Creating Profiles

Date of Publish: 2018-12-21

https://docs.hortonworks.com/

Contents

Introduction to HCP Analytics

Data Scientists are frequently responsible for performing data science life cycle activities, including training, evaluating, and scoring analytical models. HCP provides the ability to create profiles and models, analyze data using statistical and mathematical functions and Apache Zeppelin, and create runbooks for SOC analysts and investigators.

Using Profilers

The Profiler is a feature extraction mechanism that can generate a profile that describes the behavior of an entity. An entity can be a server, user, subnet, or application.

You can use any field contained within a message to generate a profile. A profile can even be produced by combining fields that originate in different data sources. You can transform the data used in a profile by leveraging the Stellar language.

Once you generate a profile defining what normal behavior looks like, you can build models that identify anomalous behavior. To identify anomalous behavior, you can summarize the streaming telemetry data consumed by HCP over sliding windows. You apply a summary statistic to the data received within a given window. Collecting this summary across many windows results in a time series that is useful for analysis.

The Profiler is automatically installed and started when you install HCP through Ambari.

HCP provides two types of profilers:

Streaming Profiler	Allows you to create profiles based on the stream of telemetry being captured, enriched, triaged, and indexed by HCP. This does not allow you to create a profile based on telemetry that was captured in the past.
Batch Profiler	Allows you to generate a profile using archived telemetry.

Install and Configure the Profiler

The Profiler is automatically installed and started when you install HCP through Ambari.

About this task

The configuration for the Profiler topology is stored in ZooKeeper at /metron/topology/profiler. These properties also exist in the default installation of HCP at $METRON_HOME/config/zookeeper/profiler.json. You can change these values two ways: with Ambari or on disk and then uploaded to ZooKeeper using $METRON_HOME/bin/zk_load_configs.sh.

Procedure

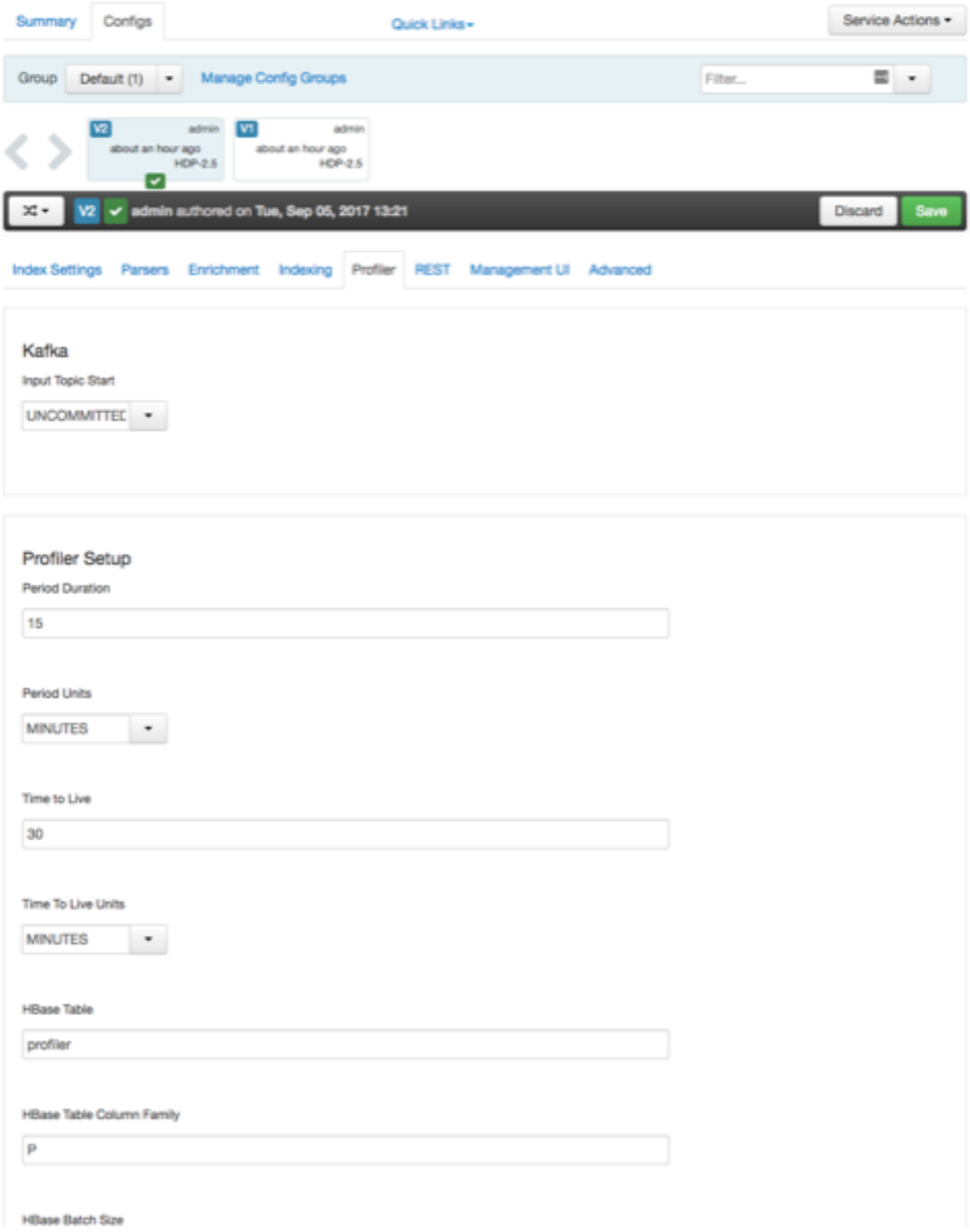
1. Display the Ambari user interface and click the **Services** tab.

   Note: You might need to work with your Platform Engineer to modify Profiler values.

2. Click **Metron** in the list of services, then click the **Configs** tab.
3. Click the **Profiler** tab.

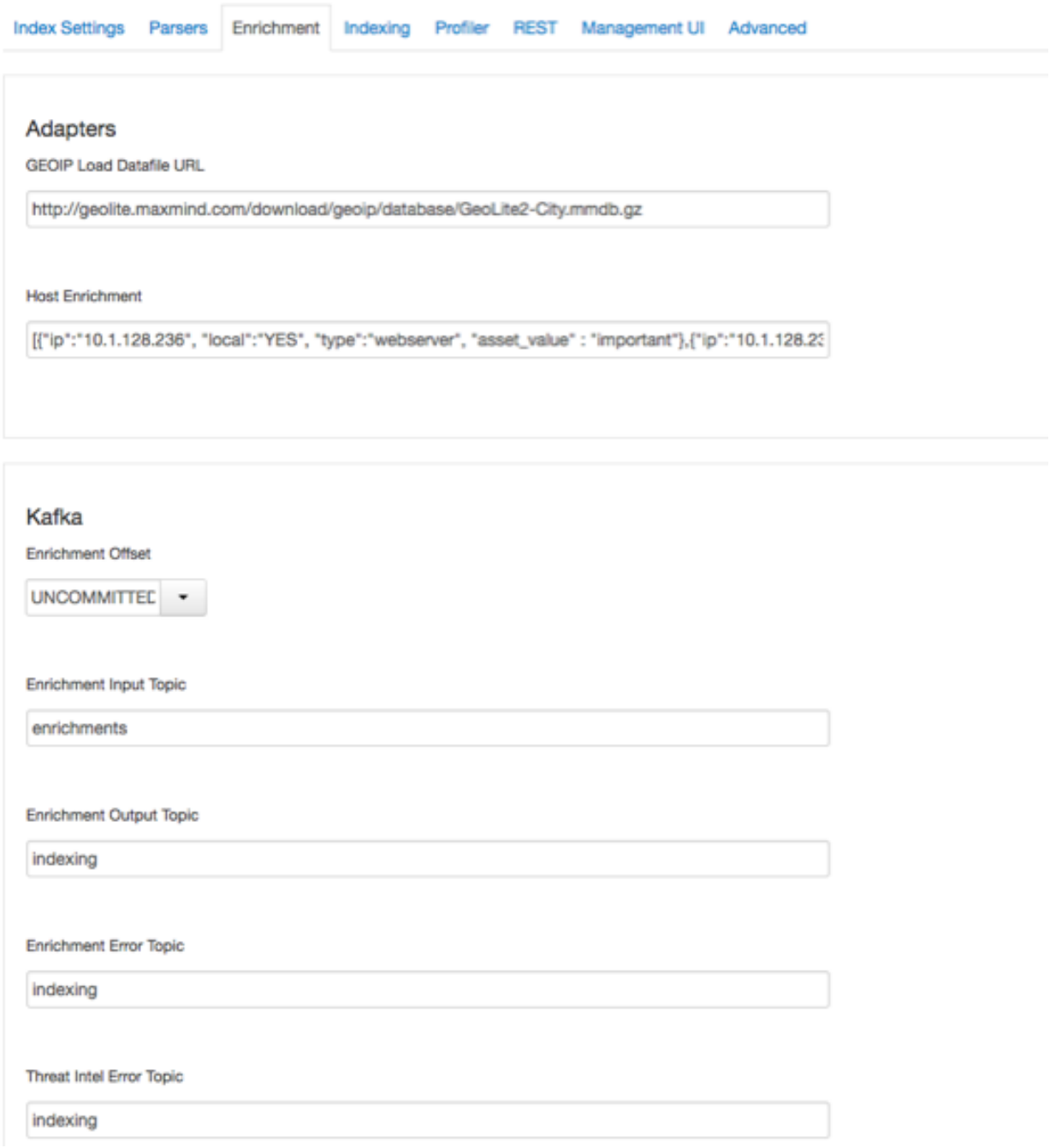
   Ambari displays a list of Profiler properties that you can use to configure Profiler.



4. Use these properties to configure the Profiler, then click the **Save** button near the top of the window.

   The profiler input topic is bound to the enrichment output topic. If that enrichment output topic is changed, then the profiler will restart as well as the enrichment topology.

   Enrichment Output Topic



Running the Profiler

The Profiler is automatically started when you install HCP. However, you can also manually stop or restart the Profiler.

Procedure

1. Display the Ambari user interface.
2. Select the **Services** tab, then select **Metron** from the list of services.
3. Make sure you have selected the **Summary**, tab then select **Metron Profiler**.
   Metron displays a complete list of Metron components.

4. Select the pull down menu next to **Metron Profiler / Metron** and select the appropriate status:
   - Restart
   - Stop
   - Turn on Maintenance Mode

Streaming Profiler

A streaming profile creates a profile based on telemetry that is currently being captured, enriched, triaged, and indexed by HCP. Streaming profiles can be used to understand real-time behaviors and trends. You can use the streaming profiler and the batch profiler to gather and understand both current and historical behaviors and trends. This information can be used to determine if the profiler feature set matches reality and has predictive value for model building.

Create a Streaming Profile

Create a streaming profile when you want to create a profile based on telemetry that is currently being captured, enriched, triaged, and indexed by HCP.

Before you begin

Ensure that the PROFILE_GET client is correctly configured to match your desired Profile configuration before using it to read that Profile.

Procedure

1. Create a streaming profile definition by editing $METRON_HOME/config/zookeeper/profiler.json.

   As an example, in the editor copy/paste the basic "Hello, World" profile below.

```
{
  "profiles": [
    {
      "profile": "hello-world",
      "foreach": "ip_src_addr",
      "onlyif":  "exists(ip_src_addr)",
      "init": {
         "count": "0"
       },
      "update": {
         "count": "count + 1"
       },
      "result":  "count"
    }
  ]
}
```

   You can also include the timestampField to:

   - List the system time, which is the time at which you are processing the data.
   - List the event time, which is the time contained in the data itself.

2. Upload the profile definition to ZooKeeper:

```
source /etc/default/metron
cd $METRON_HOME
bin/zk_load_configs.sh -m PUSH -i config/zookeeper/ -z $ZOOKEEPER
```

You can validate your upload by reading back the Metron configuration from ZooKeeper using the same script. The result should look-like the following.

```
bin/zk_load_configs.sh -m DUMP -z $ZOOKEEPER
...
PROFILER Config: profiler
{
  "profiles": [
    {
      "profile": "hello-world",
      "foreach": "ip_src_addr",
      "onlyif":  "exists(ip_src_addr)",
      "init":    { "count": "0" },
      "update":  { "count": "count + 1" },
      "result":  "count"
    }
  ]
}
```

3. Ensure that test messages are being sent to the Profiler's input topic in Kafka.

   The Profiler will consume messages from the input topic defined in the Profiler's configuration (see *Configure the Streaming Profiler*). By default this is the indexing topic.

4. Check the HBase table to validate that the Profiler is writing the profile.

   Remember that the Profiler is flushing the profile every 15 minutes. You will need to wait at least this long to start seeing profile data in HBase.

   ```
   /usr/hdp/current/hbase-client/bin/hbase shell
   hbase(main):001:0> count 'profiler'
   ```

5. Use the Profiler Client to read the profile data.

   The following PROFILE_GET command reads the data written by the hello-world profile. This assumes that 10.0.0.1 is one of the values for ip_src_addr contained within the telemetry consumed by the Profiler.

   ```
   source /etc/default/metron
   bin/stellar -z $ZOOKEEPER
   [Stellar]>>> PROFILE_GET( "hello-world", "10.0.0.1", PROFILE_FIXED(30,
    "MINUTES"))
   [451, 448]
   ```

   This result indicates that over the past 30 minutes, the Profiler stored two values related to the source IP address "10.0.0.1". In the first 15 minute period, the IP 10.0.0.1 was seen in 451 telemetry messages. In the second 15 minute period, the same IP was seen in 448 telemetry messages.

   Enter quit to exit Stellar.

Configure the Streaming Profiler

You can customize the streaming profiler to specify various properties available in the profiler.properties file, such as the name and output of the Kafka topic, the duration of the profile period, and the name of the HBase table to which the profiles are written.

Procedure

Modify the streaming profiler's properties located at $METRON_HOME/config/profiler.properties to customize them for your profiler needs.

See *Streaming Profiler Properties* for more information on these properties.

Run the Streaming Profiler

HCP provides a script called start_profiler_topology.sh to simplify running the Streaming Profiler.

Before you begin

The start_profiler_topology.sh script assumes the following:

- The script builds the profiles defined in $METRON_HOME/config/zookeeper/profiler.json.
- The properties defined in $METRON_HOME/config/profiler.properties are passed to the profiler.

Procedure

Start the streaming profile by entering the following:

```
source /etc/default/metron
cd $METRON_HOME
bin/start_profiler_topology.sh
```

Streaming Profiler Properties

Use the profiler properties to configure the streaming profiler.

Table 1: Profiler Properties

Ambari Configs Field.	Settings	Description
Profiler Setup		
Period Duration	profiler.period.duration	The duration of each profile period. This value should be defined along with profiler.period.duration.units.
Period Units	profiler.period.units	The units is used to specify the profiler.period.duration. This value should be define along with profiler.period.duration.
Window Duration	profiler.window.duration	The duration of each profile window.
Window Units	profiler.window.duration.units	The units used to specify the profiler.window.duration.
Time to Live	profiler.ttl	If a message has not been applied to a Profile in this period of time, the Profile will be terminated and its resources will be cleaned up. This value should be defined along with profiler.ttl.units. This time-to-live does not affect the persisted Profile data in HBase. It only affects the state stored in memory during the execution of the latest profile period. This state will be deleted if the time.to.live is exceeded.
Time to Live Units	profiler.ttl.units	The units used to specify the profiler.ttl.
Window Time Lag	profiler.window.log	The maximum time lag for timestamps.
Window Lag Units	profiler.window.log.units	The units used to specify the profiler.window.lag.
Max Routes Per Bolt	profiler.max.routes.per.bolt	The maximum number of routes that will be maintained by the bolt. After this value is exceeded, lesser used routes will be evicted from the internal cache.
Kafka		
Input Topic Start	profiler.kafka.start	One of EARLIEST, LATEST, UNCOMMITTED_EARLIEST, UNCOMMITTED_LATEST
Kafka Writer Batch Size	profiler.writer.batchSize	The number of records to batch when writing to Kakfa.

Kafka Writer Batch Timeout	profiler.writer.batchTimeout	The timeout in ms for batching when writing to Kakfa.
Storm		
topology.worker.childopts	profiler.topology.worker.childopts	Extra topology child opts for the storm opts.
Number of Workers	profiler.topology.workers	The profiler storm topology storm workers
Number of Acker Executors	profiler.acker.executors	The profiler storm topology acker executors
Profiler Topology Message Timeout	topology.message.timeout.secs	Maximum amount of time given to the topology to fully process a tuple tree from the core-storm API, or a batch from the Trident API, emitted by a spout. If the message is not acked within this time frame, Storm fails the operation on the spout. The default is 30 seconds.
Spout Max Pending Tuples	topology.max.spout.pending	Maximum number of messages that can be pending in a spout at any time. The default is null (no limit).
HBase		
HBase Table	profiler.hbase.table	The name of the HBase table the profiler is written to. The profiler expects that the table exists and is writable.
HBase Table Column Family	profiler.hbase.cf	The column family used to store profile data in HBase.
HBase Batch Size	profiler.hbase.batch	The number of puts that are written to HBase in a single batch.
HBase Flush Interval	profiler.hbase.flush.interval	The maximum number of seconds between batch writes to HBase.
N/A	profiler.writer.batchSize	The size of the batch that is written to Kafka at once. Defaults to 15 (size of 1 disables batching).
N/A	profiler.writer.batchTimeout	The timeout after which a batch will be flushed even if batchSize has not been met. Optional. If unspecified or set to 0, it defaults to a system-determined duration which is a fraction of the Storm parameter topology.message.timeout.secs. Ignored if batchsize is 1 because this disables batching.

Troubleshoot Streaming Profiles By Using Stellar

Troubleshooting issues when programming against a live stream of data can be difficult. The Stellar REPL (an interactive top level or language shell) is a powerful tool to help work out the kinds of enrichments and transformations that are needed. The Stellar REPL can also be used to help when developing profiles for the profiler.

About this task

Follow these steps in the Stellar REPL to see how it can be used to help create profiles.

Procedure

1. Launch the Stellar REPL.

```
$METRON_HOME/bin/stellar
Stellar, Go!
[Stellar]>>>
```

2. Ensure the following functions are accessible.

```
[Stellar]>>> %functions PROFILER
PROFILER_APPLY, PROFILER_FLUSH, PROFILER_INIT
```

3. Use the SHELL_EDIT function to create a simple hello-world profile that will count the number of messages for each ip_src_addr. The SHELL_EDIT function will open an editor into which you can add the following profiler configuration.

```
[Stellar]>>> conf := SHELL_EDIT()
[Stellar]>>> conf
{
  "profiles": [
    {
      "profile": "hello-world",
      "foreach": "ip_src_addr",
      "init":    { "count": "0" },
      "update":  { "count": "count + 1" },
      "result":  "count"
    }
  ]
}
```

   You can also include the timestampField to:

   - List the system time, which is the time at which you are processing the data.
   - List the event time, which is the time contained in the data itself.

4. Create the profile execution environment.

   The profiler will output the number of profiles that have been defined, the number of messages that have been applied, and the number of routes that have been followed. A route is defined when a message is applied to a specific profile.

   - If a message is not needed by any profile, then there are no routes.
   - If a message is needed by one profile, then one route has been followed.
   - If a message is needed by two profiles, then two routes have been followed.

```
[Stellar]>>> profiler := PROFILER_INIT(conf)
[Stellar]>>> profiler
Profiler{1 profile(s), 0 messages(s), 0 route(s)}
```

5. Create a message to simulate the type of telemetry that you expect to be profiled.

   This message can be as simple or complex as you like. For the hello-world profile, all you need is a message containing an ip_src_addr field.

```
[Stellar]>>> msg := SHELL_EDIT()
[Stellar]>>> msg
{
 "ip_src_addr": "10.0.0.1"
}
```

6. Apply some telemetry messages to your profiles.

```
[Stellar]>>> PROFILER_APPLY(msg, profiler)
Profiler{1 profile(s), 1 messages(s), 1 route(s)}

[Stellar]>>> PROFILER_APPLY(msg, profiler)
Profiler{1 profile(s), 2 messages(s), 2 route(s)}

[Stellar]>>> PROFILER_APPLY(msg, profiler)
Profiler{1 profile(s), 3 messages(s), 3 route(s)}
```

7. Flush the profiler:

```
[Stellar]>>> values := PROFILER_FLUSH(profiler)
[Stellar]>>> values
[{period={duration=900000, period=1669628, start=1502665200000,
 end=1502666100000},
profile=hello-world, groups=[], value=3, entity=10.0.0.1}]
```

   A flush occurs in the profiler every 15 minutes. The result is a list of profile measurements. Each measurement is a map containing detailed information about the profile data that has been generated. The value field is written to HBase when running the profiler in either Storm or Spark.

   There will always be one measurement for each profile, entity pair. This profile counts the number of messages by IP source address. Notice that the value is 3 for the entity 10.0.0.1 because we applied 3 messages with an ip_src_addr of 10.0.0.1.

8. In addition to testing with mock data, you can also apply real, live telemetry to your profile.

   The following example extracts 10 messages of live, enriched telemetry to test your profile(s):

```
[Stellar]>>> msgs := KAFKA_GET("indexing", 10)
[Stellar]>>> LENGTH(msgs)
10
```

   This can be useful to test your profile against the complexities that exist in real data.

9. Apply the 10 messages to your profile:

   [Stellar]>>> PROFILER_APPLY(msgs, profiler) Profiler{1 profile(s), 10 messages(s), 10 route(s)}

What to do next

After you are satisfied with the data being generated by the profile, then use the profile against your live stream of telemetry being captured by HCP.

Streaming Profile Examples

You can use the streaming profiler examples to better understand the functionality provided by the profiler. Each example shows the configuration that would be required to generate the profile.

These examples assume a fictitious input message stream that looks something like the following:

```
{
  "ip_src_addr": "10.0.0.1",
  "protocol": "HTTPS",
  "length": "10",
  "bytes_in": "234"
},
{
  "ip_src_addr": "10.0.0.2",
  "protocol": "HTTP",
  "length": "20",
  "bytes_in": "390"
},
{
  "ip_src_addr": "10.0.0.3",
  "protocol": "DNS",
  "length": "30",
  "bytes_in": "560"
}
```

Example 1

The total number of bytes of HTTP data for each host. The following configuration would be used to generate this profile.

```
{
  "profiles": [
    {
      "profile": "example1",
      "foreach": "ip_src_addr",
      "onlyif": "protocol == 'HTTP'",
      "init": {
        "total_bytes": 0.0
      },
      "update": {
        "total_bytes": "total_bytes + bytes_in"
      },
      "result": "total_bytes",
      "expires": 30
    }
  ]
}
```

This creates a profile with the following parameters:

- Named ‘example1’
- That for each IP source address
- Only if the 'protocol' field equals 'HTTP'
- Initializes a counter ‘total_bytes’ to zero
- Adds to ‘total_bytes’ the value of the message's ‘bytes_in’ field
- Returns ‘total_bytes’ as the result
- The profile data will expire in 30 days

Example 2

The ratio of DNS traffic to HTTP traffic for each host. The following configuration would be used to generate this profile.

```
{
  "profiles": [
    {
      "profile": "example2",
      "foreach": "ip_src_addr",
      "onlyif": "protocol == 'DNS' or protocol == 'HTTP'",
      "init": {
        "num_dns": 1.0,
        "num_http": 1.0
      },
      "update": {
        "num_dns": "num_dns + (if protocol == 'DNS' then 1 else 0)",
        "num_http": "num_http + (if protocol == 'HTTP' then 1 else 0)"
      },
      "result": "num_dns / num_http"
    }
  ]
}
```

This creates a profile with the following parameters:

- Named ‘example2’
- That for each IP source address
- Only if the 'protocol' field equals 'HTTP' or 'DNS'
- Accumulates the number of DNS requests

- Accumulates the number of HTTP requests
- Returns the ratio of these as the result

Example 3

The average of the length field of HTTP traffic. The following configuration would be used to generate this profile.

```
{
  "profiles": [
    {
      "profile": "example3",
      "foreach": "ip_src_addr",
      "onlyif": "protocol == 'HTTP'",
      "update": { "s": "STATS_ADD(s, length)" },
      "result": "STATS_MEAN(s)"
    }
  ]
}
```

This creates a profile with the following parameters:

- Named ‘example3’
- That for each IP source address
- Only if the 'protocol' field is 'HTTP'
- Adds the length field from each message
- Calculates the average as the result

Example 4

It is important to note that the profiler can persist any serializable Object, not just numeric values. An alternative to the previous example could take advantage of this.

Instead of storing the mean of the length, the profile could store a more generic summary of the length. This summary can then be used at a later time to calculate the mean, min, max, percentiles, or any other sensible metric. This provides a much greater degree of flexibility.

```
{
  "profiles": [
    {
      "profile": "example4",
      "foreach": "ip_src_addr",
      "onlyif": "protocol == 'HTTP'",
      "update": { "s": "STATS_ADD(s, length)" },
      "result": "s"
    }
  ]
}
```

The following Stellar REPL session shows how you might use this summary to calculate different metrics with the same underlying profile data.

Retrieve the last 30 minutes of profile measurements for a specific host.

```
$ bin/stellar -z node1:2181

[Stellar]>>> stats := PROFILE_GET("example4", "10.0.0.1", PROFILE_FIXED(30,
 "MINUTES"))
[Stellar]>>> stats
[org.apache.metron.common.math.stats.OnlineStatisticsProvider@79fe4ab9, ...]
```

Calculate different metrics with the same profile data.

```
[Stellar]>>> STATS_MEAN( GET_FIRST( stats))
15979.0625

[Stellar]>>> STATS_PERCENTILE( GET_FIRST(stats), 90)
30310.958
```

Merge all of the profile measurements over the past 30 minutes into a single summary and calculate the 90th percentile.

```
[Stellar]>>> merged := STATS_MERGE( stats)
[Stellar]>>> STATS_PERCENTILE(merged, 90)
29810.992
```

Batch Profiler

A batch profile creates a profile based on telemetry that was captured in the past. This is sometimes referrred to as profile seeding or backfilling. Batch profiles can be used to understand the historical behaviors and trends of a profile to determine if the profile has predictive value for model building. You can use the streaming profiler and the batch profiler to gather and understand both current and historical behaviors and trends. This information can be used to determine if the profiler feature set matches reality and has predictive value for model building.

Create a Batch Profile

Create a batch profile when you want to create a profile based on telemetry that was captured in the past. Batch profiles can be used to understand the historical behaviors and trends.

Procedure

1. Create a profile definition by editing $METRON_HOME/config/zookeeper/profiler.json as follows:

```
cat $METRON_HOME/config/zookeeper/profiler.json
{
  "profiles": [
    {
      "profile": "hello-world",
      "foreach": "'global'",
      "init":    { "count": "0" },
      "update":  { "count": "count + 1" },
      "result":  "count"
    }
  ],
  "timestampField": "timestamp"
}
```

   If you have not previously created a profile definition, you will need to create the profiler.json file.

   Note: All of the properties listed above including the timestampField are required in the profiler definition.

2. Upload the profile definition to ZooKeeper:

```
source /etc/default/metron
cd $METRON_HOME
bin/zk_load_configs.sh -m PUSH -i config/zookeeper/ -z $ZOOKEEPER
```

You can validate your upload by reading back the HCP configuration from ZooKeeper using the same script:

```
bin/zk_load_configs.sh -m DUMP -z $ZOOKEEPER
...
PROFILER Config: profiler
{
  "profiles": [
    {
      "profile": "hello-world",
      "onlyif":  "exists(ip_src_addr)",
      "foreach": "ip_src_addr",
      "init":    { "count": "0" },
      "update":  { "count": "count + 1" },
      "result":  "count"
    }
  ]
}
```

3. Ensure that you have archived telemetry data available for the batch profiler to consume.

   By default, HCP stores this in HDFS at /apps/metron/indexing/indexed/*/*.

   ```
   hdfs dfs -cat /apps/metron/indexing/indexed/*/* | wc -l
   ```

4. Check the HBase table to validate that the Profiler is writing the profile.

   Remember that the Profiler is flushing the profile every 15 minutes. You will need to wait at least this long to start seeing profile data in HBase.

   ```
   /usr/hdp/current/hbase-client/bin/hbase shell
   hbase(main):001:0> count 'profiler'
   ```

5. Review the batch profiler's properties located at $METRON_HOME/config/batch-profiler.properties to ensure that the properties are set appropriately for the batch profiler you want to run.

   See *Batch Profiler Properties* for more information on these properties.

6. If you want to run DEBUG logging for the profiler, edit the log4j properties file that resides in $SPARK_HOME/config:

   ```
   log4j.logger.org.apache.metron.profiler.spark=DEBUG
   ```

   If the log4j file does not exist, you can create one.

7. Run the batch profiler.

   ```
   source /etc/default/metron
   cd $METRON_HOME
   $METRON_HOME/bin/start_batch_profiler.sh
   ```

What to do next

Query for the profile data using the Profiler Client.

Run the Batch Profiler

HCP provides a script called start_batch_profiler.sh to simplify running the batch profiler.

Before you begin

The start_batch_profiler.sh script assumes the following:

- The script builds the profiles defined in $METRON_HOME/config/zookeeper/profiler.json.
- The properties defined in $METRON_HOME/config/batch-profiler.properties are passed to both the profiler and Spark. You can define both Spark and profiler properties in this same file.

- Spark is installed at /usr/hdp/current/spark-client. This can be overridden if you define an environment variable called SPARK_HOME prior to executing the script.

Procedure

Start the batch profile by entering the following:

```
source /etc/default/metron
cd $METRON_HOME
bin/start_batch_profiler.sh
```

Run the Batch Profiler in Advanced Mode

As an alternative to using the start_batch_profiler.sh you can run the batch profiler in advanced mode. Running the batch profiler in advanced mode allows you to specify certain arguments to customize the profiler.

Procedure

Start the batch profile by entering the following:

```
${SPARK_HOME}/bin/spark-submit \
    --class org.apache.metron.profiler.spark.cli.BatchProfilerCLI \
    --properties-file ${SPARK_PROPS_FILE} \
    ${METRON_HOME}/lib/metron-profiler-spark-*.jar \
    --config ${PROFILER_PROPS_FILE} \
    --profiles ${PROFILES_FILE}
```

The batch profiler accepts the following arguments when run from the command line. All arguments following the profiler jar are passed to the profiler. All argument preceeding the profiler jar are passed to Spark.

-p, --profiles	The path to a file containing the profile definition in JSON.
-c, --config	The path to a file containing key-value properties for the profiler. This file contains the properties described in Batch Profiler Properties.
-g, --globals	The path to a file containing key-value properties that define the global properties. You can use this property to customize how certain Stellar functions behave during execution.
-r, --reader	The path to a file containing key-value properties that are passed to the DataFrameReader when reading the input telemetry. This allows additional customization for how the input telemetry is read.

Configure the Batch Profiler

You can customize the batch profiler to specify where the profiler runs, the format of the telemetry input, and various other properties available in the batch profiler properties file.

Procedure

1. To run the batch profiler using Spark on Yarn, specify the location in $METRON_HOME/config/batch-profiler.properties:

```
spark.master=yarn
```

By default, the batch profiler instructs Spark to run in local mode: spark.master=local. This mode is only useful for testing with a limited set of data.

2. You might also want to set the YARN deploy mode to cluster:

```
spark.submit.deployMode=cluster
```

In cluster mode, the Spark driver runs inside an application master process which is managed by YARN on the cluster, and the client can go away after initiating the application. See the Spark documentation for more information.

3. Specify the appropriate input format for the batch profiler to consume by modifying $METRON_HOME/config/batch-profiler.properties:

```
profiler.batch.input.format=text
profiler.batch.input.path=hdfs://localhost:8020/apps/metron/indexing/indexed/*/*
```

The Profiler can consume archived telemetry stored in a variety of input formats. By default, it is configured to consume the text/json that HCP archives in HDFS. This is often not the best format for archiving telemetry.

4. Review the batch profiler's properties located at $METRON_HOME/config/batch-profiler.properties to customize them for your profiler needs.

See *Batch Profiler Properties* for more information on these properties.

Batch Profiler Properties

Use the batch profiler properties to configure the batch profiler.

By default, the configuration for the batch profiler is stored in the local filesystem at $METRON_HOME/config/batch-profiler.properties. Refer to the Spark documentation for information about Spark properties you can include in the batch profiler properties file.

Table 2: Profiler Properties

Settings.	Description
profiler.batch.input.path	The path to the input data read by the Batch Profiler. Default: hdfs://localhost:9000/apps/metron/indexing/indexed/*/*
profiler.batch.input.format	The format of the input data read by the Batch Profiler. Default: text
profiler.batch.input.begin	Only messages with a timestamp after this will be profiled. Default: undefined; no time constraint
profiler.batch.input.end	Only messages with a timestamp before this will be profiled. Default: undefined; no time constraint
profiler.period.duration	The duration of each profile period. Default: 15
profiler.period.duration.units	The units used to specify the profiler.period.duration. Default: MINUTES
profiler.hbase.salt.divisor	A salt is prepended to the row key to help prevent hot-spotting. Default: 1000
profiler.hbase.table	The name of the HBase table that profiles are written to. Default: profiler
profiler.hbase.column.family	The column family used to store profiles. Default: P

Accessing Profiles

You can use a client API to access the profiles generated by the HCP Profiler to use for model scoring. HCP provides a Stellar API to access the profile data but this section provides only instructions for using the Stellar client API.

You can use this API in conjunction with other Stellar functions such as MAAS_MODEL_APPLY to perform model scoring on streaming data.

Selecting Profile Measurements

The PROFILE_GET command allows you to select all of the profile measurements written.

This command takes the following arguments:

REQUIRED::

profile — The name of the profile

entity — The name of the entity

periods — The list of profile periods to grab. These are ProfilePeriod objects. This field is generally the output of another Stellar function which defines the times to include.

OPTIONAL:

groups_list — List (in square brackets) of groupBy values used to filter the profile. Default is an empty list, which means that groupBy was not used when creating the profile. This list must correspond to the 'groupBy' list used in profile creation.

The groups_list argument in the client must exactly correspond to the groupBy configuration in the profile definition. If groupBy was not used in the profile, groups_list must be empty in the client. If groupBy was used in the profile, then the client groups_list is not optional; it must be the same length as the groupBy list, and specify exactly one selected group value for each groupBy criterion, in the same order. For example:

```
If in Profile, the groupBy
 criteria are:  [ “DAY_OF_WEEK()”,
 “URL_TO_PORT()” ]
Then in PROFILE_GET, an allowed
 groups value would be:  [ “3”,
 “8080” ]
which will select only records from
 Tuesdays with port number 8080.
```

config_overrides — Map (in curly braces) of name:value pairs, each overriding the global config parameter of the same name. Default is the empty Map, meaning no overrides.

Note:

There is an older calling format where groups_list is specified as a sequence of group names, "varargs" style, instead of a List object. This format is still supported for backward compatibility, but it is deprecated, and it is disallowed if the optional config_overrides argument is used.

By default, the Profiler creates profiles with a period duration of 15 minutes. This means that data is accumulated, summarized, and flushed every 15 minutes. The Client API must also have knowledge of this duration to correctly retrieve the profile data. If the Client is expecting 15 minute periods, it will not be able to read data generated by a Profiler that was configured for 1 hour periods, and will return zero results.

Similarly, all six Client configuration parameters listed in the table below must match the Profiler configuration parameter settings from the time the profile was created. The period duration and other configuration parameters from the Profiler topology are stored in a local file system at $METRON_HOME/config/profiler.properties. The Stellar Client API can be configured correspondingly by setting the following properties in HCP's global configuration, on a local file system at $METRON_HOME/config/zookeeper/global.json, then uploaded to ZooKeeper (at /metron/topology/global) by using zk_load_configs.sh:

```
                                    ```
$ cd $METRON_HOME
$ bin/zk_load_configs.sh -m PUSH -i
 config/zookeeper/ -z node1:2181
```
```

Any of these six Client configuration parameters may be overridden at run time using the config_overrides Map argument in PROFILE_GET. The primary use case for overriding the client configuration parameters is when historical profiles have been created with a different Profiler configuration than is currently configured, and the analyst, needing to access them, does not want to change the global Client configuration so as not to disrupt the work of other analysts working with current profiles.

Table 3: Profiler Client Configuration Parameters

Key	Description	Required	Default
profiler.client.period.duration	The duration of each profile period. This value should be defined along with profiler.client.period.duration.units.	Optional	15

Key	Description	Required	Default
profiler.client.period.duration.units	The units used to specify the profile period duration. This value should be defined along with profiler.client.period.duration.	Optional	MINUTES
profiler.client.hbase.table	The name of the HBase table used to store profile data.	Optional	profiler
profiler.client.hbase.column.family	The name of the HBase column family used to store profile data.	Optional	P
profiler.client.salt.divisor	The salt divisor used to store profile data.	Optional	1000
hbase.provider.impl	The name of the HBaseTableProvider implementation class.	Optional	

Specifying Profile Time and Duration

The third required argument for PROFILE_GET is a list of ProfilePeriod objects. These objects allow you to specify the timing, frequency, and duration of the PROFILE_GET. This list is produced by another Stellar function. There are two options available: PROFILE_FIXED and PROFILE_WINDOW.

PROFILE_FIXED

PROFILE_FIXED specifies a fixed period to look back at the profiler data starting from now. These are ProfilePeriod objects.

REQUIRED:

- **durationAgo** — How long ago should values be retrieved from?
- **units** — The units of 'durationAgo'.

OPTIONAL:

- **config_overrides** — Map (in curly braces) of name:value pairs, each overriding the global config parameter of the same name. Default is the empty Map, meaning no overrides.

  For example, to retrieve all the profiles for the last 5 hours: PROFILE_GET('profile', 'entity', PROFILE_FIXED(5, 'HOURS'))

PROFILE WINDOW

PROFILE_WINDOW provides a finer-level of control over selecting windows for profiles. This profile selector allows you to specify the exact time, duration, and frequency for the profile. It does this by a domain specific language that mimics natural language that defines the excluded windows. You can use PROFILE_WINDOW to specify:

- Windows relative to the data timestamp (see the optional now parameter below)
- Non-contiguous windows to better handle seasonal data (for example, the last hour for every day for the last month)
- Profile output excluding holidays
- Only profile output on a specific day of the week

REQUIRED:	**windowSelector**	The statement specifying the window to select.
	now	Optional - The timestamp to use for now.
OPTIONAL:	**config_overrides**	Map (in curly braces) of name:value pairs, each overriding the global config parameter of the same name. Default is the empty Map, meaning no overrides.

For example, to retrieve all the measurements written for 'profile' and 'entity' for the last hour on the same weekday excluding weekends and US holidays across the last 14 days:

```
PROFILE_GET('profile', 'entity', PROFILE_WINDOW('1 hour window every 24
 hours starting from 14 days ago including the current day of the week
 excluding weekends, holidays:us'))
```

Note: The config_overrides parameter operates exactly as the config_overrides argument in PROFILE_GET. The only available parameters for override are:

- profiler.client.period.duration
- profiler.client.period.duration.units

Profile Selector Language

The domain specific language for the profile selector can be broken into a series of clauses, some of which are optional.

Total Temporal Duration	The total range of time in which windows may be specified
Temporal Window Width	The size of each temporal window
Skip distance	(optional) How far to skip between when one window starts and when the next begins

Inclusion/Exclusion specifiers	(optional) The set of specifiers to further filter the window

You must specify either a total temporal duration or a temporal window width. The remaining clauses are optional.

From a high level, the domain specific language fits the following three forms, which are composed of the clauses above:

- time_interval Window (INCLUDING specifier list) (EXCLUDING specifier list)

  temporal window width inclusion specifiers exclusion specifier
- time_interval WINDOW EVERY time_interval FROM time_interval (TO time_interval) (INCLUDING specifier_list) (EXCLUDING specifier list)

  temporal window width skip distance total temporal duration inclusion specifiers exclusion specifier
- FROM time_interval (TO time_interval)

  total temporal duration total temporal duration

Total Temporal Duration

Total temporal duration is specified by a phrase: FROM time_interval AGO TO time_interval AGO. This indicates the beginning and ending of a time interval. This is an inclusive duration.

FROM	Can be the words "from" or "starting from".
time_interval	A time amount followed by a unit (for example, 1 hour). Fractional amounts are not supported. The unit may be "minute", "day", "hour" with any pluralization.
TO	Can be the words "until" or "to".
AGO	(optional) The word "ago"

The TO time_interval AGO portion is optional. If this portion is unspecified then it is expected that the time interval ends now.

Due to the vagaries of the English language, the from and the to portions, if both are specified, are interchangeable with regard to which one specifies the start and which specifies the end. In other words "starting from 1 hour ago to 30 minutes ago" and "starting from 30 minutes ago to 1 hour ago" specify the same temporal duration.

Total Temporal Duration Examples

The domain specific language allows for some flexibility on how to specify a duration. The following are examples of various ways you can specify the same duration.

- A duration starting 1 hour ago and ending now:
  - from 1 hour ago
  - from 1 hour
  - starting from 1 hour ago
  - starting from 1 hour
- A duration starting 1 hour ago and ending 30 minutes ago:
  - from 1 hour ago until 30 minutes ago
  - from 30 minutes ago until 1 hour ago
  - starting from 1 hour ago to 30 minutes ago
  - starting from 1 hour to 30 minutes

Temporal Window Width

Temporal window width is the specification of a window. A window may either repeat within a total temporal duration or it may fill the total temporal duration. This is an inclusive window. A temporal window width is specified by the phrase: time_interval WINDOW.

time_interval	A time amount followed by a unit (for example, 1 hour). Fractional amounts are not supported. The unit may be "minute", day", or "hour" with any pluralization.
WINDOW	(optional) The word "window".

Temporal Window Width Examples

- A fixed window starting 2 hours ago and going until now
  - 2 hour
  - 2 hours
  - 2 hours window
- A repeating 30 minute window starting 2 hours ago and repeating every hour until now. This would result in 2 30-minute wide windows: 2 hours ago and 1 hour ago
  - 30 minute window every 1 hour starting from 2 hours ago

    temporal window width skip distance total temporal duration
  - 30 minute windows every 1 hour from 2 hours ago

    temporal window width skip distance total temporal duration

Skip Distance

Skip distance is the amount of time between when one temporal window begins and the next window starts. It is, in effect, the window period. It is specified by the phrase EVERY time_interval.

time_interval	A time amount followed by a unit (for example, 1 hour). Fractional amounts are not supported. The unit may be "minute", "day", or "hour" with any pluralization.
EVERY	The word/phrase "every" or "for every".

Skip Distance Examples

- A repeating 30 minute window starting 2 hours ago and repeating every hour until now. This would result in 2 30-minute wide windows: 2 hours ago and 1 hour ago
  - 30 minute window every 1 hour starting from 2 hours ago

    temporal window width skip distance total temporal duration
  - 30 minutes window every 1 hour from 2 hours ago

    temporal window width skip distance total temporal duration
- A repeating 30 minute window starting 2 hours ago and repeating every hour until 30 minutes ago. This would result in 2 30-minute wide windows: 2 hours ago and 1 hour ago
  - 30 minute window every 1 hour starting from 2 hours ago until 30 minutes ago

    temporal window width skip distance total temporal duration
  - 30 minutes window every 1 hour from 2 hours ago to 30 minutes ago

    temporal window width skip distance total temporal duration
  - 30 minutes window for every 1 hour from 30 minutes ago to 2 hours ago

    temporal window width skip distance total temporal duration

Inclusion/Exclusion Specifiers

Inclusion and Exclusion specifiers operate as filters on the set of windows. They operate on the window beginning timestamp.

For inclusion specifiers, windows that are passed by any of the set of inclusion specifiers are included. Similarly, windows that are passed by any of the set of exclusion specifiers are excluded. Exclusion specifiers trump inclusion specifiers.

Specifiers follow one of the following formats depending on if it is an inclusion or exclusion specifier:

- INCLUSION specifier, specifier, ...

  INCLUSION can be "include", "includes" or "including"
- EXCLUSION specifier, specifier, ...

  EXCLUSION can be "exclude", "excludes" or "excluding"

The specifiers are a set of fixed specifiers available as part of the language:

- Fixed day of week-based specifiers - includes or excludes if the window is on the specified day of the week
  - "monday" or "mondays"
  - "tuesday" or "tuesdays"
  - "wednesday" or "wednesdays"
  - "thursday" or "thursdays"
  - "friday" or "fridays"
  - "saturday" or "saturdays"
  - "sunday" or "sundays"
  - "weekday" or "weekdays"
  - "weekend" or ""weekends"
- Relative day of week-based specifiers - includes or excludes based on the day of week relative to now
  - "current day of the week"
  - "current day of week"
  - "this day of the week"
  - "this day of week"
- Specified date - includes or excludes based on the specified date
  - "date" - Takes up to 2 arguments
    - The day in yyyy/MM/dd format if no second argument is provided

      Example: date:2017/12/25 would include or exclude December 25, 2017
    - (optional) The format in which to specify the first argument

      Example: date:20171225:yyyyMMdd would include or exclude December 25, 2017
- Holidays - includes or excludes based on if the window starts during a holiday
  - "holiday" or "holidays"
    - Arguments form the jollyday hierarchy of holidays. For example, "us:nyc" would be holidays for New York City, USA

      Countries supported are those supported in jollyday

      Example: holiday:us:nyc would be the holidays of New York City, USA
    - If none is specified, it will choose based on locale.

      Example: holiday:hu would be the holidays of Hungary

Inclusion/Exclusion Specifiers Examples

The following are inclusion/exclusion specifier examples and identify the various clauses used in these examples.

Assume the following examples are executed at noon.

- A 1 hour window for the past 8 'current day of the week'
  - 1 hour window every 24 hours from 56 days ago including this day of the week

    temporal window width skip distance total temporal duration inclusion/exclusion specifiers
- A 1 hour window for the past 8 tuesdays
  - 1 hour window every 24 hours from 56 days ago including tuesdays

    temporal window width skip distance total temporal duration inclusion/exclusion specifiers
- A 30 minute window every tuesday at noon starting 14 days ago until now
  - 30 minute window every 24 hours from 14 days ago including tuesdays

    temporal window width skip distance total temporal duration inclusion/exclusion specifiers
- A 30 minute window every day except holidays and weekends at noon starting 14 days ago until now
  - 30 minutes every 24 hours from 14 days ago excluding holidays:us, weekends

    30 minutes every 24 hours from 14 days ago including weekdays excluding holidays:us, weekends

    temporal window width skip distance total temporal duration inclusion/exclusion specifiers
- A 30 minute window at noon every day from 7 days ago including saturdays and excluding weekends. Because exclusions trump inclusions, the following will never yield any windows
  - 30 minute window every 24 hours from 7 days ago including saturdays excluding weekends

    temporal window width skip distance total temporal duration inclusion/exclusion specifiers

Client Profile Example

The following are usage examples that show how the Stellar API can be used to read profiles generated by the Metron Profiler. This API would be used in conjunction with other Stellar functions like MAAS_MODEL_APPLY to perform model scoring on streaming data.

These examples assume a profile has been defined called ‘snort-alerts’ that tracks the number of Snort alerts associated with an IP address over time. The profile definition might look similar to the following.

```
{
  "profiles": [
    {
      "profile": "snort-alerts",
      "foreach": "ip_src_addr",
      "onlyif":  "source.type == 'snort'",
      "update":  { "s": "STATS_ADD(s, 1)" },
      "result":  "STATS_MEAN(s)"
    }
  ]
}
```

During model scoring, the entity being scored, in this case a particular IP address, will be known. The following examples shows how this profile data might be retrieved. Retrieve all values of ‘snort-alerts’ from ‘10.0.0.1’ over the past 4 hours.

```
PROFILE_GET('snort-alerts', '10.0.0.1', PROFILE_FIXED(4, 'HOURS'))
```

Retrieve all values of ‘snort-alerts’ from ‘10.0.0.1’ over the past 2 days.

```
PROFILE_GET('snort-alerts', '10.0.0.1', PROFILE_FIXED(2, 'DAYS'))
```

If the profile had been defined to group the data by weekday versus weekend, then the following example would apply:

Retrieve all values of ‘snort-alerts’ from ‘10.0.0.1’ that occurred on ‘weekdays’ over the past 30 days.

```
PROFILE_GET('snort-alerts', '10.0.0.1', PROFILE_FIXED(30, 'DAYS'),
 ['weekdays'] )
```

The client may need to use a configuration different from the current Client configuration settings. For example, perhaps you are on a cluster shared with other analysts, and need to access a profile that was constructed 2 months ago using different period duration, while they are accessing more recent profiles constructed with the currently configured period duration. For this situation, you may use the config_overrides argument:

Retrieve all values of ‘snort-alerts’ from ‘10.0.0.1’ over the past 2 days, with no groupBy, and overriding the usual global client configuration parameters for window duration.

```
PROFILE_GET('profile1', 'entity1', PROFILE_FIXED(2,
 'DAYS', {'profiler.client.period.duration' : '2',
 'profiler.client.period.duration.units' : 'MINUTES'}), [])
```

Retrieve all values of ‘snort-alerts’ from ‘10.0.0.1’ that occurred on ‘weekdays’ over the past 30 days, overriding the usual global client configuration parameters for window duration.

```
PROFILE_GET('profile1', 'entity1', PROFILE_FIXED(30,
 'DAYS', {'profiler.client.period.duration' : '2',
 'profiler.client.period.duration.units' : 'MINUTES'}), ['weekdays'] )
```