you start your tuning with the Storm topology defaults and smaller numbers in terms of parallelism. Then you can iteratively increase the values until you achieve your desired level of performance. Use the offset lag tool to verify your settings.

The following sections assume log type messages. However, if your data consists of emails which are much larger in size, then you should adjust your values accordingly.

Storm Topology Parallelism

To provide a uniform distribution to each machine and jvm process, you can modify the values for the number of tasks, executors, and workers properties. Start with small values and iteratively increase the values so you don't overwhelm you CPU with too many processes.

Usually your number of tasks is equal to the number of executors, which is the default in Storm. Flux does not provide a method to independently set the number of tasks, so for enrichments and indexing, which use Flux, num tasks are always equal to num executors.

You can change the number of workers in the Storm property topology.workers.

The following table lists the variables you can set to adjust the parallelism in a Storm topology and provides recommendations for their values:

Storm Topology Variables	Description	Value
num tasks	Tasks are instances of a given spout or bolt. Executors are threads in a process.	Set the number of tasks as a multiple of the number of executors.
num executors	Executors are threads in a process.	Set the number of executors as a multiple of the number of workers.
num workers	Workers are jvm processes.	Set the number of workers as a multiple of the number of machines

Maximum Number of Tuples

The topology.max.spout.pending setting sets the maximum number of tuples that can be in a field (for example, not yet acked) at any given time within your topology. You set this property as a form of back pressure to ensure that you don't flood your topology.

```
topology.max.spout.pending
```

Topology Acker Executors

The topology.ackers.executors setting specifies how many threads are dedicated to tuple acking. Set this setting to equal the number of partitions in your inbound Kafka topic.

```
topology.ackers.executors
```

Spout Recommended Defaults

As a general rule, it is optimal to set spout parallelism equal to the number of partitions used in your Kafka topic. Any greater parallelism will leave you with idle consumers because Kafka limits the maximum number of consumers to the number of partitions. This is important because Kafka has certain ordering guarantees for message delivery per partition that would not be possible if more than one consumer in a given consumer group is able to read from that partition.

You can modify the following spout settings in the spout-config.json file. However, if the spout default settings work for your system, you can omit these settings from the file. These default settings are based on recommendations from Storm and are provided in the Kafka spout itself.

```
{
  ...
  "spout.pollTimeoutMs" : 200,
  "spout.maxUncommittedOffsets" : 10000000,
  "spout.offsetCommitPeriodMs" : 30000
}
```

}

Parser Tuning

You can modify certain parser properties to tune your HCP 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" : 1000000 }
Parser bolt	Spout Config	<JSON_FILE>, e.g., { "spout.offsetCommitPeriodMs" : 30000 }
	Parser Num Tasks	-pnt,--parser_num_tasks <NUM_TASKS>
	Parser Parallelism	-pp,--parser_p <PARALLELISM_HINT>
	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](#).

Enrichment Tuning

Because all of the data is coming together in enrichments, you will probably need larger enrichment settings than your parallelism settings. Enrichment settings focus more on the compute workload than on the mapping workload in parsers or the IO driven workload in indexing. Enrichment makes significant use of caching for performance.

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.

The enrichment properties materialize as follows:

```
Ambari UI -> properties file -> Flux -> Storm
```

Modifying Split-Join Topology Enrichment Properties Using Ambari

You can modify various enrichment properties using Ambari.

To modify the enrichment properties, navigate to Ambari>Metron>Enrichment.

The following table lists the split-join topology enrichment properties you can modify in Ambari:

Category	Ambari Property Name
Storm topology config	enrichment_workers
	enrichment_acker_executors
	enrichment_topology_max_spout_pending
Kafka spout	enrichment_kafka_spout_parallelism
Enrichment splitter	enrichment_split_parallelism
Enrichment joiner	enrichment_join_parallelism
Threat intel splitter	threat_intel_split_parallelism
Threat intel joiner	threat_intel_join_parallelism

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

Category	Flux Property or Function	Flux Section Location
	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's session.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
```

Modifying Unified Topology Enrichment Properties Using Ambari

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

To modify the enrichment properties, navigate to Ambari>Metron>Enrichment.

The following list provides tuning guidelines for the unified topology 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

topology.worker.childopts

capacity of this bolt is too high, then increase the number of ackers.

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.

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.

	<p>If the throughput of the topology is too low, increase this value. This allows additional tuples to be enriched in parallel.</p> <p>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.</p>
threat.intel.parallelism	<p>The parallelism hint for the threat intel bolt. Defines the maximum number of executors within each worker dedicated to running the threat intel bolt.</p> <p>If the capacity of the threat intel bolt is high, increasing the parallelism will introduce additional executors to bring the bolt capacity down.</p> <p>If the throughput of the topology is too low, increase this value. This allows additional tuples to be enriched in parallel.</p> <p>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.</p>
kafka.writer.parallelism	<p>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.</p> <p>If the capacity of the output bolt is high, increasing the parallelism will introduce additional executors to bring the bolt capacity down.</p>
enrichment.cache.size	<p>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.</p> <p>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.</p> <p>Increasing the size of the cache may require that you increase the worker heap size using <code>`topology.worker.childopts`</code>.</p>
threat.intel.cache.size	<p>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.</p> <p>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.</p> <p>Increase the size of the cache to improve the rate of cache hits.</p>

enrichment.threadpool.size

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

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.

Index Tuning

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.

You can modify many performance tuning properties for indexing 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.

The indexing properties materialize as follows:

```
Ambari UI -> properties file -> Flux -> Storm
```

Modifying Index Parameters Using Ambari

You can modify various 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.

To modify the indexing properties, navigate to Ambari>Metron>Indexing.

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
	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

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
	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
```