Prometheus | Apache Flink

http://localhost:53197

TABLE OF CONTENTS

Prometheus Sink

This sink connector can be used to write **data** to Prometheus-compatible storage, using the Remote Write Prometheus interface.

The Prometheus-compatible backend must support Remote Write 1.0 standard API, and the Remote Write endpoint must be enabled.

> This connector is not meant for sending internal Flink metrics to Prometheus. To publish Flink metrics, for monitoring health and operations of the Flink cluster, you should use Metric Reporters.

To use the connector, add the following Maven dependency to your project:

Only available for stable versions.

Usage

The Prometheus sink provides a builder class to build a `PrometheusSink` instance. The code snippets below shows how to build a `PrometheusSink` with a basic configuration, and adding an optional request signer.

```
PrometheusSink sink = PrometheusSink.builder()
        .setPrometheusRemoteWriteUrl(prometheusRemoteWriteUrl)
        .setRequestSigner(new AmazonManagedPrometheusWriteRequestSigner(prometheusRemoteWriteUrl, prometheusRegion)) // Optional
        .build();
```

The only **required** configuration `prometheusRemoteWriteUrl`. All other configurations are optional.

If your sink has parallelism > 1, you need to ensure the stream is keyed using the `PrometheusTimeSeriesLabelsAndMetricNameKeySelector` key selector, so that all samples of the same time-series are in the same partition and order is not lost. See Sink parallelism and keyed streams for more details.

Input data objects

The sink expects `PrometheusTimeSeries` records as input. Your input data must be converted into `PrometheusTimeSeries`, using a map or flatMap operator, before the sending to the sink.

`PrometheusTimeSeries` instances are immutable and cannot be reused. You can use the builder to create and populate instances.

A `PrometheusTimeSeries` represents a single time-series record of sent to the Remote Write interface. Each time-series record may contain multiple samples.

> In the context of Prometheus the term “time-series” is overloaded. It means both *a series of samples with a unique set of labels* (a time-series in the underlying time-series database), and *a record sent to the Remote Write interface*. A `PrometheusTimeSeries` instance represents a record sent to the interface.
>
> The two concepts are related, because time-series “records” with the same sets of labels are sent to the same “database time-series”.

Each `PrometheusTimeSeries` record contains:

- One **`metricName`**. A string that translated into the value of the `__name__` label.
- Zero or more **`Label`**. Each label has a `key` and a `value`, both `String`. Labels represent additional dimensions of the samples. Duplicate Label keys are not allowed.
- One or more **`Sample`**. Each sample has a `value` (`double`) representing the measure, and a `timestamp` (`long`) representing the time of the measure, in milliseconds from the Epoch. Duplicate timestamps in the same record are not allowred.

The following pseudocode represents the structure of a `PrometheusTimeSeries` record:

```
PrometheusTimeSeries
  + --> (1) metricName <String>
  + --> (0..*) Label
            + name <String>
            + value <String>
  + --> 1..* Sample
            + timestamp <long>
            + value <double>
```

> The set of Labels and metricName are the unique identifier of the database time-series. A composite of all Lables and metricName is also the key you should use to partition data, inside the Flink application and upstream, to guarantee ordering per time-series is retained.

Populating a PrometheusTimeSeries

`PrometheusTimeSeries` provides a builder interface.

```
PrometheusTimeSeries inputRecord =
        PrometheusTimeSeries.builder()
                .withMetricName(metricName)
                .addLabel("DeviceID", instanceId)
                .addLabel("RoomID", roomId)
                .addSample(measurement1, time1)
                .addSample(measurement2, time2)
                .build();
```

Each `PrometheusTimeSeries` instance can contain multiple samples. Call `.addSample(...)` multiple times to add each of them. The order samples are added is retained. The max number of samples per record is limited by the `maxBatchSizeInSamples` configuration.

Aggregating multiple samples into a single `PrometheusTimeSeries` record may improve write performances.

Prometheus remote-write constraints

Prometheus imposes strict constrains on data format and on ordering. Any write request containing records that violate these constraints is rejected.

See Remote Write specification for details about these constrains.

In practice, the behavior when write data to a Prometheus-compatible backend, depends on the Prometheus implementation and configuration. In some cases, these constraints are relaxed, and writes violating the Remote Write specifications may be accepted.

For this reason, this connector **does not enforce** any data constraints directly. The user is responsible of sending to the sink data that does not violate the actual constraints of your Prometheus implementation. See User responsibilities for more details.

Ordering constraints

Remote Write specification require multiple ordering constraints:

1. **Labels** within a `PrometheusTimeSeries` record must be in lexicographical **order by key**.
2. **Samples** within a `PrometheusTimeSeries` record must be in **`timestamp` order**, from older to newer.
3. **All samples** belonging to the **same time-series** (a unique set of labels and metricName) must be written in **`timestamp` order**.
4. Withing the **same time-series**, duplicate samples with the **same timestamp** are not allowed

When the Prometheus-compatible backend implementation supports *out-of-order time window* and the option is enabled, sample ordering constraint is relaxed. You can send out of order data withing the configured window.

Format constraints

The `PrometheusTimeSeries` records sent to the sink must also respect the following constraints:

- **`metricName`** must be defined and non-empty. The connector translates this property into the value of the `__name__` label.
- Label **names** must follow the regex `[a-zA-Z:_]([a-zA-Z0-9_:])`.
- Label **names** must not begin with __ (double underscore). These label names are reserved.
- No duplicate Label **names** is allowed.
- Label **values** and `metricName` may contain any UTF-8 character.
- Label **values** cannot be empty (null or empty string).

The `PrometheusTimeSeries` builder does not enforce these constraints.

User responsibilities

The user is responsible for sending to the sink records (`PrometheusTimeSeries`) respecting format and ordering constraints required by your Prometheus implementation. The connector does not perform any validation or reordering.

Sample ordering by timestamp is particularly important. Samples belonging to the same time-series, i.e. with the same set of Labels and the same metric name, must be written in timestamp order. Source data must be generated in order. The order must also be retained before the sink. When partitioning the data, records with same set of labels and metric name must be sent to the same partition, in order to retain ordering.

Malformed or out of order records written to the Remote Write endpoint are rejected and dropped by the sink. This may cause data loss. See

Any record violating ordering sent to the sink is dropped and may cause other records batched in the same write-request to be dropped. For more details, see Connector guarantees.

Sink parallelism and keyed streams

Each sink operator sub-task uses a single thread to send write requests to the Remote Write endpoint, and `PrometheusTimeSeries` records are written in the same order as theyare received by the sub-task.

To ensure all records belonging to the same time-series (i.e. `PrometheusTimeSeries` with identical list of `Label` and `metricName`) are written by the same sink subtask, the stream of `PrometheusTimeSeries` must be keyed using `PrometheusTimeSeriesLabelsAndMetricNameKeySelector`.

```
DataStream<MyRecord> inputRecords;
// ...
KeyedStream<PrometheusTimeSeries> timeSeries = inputRecords
        .map(new MyRecordToTimeSeriesMapper())
        .keyBy(new PrometheusTimeSeriesLabelsAndMetricNameKeySelector());

timeSeries.sinkTo(prometheusSink);
```

Using this key selector prevents accidental out-of-orderness due to repartitioning before the sink operator. However, the user is responsible to retain ordering to this point, by partitioning the records correctly.

Error handling

This paragraph covers handling of errors conditions when writing data to the Remote Write endpoint.

There are four types of error conditions:

1. Retryable errors due to temporary error conditions in the Remote-Write server or due to throttling: `5xx` or `429` http responses, connectivity issues.
2. Non-retryable errors due to data violating any of the constraints, malformed data or out-of-order samples: `4xx` http responses, except `429`, `403` and `404`.
3. Fatal error response: authentication failures (`403` http response) or incorrect endpoint path (`404` http response).
4. Any other unexpected failure while writing, due to exceptions while writing to the Prometheus endpoint.

On-error behaviors

When any of the above error is encountered, the connector implements one of these two behaviors:

1. `FAIL`: throw an unhandled exception, the job fails
2. `DISCARD_AND_CONTINUE`: discard the request that caused the error, and continue with the next record.

When a write request is discarded on `DISCARD_AND_CONTINUE`, all the following happen:

1. Log a message at `WARN` level with the cause of the error. When the error is caused by a response from the endpoint, the payload of the response from the endpoint is included.
2. Increase counters metrics, to count the number of rejected samples and write requests.
3. **Drop the entire write request**. Note that, due to batching, a write request may contain multiple `PrometheusTimeSeries`.
4. Continue with the next input record.

> Prometheus Remote Write does not support partial failures. Due to batching, a single write request may contain multiple input records (`PrometheusTimeSeries`). If a request contains even a single offending record, the entire write request (the entire batch) must be discarded.

Retriable error responses

A typical retriable error condition is enpoint throttling, with a `429, Too Many Requests` response.

On retriable errors, the connector will retry, using a configurable backoff strategy. When the maximum number of retries is exceeded, the write request fails. What happens at this point depends on the `onMaxRetryExceeded` error handling configuration.

- If `onMaxRetryExceeded` is `FAIL` (default): the job fails and restarts from checkpoint.
- If `onMaxRetryExceeded` is `DISCARD_AND_CONTINUE`: the entire write request is dropped and the sink continues with the next record.

Non retriable error responses

A typical non-retriable condition is due to malformed or out of order samples that are rejected by Prometheus with a `400, Bad Request` response.

When such an error is received, the connector applies the `DISCARD_AND_CONTINUE` behavior. This behavior is currently not configurable.

Fatal error responses

`403, Forbidden` responses, caused by incorrect or missing authentication, and `404, Not Found` responses, caused by incorrect endpoint URL, are always considered fatal. Behavior is always `FAIL` and not configurable.

Other I/O errors

Any I/O error that happens in the http client is also fatal (behavior is always `FAIL`, not configurable).

Error handling configuration

The error handling behavor is partly configurable. You can configure the behavior when building the instance of the sink.

```
PrometheusSink sink = PrometheusSink.builder()
        // ...
        .setErrorHandlingBehaviourConfiguration(
                SinkWriterErrorHandlingBehaviorConfiguration.builder()
                    .onMaxRetryExceeded(OnErrorBehavior.DISCARD_AND_CONTINUE)
                    .onPrometheusNonRetriableError(OnErrorBehavior.DISCARD_AND_CONTINUE)
                    .build())
        .build();
```

At the moment, the only supported configuration, `onMaxRetryExceeded`, controls the behavior when the maximum nunmber of retries, after a retriable error, is exceeded. The default behavior is `FAIL`.

The configuration also allows setting the behavior on Prometheus non-retriable error (`onPrometheusNonRetriableError`), but the only allowed value at the moment is `DISCARD_AND_CONTINUE`.

Retry configuration

When a retriable error condition is encoutered, the sink retries with an exponential backoff strategy.

The retry strategy can be configured with the following parameters:

- `initialRetryDelayMS`: (default `30` millis) Initial retry delay. The retry delay doubles on every subsequent retry, up to the maximum retry delay.
- `maxRetryDelayMS`: (default `5000` millis) Maximum retry delay. When this delay is reached, every subsequent retry has the same delay. Must be bigger than `InitialRetryDelayMS`
- `maxRetryCount`: (default `100`) Maximum number of retries for a single write request. Set max retries to `Integer.MAX_VALUE` if you want to (practically) retry forever.

When `maxRetryCount` is exceeded, the connector stops retrying. What happens at this point depends on the `onMaxRetryExceeded` error handling behavior.

You can configure the retry strategy when building the sink.

```
PrometheusSink sink = PrometheusSink.builder()
        // ...
        .setRetryConfiguration(
                RetryConfiguration.builder()
                    .setInitialRetryDelayMS(10L)
                    .setMaxRetryDelayMS(5000L)
                    .setMaxRetryCount(Integer.MAX_VALUE)
                .build())
        // ...
        .build();
```

Batching

To optimize write throughput, the sink batches writes. Multiple `PrometheusTimeSeries` are batched in a single write request to the Remote Write endpoint.

Batching is based on the number of samples per request, and a max buffering time.

The connector starts with writing a single `PrometheusTimeSeries` per write request. If the write succeeds, the batch size is increased up to a configurable limit of samples per request. The actual number of `PrometheusTimeSeries` per request may vary, because `PrometheusTimeSeries` may contain a variable number of samples.

Buffered `PrometheusTimeSeries` are stored in Flink state, and are not lost on application restart.

Batching can be controlled using the following parameters:

- `maxBatchSizeInSamples`: (default: `500`) max number of samples in a write request.
- `maxTimeInBufferMS`: (default: `5000` millis) Max time inpout `PrometheusTimeSeries` are buffered before emitting the write request.
- `maxRecordSizeInSamples`: (default: `500`) max number of samples in a single `PrometheusTimeSeries`. It must be less or equal `MaxBatchSizeInSamples`.

> If a record containing more samples than `maxRecordSizeInSamples` is encountered, the sink throws and exception causing the job to fail and restart from checkpoint, putting the job in an endless loop.

You can configure batching when building the sink:

```
PrometheusSink sink = PrometheusSink.builder()
        .setMaxBatchSizeInSamples(100)
        .setMaxTimeInBufferMS(10000)
        // ...
        .build();
```

Larger batches improve write performance but also increase the number of record potentially lost when a write request is rejected, due to `DISCARD_AND_CONTINUE` behavior. Reduging `maxBatchSizeInSamples` minimize data loss in this case, but may heavly reduce the throughput that Prometheus can injest. The default `maxBatchSizeInSamples` of 500 maximizes the ingestion throughput.

Request Signer

Remote Write specification does not specify any authentication scheme. Authentication is delegated to the specific Prometheus-compatible backend implementation.

The connector allows to specify a request signer that can add headers to the http request. These headers can be based on the request body or any of the existing headers. This allows to implement authentication schemes that requires passing authentication or signature tokens in headers.

A request signer must implement the `PrometheusRequestSigner` interface.

```
public interface PrometheusRequestSigner extends Serializable {
  void addSignatureHeaders(Map<String, String> requestHeaders, byte[] requestBody);
}
```

Refer to the JavaDoc or the source code for more details.

A request signer can be added to the sink in the builder:

```
PrometheusSink sink = PrometheusSink.builder()
        // ...
        .setRequestSigner(requestSigner)
        .build();
```

Amazon Managed Prometheus request signer

An implementation of `PrometheusRequestSigner` supporting Amazon Managed Prometheus (AMP) is provided.

An additional dependency is required to use the AMP request signer:

Only available for stable versions.

The AMP signer retrieves AWS credentials using `DefaultCredentialsProvider`. Credentials are retrieved and used to sign every request to the endpoint.

You can add the AMP request signer to the sink:

```
PrometheusSink sink = PrometheusSink.builder()
        // ...
        .setRequestSigner(new AmazonManagedPrometheusWriteRequestSigner(prometheusRemoteWriteUrl, prometheusRegion))
        .build();
```

HTTP client configuration

You can configure the HTTP client that sends write requests to the Remote Write endpoint.

- `socketTimeoutMs`: (default: `5000` millis) HTTP client socket timeout
- `httpUserAgent`: (default: `Flink-Prometheus`) User-Agent header

```
PrometheusSink sink = PrometheusSink.builder()
        // ...
```

```
        .setSocketTimeoutMs(5000)
        .setHttpUserAgent(USER_AGENT)
        .build();
```

Connector metrics

The connector exposes custom metrics, counting data successfully written to the endpoint, and data dropped due to `DISCARD_AND_CONTINUE`.

Metric name	Description
`numSamplesOut`	Count of **samples** successfully written to Prometheus
`numWriteRequestsOut`	Count of **samples** successfully written to Prometheus
`numWriteRequestsRetries`	Count of **write requests** reties, due to retryable errors (e.g. throttling)
`numSamplesDropped`	Count of **samples** that have been dropped (data loss!) due to `DISCARD_AND_CONTINUE`
`numSamplesNonRetriableDropped`	Count of **samples** that have been dropped (data loss!) due to `onPrometheusNonRetriableError` set to `DISCARD_AND_CONTINUE` (default)
`numSamplesRetryLimitDropped`	Count of **samples** that have been dropped (data loss!) due to `onMaxRetryExceeded` set to `DISCARD_AND_CONTINUE`, when the retry limit was exceeded
`numWriteRequestsPermanentlyFailed`	Count of **write requests** permanently failed, due to any reasons

> The `numByteSend` metric should be ignored. This metric does not actually measure bytes, due to limitations of AsyncSink this connector is based on. Use `numSamplesOut` and `numWriteRequestsOut` to monitor the actual output of the sink.

The metric group name is “Prometheus” by default. It can be changed:

```
PrometheusSink sink = PrometheusSink.builder()
        // ...
        .setMetricGroupName("my-metric-group")
        .build();
```

Connector guarantees

The connector provides **at-most-once** guarantees. Data loss can happen, in particular if input data is malformed or out of order.

Data may be lost due to the `DISCARD_AND_CONTINUE` on-error behaviour. This behavior may optionally be enabled when the maximum number of retries is exceeded, but is always enabled when non-retriable error conditions are encountered.

This behavior is due to the design of Prometheus Remote Write interface, that does not allow out-of-order writes in the same time-series. To prevent from putting the job in an endless loop of fail and restart from checkpoint, when out of order data is encontered, is to discard and continue. Out of order writes also happen when the job restart from checkpoint. Without discard and continue the sink would not allow the job to recover from checkpoint at all.

At the same time, Prometheus imposes per time-series ordering by timestamp. The sink guarantees the order is retained per partition. Key-by using `PrometheusTimeSeriesLabelsAndMetricNameKeySelector` guarantees that the input is partitioned by time-series, and no accidental reordering happens during the write. The user is responsible to partition data before the sink to ensure order is retained.

Example application

You can find a complete application demonstrating the configuration and usage of this sink in the tests of the connector.

Check out the source of `org.apache.flink.connector.prometheus.sink.examples.DataStreamExample`.

This class contains a full application that generates random data internally and writes to Prometheus.

Want to contribute translation?

Back to top

Edit This Page

Just One Page PDF

Just One Page PDF, save your entire webpage as a one-page PDF.

Title:	Prometheus	Apache Flink
Created By:	SNW
Save Date:	30/09/2024, 19:10:30
Source:	http://localhost:53197/docs/connectors/datastream/prometheus/