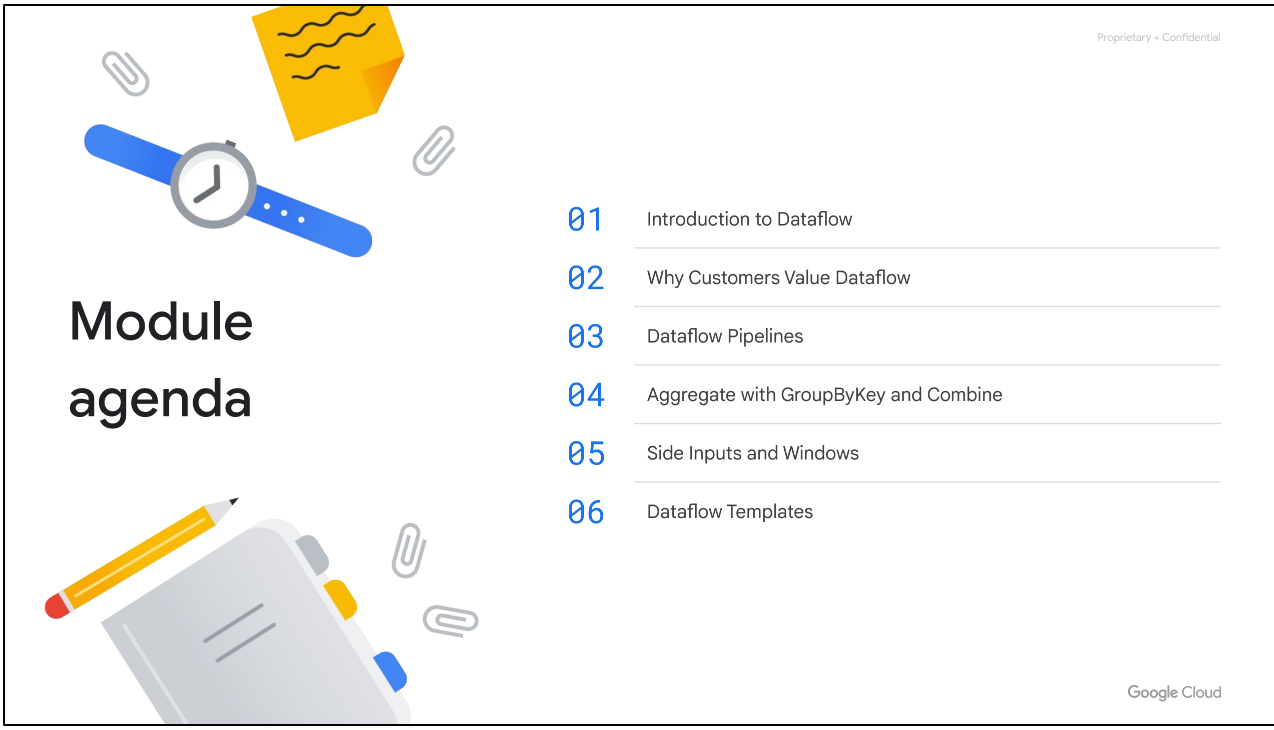
Google Cloud

Serverless Data Processing with Dataflow

Earlier in the course, you saw how to do batch data processing with Dataproc and other methods. Now it's time to introduce you to a key serverless tool that should be in your data engineering toolkit, Dataflow.



This entire module will cover batch Dataflow pipelines and why Dataflow is a commonly used data pipeline tool on Google Cloud.

Not to give away too much of the answer, but you can write the same code to do both batch and streaming pipelines with Dataflow. We'll cover streaming pipelines later.

So the topics we will address are; how to decide between Dataflow and Dataproc, why customers value Dataflow, Dataflow pipelines, and Dataflow templates. Let's get started.

01

Introduction to Dataflow

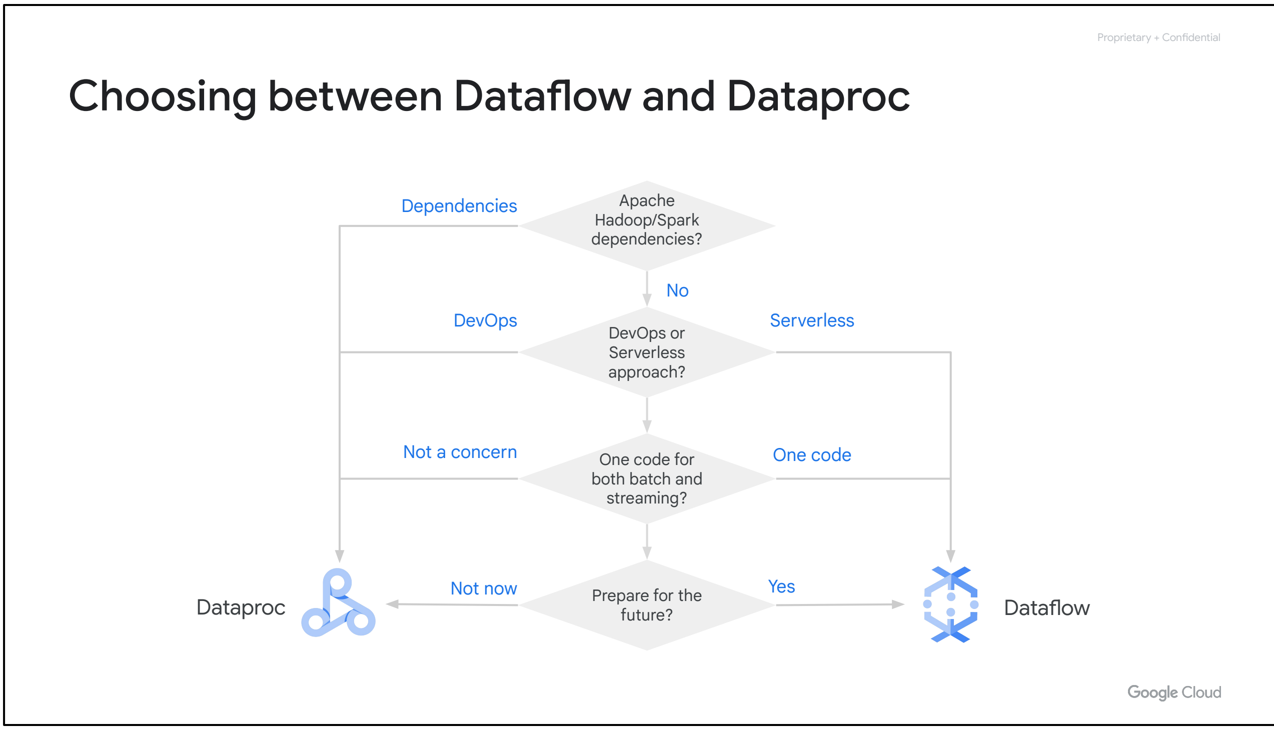
Let’s start by exploring Dataflow in more detail.

Dataflow versus Dataproc

	Dataflow	Dataproc
Recommended for:	New data processing pipelines, unified batch and streaming	Existing Hadoop/Spark applications, machine learning/data science ecosystem, large-batch jobs, preemptible VMs
Fully-managed:	Yes	No
Auto-scaling:	Yes, transform-by-transform (adaptive)	Yes, based on cluster utilization (reactive)
Expertise:	Apache Beam	Hadoop, Hive, Pig, Apache Big Data ecosystem, Spark, Flink, Presto, Druid

The reason Dataflow is the preferred way to do data processing on Google Cloud is that Dataflow is serverless. You don’t have to manage clusters at all. Unlike with Dataproc, the autoscaling in Dataflow scales step-by-step. It’s very fine-grained. Plus, as we will see in the next course, Dataflow allows you to use the same code for both batch and stream. This is becoming increasingly important.

When building a new data processing pipeline, we recommend that you use Dataflow. If, on the other hand, you have existing pipelines written using Hadoop technologies, it may not be worth it to rewrite everything. Migrate it over to Google Cloud using Dataproc, and then modernize it as necessary.



As a data engineer, we recommend that you learn both Dataflow and Dataproc and make the choice based on what's best for a specific use case.

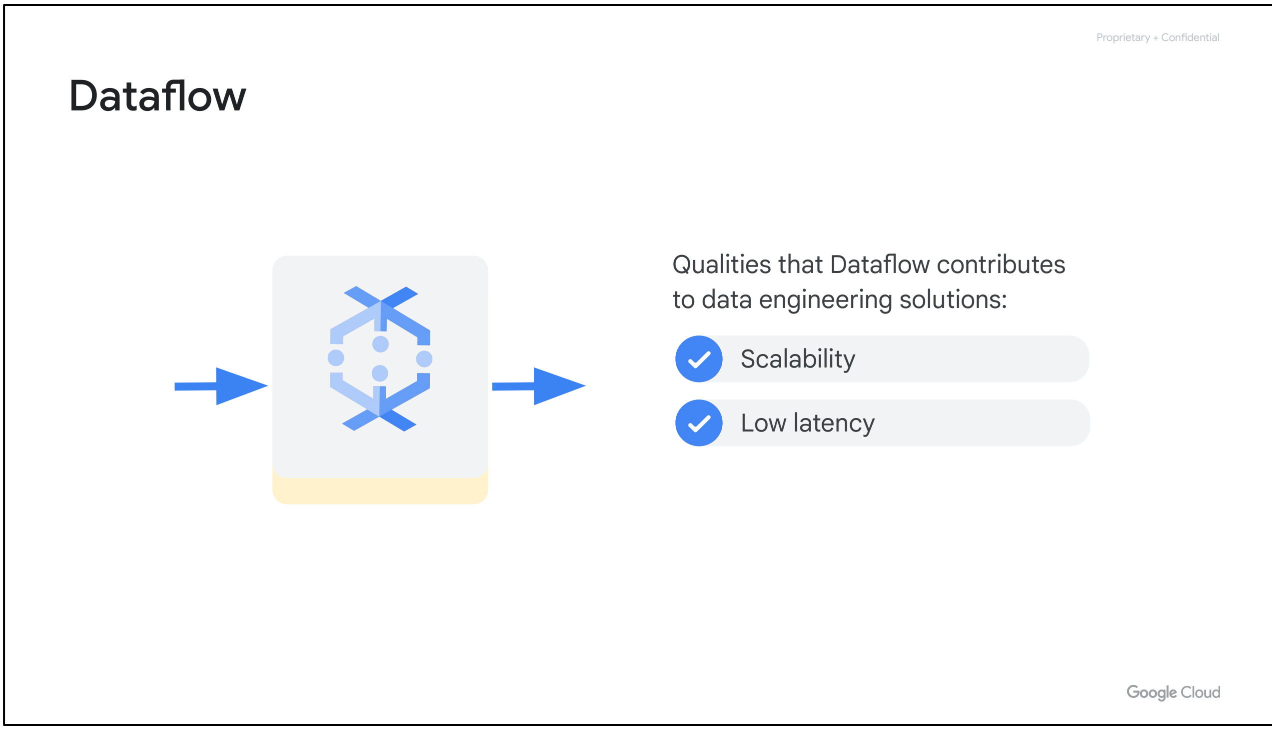
If the project has existing Hadoop or Spark dependencies, use Dataproc.

Please keep in mind that there are many subjective issues when making this decision and that no simple guide will fit every use case.

Sometimes, the production team might be much more comfortable with a DevOps approach where they provision machines than with a serverless approach. In that case, too, you might pick Dataproc.

If you don't care about streaming and your primary goal is to move existing workloads, then Dataproc would be fine.

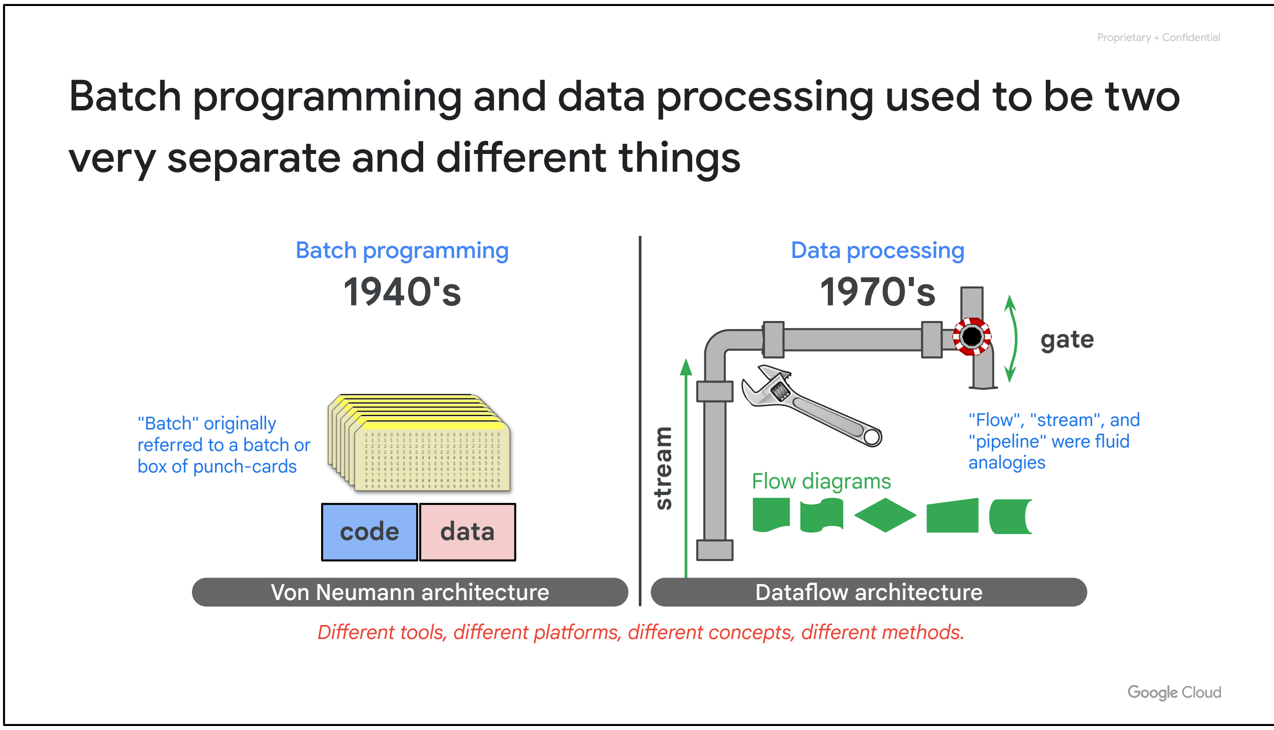
Dataflow, however, is our recommended approach for building pipelines.



Dataflow provides a serverless way to execute pipelines on batch and streaming data.

It's scalable -- to process more data, Dataflow will scale out to more machines. It will do this transparently.

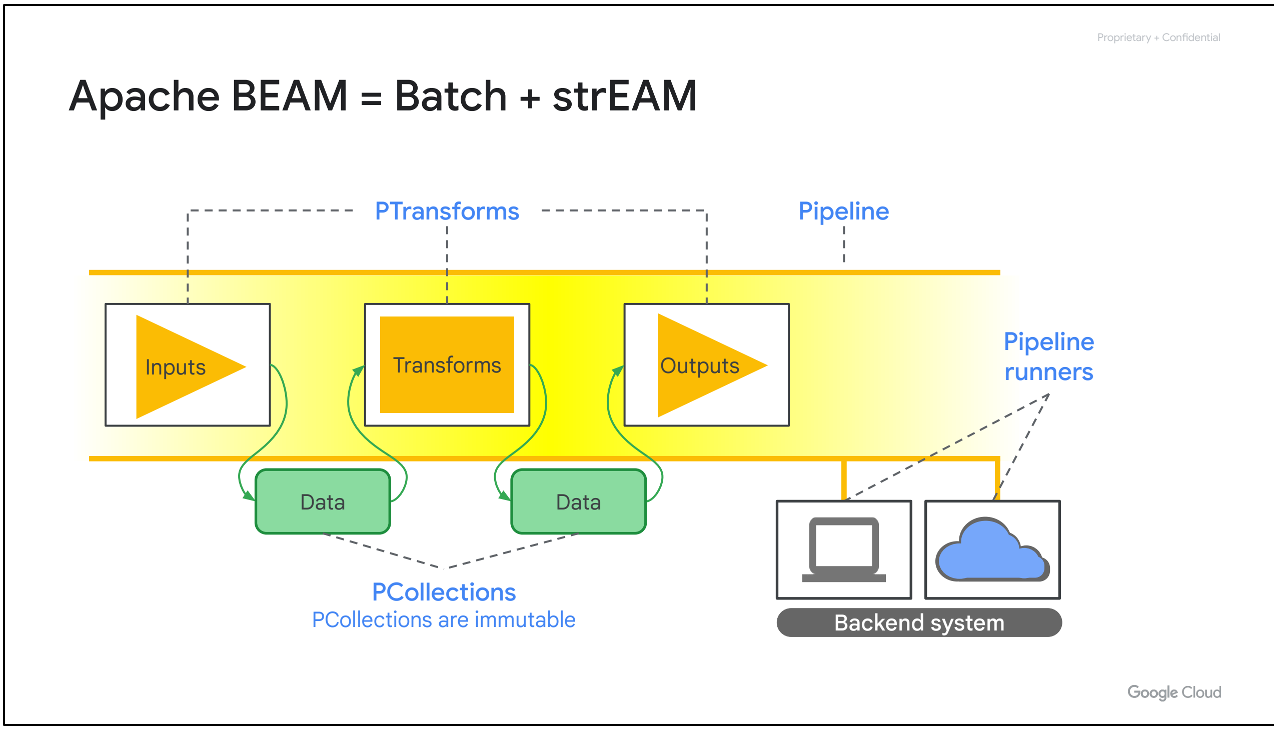
The stream processing capability also makes it low latency. You can process the data as it comes in.



This ability to process batch and stream with the same code is rather unique. For a long time, batch programming and data processing used to be two very separate and different things.

Batch programming dates to the 1940s, and the early days of computing where it was realized that you can think of two separate concepts -- code and data. Use code to process data. Of course, both of these were on punch cards, so that's what you were processing -- a box of punch-cards, called a batch. It was a job that started and ended when the data was fully processed.

Stream processing, on the other hand, is more fluid. It arose in the 1970s with the idea of data processing being something that is ongoing. The idea is that data keeps coming in, and you process the data. The processing itself tended to be done in micro-batches.



The genius of Beam is that it provides abstractions that unify traditional batch programming concepts and traditional data processing concepts. Unifying programming and processing is a big innovation in data engineering.

The four main concepts are PTransforms, PCollections, Pipelines, and Pipeline runners.

- A **pipeline** identifies the data to be processed and the actions to be taken on the data. The data is held in a distributed data abstraction called a PCollection. The PCollection is immutable. Any change that happens in a pipeline ingests one PCollection and creates a new one. It does not change the incoming PCollection.

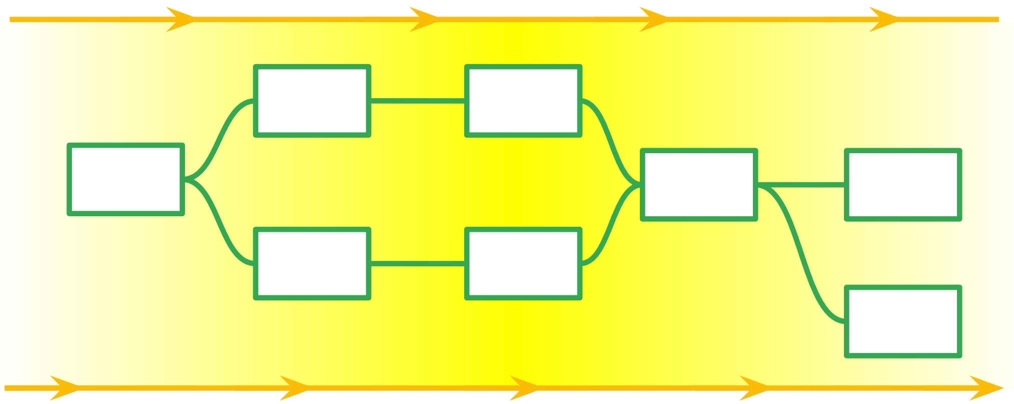
- The actions or code is contained in an abstraction called a **PTransform**. The PTransform handles input, transformation, and output of the data.

- The data in a **PCollection** is passed along a graph from one PTransform to another.

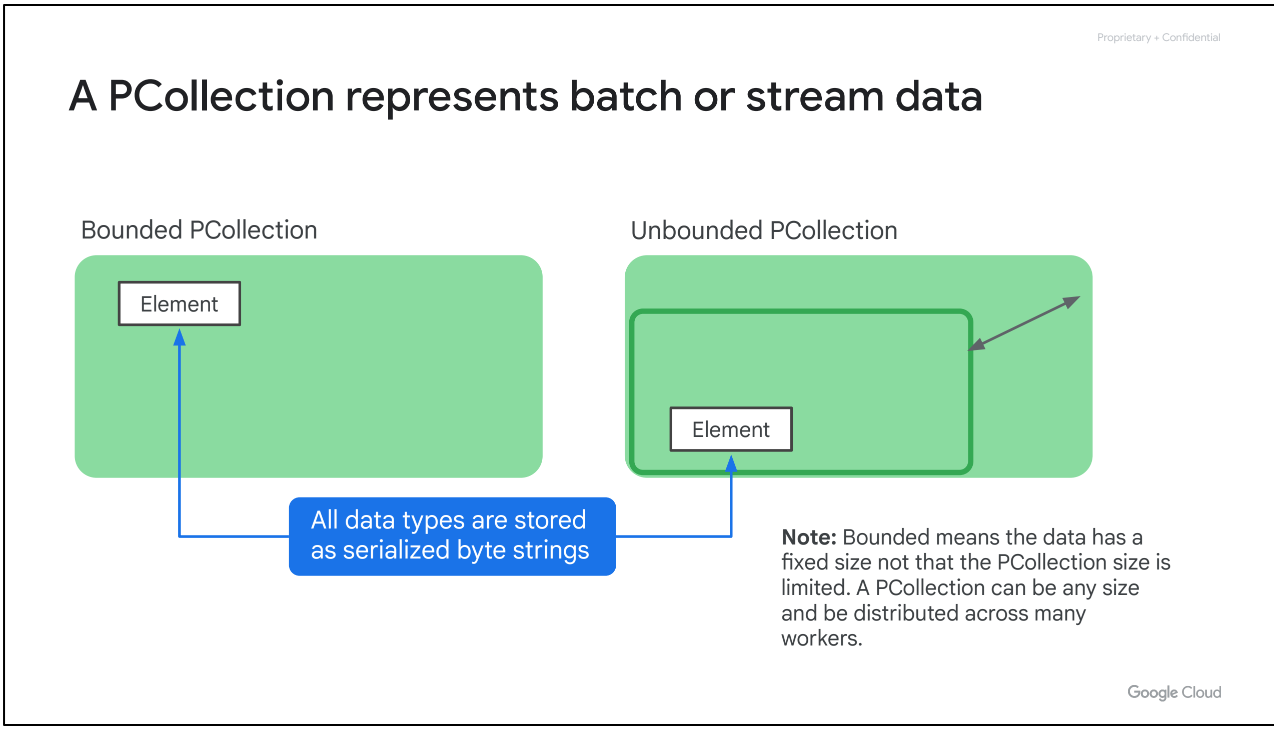
- **Pipeline runners** are analogous to container hosts, such as Google Kubernetes Engine. The identical pipeline can be run on a local computer, data center VM, or on a service such as Dataflow in the cloud. The only difference is scale and access to platform-specific services. The services the runner uses to execute the code is called a backend system.

Immutable data is one of the key differences between batch programming and data processing. Immutable data, where each transform results in a new "copy" means there is no need to coordinate access control or sharing of the original ingest data. So it enables (or at least simplifies) distributed processing.

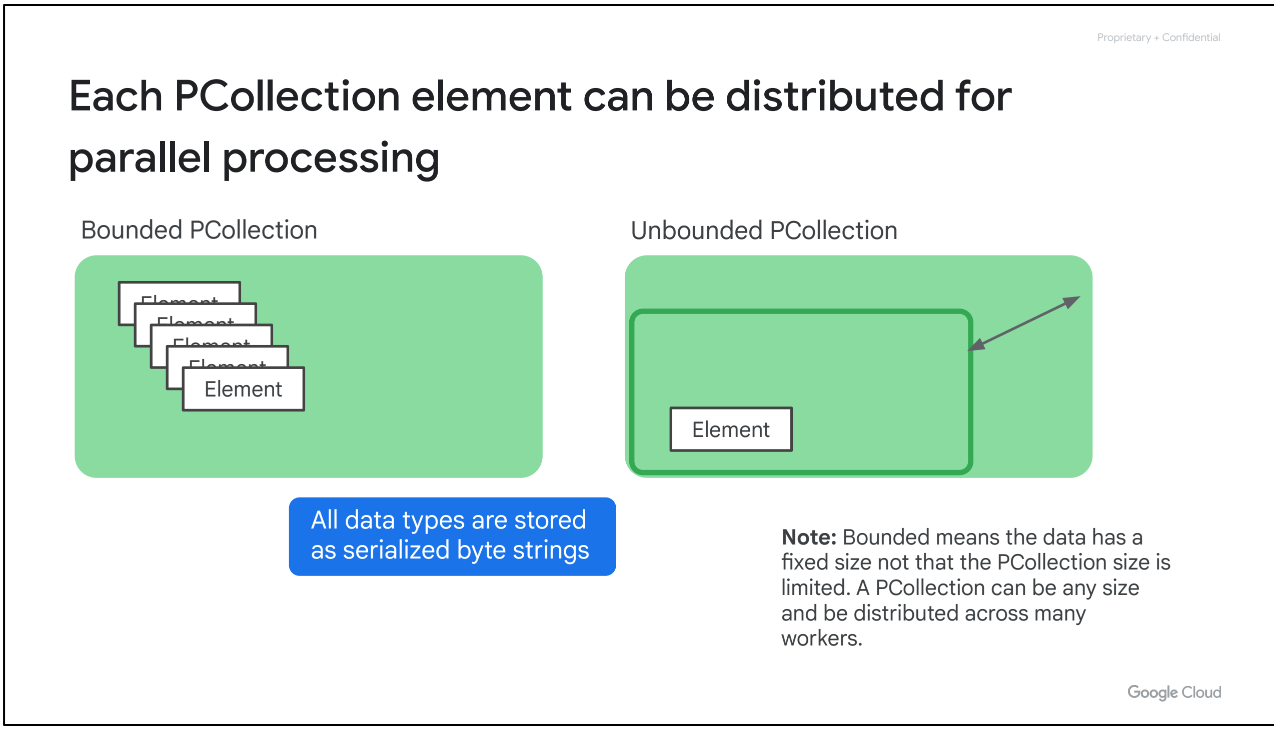
A Dataflow pipeline is a directed graph of steps



The shape of a pipeline is not actually just a single linear progression, but rather, a directed graph with branches and aggregations. For historical reasons we refer to it as a pipeline, but a datagraph or dataflow might be a more accurate description.



A PCollection represents both streaming data and batch data.There is no size limit to a PCollection. Streaming data is an unbounded PCollection that doesn't end.

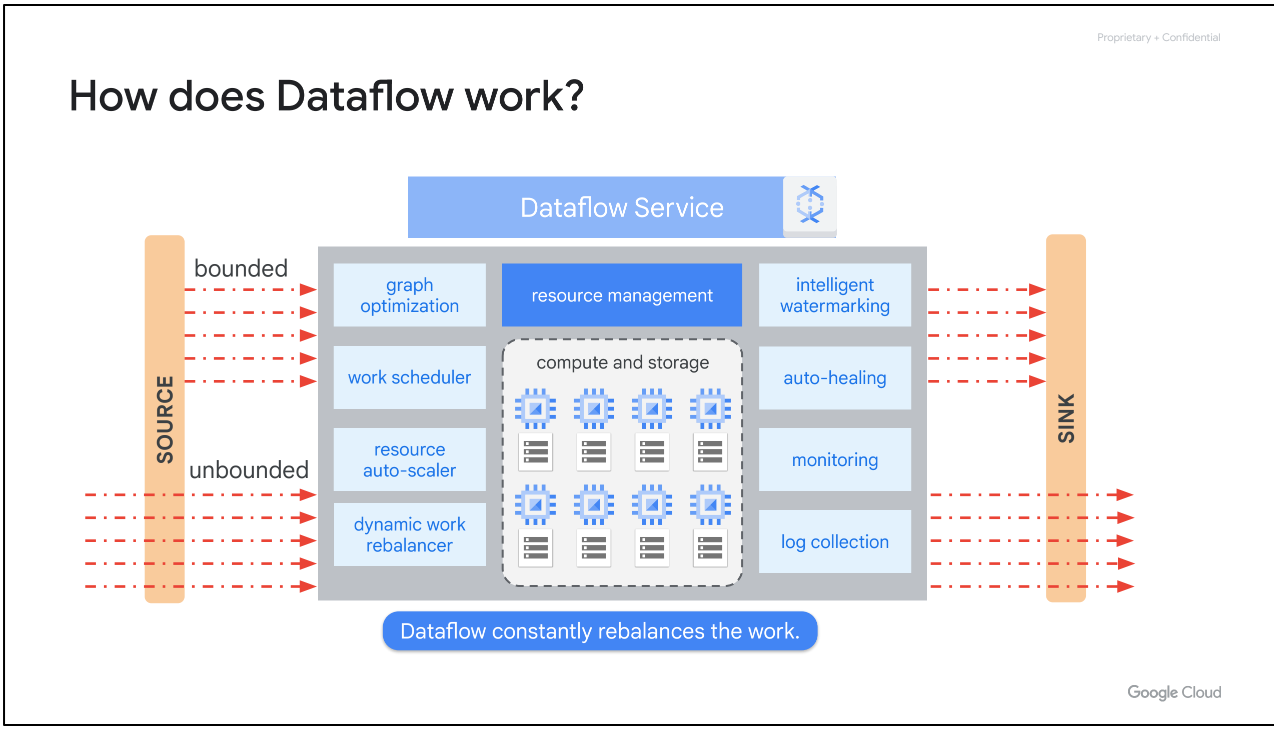


Each element inside a PCollection can be individually accessed and processed. This is how distributed processing of the PCollection is implemented. So you define the pipeline and the transforms on the PCollection, and the runner handles implementing the transformations on each element, distributing the work as needed for scale and with available resources.

Once an element is created in a PCollection it is immutable. So it can never be changed or deleted. Elements represent different data types.

In traditional programs, a data type is stored in memory with a format that favors processing. Integers in memory are different from characters which are different from strings and compound data types. In a PCollection all data types are stored in a serialized state as byte strings. This way there is no need to serialize data prior to network transfer and deserialize it when it is received. Instead, the data moves through the system in a serialized state and is only deserialized when necessary for the actions of a PTransform.

So we’ve discussed what Dataflow is. But why do data engineers value Dataflow over other alternatives for data processing?



To understand that, it helps to understand a bit about how Dataflow works. Dataflow provides an efficient execution mechanism for Apache Beam.

The Beam pipeline specifies WHAT has to be done. The Dataflow services chooses HOW to run the pipeline.

The pipeline typically consists of reading data from one or more sources, applying processing to the data, and writing it to one or more sinks. In order to execute the pipeline, the Dataflow service first optimizes the graph by, for example, fusing transforms together.

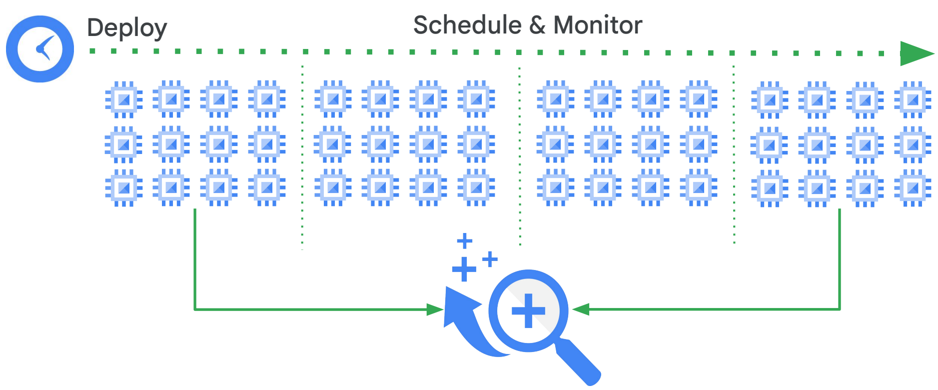
It then breaks the jobs into units of work and schedules them to various workers.

One of the great things about Dataflow is that the optimization is always ongoing. Units of work are continually rebalanced.

Resources -- both compute and storage -- are 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.

The watermarking handles late arrivals of data, and comes with restarts, monitoring, and logging. No more waiting for other jobs to finish. No more preemptive scheduling. Dataflow provides a reliable, serverless, job-specific way to process your data.

Why customers value Dataflow: Fully-managed and auto-configured

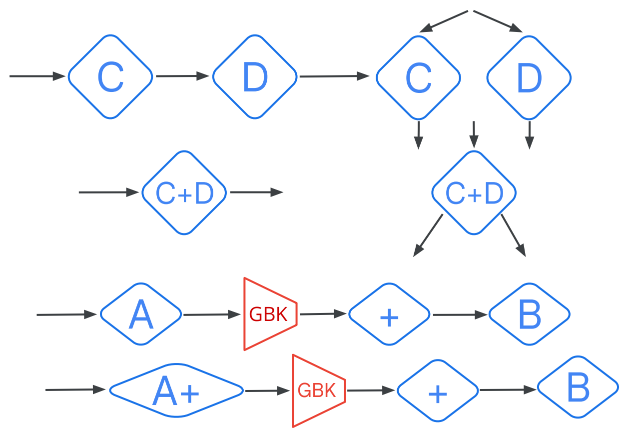


To summarize, the advantages of Dataflow are:

First, Dataflow is fully-managed and auto-configured. Just deploy your pipeline.

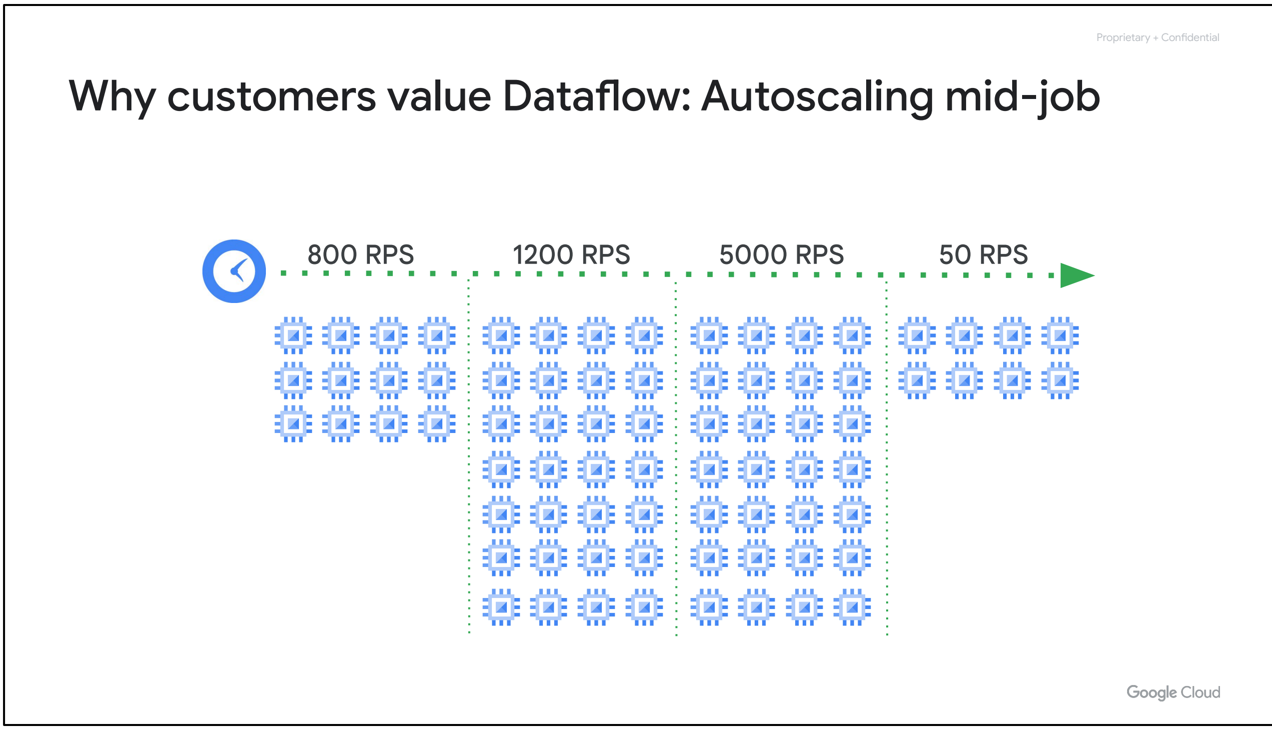
Proprietary + Confidential

Why customers value Dataflow: Graph is optimized for best execution path

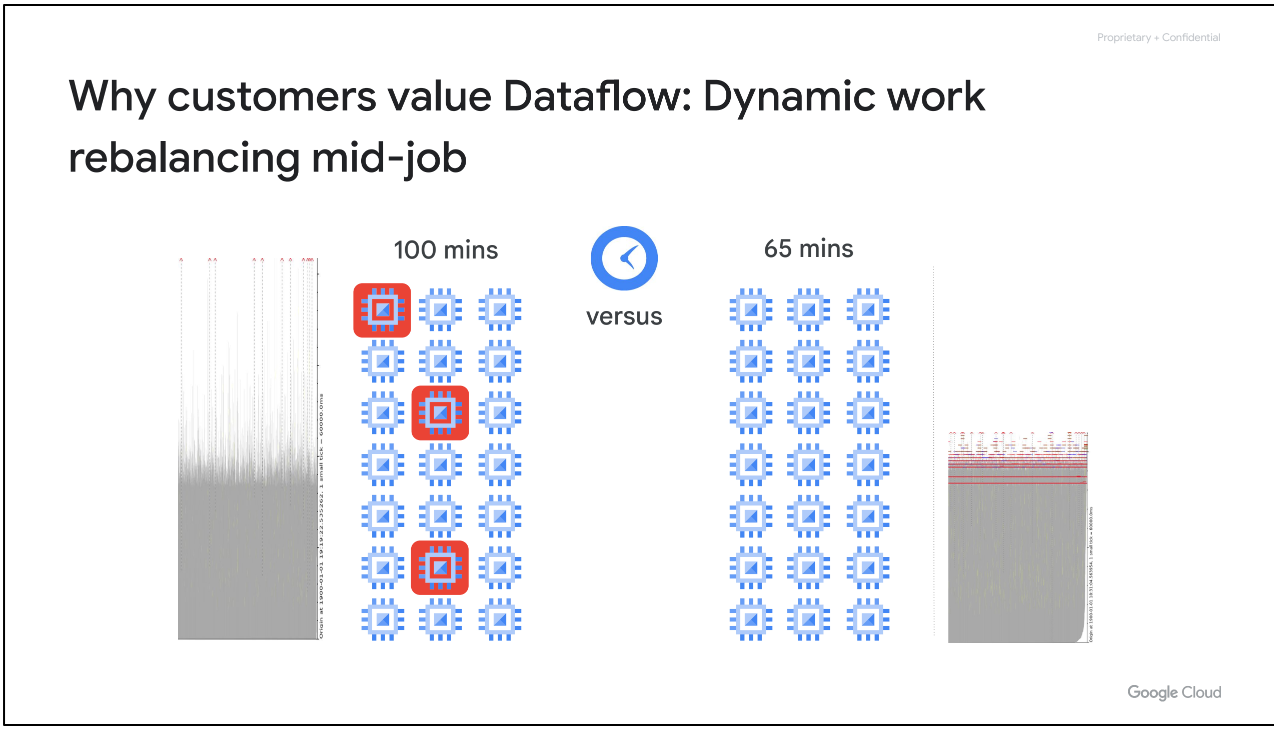


Google Cloud

Second, Dataflow doesn't just execute the Apache Beam transforms as-is. It optimizes the graph, fusing operations as we see with C and D. Also, it doesn't wait for a previous step to finish before starting a new step. We see this with A and the Group-by-key.



Third, autoscaling happens step-by-step, in the middle of a job. As the jobs needs more resources, it receives more resources. You don't have to manually scale resources to match job needs.



If some machines have finished their tasks, and others are still going on, the tasks queued up for the busy ones are rebalanced out to the idle machines. This way, the overall job finishes faster.

Dynamic work rebalancing in mid-job removes the need to spend operational or analyst resource time hunting down hotkeys.

Proprietary + Confidential

Why customers value Dataflow: Strong streaming semantics

- Exactly once aggregations
- Rich time tracking
- Good integration with other Google Cloud services

Google Cloud

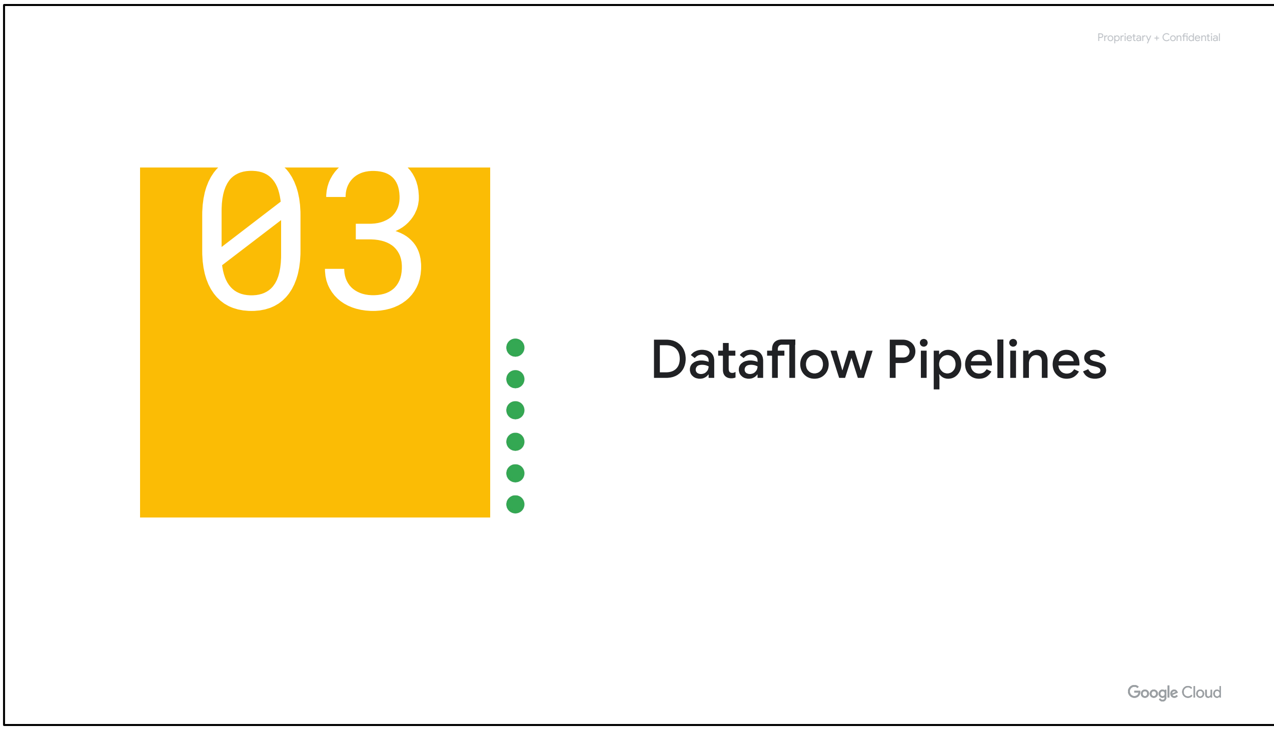
All this happens while maintaining strong streaming semantics.

Aggregations, like sums and counts, are correct even if the input source sends duplicate records.

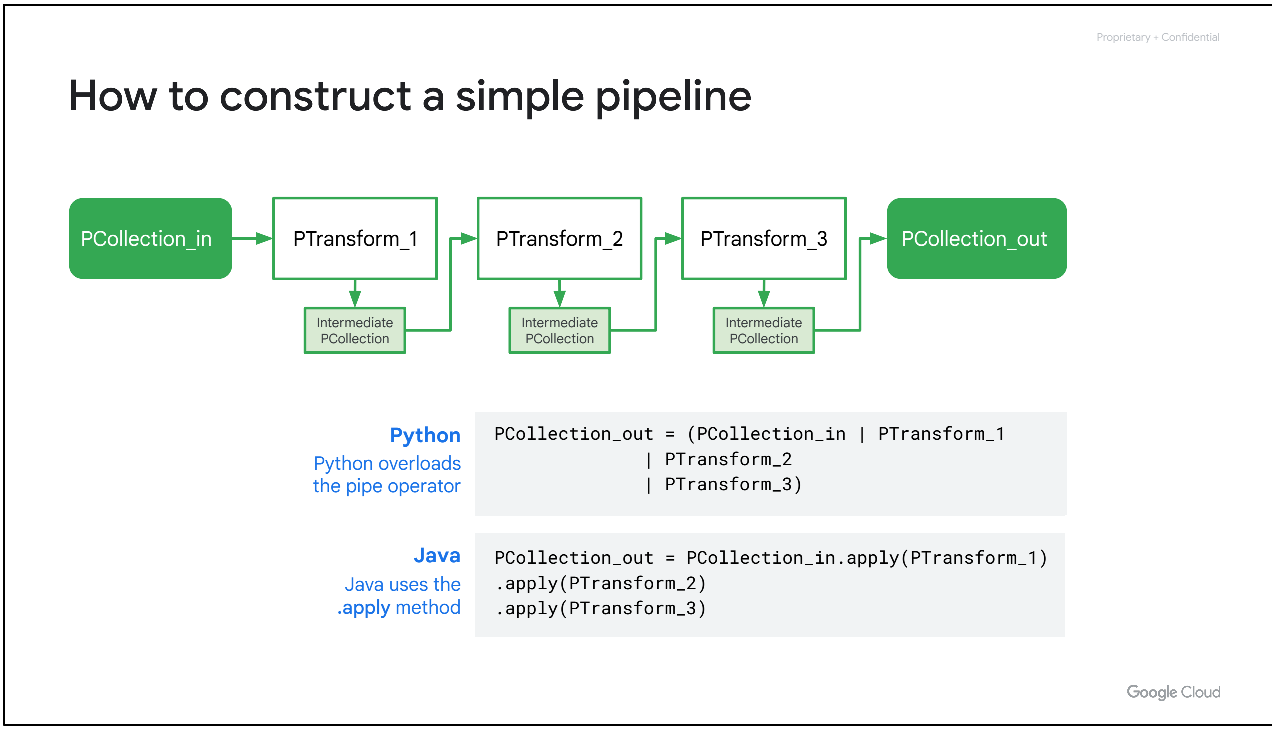
Dataflow is able to handle late arriving records.

Finally, Dataflow functions as the glue that ties together many of the services on Google Cloud. Do you need to read from BigQuery and write to Bigtable? Use Dataflow.

Do you need to read from Pub/Sub and write to Cloud SQL? Use Dataflow.



Let’s look in greater detail at an example Dataflow pipeline.



Here's how to construct a simple pipeline where you have an input PCollection and pass it through three PTransforms and get an output PCollection.

The syntax is shown in Python. You have the input, the pipe symbol, the first PTransform, the pipe symbol, the second PTransform, etc.

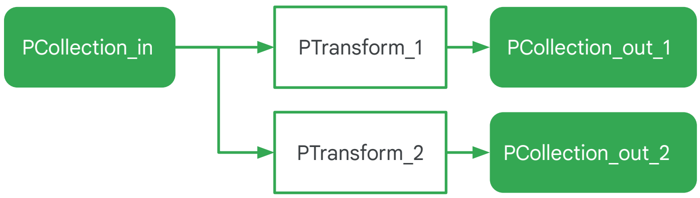
The pipe operator essentially applies the transform to the input PCollection and sends out an output PCollection.

The first three times, we don't give the output a name, simply pass it on the next step.

The output of PTransform 3, though, we save into a PCollection variable named PCollection out.

In Java, it is the same thing except that, instead of the pipe symbol, we use the apply method.

How to construct a branching pipeline



Python

```
PCollection_out_1 = PCollection_in | PTransform_1
PCollection_out_2 = PCollection_in | PTransform_2
```

Java

```
PCollection_out_1 = PCollection_in.apply(PTransform_1)
PCollection_out_2 = PCollection_in.apply(PTransform_2)
```


If you want to do branching, just send the same PCollection through two different transforms.

Give the output PCollection variable in each case a name. Then, you can use it in the remainder of your program. Here, for example, we take the PCollection_in and pass the collection first through both PTransform 1, then through PTransform 2.

The result in the first case, we store as PCollection out 1. In the second case, we store it as PCollection out 2.

Proprietary + Confidential

A pipeline is a directed graph of steps

Python

```
import apache_beam as beam

if __name__ == '__main__':

   with beam.Pipeline(argv=sys.argv) as p:

        (p
           | beam.io.ReadFromText('gs://...')
           | beam.FlatMap(count_words)
           | beam.io.WriteToText('gs://...')
        )

   # end of with-clause: runs, stops the pipeline
```

Create a pipeline parameterized by command line flags

Read input

Apply transform

Write output

Google Cloud

What we showed you so far was the middle part of a pipeline. You already had a PCollection and you applied a bunch of transforms and you end up with a PCollection.

But where does the pipeline start? How do you get the first PCollection? You get it from a source.

What does a pipeline do with the final PCollection? Typically, it writes out to a sink.

That's what we are showing here. This is Python.

We create a PCollection by taking the pipeline object P and passing it over a text file in Google Cloud Storage. That's the ReadFromText line.

Then, we apply the PTransform called FlatMap to the lines read from the text file. What FlatMap does is that it applies a function to each row of the input and concatenates all the outputs. When the function is applied to a row, it may return zero or more elements that g o to the output PCollection.

The function in this case is the function called count words. It takes a line of text and returns an integer.

The output PCollection then consists of a set of integers. These integers are written to a text file in Cloud Storage.

Because the pipeline was created in a WITH clause, and because this is not a streaming pipeline, exiting the WITH clause automatically stops the pipeline.

Run a pipeline on Dataflow

Python

```
import apache_beam as beam

options = {'project': <project>,
           'runner': 'DataflowRunner',
           'region': <region>,
           'setup_file': <setup.py file>}
pipeline_options = beam.pipeline.PipelineOptions(flags=[],
**options)
pipeline = beam.Pipeline(options = pipeline_options)
```

Where to run

This creates the pipeline

Once you have written the pipeline, it is time to run it. Executing the Python program on the previous slide will run the program. By default, the program is run using the DefaultRunner, which runs on the same machine where the Python program was executed.

When you create the pipeline, you can pass in a set of options. One of these options is the runner. Specify that as Dataflow to have the pipeline run on Google Cloud.

This example contains hard-coded variables which in most cases is not a preferred practice for programming at scale.

Pipeline Execution using DataflowRunner

Run local

```
python ./grep.py
```

Run on cloud

```
python ./grep.py \
      --project=$PROJECT \
      --job_name=myjob \
      --staging_location=gs://$BUCKET/staging/ \
      --temp_location=gs://$BUCKET/tmp/ \
      --runner=DataflowRunner
```


Of course, normally, you will set up command-line parameters to transparently switch between local and cloud.

Simply running main runs the pipeline locally. To run on cloud, specify cloud parameters.

Proprietary + Confidential

Read data from local file system, Cloud Storage, Pub/Sub, BigQuery, ...

```
with beam.Pipeline(options=pipeline_options) as p:
```

Read from Cloud Storage (returns a string)

```
lines = p | beam.io.ReadFromText("gs://.../input-*.csv.gz")
```

Read from Pub/Sub (returns a string)

```
lines = p | beam.io.ReadStringsFromPubSub(topic=known_args.input_topic)
```

Read from BigQuery (returns rows)

```
query = "SELECT x, y, z FROM `project.dataset.tablename`"
BQ_source = beam.io.BigQuerySource(query = <query>, use_standard_sql=True)
BQ_data = pipeline | beam.io.Read(BQ_source)
```

Setup

Read

Google Cloud

To design pipelines you need to know how each step works on the individual data elements contained inside of a PCollection. Let's start with the input and outputs of the pipeline.

First, we set up our Beam pipeline with beam.pipeline and pass through any options. Here we'll call the pipeline P.

Now it's time to get some data as input. If we wanted to read a series of CSV files in Cloud Storage, we could use beam.io.ReadFromText and simply parse in the Cloud Storage bucket and filename. Note the use of an asterisk wildcard can handle multiple files.

If we wanted to read instead from a Pub/Sub topic, you would still use beam.io but instead it's ReadStringsFromPubSub and you'd have to parse in the topic name.

What about if you wanted to read in data that's already in BigQuery? Here's how that would look. You'd prepare your SQL query and specify BigQuery as your input source and then parse in the query and source as a read function to Dataflow. These are just a few of the data sources from which Dataflow can read. But now what about writing to sinks?

Write to a BigQuery table

Establish reference to BigQuery table

```
from apache_beam.io.gcp.internal.clients import bigquery

table_spec = bigquery.TableReference(
    projectId='clouddataflow-readonly',
    datasetId='samples',
    tableId='weather_stations')
```

Write to BigQuery table

```
p | beam.io.WriteToBigQuery(
    table_spec,
    schema=table_schema,
    write_disposition=beam.io.BigQueryDisposition.WRITE_TRUNCATE,
    create_disposition=beam.io.BigQueryDisposition.CREATE_IF_NEEDED)
```


Take the BigQuery example but as a data sink this time. With Dataflow you can write to a BigQuery table as you can see here. First, you establish the reference to the BigQuery table with what BigQuery expects, your project ID, dataset ID and table name.

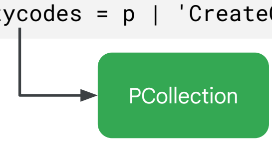
Then you use beam.io.WriteToBigQuery as a sink to your pipeline. Note that we are using the normal BigQuery options here for rate disposition. Here we're truncating the table if it exists, meaning to drop data rows. If the table doesn't exist we can create it if needed. Naturally, this is a batch pipeline if we're truncating the table with each load.

Create a PCollection from in-memory data

Python

```python
city_zip_list = [
    ('Lexington', '40513'),
    ('Nashville', '37027'),
    ('Lexington', '40502'),
    ('Seattle', '98125'),
    ('Mountain View', '94041'),
    ('Seattle', '98133'),
    ('Lexington', '40591'),
    ('Mountain View', '94085'),
]
citycodes = p | 'CreateCityCodes' >> beam.Create(city_zip_list)
```

This is the display name of the pipeline step



You can also create a PCollection in memory without reading from a particular source. Why might you do this? If you have a small dataset like a lookup table or a hard-coded list, you could create the PCollection yourself as you can see here. Then we can call a pipeline step on this new PCollection just as if we sourced it from somewhere else.

Map and FlatMap

Use Map for 1:1 relationship between input and output

```
'WordLengths' >> beam.Map(word, len(word))
```

Map (fn) uses a callable fn to do a one-to-one transformation.

Use FlatMap for non 1:1 relationships, usually with a generator

```
def my_grep(line, term):
   if term in line:
      yield line     Generator

'Grep' >> beam.FlatMap(my_grep(line, searchTerm))
```

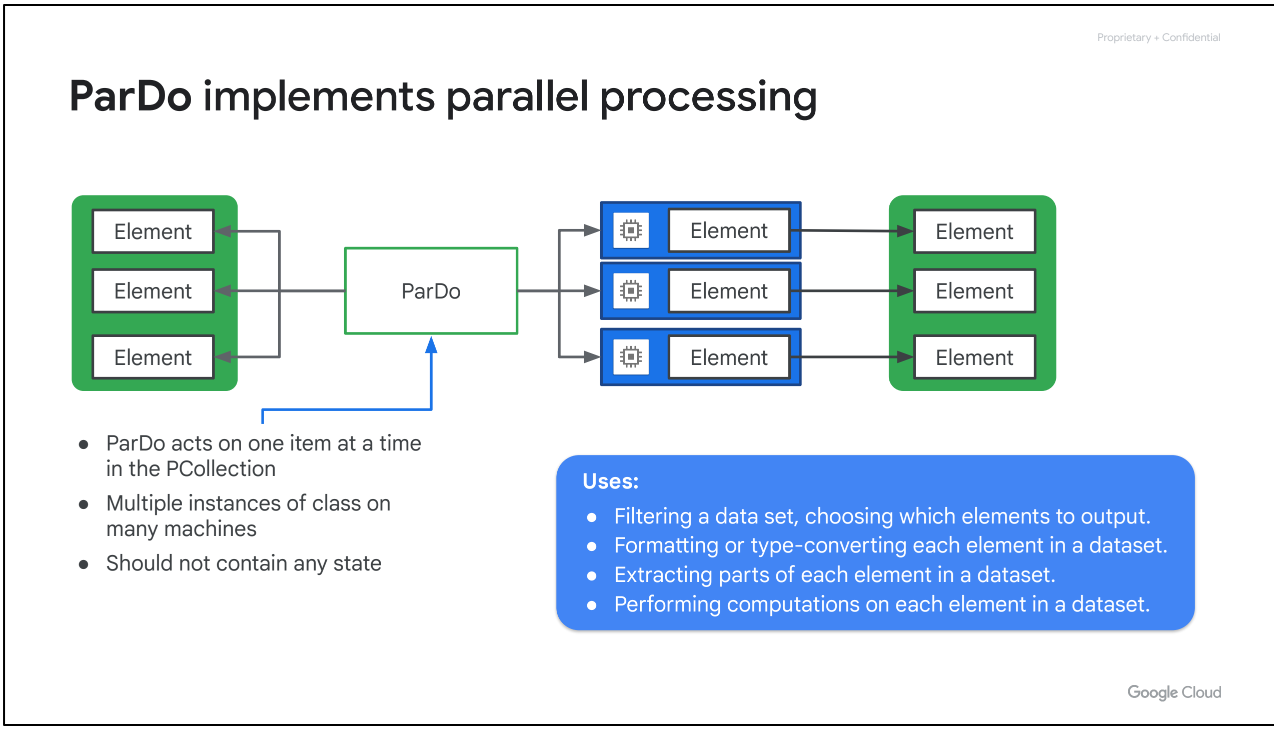
FlatMap is similar to Map, but fn returns an iterable of zero or more elements.
The iterables are flattened into one PCollection.

Now that we have looked at how to get the data in, let's look at how we transform each data element in the PCollection with PTransforms. The first step of any map produced process is the map phase where you're doing something in parallel.

In the word length example, there is one length output for each word input. So the word dog would map to three for length. In the bottom graph example the function my_grep returns each instance of the term it's searching for in the line.

There may be multiple instances of the term in a single line a one-to-many relationship. In this case, you may want my-grep to return the next instance each time it's called, which is why the function has been implemented with a generator using yields. The yield command has the effect of preserving the seed of the function so that the next time it's called, it can continue from where it left off. FlatMap has the effect of iterating over one-to-many relationship.

The map example returns a key value pair. In Python, this is simply a two-tuple for each word. The FlatMap example yields the line only for lines that contain the search term.



ParDo is a common intermediate step in a pipeline. You might use it to extract certain fields from a set of raw input records, or convert raw input into a different format; you might also use ParDo to convert processed data into an output format, like table rows for BigQuery or strings for printing.

- You can use ParDo to consider each element in a PCollection and either output that element to a new collection, or discard it.
- If your input PCollection contains elements that are of a different type or format than you want, you can use ParDo to perform a conversion on each element and output the result to a new PCollection.
- If you have a PCollection of records with multiple fields, for example, you can use a ParDo to parse out just the fields you want to consider into a new PCollection.
- You can use ParDo to perform simple or complex computations on every element, or certain elements, of a PCollection and output the results as a new PCollection.

ParDo requires code passed as a DoFn object

Python

```
words = ...

class ComputeWordLengthFn(beam.DoFn):
  def process(self, element):
    return [len(element)]

word_lengths = words | beam.ParDo(ComputeWordLengthFn())
```

DoFN

ParDo

The **input** is a PCollection of strings.

The DoFn to perform on each element in the input PCollection.

The **output** is a PCollection of integers.

Apply a ParDo to the PCollection "words" to compute lengths for each word.

When you apply a ParDo transform, you need to provide code in the form of a DoFn object. A Do function is a beam SDK class that defines a distributed processing function. Your DoFn code must be fully serializable, item potent, and thread safe. In this example, we're just counting the number of words in a line, and returning the length of the line. Transformations are always going to work on one element at a time here.

Proprietary + Confidential

ParDo method can emit multiple variables

```
results = (words | beam.ParDo(ProcessWords(),
   cutoff_length=2, marker='x')
   .with_outputs('above_cutoff_lengths', 'marked strings', main='below_cutoff_strings'))

below  = results.below_cutoff_strings
above  = results.above_cutoff_lengths
marked = results['marked strings']
```

Google Cloud

Here we have an example from Python which can return multiple variables. In this example, we have below and above some cut off in our data elements. And return two different types below and above two different variables by referencing these properties of the results.

Proprietary + Confidential

Lab Intro

Serverless Data Analysis with Dataflow: A Simple Dataflow Pipeline (Python/Java)

Google Cloud

Next, let's do a lab; a simple Dataflow pipeline to perform serverless data analysis using Python or Java. You can select which version of the lab you'd like to do.

Now, let's look at more capabilities of the Dataflow model.

GroupByKey explicitly shuffles key-values pairs

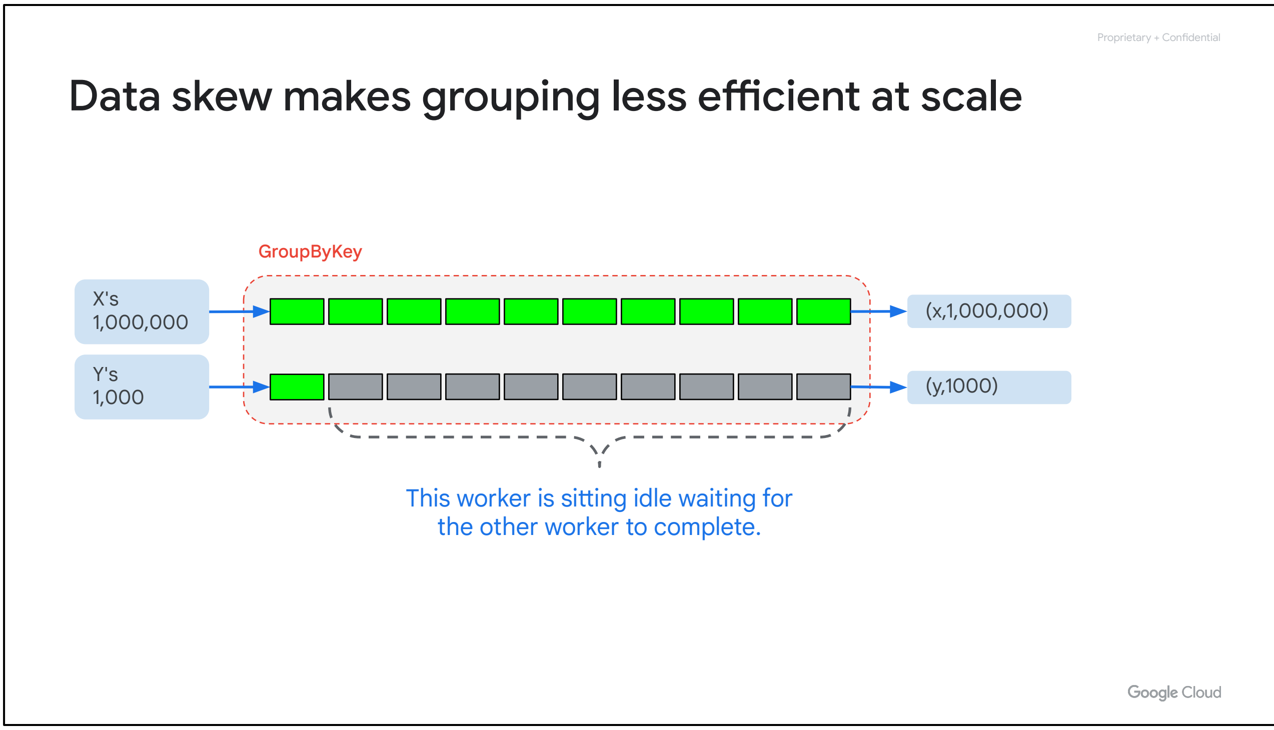
```
cityAndZipcodes = p | beam.Map(fields[0], fields[1])

grouped = cityAndZipCodes | beam.GroupByKey()
```

```
Lexington, 40513
Nashville, 37027
Lexington, 40502
Seattle, 98125
Mountain View, 94041
Seattle, 98133
Lexington, 40591
Mountain View, 94085
```

```
Lexington, [40513, 40502, 40591]
Nashville, [37027]
Seattle, [98125, 98133]
Mountain View, [94041, 94085]
```


What do you do after the map phase? The unnamed phase is the shuffle phase where you group together like keys. This works on a PCollection of key-value pairs or two elements tuples. Groups by common key and returns a single key-value pair where the value is actually a group of values.

The idea here is that we want to find all the zip codes associated with the city. For example, New York is a city and it may have one-zero-zero-zero-one and one-zero-zero-zero-two zip codes. You could first create a key-value pair and a ParDo and then group by the key. The resulting key-value pairs are simply two tuples.

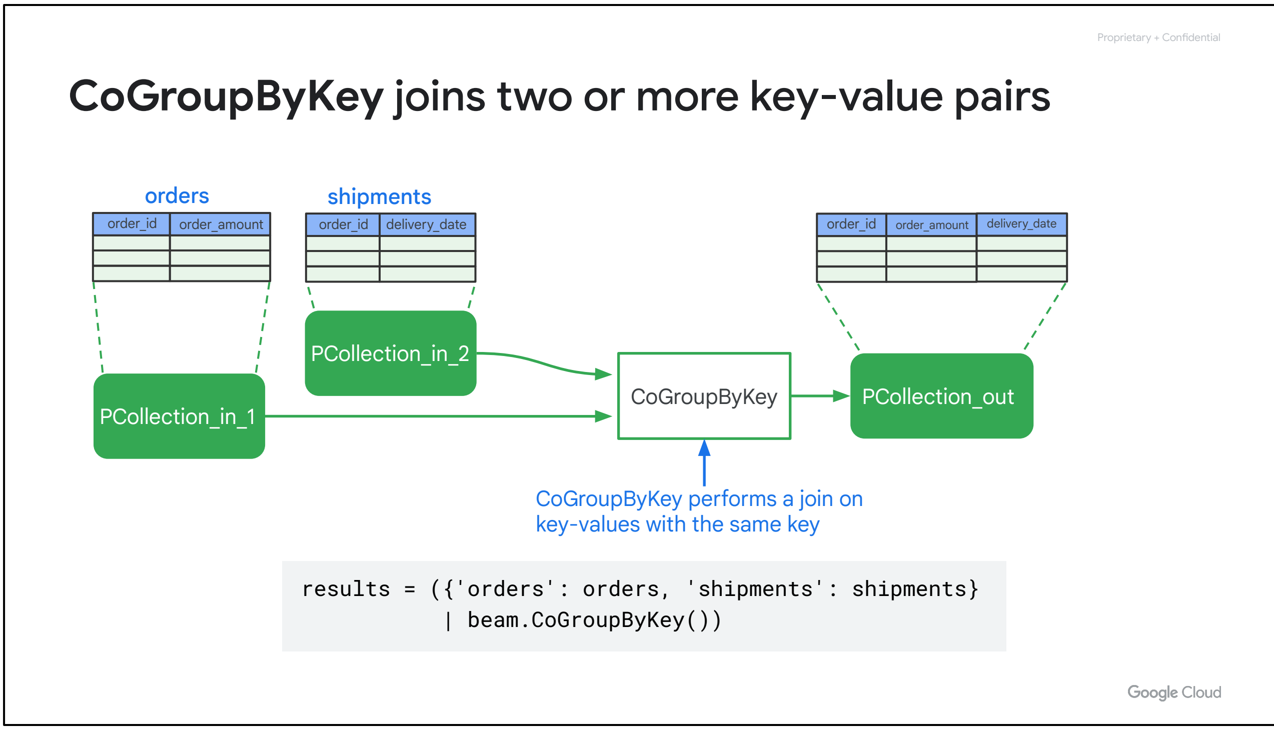


We do have to be aware of data skew on we're doing this. When the same example is scaled up in the presence of skewed data, the situation becomes much worse. Let's say that you're doing your **GroupByKey**, but your group has 1 million items in it. One million is not too big of a deal on modern hardware, but with one billion you're forcing all of those elements to go to a single work group to be counted. This could definitely run into some issues on the network. This is the same performance concern when doing high cardinality group by queries on billions of records in BigQuery.

In this example, there are a million X values and only a thousand Y values. **GroupByKey** will group all of the X values on one worker. The worker will take much longer to do its processing on the million values than the other worker, which only has a thousand values to process.

Of course, you're paying for the worker that sits idle waiting for the other worker to complete. Dataflow is designed to avoid efficiencies by keeping the data balance.

You can help by designing your application to divide work into aggregation steps and subsequent steps and to avoid grouping or to push grouping towards the end of the processing pipeline.



CoGroupByKey is very similar. It groups results across several PCollections by key. The result for each key is a tuple of the values associated with that key in each input collection.

Combine (reduce) a PCollection

Applied to a PCollection of values

```
totalAmount = salesAmounts | CombineGlobally(sum)
```

Applied to a grouped Key-Value pair

```
totalSalesPerPerson = salesRecords | CombinePerKey(sum)
```

Each element of salesRecords is a tuple: (salesPerson, salesAmount)

Pre-built combine functions for many common numeric combination operations such as sum, mean, min, and max

Now, we can move to the reduce phase. How do we calculate totals or averages or other aggregations on our PCollections? Combined is used to combine collections of elements or values in your data. Combine has variants that work on entire PCollections and some that combine the values for each key and PCollections of key-value pairs. **CombineGloballlyfn** reduces a PCollection to a single value by applying the FN or the function.

CombinePerKey is similar to GroupByKey, but combines the values by a combined function or a callable that takes an iterable action such as sum or max.

When you apply a combine transform, you must provide the function that contains the logic for combining the elements or values. There are pre-built combined functions for common numeric combination operations such as sum, min, and max. Simple combine operations such as sums can usually be implemented as a simple function.

More complex combination operations might require you to create a subclass of a combine function that has an accumulation type distinct from the input and or output site.

CombineFn works by overriding existing operations

You must provide four operations by overriding the corresponding methods

```
class AverageFn(beam.CombineFn):

  def create_accumulator(self):
    return (0.0, 0)

  def add_input(self, sum_count, input):
    (sum, count) = sum_count
    return sum + input, count + 1

  def merge_accumulators(self, accumulators):
    sums, counts = zip(*accumulators)
    return sum(sums), sum(counts)

  def extract_output(self, sum_count):
    (sum, count) = sum_count
    return sum / count if count else float('NaN')
```

```
pc = ...
average = pc | beam.CombineGlobally(AverageFn())
```

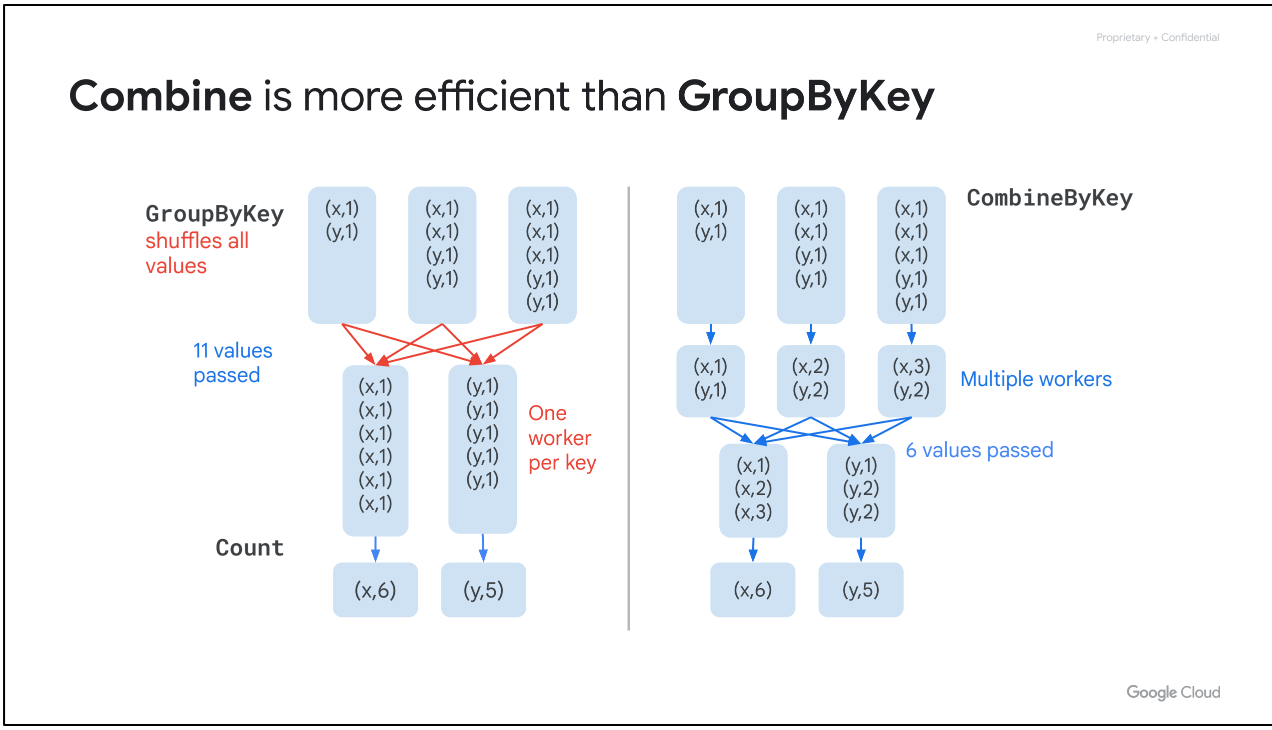

The combining function should be commutative and associative, as the function is not necessarily invoked exactly once on all values within a given key.

Because the input data including the value collection may be distributed across multiple workers, the combining function might be called multiple times to perform multiple combining on subsets of the value collection.

For more complex combined functions, you can define a subclass of combine function.

You should use the combine function if the action needed requires a more sophisticated accumulator, must perform additional pre or post processing, might change the output type, or takes the key into account.

A general combining operation consists of four operations. When you create a subclass of combine function, you must provide four operations by overriding the corresponding methods. Create accumulator creates a new local accumulator. In the example case taking a mean average, a local accumulator tracks the running sum of values.



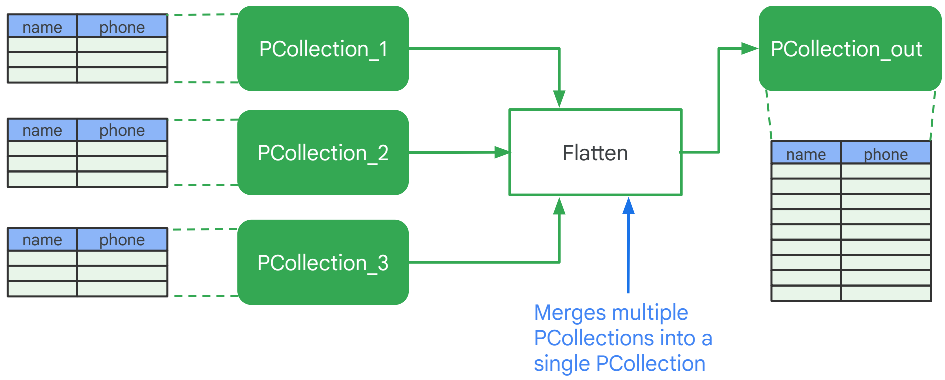
Combine is orders of magnitude faster than GroupByKey because Dataflow knows how to parallelize a combine step.

The way that **GroupByKey** works, Dataflow can use no more than one worker per key. In this example, **GroupByKey** causes all the values to be shuffled so they are all transmitted over the network. And then there is one worker for the 'x' key and one worker for the 'y' key.

Combine allows Dataflow to distribute a key to multiple workers and process it in parallel. In this example, **CombineByKey** first aggregates values and then processes the aggregates with multiple workers. Also, only 6 aggregate values need to be passed over the network.

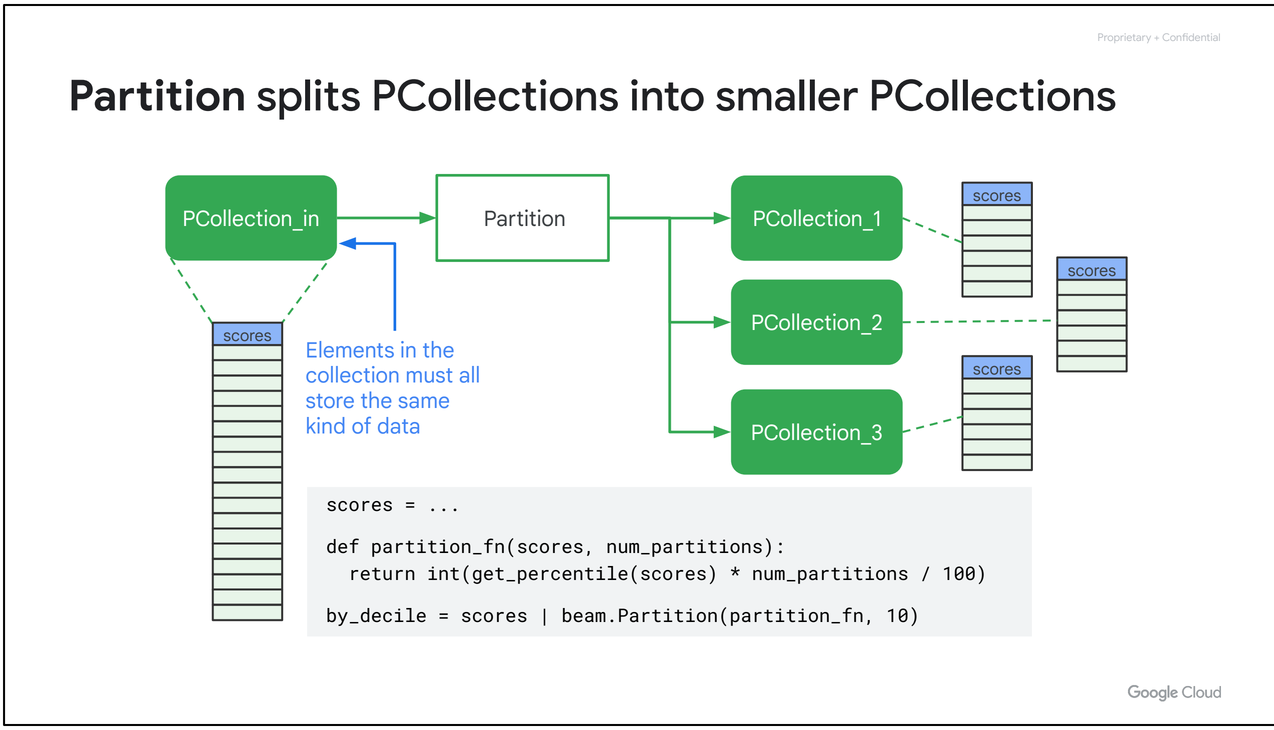
Combine is a Java interface that tells Dataflow that the combine operation (like Count) is both commutative and associative. This allows Dataflow to shard within a key vs. having to group each key first. As a developer, you can create your own custom **Combine** class for any operation that has commutative and associative properties.

Flatten merges identical PCollections




```
merged = ((pcoll1, pcoll2, pcoll3) | beam.Flatten())
```


Flatten works a lot like a SQL UNION. It's a beam transform for PCollection objects that store the same data type. Flatten merges multiple PCollection objects into a single logical PCollection. Partition is also a beam transform for PCollection objects that store the same data type.



Partition splits a single PCollection into a fixed number of smaller collections. You might use partition if, for example, you wanted to calculate percentages or quartiles and the top quartile has different processing than all the others.

Lab Intro

Serverless Data Analysis with Beam: MapReduce in Beam (Python/Java)

In this next lab, you'll practice creating and performing Map and Reduce operations on PCollections as part of your pipeline. You can choose between Python and Java.

Proprietary + Confidential

Lab objectives

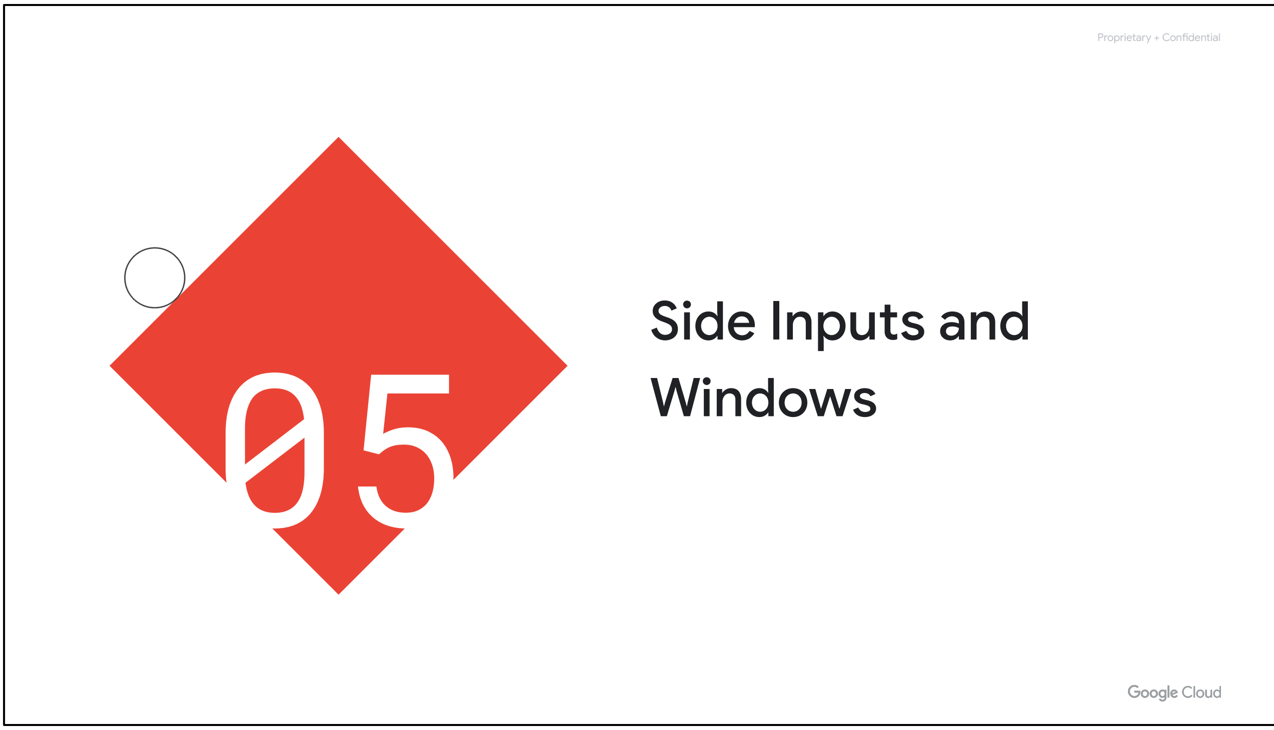
01 Identify Map and Reduce operations

02 Execute the pipeline

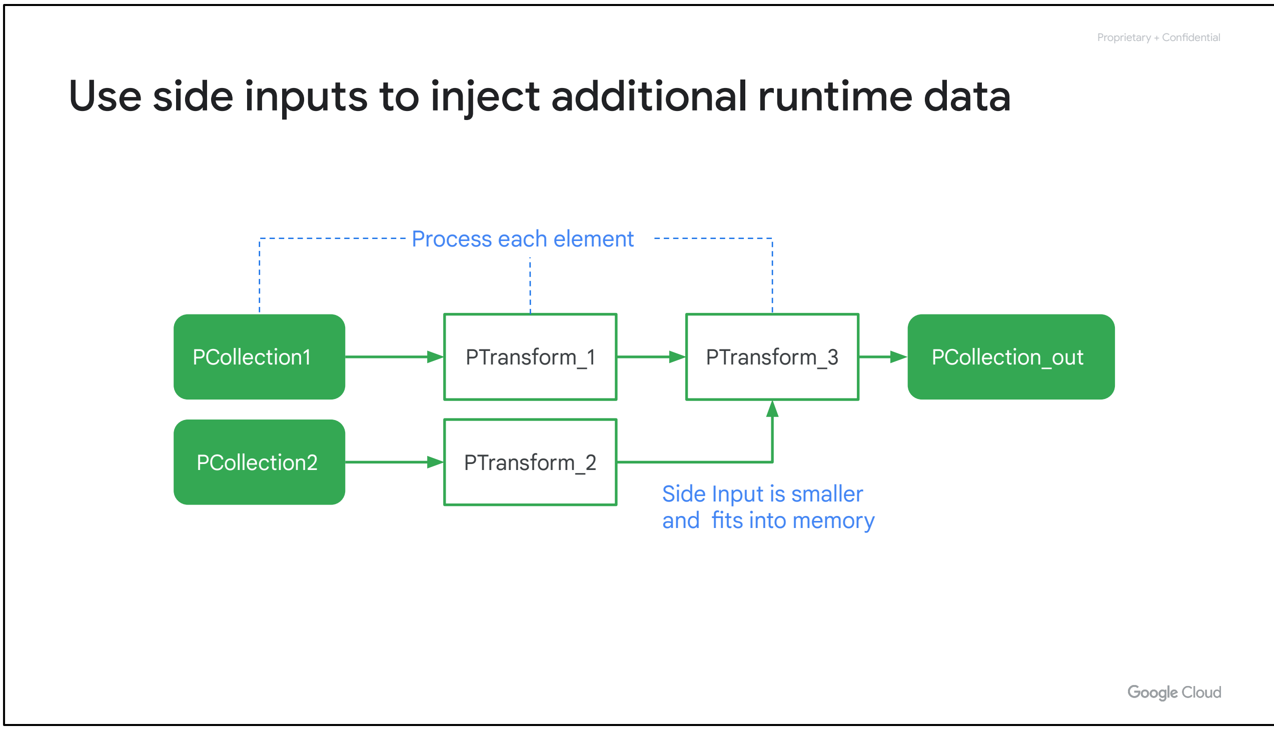
03 Use command line parameters

Google Cloud

You'll first identify the Map and Reduce operations, then execute the pipeline, and lastly modify command-line parameters.



In this lesson, you’ll learn about the role of side inputs and windows.



In addition to the main input PCollection, you can provide additional inputs to a ParDo transform in the form of side inputs. A side input is an additional input that your DoFn can access each time it processes an element in the input PCollection. When you specify a side input, you create a view of some other data that can be read from within the ParDo transform's DoFn while processing each element.

Side inputs are useful if your ParDo needs to inject additional data when processing each element in the input PCollection, but the additional data needs to be determined at runtime and not hard-coded. Such values might be determined by the input data, or depend on a different branch of your pipeline.

How side inputs work

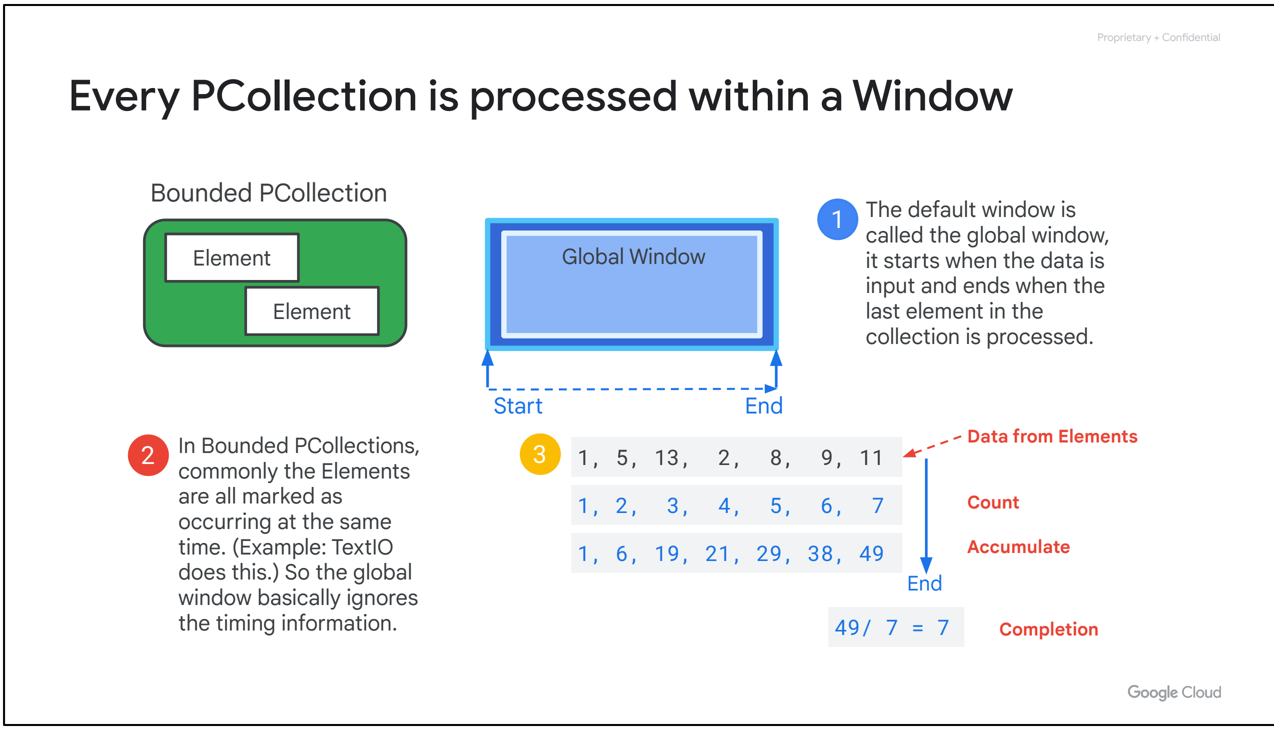
```
words = …

def filter_using_length(word, lower_bound, upper_bound=float('inf')):
  if lower_bound <= len(word) <= upper_bound:
    yield word

small_words = words | 'small' >> beam.FlatMap(filter_using_length, 0, 3)

avg_word_len = (words
                | beam.Map(len)
                | beam.CombineGlobally(beam.combiners.MeanCombineFn()))


larger_than_average = (words | 'large' >> beam.FlatMap(
    filter_using_length,
    lower_bound=pvalue.AsSingleton(avg_word_len)))
```

Side input

Here's how side inputs work. This is an example in Python.

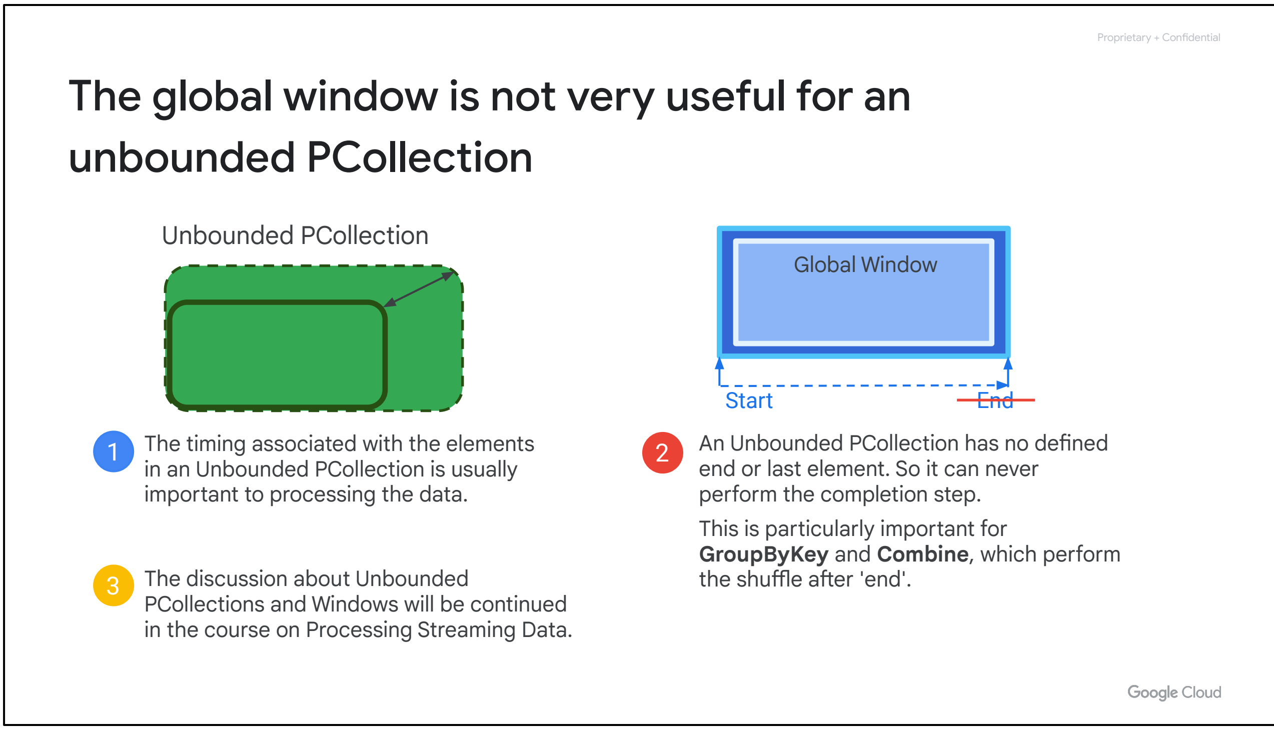
This set of steps is actually a subgraph of our overall graph. It begins with words that run through the map function to get the length and then combine globally to compute the total lengths across the whole dataset.

So if we were trying to figure out if any given word is shorter or longer than the average word length, first we need to compute the average word length using these steps. But then this whole branch can be fed into this method. That's what creates the view which is static and then becomes available to all the worker nodes for later use. That is a side input you see here.



Before we go to the next lab, here are a few notes about additional capabilities.

Many transforms have two parts, one occurs item-at-a-time until all items are processed, and another occurs after the last item is processed. One of the easiest analogies is the arithmetic mean. You add up the value of each element and keep count. This is the accumulation step. After you have processed all the elements, you have a total of all the values read and a count of the number of values read. The last thing to do is divide the total by the count. This is fine so long as you know you have read the last item. But if you have an unbounded dataset, there is no predetermined end. So you just keep adding and never break out of the loop and perform the division.



The global window is not very useful for an unbounded PCollection, meaning streaming data. The timing associated with the elements and an unbounded PCollection is usually important to processing the data. An unbounded PCollection has no defined end or last element, so it can never perform the completion step. This is particularly important for GroupByKey and Combined, which perform the shuffle after end. The discussion about unbounded PCollections and Windows will be continued in the course on processing streaming data.

Setting a single global window for a PCollection

Single global window

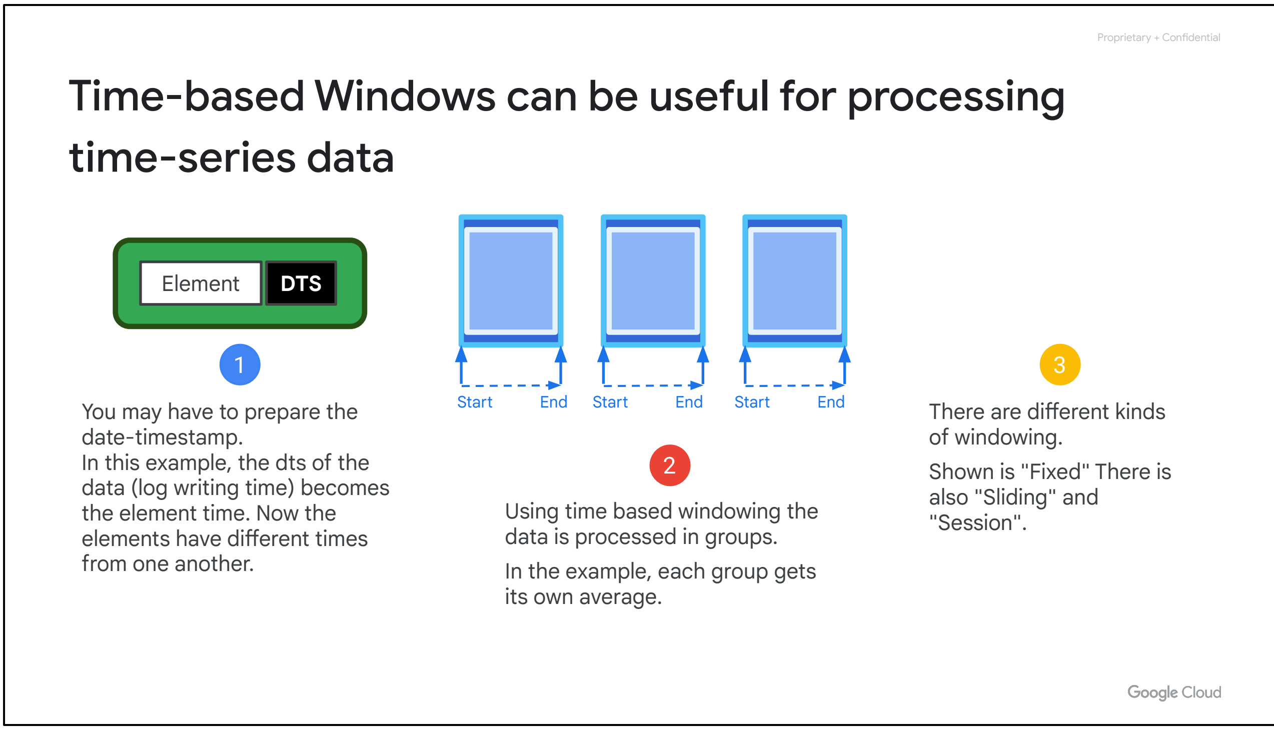
Python

```
from apache_beam import window
session_windowed_items = (
    items | 'window' >> beam.WindowInto(window.GlobalWindows()))
```

This is the default.

This code illustrates how you could explicitly set it.

The global window is a default and here you see how you can set it with beam.WindowInto window.GlobalWindows.



So are streaming pipelines out of luck if they can't use the global window?

No, you can use time-based Windows, which can be useful for processing data that comes in streaming at different times. We'll cover this in detail in the streaming course.

Using Windowing with Batch (group by time)

Python

```
lines = p | 'Create' >> beam.io.ReadFromText('access.log')
windowed_counts = (
    lines
    | 'Timestamp' >> beam.Map(beam.window.TimestampedValue(x, extract_timestamp(x)))
    | 'Window' >> beam.WindowInto(beam.window.SlidingWindows(60, 30))
    | 'Count' >>
(beam.CombineGlobally(beam.combiners.CountCombineFn()).without_defaults())
)
windowed_counts =  windowed_counts | beam.ParDo(PrintWindowFn())
```

access.log (example)

```
131.108.5.17 - - [29/Apr/2019:04:53:15 -0800] "GET /view HTTP/1.1" 200 7352
131.108.5.17 - - [29/Apr/2019:05:21:35 -0800] "GET /view HTTP/1.1" 200 5253
```

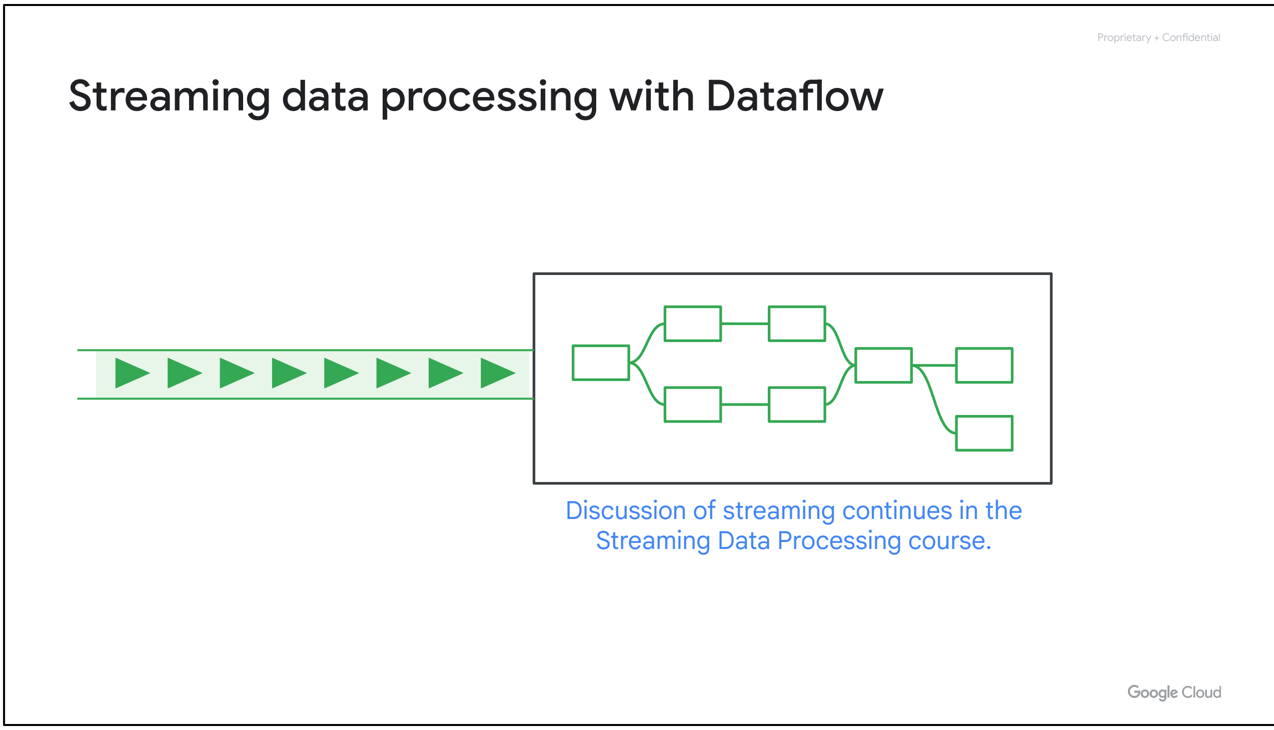
Date Time Stamp

For batch inputs, you can group by time as well. You can explicitly admit a timestamp in your pipeline instead of standard output. In this example, an offline access log is being read, and the date/timestamp is extracted and used for windowing. Here we're using Windows to aggregate our batch data by time.

Subsequent groups, aggregations, and so forth are computed only within the time window.

This example here uses a sliding window, as you can see with beam.WindowInto beam.window.SlidingWindows 60 30, which means capture 60 seconds worth of data, but start a new window every 30 seconds.

So for example, say you had all of your sales records and you wanted to compute sales by day. You'd just extract that timestamp field that represents the timestamp. Then you would create fixed windows with a one-day duration and data flow automatically will compute the sum over each window to compute those totals. The main thing to remember here is that you can do this in batch.



Discussion of streaming continues in the Streaming Data Processing course.

Proprietary + Confidential

Lab Intro

Serverless Data Analysis with Dataflow: Side Inputs (Python/Java)

Google Cloud

In this lab, you'll practice creating Site Inputs to your Dataflow pipeline. You once again have the choice to complete the lab using Python or Java.

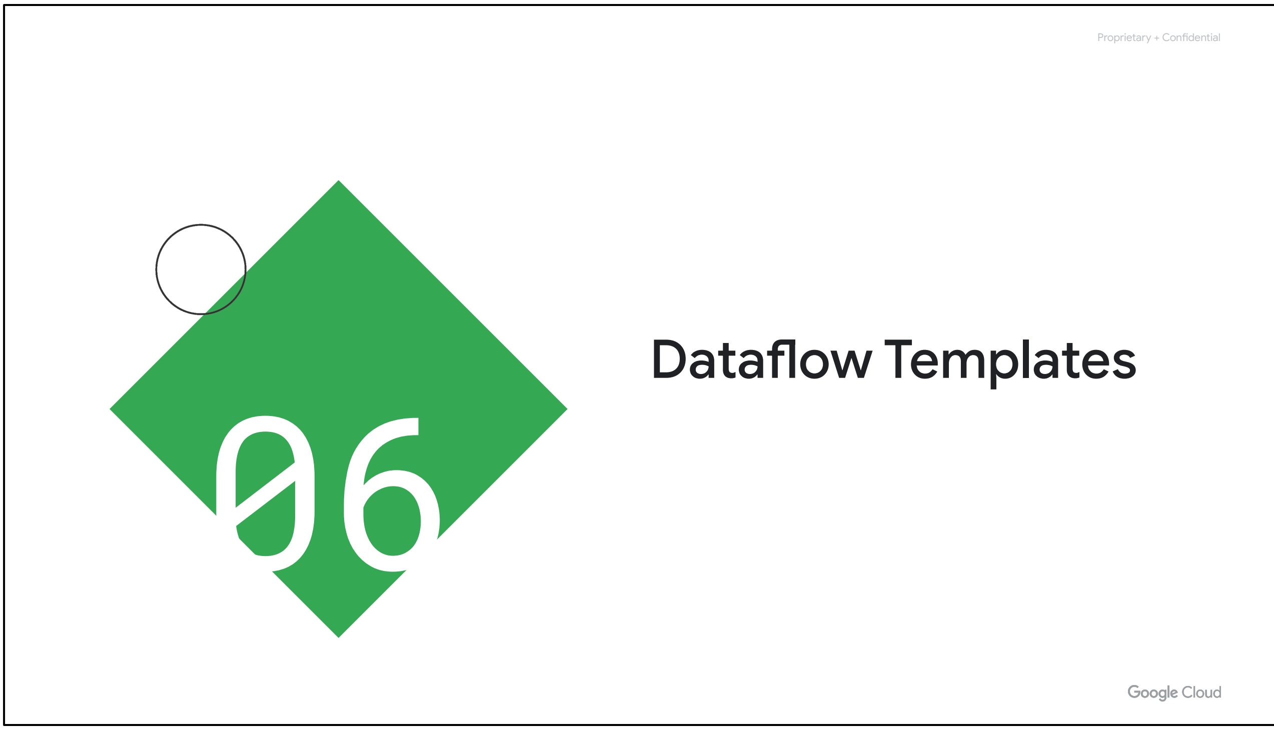
Lab objectives

01 Try out a BigQuery query

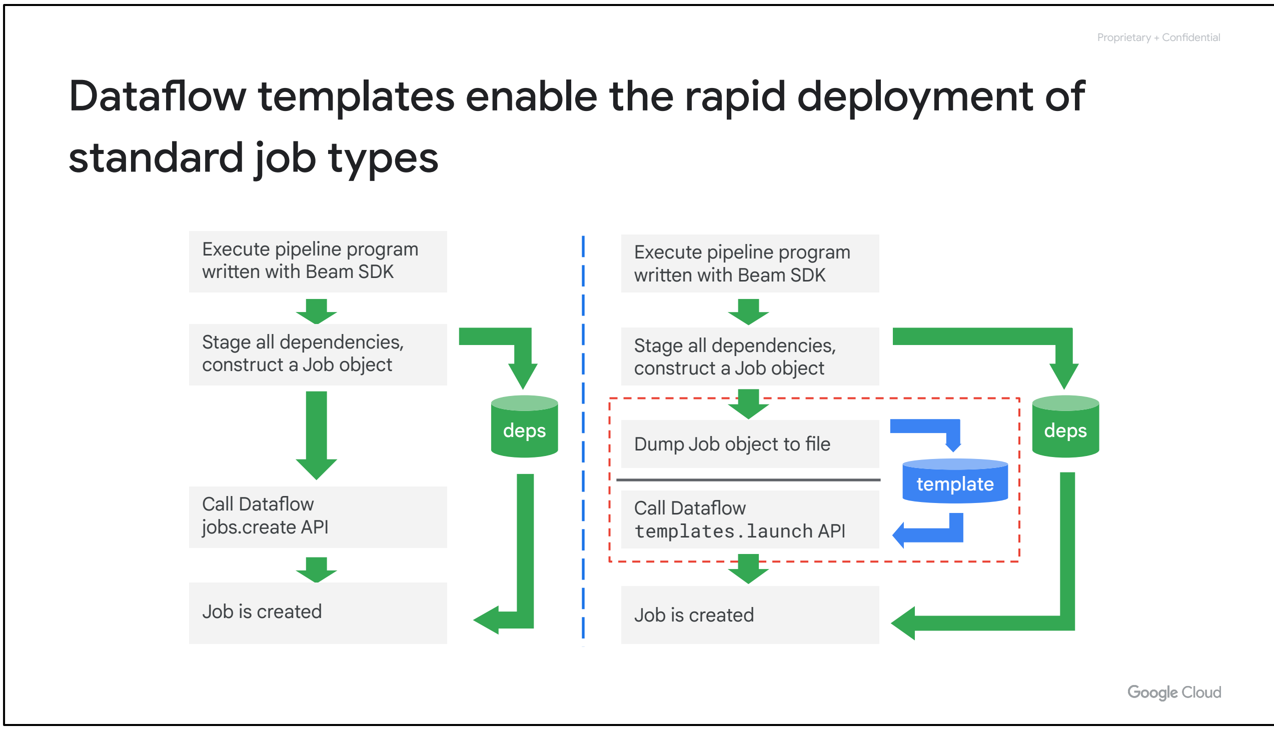
02 Explore the pipeline code

03 Execute the pipeline

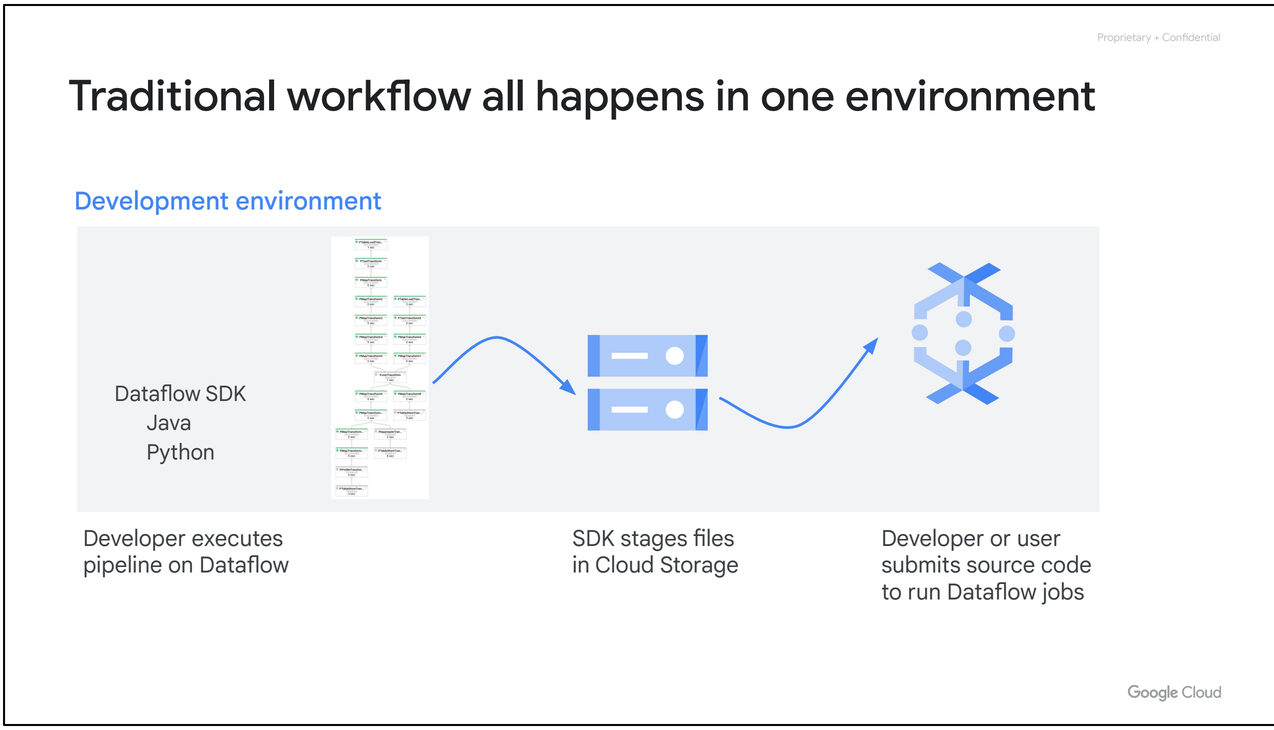
Specifically, you'll bring in data from BigQuery into your pipeline and then execute the job on Dataflow.



Next, we'll look at Dataflow templates where you as a data engineer can create new templates for your team to leverage. You can also start from some of Google's pre-existing templates, which we'll cover as well.



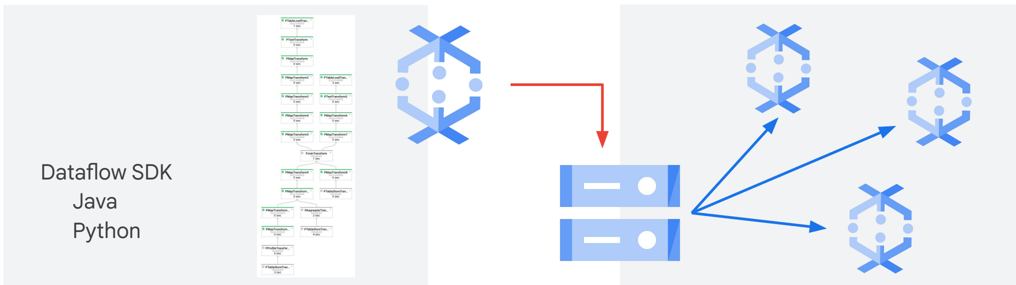
Dataflow templates allow users who don't have any coding capability to execute their Dataflow job. It enables the rapid deployment of standard types of data transformation jobs, removing the need to develop the pipeline code, and removing the need to consider the management of components’ dependencies in the pipeline code.



In the traditional workflow the developer creates the pipeline in the development environment using the Dataflow SDK in Java or Python. And there are dependencies to the original language and SDK files. Whenever a job is submitted it is re-processed entirely or re-compiled. There is no separation of developers from users. So the users basically have to be developers or have the same access and resources as developers.

Template workflow supports non-developer users

Development environment | **Production environment**



Developer creates pipeline in the development environment

Dataflow stores template in Cloud Storage

Users submit templates to run jobs

Dataflow templates enable a new development and execution workflow. The templates help separate the development activities and the developers from the execution activities and the users. The user environment no longer has dependencies back to the development environment. The need for recompilation to run a job is limited. The new approach facilitates the scheduling of batch jobs and opens up more ways for users to submit jobs, and more opportunities for automation.

Proprietary + Confidential

Get started with Google-provided templates

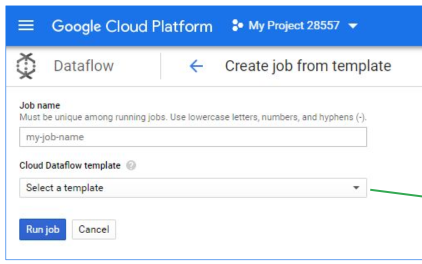
Pre-written Dataflow pipelines for common data tasks that can be triggered with a single command or UI form.

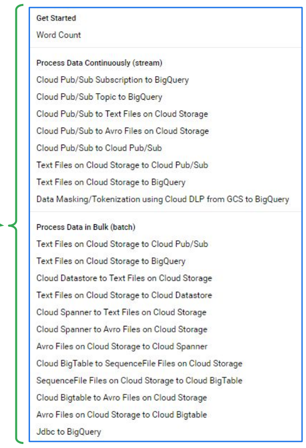
Target users	Exposure	Data Fusion
• App developers • DB admins • Analysts • Data scientists • Data engineers	• Through Google-provided Dataflow templates • Embedded in other Google Cloud products calling templates API	• Branded Google product • UI pipeline builder • Scheduler/orchestrator

Google Cloud

App developers, database administrators, analysts, and data scientists can use templates as a solution.

Execute templates with the Cloud Console, `gcloud` command-line tool, or the REST API



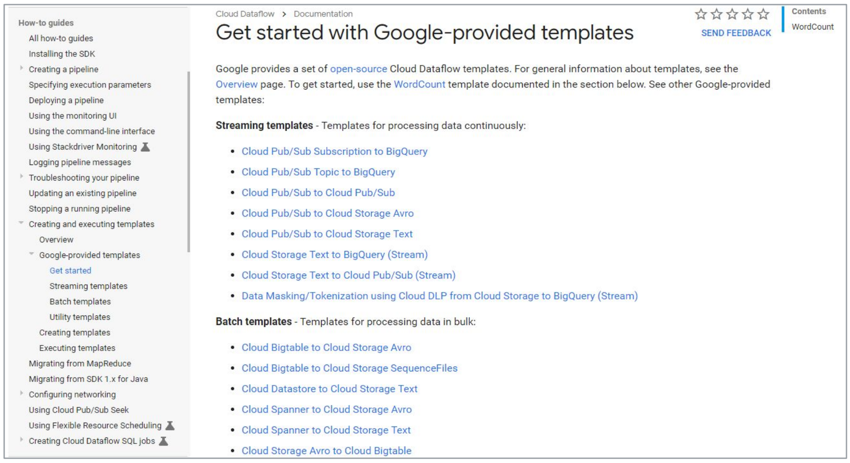



```
gcloud dataflow jobs run \
--gcs-location=gs://df-ts/latest/PubsubToBigQuery \
--parameters inputTopic=X outputTable=Y
```

Google Cloud

You can also run them using the command-line tool, or REST API as you see here. Simply specify the Cloud Storage location of your template that you already have.

Google-provided templates documentation



How-to guides
All how-to guides
Installing the SDK
Creating a pipeline
Specifying execution parameters
Deploying a pipeline
Using the monitoring UI
Using the command-line interface
Using Stackdriver Monitoring
Logging pipeline messages
Troubleshooting your pipeline
Updating an existing pipeline
Stopping a running pipeline
Creating and executing templates
Overview
Google-provided templates
Get started
Streaming templates
Batch templates
Utility templates
Creating templates
Executing templates
Migrating from MapReduce
Migrating from SDK 1.x for Java
Configuring networking
Using Cloud Pub/Sub Seek
Using Flexible Resource Scheduling
Creating Cloud Dataflow SQL jobs

Cloud Dataflow > Documentation

Get started with Google-provided templates

SEND FEEDBACK

Contents
WordCount

Google provides a set of open-source Cloud Dataflow templates. For general information about templates, see the Overview page. To get started, use the WordCount template documented in the section below. See other Google-provided templates:

Streaming templates - Templates for processing data continuously:

- Cloud Pub/Sub Subscription to BigQuery
- Cloud Pub/Sub Topic to BigQuery
- Cloud Pub/Sub to Cloud Pub/Sub
- Cloud Pub/Sub to Cloud Storage Avro
- Cloud Pub/Sub to Cloud Storage Text
- Cloud Storage Text to BigQuery (Stream)
- Cloud Storage Text to Cloud Pub/Sub (Stream)
- Data Masking/Tokenization using Cloud DLP from Cloud Storage to BigQuery (Stream)

Batch templates - Templates for processing data in bulk:

- Cloud Bigtable to Cloud Storage Avro
- Cloud Bigtable to Cloud Storage SequenceFiles
- Cloud Datastore to Cloud Storage Text
- Cloud Spanner to Cloud Storage Avro
- Cloud Spanner to Cloud Storage Text
- Cloud Storage Avro to Cloud Bigtable

Alternatively, you can use the Google-provided templates.

Proprietary + Confidential

Which means now you can...

- Launch Dataflow jobs programmatically (via API).
- Launch Dataflow jobs instantaneously.
- Re-use Dataflow jobs.
- Letting you customize the execution of your pipeline.

Google Cloud

After you create and stage your Dataflow template, execute the template with the Cloud Console, REST API, or gcloud command-line tool. You can deploy Dataflow template jobs from many environments, including App Engine standard environment, Cloud Functions, and other constrained environments.

What if you want to create your own template?

1. Modify pipeline options with ValueProviders.
2. Generate template file.

```
mvn compile exec:java \
 -Dexec.mainClass=com.example.myclass \
 -Dexec.args="--runner=DataflowRunner \
              --project=[YOUR_PROJECT_ID] \
              --stagingLocation=gs://[YOUR_BUCKET_NAME]/staging \
              --output=gs://[YOUR_BUCKET_NAME]/output \
              --templateLocation=gs://[YOUR_BUCKET_NAME]/templates/MyTemplate"
```

3. Call it from an API.

```
POST https://dataflow.googleapis.com/v1b3/projects/[YOUR_PROJECT_ID]/templates:launch?gcsPath=gs://[
{
    "jobName": "[JOB_NAME]",
    "parameters": {
        "inputFile" : "gs://[YOUR_BUCKET_NAME]/input/my_input.txt",
        "outputFile": "gs://[YOUR_BUCKET_NAME]/output/my_output"
    },
    "environment": {
        "tempLocation": "gs://[YOUR_BUCKET_NAME]/temp",
        "zone": "us-central1-f"
    }
}
```

Google Cloud

What if you wanted to create your own template? To create your own template, you'll add your own Value Providers. This is what parses the command-line or optional arguments to your template, and that is how users can specify optional arguments. Once a template file is created, you call it from an API.

Templates require modifying parameters for runtime

Python

```
class WordcountOptions(PipelineOptions):
    @classmethod
    def _add_argparse_args(cls, parser):
      parser.add_value_provider_argument(
          '--input',
          default='gs://dataflow-samples/shakespeare/kinglear.txt',
          help='Path of the file to read from')
      parser.add_argument(
          '--output',
          required=True,
          help='Output file to write results to.')
  pipeline_options = PipelineOptions(['--output', 'some/output_path'])
  p = beam.Pipeline(options=pipeline_options)

  wordcount_options = pipeline_options.view_as(WordcountOptions)
  lines = p | 'read' >> ReadFromText(wordcount_options.input)
```

Run-time parameters

Non run-time parameters can stay

Runtime parameters must be modified.

You might not have considered this before, but values like "user options" and "input file" that are compiled into your job. They aren't just parameters, they are compile-time parameters. To make these values available to non-developer users, they have to be converted to runtime parameters. These work through the ValueProvider interface so that your users can set these values when the template is submitted. ValueProvider can be used in I/O, transformations, and your functions. There are also Static and Nested versions of ValueProvider for more complex cases.

Proprietary + Confidential

Creating a template

- ValueProviders are passed down throughout the whole pipeline construction phase
- ValueProvider.get() only available in processElement()
  - Because it is fulfilled via API call

```java
public interface SumIntOptions extends PipelineOptions {
    // New runtime parameter, specified by the --int
    // option at runtime.
    ValueProvider<Integer> getInt();
    void setInt(ValueProvider<Integer> value);
}

class MySumFn extends DoFn<Integer, Integer> {
    ValueProvider<Integer> mySumInteger;

    MySumFn(ValueProvider<Integer> sumInt) {
        // Store the value provider
        this.mySumInteger = sumInt;
    }

    @ProcessElement
    public void processElement(ProcessContext c) {
       // Get the value of the value provider and add it to
       // the element's value.
       c.output(c.element() + mySumInteger.get());
    }
}

public static void main(String[] args) {
  SumIntOptions options =
        PipelineOptionsFactory.fromArgs(args).withValidation()
          .as(SumIntOptions.class);
```

Google Cloud

This is a Java example for creating your own template. Note that ValueProviders are passed down throughout the whole pipeline construction phase.

Proprietary + Confidential

Nested Value Providers

```
public static void main(String[] args) {
  pipeline
    .apply(Create.of(1, 2, 3).withCoder(BigEndianIntegerCoder.of()));
    // Write to the computed complete file path.
    .apply("OutputNums", TextIO.write().to(NestedValueProvider.of(
      options.getFileName(),
      new SerializableFunction<String, String>() {
        @Override
        public String apply(String file) {
          return "gs://bucket/" + file;
        }
      })));

  pipeline.run();
}
```

Google Cloud

Sometimes we need to transform a value from what the user passes at runtime, to what a source or sink expects to consume. Nested ValueProviders meet this need.

Template metadata

Located at the same directory, named <template_name>_metadata

```
{
  "name": "WordCount",
  "description": "An example pipeline that counts words in the input file.",
  "parameters": [{
    "name": "inputFile",
    "label": "Input Cloud Storage File(s)",
    "help_text": "Path of the file pattern glob to read from.",
    "regexes": ["^gs:\/\/[^\n\r]+$"],
    "is_optional": true
  },
  {
    "name": "output",
    "label": "Output Cloud Storage File Prefix",
    "help_text": "Path and filename prefix for writing output files. ex: gs://MyBucket/counts",
    "regexes": ["^gs:\/\/[^\n\r]+$"]
  }]
}
```


Each template has associated metadata with it upon creation. This will help your downstream users know what your template is doing and what parameters it expects. The metadata file is located in the same directory as your template and simply has the underscore metadata suffix to the name.

Proprietary + Confidential

Summary

Dataflow versus Dataproc

Google Cloud

Earlier in the course, you learned how to do batch processing of your Hadoop and Spark jobs using Dataproc. This is an ideal first step to the Cloud for existing jobs, simply run them on Dataproc and they just work. You learned that Dataflow takes a lot of the cluster resizing and other management tasks and automates them for you as a true serverless product. Use Dataflow if you're writing new pipelines or if you're ready to rewrite and migrate your Hadoop jobs to faster processing with Apache Beam on Dataflow.

Proprietary + Confidential

Summary

Dataflow versus Dataproc

Building Dataflow pipelines in code

Google Cloud

You then saw how to build pipelines using Apache Beam which is open source. For the pipelines to work, we created inputs with a beam.io syntax and walked through how you can read CSV files from Cloud Storage, streaming message queues from Pub/Sub, and structure data already living in BigQuery.

Proprietary + Confidential

Summary

Dataflow versus Dataproc

Building Dataflow pipelines in code

Key considerations with designing pipelines

Google Cloud

We then looked at some key considerations when designing your pipeline. Recall that you should consider using Combine when you can instead of GroupByKey, especially if your data is heavily skewed. This will prevent a single worker from being a bottleneck if you have a high cardinality dataset.

Summary

Dataflow versus Dataproc

Building Dataflow pipelines in code

Key considerations with designing pipelines

Transforming data with PTransforms

To do the actual transformations, you practiced writing PTransforms in your labs. Remember that the P in PTransforms and PCollections means parallel.

Summary

Dataflow versus Dataproc

Building Dataflow pipelines in code

Key considerations with designing pipelines

Transforming data with PTransforms

Aggregating with GroupByKey and Combine

For the “reduce” part of MapReduce, we looked at aggregation functions like GroupByKey and Combine. Keep in mind you can have multiple parallel parts of your pipeline Combine into a single PTransform like an aggregation. The pipeline does not have to execute in serial unless you've set it up that way with dependencies.

Proprietary + Confidential

Summary

Dataflow versus Dataproc

Building Dataflow pipelines in code

Key considerations with designing pipelines

Transforming data with PTransforms

Aggregating with GroupByKey and Combine

Side inputs and windows of data

Google Cloud

After that you practiced with side inputs in your lab and how to create windows of data even for batch datasets.

Proprietary + Confidential

Summary

Dataflow versus Dataproc

Building Dataflow pipelines in code

Key considerations with designing pipelines

Transforming data with PTransforms

Aggregating with GroupByKey and Combine

Side inputs and windows of data

Creating and reusing Pipeline Templates

Google Cloud

Lastly, you saw how to create and save new Dataflow templates for your team to use and where you can see Google's pre-made templates in our public GitHub.