Tab 1

FLIP-524 - CloudWatch Metric Sink Connector

Discussion thread	*https://lists.apache.org/thread/9yyx5oxsy5hf8nbhds4do6tmm5bchh18*
Vote thread	*TBC*
Author	Daren Wong
JIRA	*https://issues.apache.org/jira/browse/FLINK-37688*
Release	TBC

Motivation

There is demand within the community for an Amazon CloudWatch Metric Sink connector. This sink will allow users to write custom metrics to their CloudWatch. An example use case could be that the user might want to consume a stream of metrics, transform/reduce the cardinality of the metrics space, and finally publish it to CloudWatch as their metrics database.

Public Interfaces

- Sink:
    - FLIP-143: Unified Sink API
    - FLIP-191: SinkV2
    - FLIP-171: Async Sink

Proposed Changes

We are putting forward a proposal to develop a submodule called "flink-connector-cloudwatch" within the existing framework of "flink-connector-aws" By integrating it into "flink-connector-aws," we can leverage the authentication and essential utilities already present in the AWS-specific modules. This will streamline the development process and enable direct utilization of these resources.

Below are the design considerations for the new sink:

- Sink
- Supports both Bounded (Batch) and Unbounded (Streaming)
- Usable in both DataStream and Table API/SQL

Data Input Format

CloudWatch PutMetricRequest requires a structured `EntityMetricData` or `MetricDatum` object as input. Therefore, the connector will provide an `ElementConverter` with generic input type but a static output type. The static output type will be called `MetricWriteRequest`, which will contain all the properties required to build a `MetricDatum` object and can be simply extended with an `Entity` field to support `EntityMetricData` in the future.

For example, `MetricWriteRequest` can be defined as follows:

```
@PublicEvolving
public class MetricWriteRequest implements Serializable {
    private final String metricName; // Required
    private final Dimension[] dimensions; // Optional
    private final double[] values; // Optional
    private final double[] counts; // Optional
    private final Instant timestamp; // Optional
    private final String unit; // Optional
    private final int storageResolution; // Optional
    private final double statisticMax; // Optional
    private final double statisticMin; // Optional
    private final double statisticSum; // Optional
    private final double statisticCount; // Optional
```

Users can provide their custom `ElementConverter` to convert from user defined input type to `MetricWriteRequest`. For example,

```
CloudWatchSink.<Sample>builder()
                .setNamespace("CloudWatchSinkTest")
                .setCloudWatchClientProperties(sinkProperties)
                .setCloudWatchSinkElementConverter(new MetricWriteRequestElementConverter<>() {
                    @Override
                    public MetricWriteRequest apply(Sample sample, SinkWriter.Context context) {
                        return MetricWriteRequest.builder()
                                .withMetricName(sample.getMetricName())
                                .addValue(sample.getValue())
                                .addCount(sample.getCount())
                                .withTimestamp(Instant.ofEpochMilli(sample.getTimestamp()))
                                .build();
                    }
                })
                .build();
```

Sink Writer Design

Batch Writing to CloudWatchAsyncClient - A list of `MetricWriteRequest` will be batched based on `maxBatchSize` which is then submitted as a PutMetricDataRequest.

Example Sink Writer submitRequestEntries:

```
   @Override
   protected void submitRequestEntries(
           List<MetricWriteRequest> entries,
ResultHandler<MetricWriteRequest> resultHandler) {

       final PutMetricDataRequest putMetricDataRequest =
               PutMetricDataRequest.builder()
                       .namespace(namespace)
                       .metricData(getMetricData(entries))
                       .strictEntityValidation(true)
                       .build();

       CompletableFuture<PutMetricDataResponse> future =

clientProvider.getClient().putMetricData(putMetricDataRequest);
```

Sink Writer Configuration/Limitation
Note that CloudWatch PutMetricDataRequest has some constraints and will be taken into consideration in connector's configurations as follows:

- Maximum size per CW PutMetricDataRequest is 1MB → `maxBatchSizeInBytes` cannot be more than 1 MB
- Maximum number of MetricDatum per CW PutMetricDataRequest is 1000 → `maxBatchSize` cannot be more than 1000
- Maximum 150 unique values in MetricDatum.Values → `maxRecordSizeInBytes` cannot be more than 150 Bytes (assuming each 1 value size is 1 byte)
- CloudWatch API uses Java double, but it doesn't support Double.NaN and → Set `strictEntityValidation` to true
- MetricDatum Timestamp limitations (up to 2 weeks in the past and up to 2 hours into the future) → `MetricWriteRequest` will have validation against this upon creation
- Out of data ordering is accepted by CloudWatch.
- CloudWatchSink will be configured per namespace, and each Sink can put metric of one of multiple metricName, this is aligned with CloudWatch PutMetricDataRequest API as well.

Sink Writer Error Handling
CloudWatch PutMetricDataRequest does not support partial failure. If the batch contains one `MetricDatum` poison pill, the request will fail and be handled as a fully failed request. In addition, CloudWatch rejects any metric that's more than 2 weeks old, we will add a configurable option for users to determine the error handling behavior of either: 1) drop the records OR 2) trigger a job failure OR 3) keep retrying the batch.

Example Sink Usage
A sample using the connector is shown below:

```
CloudWatchSink.builder()
        .setNamespace("CloudWatchSinkTestNamespace")
        .setElementConverter(new SampleMetricWriteRequestElementConverter())
        .setCloudWatchClientProperties(sinkProperties)
        .build();
```

TableAPI Design
To convert a `RowData` to `MetricWriteRequest`, users will have to define configuration to identify column names associated with the cloudwatch namespace, metric name, dimensions, etc. A sample configuration can be seen below:

```
CREATE TABLE CloudWatchTable (
 `cw_metric_name` STRING,
 `cw_dim` STRING,
 `cw_value` BIGINT,
 `cw_count` BIGINT
)
WITH (
 'connector' = 'cloudwatch',
 'aws.region' = 'us-east-1',
 'metric.namespace' = 'cw_connector_namespace',
 'metric.name.key' = 'cw_metric_name',
 'metric.dimension.keys' = 'cw_dim',
 'metric.value.key' = 'cw_value',
 'metric.count.key' = 'cw_count'
);
```
User can then insert values into the sink, for example:
```
INSERT INTO CloudWatchTable VALUES ("cpu", "sensor_1", 98, 1);
INSERT INTO CloudWatchTable VALUES ("memory", "sensor_1", 160, 1);
```

TableAPI Configuration List
```
- metric.namespace // Required
- metric.name.key // Optional
- metric.dimension.keys // Optional
- metric.value.key // Optional
- metric.count.key // Optional
- metric.unit.key // Optional
- metric.storage-resolution.key // Optional
- metric.timestamp.key // Optional
- metric.statistic.max.key // Optional
- metric.statistic.min.key // Optional
- metric.statistic.sum.key // Optional
- metric.statistic.sample-count.key // Optional
- sink.invalid-metric.retry-mode // Optional
```

At a high level, there are 3 main types of key:
- `metric.namespace` - Required in every CW PutMetricDataRequest
- `metric.X.key` - Column key identifier to map the Table column to the respective fields in the CW PutMetricDataRequest. For example, “metric.timestamp.key = my_timestamp” means the TableSink will look for column name/field “my_timestamp” to extract it’s value to be used as timestamp in CW PutMetricDataRequest.

- `sink.invalid-metric.retry-mode` - Error handling behavior when an Invalid record is present, i.e invalid timestamp. 3 retry mode options are: 1) drop the records OR 2) trigger a job failure OR 3) keep retrying the batch.

Semantics
CloudWatch currently does not support two phase commits and hence this sink will provide at least once guarantee.

Compatibility, Deprecation, and Migration Plan

The connectors are compatible with CloudWatch. With respect to Flink, this is a new feature, no compatibility, deprecation and migration plan is expected.

Test Plan

We will add the following tests:

- Unit tests
- Integration tests that perform end to end tests against a CloudWatch localstack container

Rejected Alternatives

None