Google Dataflow Streaming Time Series Sample Library

User Guide

Version 0.4.0

Table of contents

- Overview
  - Library
  - Pipeline Construction
  - Features
- Tutorials
  - Workflow
    - Options
    - Data ingestion
    - Metrics generation
    - Output results
  - Before you begin
  - Output metrics as log files
  - Output generated metrics as TFExamples to a file system
  - Output generated metrics as rows to BigQuery
  - Output generated metrics as a JSON string to Pub/Sub
- How-tos
  - Convert Source Data
  - Validation
  - Gap filling
  - Metrics
    - Set window options for metrics creation
    - Merged output
      - Type 1
      - Type 2
    - Time snapshots
    - Integrate with TFX via tf.train.Example serialization
      - Features for model building
      - Features to inference
  - Metadata
  - Data
  - Feature names
  - Window Snapshot
    - Constraints
    - Window snapshot support for TSAccum and TSAccumSequence
    - Implementation
    - Performance

Singleton snapshot object

Appendix A: Internal data types

Data

TSKey

TSDataPoint

TSAccum

TSAccumSequence

Overview

Note: This document is based on version 0.4.0 of the Dataflow streaming time series sample library.

This document provides guidance for evaluating and using the Dataflow streaming time series sample library to perform real-time gap filling and imputation for streaming *time series*, defined as a series of data points indexed in chronological order. The sample library helps developers generate metrics from streaming data *as it streams*, to facilitate downstream processes including machine learning (ML) model training, traditional business intelligence dashboards, and reporting tools.

While this overview helps data and ML engineers evaluate the Dataflow streaming time series sample library's suitability for specific use cases, the how-tos provide development examples for using the library, including descriptions of the library components and configurations.

This document assumes that readers are experienced with Apache Beam and Dataflow. Future documentation may address an audience that is less familiar with these technologies.

Library

Comparing two or more streams of time series data can involve the development of complex algorithms. For example, when events that comprise a time series occur spontaneously based on extraneous processes, it is necessary to perform temporal alignment of individual time series to support meaningful analysis. Correlation analysis, in which each time series must offer a data point at every time step, is one example of this use case.

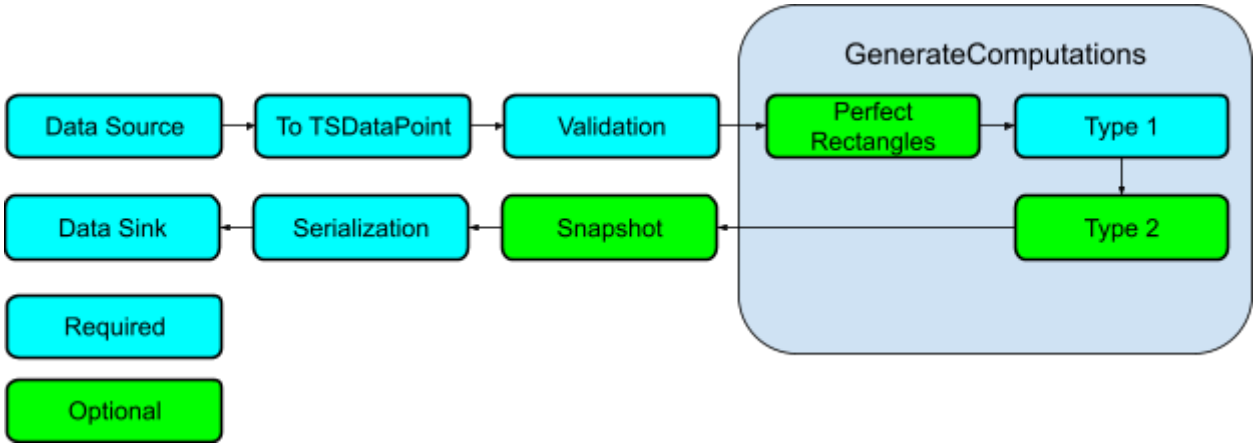
When run using cloud-native services, pipelines built with the Apache Beam model simplify streaming time series comparison by reducing the user's operational and scaling burden. The open-source sample library illustrates this approach.

The library includes examples of real-time calculations for producing derived time series, and supports integration with TensorFlow Extended, a complete Machine Learning Operations (MLOps) SDK, through output serialization via TF.train.Example's modifications to TFExampleGen. Through the Apache Beam model, the library offers the advantage of synchronization of many streams.

Pipeline Construction

According to the Apache Beam glossary, a pipeline is “an encapsulation of your entire data processing task, including reading input data from a source, transforming that data, and writing output data to a sink. ” In the samples provided by the Dataflow streaming time series sample library, a collection of transforms are made available for use, and it’s up to users to put the transforms into a pipeline. Future versions of the sample library will offer the pipelines as a primitive, which will abstract the pipeline construction task away from the user. Power users will still be able to make use of the raw transforms.

A typical Dataflow streaming time series sample library pipeline takes the following shape:



The Apache Beam glossary also mentions that the transforms in a pipeline can be represented as a directed acyclic graph (DAG).The GenerateComputations transform composes the gap filling and metrics calculation in a subgraph, e.g., a subset of the DAG’s nodes and edges. This transform is used to bring together the gap-filling and metric creation transforms.

Features

The Dataflow streaming time series sample library provides five features to support the creation of real-time metrics:

Feature	Description	Use the feature
Type 1 and 2 metric generation	The Dataflow streaming time series library samples provide a framework for computing two forms of metrics: ● Type 1: These are transitive and associative metrics that can be computed in a distributed manner, for	See Metrics

	example, SUM, MIN, MAX, FIRST and LAST. ● Type 2: These metrics must be computed per-key-and-window using an ordered sequence of data, for example, absolute gain, or “relative strength index” in the financial domain. To support performance and scaling, the library allows for the computation of Type 2 metrics only from Type 1 metrics.	
Gap filling	The Dataflow streaming time series library supports more complex gap-filling use cases in which time series aggregations must be uninterrupted, even when data streams carry no records. For example: ● IOT: A device emits a signal for each change, and emits no signal for no change (e.g. to conserve battery power). However, the absence of downstream data for a given period does not indicate the period has no information; rather, it indicates that the period’s information (either no activity, or lost device function) has not been observed. In the absence of new data, either the last known value can be assumed, or a value can be inferred until some per-key time-to-live is reached. ● Finance: As the price of an asset changes, ticks are produced. The bid or ask can update independently, but while there is no change the last observed value can be assumed.	See Gap filing
TF.Example serialization	ML models and inference paths will likely be key consumers of the library,	See TF.Example serialization

	which means the output must be serialized in a format readily consumable by MLOps systems. The Dataflow streaming time series sample library is standardized on TensorFlow examples, and accordingly generates the data as ordered lists. The data then matches the [timesteps,features] shape expected by sequence models like long short term memory (LSTM).	
Time snapshots	Consumers of metrics from the time series sample library need continuous snapshots of all keys within a specific window of time. Many use cases require output metrics that are strongly consistent and perfectly time-aligned based on Apache Beam watermarks.	See Time Snapshots
TFX pipeline example	The sample library's pipeline example is deployed using the TensorFlow Extended (TFX) platform. This model is saved as a save_model.pb file for use with the tfx-bsl RunInference transform.	See ML pipeline examples in the GitHub repository

Tutorials

The tutorials in this section describe how to download the sample library and create metrics from an example pipeline that uses a generated stream of data. The tutorials represent the four modes of the SinWaveExample class:

- Generated metrics are output as logs
- Generated metrics are output as TFExamples to a file system
- Generated metrics are output as rows to BigQuery
- Generated metrics are output as a JavaScript Object Notation (JSON) string to Pub/Sub

The example pipeline is an application that uses the Apache Beam library and can run on the Dataflow service as well as other runners like Apache Flink. While the pipeline is running, one tick -- one price for a given point in time -- occurs every 500 ms, for which the function produces a repeating wave of data. SimpleDataStreamGenerator class contains the code that will be run. The simple data usage demonstrates the end-to-end data engineering of the library, from time series pre-processing to model creation using TensorFlow Extended.

Workflow

The tutorials walk users through the sequence of steps needed to build and use the sample library:

1. Options
2. Data ingestion
3. Generation of metrics based on the data
4. Output of the data to external systems

Options

The recommended way to configure the library is via pipeline options. In the example we hard code a few of the options for this sample application to make the sample easier to follow. Normally all options would be passed in via args in format --optionName=value

The SinWaveExampleOptions interface extends several interfaces from the library for the core options needed to configure the pipeline. It also adds a few options specific to the example's output paths, which can be modified when the pipeline is run.

```
SinWaveExampleOptions options =
   PipelineOptionsFactory.fromArgs(args).as(SinWaveExampleOptions.class);
```

Data ingestion

To build and use the sample library, the user's data needs to be converted into the library's internal representation of time series data points; the data is expected to come from a streaming data source like Pub/Sub. In the SinWaveExample, `GenerateSequence` is an Apache Beam transform which acts like a streaming source (unbounded) and helps simulate a source like Pub/Sub.

The time series data point, TSDatapoint, is a proto buffer that is populated with values from the user's data. Apache Beam transforms, referred to as Adaptors, are used to map properties from user data to TSDatapoints. TSDatapoints include these critical pieces of data:

- **Key:** The time series key, TSKey, comprises a major and minor key, which enable the multivariate time series data to be converted into univariate, which is required by the pipeline. At the end of the pipeline, data can be reassembled into multivariate, depending on user needs. For more information on TSKey and major and minor keys, including examples, see Converting Source Data.
- **Data:** The sample library currently supports only numeric values, of types like double or float, that can be set on TSDatapoint.
- **Timestamp:** The timestamp should match the timestamp provided by the source in Apache Beam. For example, in Pub/Sub there is the timestamp ID that can be provided as metadata to an element as it is put on the Pub/Sub topic; this should match the internal timestamp of TSDatapoint.

  Note: The timestamp is critical to the proper functioning of the sample library. This value should also match the timestamp attached to the element from the Source. For example use the TimestampedID option for PubSubIO source.

The SinWaveExample generates trivial data points that follow a very simple pattern. Over 12 hours the value will cycle through 0 up and then back to 0. There will be a tick every 500 ms. This is done in the method getSyntheticStream() in the SinWaveExample class, and used to create a PCollection of TSDataPoint.

```
PCollection<TSDataPoint> stream = getSyntheticStream(p, outlierEnabled, key);
```

To learn more about TSDataPoint, see Internal data types.

Metrics generation

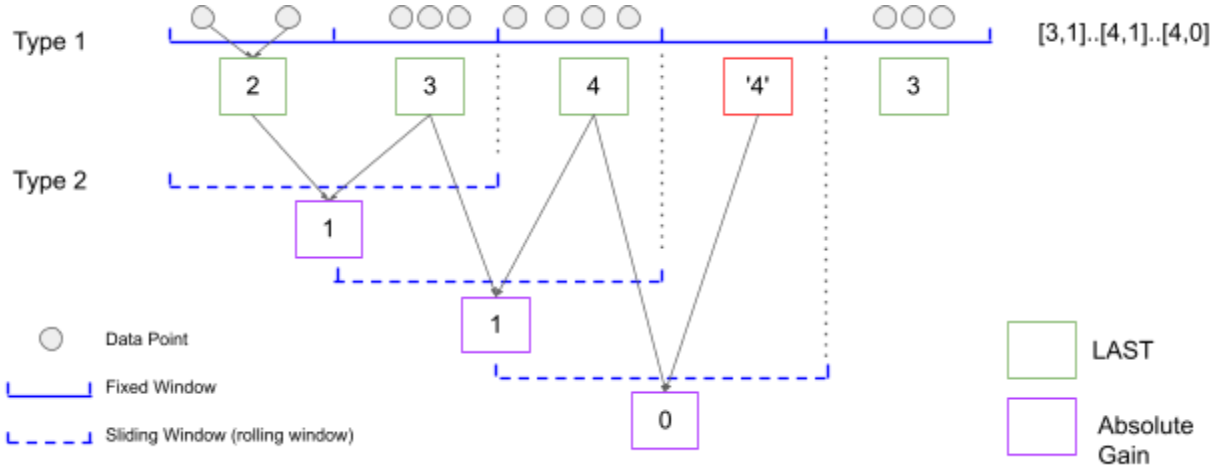
The sample library provides the GenerateComputations transform, which users must configure to specify the computations to be performed on the data.

Users first choose metrics to be passed to the Computations transform. The sample library uses two types of metrics:

- **Type 1:** The sample library requires all elements to pass through a Type 1 metric before passing to a Type 2 metric. A Type 1 metric is transitive and associative and acts as a reducing function, greatly decreasing the amount of data (by a temporal dimension) that needs to be processed for the second type of computation, or Type 2. Within a window, the Type 1 metric is the sum of the data points.
- **Type 2:** This computation requires a sequenced, ordered list of data points. Once the data passes through a Type 1 computation, users can create metrics like relative straight indexes or absolute gain.

For more information on metrics, including examples, see Metrics.

There are several windows values that must be set for this transformation. To understand these values, use the following diagram:



The first value to set is the length of the Type 1 fixed window, which generates transitive and associative aggregations like Min,Max,First,Last etc. This value becomes a 'fixed window' within Apache Beam which combiners use to combine our data into an aggregation.

```
options.setTypeOneComputationsLengthInSecs(1);
```

The second value to set is the length of the sliding window, this length must be a multiple of the Type 1 window. As seen in the diagram this property allows us to merge Type 1 and Type 2 data points by aligning the length of the slide to be exactly one fixed window length. If the Type 1 window is 1 sec and the Type 2 window is 5 sec with a slide every 1 sec we will have the ability to output a value for both Type 1 and Type 2 aligned to every second. For the first 5 seconds of the pipeline's life cycle, the sliding window will be incomplete as it will not have 5 seconds' worth of data points. This is the bootstrap problem that can be dealt with by 'bootstrapping' the pipeline with data from a historical source on startup.

```
options.setTypeTwoComputationsLengthInSecs(5);
```

As can be seen in the diagram the output is an array of size [metrics] every second. However, for many of the ML use cases there is a need to have a history of metrics, so an output of [timesteps,metrics] to achieve this the OutputTimestepLength should be used to define the length (in secs) of the history required. This value must also be a multiple of the fixed window length.

```
options.setOutputTimestepLengthInSecs(5);
```

Users need to configure their needs around gap filling, also called the perfect rectangle. Gap filling addresses situations in which there is sparse data across many columns in properties, which results in gaps; whereas in the perfect rectangle there is a data point for every line in the table across the x- and y-axis. Users choose whether they want to use gap filling, and they specify whether they want to use the default 0 to fill the gaps, or the last known value. For more information, see Gap filling.

To enable gap filling, set the option to `true`.

```
options.setGapFillEnabled(true);
```

By default when a gap is detected a zero is used to fill the gap. In this example we elect to make use of the default. For many use cases the value will be true, for example if this was the Price of a stock, then the last known value should be propagated into the gap as without new information it is the best known value.

```
options.setEnableHoldAndPropogateLastValue(false);
```

The amount of time that a specific key will output values in gaps before stopping. This can be set to MAX time if this value should be indefinite.

```
options.setTTLDurationSecs(1);
```

```
options.setTypeOneBasicMetrics(
ImmutableList.of("typeone.Sum", "typeone.Min", "typeone.Max"));

options.setTypeTwoBasicMetrics(
ImmutableList.of("typetwo.basic.ma.MAFn", "typetwo.basic.stddev.StdDevFn"));

options.setTypeTwoComplexMetrics(ImmutableList.of("complex.fsi.rsi.RSIGFn"));
```

```
GenerateComputations generateComputations =
   GenerateComputations.fromPiplineOptions(options).build();

PCollection<KV<TSKey, TSAccum>> computations = stream.apply(generateComputations);
```

To learn about TSAccum, see Internal data types.

Output results

The sample library supports multiple metrics output options, including a consistent snapshot of all inputs across time series into a single object for the output, which lets users see streamed output sequenced at a particular moment in time. For example, if a user has ten time series being processed, they will generate metrics that flow through the pipeline at different times. Taking snapshots of the output ensures that metrics are co-located in a single output object with consistent timing. Using this approach, the snapshot captures the value of, for example, time series 0 through 9, at a specific point in time.

Note: Users may experience performance limitations, as the process slows when there are large numbers of metrics to be processed.

In the SinWaveExample, metrics are output at different stages:

The metrics are sent to BigQuery before the above-described snapshot technique is applied.

```
DateTimeFormatter formatter = DateTimeFormat.forPattern("YYYY_MM_dd_HH_mm_ss");

if (options.getBigQueryTableForTSAccumOutputLocation() != null) {
 computations
     .apply(Values.create())
     .apply(new TSAccumToRow())
     .apply(
         BigQueryIO.<Row>write()
             .useBeamSchema()
             .withCreateDisposition(CreateDisposition.CREATE_IF_NEEDED)
             .withWriteDisposition(WriteDisposition.WRITE_APPEND)
             .to(
                 String.join(
                     "_",
                     options.getBigQueryTableForTSAccumOutputLocation(),
                     Optional.of("SimpleExample").orElse(""),
                     Instant.now().toString(formatter))));
}
```

The next output is done via a snapshot, which is carried out using the following transforms:

```
PCollection<KV<TSKey, TSAccumSequence>> sequences =
   computations.apply(
       ConvertAccumToSequence.builder()
           .setCheckHasValueCountOf(CommonUtils.getNumOfSequenceTimesteps(options))
           .build());


PCollection<Iterable<TSAccumSequence>> metrics =
   sequences.apply(MajorKeyWindowSnapshot.generateWindowSnapshot());
```

To learn more about TSAccumSequence, see Internal data types.

3. We can output our snapshots via:

```
PCollection<Example> examples = metrics.apply(new TSAccumIterableToTFExample());

examples.apply(OutPutTFExampleToFile.create());


examples.apply(

OutPutTFExampleToPubSub.create(options.getPubSubTopicForTSAccumOutputLocation()));
```

Before you begin

1. Install gradle.
2. Download the repository and git clone.
3. Build the library:

```
cd dataflow-sample-applications/timeseries-streaming/timeseries-java-applications/

./gradlew build
```

Output metrics as log files

Use this tutorial to spin up a local pipeline using the Direct Runner and send the metrics output to log files.

1. Run `./gradlew run_example --args='--enablePrintMetricsToLogs'.`
   You will see several messages about incomplete TSAccumSequences. Once there is enough data, which is 5 secs of data, you will see values output to the logs.
2. To stop metrics output, press **CTRL-C**.
3. To start the demo with outlier data included in the stream, which is a value outside of the norm every 50 ticks, run:

```
./gradlew run_example --args='--enablePrintMetricsToLogs --withOutliers=true'
```

Output generated metrics as TFExamples to a file system

1. Set up a directory where files will be output.
   In this example, we use `/tmp/simple-data/`:

```
mkdir /<your-directory>/simple-data
./gradlew run_example --args='--enablePrintMetricsToLogs
--interchangeLocation=/<your-directory>/simple-data/'
```

2. Check the folder you specified in `--interchangeLocation` for new files.
3. Start the demo with outlier data included in the stream, which is a value outside of the norm every 50 ticks:

```
./gradlew run_example --args='--enablePrintMetricsToLogs
--interchangeLocation=/<your-directory>/simple-data/ --withOutliers=true'
```

Output generated metrics as rows to BigQuery

Note: This procedure uses resources on your Google Cloud Platform account, which will incur costs.

1. Set up a BigQuery dataset, to which your local environment will send data.
2. Obtain BigQuery authentication for your user account. For more information, see Introduction to authentication.
3. In the following command, add your project ID and provide a table prefix, which the code will use when creating the table:

```
./gradlew run_example
--args='--bigQueryTableForTSAccumOutputLocation=<project>:<dataset>.<table_prefix>'
```

4. To observe the results, run the following SQL command in BigQuery:

```
SELECT
  lower_window_boundary,
  upper_window_boundary,
  is_gap_fill_value,
  DATA.metric,
  CASE DATA.metric
    WHEN 'FIRST_TIMESTAMP' THEN CAST(TIMESTAMP_MILLIS(DATA.lng_data) AS
STRING )
    WHEN 'LAST_TIMESTAMP' THEN CAST(TIMESTAMP_MILLIS(DATA.lng_data) AS
STRING)
  ELSE
  CAST(DATA.dbl_data AS STRING)
END
FROM
  `<yourtable>`
CROSS JOIN
  UNNEST(DATA) AS DATA
WHERE
  DATA.metric IN ("LAST",
    "FIRST",
    "DATA_POINT_COUNT",
    "FIRST_TIMESTAMP",
    "LAST_TIMESTAMP")
ORDER BY
  lower_window_boundary,
  DATA.metric
```

5. To start the demo with outlier data included in the stream, which is a value outside of the norm every 50 ticks, run the following:

```
./gradlew run_example
--args='--bigQueryTableForTSAccumOutputLocation=<project>:<dataset>.<table_prefix>
--withOutliers=true'
```

Output generated metrics as a JSON string to Pub/Sub

In this example, the values are sent to Pub/Sub and the interchangeLocation. The data sent to Pub/Sub is parsed into JSON format.

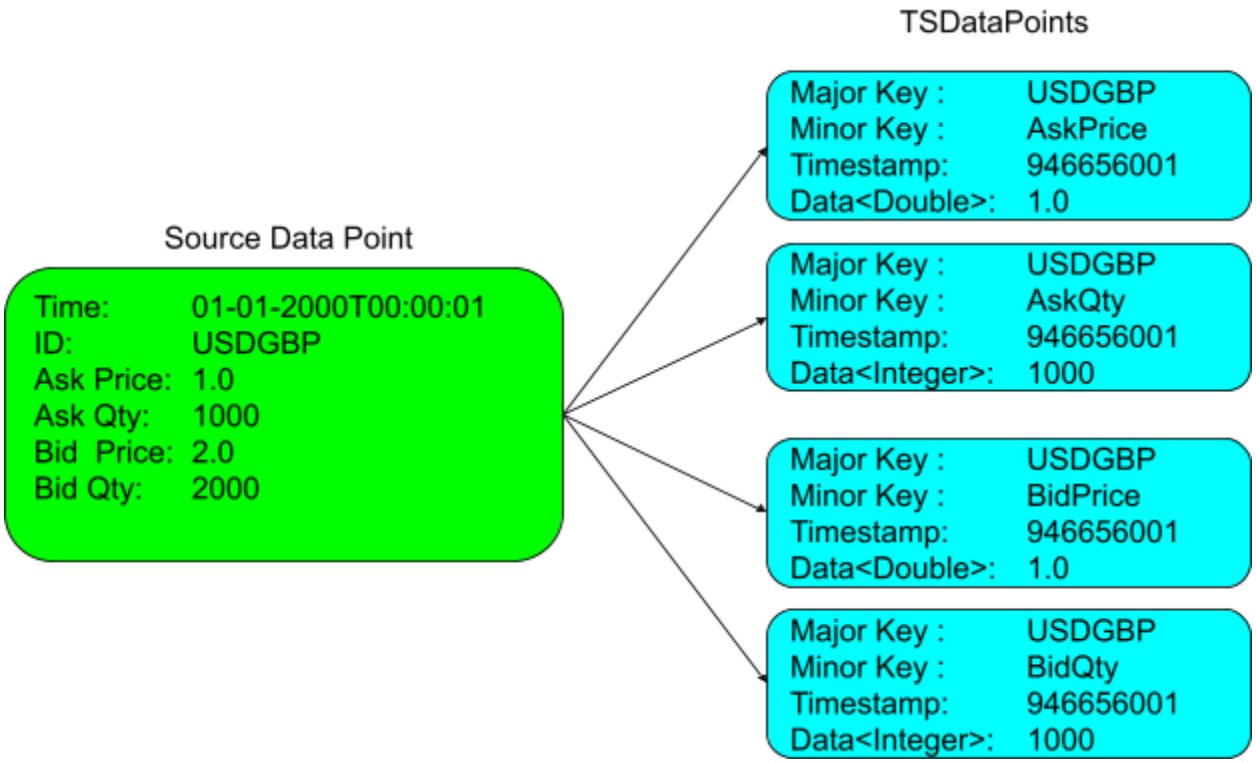
```
./gradlew run_example --args='--withOutliers=true
--pubSubTopicForTSAccumOutputLocation=projects/<your-project>/topics/outlier-detect
ion'
```

How-tos

Convert Source Data

While the library works with both batch and stream sources, the library's primary use case is stream sources. The library internal serialization mechanism is the TSDataPoint object, which is a protocol buffer. To learn more about the benefits of protocol buffer objects, read Why Use Protocol Buffers?

Because TSDataPoint is the data encapsulation that the library interacts with, users must convert incoming data to TSDataPoint objects. Users are responsible for the creation of a source `PCollection<TSDataPoint>`, which is then used in the Validation transform.



A TSDataPoint has several attributes:

Attribute	Description
TSKey	There are two components to any time series Key. The first is the Major key, which corresponds to the name of a Multivariate or Univariate time series. For example, with a financial services and insurance (FSI) scenario, this would be GBP-USD currency rates. For an Internet of Things (IoT) scenario, this would be a sensor that records one or many data points. The Minor Key, when present, is a single property of the Major key's values. For

	example, for GBP-USD, the Minor Key would be bid price or ask price. The minor key is important as it allows us to take a Multivariate time series value and break it into different properties which may have different types and require different computations. In a Multivariate time series there will be 1:n minor keys where n = number of properties of the time series.
Timestamp	This timestamp is an instance of a protocol buffer class and must contain the timestamp from the source data point object. For example, if the source is a Google Cloud Pub/Sub stream, then the stream should be read with a `PTransform` instantiated with `PubSubIO`'s `withTimestampAttribute()` method.
Data	This object encapsulates the minor key's value; the value's type will vary across instances.
Metadata	This attribute is currently unused in the library.

```
TSKey key = TSKey.newBuilder()
                 .setMajorKey("GBPUSD")
                 .setMinorKeyString("ask_price")
                 .build();

TSDataPoint.newBuilder().setKey(key)
                 .setData(Data.newBuilder().setDoubleVal(5.0D))
                 .setTimestamp(
                       Timestamps.fromMillis(timestamp)).build());
```

Validation

`TSDataPointVerifier`

This `PTransform` subclass carries out various checks against the incoming data and throws an `IllegalArgumentException` if any data points fail a check, resulting in pipeline failure.

The following checks are defined in `TSDatapointVerifier`:

Check	**Exception String**
TSDataPoint has a data value.	`TSDataPoint has no data.`
Data object does not have categorical data type.	`Categorical Values are not yet supported.`
TSDataPoints has a TSKey property.	`TSDataPoint has no key.`

TSDataPoint has a valid timestamp.	`TSDataPoint has no timestamp.`
The Beam timestamp of the method assigned by the source or previous transforms is not within the GlobalWindow's allowable range. If using a batch source, ensure that a TimestampedValue is used to ensure event time is set for the element.	`Detected TSDataPoints which are at min or max values for the Global Window.`

To implement a dead letter pattern, subclass `TSDataPointVerifier` with a class that sends a message to a dead letter queue if one of the class' checks fails.

Gap filling

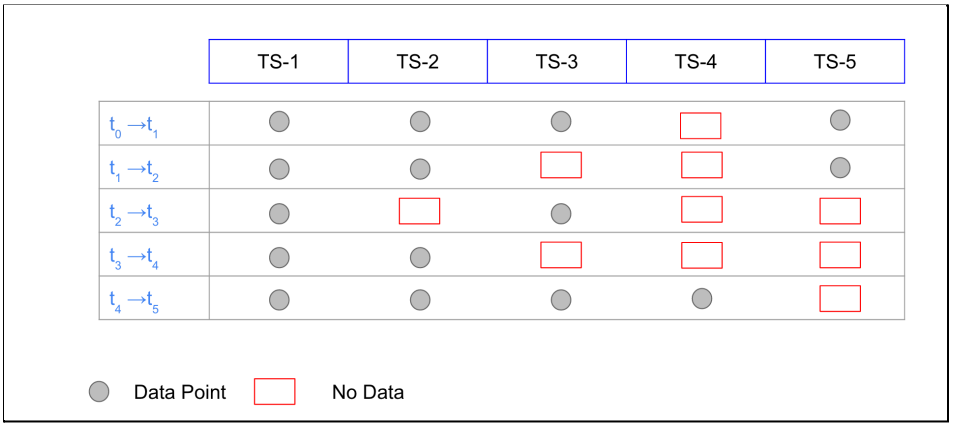
`PerfectRectangles`

A `GenerateComputations` transform that is configured to fill time series gaps should be instantiated using `generateComputations.setPerfectRectangles(PerfectRectangles.fromPipelineOptions(options)).`

Creating aggregations that rely purely on the data observed in the stream is straightforward. The Dataflow windowing functions, requiring only a couple of lines of code, can deal with the transitive and associative aggregations of finding the first, last, min, and max values within a time window. However, if the specific stream experiences periods of no data, the situation is not so straightforward.

Let's look at some examples in the table below, where:

- The y-axis represents time intervals, for example 10:00:00 to 10:00:05, which Apache Beam represents as [10:00:00, 10:00:05].
- The x-axis denotes four different time series, TS1 to TS5.



There are 4 cases to consider:

- TS-1: There is continuous data within the stream for all data windows.
    - This is the base case and while it is simple to process, unless the windows are large, most streams do not have this characteristic.
- TS-2 & TS-3: There are missing data points in the stream, but the key will continue to send data at some point.
    - How this is dealt with in stream mode, will depend on the use case;
        - If no data / action is required on the absence of data, then the basic window functions within Apache Beam can be used.
        - If we do want data when there is no data entering the pipeline for a given window, then we can make use of the Looping timers pattern to create an entry point for that window.
- TS-4: There is no data at the bootstrap phase of the stream when compared with other time series in the same data flows.
    - If a key has not been seen when the streaming pipeline starts, then the system does not know the key exists and the looping timer cannot start. This problem can be solved by bootstrapping the pipeline with a flush of all needed keys on startup. The downside of this approach is that the data used for the flush must originate from the past, and this can cause duplicated data within the pipeline.
- TS-5: There is some data, but then no more data arrives for that key.
    - A simple example of this is an IOT device that is decommissioned and will never send data again. In this case if we have chosen to do gap-filling as described above, then we will need a way to signal a stop to the system.

In order to work through these use cases we will need to make use of the PerfectRectangles class, so named because it generates a full set of data points for each time interval, even when no data points (or sparse data points) were observed during that interval.
To configure and instantiate this class, call any combination of these four methods:

Option	**Description**
`enablePreviousValueFill()`	Configures the returned instance to use the last known Data value when no such value is observed during a time interval.
`withWindowAndTTLDuration()`	Specifies the returned instance's fixed window duration, as well as the time interval that the returned instance will wait to complete gap filling, following the last observed value for a minor key.
`withWindowAndAbsoluteStop()`	Specifies the returned instance's fixed window duration, as well as the instant at which gap filling will stop. This is useful for bootstrap pipelines which backfill metrics.

`withPreviousValueFillExcludeList()`	Specifies an optional list of TSKeys for which the returned instance will not carry forward the last known Data value, even when enablePreviousValueFill is set. If the TSKey major key is not set then the minor key will be assumed to match all major keys.

Example 1 - Similar to TS-2 in table above. Given a time series with Key (GBPUSD, AskPrice) and values:

Time	Price
00:00:00	1.0
00:00:01	1.5
00:00:02	No Data
00:00:03	No Data
00:00:04	2.0

The output with different values of TtlDuration with `enablePreviousValueFill=True`

Time	Price	TtlDuration = 1	TtlDuration = 2
00:00:00	1.0	1.0	1.0
00:00:01	1.5	1.5	1.5
00:00:02	No Data	1.5	1.5
00:00:03	No Data	No Data	1.5
00:00:04	2.0	2.0	2.0

The output with different values of TtlDuration with `enablePreviousValueFill=False`

Time	Price	Stop at 00:00:02	Stop at
00:00:00	1.0	1.0	1.0
00:00:01	1.5	1.5	1.5
00:00:02	No Data	0	0

00:00:03	No Data	No Data	0
00:00:04	2.0	2.0	2.0

Metrics

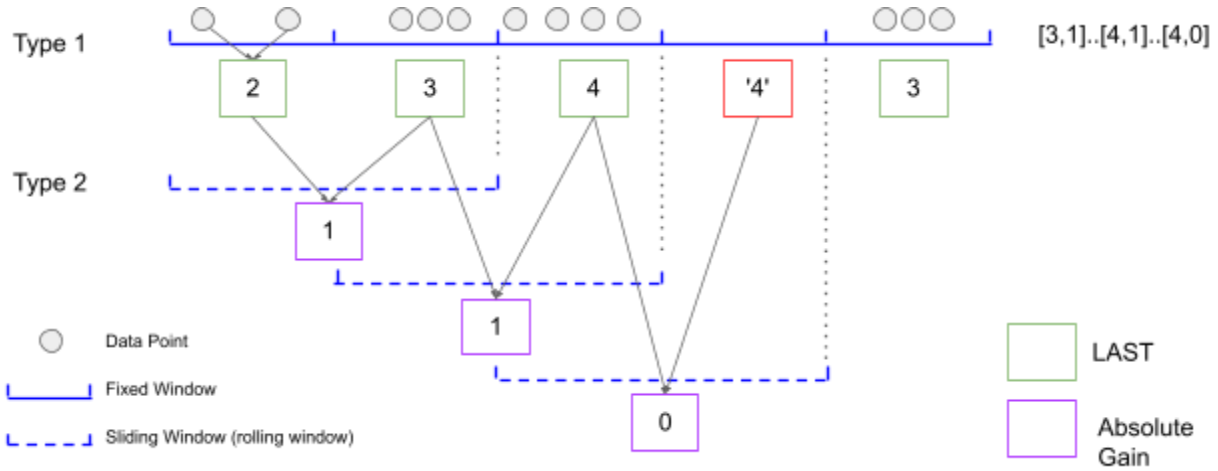
The samples provide a framework to compute two forms of metrics, Type 1 and Type 2.

- **Type 1:** These metrics are transitive and associative computations that can be computed in a distributed manner, for example SUM, MIN, MAX, FIRST and LAST.

- **Type 2:** These metrics are computations that need to happen per key per window using an ordered sequence of data. For example, the absolute gain or from FSI domain, relative strength index. An important note, the library only allows for the computation of Type 2 metrics from Type 1 metrics, this is for scaling and performance considerations and is discussed in more detail in the Type 2 section of the guide.

To learn more about generating metrics, see Computing Time Series metrics at scale in Google Cloud.

In the example below, the Type 1 metric LAST is being used; this is the LAST value within the window. The Type 2 computation uses the LAST value to compute the absolute gain.



Set window options for metrics creation

When instantiating the `GenerateComputation` class, pass a `TimeseriesStreamingOptions` class. On `TimeseriesStreamingOptions`, use the below setter methods to set the window options used for the metrics creation:

Option	Description
`setTypeOneComputationsLengthInSecs`	The window duration which is used for the computation of Type 1 metrics.
`setTypeTwoComputationsLengthInSecs`	The sliding window duration used for the computation of Type 2 metrics.

Merged output

The output from both Type 1 and Type 2 computations is always output in lockstep, which is achieved by the library not allowing the sliding window offset value to be different from the Type 1 window lengths. As shown in the diagram below, this approach allows for clean output arrays where Type 1 and Type 2 values for the close of the window are output together. Note at the start of the window there will be no Type 2 values until the sliding window length has completed. These outputs are filtered out by the library.

Type 1 and Type 2 metrics are available out of the box.

Type 1

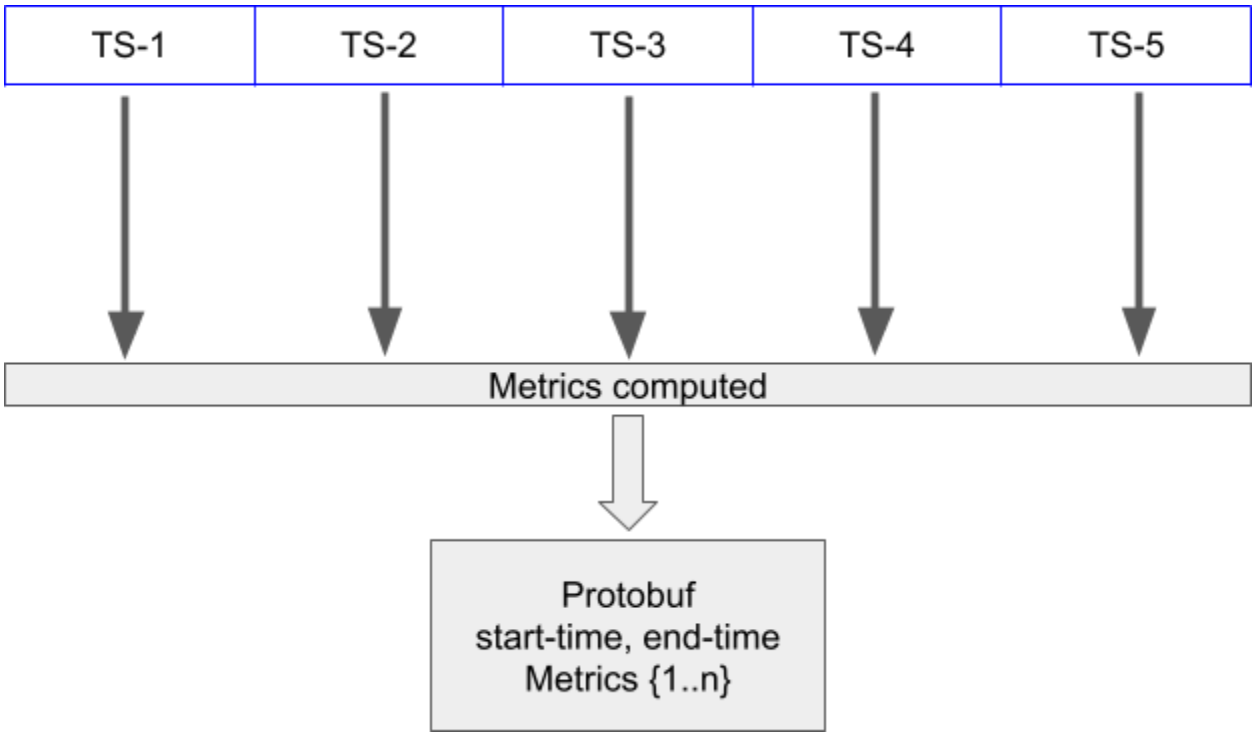
Metric	**Description**	**Symbol**	**Options**
sum	The sum of all the values within the Window.	SUM	N/A
min	The min of all values within the window	MIN	N/A
max	The max of all values within the window	MAX	N/A
last	The last value within the window	LAST	N/A
first	The first value within the window	FIRST	N/A
last timestamp	The last timestamp	LAST_TIMESTAMP	N/A

	seen in the window		
first timestamp	The first timestamp seen in the window	FIRST_TIMESTAMP	N/A

Type 2

Metric	**Description**	**Symbol**	**Options**
Moving Average	The simple or exponential moving average	SIMPLE_MOVING_AVERAGE EXPONENTIAL_MOVING_AVERAGE	setMAAvgComputeMethod
Standard Deviation	Measures volatility/risk/dispersion	STANDARD_DEVIATION	N/A
Bollinger Bands	Measures price volatility in bands	BB_MIDDLE_BAND_SMA/EMA BB_UPPER_BAND_SMA/EMA BB_BOTTOM_BAND_SMA/EMEA	setBBAvgComputeMethod setBBDevFactor setBBAlpha
Relative strength index	Measures historical and current pricing trends	RELATIVE_STRENGTH_INDICATOR	N/A
Log Returns	Continuously Compounded Return	LOG_RTN	N/A

Time snapshots



Integrate with TFX via tf.train.Example serialization

There are two areas where the data from the library can be used using Tensorflow Extended (TFX), an end-to-end platform for deploying production ML pipelines. The important connection between the folowing paths is the ability to output the data serialized in tf.train.Example format. The library has several helpers which do this translation for us.

Features for model building

The metrics that the library produces can be used as features for building models, for example, through long short term memory (LSTM) autoencoder/decoder models which can be used for anomaly detection. The LAST Type 1 metric can be a simple 'raw' feature, but also computations like moving average or relative strength index (RSI) could be valuable for a model. TFX provides ImportExampleGen as a component that can convert tf.train.Example or tf.train.SequenceExample protos stored in TFRecord files ready for processing by the machine learning pipeline.

Features to inference

By building the model using an output of features generated by the library, we can also make use of the library to run inference ensuring a hermetic seal between model building and streamed inference. This is achieved using tfx-bsl RunInference transform.

Note: When the sample library was originally built, TFX did not support tf.train.SequenceExample as a import format out of the box. As a consequence, the library overloaded the use of tf.train.Example to be able to accommodate sequences of ordered input, which is needed for time series data. Later on we describe the layout of the data within tf.train.Example format.

Within the Dataflow streaming time series library there are three primary data storage objects, `TSDataPoint`, `TSAccum`, and `TSAccumSequence`. TSAccumSequence provides a sequence of timesteps for metrics. For example 20 timesteps of the MIN/MAX/LAST/FIRST value and this is the data structure that we convert to a format for consumption by ImportExampleGen from the TFX library.

tf.train.example output is a dictionary of values, with two categories: Metadata, which describe the example, and the data points themselves. As the values are in tf.example format, the nesting of the JSON human readable format will be:

```
{
  "features": {
    "feature": {
```

Where `feature` has both the Metadata and data values[1].

Metadata

All metadata features start with METADATA_.

METADATA_SPAN_START_TS	The start of the SPAN of data in Timestamp.
METADATA_SPAN_END_TS	The end of the SPAN of data in Timestamp.

Data

Data arrays have an array of data. The array length is (METADATA_SPAN_END_TS - METADATA_SPAN_START_TS) / FixedWindow size of Type 1 computation.

In the GBP-USD case, Type 1 computations will generate First / Last / Min / Max which is the same as OHLC in finance terms. This will have a period equal to the Type 1 Fixed window. So if we have Span End - Span Start of 2 mins and a fixed window of 1 sec, then the array will contain 120 elements.

---

[1] When the samples were first built, TFX did not support TF.ExampleSequence, which resulted in the overloading of TF.Example. This will be changed in the future.

In tf.example, data is always represented as one of :

```
feature-name : {
        "floatList": {
          "value": [1.0,2.0,3.0.x]

feature-name : {
        "int64List": {
          "value": [0,2,3.x]
```

Feature names

Feature names are constructed using the Major and Minor keys from TSKey. For the GBP-USD time series, these will be of the form:

```
GBP-USD-ask_price-METRIC_NAME
GBP-USD-bid_price-METRIC_NAME
```

Window Snapshot

For consumers of metrics from the time series library, there is a need to have continuous snapshots of all keys within a specific window of time.

Using the FSI example, given three time series for prices for the currency futures contracts GBP-USD, USD-EUR, and USD-JPY, we would want all Type 1 and Type 2 metrics from all major keys to be available every 30 secs to produce a snapshot of the market.

Time - 00:00:00

```
{GBP-USD: {FIRST: 1.30, LAST:1.31}, USD-JPY: {FIRST:106.27, LAST:106.28} , USD-EUR:
{FIRST: 0.85, LAST:0.86}}
```

Time - 00:00:30

```
{GBP-USD: {FIRST: 1.30, LAST:1.31}, USD-JPY: {FIRST:106.27, LAST:106.28} , USD-EUR:
{FIRST: 0.85, LAST:0.86}}
```

Constraints

- The snapshot should be an all-or-nothing output; partial outputs are not acceptable.
- The output should be based on event time, not processing time.
- The output must be atomic, as there needs to be a clear signal that the output has completed.

Window snapshot support for TSAccum and TSAccumSequence

There are two types of accumulators in the library, TSAccum and TSAccumSequence. It should be up to the user which of these they want as output. The TSAccumSequence is the primary snapshot consumed by the LSTM models already.

Note: These are protocol buffer classes with automatically generated getter and setter methods.

Implementation

As this is an area where performance is a consideration, we will cover a little of the implementation details.

All keys in the time series library have a major/minor component.

Note: Both the TSAccum and TSAccumSequence objects have TSKey as a property. However, very often in the library we have PCollections of <KV<TSKey,TSAccum>> or <KV<TSKey,TSAccumeSequence>>, these KV allow us to manipulate the TSKey used for grouping functions without mutating the 'real' TSKey value in the Accumulator.

For performance reasons, we change the Major-Key value to equal the interval window object from Apache Beam. This can cause namespace clashes in the Minor-Key value, as shown in this example:

{Major-Key : USD-GBP, Minor-Key : LAST}, {Major-Key : USD-JPY, Minor-Key : LAST}

To create this snapshot, the user needs to carry out these steps:

1. Reify the major-key into the minor-key:
   {Major-Key : USD-GBP, Minor-Key : USD-GBP-LAST}, {Major-Key : USD-JPY, Minor-Key : USD-JPY-LAST}
2. Change the Major-key to become the Window Context.
3. GBK.
4. Create a Snapshot object that contains all of the information.

Performance

Each window can be processed independently of other windows; however, as an optimization step, some runners will have all values with a specific Key end up at the same machine. By creating the Window as the key, we ensure that different windows can end up on different machines.

Singleton snapshot object

All data coming to this part of the pipeline has gone through a reducer across a temporal dimension. For example, stock ticks will have moved through Type 1 computations with a fixed window size of 30 sec. In many use cases, this reduction allows for the snapshot object to become a singleton object which is serialized out of the pipeline as a single Record.

While singleton snapshot objects are easy to implement, consider that the entire snapshot needs to be something that can be loaded into the memory of a single worker. The amount of memory used will be:

Window Duration * number of keys * Σ(feature_metdata+ features_byte_size)

Appendix A: Internal data types

There are three core objects used within the time series pipelines which are defined in TS.proto

1. TSDatapoint
2. TSAccum
3. TSAccumSequence

The TS prefix stands for “time series” and is used to distinguish the objects specific to this library.

The rationale for using a public proto type rather than an internal representation are:

- The time series objects used within the pipeline also need to exist outside of the pipeline.
- The user of the library should be free to provide any data type (Integer, Float, Double, String) and have the library be able to process the data as-is and without conversion.
- The data object should hold an arbitrary list of data points and computations with arbitrary type along with metadata.

Data

The data proto holds a single datapoint object which is a oneof for different types of data. This is similar in concept to TensorFlow’s tensor.proto; however, that proto makes use of a DataType to describe the type of a proto. In this data object, the oneof enforces only a single type of data point.

TSKey

There are two components to any time series Key:

- The Major key, which corresponds to the name of a Multivariate or Univariate time series. For example, in an FSI scenario, this would be GBP-USD currency rates. For an IOT scenario, this would be a sensor that records one or many data points.
- When present, the Minor Key is a single property of the Major keys values. For example, for GBP-USD it would be bid price or ask price. The minor key is important as it allows us to take a Multivariate time series value and break it into different properties which may have different types and require different computations.

TSDataPoint

The TSDataPoint holds both the metadata and the data about the time series data point. It contains the Data point, a timestamp and a Map to represent any metadata that would need to be added to the data point.

TSAccum

The TSAccum contains the derived data from a univariate time series. For example it will hold the First, Last, Sum, Min, Max values for a specific MajorKey-MinorKey. This object is the primary object holding data after computations and makes use of a Map to ensure any data point can be represented.

TSAccumSequence

The Accum Sequence is an ordered collection of Accums, along with metadata about the window boundaries and associated metadata.