**Serverless Data Processing with Dataflow.**

**Module Introduction.**

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.

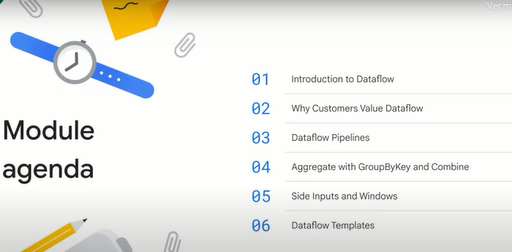
This entire module will cover batch Dataflow pipelines and why Dataflow is a commonly used data pipeline tool on Google Cloud.

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..



**Introduction to Dataflow.**

Let's start by exploring Dataflow in more detail.

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.

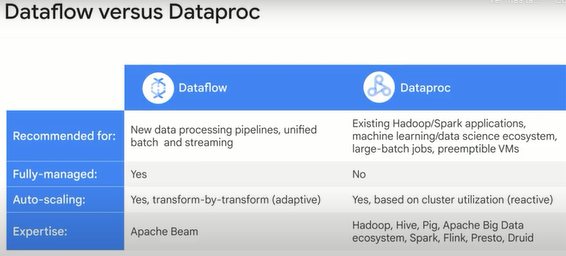
Unlike with Dataproc, the auto scaling in Dataflow scales step by step, it's very fine grained.

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.

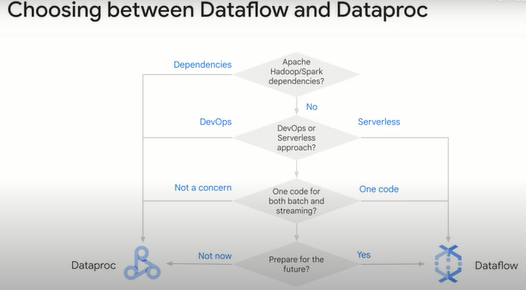
Please keep in mind that there are many subjective issues when making this decision, and that no simple guide will fit every use case.

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.

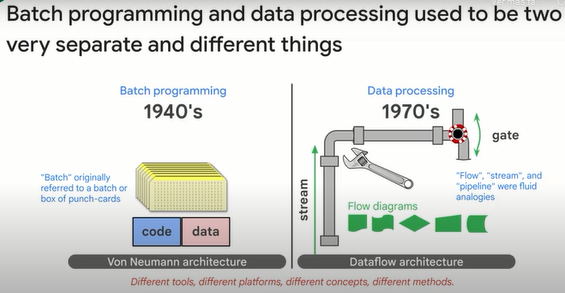
So that's what you were processing, a box of punch cards call a batch.

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 P transforms, P collections, 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 P collection.

The P collection is immutable.

Any change that happens in a pipeline ingests one P collection and creates a new one.

It does not change the incoming P collection.

The action or code is contained in an abstraction called a P transform.

P transform handles input, transformation, an output of the data.

The data in the P collection is passed along a graph from one P transform 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.

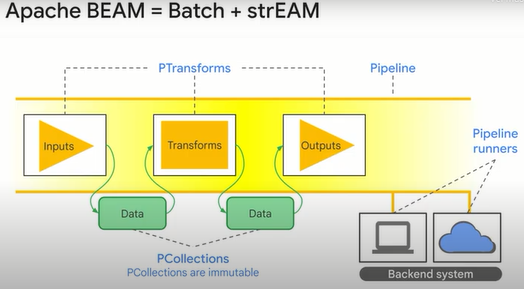
The only difference is scale and access to platform specific services.

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.



The shape of a pipeline is not actually just a singular linear progression but rather a directed graph with brunches and aggregations.

For historical reasons, we refer to it as a pipeline, but a data graph or Dataflow might be a more accurate description.

A P collection represents both streaming data and batch data.

There is no size limit to a P collection.

Streaming data is an unbounded P collection that doesn't end.

Each element inside a P collection can be individually accessed and processed.

This is how distributed processing of the P collection is implemented.

So you define the pipeline and the transforms on the P collection 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 P collection, it is immutable, so it can never be changed or deleted.

Elements represent different data types.

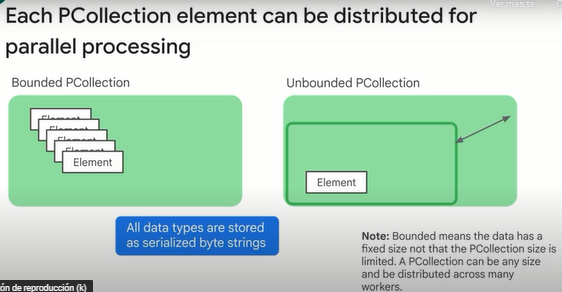
In traditional programs, a data type is stored in memory with a format that favors processing.

Integers in memory are different from characters which are different from strings and compound data types.

In a P collection, 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 P transform.



**Why customers value Dataflow.**

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 the end of a job stage or on downscaling.

Work scheduled on a resource is guaranteed to be processed.

Work can be dynamically rebalanced across resources.

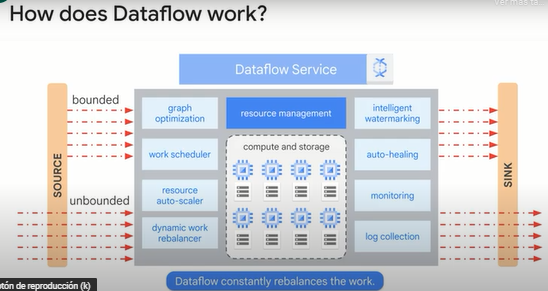
This provides fault tolerance.

The watermarking handles late arrivals of data and comes with restarts, monitoring and logging.

No more waiting for other jobs to finish.

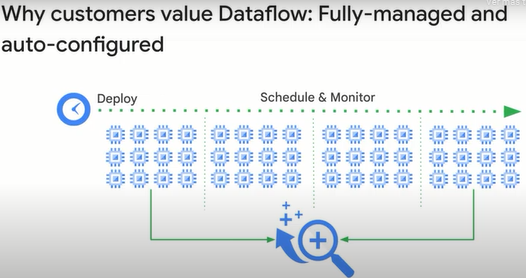
No more preemptive scheduling.

Dataflow provides a reliable, serverless, job-specific way to process your data.



To summarize, the advantages of Dataflow are, first, Dataflow is fully managed and auto configured.

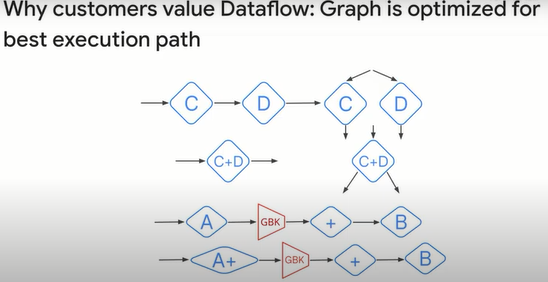
Just deploy your pipeline.



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.

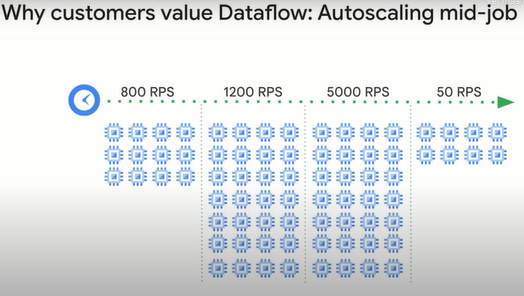
We see this with A and the Group By Key.



Third, autoscaling happens step-by-step in the middle of a job.

As the job needs more resources, it receives more resources.

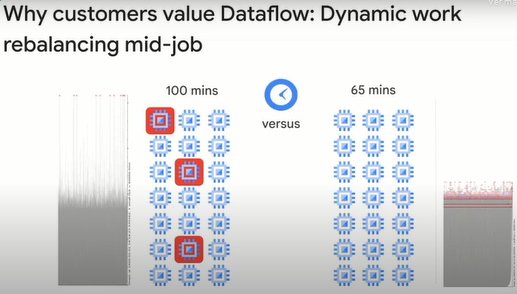
You don't have to manually scale resources to match job needs.



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.



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.



**Building Dataflow pipelines in code.**

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.

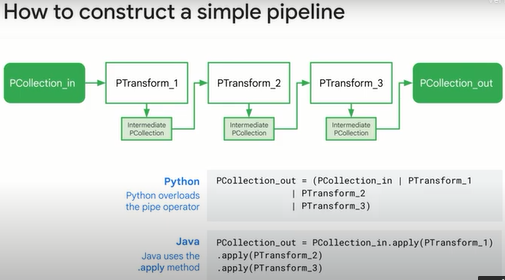
You have the input, the pipe symbol, the first PTransform, the pipe symbol, the second PTransform, et cetera.

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.



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

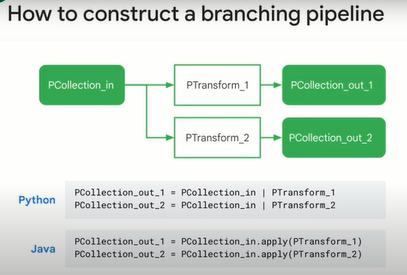
Give the output PCollection variable in each case a name.

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 of the first case, we store as PCollection\_out\_1.

In the second case, we store it as PCollection\_out\_2.



What we showed you so far is 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 cloud storage.

That's the read from text 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 go to the output PCollection.

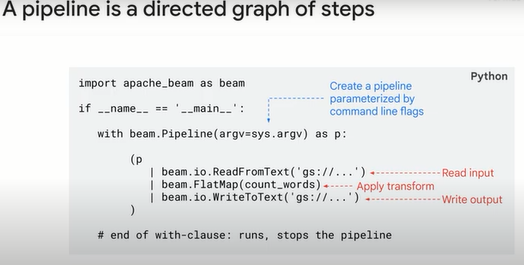
The function in this case is the function called count\_words.

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.



Once you have written the pipeline, it's time to run it.

Executing the Python program on the previous slide will run the program.

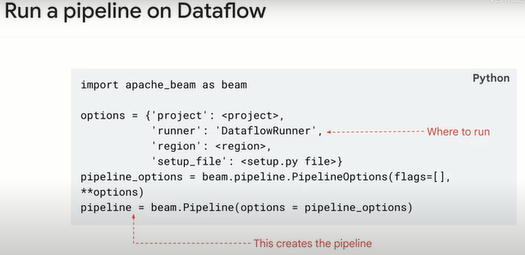
By default, the program is run using the default runner, which runs on the same machine where the Python program was executed.

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.

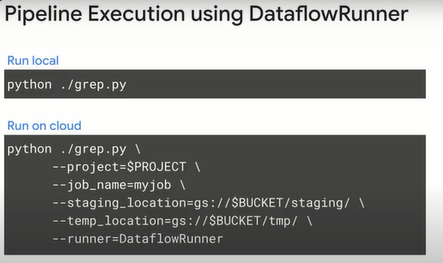
This example contains hard coded variables, which in most cases is not a preferred practice for programming at scale.



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.



**Key considerations with designing pipelines.**

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 file name.

Note the use of an asterisk wild card 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.

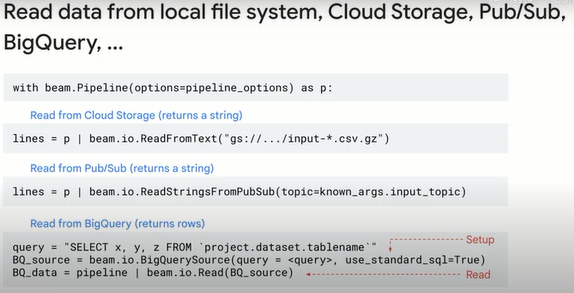
What about if you wanted to read in data that's already in BigQuery?

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? (“repositories”)



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, data set ID and table name.

Then you use beam.io.

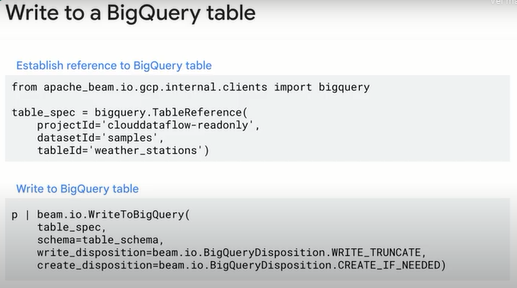
WriteToBigQuery as a sink to your pipeline.

Note that we are using the normal BigQuery options here for write\_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.

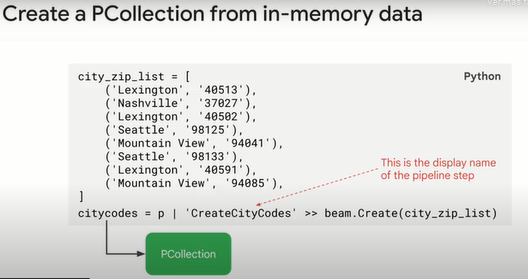


You can also create a PCollection in memory without reading from a particular source.

Why might you do this?

If you have a small data set, 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.



**Lab: A Simple Dataflow Pipeline (Python) 2.5.**

**Task 1. Ensure that the Dataflow API is successfully enabled**

* Execute the following block of code in the Cloud Shell:

gcloud services disable dataflow.googleapis.com --force

gcloud services enable dataflow.googleapis.com

**Task 2. Preparation**

Open the SSH terminal and connect to the training VM

You will be running all code from a curated training VM.

1. In the console, on the **Navigation menu** (Navigation menu icon), click **Compute Engine** > **VM instances**.
2. Locate the line with the instance called **training-vm**.
3. On the far right, under **Connect**, click on **SSH** to open a terminal window.
4. In this lab, you will enter CLI commands on the **training-vm**.

Download code repository

* Download a code repository to use in this lab. In the **training-vm** SSH terminal enter the following:

git clone https://github.com/GoogleCloudPlatform/training-data-analyst

Create a Cloud Storage bucket

Follow these instructions to create a bucket.

1. In the console, on the **Navigation menu**, click **Home**.
2. **Select and copy** the Project ID.

For simplicity use the Project ID found in the Lab details panel is already globally unique. Use it as the bucket name.

1. In the console, on the **Navigation menu**, click **Cloud Storage** > **Browser**.
2. Click **Create Bucket**.
3. Specify the following, and leave the remaining settings as their defaults:

|  |  |
| --- | --- |
| **Property** | **Value (type value or select option as specified)** |
| **Name** | <your unique bucket name (Project ID)> |
| **Location type** | Multi-Region |
| **Location** | <Your location> |

1. Click **Create**.

Record the name of your bucket to use in subsequent tasks.

1. In the **training-vm** SSH terminal enter the following to create an environment variable named "BUCKET" and verify that it exists with the echo command:

BUCKET="<your unique bucket name (Project ID)>"

echo $BUCKET

You can use $BUCKET in terminal commands. And if you need to enter the bucket name <your-bucket> in a text field in the console, you can quickly retrieve the name with echo $BUCKET.

**Task 3. Pipeline filtering**

The goal of this lab is to become familiar with the structure of a Dataflow project and learn how to execute a Dataflow pipeline.

1. Return to the **training-vm** SSH terminal and navigate to the directory /training-data-analyst/courses/data\_analysis/lab2/python and view the file grep.py.
2. View the file with Nano. **Do not make any changes to the code:**

cd ~/training-data-analyst/courses/data\_analysis/lab2/python

nano grep.py

1. Press CTRL+X to exit Nano.

Can you answer these questions about the file grep.py?

* What files are being read?
* What is the search term?
* Where does the output go?

There are three transforms in the pipeline:

* What does the transform do?
* What does the second transform do?
* Where does its input come from?
* What does it do with this input?
* What does it write to its output?
* Where does the output go?
* What does the third transform do?

**Task 4. Execute the pipeline locally**

1. In the **training-vm** SSH terminal, locally execute grep.py:

python3 grep.py

The output file will be output.txt. If the output is large enough, it will be sharded into separate parts with names like: output-00000-of-00001.

1. Locate the correct file by examining the file's time:

ls -al /tmp

1. Examine the output file(s).
2. You can replace "-\*" below with the appropriate suffix:

cat /tmp/output-\*

Does the output seem logical?

**Task 5. Execute the pipeline on the cloud**

1. Copy some Java files to the cloud. In the **training-vm** SSH terminal, enter the following command:

gsutil cp ../javahelp/src/main/java/com/google/cloud/training/dataanalyst/javahelp/\*.java gs://$BUCKET/javahelp

Copied!

content\_copy

1. Using Nano, edit the Dataflow pipeline in grepc.py:

nano grepc.py

1. Replace PROJECT and BUCKET with your Project ID and Bucket name.

Example strings before you update:

PROJECT='cloud-training-demos'

BUCKET='cloud-training-demos'

Example strings after edit (use your values):

PROJECT='qwiklabs-gcp-your-value'

BUCKET='qwiklabs-gcp-your-value'

Save the file and close Nano by pressing the CTRL+X key, then type Y, and press Enter.

1. Submit the Dataflow job to the cloud:

python3 grepc.py

**Note:** Ignore the message: **WARNING:root:Make sure that locally built Python SDK docker image has Python 3.7 interpreter.** Your Dataflow job will start successfully.

Because this is such a small job, running on the cloud will take significantly longer than running it locally (on the order of 7-10 minutes).

1. Return to the browser tab for the console.
2. On the **Navigation menu**, click **Dataflow** and click on your job to monitor progress.
3. Wait for the **Job status** to be **Succeeded**.
4. Examine the output in the Cloud Storage bucket.
5. On the **Navigation menu**, click **Cloud Storage > Browser** and click on your bucket.
6. Click the **javahelp** directory.

This job generates the file output.txt. If the file is large enough, it will be sharded into multiple parts with names like: output-0000x-of-000y. You can identify the most recent file by name or by the **Last modified** field.

1. Click on the file to view it.

Alternatively, you can download the file via the **training-vm** SSH terminal and view it:

gsutil cp gs://$BUCKET/javahelp/output\* .

cat output\*

**End your lab**

**Lab: Serverless Data Analysis with Dataflow: A Simple Dataflow Pipeline (Java).**

**Task 1. Preparation**

For this lab, you will need the training-data-analyst files and a Cloud Storage bucket.

Verify that the repository files are in Cloud Shell Editor

1. Clone the repository from the Cloud Shell command line:

git clone https://github.com/GoogleCloudPlatform/training-data-analyst

1. You should see the **training-data-analyst** directory.

Verify that you have a Cloud Storage bucket

If you don't have a bucket, you can follow these instructions to create a bucket.

1. In the Console, on the **Navigation menu** (navigation_menu), click **Cloud overview**.
2. **Select and copy** the Project ID. For simplicity you will use the Qwiklabs Project ID, which is already globally unique, as the bucket name.
3. In the Console, on the **Navigation menu** (navigation_menu), click **Cloud Storage** > **Browser**.
4. Click **Create Bucket**.
5. Specify the following, and leave the remaining settings as their defaults:

|  |  |
| --- | --- |
| **Property** | **Value (type value or select option as specified)** |
| **Name** | <your unique bucket name (Project ID)> |
| **Default storage class** | Multi-Regional |
| **Location** | <Your location> |

1. Click **Create**.
2. Record the name of your bucket. You will need it in subsequent tasks.
3. In Cloud Shell enter the following to create an environment variable named "BUCKET" and verify that it exists with the echo command.

BUCKET="<your unique bucket name (Project ID)>"

echo $BUCKET

You can use $BUCKET in Cloud Shell commands. And if you need to enter the bucket name <your-bucket> in a text field in Console, you can quickly retrieve the name with echo $BUCKET.

Ensure that the Dataflow API is properly enabled

1. Execute the following block of code in the Cloud Shell.

gcloud services disable dataflow.googleapis.com --force

gcloud services enable dataflow.googleapis.com

**Task 2. Create a new Dataflow project**

The goal of this lab is to become familiar with the structure of a Dataflow project and learn how to execute a Dataflow pipeline. You will use the powerful build tool [Maven](https://maven.apache.org/) to create a new Dataflow project.

1. Return to the browser tab containing Cloud Shell. In Cloud Shell navigate to the directory for this lab:

cd ~/training-data-analyst/courses/data\_analysis/lab2

1. Copy and paste the following Maven command:

mvn archetype:generate \

-DarchetypeArtifactId=google-cloud-dataflow-java-archetypes-starter \

-DarchetypeGroupId=com.google.cloud.dataflow \

-DgroupId=com.example.pipelinesrus.newidea \

-DartifactId=newidea \

-Dversion="[1.0.0,2.0.0]" \

-DinteractiveMode=false

* What directory has been created?
* What package has been created inside the src directory?

1. Examine the Maven command that was used to create the lab code:

cat ~/training-data-analyst/courses/data\_analysis/lab2/create\_mvn.sh

* What directory will get created?
* What package will get created inside the src directory?

**Task 3. Pipeline filtering**

1. In the Cloud Shell code editor navigate to the directory /training-data-analyst/courses/data\_analysis/lab2.

Then select the path javahelp/src/main/java/com/google/cloud/training/dataanalyst/javahelp/ and view the file Grep.java.

Alternatively, you could view the file with nano editor. **Do not make any changes to the code**.

cd ~/training-data-analyst/courses/data\_analysis/lab2/javahelp/src/main/java/com/google/cloud/training/dataanalyst/javahelp/

nano Grep.java

Can you answer these questions about the file Grep.java?

* What files are being read?
* What is the search term?
* Where does the output go?

There are three apply statements in the pipeline:

* What does the first apply() do?
* What does the second apply() do?
* Where does its input come from?
* What does it do with this input?
* What does it write to its output?
* Where does the output go to?
* What does the third apply() do?

**Task 4. Execute the pipeline locally**

1. In Cloud Shell, paste the following Maven command:

cd ~/training-data-analyst/courses/data\_analysis/lab2

export PATH=/usr/lib/jvm/java-8-openjdk-amd64/bin/:$PATH

cd ~/training-data-analyst/courses/data\_analysis/lab2/javahelp

mvn compile -e exec:java \

-Dexec.mainClass=com.google.cloud.training.dataanalyst.javahelp.Grep

1. The output file will be output.txt. If the output is large enough, it will be sharded into separate parts with names like: output-00000-of-00001. If necessary, you can locate the correct file by examining the file's time.

ls -al /tmp

1. Examine the output file.

cat /tmp/output\*

Does the output seem logical?

**Task 5. Execute the pipeline on the cloud**

1. Copy some Java files to the cloud.

gsutil cp ../javahelp/src/main/java/com/google/cloud/training/dataanalyst/javahelp/\*.java gs://$BUCKET/javahelp

Click *Check my progress* to verify the objective.

Copy Java files to the Cloud

Check my progress

1. Edit the Dataflow pipeline in Grep.java. In the Cloud Shell code editor navigate to the directory /training-data-analyst/courses/data\_analysis/lab2/javahelp/src/main/java/com/google/cloud/training/dataanalyst/javahelp and edit the file Grep.java.

cd ~/training-data-analyst/courses/data\_analysis/lab2/javahelp/src/main/java/com/google/cloud/training/dataanalyst/javahelp

1. Replace Input and Output variables with your Bucket name. These must be the actual value, not the environment variable. Recall your Bucket name:

echo $BUCKET

Replace the variables.

String input = "gs://<YOUR-BUCKET-NAME>/javahelp/\*.java";

String outputPrefix = "gs://<YOUR-BUCKET-NAME>/javahelp/output";

**Note:** Make sure that you *changed* the **input** and **outputPrefix** strings that are already present in the source code (do not copy-and-paste the entire line above because you will then end up with two variables named input).

Example lines before:

String input = "src/main/java/com/google/cloud/training/dataanalyst/javahelp/\*.java";

String outputPrefix = "/tmp/output";

Example lines after edit (use your values):

String input = "gs://qwiklabs-gcp-your-value/javahelp/\*.java";

String outputPrefix = "gs://qwiklabs-gcp-your-value/javahelp/output";

1. Examine the script to submit the Dataflow to the cloud:

cd ~/training-data-analyst/courses/data\_analysis/lab2/javahelp

cat run\_oncloud1.sh

What is the difference between this Maven command and the one to run locally?

1. Submit the Dataflow job to the cloud.

bash run\_oncloud1.sh $DEVSHELL\_PROJECT\_ID $BUCKET Grep

Because this is such a small job, running on the cloud will take significantly longer than running it locally (on the order of 2-3 minutes).

Example completion of command line:

[INFO] ------------------------------------------------------------------------

[INFO] BUILD SUCCESS

[INFO] ------------------------------------------------------------------------

[INFO] Total time: 01:50 min

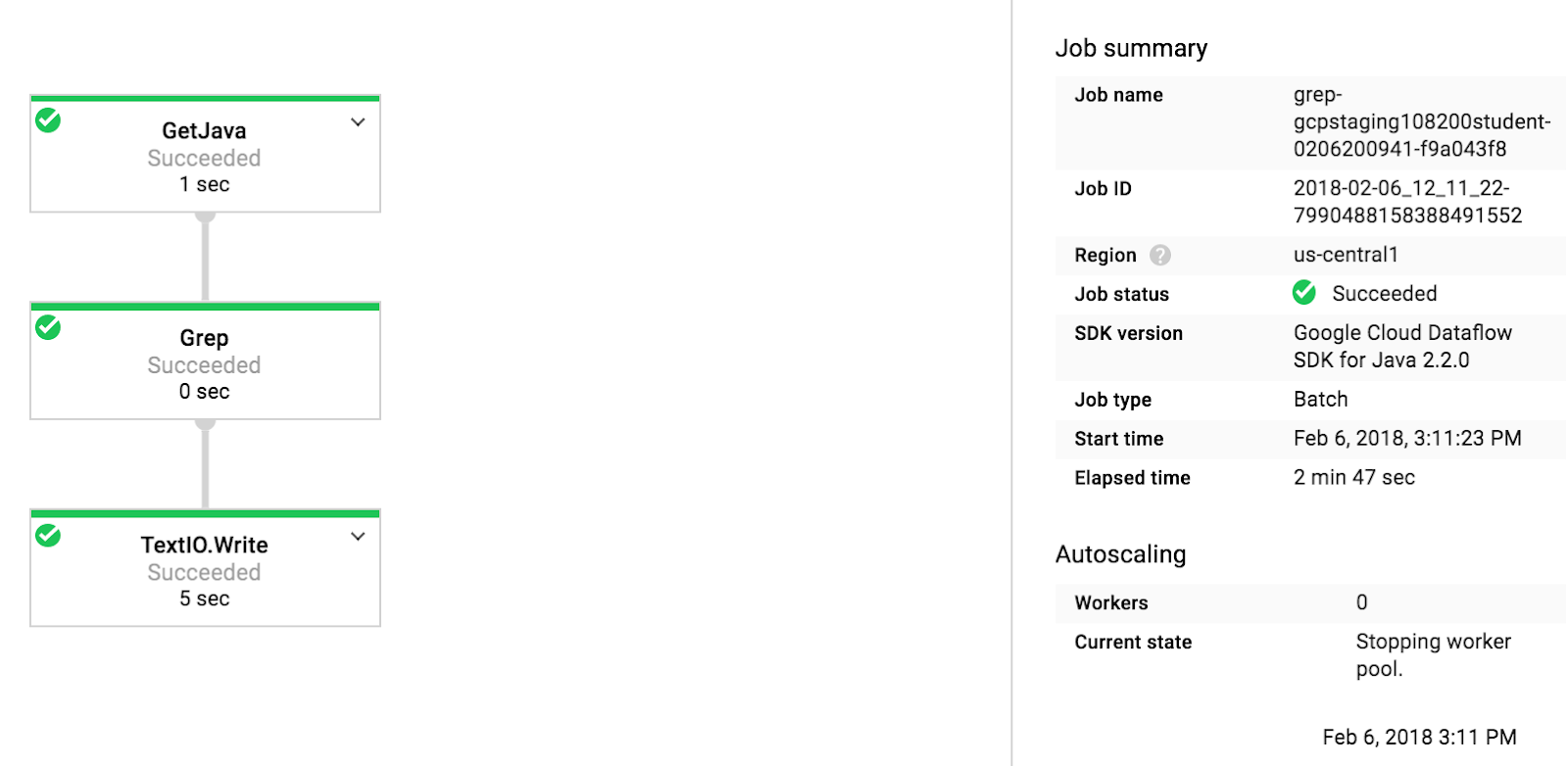
[INFO] Finished at: 2018-02-06T15:11:23-05:00

[INFO] Final Memory: 39M/206M

[INFO] ------------------------------------------------------------------------

1. Return to the browser tab for Console. On the **Navigation menu** (navigation_menu), click **Dataflow** and click on your job to monitor progress.

**Example**:



1. Wait for the job status to turn to **Succeeded**. At this point, your Cloud Shell will display a command-line prompt.**Note:** If Dataflow job fails the first time, then re-run the previous command to submit a fresh Dataflow job to the cloud

Click *Check my progress* to verify the objective.

Submit the Dataflow job to the Cloud

Check my progress

1. Examine the output in the Cloud Storage bucket. On the **Navigation menu** (navigation_menu), click **Cloud Storage > Browser** and click on your bucket.
2. Click the **javahelp** directory. This job will generate the file output.txt. If the file is large enough it will be sharded into multiple parts with names like: output-0000x-of-000y. You can identify the most recent file by name or by the **Last modified** field. Click on the file to view it.

Alternatively, you could download the file in **Cloud Shell** and view it:

gsutil cp gs://$BUCKET/javahelp/output\* .

cat output\*

**End your lab**

**Aggregate with GroupByKey and Combine.**

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

What do you do after the map phase?

The unnamed phase is the shuffle phase where you group together like keys.

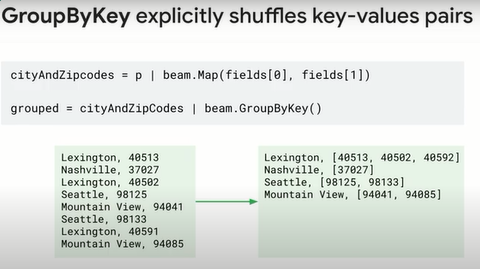
This works on a PCollection of key-value pairs or two element tuples, groups by common key and returns a single key-value pair where the value is actually a group of values.

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 10001 and 10002 zip codes.

You could first create a key-value pair and a pardue and then group by the key.

The resulting key-value pairs are simply two tuples.



We do have to be aware of data skew when we're doing this.

When the same example is scaled up in 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 1 billion you're forcing all of those elements to go to a single workgroup to be counted.

This could definitely run into some issues on the network.

This is the same performance concern when doing high-cardinality GroupBy queries on billions of records in BigQuery.

In this example, there are a million X values and only 1,000 Y values.

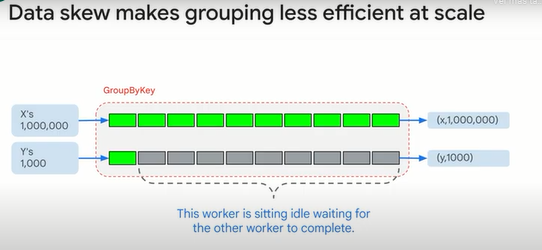
GroupByKey will group all of the X values on one worker.

The worker will take much longer to do its processing on the million values than the other worker which only has 1,000 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 inefficiencies by keeping the data balance.

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



CoGroupByKey is very similar.

It groups results across several PCollections by key.

For example, input KV ends KW output K with an iteration of V and an iteration of W. CoGroupByKey performs a relational join of two or more key-value PCollections that have the same key type.

Now we can move to the reduce phase.

How do we calculate totals or averages or other aggregations on our PCollections?

Combine is used to combine collections of elements or values in your data.

Combine is variants that work on entire PCollection and some that combine the values for each key on PCollections of key-value pairs.

CombineGlobally FN reduces a PCollection to a single value by applying the FN, or the function.

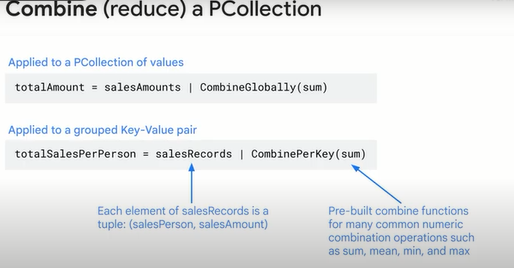
CombinePerKey is similar to GroupByKey but combines the values by a combine function or a callable that takes an iterable action such as sum or max.

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 combine functions for common numeric combination operations such as sum, min and max.

Simple combine operations, such as sum, can usually be implemented as a simple function.

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



The combining function should not 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 sub-class of combine function.

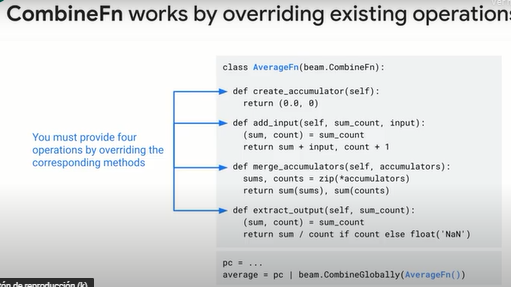
You could use the combine function if the action needed requires a more sophisticated accumulator, must perform additional pre- or post-processing, might change the output type or takes the key into account.

A general combining operation consists of four operations.

When you create a sub-class 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.

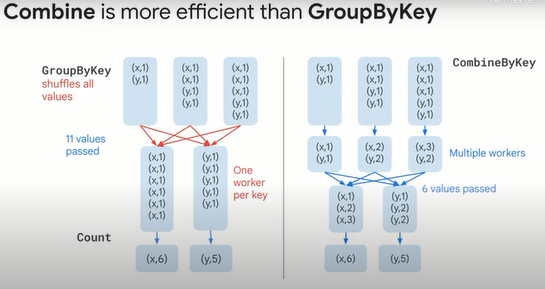
In this example, CombineByKey first aggregates values and then processes the aggregates with multiple workers.

Also only six aggregate values need to be passed over the network.

Combine is a JAVA interface that tells Dataflow that the combine operational account is both commutative and associative.

This allows Dataflow to shard within a key versus 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 works a lot like SQL UNION.

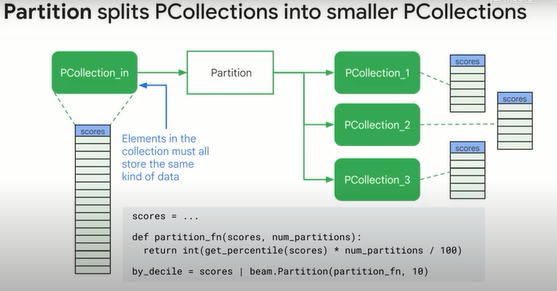
It's a beam transform for PCollection objects that store the same data type.

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 want to calculate percentages or quartiles, and the top quartile has different processing than all the others.



**Lab: MapReduce in Beam (Python) 2.5.**

**Task 1. Lab Preparations**

Specific steps must be completed to successfully execute this lab:

Open the SSH terminal and connect to the training VM

You will be running all code from a curated training VM.

1. In the Console, on the **Navigation menu** (Navigation menu icon), click **Compute Engine** > **VM instances**.
2. Locate the line with the instance called **training-vm**.
3. On the far right, under **Connect**, click on **SSH** to open a terminal window.
4. In this lab, you will enter CLI commands on the **training-vm**.

Clone the training github repository

In the **training-vm** SSH terminal enter the following command:

git clone https://github.com/GoogleCloudPlatform/training-data-analyst

**Task 2. Identify Map and Reduce operations**

1. Return to the **training-vm** SSH terminal and navigate to the directory /training-data-analyst/courses/data\_analysis/lab2/python and view the file is\_popular.py with Nano. **Do not make any changes to the code.** Press **Ctrl+X** to exit Nano.

cd ~/training-data-analyst/courses/data\_analysis/lab2/python

nano is\_popular.py

Can you answer these questions about the file is\_popular.py?

* What custom arguments are defined?
* What is the default output prefix?
* How is the variable output\_prefix in main() set?
* How are the pipeline arguments such as --runner set?
* What are the key steps in the pipeline?
* Which of these steps happen in parallel?
* Which of these steps are aggregations?

**Task 3. Execute the pipeline**

1. In the **training-vm** SSH terminal, run the pipeline locally:

python3 ./is\_popular.py

1. Identify the output file. It should be **output**<suffix> and could be a sharded file.

ls -al /tmp

1. Examine the output file, replacing '-\*' with the appropriate suffix.

cat /tmp/output-\*

**Task 4. Use command line parameters**

1. In the **training-vm** SSH terminal, change the output prefix from the default value:

python3 ./is\_popular.py --output\_prefix=/tmp/myoutput

1. What will be the name of the new file that is written out?
2. Note that we now have a new file in the **/tmp** directory:

ls -lrt /tmp/myoutput\*

**End your lab**

**Lab: Serverless Data Analysis with Beam: MapReduce in Beam (Java).**

**Task 1. Lab Preparations**

Specific steps must be completed to successfully execute this lab:

* Create Cloud Storage bucket (which was complete for you automatically when the lab environment started)
* Clone github [repository](https://github.com/GoogleCloudPlatform/training-data-analyst" \t "_blank) to Cloud Shell

git clone https://github.com/GoogleCloudPlatform/training-data-analyst

**Task 2. Identify Map and Reduce operations**

1. In the Cloud Shell code editor navigate to the directory /training-data-analyst/courses/data\_analysis/lab2/javahelp/src/main/java/com/google/cloud/training/dataanalyst/javahelp and view the file IsPopular.java in the Cloud Shell editor. **Do not make any changes to the code.**

Alternatively, you could view the file with nano. **Do not make any changes to the code.**

cd ~/training-data-analyst/courses/data\_analysis/lab2/javahelp/src/main/java/com/google/cloud/training/dataanalyst/javahelp

nano IsPopular.java

Normally, you would develop this Java code in an Integrated Development Environment such as Eclipse or IntelliJ (not in CloudShell).

Can you answer these questions about the file IsPopular.java?

* What getX() methods are present in the class MyOptions?
* What is the default output prefix?
* How is the variable outputPrefix in main() set?
* What are the key steps in the pipeline?
* Which of these steps happen in parallel?
* Which of these steps are aggregations?

**Task 3. Execute the pipeline**

1. Copy and paste the following Maven command in Cloud Shell:

export PATH=/usr/lib/jvm/java-8-openjdk-amd64/bin/:$PATH

cd ~/training-data-analyst/courses/data\_analysis/lab2/javahelp

mvn compile -e exec:java \

-Dexec.mainClass=com.google.cloud.training.dataanalyst.javahelp.IsPopular

It will take 4-5 mintues to complete the process.

1. Examine the output file:

cat /tmp/output.csv

**Task 4. Use command line parameters**

1. Change the output prefix from the default value:

mvn compile -e exec:java \

-Dexec.mainClass=com.google.cloud.training.dataanalyst.javahelp.IsPopular \

-Dexec.args="--outputPrefix=/tmp/myoutput"

1. What will be the name of the new **.csv** file that is written out?
2. Note that we now have a new file in the **/tmp** directory:

ls -lrt /tmp/\*.csv

**End your lab**

**Side inputs and windows of data.**

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

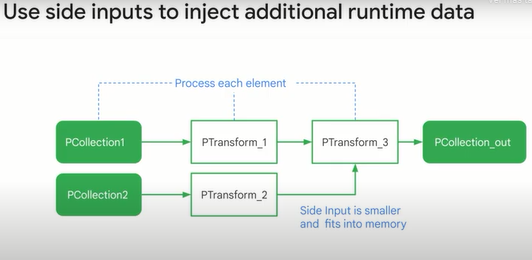
In addition to the main input PCollection, you can provide additional inputs to a ParDo transform in the form of side inputs.

A side input is an additional input that your do function 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 do function 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.



Here's how side inputs work.

This is an example in Python.

This set of steps is actually a subgraph of our overall graph.

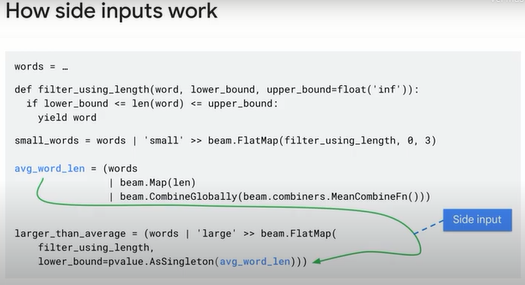
It begins with words that run through the map function to get the length and then combine globally to compute the total lengths across the whole data set.

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.

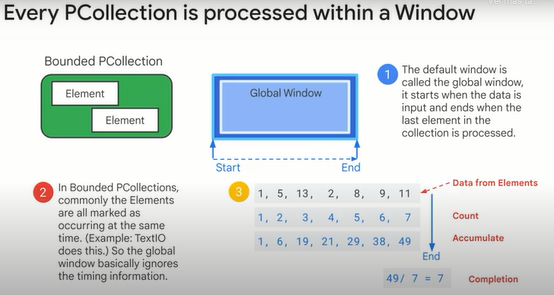
You can add up the value of each element and keep count.

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 data set, there is no predetermined end, so you just keep adding and never break out of the loop and perform the division.



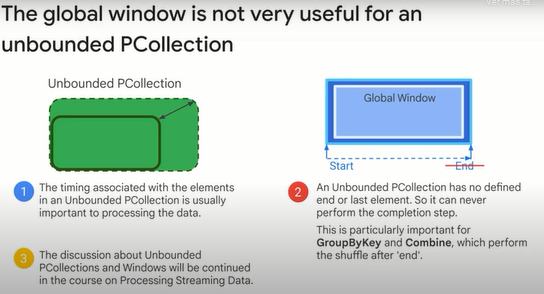
The global window is not very useful for an unbounded PCollection, meaning streaming data.

The timing associated with the elements in 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 Combine, which perform the shuffle after end.

The discussion about unbounded PCollections and Windows will be continued in the course on processing streaming data.



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

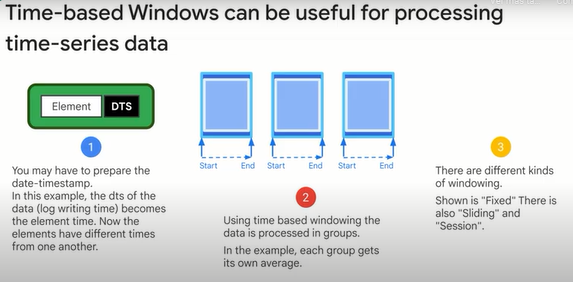
WindowInto(window. GlobalWindows).

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.



For batch inputs, you can group by time as well.

You can explicitly admit a time stamp in your pipeline instead of standard output.

In this example, an offline access log is being read, and the date/time stamp 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(bean.window.

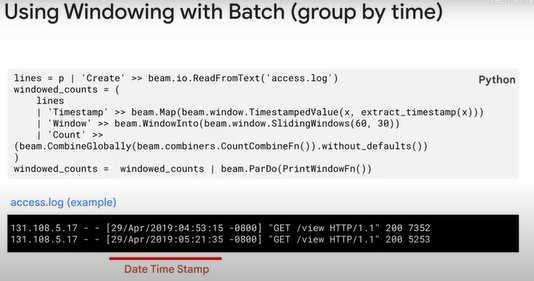
SlidingWindows(60, 30)), which means capture 60 seconds worth of data but start a new window every 30 seconds.

So for example, say you had all of your sales records, and you wanted to compute sales by day.

You'd just extract that time stamp field that represents the time stamp.

Then you would create fixed windows with a 1-day duration, and Dataflow automatically will compute the sum over each window to computer 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.

**Lab: Serverless Data Analysis with Dataflow: Side Inputs (Python).**

**Overview**

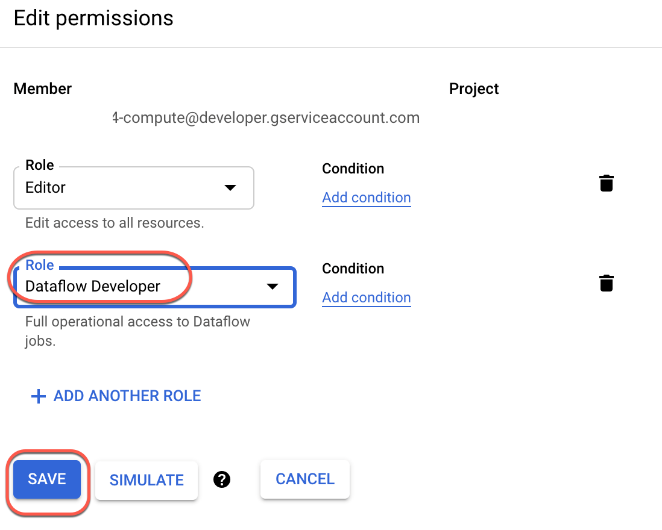
In this lab, you learn how to load data into BigQuery and run complex queries. Next, you will execute a Dataflow pipeline that can carry out Map and Reduce operations, use side inputs and stream into BigQuery.

**Task 1. Preparation**

Assign the Dataflow Developer role

If the account does not have the Dataflow Developer role, follow the steps below to assign the required role.

1. On the **Navigation menu**, click **IAM & Admin** > **IAM**.
2. Select the default compute Service Account {project-number}-compute@developer.gserviceaccount.com.
3. Select the **Edit** option (the pencil on the far right).
4. Click **Add Another Role**
5. Click inside the box for **Select a Role**. In the **Type to filter** selector, type and choose **Dataflow Developer**.
6. Click **Save**.



Ensure that the Dataflow API is successfully enabled

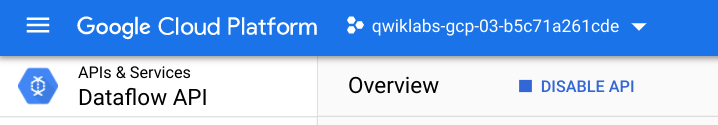
To ensure access to the necessary API, restart the connection to the Dataflow API.

1. In the Cloud Console, enter **Dataflow API** in the top search bar. Click on the result for **Dataflow API**.
2. Click **Manage**.
3. Click **Disable API**.

If asked to confirm, click **Disable**.

1. Click **Enable**.

When the API has been enabled again, the page will show the option to disable.



Open the SSH terminal and connect to the training VM

You will be running all code from a curated training VM.

1. In the Console, on the **Navigation menu** (Navigation menu icon), click **Compute Engine** > **VM instances**.
2. Locate the line with the instance called **training-vm**.
3. On the far right, under **Connect**, click on **SSH** to open a terminal window.
4. In this lab, you will enter CLI commands on the **training-vm**.

Download Code Repository

* Next you will download a code repository for use in this lab. In the **training-vm** SSH terminal enter the following:

git clone https://github.com/GoogleCloudPlatform/training-data-analyst

Create a Cloud Storage bucket

Follow these instructions to create a bucket.

1. In the Console, on the **Navigation menu**, click **Home**.
2. **Select and copy** the Project ID.

For simplicity you will use the Qwiklabs Project ID, which is already globally unique, as the bucket name.

1. In the Console, on the **Navigation menu**, click **Cloud Storage** > **Browser**.
2. Click **Create Bucket**.
3. Specify the following, and leave the remaining settings as their defaults:

|  |  |
| --- | --- |
| **Property** | **Value (type value or select option as specified)** |
| **Name** | <your unique bucket name (Project ID)> |
| **Location type** | Multi-Region |
| **Location** | <Your location> |

1. Click **Create**.

Record the name of your bucket. You will need it in subsequent tasks.

1. In the **training-vm** SSH terminal enter the following to create two environment variables. One named "BUCKET" and the other named "PROJECT". Verify that each exists with the echo command:

BUCKET="<your unique bucket name (Project ID)>"

echo $BUCKET

PROJECT="<your unique project name (Project ID)>"

echo $PROJECT

**Task 2. Try using BigQuery query**

1. In the console, on the **Navigation menu** (Navigation menu icon), click **BigQuery**.
2. If prompted click **Done**.
3. Click **Compose new query** and type the following query:

SELECT

content

FROM

`fh-bigquery.github\_extracts.contents\_java\_2016`

LIMIT

10

Copied!

content\_copy

1. Click on **Run**.

What is being returned?

The BigQuery table fh-bigquery.github\_extracts.contents\_java\_2016 contains the content (and some metadata) of all the Java files present in GitHub in 2016.

1. To find out how many Java files this table has, type the following query and click **Run**:

SELECT

COUNT(\*)

FROM

`fh-bigquery.github\_extracts.contents\_java\_2016`

**Why do you think zero bytes of data were processed to return the result?**

There were 0 records returned in the result.

**BigQuery stores common metadata about the table (like row count). Querying metadata processes 0 bytes.**

This dataset is not properly set up for billing.

Cache is enabled so all queries process 0 bytes.

How many files are there in this dataset?

Is this a dataset you want to process locally or on the cloud?

**Task 3. Explore the pipeline code**

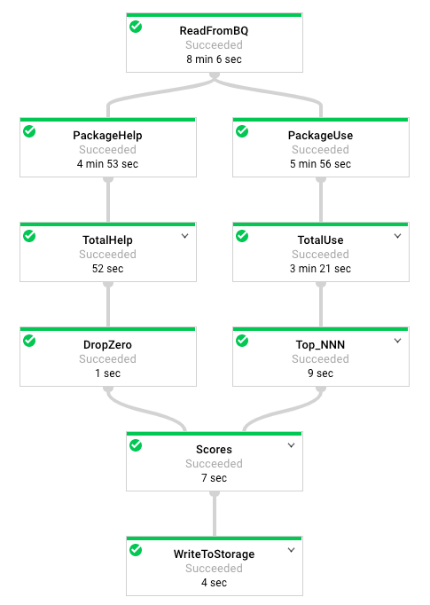
1. Return to the **training-vm** SSH terminal and navigate to the directory /training-data-analyst/courses/data\_analysis/lab2/python and view the file JavaProjectsThatNeedHelp.py.

View the file with Nano. **Do not make any changes to the code.** Press **Ctrl+X** to exit Nano.

cd ~/training-data-analyst/courses/data\_analysis/lab2/python

nano JavaProjectsThatNeedHelp.py

Refer to this diagram as you read the code. The pipeline looks like this:



1. Answer the following questions:

* Looking at the class documentation at the very top, what is the purpose of this pipeline?
* Where does the content come from?
* What does the left side of the pipeline do?
* What does the right side of the pipeline do?
* What does ToLines do? (Hint: look at the content field of the BigQuery result)
* Why is the result of ReadFromBQ stored in a named PCollection instead of being directly passed to another step?
* What are the two actions carried out on the PCollection generated from ReadFromBQ?
* If a file has 3 FIXMEs and 2 TODOs in its content (on different lines), how many calls for help are associated with it?
* If a file is in the package com.google.devtools.build, what are the packages that it is associated with?
* popular\_packages and help\_packages are both named PCollections and both used in the Scores (side inputs) step of the pipeline. Which one is the main input and which is the side input?
* What is the method used in the Scores step?
* What Python data type is the side input converted into in the Scores step?

**Note:**The Java version of this program is slightly different from the Python version. The Java SDK supports AsMap and the Python SDK doesn't. It supports AsDict instead. In Java, the PCollection is converted into a View as a preparatory step before it is used. In Python, the PCollection conversion occurs in the step where it is used.

**Task 4. Execute the pipeline**

1. The program requires BUCKET and PROJECT values and whether you want to run the pipeline locally using --DirectRunner or on the cloud using --DataFlowRunner.
2. Execute the pipeline locally by typing the following into the **training-vm** SSH terminal:

python3 JavaProjectsThatNeedHelp.py --bucket $BUCKET --project $PROJECT --DirectRunner

**Note:**Please ignore the warning if any and move forward.

1. Once the pipeline has finished executing, On the **Navigation menu** (Navigation menu icon), click **Cloud Storage > Browser** and click on your bucket. You will find the results in the **javahelp** folder. Click on the **Result** object to examine the output.
2. Execute the pipeline on the cloud by typing the following into the **training-vm** SSH terminal:

python3 JavaProjectsThatNeedHelp.py --bucket $BUCKET --project $PROJECT --DataFlowRunner

**Note:**Please ignore the warning if any and move forward.

1. Return to the browser tab for Console. On the **Navigation menu** (Navigation menu icon), click **Dataflow** and click on your job to monitor progress.
2. Once the pipeline has finished executing, On the **Navigation menu** (Navigation menu icon) click **Cloud Storage > Browser** and click on your bucket. You will find the results in the **javahelp** folder. Click on the **Result** object to examine the output. The file name will be the same but you will notice that the file creation time is more recent.

Click *Check my progress* to verify the objective.

Execute the pipeline

**End your lab**

**Lab: Serverless Data Analysis with Dataflow: Side Inputs (Java).**

**Overview**

In this lab, you learn how to load data into BigQuery and run complex queries. Next, you will execute a Dataflow pipeline that can carry out Map and Reduce operations, use side inputs and stream into BigQuery.

**Task 1. Preparation**

For this lab, you will need the training-data-analyst files.

Verify that the repository files are in Cloud Shell

1. Clone the repository from the Cloud Shell command line:

git clone https://github.com/GoogleCloudPlatform/training-data-analyst

Copied!

content\_copy

1. Click on the **Refresh** icon in the left navigator panel. You should see the **training-data-analyst** directory.

Verify that you have a Cloud Storage bucket

If you don't have a bucket, you can follow these instructions to create a bucket.

1. In the Console, on the **Navigation menu** (), click **Home**.
2. **Select and copy** the Project ID. For simplicity, you will use the Qwiklabs Project ID, which is already globally unique, as the bucket name.
3. In the Console, on the **Navigation menu** (), click **Cloud Storage** > **Browser**.
4. Click **Create Bucket**.
5. Specify the following, and leave the remaining settings as their defaults:

|  |  |
| --- | --- |
| **Property** | **Value (type value or select option as specified)** |
| **Name** | <your unique bucket name (Project ID)> |
| **Default storage class** | Multi-Region |
| **Location** | <Your location> |

1. Click **Create**.
2. Record the name of your bucket. You will need it in subsequent tasks.
3. In Cloud Shell enter the following to create an environment variable named "BUCKET" and verify that it exists with the echo command.

BUCKET="<your unique bucket name (Project ID)>"

echo $BUCKET

You can use $BUCKET in Cloud Shell commands. And if you need to enter the bucket name <your-bucket> in a text field in Console, you can quickly retrieve the name with echo $BUCKET.

Verify environment variable for your Project ID

1. Cloud Shell creates a default environment variable that contains the current Project ID.

echo $DEVSHELL\_PROJECT\_ID

**Task 2. Try out BigQuery query**

1. Return to the BigQuery web UI. If it is not already open, open [Console](http://console.cloud.google.com/" \t "_blank). On the **Navigation menu** (), click **BigQuery** and then click **Done**.
2. Click **+ Create** to compose new query and type the following query.

SELECT

content

FROM

`fh-bigquery.github\_extracts.contents\_java\_2016`

LIMIT

10

1. Click on **Run**.

What is being returned?

The BigQuery table fh-bigquery.github\_extracts.contents\_java\_2016 contains the content (and some metadata) of all the Java files present in GitHub in 2016.

To find out how many Java files this table has, click **+ Create** to compose new query and type the following query:

SELECT

COUNT(\*)

FROM

`fh-bigquery.github\_extracts.contents\_java\_2016`

Click on **Run**.

**Why do you think zero bytes of data were processed to return the result?**

There were 0 records returned in the result.

**BigQuery stores common metadata about the table (like row count). Querying metadata processes 0 bytes.**

This dataset is not properly setup for billing.

Cache is enabled so all queries process 0 bytes.

How many files are there in this dataset?

Is this a dataset you want to process locally or on the cloud?

**Task 3. Explore the pipeline code**

1. In Cloud Shell editor, or in Cloud Shell, navigate to the lab directory:

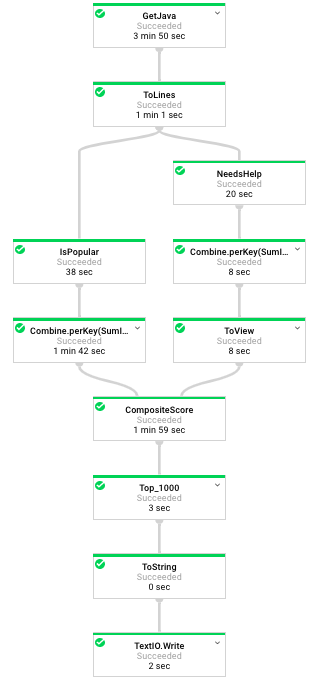
cd ~/training-data-analyst/courses/data\_analysis/lab2

1. View the pipeline code using Cloud Shell editor or nano. **Do not make any changes to the code.**

cd ~/training-data-analyst/courses/data\_analysis/lab2/javahelp

nano src/main/java/com/google/cloud/training/dataanalyst/javahelp/JavaProjectsThatNeedHelp.java

Refer to this diagram as you read the code. The pipeline looks like this:



1. Answer the following questions:

* Looking at the class documentation at the very top, what is the purpose of this pipeline?
* Where does GetJava get Java content from?
* What does ToLines do? (Hint: look at the content field of the BigQuery result)
* Why is the result of ToLines stored in a named PCollection instead of being directly passed to another apply()?
* What are the two actions carried out on javaContent?
* If a file has 3 FIXMEs and 2 TODOs in its content (on different lines), how many calls for help are associated with it?
* If a file is in the package com.google.devtools.build, what are the packages that it is associated with?
* Why is the numHelpNeeded variable not enough? Why do we need to do Sum.integersPerKey()? (Hint: there are multiple files in a package)
* Why is this converted to a View?
* Which operation uses the View as a side input?
* Instead of simply ParDo.of(), this operation uses
* Besides c.element() and c.output(), this operation also makes use of what method in ProcessContext?

**Task 4. Execute the pipeline**

1. Execute the pipeline by typing the following into Cloud Shell.

cd ~/training-data-analyst/courses/data\_analysis/lab2/javahelp

./run\_oncloud3.sh $DEVSHELL\_PROJECT\_ID $BUCKET JavaProjectsThatNeedHelp

Wait until the command is fully executed. It will take around 5 to 7 minutes.

1. Return to the browser tab for Console. On the **Navigation menu** (), click **Dataflow** and click on your job to monitor progress.
2. Once the pipeline has finished executing, download and view the output:

gsutil cp gs://$BUCKET/javahelp/output.csv .

head output.csv

Click *Check my progress* to verify the objective.

**End your lab**

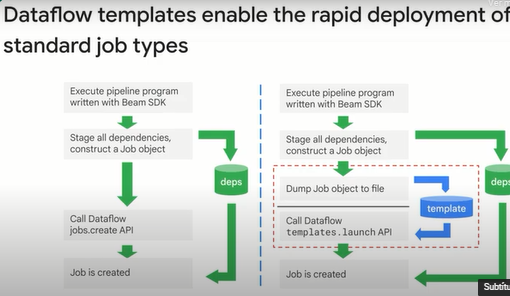
**Creating and re-using pipeline templates.**

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.

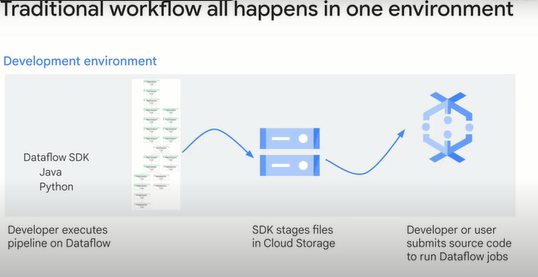
It enables the rapid deployment of standard types of data transformation jobs, removing the need to develop the pipeline code and removing the need to consider the management of components dependencies in the pipeline code.



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 reprocessed entirely or recompiled.

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.



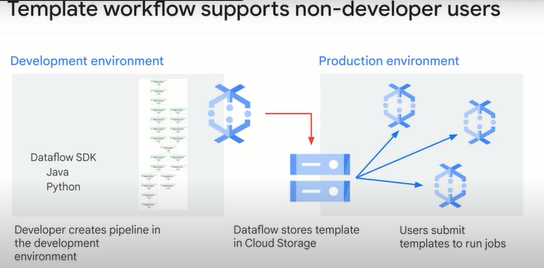
Dataflow Templates enable a new development in execution workflow.

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.

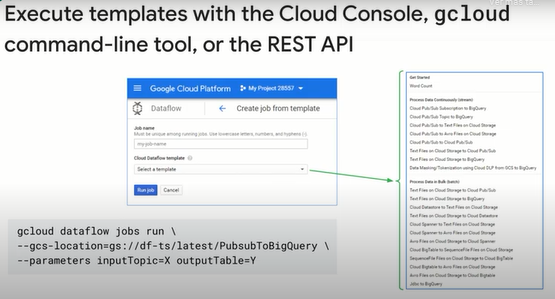
The new approach facilitates the scheduling of batch jobs and opens up more ways for users to submit jobs and more opportunities for automation.



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

You can also run them using the command-line tool or REST API as you see here.

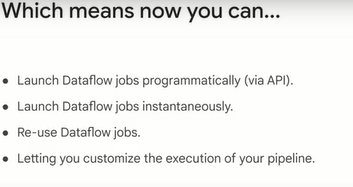
Simply specify the Cloud Storage location of your template that you already have.



Alternatively, you can use the Google provided templates.

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.

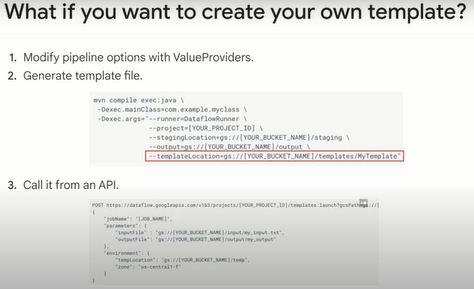


What if you wanted to create your own template?

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.



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.

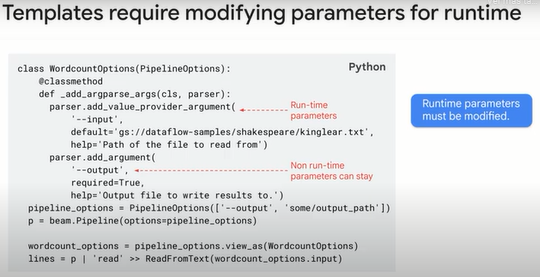
They are compiled time parameters.

To make these values available to non-developer users, they have to be converted to runtime parameters.

These work through the Value Provider interface so that your users can set these values when the template is submitted.

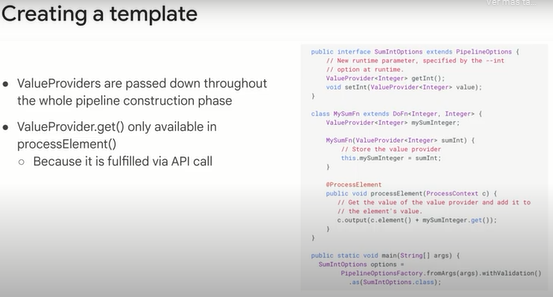
Value Provider can be used in IO, transformations and your functions.

There are also static and nested versions of Value Provider for more complex cases.



This is a Java example for creating your own template.

Note that Value Providers are passed down throughout the whole pipeline construction phase.



Sometimes we need to transform a value from what the user passes at runtime to what a source or sync expects to consume.

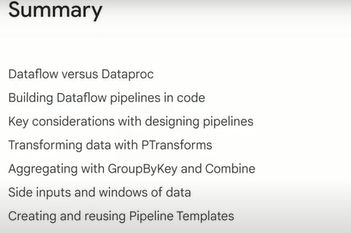
Nested value providers meet this need.

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.

**Summary.**



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 through 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.

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 structured data already living in BigQuery.

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 data set.

To do the actual transformations, you practiced writing PTransforms in your labs.

Remember that the P in PTransforms and PCollections means parallel.

Recall that the PCollection itself is immutable.

Data is never processed in place.

A new PCollection is always created, and the individual elements of a PCollection are massively distributed over many workers to perform the parallel transform.

This is a whole map part of map reduce.

For the reduce part of map reduce, 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 in aggregation.

The pipeline does not have to execute in serial unless you've set it up that way with dependencies.

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

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

**Quizz: Serverless Data Processing with Dataflow.**

1. Which of the following statements are true? (Select all 2 correct responses)

Map operations in a MapReduce can be performed by Combine transforms in Dataflow

Side-inputs in Dataflow are a way to export data from one pipeline to share with another pipeline

**Dataflow executes Apache Beam pipelines**

**Dataflow transforms support both batch and streaming pipelines**

2. Match each of the Dataflow terms with what they do in the life of a dataflow job:

Term Definition

\_\_ 1. Transform A. Output endpoint for your pipeline

\_\_ 2. PCollection B. A data processing operation or step in your pipeline

\_\_ 3. Sink C. A set of data in your pipeline

A-C-B

C-B-A

**B-C-A**

B-A-C