HCP Tuning Guide 1

Hortonworks Cybersecurity Platform

Date of Publish: 2018-07-30

http://docs.hortonworks.com

Contents

Introduction to Tuning HCP

Tuning your Hortonworks Cybersecurity Platform (HCP) architecture can help maximize the performance of the Apache Metron Storm topologies.

In the simplest terms, HCP powered by Apache Metron is a streaming architecture created on top of Kafka and three main types of Storm topologies: parsers, enrichment, and indexing. Each parser has its own topology and there is also a highly performant, specialized spout-only topology for streaming PCAP data to HDFS.

The HCP architecture can be tuned almost exclusively using a few primary Storm and Kafka parameters along with a few Metron-specific options. You can think of the data flow as being similar to water flowing through a pipe, and the majority of these options assist in tweaking the various pipe widths in the system.

General Tuning Suggestions

Tuning Hortonworks Cybersecurity Platform (HCP) depends in large part on tuning three areas: Kafka, Storm, and indexing.

Indexing is where most of your turning issues are likely to occure because it is the most IO intensive.

The second area that needs tuning is parallelism in both Kafka and Storm. The performance of the Storm topology. and therefore the performance of Metron, degrades when it cannot ingest data fast enough to keep up with the data source. Therefore, much of Metron tuning focuses on adjusting the data throughput of the Storm topologies. For more information on tuning a Storm topology, see Tuning an Apache Storm Topology.

The third area that requires analysis and tuning is consumer lags on the key Kafka topics: enrichment, indexing, parser.

When tuning your Metron configuration, consider the following:

- Look at Elasticsearch and Solr tuning
- Assign small values for parallelism, and increase values incrementally
- Aim for an even balance across your topologies
- Check your system logs for the following:
    - Empty results - may indicate that your data is broken
    - Kafka - Consumer lags on key Kafka topics
    - Load average or system latency - a high load average might indicate underlying stress on the machine
    - Exceptions - Any exceptions shown in the Storm log or key topologies can indicate possible problems with underlying systems and data
- What topology do I want to tune?
- What is the capacity of Storm topology?

It is also important to consider the growth of your cluster and data flow. You might 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.

Component Tuning Levers

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.

For more information, see Confluent - How to choose the number of topics/partitions in a Kafka cluster?.

Related Information

How to Choose the Number of Topic or Partitions in a Kafka Cluster

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
}
```

Related Information

What is the Task in Storm Parallelism

Understanding the Parallelism of a Storm Topology

Reading and Understanding the Storm UI

Parser Tuning

You can modify certain parser properties to tune your HCP architecture using the Management module. Modifying properties using the Management module 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 module. 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 module:

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>
	Parser Parallelism	-pp,--parser_p <PARALLELISM_HINT>

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

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 enrichments 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 Enrichment Properties Using Ambari

You can modify various enrichment properties using Ambari.

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

> **Note:** Many of the following settings are relevant only to the split-join topology.

The following table lists the 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
	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
```

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 commiting 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
```

Use Case Specific Tuning Suggestions

The following discussion outlines a specific tuning exercise we went through for driving 1 Gbps of traffic through a Metron cluster running with 4 Kafka brokers and 4 Storm Supervisors.

General machine specs:

- 10 GB network cards
- 256 GB memory
- 12 disks
- 32 cores

Performance Monitoring Tools

Before we get to tuning our cluster, it helps to describe what we might actually want to monitor as well as any potential pain points.

Prior to switching over to the new Storm Kafka client, which leverages the new Kafka consumer API under the hood, offsets were stored in ZooKeeper. While the broker hosts are still stored in ZooKeeper, this is no longer true for the offsets which are now stored in Kafka itself. This is a configurable option, and you may switch back to ZooKeeper if you choose, but Metron is currently using the new defaults. With this in mind, there are some useful tools that come with Storm and Kafka that we can use to monitor our topologies.

Tooling

You can use the Storm and Kafka tools to monitor your topologies.

Kafka

- Consumer group offset lag viewer
- There is a GUI tool to make creating, modifying, and generally managing your Kafka topics a bit easier - see kafka-manager
- Console consumer - useful for quickly verifying topic contents

Storm

For more information on the Storm user interface, see Reading and Understanding the Storm UI.

View Kafka Offset Lags Example

You can use the Kafka consumer group offset lag viewer to monitor the delta calculations between the current and end offset for a partition.

Procedure

1. Set up some environment variables.

```
export BROKERLIST  your broker comma-delimated list of host:ports>
export ZOOKEEPER  your zookeeper comma-delimated list of host:ports>
export KAFKA_HOME  kafka home dir>
export METRON_HOME  your metron home>
export HDP_HOME  your HDP home>
```

2. If you have Kerberos enabled, set up the security protocol.

```
$ cat /tmp/consumergroup.config
security.protocol=SASL_PLAINTEXT
```

3. Enter the following command to display a table containing offsets for all partitions and consumers associated with a running topology's consumer group:

   ${KAFKA_HOME}/bin/kafka-consumer-groups.sh \ --command-config=/tmp/consumergroup.config \ --describe \ --group enrichments \ --bootstrap-server $BROKERLIST \ --new-consumer

   The command displays the following table:

```
GROUP                          TOPIC              PARTITION  CURRENT-
OFFSET  LOG-END-OFFSET  LAG             OWNER
enrichments                    enrichments        9          29746066
   29746067        1              consumer-2_/xxx.xxx.xxx.xxx
enrichments                    enrichments        3          29754325
   29754326        1              consumer-1_/xxx.xxx.xxx.xxx
enrichments                    enrichments        43         29754331
   29754332        1              consumer-6_/xxx.xxx.xxx.xxx
...
```

   Note: Output displays only when the topology is running because the consumer groups only exist while consumers in the spouts are up and running.

The LAG column lists the current delta calculation between the current and end offset for the partition. The column value indicates how close you are to keeping up with incoming data. It also indicates whether there are any problems with specific consumers getting stuck.

4. To watch the offsets and lags change over time, add a watch command and set the refresh rate to 10 seconds:

```
watch -n 10 -d ${KAFKA_HOME}/bin/kafka-consumer-groups.sh \
    --command-config=/tmp/consumergroup.config \
    --describe \
    --group enrichments \
    --bootstrap-server $BROKERLIST \
    --new-consumer
```

The watch command runs every 10 seconds and refreshes the screen with new information. The command also highlights the differences from the current output and the last output screens.

Parser Tuning Example

We'll be using the Bro sensor in this parser tuning example.

We started with a single partition for the inbound Kafka topics and eventually worked our way up to 48 partitions. And we're using the following pending value, as shown below. The default is 'null' which would result in no limit.

storm-bro.config

```
{
    ...
    "topology.max.spout.pending" : 2000
    ...
}
```

And the following default spout settings. Again, this can be omitted entirely since we are using the defaults.

spout-bro.config

```
{
    ...

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

And we ran our Bro parser topology with the following options. We did not need to fully match the number of Kafka partitions with our parallelism in this case, though you could certainly do so if necessary. Notice that we only needed 1 worker.

```
/usr/metron/0.4.0/bin/start_parser_topology.sh -k $BROKERLIST -z $ZOOKEEPER
-s bro -ksp SASL_PLAINTEXT
    -ot enrichments
    -e ~metron/.storm/storm-bro.config \
    -esc ~/.storm/spout-bro.config \
    -sp 24 \
    -snt 24 \
    -nw 1 \
    -pnt 24 \
    -pp 24 \
```

From the usage docs, here are the options we've used.

```
-e,--extra_topology_options (JSON_FILE)        Extra options in the form of
 a JSON file with a map
                                                for content.
-esc,--extra_kafka_spout_config (JSON_FILE)    Extra spout config options in
 the form of a JSON file
                                                with a map for content.
                                                Possible keys are:
                                                retryDelayMaxMs,retryDelay

 Multiplier,retryInitialDelayMs,stateUpdateIntervalMs,

 bufferSizeBytes,fetchMaxWait,fetchSizeBytes,maxOffset

 Behind,metricsTimeBucketSizeInSecs,socketTimeoutMs
-sp,--spout_p (SPOUT_PARALLELISM_HINT)         Spout Parallelism Hint
-snt,--spout_num_tasks (NUM_TASKS)             Spout Num Tasks
-nw,--num_workers (NUM_WORKERS)                Number of Workers
-pnt,--parser_num_tasks (NUM_TASKS)            Parser Num Tasks
-pp,--parser_p (PARALLELISM_HINT)              Parser Parallelism Hint
```

Enrichment Tuning Example

We landed on the same number of partitions for enrichment and indexing as we did for bro - 48.

For configuring Storm, there is a flux file and properties file that we modified. Here are the settings we changed for Bro in Flux. +Note that the main Metron-specific option we've changed to accommodate the desired rate of data throughput is max cache size in the join bolts.

More information on Flux can be found here - https://storm.apache.org/releases/1.1.0/flux.html

general storm settings

```
topology.workers: 8
topology.acker.executors: 48
topology.max.spout.pending: 2000
```

Spout and Bolt Settings

```
kafkaSpout
    parallelism=48
    session.timeout.ms=29999
    enable.auto.commit=false
    setPollTimeoutMs=200
    setMaxUncommittedOffsets=10000000
    setOffsetCommitPeriodMs=30000
enrichmentSplitBolt
    parallelism=4
enrichmentJoinBolt
    parallelism=8
    withMaxCacheSize=200000
    withMaxTimeRetain=10
threatIntelSplitBolt
    parallelism=4
threatIntelJoinBolt
    parallelism=4
```

```
     withMaxCacheSize=200000
     withMaxTimeRetain=10
 outputBolt
     parallelism=48
```

Indexing (HDFS) Tuning

There are 48 partitions set for the indexing partition, per the previous enrichment exercise.

These are the batch size settings for the Bro index.

```
cat ${METRON_HOME}/config/zookeeper/indexing/bro.json
{
"hdfs" : {
"index": "bro",
    "batchSize": 50,
    "enabled" : true
  }...
}
```

And here are the settings we used for the indexing topology

General storm settings

```
topology.workers: 4
topology.acker.executors: 24
topology.max.spout.pending: 2000
```

Spout and Bolt Settings

```
hdfsSyncPolicy
    org.apache.storm.hdfs.bolt.sync.CountSyncPolicy
    constructor arg=100000
hdfsRotationPolicy
    bolt.hdfs.rotation.policy.units=DAYS
    bolt.hdfs.rotation.policy.count=1
kafkaSpout
    parallelism: 24
    session.timeout.ms=29999
    enable.auto.commit=false
    setPollTimeoutMs=200
    setMaxUncommittedOffsets=10000000
    setOffsetCommitPeriodMs=30000
hdfsIndexingBolt
    parallelism: 24
```

PCAP Tuning Example

PCAP is a specialized topology that is a Spout-only topology. Both Kafka topic consumption and HDFS writing is done within a spout to avoid the additional network hop required if using an additional bolt.

General Storm topology properties

```
topology.workers=16
topology.ackers.executors: 0

__Spout and Bolt properties__

kafkaSpout
```

```
    parallelism: 128
    poll.timeout.ms=100
    offset.commit.period.ms=30000
    session.timeout.ms=39000
    max.uncommitted.offsets=200000000
    max.poll.interval.ms=10
    max.poll.records=200000
    receive.buffer.bytes=431072
    max.partition.fetch.bytes=10000000
    enable.auto.commit=false
    setMaxUncommittedOffsets=20000000
    setOffsetCommitPeriodMs=30000

writerConfig
    withNumPackets=1265625
    withMaxTimeMS=0
    withReplicationFactor=1
    withSyncEvery=80000
    withHDFSConfig
        io.file.buffer.size=1000000
        dfs.blocksize=1073741824
```

Issues

You can run into issues when you tune your system.

```
__Error__


org.apache.kafka.clients.consumer.CommitFailedException: Commit cannot be
 completed since the group has already rebalanced and assigned
the partitions to another member. This means that the time
 between subsequent calls to poll() was longer than the configured
 session.timeout.ms,
which typically implies that the poll loop is spending too much time message
 processing. You can address this either by increasing the
session timeout or by reducing the maximum size of batches returned in
 poll() with max.poll.records
```

Suggestions

This implies that the spout hasn't been given enough time between polls before committing the offsets. In other words, the amount of time taken to process the messages is greater than the timeout window. In order to fix this, you can improve message throughput by modifying the options outlined above, increasing the poll timeout, or both.