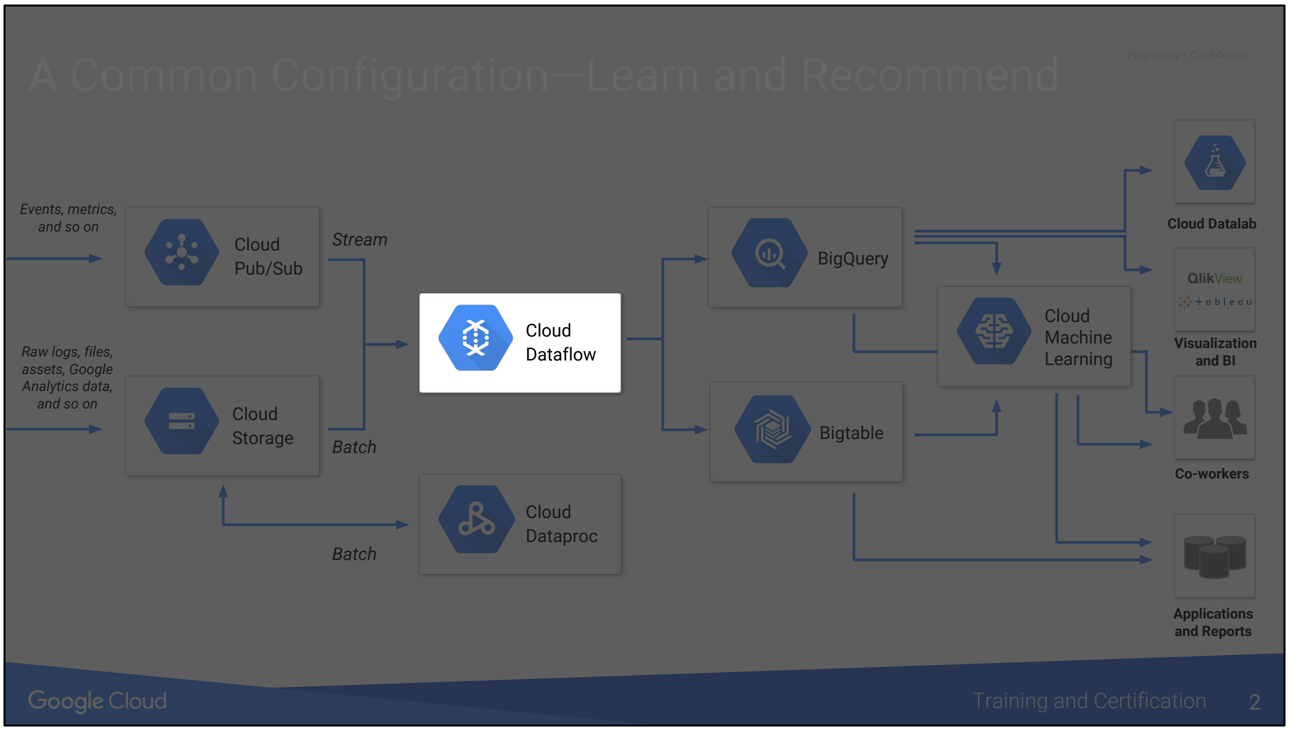
Google Cloud

Implementing streaming pipelines

Data Engineering on Google Cloud Platform

1.5 hours + 30 min lab = 2 hours



A Common Configuration—Learn and Recommend

Proprietary + Confidential

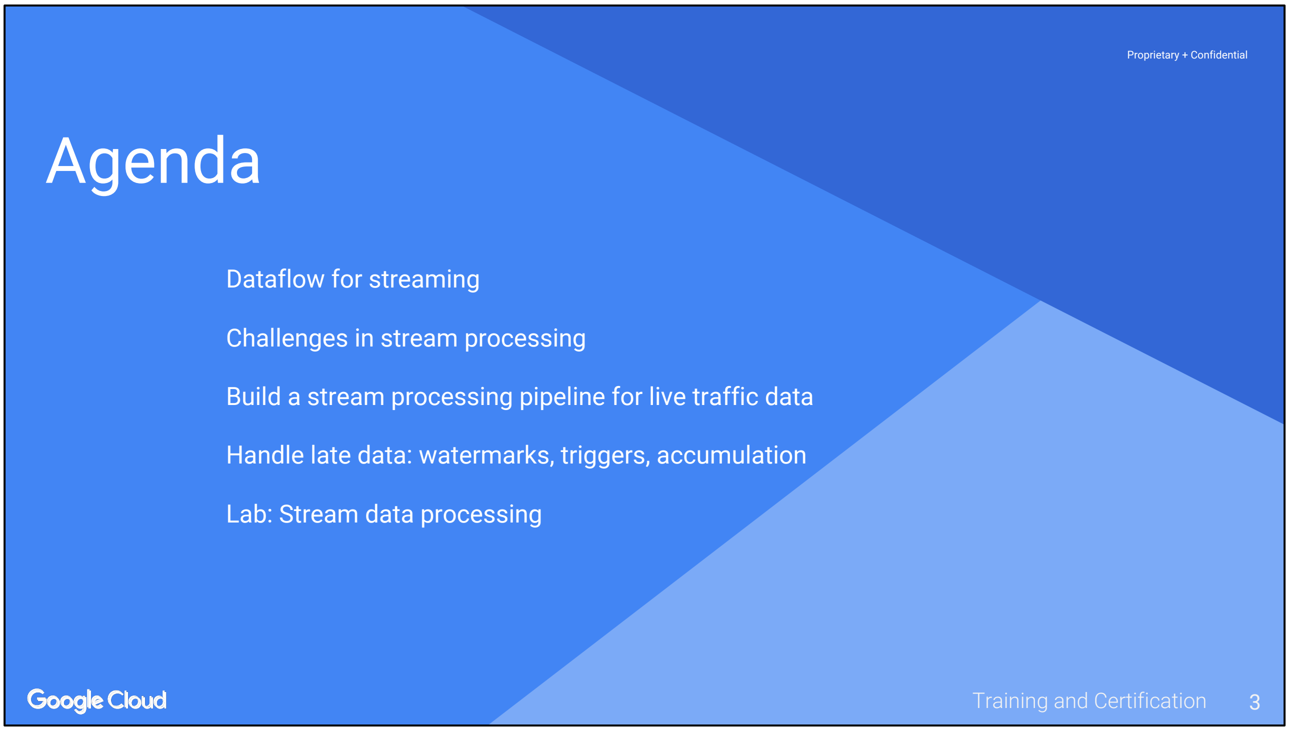
Google Cloud Training and Certification 2

Notes:

How our engagement model has changed:

And now, in the new model, we publish the code in tandem with the paper. Case in point, for Dataflow we put the SDK out first (12/2014) and then published the
paper (8/2015).

Graphically, the most obvious way to do this would be to use the timeline slide as starting point and do a build-up in 4 steps:
- Show our papers (GFS, MapReduce, Dremel, BigTable, FlumeJava, Millwheel)
- Show how they were followed by open source implementations (see list above)
- Show how for Dataflow we published the SDK ourselves, and it was followed by the paper



Agenda

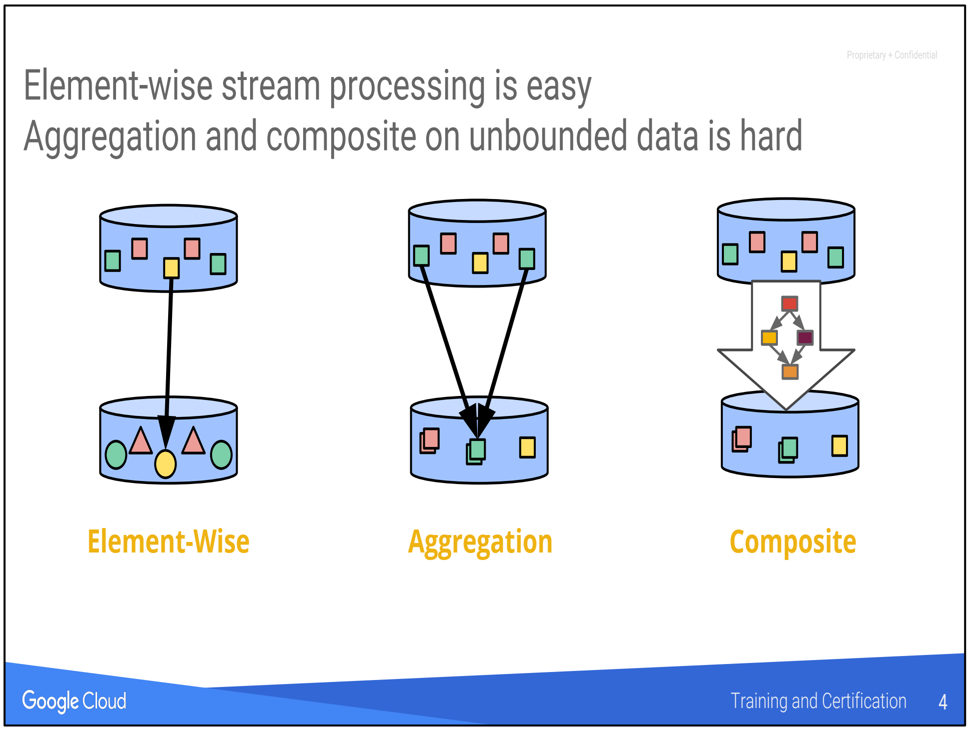
Dataflow for streaming

Challenges in stream processing

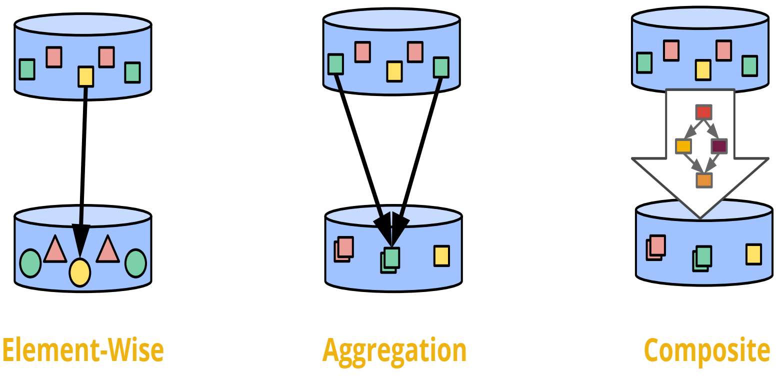
Build a stream processing pipeline for live traffic data

Handle late data: watermarks, triggers, accumulation

Lab: Stream data processing



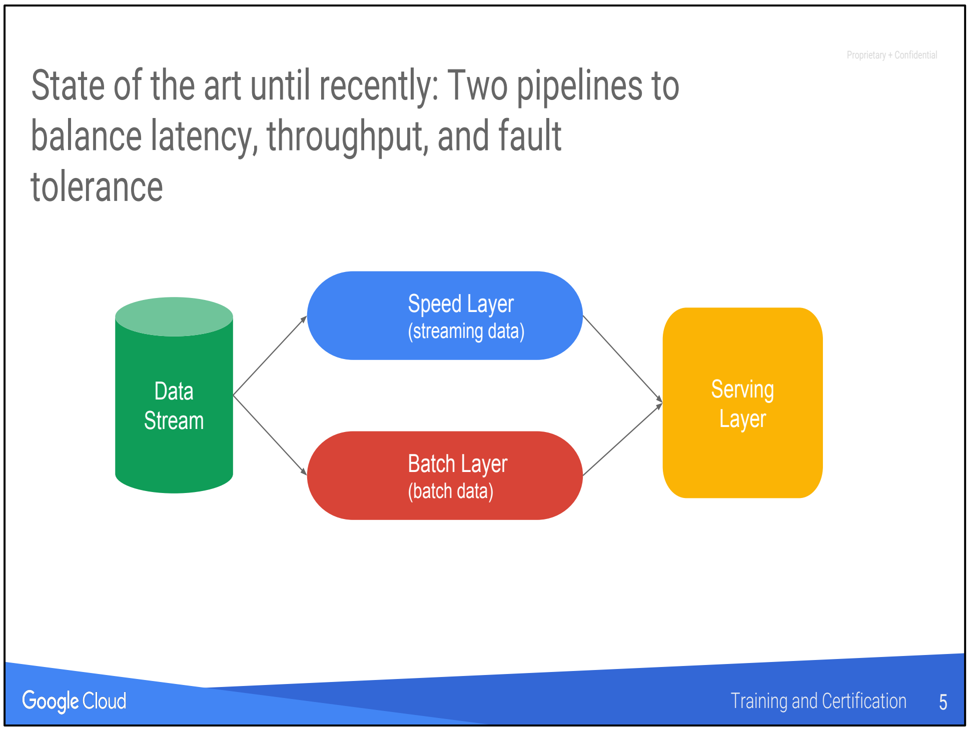
Element-wise stream processing is easy
Aggregation and composite on unbounded data is hard



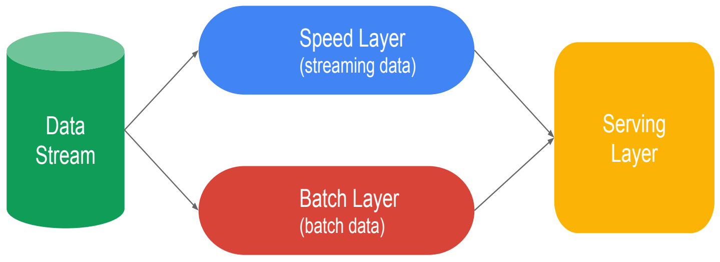
Notes:

For element wise operations, each element is independently processed so you can do it as soon as it arrives, whereas aggregations require all the data elements to be available in order to perform the operation and give an accurate answer, for example top scorer in the last 30 mins game activity.

You want aggregates to be correct...even when data comes in late....or out of order



State of the art until recently: Two pipelines to balance latency, throughput, and fault tolerance

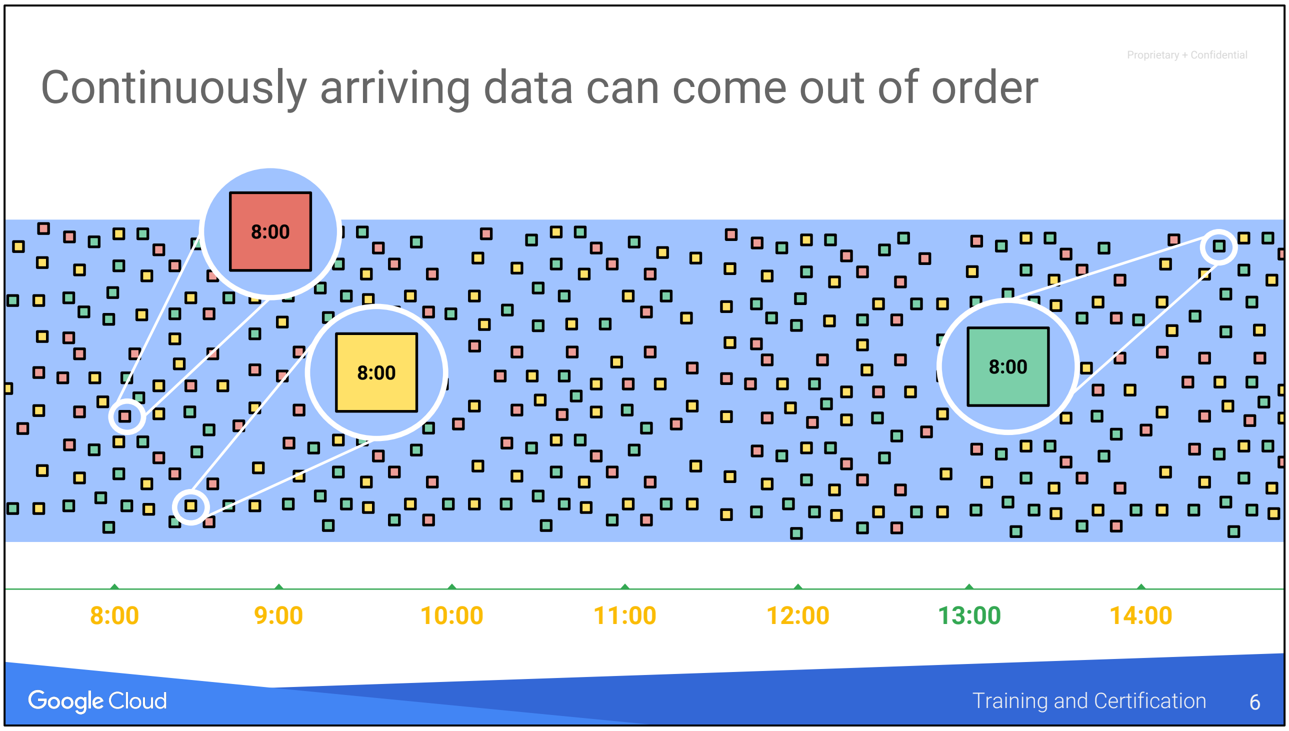


Notes:

Yes...2 pipelines!!! Very complex….streaming and batch pieces typically require different codebases

Serving layer does the combine….and goes back to speed layer…..and figure out new data..very convoluted!

Typically batch and streaming need different architecture, infrastructure, solutions, and programming models to handle batch and real-time workloads, respectively. Because compromises need to be made to weigh the tradeoffs — accuracy vs. latency, but not both.

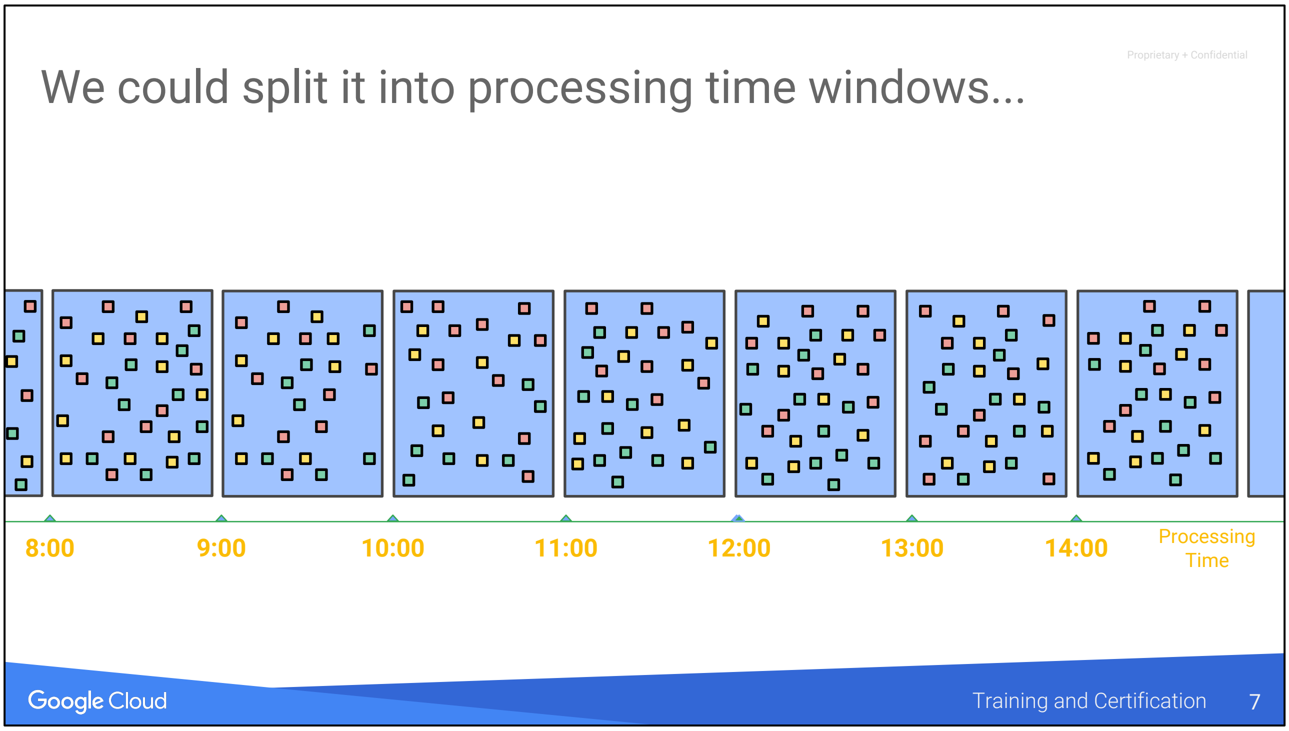


Continuously arriving data can come out of order

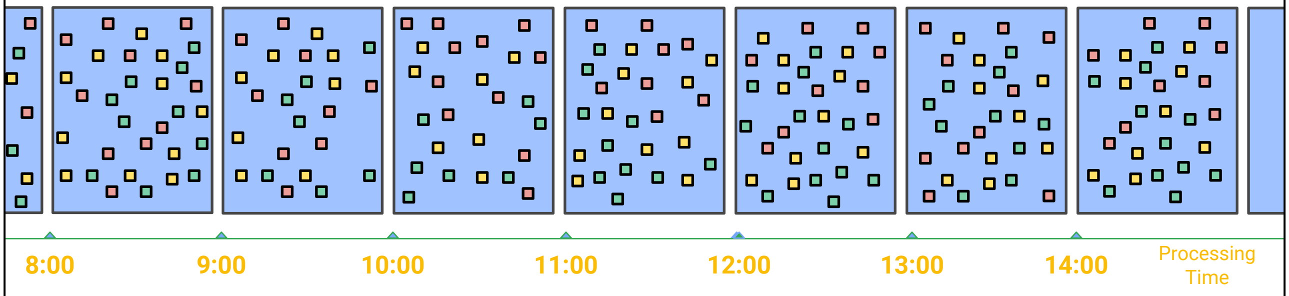
Notes:

U decide to process at 8:45...you get the first 2...but the third one comes at 1 pm. So when you publish report at 9…..it won't be complete. Hence 2 pipeline design.

- How to deal with unbounded data?
- This could happen for a multitude of reasons - for example an event occurred on a mobile device, and then it was put in airplane mode. Or there was a delay in the network. Or a server crashed somewhere and took a long time to come back up.
- Regardless of the reason, if you are trying to analyze records as they are happening, you need to deal with these sorts of delays. What you need to do changes depending on what you are trying to accomplish
- If you are doing for element-wise processing, this may not be a big deal as you may only care about a single event at a time.



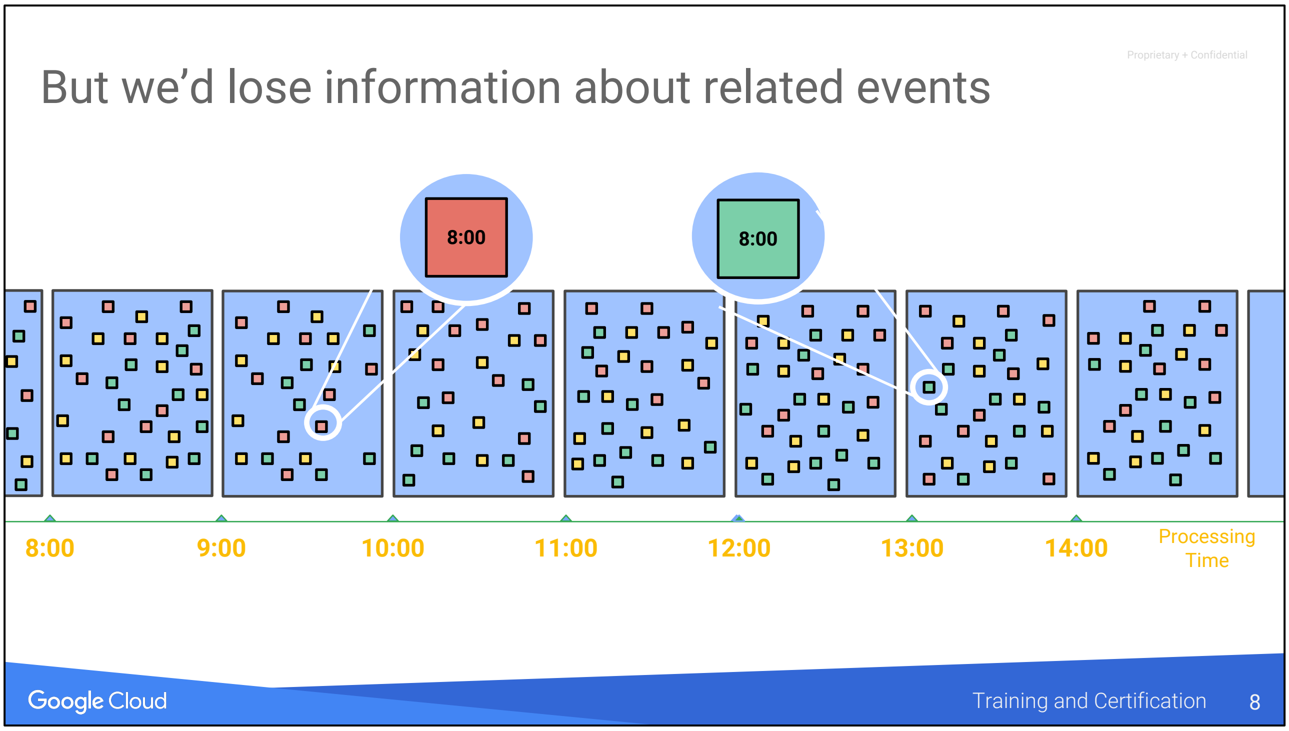
We could split it into processing time windows...



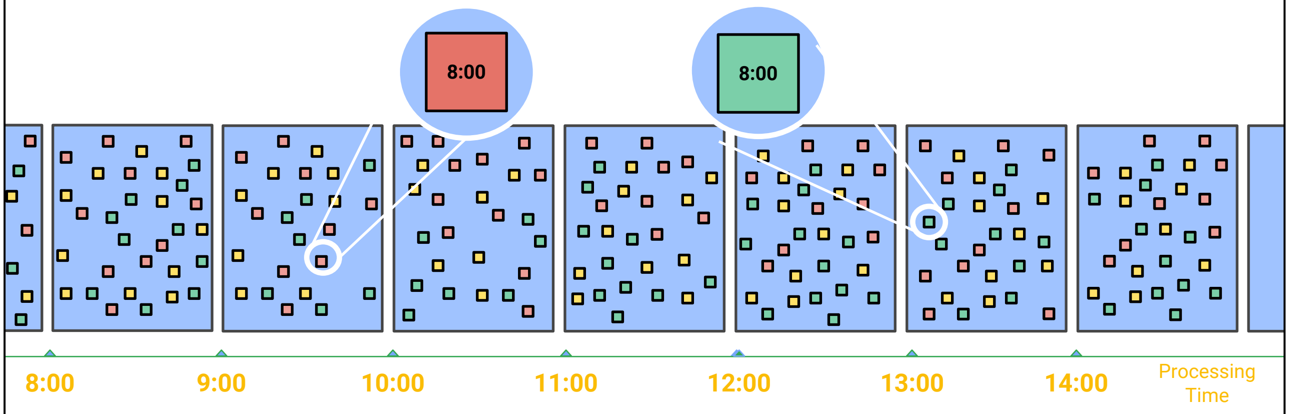
Notes:

When you bundle with windows….you group some...but those that come much later...you still lose them

- Things get trickier if what you want to do involves grouping or aggregations.
- One obvious strategy is to use processing time windows.
- Imagine you slice the stream into windows by processing time, so for example every time the clock ticks over one hour, you draw a boundary in your data stream.
- This is easy to understand, but often does not do what you want.
- If elements are delayed or out of order, you may not be analyzing the event in the context in which it occurred.

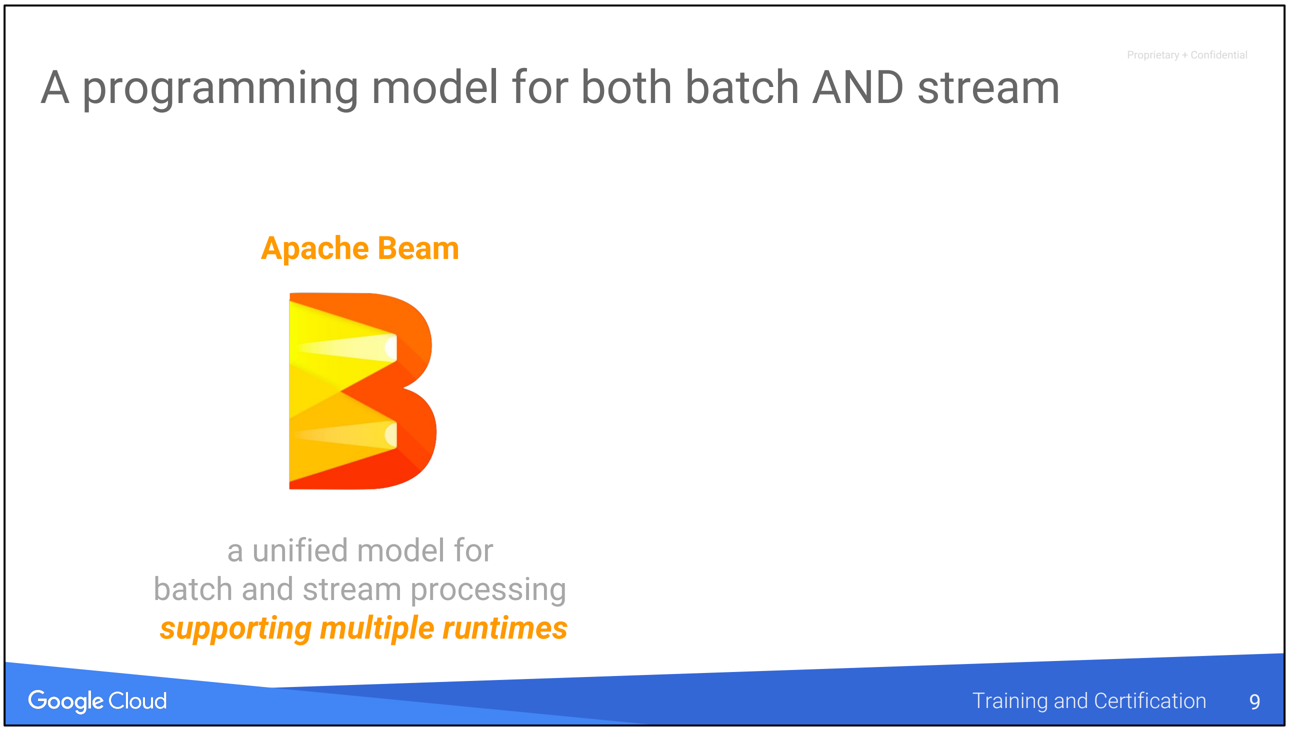


But we'd lose information about related events



Notes:

- For example our two messages that both happened at 8 are now spread across two windows.
- If your aggregation involves grouping together all messages that occurred around 8, then this processing time windows strategy clearly is not what you would like
- For example if you are counting up taxis that showed up around the same time, processing time windows won't work.



A programming model for both batch AND stream

Apache Beam

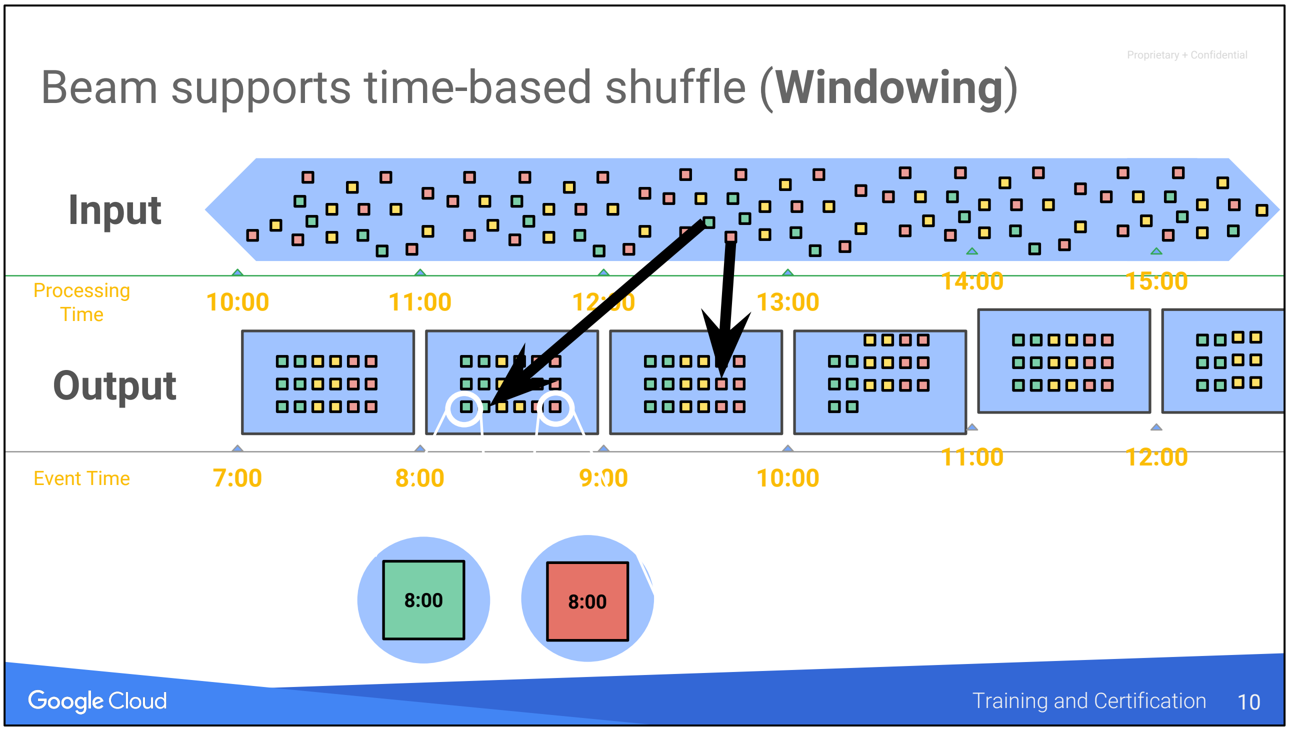
a unified model for
batch and stream processing
supporting multiple runtimes

Notes:

Providing a unified programming model for both batch and streaming data is an important differentiator. So what does this mean?

Typically batch and streaming need different architecture, infrastructure, solutions, and programming models to handle batch and real-time workloads, respectively. Because compromises need to be made to weigh the tradeoffs — accuracy vs. latency, but not both. (See lambda.)

The open-source incubation project is Apache Beam, a combined word of "batch" and "stream", to signify that fact.

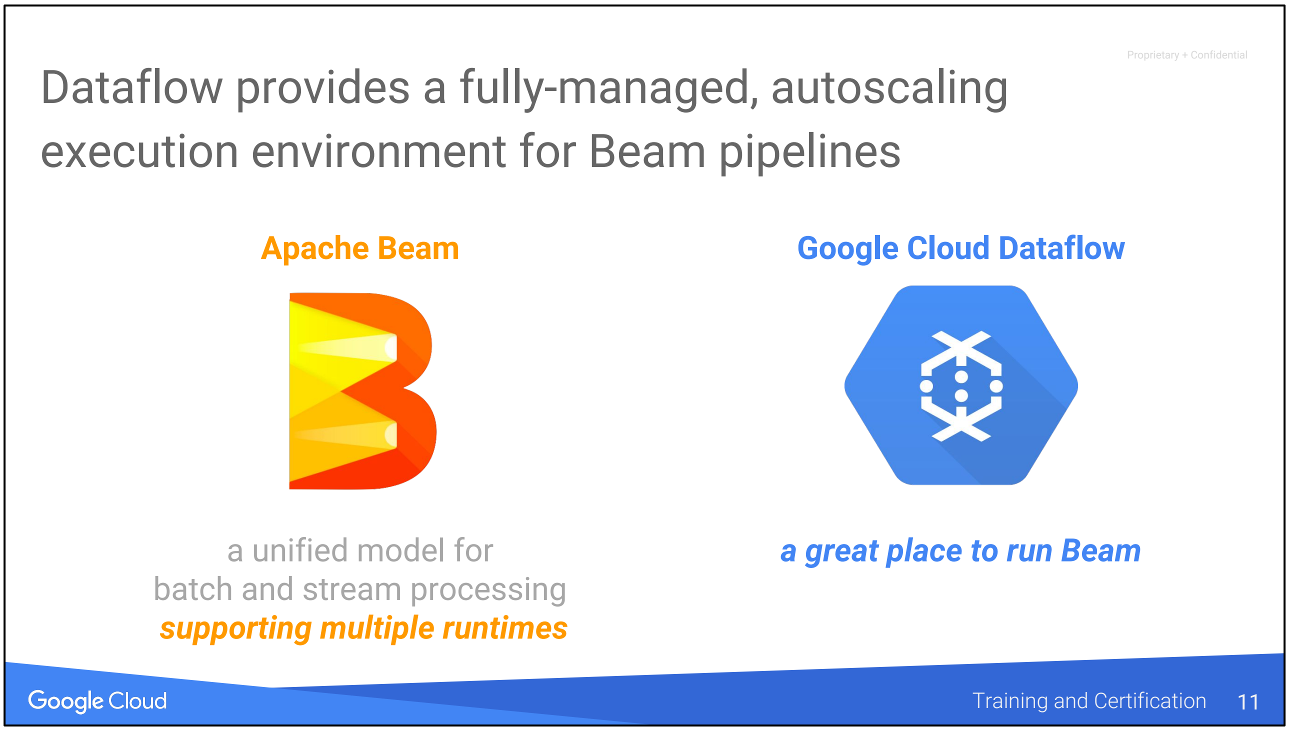


Beam supports time-based shuffle (**Windowing**)

Notes:

Time based shuffle takes input data and treat time as key

- What we'd really like to have is to have event-time windowing.
- As input is arriving, we are performing a time-based shuffle, to place the records into windows based on their event times.
- The way to do this in dataflow, is via the Windowing API.
- Windowing divides events into finite time based chunks and lets you reason about them.



Dataflow provides a fully-managed, autoscaling execution environment for Beam pipelines

Apache Beam

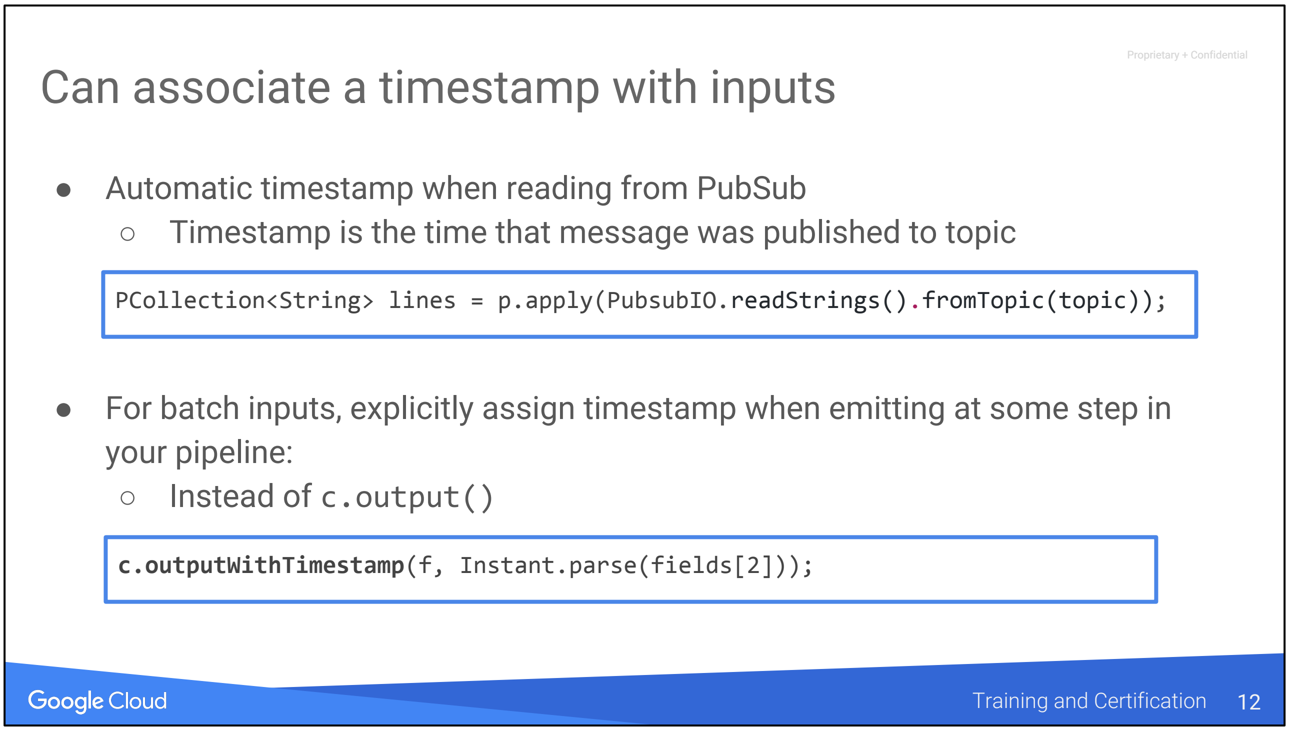
a unified model for
batch and stream processing
supporting multiple runtimes

Google Cloud Dataflow

a great place to run Beam

Notes:

Dataflow unifies the programming models and consolidates the underlying infrastructure. The fact that infrastructure provisioning is fully controlled by the developers who design the pipelines is both liberating and revolutionary: no longer does one need to requisition resources (typically controlled by a different team) before compute is possible.



Can associate a timestamp with inputs

- Automatic timestamp when reading from PubSub
  - Timestamp is the time that message was published to topic

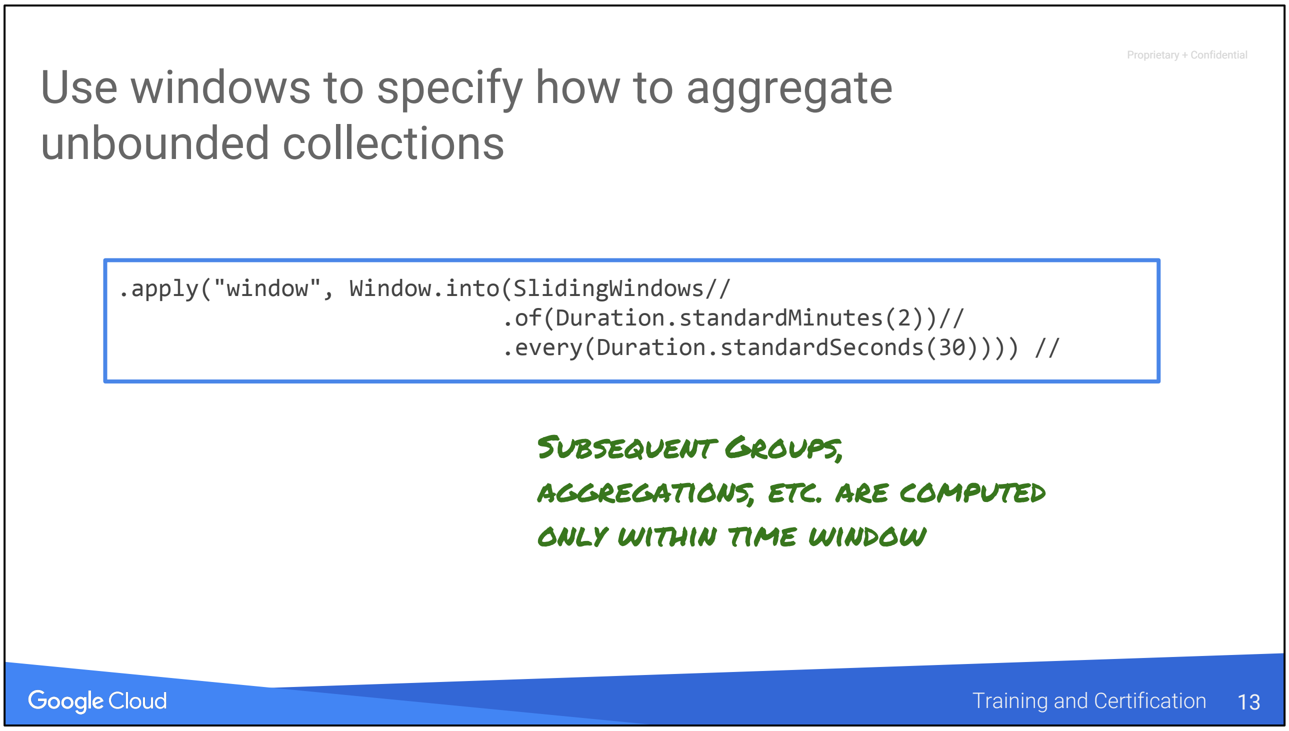
```
PCollection<String> lines = p.apply(PubsubIO.readStrings().fromTopic(topic));
```

- For batch inputs, explicitly assign timestamp when emitting at some step in your pipeline:
  - Instead of `c.output()`

```
c.outputWithTimestamp(f, Instant.parse(fields[2]));
```

Notes:

This timestamp will be the time at which the element was published to PubSub. If you want to use a custom timestamp, it must be published as a PubSub attribute, and you tell Dataflow about it using the `timestampLabel` setter.



Use windows to specify how to aggregate unbounded collections

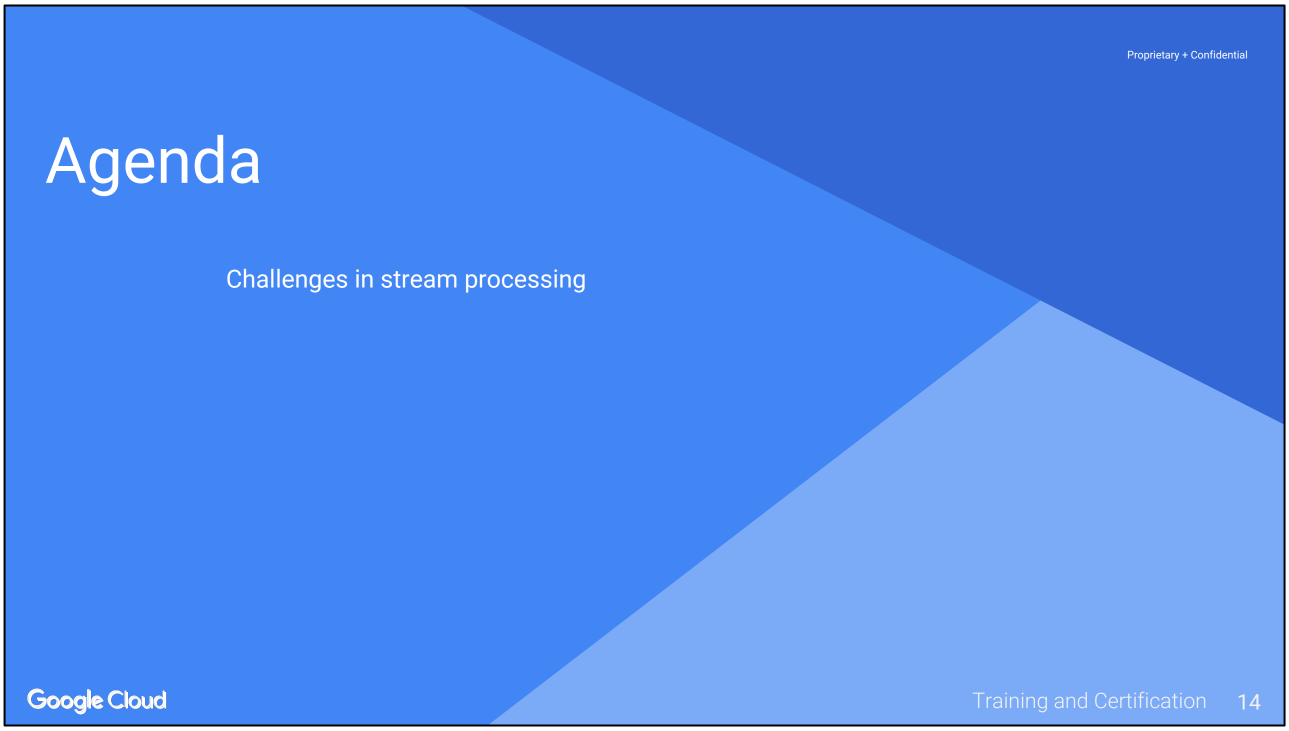
```
.apply("window", Window.into(SlidingWindows//
                             .of(Duration.standardMinutes(2))//
                             .every(Duration.standardSeconds(30)))) //
```

SUBSEQUENT GROUPS, AGGREGATIONS, ETC. ARE COMPUTED ONLY WITHIN TIME WINDOW

Notes:

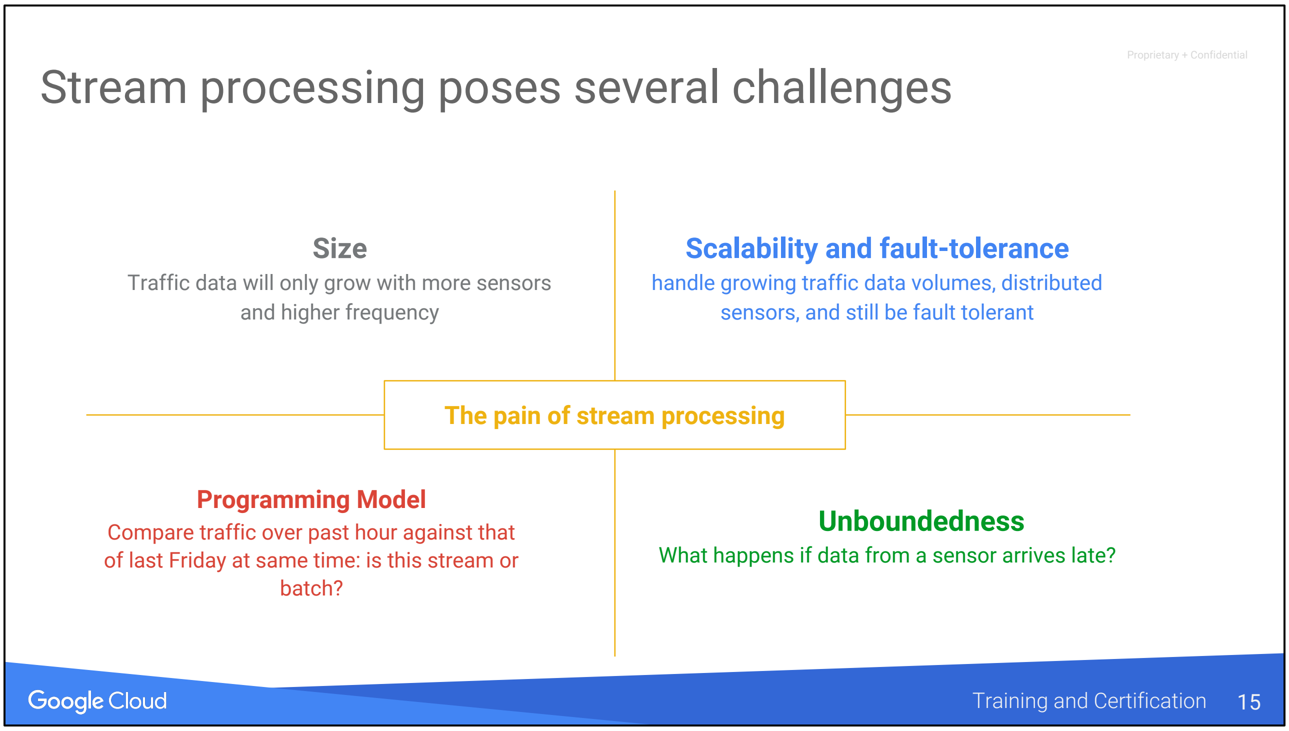
The collection that you read from Pub/Sub is unbounded; need to bound it

Every 60 minutes on windows of 60 minutes; other options exist, of course



Agenda

Challenges in stream processing



Proprietary + Confidential

Stream processing poses several challenges

The pain of stream processing

Size
Traffic data will only grow with more sensors and higher frequency

Scalability and fault-tolerance
handle growing traffic data volumes, distributed sensors, and still be fault tolerant

Programming Model
Compare traffic over past hour against that of last Friday at same time: is this stream or batch?

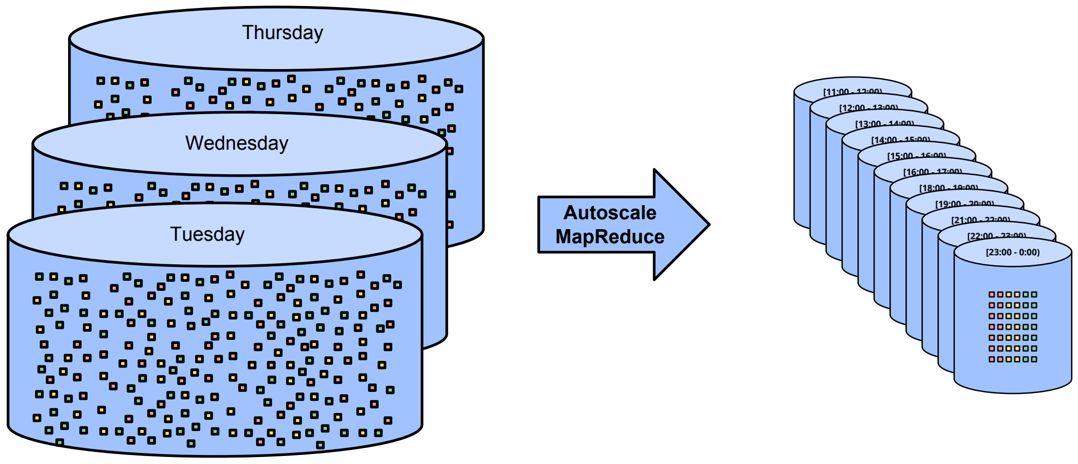
Unboundedness
What happens if data from a sensor arrives late?

Google Cloud Training and Certification 15

Notes:

The next few slides basically look at these four challenges.

Need to process variable amounts of data that will grow over time

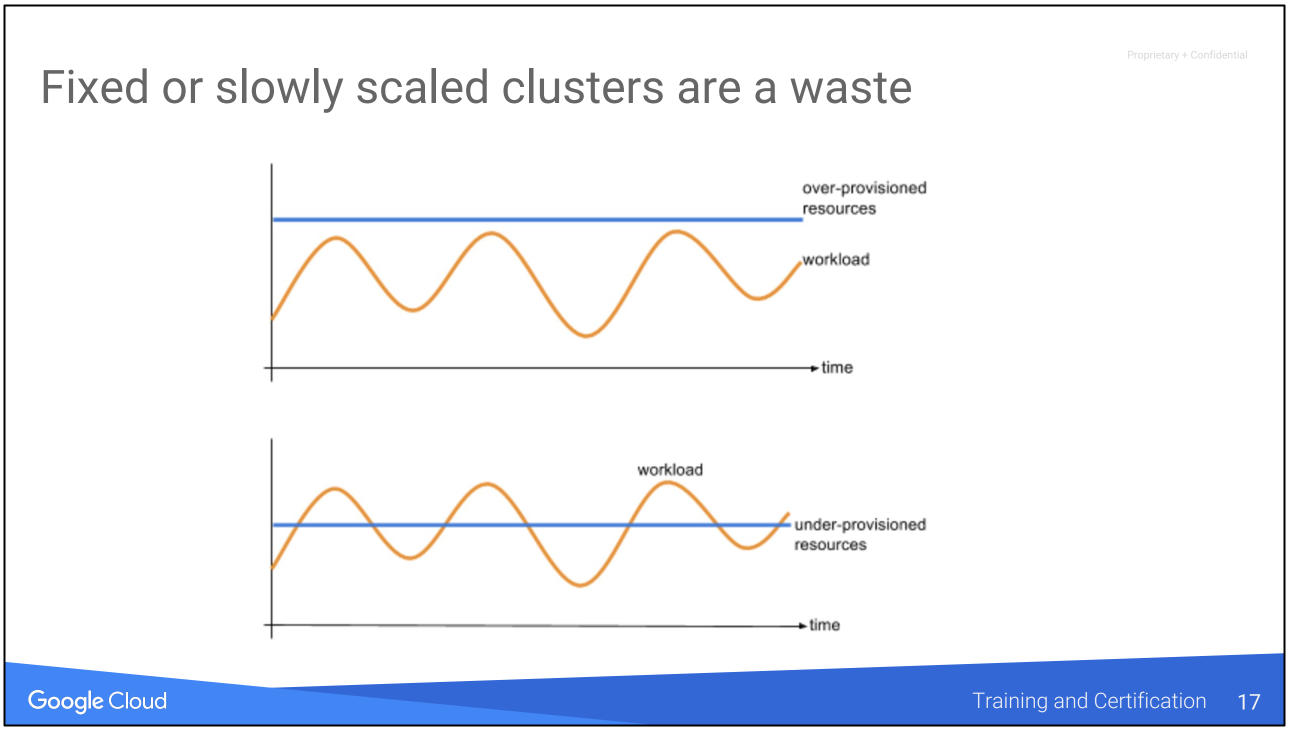


Notes:

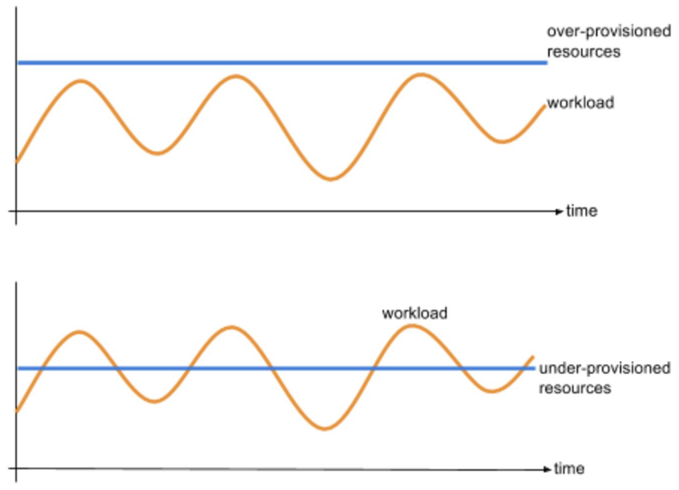
Variable amount of traffic….do in the time window shuffle….according to the amount of traffic.

Autoscale to handle variable amounts. Sometimes use 30 machines, other times use 50.

MapReduce to handle growing data volumes. Too much data to process using only one machine.



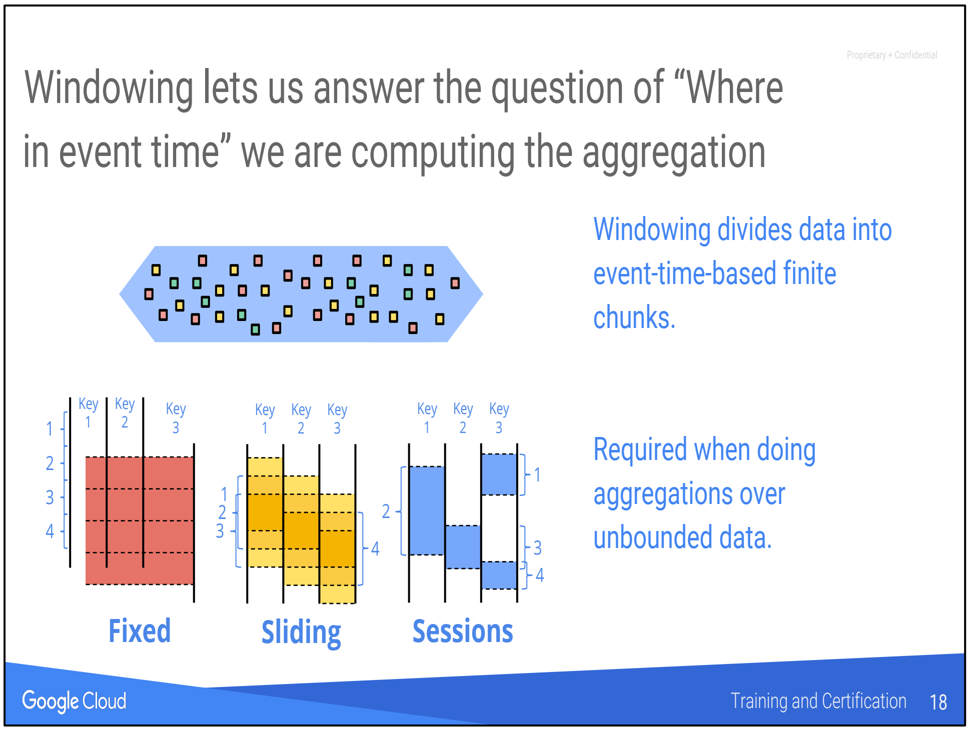
Fixed or slowly scaled clusters are a waste



Google Cloud Training and Certification 17

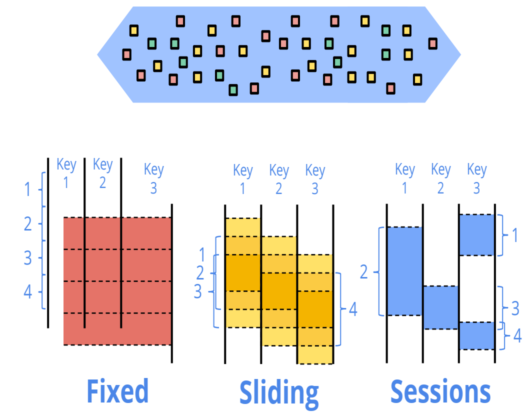
Notes:

In terms of scalability you don't want to over-provision (wasteful) or under-provision (dont just average out everything) You want to autoscale.



Proprietary + Confidential

Windowing lets us answer the question of "Where in event time" we are computing the aggregation



Windowing divides data into event-time-based finite chunks.

Required when doing aggregations over unbounded data.

Google Cloud

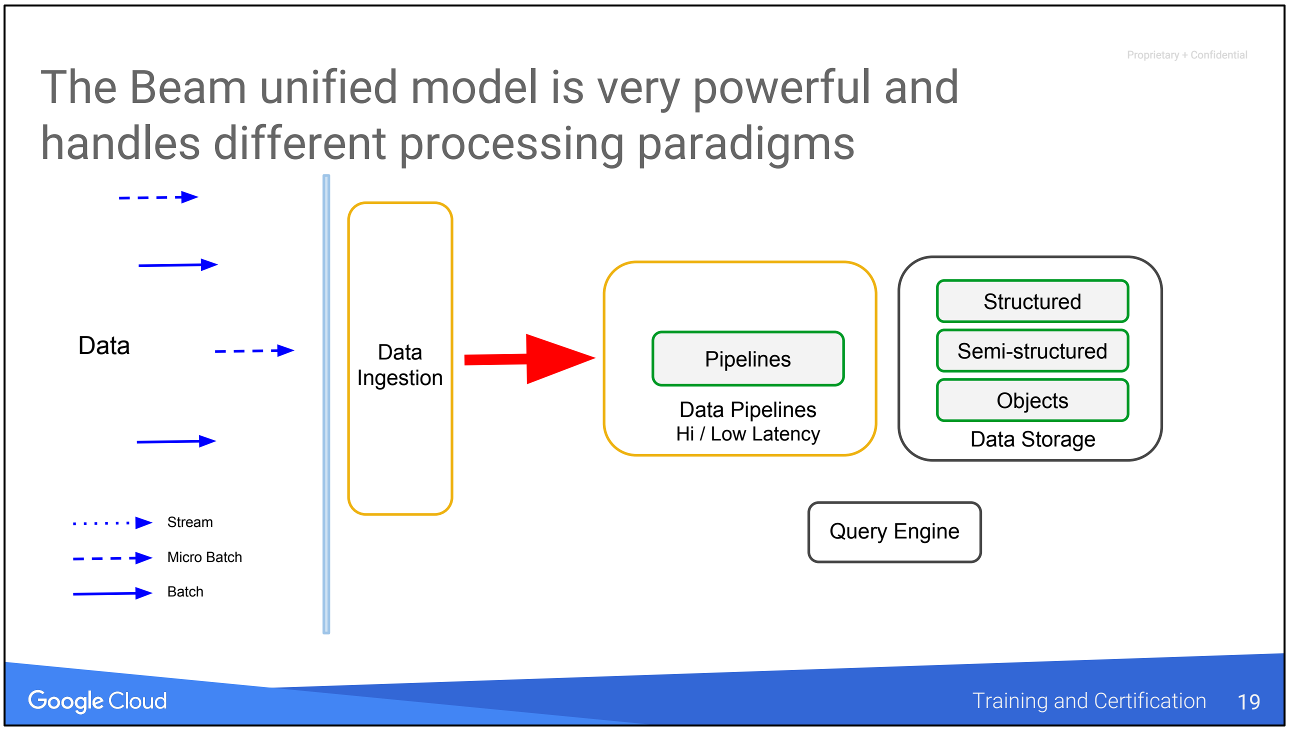
Training and Certification 18

Notes:

Windowing lets you do aggregations, etc

- Different windows are possible in Dataflow
- Some examples, are fixed windows, for example, hourly, daily, every minute, and so on. These windows typically are the same across all processing keys.
- Another example is sliding windows. These are similar to fixed, windows, but made up of may smaller "panes" of fixed windows, that allow the window to slide along as time moves forward.
- Session windows are another example. Sessions are periods of event activity separated by periods of inactivity. Session based looks for gaps.
- An Interesting thing about sessions is that session window boundaries are functions of the data, and are not knowable prior.
- Only at the time of processing are we able to determine where in event time the window boundaries will be.
- Sessions are an example of windowing strategies that are dynamically

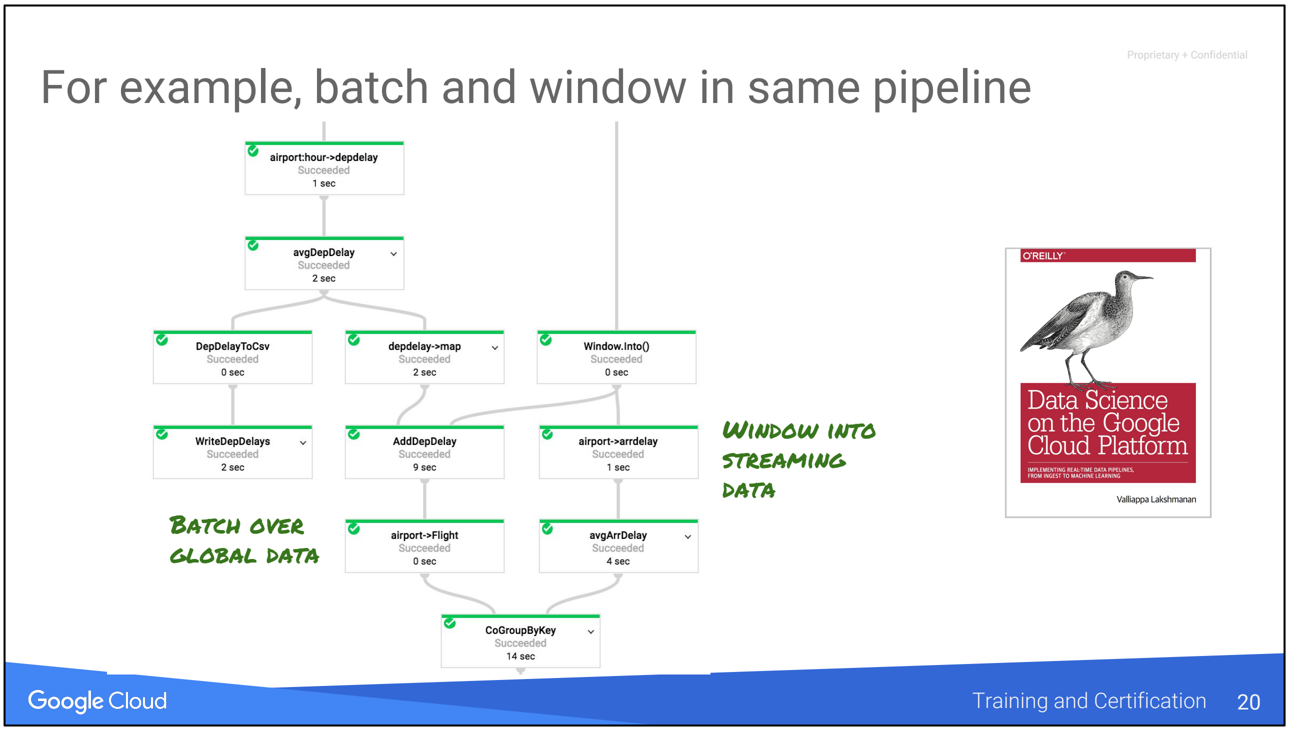
- based on data, and therefore not easily possible with traditional batch systems.



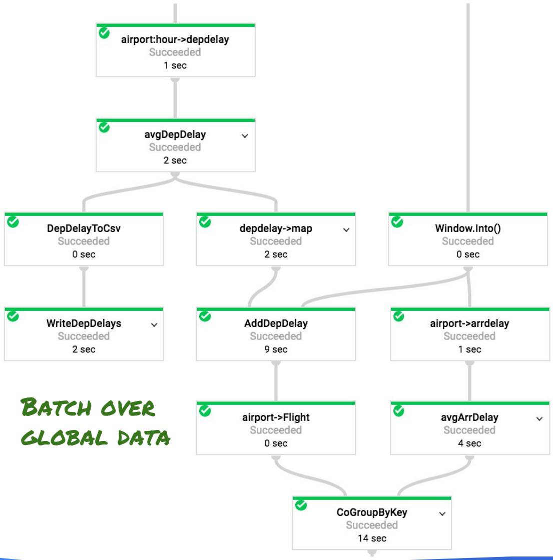
The Beam unified model is very powerful and handles different processing paradigms

Notes:

Whether stream, or micro-batch, or batch.

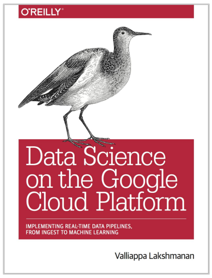


For example, batch and window in same pipeline



WINDOW INTO STREAMING DATA

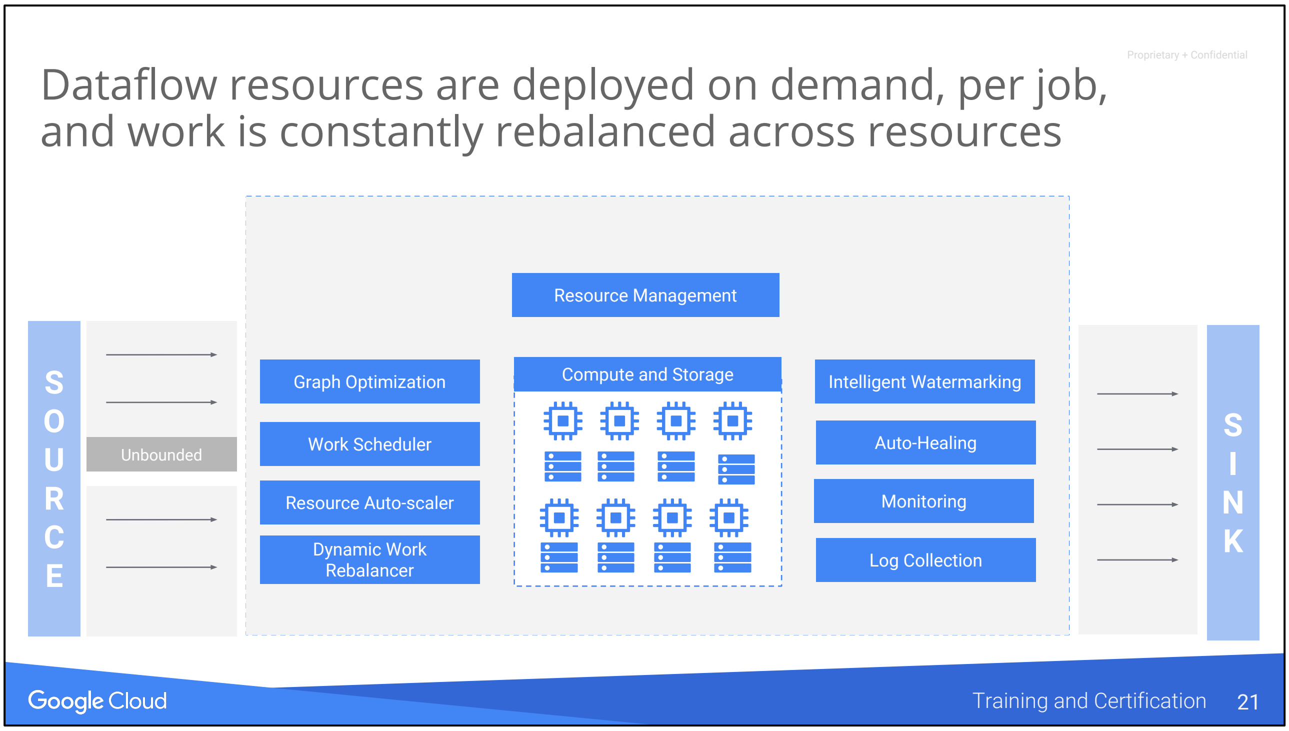
BATCH OVER GLOBAL DATA


O'REILLY
Data Science on the Google Cloud Platform
IMPLEMENTING REAL-TIME DATA PIPELINES, FROM INGEST TO MACHINE LEARNING
Valliappa Lakshmanan

Notes:

This is a pipeline from “Data Science on Google Cloud Platform”.

Average waiting at different airports are different. Departure delay is global....over historical data.



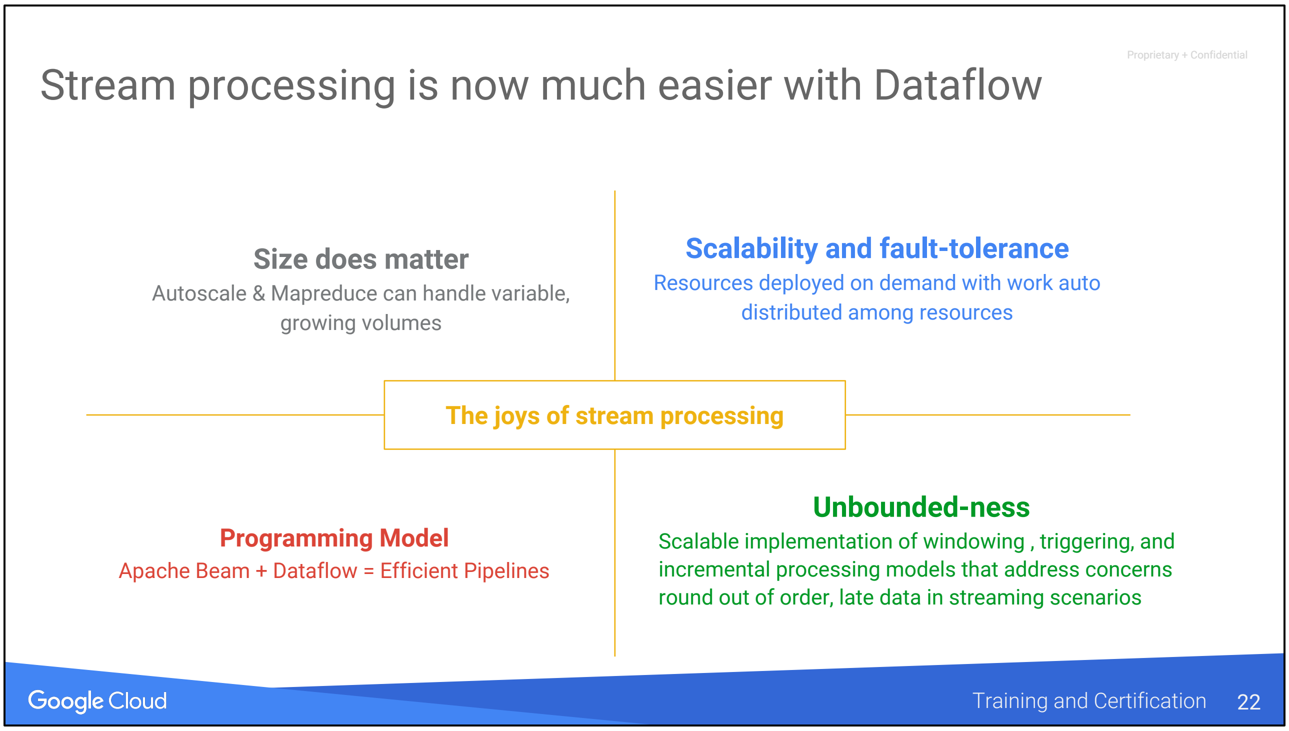
Dataflow resources are deployed on demand, per job, and work is constantly rebalanced across resources

Notes:

Units of work is constantly going to be rebalanced. Execution framework is smart in how it manages the graph.

Cloud Dataflow Resource Management:
Resources deployed on demand and on a per job basis
Resources are torn down at end of job, stage, or on downscaling
Work scheduled on a resource is guaranteed to be processed
Work can be dynamically rebalanced across resources -- this provides fault-tolerance

No more waiting for other jobs to finish
No more preemptive scheduling



Stream processing is now much easier with Dataflow

Size does matter
Autoscale & Mapreduce can handle variable, growing volumes

Scalability and fault-tolerance
Resources deployed on demand with work auto distributed among resources

The joys of stream processing

Programming Model
Apache Beam + Dataflow = Efficient Pipelines

Unbounded-ness
Scalable implementation of windowing , triggering, and incremental processing models that address concerns round out of order, late data in streaming scenarios

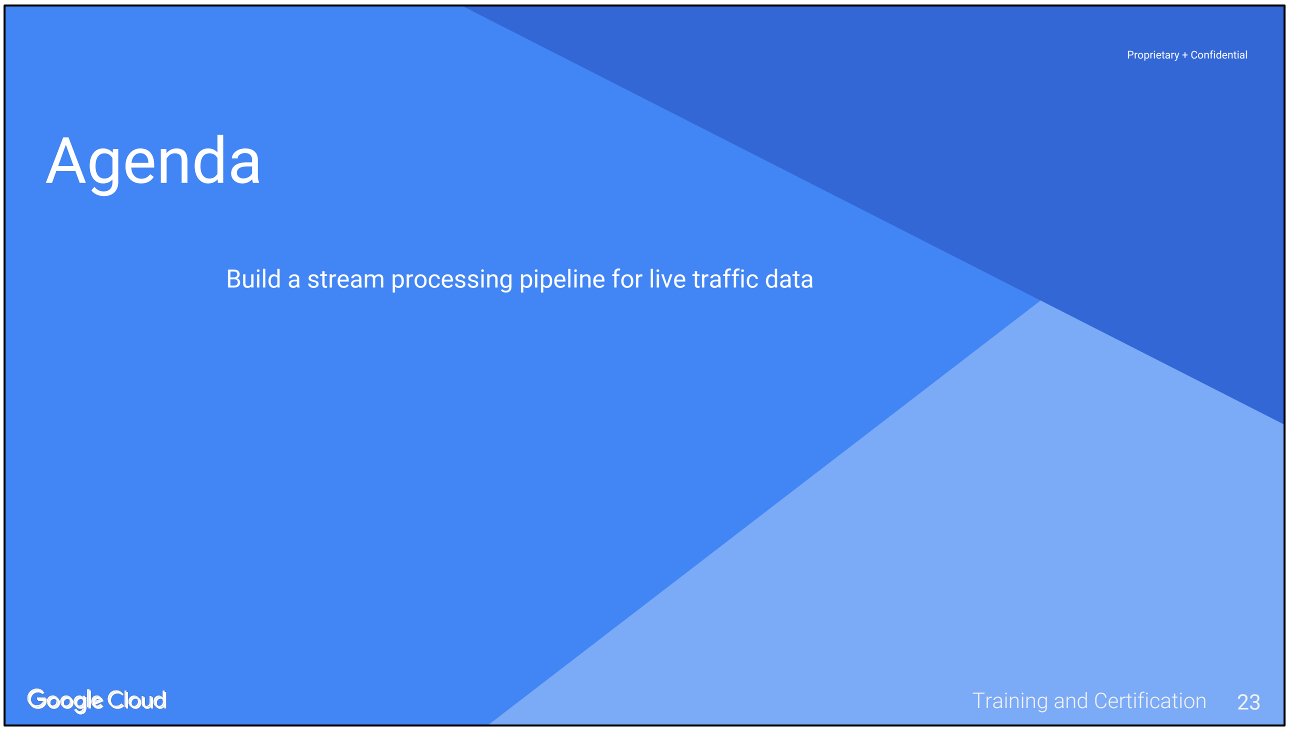
Notes:

Data size: our traffic data from sensors will only grow bigger,

Unbounded-ness: As long as there are cars on the roads, there will be streaming data that needs processing, but event vs processing times discrepancy needs to be addressed (late, out-of-order, speculate). In other words solve the When,Where and How without affecting the What (hopefully)

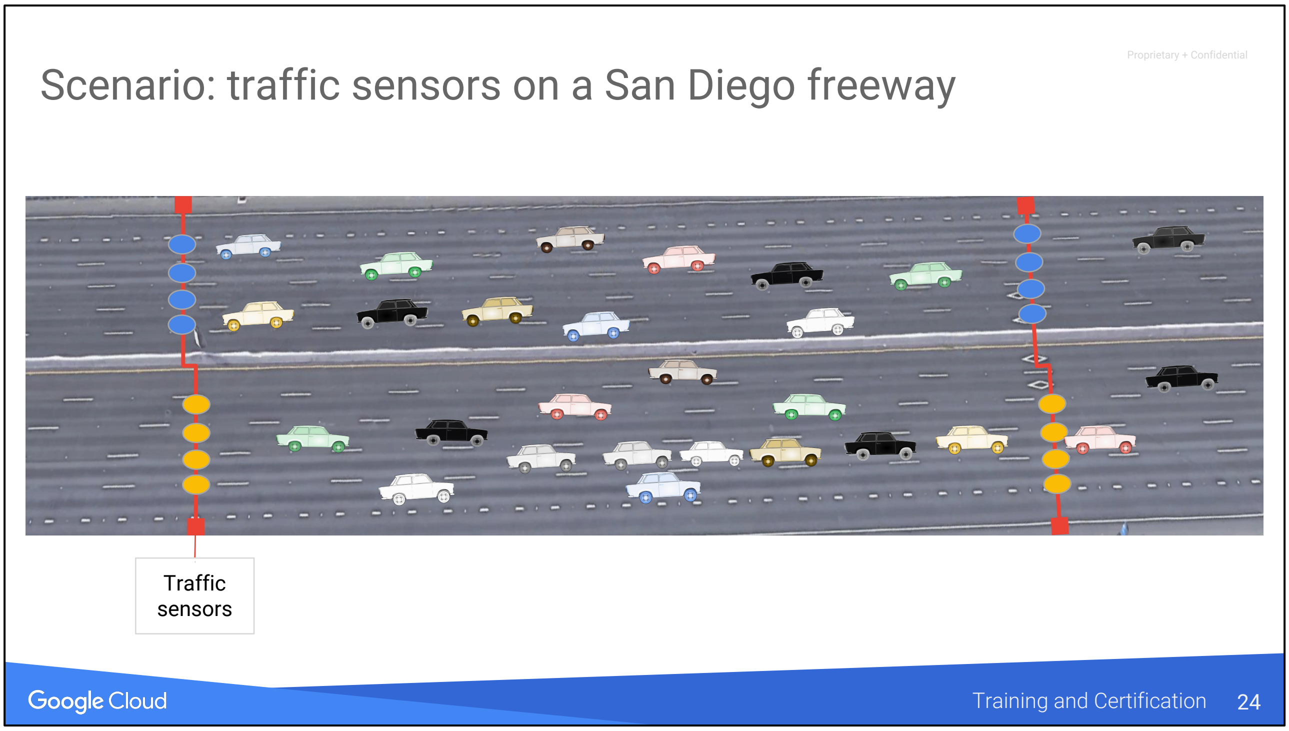
Programming model and application of model to goals: You may find to know how is today’s (Tuesday) traffic looking vs traffic last Tuesday aka you need batch and stream processing...aka ideally a model that can handle both so your speed vs accuracy tradeoff is minimal

Resource management (optimizing performance AND price!)
Scalability and optimization
Fault tolerance

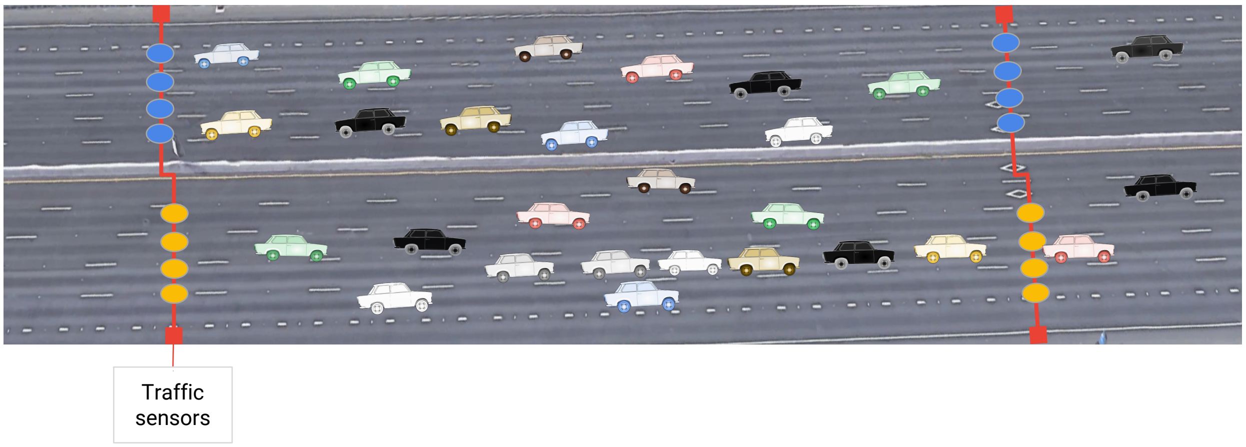


Agenda

Build a stream processing pipeline for live traffic data

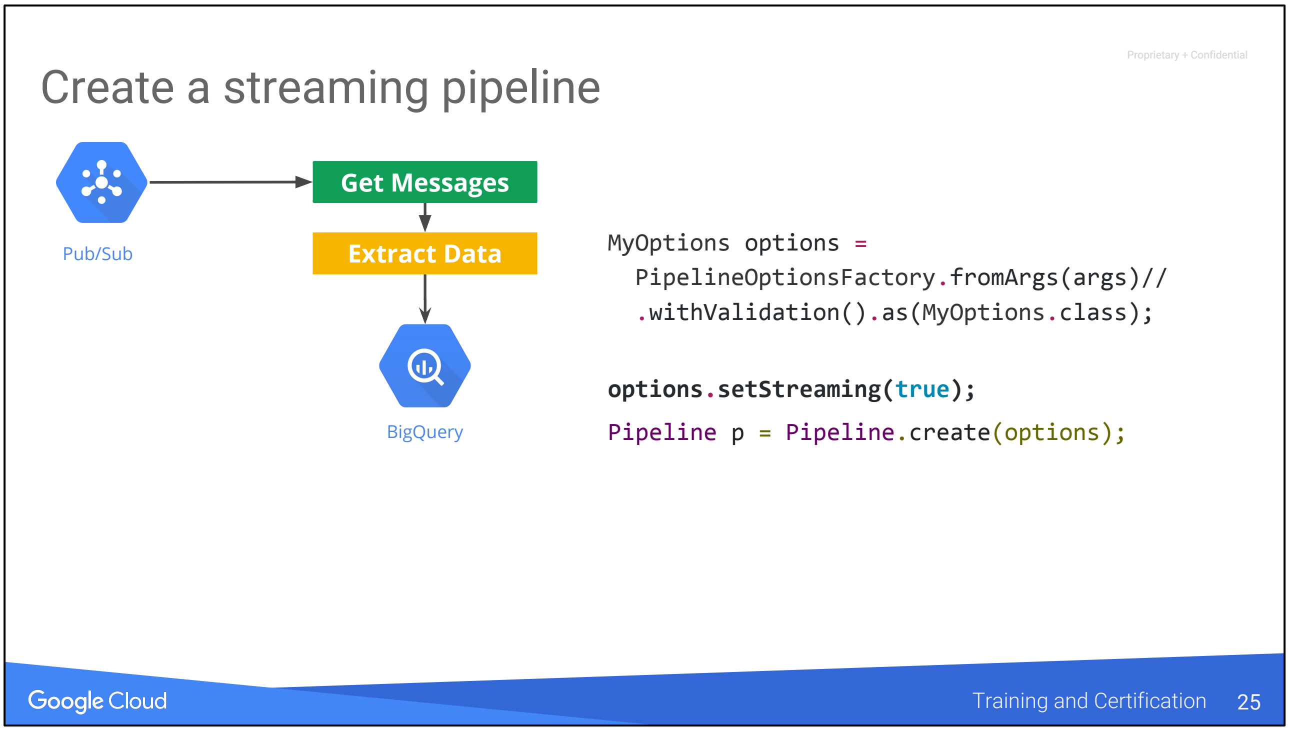


Scenario: traffic sensors on a San Diego freeway

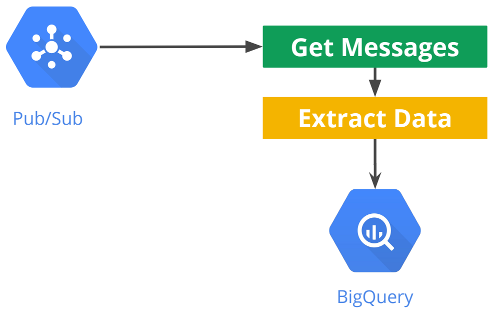


Notes:

Car image: https://pixabay.com/en/trabant-car-transport-white-drive-782799/ (cc0)
The traffic sensors report the speed of the traffic in the lane by computing # of cars that cross in some time-period (it is not clear how long that time-period is) -- let's assume 30s, which is the frequency at which the data are reported.



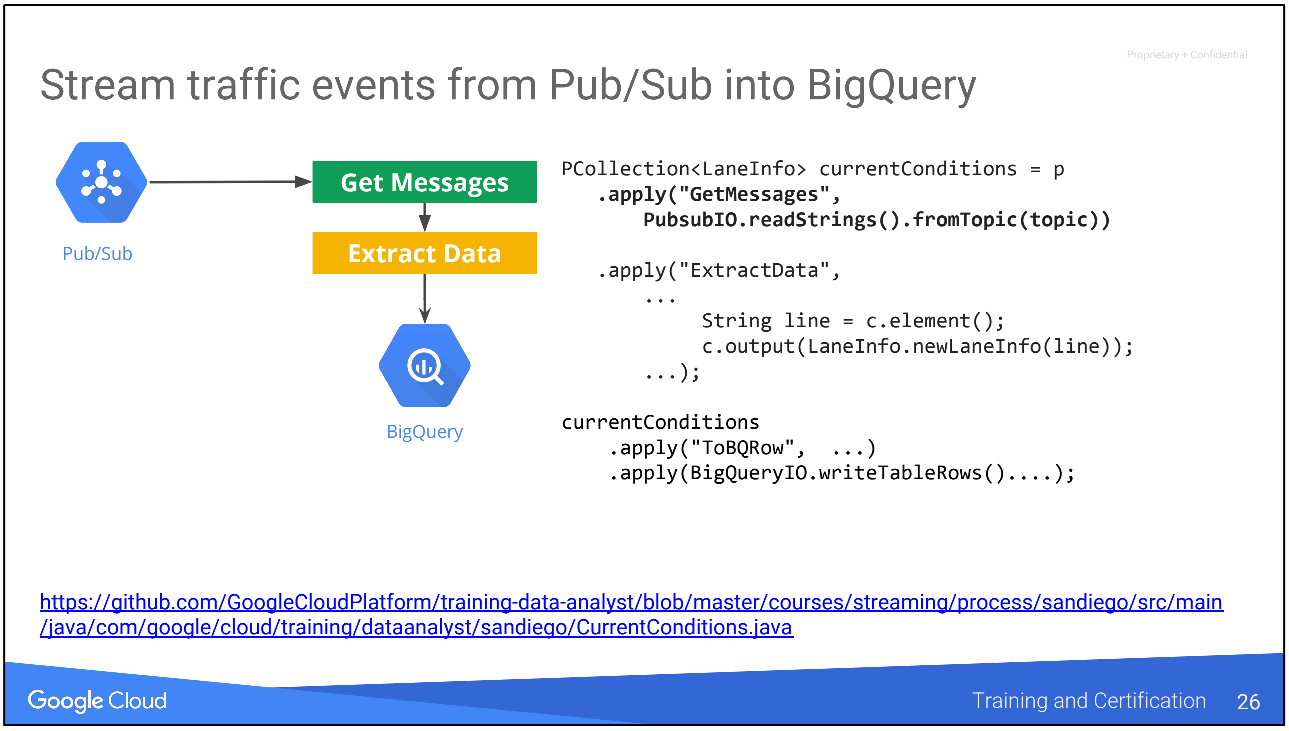
Create a streaming pipeline



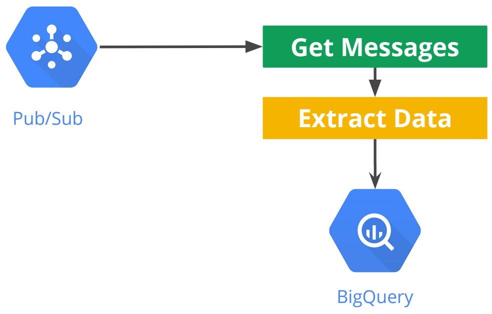
```
MyOptions options =
  PipelineOptionsFactory.fromArgs(args)//
  .withValidation().as(MyOptions.class);

options.setStreaming(true);
Pipeline p = Pipeline.create(options);
```

Notes:

First step is pipeline streaming option is true….i.e., Pipeline will never exit...always stay alive.



Stream traffic events from Pub/Sub into BigQuery

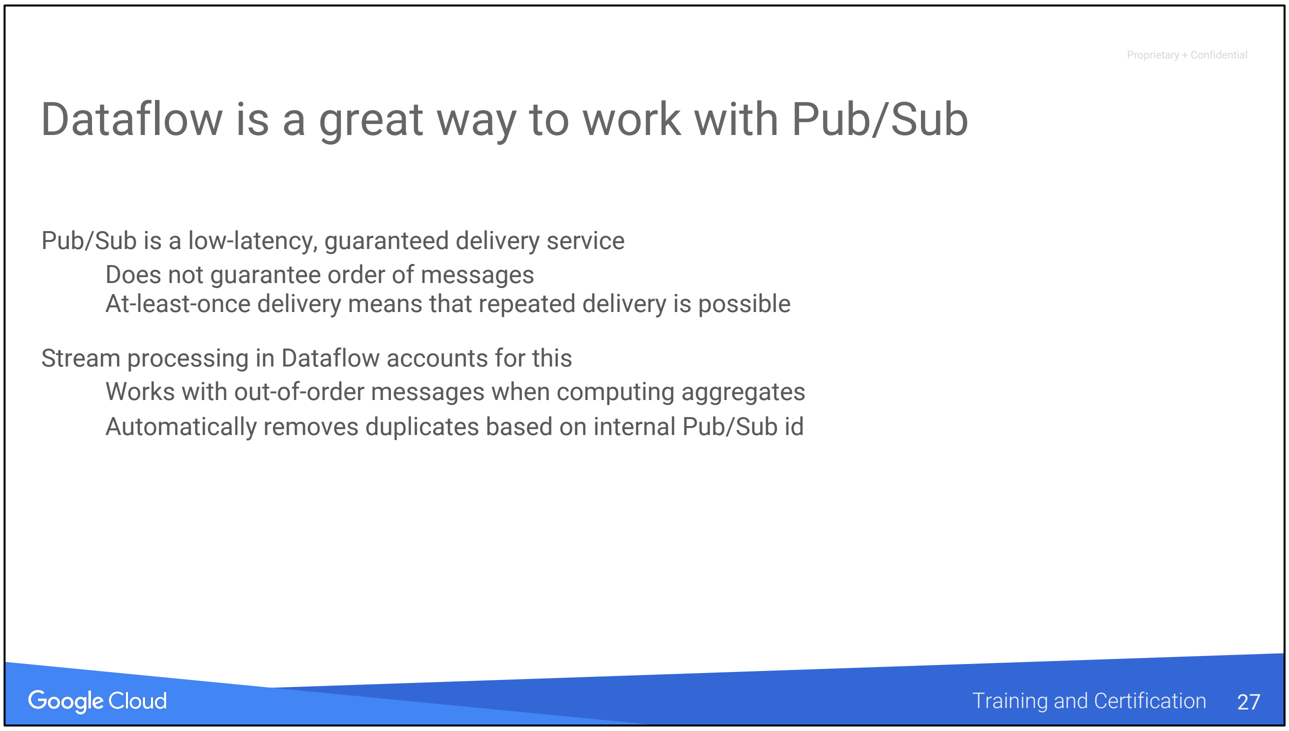


```
PCollection<LaneInfo> currentConditions = p
    .apply("GetMessages",
        PubsubIO.readStrings().fromTopic(topic))

    .apply("ExtractData",
        ...
             String line = c.element();
             c.output(LaneInfo.newLaneInfo(line));
        ...);

currentConditions
     .apply("ToBQRow",  ...)
     .apply(BigQueryIO.writeTableRows()....);
```

https://github.com/GoogleCloudPlatform/training-data-analyst/blob/master/courses/streaming/process/sandiego/src/main/java/com/google/cloud/training/dataanalyst/sandiego/CurrentConditions.java

Notes:

Pubsub is at least once delivery….delivery twice is ok….out of order is ok.
Other than the PubSubIO.readStrings(), nothing here should be surprising.
The code link has the full code.



Dataflow is a great way to work with Pub/Sub

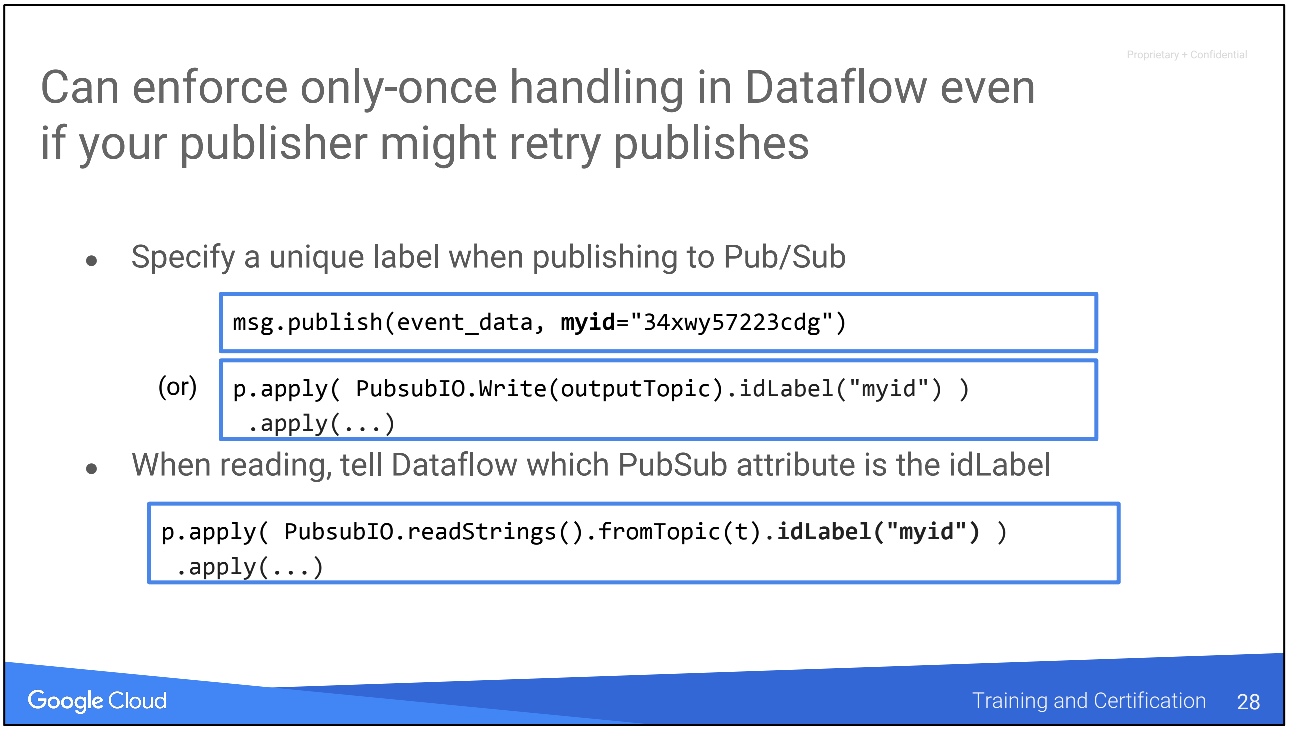
Pub/Sub is a low-latency, guaranteed delivery service
- Does not guarantee order of messages
- At-least-once delivery means that repeated delivery is possible

Stream processing in Dataflow accounts for this
- Works with out-of-order messages when computing aggregates
- Automatically removes duplicates based on internal Pub/Sub id

Notes:

It’s an integrated platform -- because Pub/Sub is about low-latency, guaranteed delivery … out-of-the-box, Dataflow “just works” around these tradeoffs that the Pub/Sub model imposes. So, the end-result is that you get low-latency, guaranteed delivery, and no-duplicates!

But what if your Pub/Sub publisher actually *published* multiple messages? Then, just using the internal Pub/Sub id is not going to be enough, because you have two duplicate messages in the pipeline. In other words, what if the problem is not that Pub/Sub has published the same message twice because it didn’t receive the acknowledge in-time, but the problem is that your server crashed, came back up, and republished the message to Pub/Sub?



Can enforce only-once handling in Dataflow even if your publisher might retry publishes

- Specify a unique label when publishing to Pub/Sub

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

(or)

```
p.apply( PubsubIO.Write(outputTopic).idLabel("myid") )
 .apply(...)
```

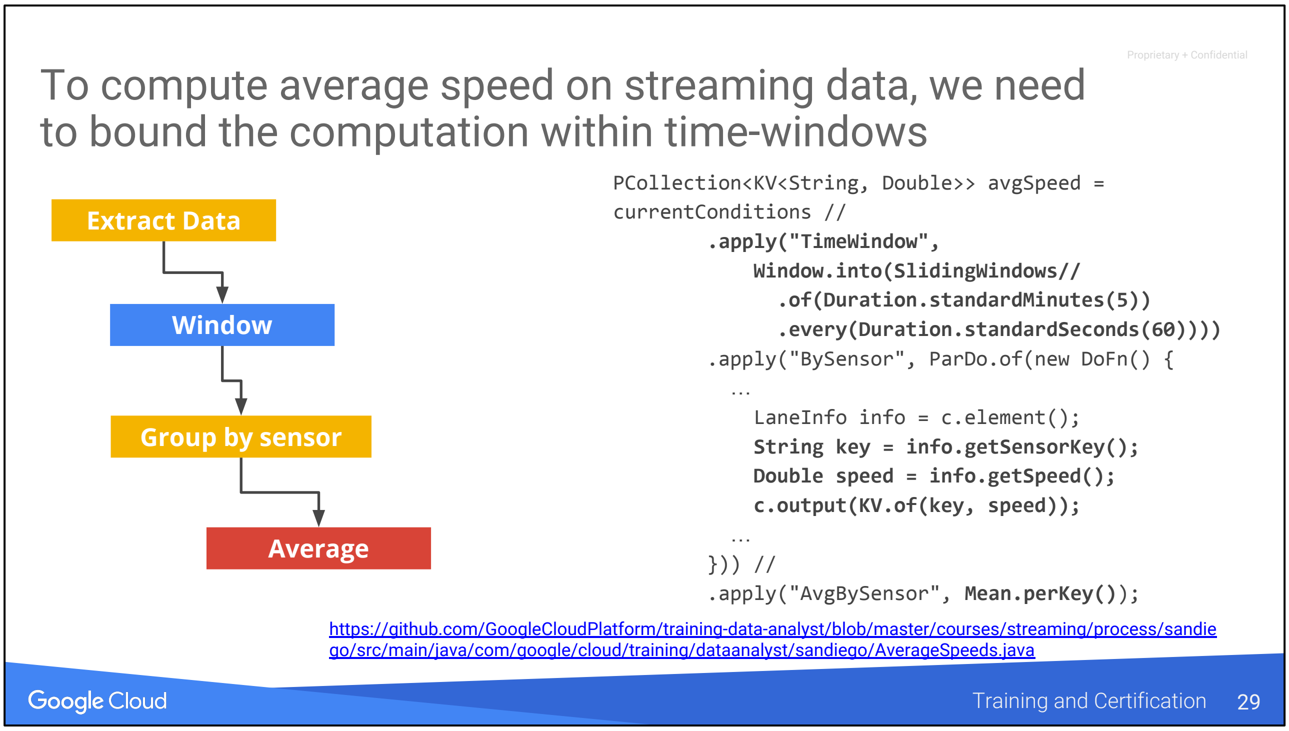
- When reading, tell Dataflow which PubSub attribute is the idLabel

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

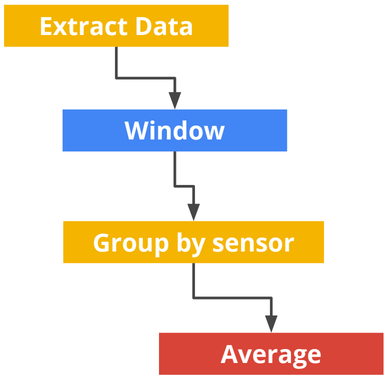
Notes:

For the most part, the default Dataflow behavior (on previous slide) is enough

In the first bullet, the first example (msg.publish) is if you are using the PubSub Python API to publish.
The second example (p.apply) is if you are using Dataflow to publish -- Dataflow will create an appropriately unique ID. (parallel behavior for timestampLabel too).



To compute average speed on streaming data, we need to bound the computation within time-windows

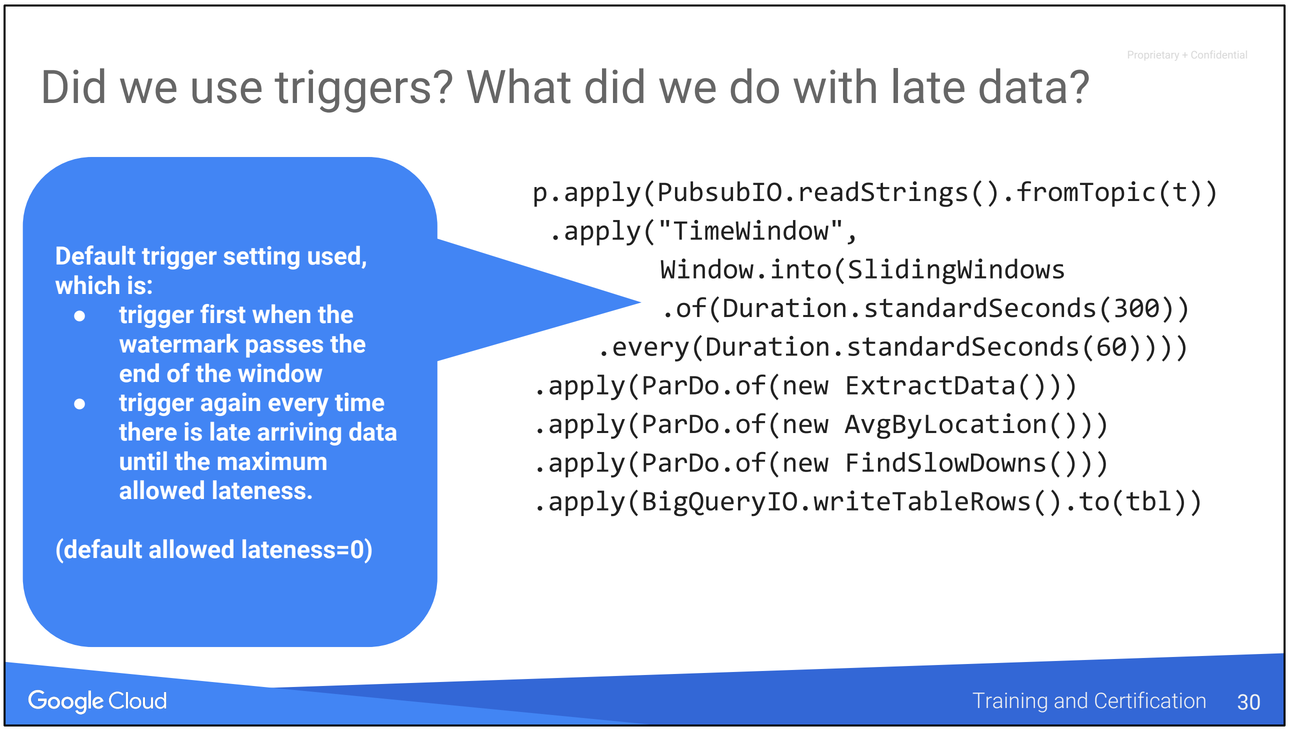


```
PCollection<KV<String, Double>> avgSpeed =
currentConditions //
        .apply("TimeWindow",
            Window.into(SlidingWindows//
              .of(Duration.standardMinutes(5))
              .every(Duration.standardSeconds(60))))
        .apply("BySensor", ParDo.of(new DoFn() {
          …
            LaneInfo info = c.element();
            String key = info.getSensorKey();
            Double speed = info.getSpeed();
            c.output(KV.of(key, speed));
          …
        })) //
        .apply("AvgBySensor", Mean.perKey());
```

https://github.com/GoogleCloudPlatform/training-data-analyst/blob/master/courses/streaming/process/sandiego/src/main/java/com/google/cloud/training/dataanalyst/sandiego/AverageSpeeds.java

Notes:

For example, average is calculated for 5 min windows every 1 minute.
It is important to realize that windows are applied only at the time of a group-by-key. So, simply adding a window to a pipeline doesn't cause anything to happen. It's at the group-by-key stage that the window has an impact. Hence, we apply the window when we need to -- when we need to compute the average within the time-window.



Did we use triggers? What did we do with late data?

Default trigger setting used, which is:

- **trigger first when the watermark passes the end of the window**
- **trigger again every time there is late arriving data until the maximum allowed lateness.**

(default allowed lateness=0)

```
p.apply(PubsubIO.readStrings().fromTopic(t))
 .apply("TimeWindow",
        Window.into(SlidingWindows
        .of(Duration.standardSeconds(300))
    .every(Duration.standardSeconds(60)))
.apply(ParDo.of(new ExtractData()))
.apply(ParDo.of(new AvgByLocation()))
.apply(ParDo.of(new FindSlowDowns()))
.apply(BigQueryIO.writeTableRows().to(tbl))
```

Notes:

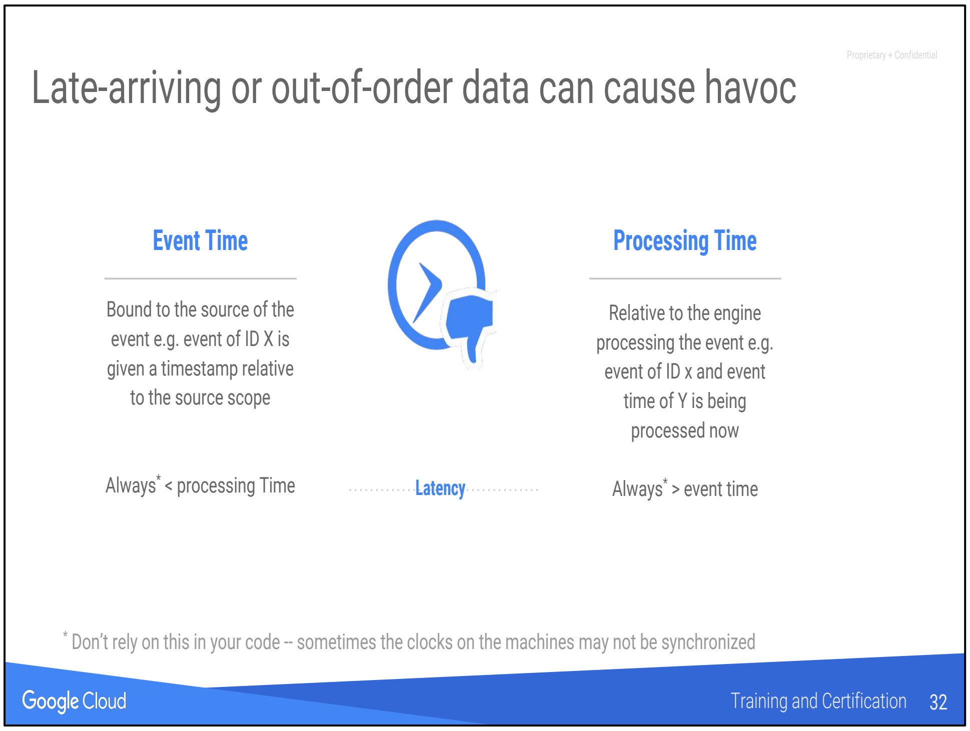
You would typically set trigger after window settings, and also may specify what to do with late data ? do we consider it or throw it? If you want to use it, do you accumulate it

- Triggerng controls how results are delivered to the next transforms.
- The default trigger is a discarding trigger rather than accumulation...which means that late arriving data will not be considered.
- In our traffic example, that means that if data belonging to a window arrives late, it will not be included in the average calculations. This is because allowed_lateness=0. **Default behavior is to ignore late data.**

V1.1: Clarify default trigger behavior. It is to throw away late data but only because allowed_lateness=0; by changing allowed_lateness and keeping default trigger, you will get to process late results. So, non-default window (i.e. window+allowed lateness) & default trigger will involve late-event processing. So, the default trigger behavior

Agenda

Handle late data: watermarks, triggers, accumulation



Proprietary + Confidential

Late-arriving or out-of-order data can cause havoc

Event Time

Bound to the source of the event e.g. event of ID X is given a timestamp relative to the source scope

Always* < processing Time

Processing Time

Relative to the engine processing the event e.g. event of ID x and event time of Y is being processed now

Always* > event time

Latency

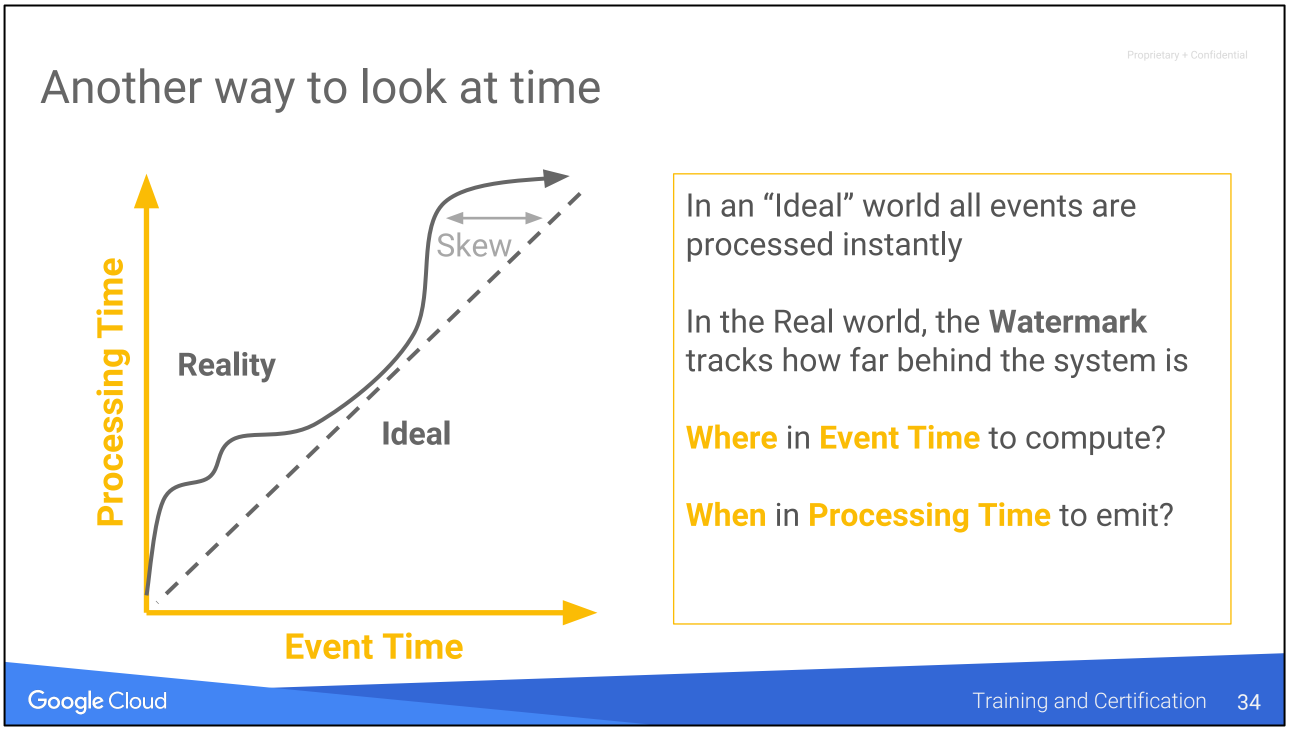
* Don't rely on this in your code -- sometimes the clocks on the machines may not be synchronized

Google Cloud

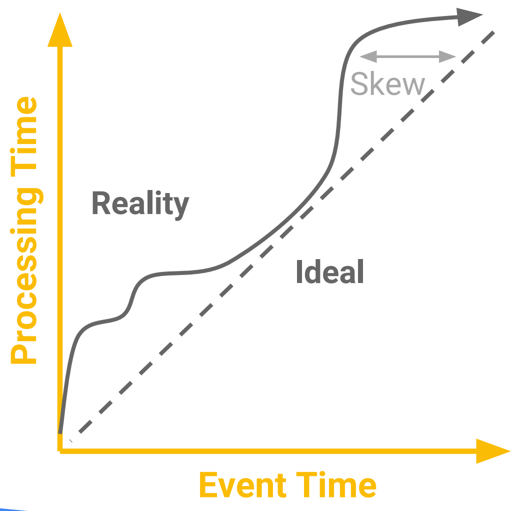
Notes:

Addressing data in event time is a necessary architectural principle to rationalize data completeness.

You want to use a system that has a concept of event time and processing time.



Another way to look at time



In an "Ideal" world all events are processed instantly

In the Real world, the **Watermark** tracks how far behind the system is

Where in **Event Time** to compute?

When in **Processing Time** to emit?

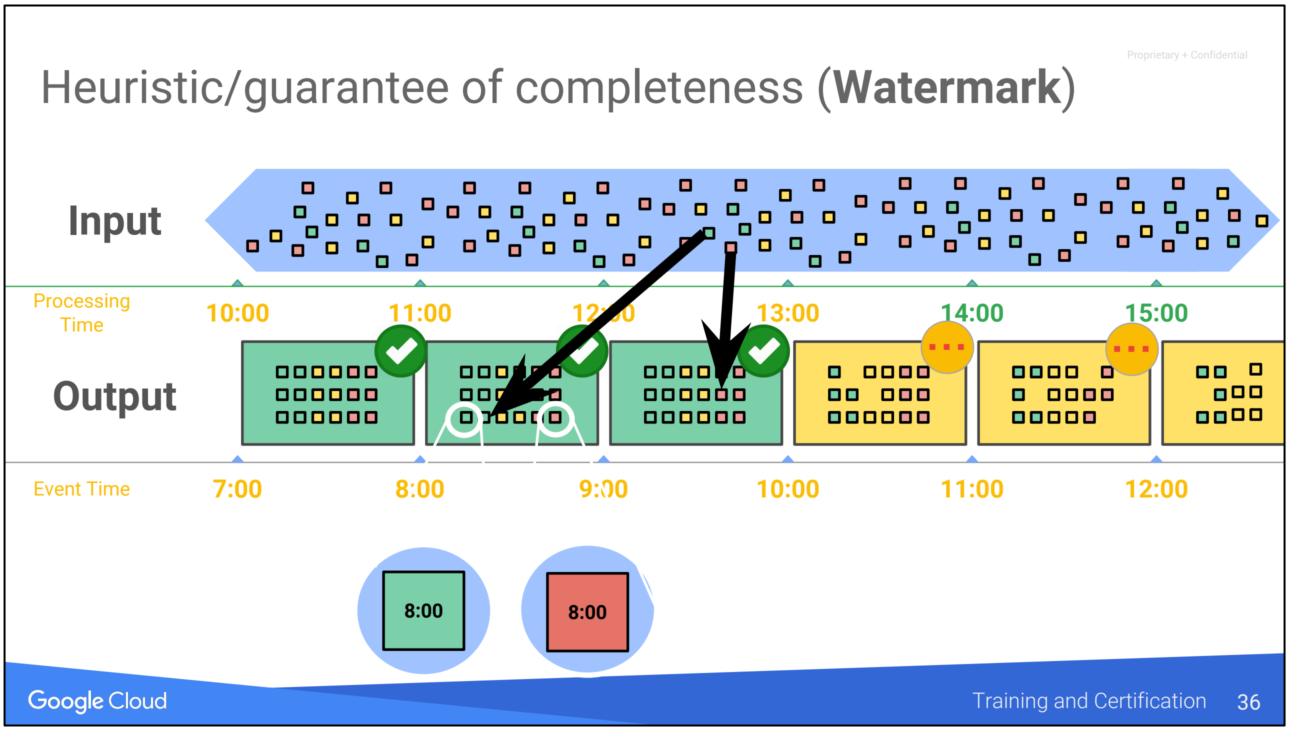
Notes:

In reality, processing time is always greater than event time...because of latency….and that can change over the course of your processing pipeline. This difference is called a watermark. We say when in terms of processing time...not event time. We emit when watermark is greater than X. so you calculate aggregate for 8 -9, but u emit at 9:15

Watermark is age of oldest unprocessed record.if you want more current data...you use short fixed windows or sliding windows

- So windows let us answer the question of "where in event time" we are computing the aggregation.
- We still need to answer the question of "when in processing time" we are ready to emit the answer.
- In order to do that, we need to first better define the relationship between event time and processing time.
- Lets look at this graph
- Within any pipeline when an event is arriving, it has some event time,

- and some time at which it is processed.
- The “ideal” would be that as soon as a record occurs, we immediately process it, and give you a result.
- If that were the case, on this graph here, all the records would line up on the “ideal” diagonal where processing time is equal to event time.
- Reality is not as nice, delays are possible for any number of reasons. The network is slow, processing can be slow, and so on.
- So the events wind up being processed somewhere above this diagonal.
- Dataflow provides something called the watermark which helps reason about completeness. The watermark tells us how far above this diagonal we can draw a line beyond which we don't expect to see any more events.
- The watermark is represented as the read line on this graph.
- If we know everything about our sources perfectly, the watermark too is perfect, meaning it is a guarantee of completeness
- If our knowledge of the sources is imperfect, meaning that our sources can generate unexpected out of order data, the watermark is only a heuristic.
- Either way, the watermark can tell us when we expect to have all data for a given window to the best of the system’s knowledge, and therefore when we should emit the aggregation for that window.



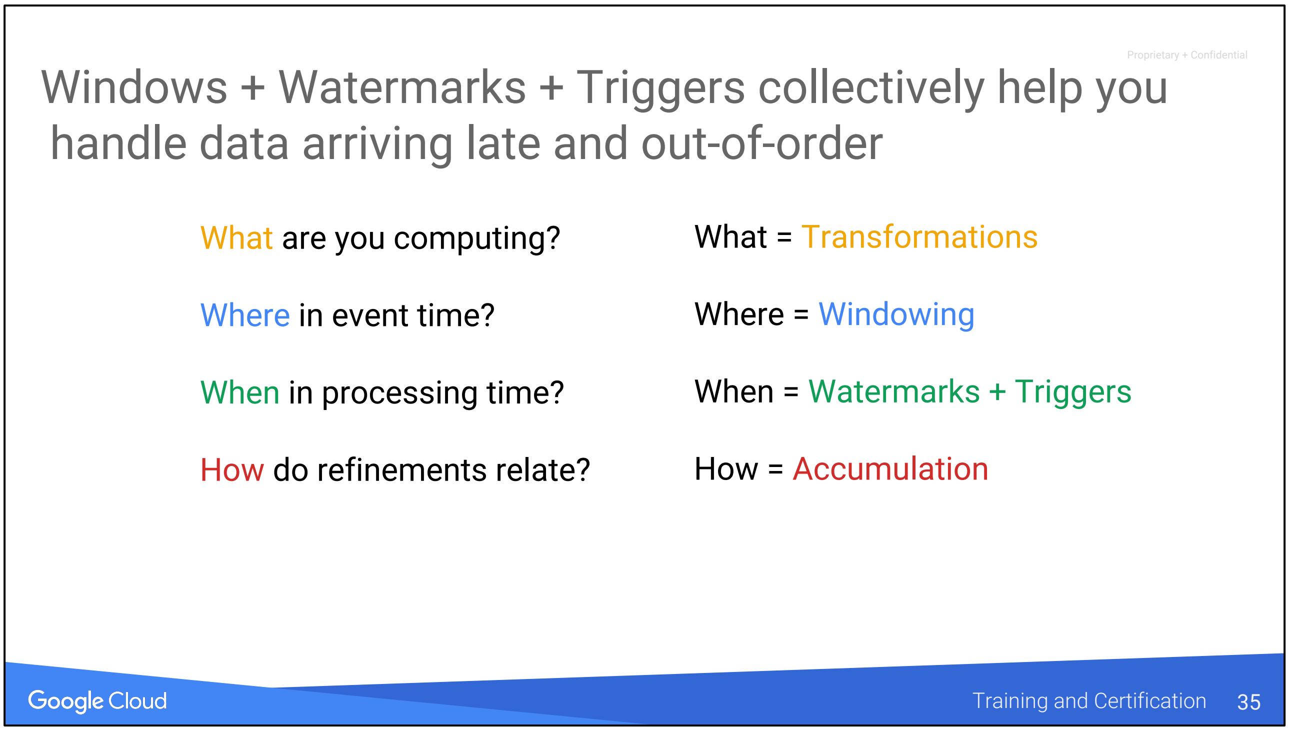
Heuristic/guarantee of completeness (**Watermark**)

Notes:

Watermark as when we do the computation

- If we look again at windowing as a time based shuffle, the watermark tells us when in processing time, the event time windows are expected to be complete, and therefore we can trigger the aggregation.
- The watermark may not be everything that you want to use for triggering output however.
- there are Two major issues to consider with just using watermarks to trigger aggregation.
- The watermark may be “too slow”.
- The watermark tries to be conservative, by keeping track of as many events as it can. A single record can hold up watermark for a while, which is great from the standpoint of data completeness, but maybe you want output faster.
- For example if you are using daily windows, you won’t get the output until the end of the day - but you might want some early results.
- The dataflow triggering API provides support for emitting speculative early results,
- The watermark may also be “too fast”. We already talked about the case

- when we don’t know enough about the input source, and the watermark is only a heuristic.
- Then the watermark can advance even before we see all the data, so we need a way of triggering based on late data, if it is important for your application to deal with it.
- Datataflow triggers api also provides support for this with triggers that handle late data.



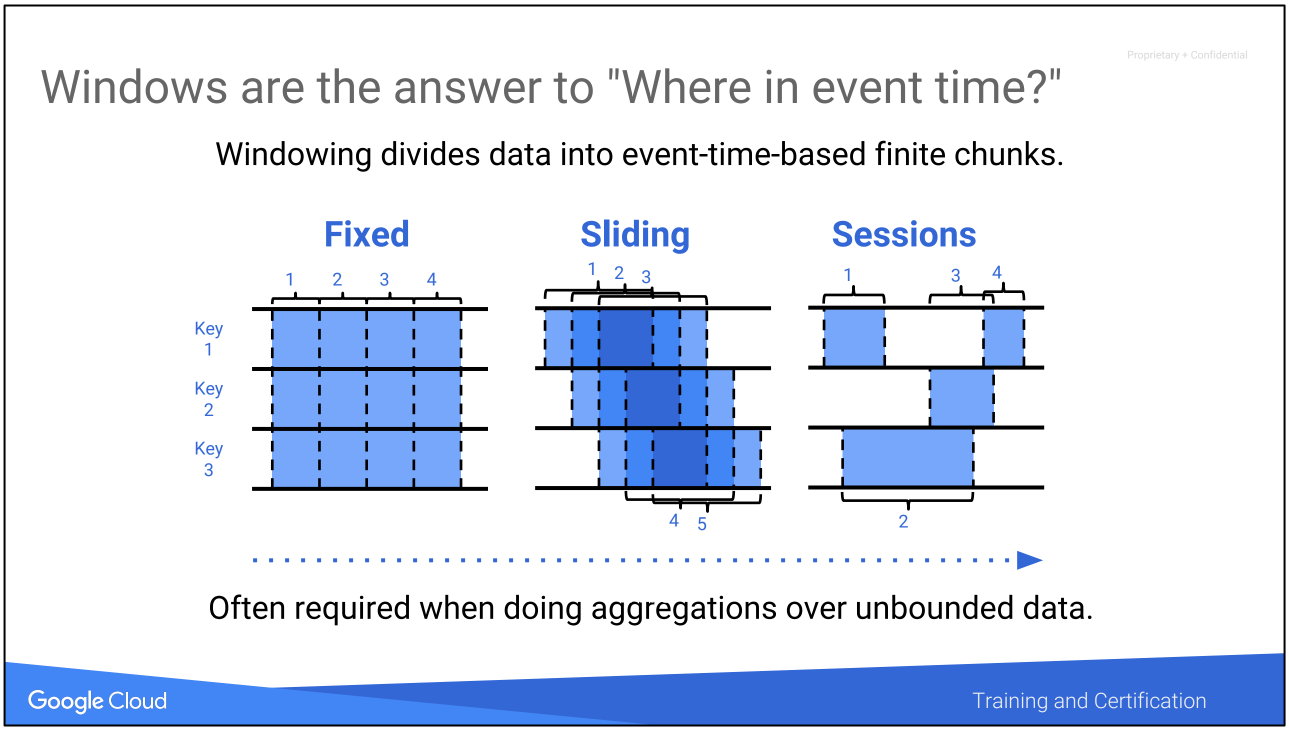
Windows + Watermarks + Triggers collectively help you handle data arriving late and out-of-order

What are you computing?	What = Transformations
Where in event time?	Where = Windowing
When in processing time?	When = Watermarks + Triggers
How do refinements relate?	How = Accumulation

Notes:

Following slides show Sample code with triggers and handling late data.

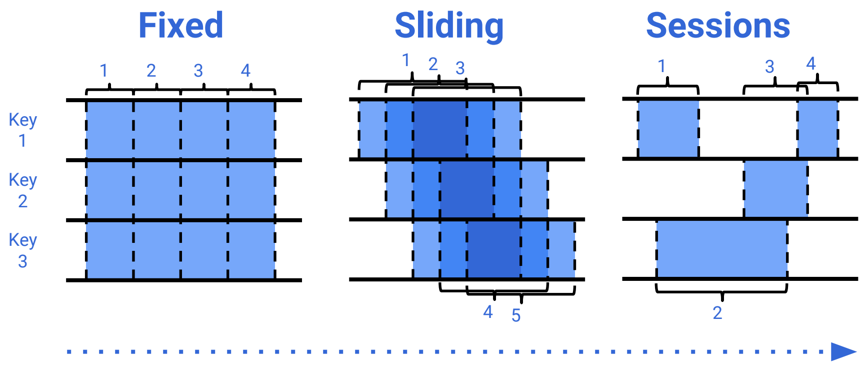
When in processing time is where we bring in our concept of latency.



Proprietary + Confidential

Windows are the answer to "Where in event time?"

Windowing divides data into event-time-based finite chunks.



Often required when doing aggregations over unbounded data.

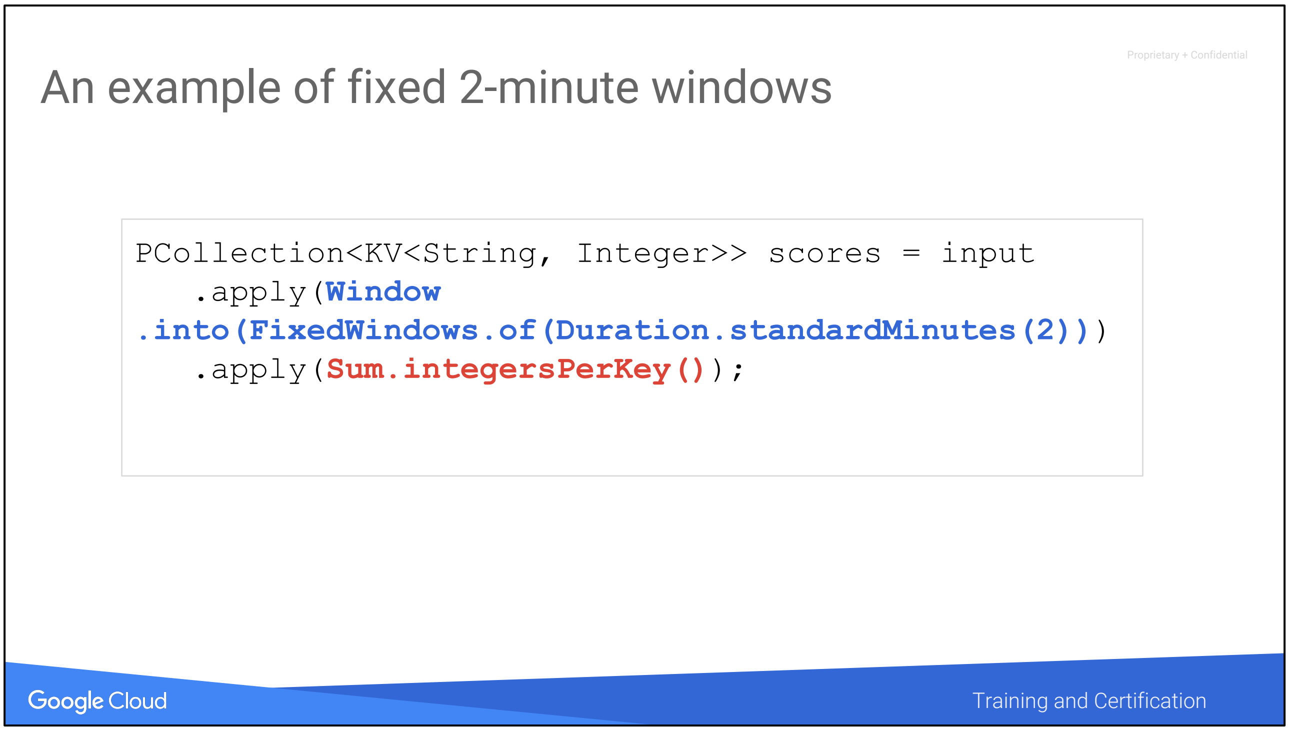
Google Cloud Training and Certification

Windowing answers the question by creating individual results for different slices of event time.

Windowing divides a PCollection up into finite chunks based on the event time of each element. It can be useful for computations over both bounded and unbounded PCollections, but it's required when trying to create aggregations on infinite data.

There are many different ways to implement windows on data, but some of the most common methods include fixed time (for example; hourly, daily, monthly), overlapping sliding windows (for example; the last 24 hours worth of data, every hour), and session-based windows that capture bursts of user activity.

Based on original slides by Tyler Akidau & Frances Perry, April 2016



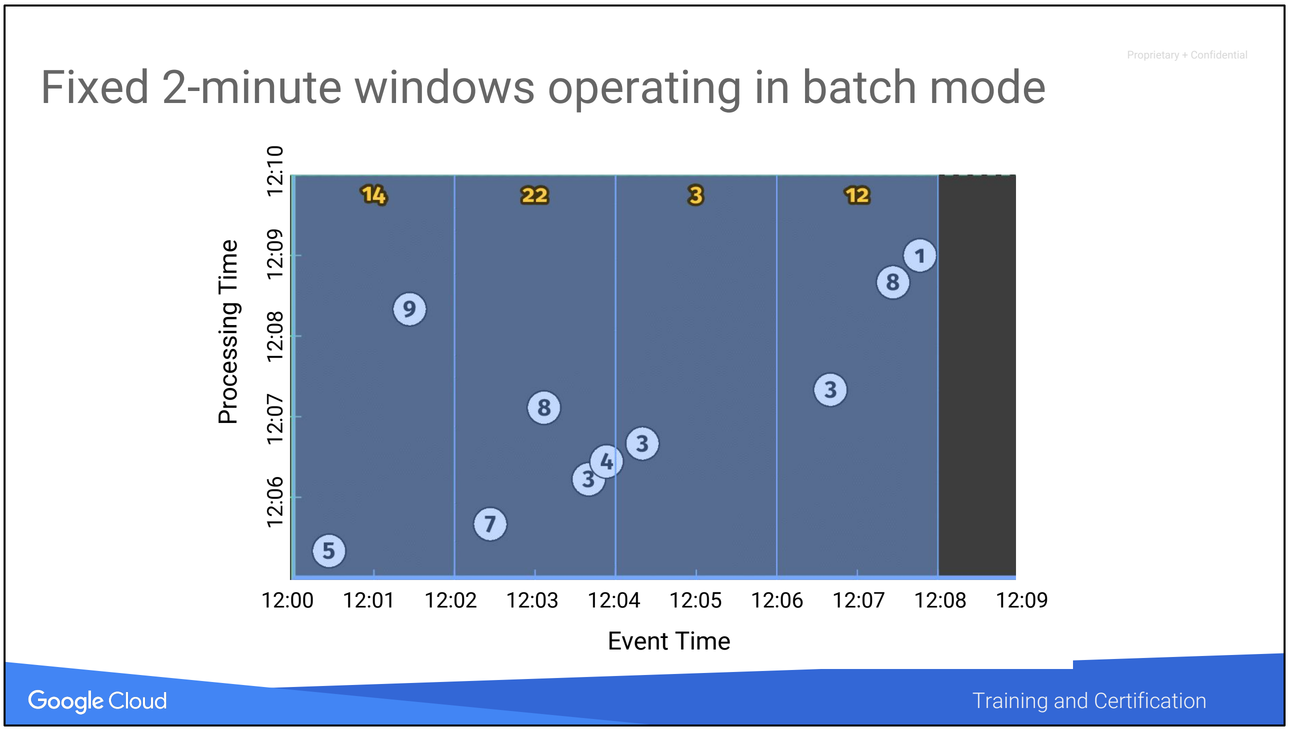
An example of fixed 2-minute windows

```
PCollection<KV<String, Integer>> scores = input
   .apply(Window
.into(FixedWindows.of(Duration.standardMinutes(2)))
   .apply(Sum.integersPerKey());
```

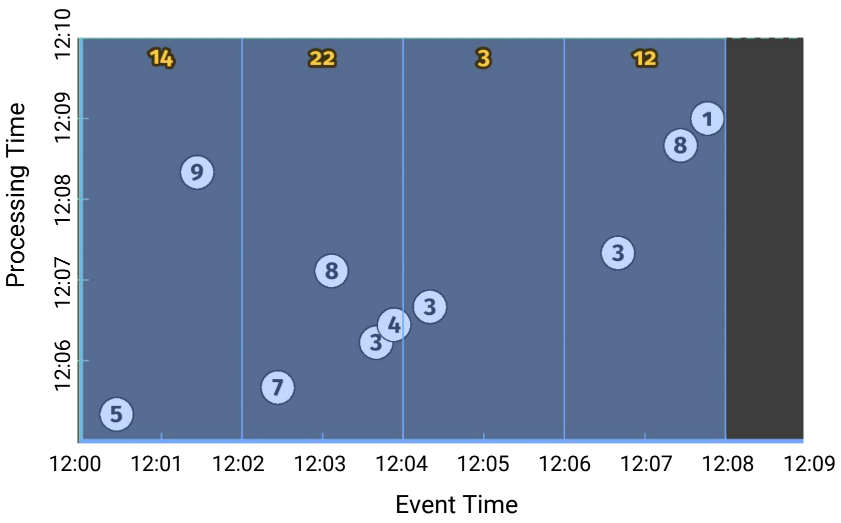
This example uses fixed windows that are 2 minutes long. The critical code is highlighted and colored.

How would this code behave in batch execution?

Based on original slides by Tyler Akidau & Frances Perry, April 2016



Fixed 2-minute windows operating in batch mode

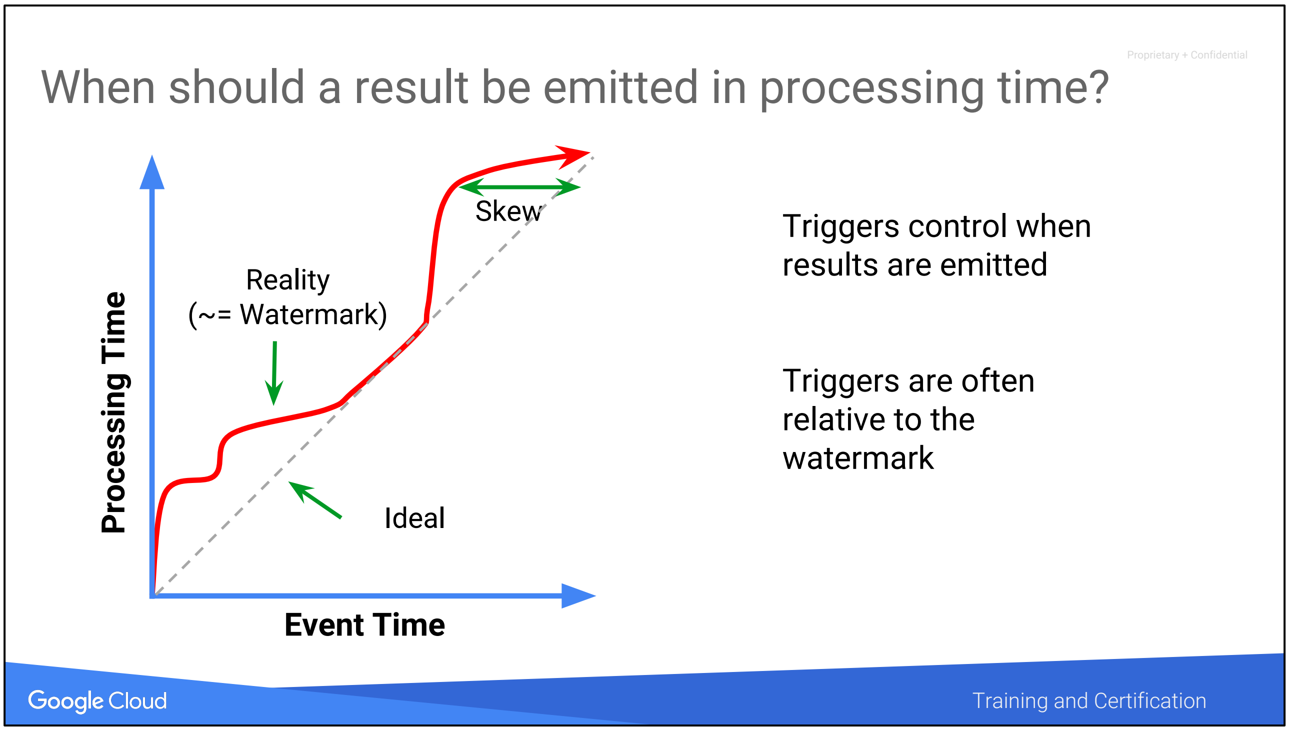


Examine the execution. An independent answer is calculated for every two minute period of event time.

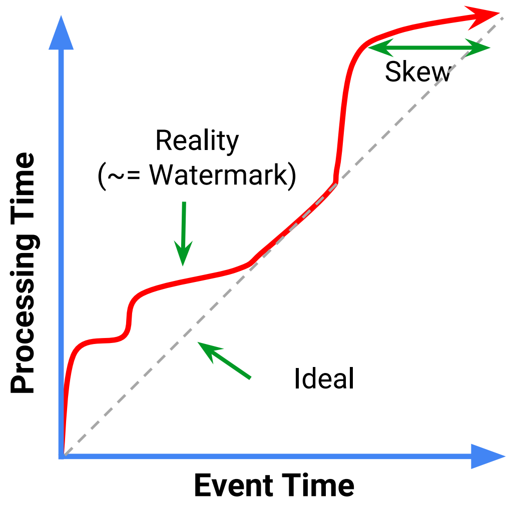
But the system is still waiting until the entire computation has completed before it emits any results. That works fine for bounded data sets, where processing will eventually finish. Bit it will not work when attempting to process an infinite amount of data (unbounded).

To produce results when they are ready, a mechanism is required to control when each result will be produced.

Based on original slides by Tyler Akidau & Frances Perry, April 2016



When should a result be emitted in processing time?



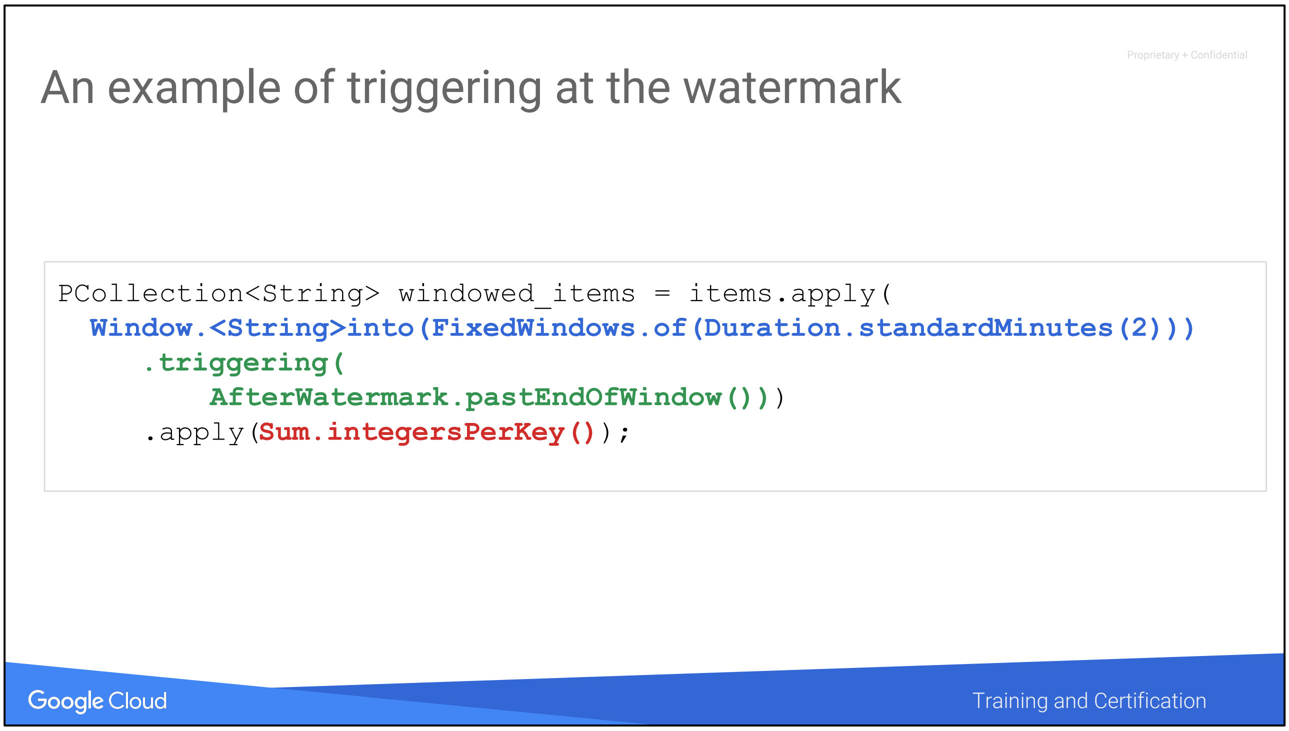
Triggers control when results are emitted

Triggers are often relative to the watermark

Triggers control when a result should be emitted for a window.

Triggers are often relative to the watermark, which is a heuristic about event time progress.

Based on original slides by Tyler Akidau & Frances Perry, April 2016



An example of triggering at the watermark

```
PCollection<String> windowed_items = items.apply(
  Window.<String>into(FixedWindows.of(Duration.standardMinutes(2)))
     .triggering(
         AfterWatermark.pastEndOfWindow()))
     .apply(Sum.integersPerKey());
```

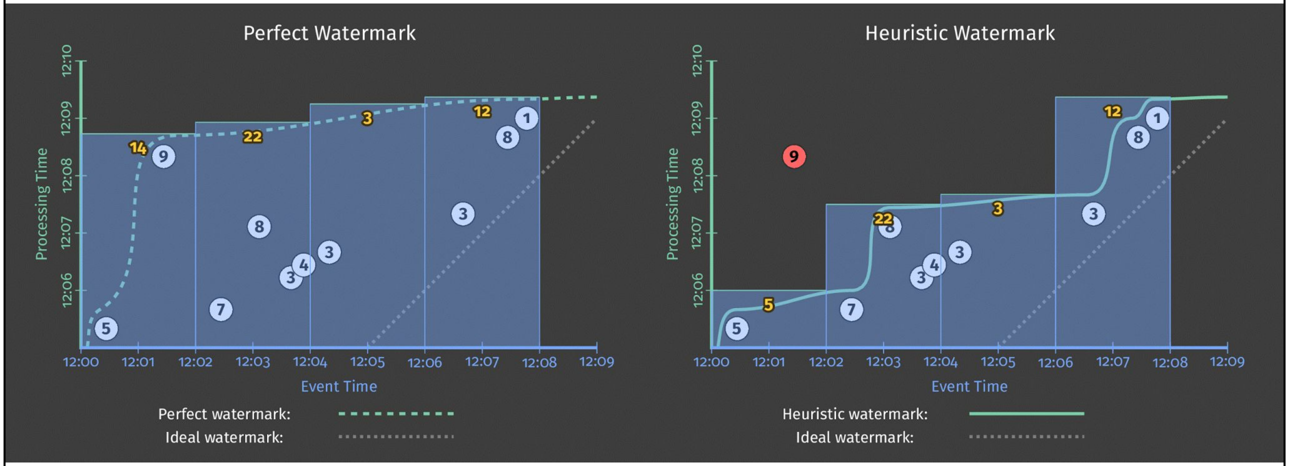
This code implements that
A request for results to be emitted will be sent when all elements for a given window have been seen.

Triggering at the watermark is the default -- It is shown here explicitly for clarity.

How does this trigger behave when executing the pipeline?

Based on original slides by Tyler Akidau & Frances Perry, April 2016

Triggering at the watermark and late arriving data

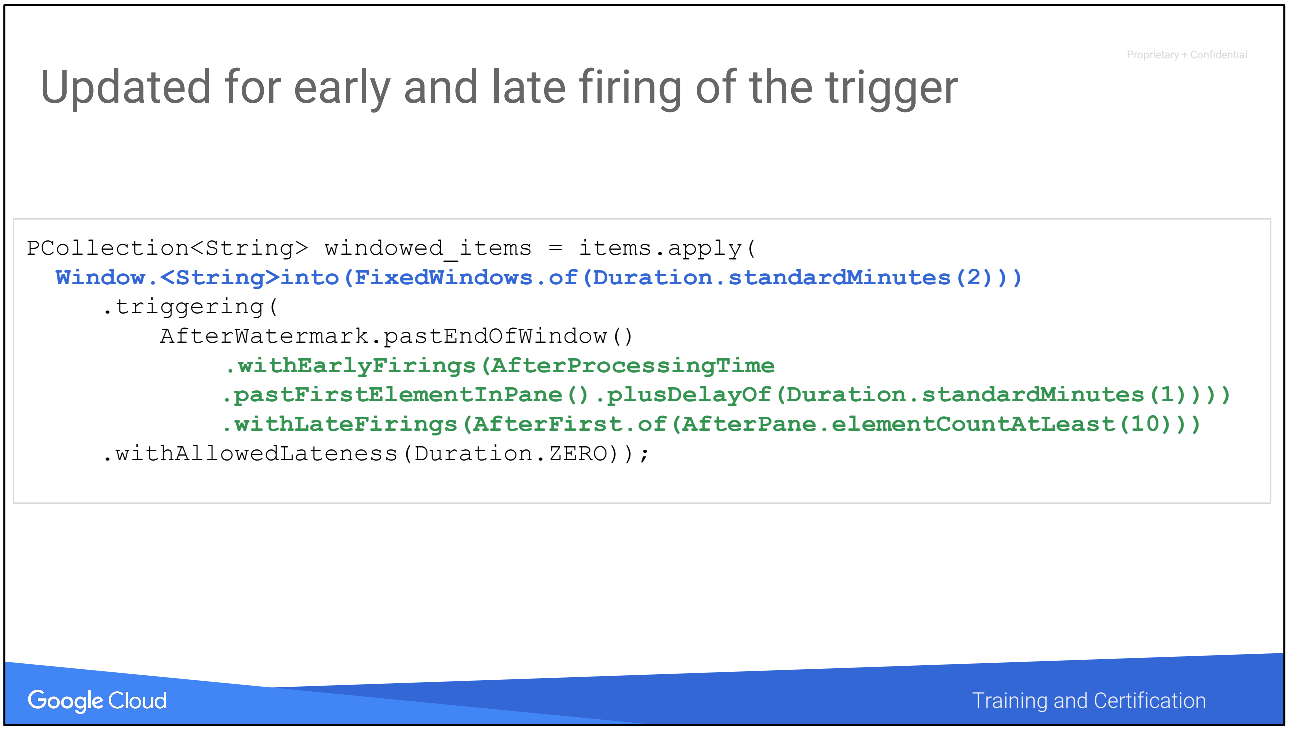


There are four sets of two minute windows. The result from each window is emitted as soon as the watermark passes.

The graph on the left shows what would happen with a perfect watermark. But the graph on the right is what happens when relying on a heuristic watermark. Element 9 was very late due to a lack of cell signal in the elevator, and was much more delayed than expected -- so in this case it came in late after the watermark and was never included in the results.

Even though the results are generated as soon as the watermark passes, it can be useful to get early speculative results. For example, in an application that is based on daily windows, it might still be valuable to get results every two hours. For circumstances like this, more advanced trigger scenarios are possible.

Based on original slides by Tyler Akidau & Frances Perry, April 2016



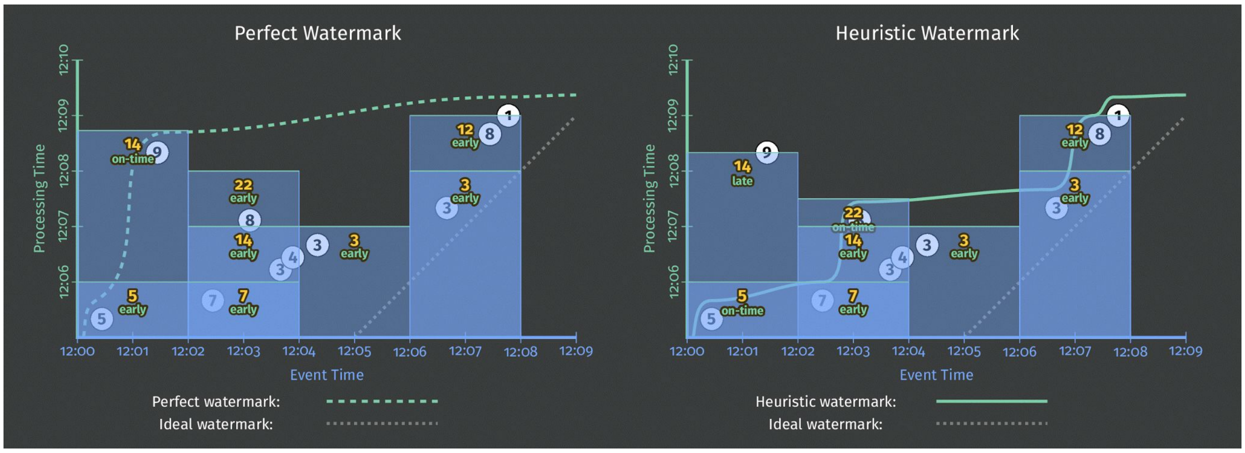
Updated for early and late firing of the trigger

```
PCollection<String> windowed_items = items.apply(
  Window.<String>into(FixedWindows.of(Duration.standardMinutes(2)))
      .triggering(
          AfterWatermark.pastEndOfWindow()
               .withEarlyFirings(AfterProcessingTime
               .pastFirstElementInPane().plusDelayOf(Duration.standardMinutes(1))))
               .withLateFirings(AfterFirst.of(AfterPane.elementCountAtLeast(10)))
      .withAllowedLateness(Duration.ZERO));
```

In this example the trigger has been updated so that it not only emits elements at the watermark, but also creates speculative results every minute. It continues to fire as the watermark passes and also when late data arrives.

Based on original slides by Tyler Akidau & Frances Perry, April 2016

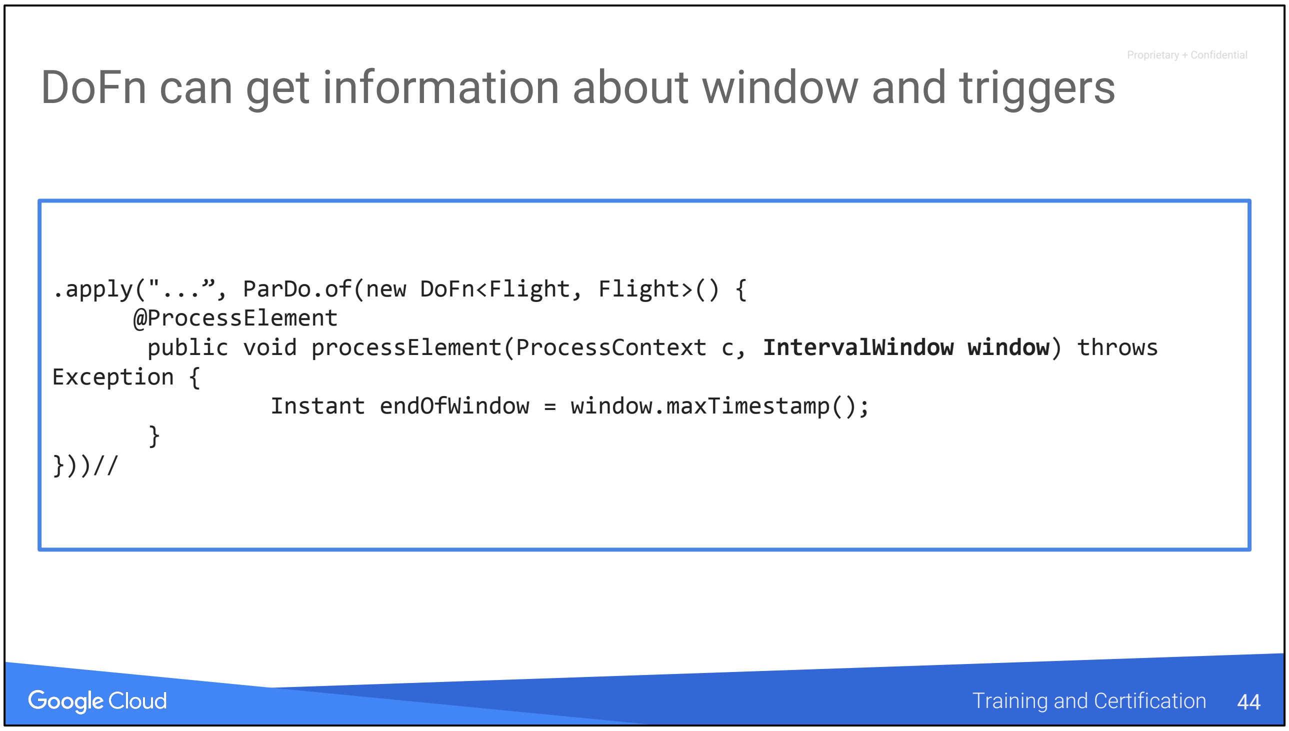
Update (late) results and speculative (early) results



Now the first window produces a result at the watermark, but also gives an updated result when late value 9 comes in.

The second window has two speculative results, with values 7 and 14, before the outputting the final result of 22.

Based on original slides by Tyler Akidau & Frances Perry, April 2016

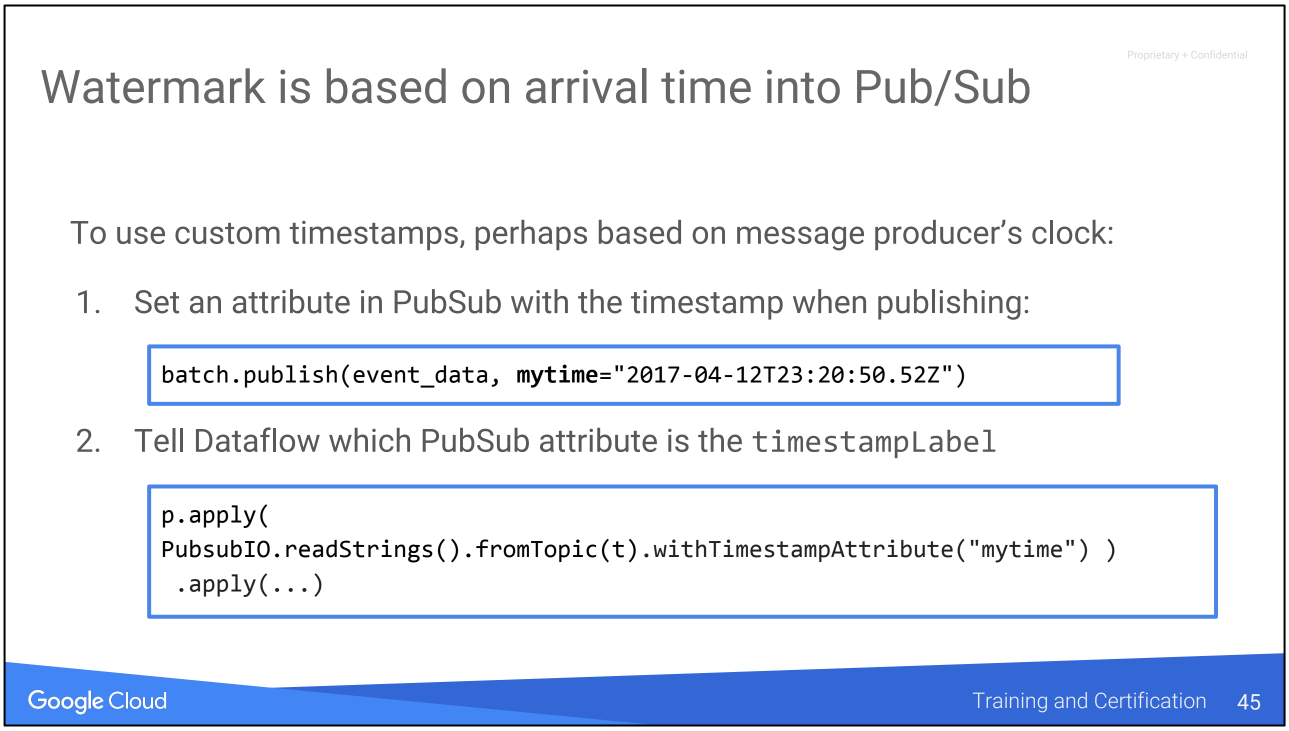


DoFn can get information about window and triggers

```
.apply("...", ParDo.of(new DoFn<Flight, Flight>() {
      @ProcessElement
       public void processElement(ProcessContext c, IntervalWindow window) throws
Exception {
                Instant endOfWindow = window.maxTimestamp();
       }
}))//
```

Notes:

The IntervalWindow will be injected into your callback. Just define it as a parameter. Get more information about the window.



Watermark is based on arrival time into Pub/Sub

To use custom timestamps, perhaps based on message producer's clock:

1. Set an attribute in PubSub with the timestamp when publishing:

```
batch.publish(event_data, mytime="2017-04-12T23:20:50.52Z")
```

2. Tell Dataflow which PubSub attribute is the `timestampLabel`

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

Notes:

In the example, "myname" is the name of the timestampLabel attribute.

The timestamp of the specific message is specified each time you write. It's in RFC 3339 format.

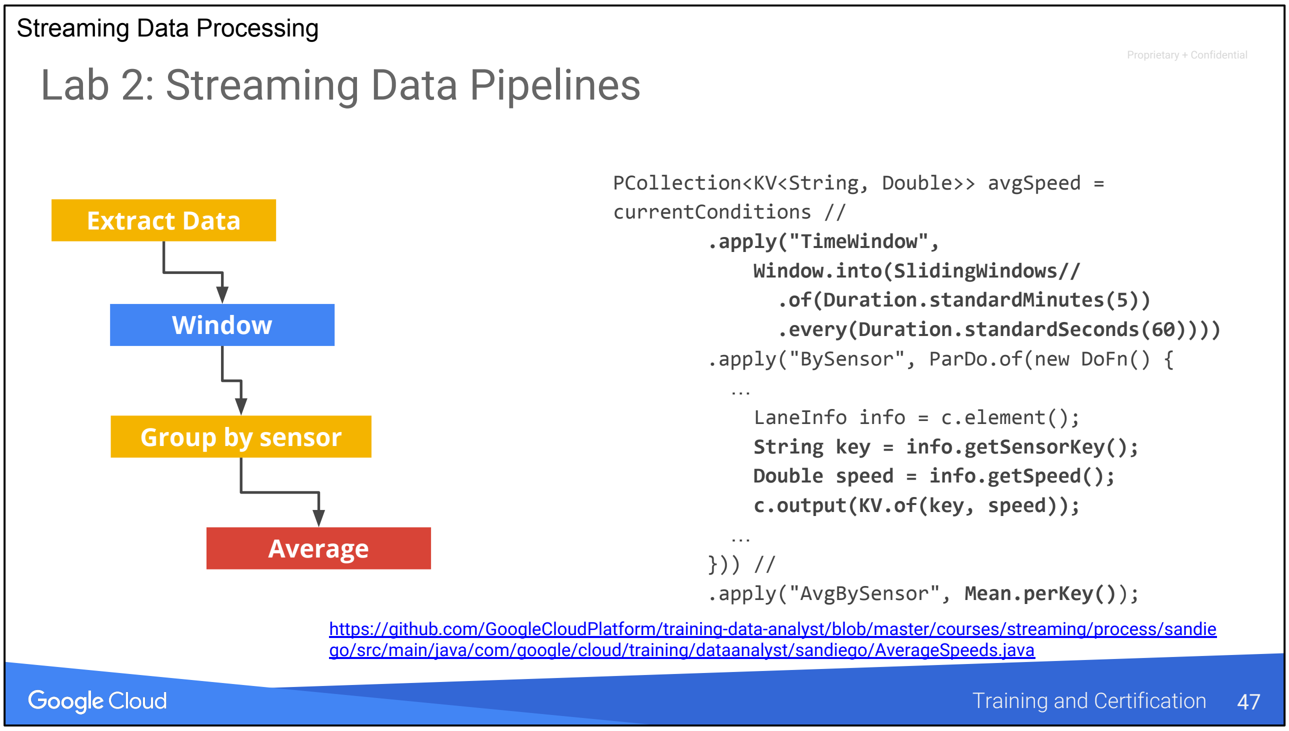
When reading, just specify which PubSub attribute carries the timestamp.

The reason watermark is based on arrival time into Pub/Sub: Dataflow can guarantee whether or not there will be any late records.

Agenda

Lab: Streaming Data Processing

Part 2: Streaming Data Pipelines



Lab 2: Streaming Data Pipelines

Extract Data
Window
Group by sensor
Average

```
PCollection<KV<String, Double>> avgSpeed =
currentConditions //
        .apply("TimeWindow",
            Window.into(SlidingWindows//
              .of(Duration.standardMinutes(5))
              .every(Duration.standardSeconds(60))))
        .apply("BySensor", ParDo.of(new DoFn() {
          ...
            LaneInfo info = c.element();
            String key = info.getSensorKey();
            Double speed = info.getSpeed();
            c.output(KV.of(key, speed));
          ...
        })) //
        .apply("AvgBySensor", Mean.perKey());
```

https://github.com/GoogleCloudPlatform/training-data-analyst/blob/master/courses/streaming/process/sandiego/src/main/java/com/google/cloud/training/dataanalyst/sandiego/AverageSpeeds.java

Notes:

For example, average is calculated for 5 min windows every 1 minute.
It is important to realize that windows are applied only at the time of a group-by-key. So, simply adding a window to a pipeline doesn't cause anything to happen. It's at the group-by-key stage that the window has an impact. Hence, we apply the window when we need to -- when we need to compute the average within the time-window.

cloud.google.com

Images by Connie Zhou