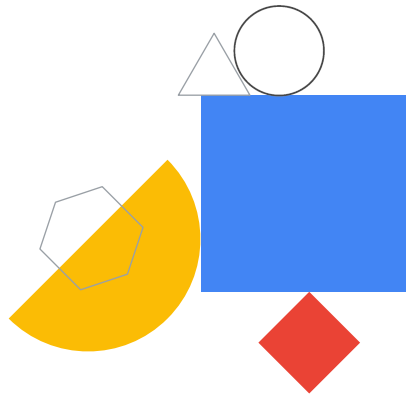# Dataflow Streaming Features

In the previous module, we discussed using Pub/Sub to receive streaming data from a variety of sources. Now we are ready to process it or prepare it for further analysis. Let us see how Dataflow can help you to do this. More specifically, let's look at Dataflow's streaming features.

Module agenda

Google Cloud

In this module, we'll discuss some of the challenges associated with streaming data, and then the different windowing capabilities provided by Dataflow.

01

Streaming Data Challenges

Let's start by identifying some of the challenges associated with processing streaming data.

Streaming features of Dataflow

Qualities that Dataflow contributes to data engineering solutions:
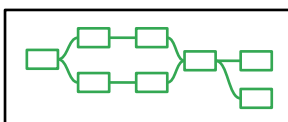
✓ Scalability
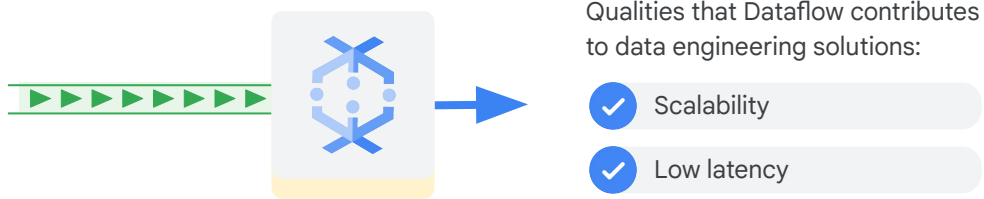
✓ Low latency

Dataflow, as we already know, provides a serverless service for processing batch and streaming data. It is scalable and for streaming has a low latency processing pipeline for incoming messages.

Continuing from the Data Processing course

Unbounded PCollection

Pipeline

Streaming Jobs

> job

We have discussed that we can have bounded and unbounded collections, so now we are examining an unbounded pipe that results from a streaming job. All of the things we have done thus far like, branching, merging, we can do all as well with Dataflow for streaming pipelines.

However, now, every step of the pipeline is going to act in real time on incoming messages rather than in batches.

There are challenges with processing streaming data

Scalability

Streaming data generally only grows larger and more frequent

Fault Tolerance

Maintain fault tolerance despite increasing volumes of data

Model

Is it streaming or repeated batch?

Timing

What if data arrives late?

Google Cloud

What are some of the challenges with processing streaming data?

- One challenge is **scalability**, being able to handle the volume of data as it gets larger and/or more frequent.
- The second challenge is **fault tolerance**. The larger you get, the more sensitive you are to going down unexpectedly.
- The third challenge is the **model** being used, streaming or repeated batch.
- Another challenge is timing or the latency of the data. For example, what if the network has a delay or a sensor goes bad and messages can't be sent?

How do you aggregate an unbounded set?

Data Stream

X

Non-aggregating operations such as filtering are straightforward

| 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
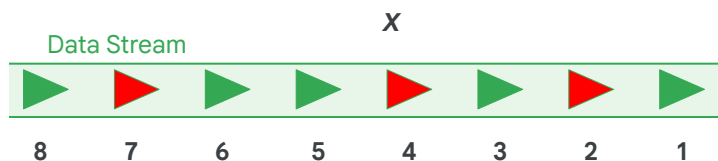
But how do you take an average on an unbounded stream of data?

$$\sum_{i=1}^{n} x_i$$

What is the stopping limit?

Data Stream

| 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |

Google Cloud
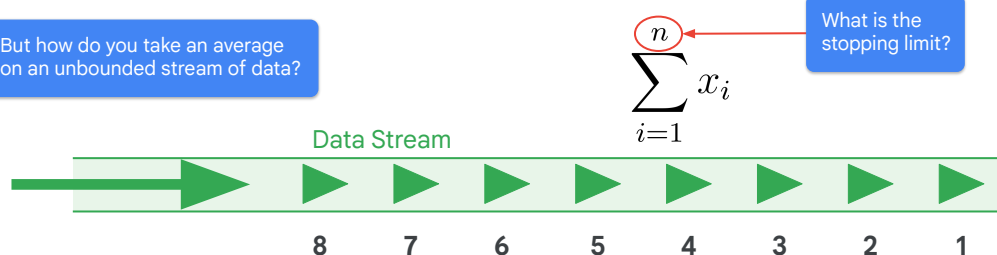
Additionally, there is a challenge around any kind of aggregation you might be trying to do. For example, if you are trying to take the average of data, but it is in a streaming scenario. You cannot just plug values into the formula for an average, the sum from 1 to n, because n is an ever-growing number.

Divide the stream into a series of finite windows
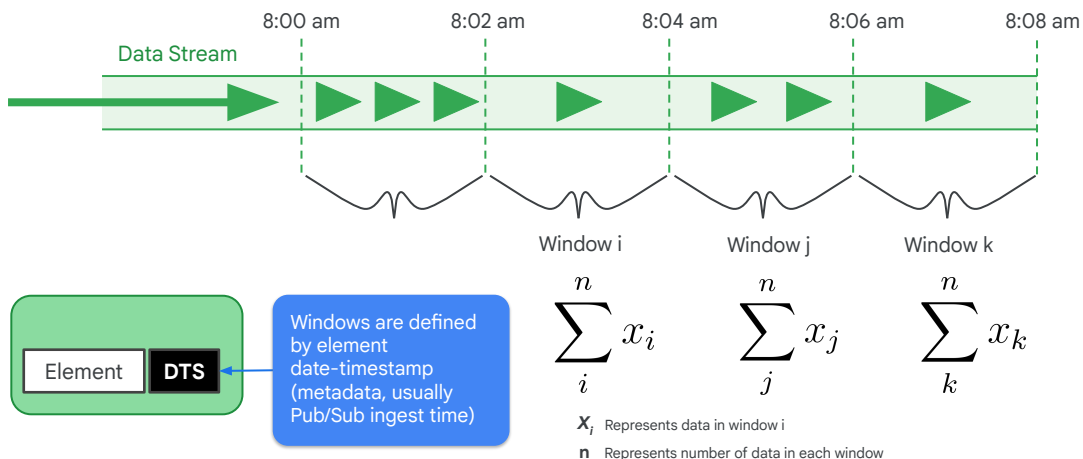
So, in a streaming scenario, you have to divide time into windows and we can get the average within a given window. This can be a pain if you have ever had to write a system like this. You can imagine it can be difficult to maintain windowing, time roll threads, etc,. The good news is that Dataflow is going to do this for you automatically.

In Dataflow, when you are reading messages from Pub/Sub, every message will have a timestamp that is a Pub/Sub message timestamp and then you will be able to use this timestamp to put the data into the different time windows and aggregate all of those windows.

Message ordering and late data: The timestamp matters ... and windowing

Pub/Sub publisher code

```
msg.publish(event_data, mytime="2020-04-12T23:20:50.52Z")
```

| Data | **DTS** |

Dataflow pipeline code

```
p.apply( PubsubIO.readStrings()
    .fromTopic(t).withTimestampAttribute("mytime") )
```

Element

| Data | **DTS** | **DTS** |

| Publisher event creation time | **lag** | Pub/Sub event arrival time |
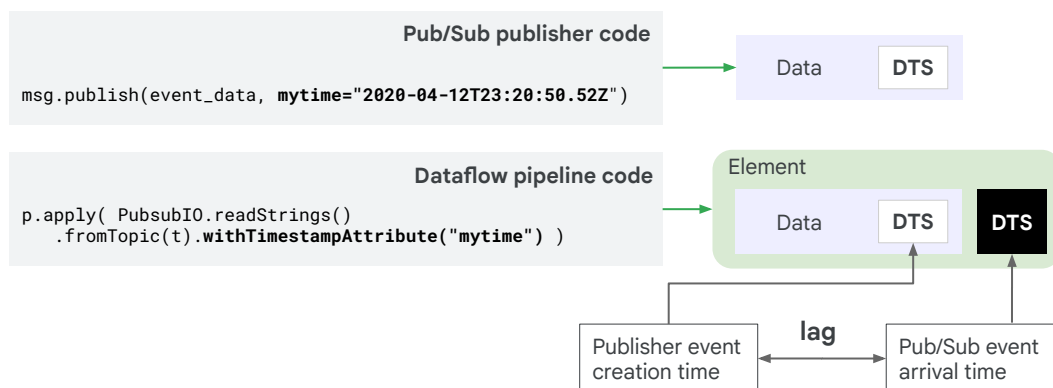
Message ordering matters and there may be a latency between the time that a sensor is read and the message is sent. You may need to modify the timestamps if this latency is significant.

Modify the date-timestamp with a PTransform if needed
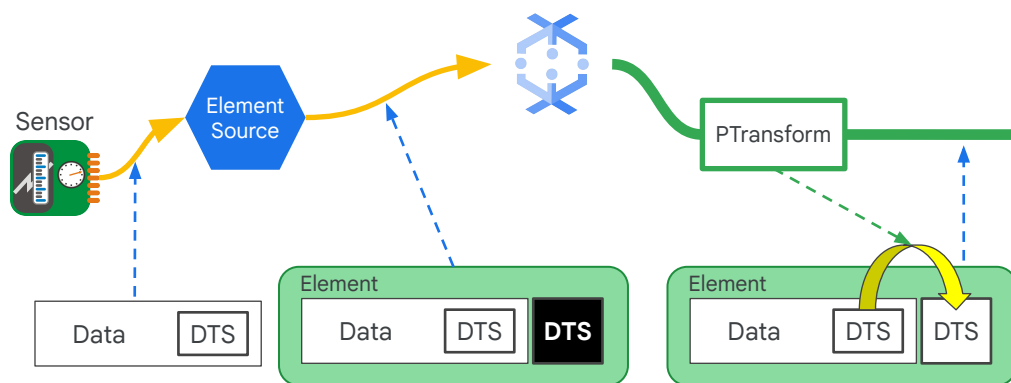
If you want to modify a timestamp and have it based on some property of your data itself, you can do that. Every message that comes in, for example, the sensor provides its own date-timestamp as part of the message.

Element source adds a default date-timestamp or DTS which is the time of entry to the system, rather than the time the sensor data was captured.

A PTransform extracts the date-timestamp from the data portion of the element and modifies the DTS metadata so the time of data capture can be used in window processing.

Code to modify date-timestamp

Python

```
unix_timestamp = extract_timestamp_from_log_entry(element)
    # Wrap and emit the current entry and new timestamp in a
TimestampedValue.
    yield beam.window.TimestampedValue(element, unix_timestamp)
```

Java

```
c.outputWithTimestamp (element, timestamp);
```

PTransform

Element

Data | DTS | DTS

Google Cloud
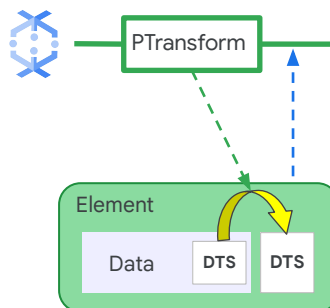
Here is the code used to modify the date timestamp by replacing the message timestamp with the timestamp from the element data.

Duplication will happen: Exactly-once processing with Pub/Sub and Dataflow

Pub/Sub publisher code

```
msg.publish(event_data, myid="34xwy57223cdg")
```

Dataflow pipeline code

```
p.apply(
   PubsubIO.readStrings().fromTopic(t).idLabel("myid") )
```

Google Cloud

If PubsubIO is configured to use custom message IDs, Dataflow deduplicates messages by maintaining a list of all the custom IDs it has seen in the last 10 minutes. If a new message's ID is in this list, the message is assumed to be a duplicate and discarded.

02

Dataflow Windowing

Next, let's look at Dataflow Windowing capabilities. This is really Dataflow's strength when it comes to streaming.

Three kinds of windows fit most circumstances

- Fixed (also called Tumbling in Dataflow)
- Sliding (also called Hopping in Dataflow)
- Sessions

Google Cloud

Dataflow gives us three different types of windows, fixed, sliding, and sessions. Apache Beam calls the first a Fixed Window, whereas Dataflow also calls it Tumbling. Same for the second one, it's called Sliding in Apache Beam and Hopping in Dataflow.

Three kinds of windows fit most circumstances

Fixed (Tumbling) · **Sliding (Hopping)** · **Sessions**

Windowing divides data into time-based finite chunks

Often required when doing aggregations over unbounded data
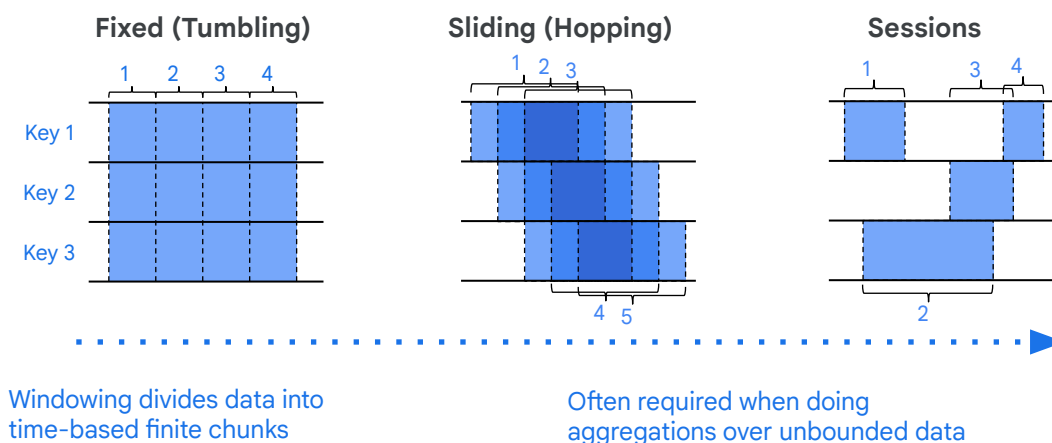
Fixed windows are those that are divided into time slices, for example, hourly, daily, monthly. Fixed time windows consist of consistent non-overlapping intervals.

Sliding windows are those that you use for computing. For example, give me 30 minutes worth of data and compute that every 5 minutes. Sliding time windows can overlap, for example, in a running average. Sliding windows are defined by a minimum gap duration and the timing is triggered by another element.

Session windows are defined by a minimum gap duration, and the timing is triggered by another element. Session windows are for situations where the communication is bursty. It might correspond to a web session. An example might be if a user comes in and uses four to five pages and leaves. You can capture that as a session window. Any key in your data can be used as a session key. It will have a timeout period, and it will flush the window at the end of that time out period.

Setting time windows

Fixed-time (Tumbling) windows

```python
from apache_beam import window                                    Python
fixed_windowed_items = (
    items | 'window' >> beam.WindowInto(window.FixedWindows(60)))
```

Sliding time (Hopping) windows

```python
from apache_beam import window                                    Python
sliding_windowed_items = (
    items | 'window' >> beam.WindowInto(window.SlidingWindows(30, 5)))
```

Session windows

```python
from apache_beam import window                                    Python
session_windowed_items = (
    items | 'window' >> beam.WindowInto(window.Sessions(10 * 60)))
```
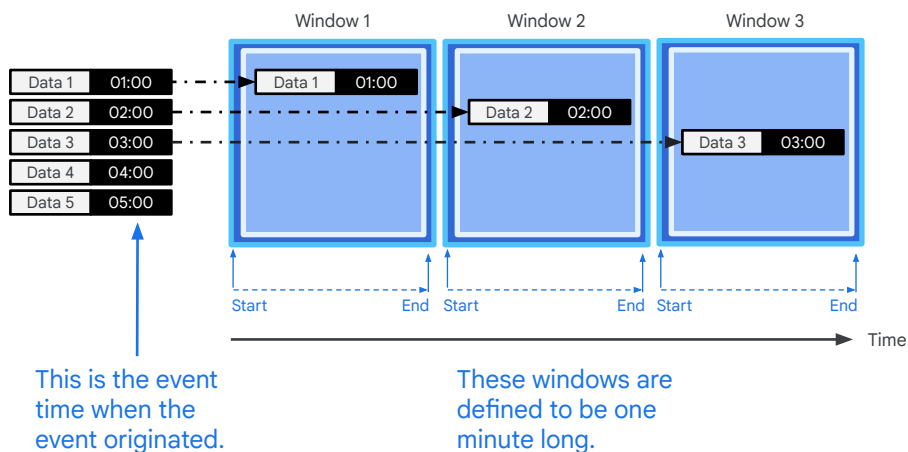
Remember:
you can apply windows to batch data, although you may need to generate the metadata date-timestamp on which windows operate.

Google Cloud

Here is how we can set these different types of Windows in Python. In the fixed-time window example, we can use the functions beam.WindowInto, and window.FixedWindows with argument 60 to get fixed windows starting every 60 seconds.

In the second example, with a sliding time window we use window.SlidingWindows with argument 30 and 5. Here, the first argument refers to the length of the window, that is, 30 seconds. And the second argument refers to how often new windows open, that is, five seconds.

Finally, we have the example of a session window. We use windows. Sessions with an argument of 10 multiplied by 60 to define a session window with time out of ten minutes, that is, 600 seconds.

Windowing by time if there is no latency

Window 1   Window 2   Window 3

Data 1	01:00
Data 2	02:00
Data 3	03:00
Data 4	04:00
Data 5	05:00

This is the event time when the event originated.

These windows are defined to be one minute long.

How does windowing work? All things being equal, this is how windowing ought to work. If there was no latency, if we had an ideal world, if everything was instantaneous, then these fixed time windows would just flush at the close of the window. At the very microsecond at which it becomes 8:05:00, a five minute window terminates, and flushes all of the data. This is only IF there is no latency.

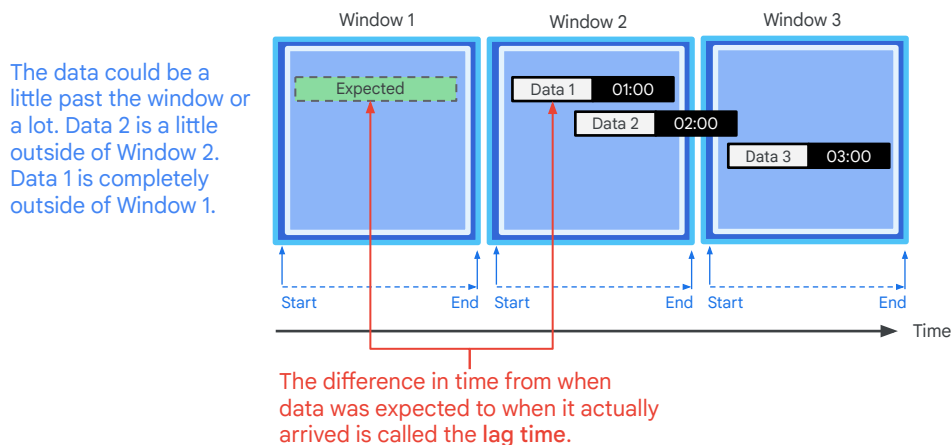# Pipeline processing can introduce latency

Pipeline

Data 1 | 01:00
Data 2 | 02:00
Data 3 | 03:00
Data 4 | 04:00
Data 5 | 05:00

Data 1 | 01:00
Data 3 | 03:00
Data 2 | 02:00

This is the event time when the event originated.

The time relationships at origination are not preserved.

They are arriving later than anticipated. And some of them are outside the original 1 minute window.
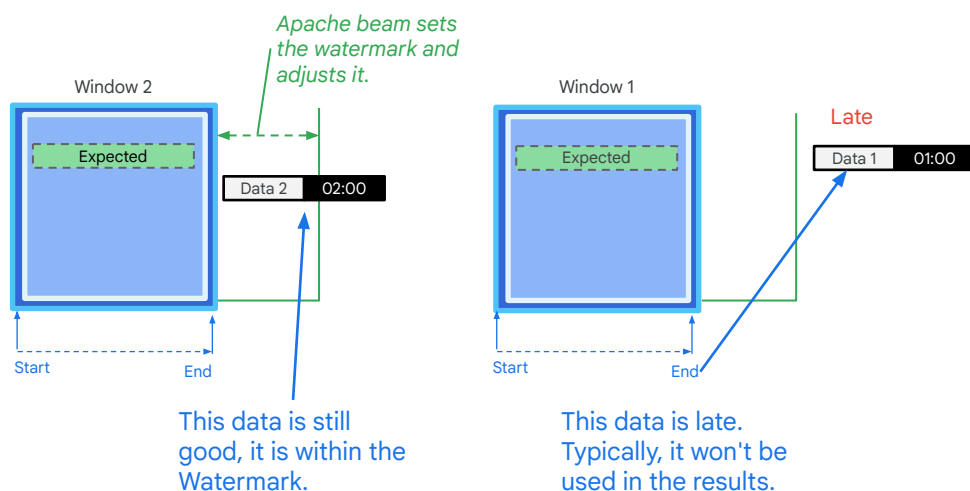
Google Cloud
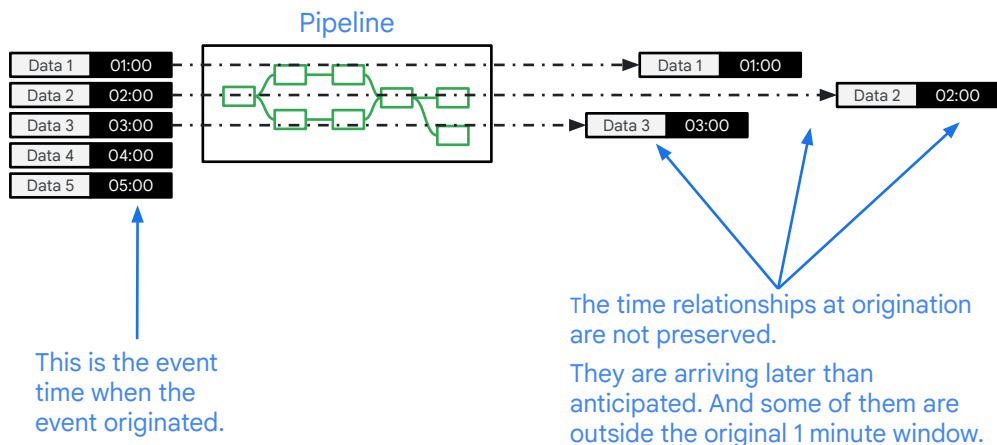
But in the real world, latency happens. We have network delays, system backlogs, processing delays, Pub/Sub latency, etc., So, when do we want to close the window? Should we wait a little bit longer than 8:05, maybe a few more seconds?

How should Dataflow deal with this situation?

The data could be a little past the window or a lot. Data 2 is a little outside of Window 2. Data 1 is completely outside of Window 1.

Window 1   Window 2   Window 3

Expected

Data 1   01:00

Data 2   02:00

Data 3   03:00

Start   End   Start   End   Start   End

Time

The difference in time from when data was expected to when it actually arrived is called the **lag time**.

This is what we call the watermark and Dataflow keeps track of it automatically. Basically, it is going to keep track of the lag time. For example, if you are using the Pub/Sub connector, because it knows the time of the oldest, unprocessed message in Pub/Sub. And also knows the latest message it has processed through Dataflow. It, then, takes this difference and that is the lag time.

Watermarks provide flexibility for a little lag time

Apache beam sets the watermark and adjusts it.

Window 2 — Expected — Data 2 — 02:00
Window 1 — Expected — Late — Data 1 — 01:00

Start — End

This data is still good, it is within the Watermark.

This data is late. Typically, it won't be used in the results.

So, what Dataflow is going to do is continuously compute the watermark, which is how far behind we are. Dataflow ordinarily is going to wait until the watermark it has computed has elapsed, so if it is running a system lag of three of four seconds, it is going to wait four seconds before it flushes the window because that is when it believes all of the data should have arrived for that time period.

What, then, happens to late data? Let's say it gets an event with a timestamp of 8:04, but now it is 8:06. It is two minutes late, one minute after the close of the window, what does it do with that data? The answer is, you get to choose that. The default is just to discard it, but you can also tell it to reprocess the window based on those late arrivals.

Beam's default windowing configuration tries to determine when all data has arrived (based on the type of data source) and then advances the watermark past the end of the window. This default configuration does not allow late data.

Custom triggers
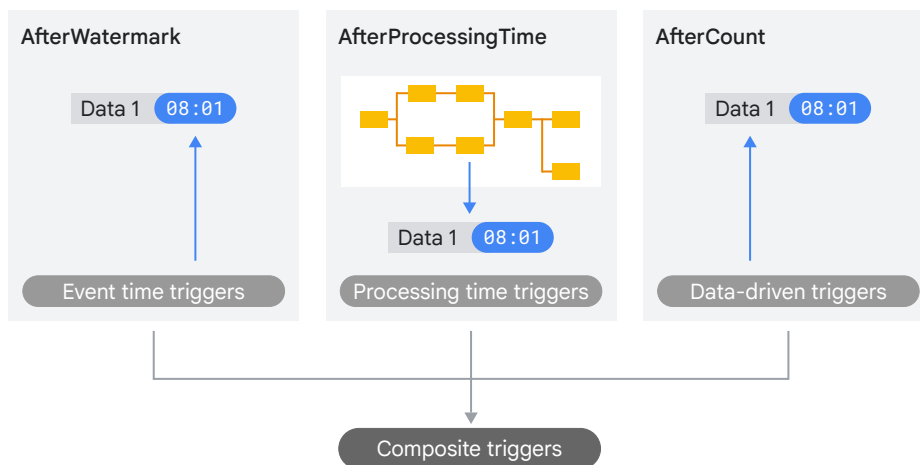
The default behavior is to trigger at the watermark. So if you don't specify a trigger, you are actually using the trigger "AfterWatermark".

"AfterWatermark" is an event time trigger. We could also apply any other trigger using event time. The messages' timestamps are used to measure time with these triggers.

But we can also add custom trigger(s).

If the trigger is based on processing time, the actual clock / real time is used to decide when to emit results. For instance, you could decide to emit exactly every 30 seconds, regardless of the timestamps of the messages that have arrived to the window.

"AfterCount" is an example of a data-driven trigger. Rather than emitting results based on time, here we trigger based on the amount of data that has arrived within the window.

The combination of several types of triggers opens a world of possibilities with streaming pipelines. We may emit some results early (using AfterProcessingTime), and then again at the watermark (when data is complete), and then for the next five messages that arrive late (after the watermark).

Accumulation modes: What to do with additional events
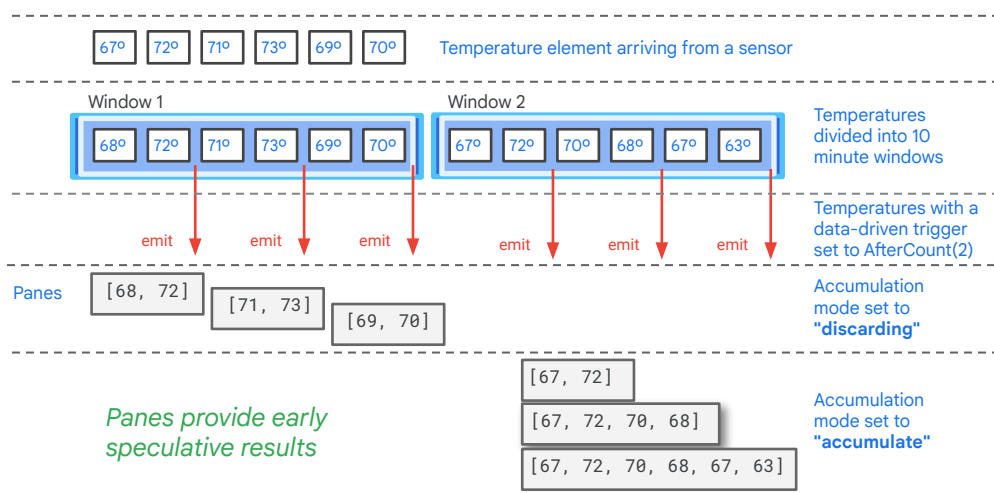
When you set a trigger, you need to choose either accumulate mode or discard mode. This example shows the different behaviors caused by the intersection of windowing, triggers, and accumulation mode.

Some example triggers

```
pcollection | WindowInto(
    SlidingWindows(60, 5),                        # Sliding window of 60 seconds, every 5 seconds
    trigger=AfterWatermark(                       # Relative to the watermark, trigger:
      early=AfterProcessingTime(delay=30),        # -- fires 30 seconds after pipeline commences
      late=AfterCount(1))                         # -- and for every late record (< allowedLateness)
    accumulation_mode=AccumulationMode.ACCUMULATING) # the pane should have all the records
    allowed_lateness=Duration(seconds=2*24*60*60))   # 2 days
```

```
pcollection | WindowInto(
    FixedWindows(60),                             # Fixed window of 60 seconds
    trigger=Repeatedly(                           # Set up a composite trigger that triggers ...
        AfterAny(                                 # whenever either of these happens:
            AfterCount(100),                      # -- 100 elements accumulate
            AfterProcessingTime(1 * 60))),        # -- every 60 seconds (ignore watermark)
    accumulation_mode=AccumulationMode.DISCARDING)  # the trigger should be with only new records
```

https://beam.apache.org/documentation/programming-guide/#composite-triggers

Google Cloud

Now we know different techniques to handle accumulation late arrival data. We also know that triggers are used to initiate the accumulation, and watermarks help in deciding the lag time and related corrective actions for computing accumulations.

The code in this example creates a sample trigger. As you can see in the code, we are creating a sliding window of 60 seconds, and it slides every five seconds. The function AfterWatermark method gives us details about when to trigger the accumulation. The code uses two options. First, early or speculative figuring, which is set to 30 seconds. Second, late, for each late arriving item.

The second code segment demonstrates the composite trigger. The composite trigger will get activated either after one-hundred elements are available for accumulation, or every 60 seconds irrespective of watermark. This code segment uses a fixed window of one minute's duration.

You can allow late data past the watermark

Allowing Late Data

```java
                                                        Java
PCollection<String> items = ...;

    PCollection<String> fixedWindowedItems = items.apply(

Window.<String>into(FixedWindows.of(Duration.standardMinutes(1)))
              .withAllowedLateness(Duration.standardDays(2)));
```

```python
                                                              Python
pc = [Initial PCollection]
    pc | beam.WindowInto(
            FixedWindows(60),
            trigger=trigger_fn,
            accumulation_mode=accumulation_mode,
            timestamp_combiner=timestamp_combiner,
            allowed_lateness=Duration(seconds=2*24*60*60)) # 2 days
```
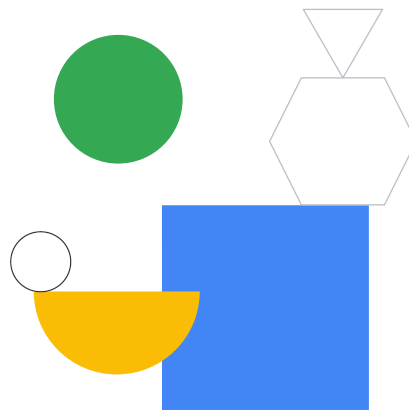
Google Cloud

This is how the window re-processes. This late processing works in Java and Python.

Lab Intro

Streaming Data Processing:
Streaming Data Pipelines

Google Cloud

In this lab, you use Dataflow to collect traffic events from simulated traffic sensor data made available through Pub/Sub, process them into an actionable average, and store the raw data in BigQuery for later analysis. You will learn how to start a Dataflow pipeline, monitor it, and, lastly, optimize it.