Dataflow Streaming Features

Agenda

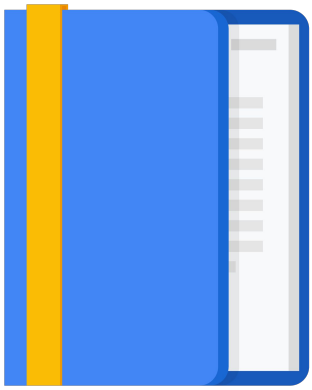
Processing Streaming Data

Cloud Pub/Sub

Cloud Dataflow Streaming Features

BigQuery and Bigtable Streaming Features

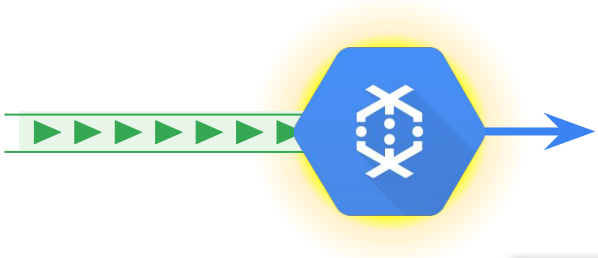
Advanced BigQuery Functionality



Google Cloud

Now, let's look at Cloud Dataflow Streaming features.

Streaming features of Cloud Dataflow



Cloud Dataflow

Qualities that Cloud Dataflow contributes to Data Engineering solutions:

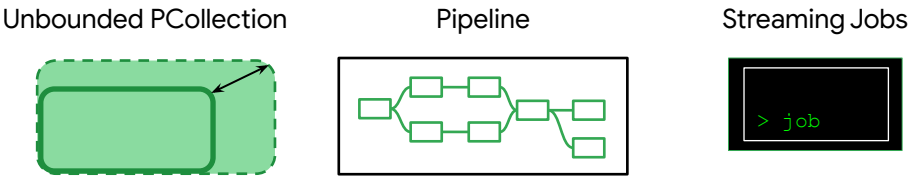
- Scalability
- Low latency

Google Cloud

Cloud 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.

https://pixabay.com/illustrations/chess-black-and-white-pieces-3413429/

Continuing from the Data Processing course



Google Cloud

We have discussed that we can have bounded and unbounded collections, so now we are examining unbounded pipe that result from streaming jobs. And, all of the things we have done thus far like, branching, merging, we can do all of that but with streaming pipelines as well using Dataflow.

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

There are challenges with processing streaming data

Scalability
Streaming data generally only grows larger and more frequent

Google Cloud

What are some of the challenges with processing streaming data?

Once challenge is scalability, being able to handle the volume of data as it gets larger and/or more frequent.

https://pixabay.com/illustrations/arrows-growth-hacking-marketing-1229848/
https://pixabay.com/photos/crack-cracky-cracked-old-paint-1827108/
https://pixabay.com/illustrations/time-time-management-stopwatch-3222267/
https://pixabay.com/photos/doors-choices-choose-open-decision-1587329/

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

Google Cloud

What are some of the challenges with processing streaming data?

The second challenge is fault tolerance. The larger you get, the more sensitive you are to going down unexpectedly.

https://pixabay.com/illustrations/arrows-growth-hacking-marketing-1229848/
https://pixabay.com/photos/crack-cracky-cracked-old-paint-1827108/
https://pixabay.com/illustrations/time-time-management-stopwatch-3222267/
https://pixabay.com/photos/doors-choices-choose-open-decision-1587329/

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?

Google Cloud

What are some of the challenges with processing streaming data?

The third challenge is model being used, streaming or repeated batch.

https://pixabay.com/illustrations/arrows-growth-hacking-marketing-1229848/
https://pixabay.com/photos/crack-cracky-cracked-old-paint-1827108/
https://pixabay.com/illustrations/time-time-management-stopwatch-3222267/
https://pixabay.com/photos/doors-choices-choose-open-decision-1587329/

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?

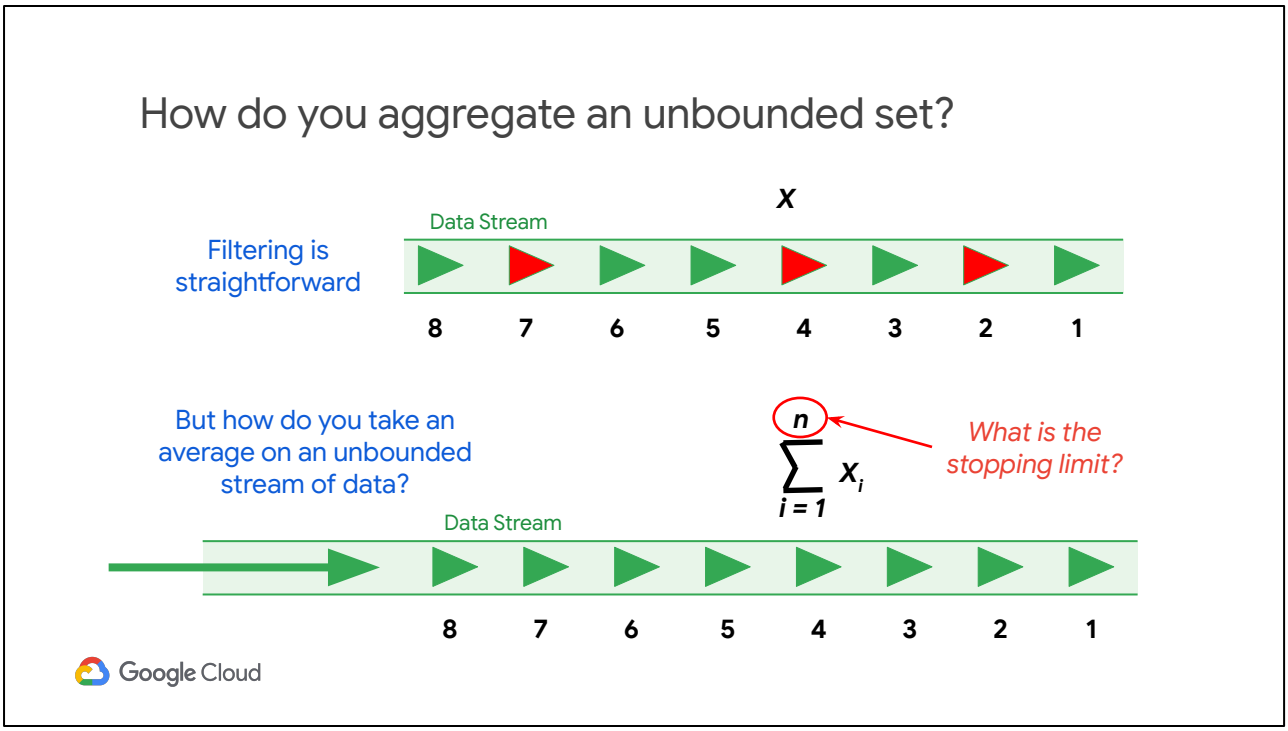
Timing
What if data arrives late?

Google Cloud

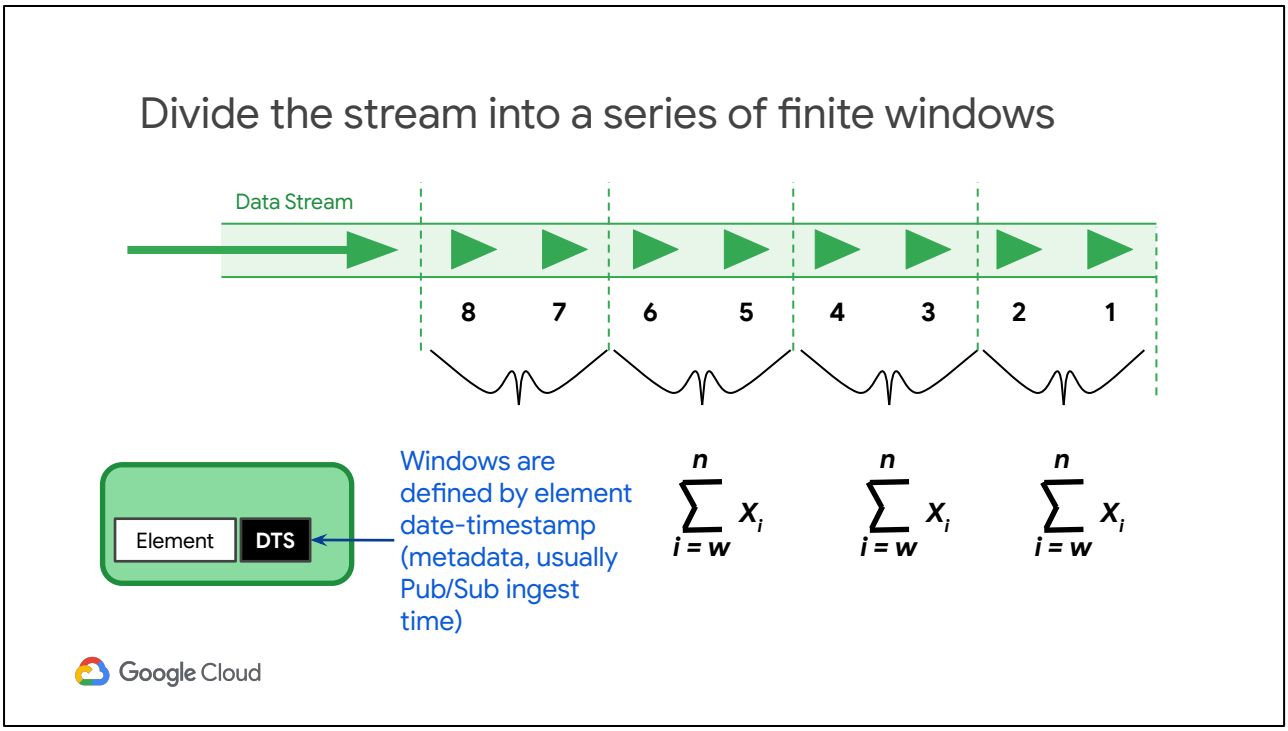
What are some of the challenges with processing streaming data?

Another challenge is timing, 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?

https://pixabay.com/illustrations/arrows-growth-hacking-marketing-1229848/
https://pixabay.com/photos/crack-cracky-cracked-old-paint-1827108/
https://pixabay.com/illustrations/time-time-management-stopwatch-3222267/
https://pixabay.com/photos/doors-choices-choose-open-decision-1587329/

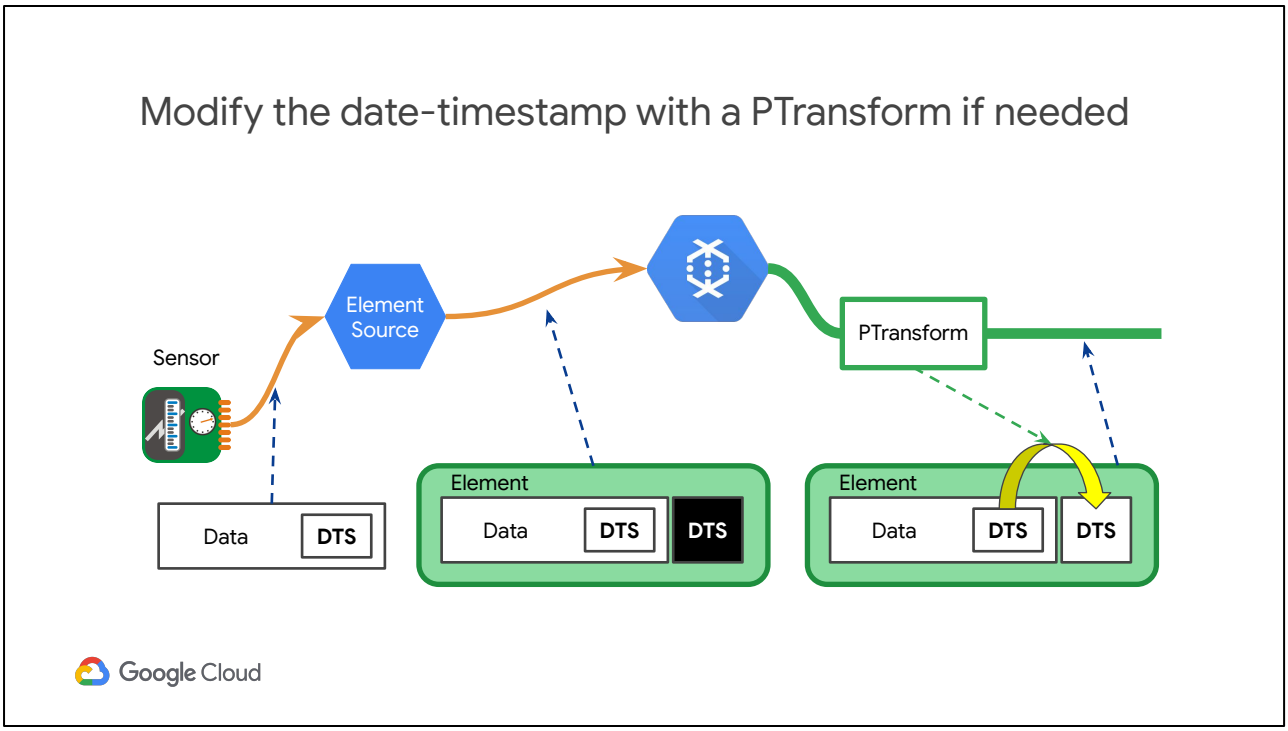


Additionally, there is a challenge around any kind of aggregation you might be trying to do. For example, if you are trying to take the average of data, but it is in a streaming scenario, just plug that into the formula for average, and the sum from 1 to n, but n is an ever-growing number.



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 just a Pub/Sub message timestamp and then you will be able to use this timestamp to put it into the different time windows and aggregate all of those windows.



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 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.

https://pixabay.com/vectors/sensor-computer-hardware-strain-152662/

```
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)
```

Code to modify date-timestamp

Python

```
yield beam.window.TimestampedValue(element, unix_timestamp)
```

Java

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

Google Cloud

Here is the code used to modify the date-timestamp.

Cloud Dataflow Windowing

Google Cloud

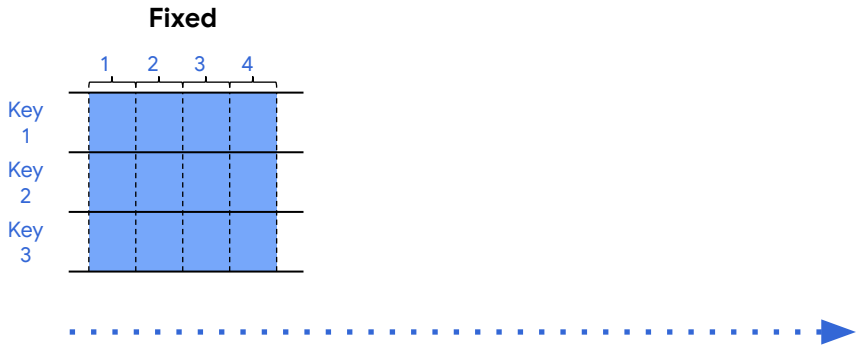
Next, let's look at Cloud Dataflow Windowing capabilities. This is really Dataflow's strength when it comes to streaming.

Three kinds of windows fit most circumstances

- Fixed
- Sliding
- Sessions

Dataflow will give us three different types of windows, fixed, sliding, and sessions.

Three kinds of windows fit most circumstances

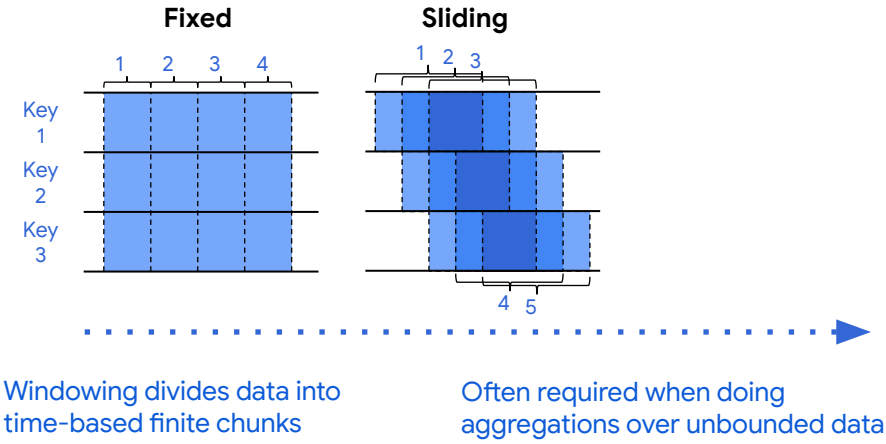


Windowing divides data into time-based finite chunks

Google Cloud

Fixed are those that are divided into time slices, for example, hourly, daily, monthly. Fixed time windows consist of consistent non-overlapping intervals.

Three kinds of windows fit most circumstances

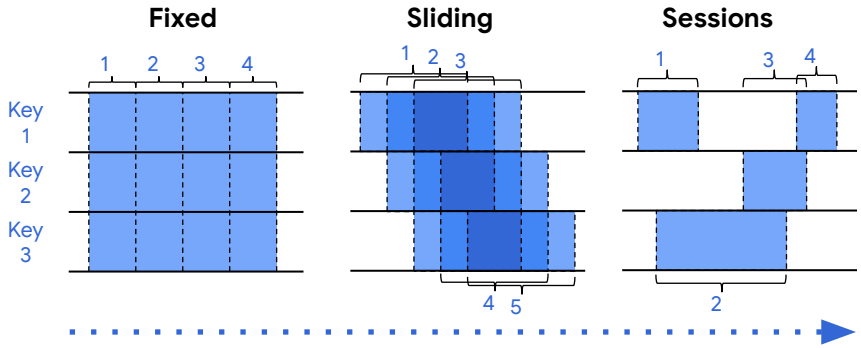


Google Cloud

Sliding windows are those or example; the last 24 hours worth of data, every hour), and session-based windows that capture bursts of user activity.

Sliding time windows can overlap, for example, in a running average.
Session windows are defined by a minimum gap duration and the timing is triggered by another element.

Three kinds of windows fit most circumstances



Windowing divides data into time-based finite chunks

Often required when doing aggregations over unbounded data

Google Cloud

Sliding windows are those or example; the last 24 hours worth of data, every hour), and session-based windows that capture bursts of user activity.

You could implement window-based processing on bounded data based on the date-timestamp of elements. However, it is necessary when performing aggregation processing on streaming data.

Sliding time windows can overlap, for example, in a running average.
Session windows are defined by a minimum gap duration and the timing is triggered by another element.

Setting time windows

Remember:

you can apply windows to batch data, although you may need to generate the metadata date-timestamp on which windows operate.

Fixed-time windows

Python

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

Sliding time windows

Python

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

Session windows

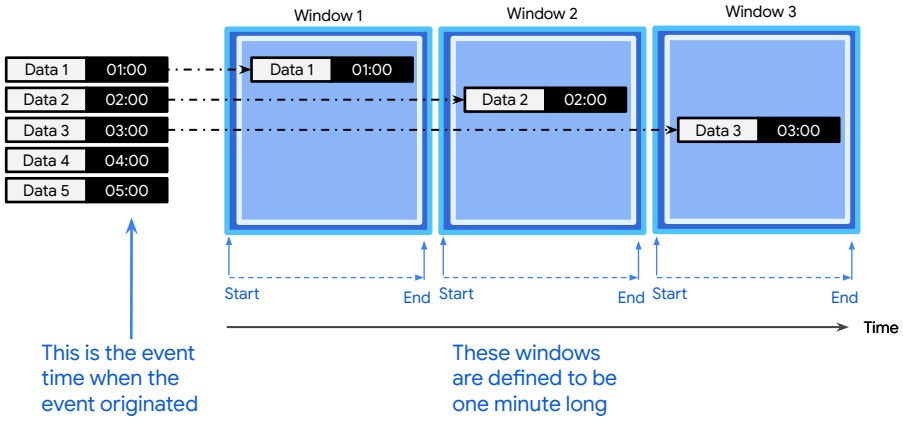
Python

```
from apache_beam import window
session_windowed_items = (
    items | 'window' >> beam.WindowInto(window.Sessions(10 * 60)))
```

Google Cloud

<INSTRUCTOR>
Sample code not tested.
</INSTRUCTOR>

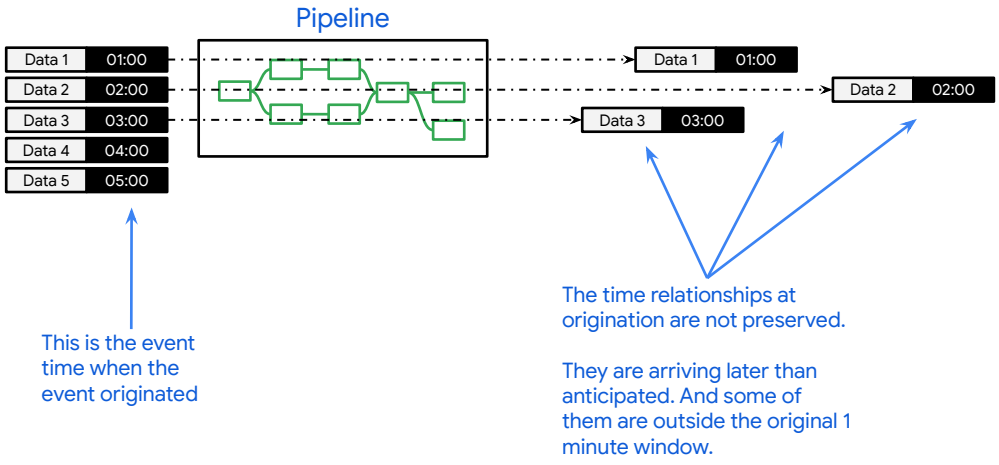
Windowing by time if there is no latency



Google Cloud

How does windowing work? All things being equal, this is how windowing ought to work. If there was no latency, if we had in 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 becomes 8:05:00, a five minute window terminates, and flushes all of the data. This is only IF there is not latency.

Pipeline processing can introduce latency

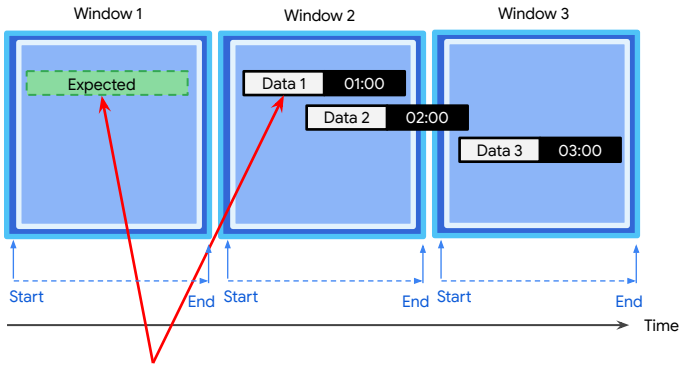


Google Cloud

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 Cloud Dataflow deal with this situation?

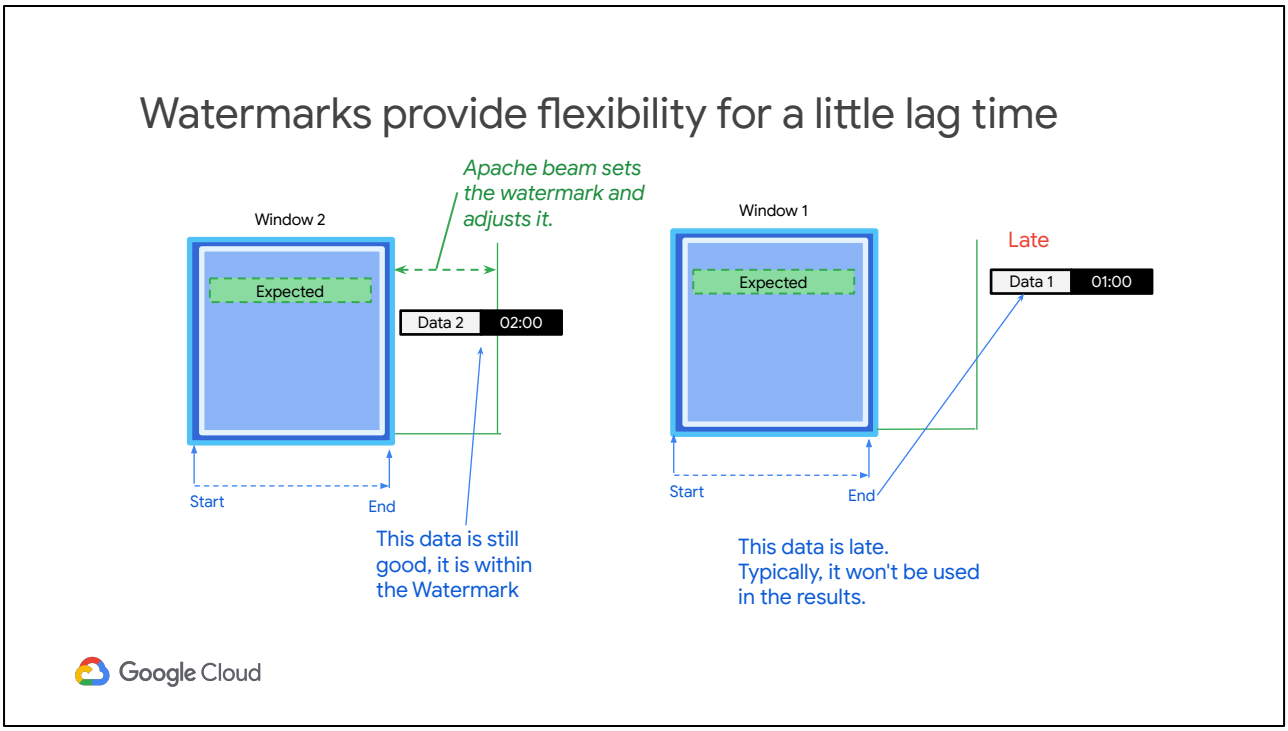
The data could be a little past the window or a lot. Data 2 is a little outside of Window 2. Data 1 is completely outside of Window 1.



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

Google Cloud

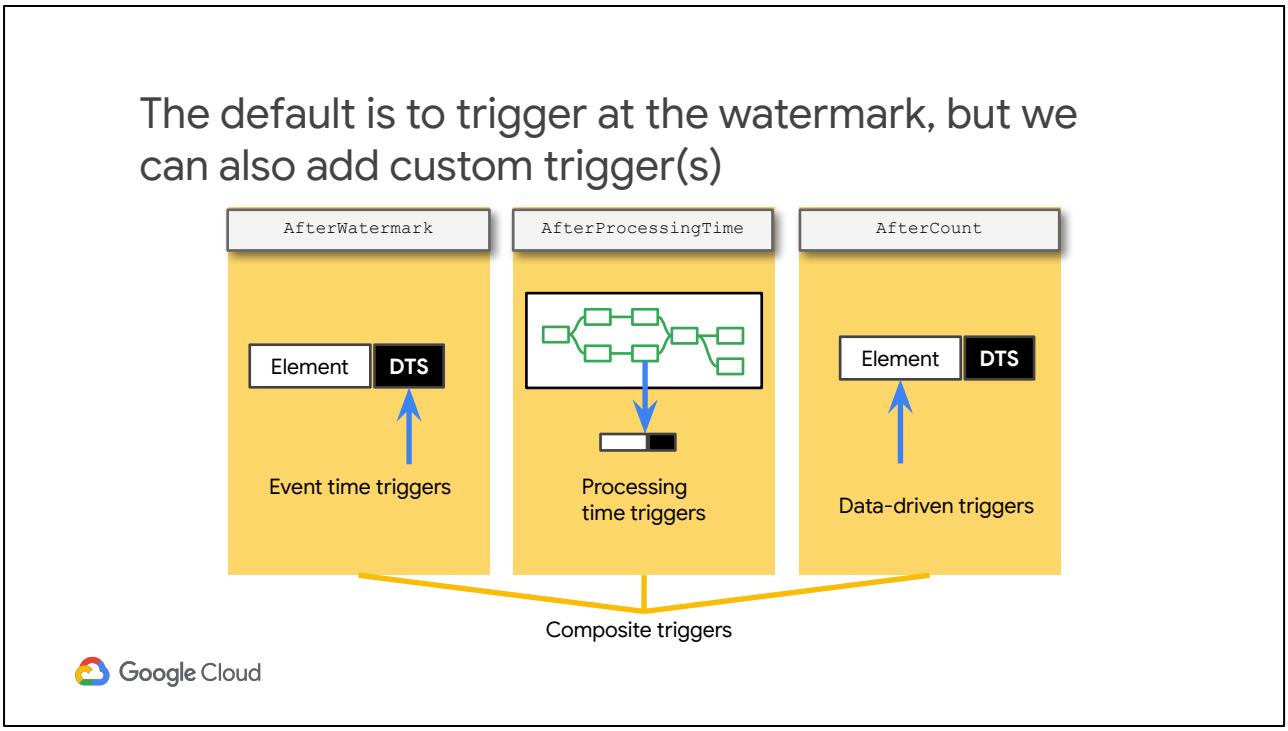
This is what we call the watermark and Dataflow keeps track of it automatically. Basically, it is going to keep track of the lag time, and it is able to do this, for example, if you are using the Pub/Sub connector, because it knows the time of the oldest, unprocessed message in Pub/Sub. And then it knows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



So, what Dataflow is going to do is continuously compute the watermark, which is how far behind are we. 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 determines 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."
https://beam.apache.org/documentation/programming-guide/#windowing-basics



Event time triggers operate on the date-timestamp associated with each element. The default trigger is of this type. The **AfterWatermark** trigger is the only event-type trigger currently supported. Apache Beam determines when all the elements with a date-timestamp that falls within the window have been processed. This is the Watermark. The passing of the Watermark causes the aggregation step to be performed.And after the Watermark has passed, the default event time trigger is activated. Its behavior is to emit the results of the aggregation one time, and to discard any data that arrives late. In Java pipelines you can override this behavior and do something with late data. In a Python pipeline, currently, late data is discarded unconditionally.

Processing time triggers operate on the time at which an element is processed at some point in the pipeline as determined by a system clock. You could set an **AfterProcessingTime** trigger on unbounded data contained in a global window. For example, emitting data every 30 seconds. The data never ends. The Window never closes. But interim results are emitted every 30 seconds by the trigger.

And a **data-driven trigger** is associated with the condition of data contained in the element itself. Currently, this simply counts each element that has been processed in the window. You could set **AfterCount** to 15, and every 15 elements processed would cause and emit.

Composite triggers combine effects. For example, consider if you had a data-driven AfterCount trigger set to 15. Every 15 elements it would emit. However, if there were

14 elements in the PCollection, and no more data arrived, the 14 would sit in the window forever. In this case you could add in an Event time trigger to ensure that the last 14 were serviced by an emit.

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
```

```
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
```

Google Cloud

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

The opposite of Repeatedly is orFinally which tells the trigger to stop

We'll talk about accumulation mode shortly

You can allow late data past the watermark

Allowing Late Data

Java

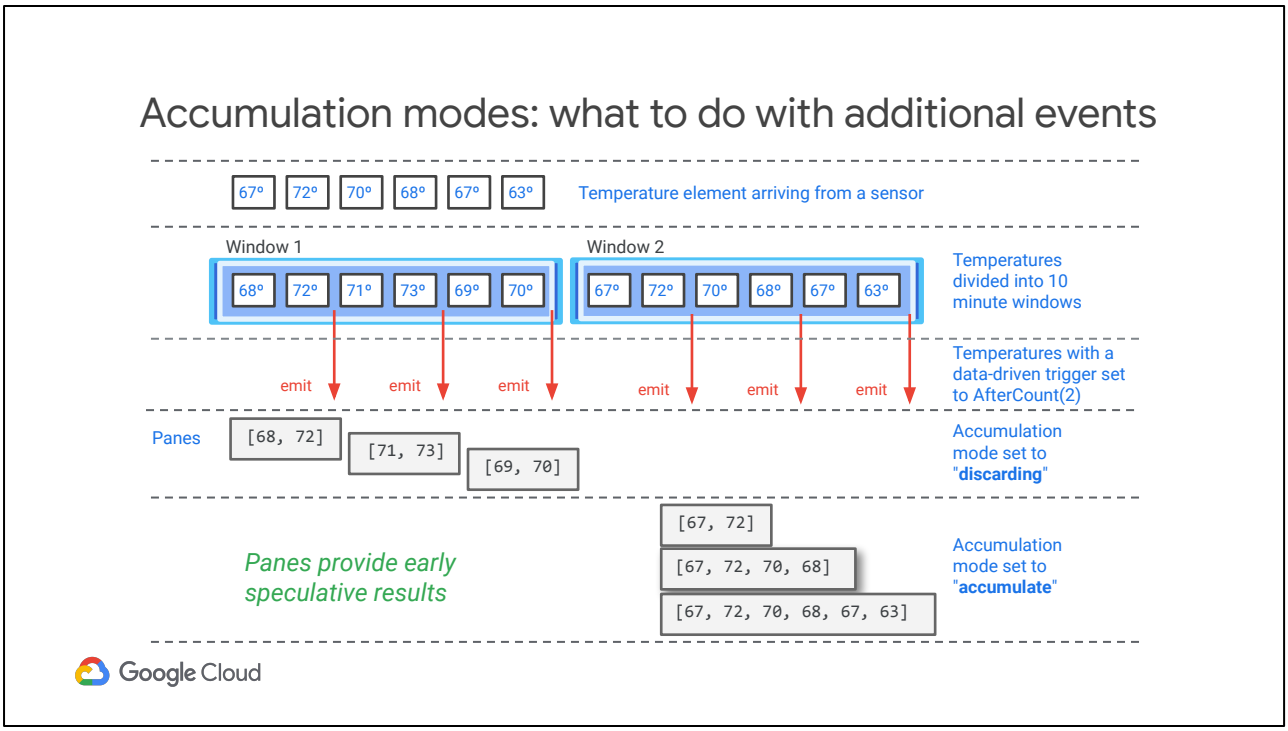
```
PCollection<String> items = ...;

    PCollection<String> fixedWindowedItems = items.apply(

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

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 both Java and Python. Implementation of Apache Beam watermark support is part of the Open Source Software and not directly implemented by Google.



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.

Streaming Data Pipelines

Objectives

- Launch Dataflow and run a Dataflow job
- Understand how data elements flow through the transformations of a Dataflow pipeline
- Connect Dataflow to Pub/Sub and BigQuery
- Observe and understand how Dataflow autoscaling adjusts compute resources to process input data optimally
- Learn where to find logging information created by Dataflow
- Explore metrics and create alerts and dashboards with Cloud Monitoring