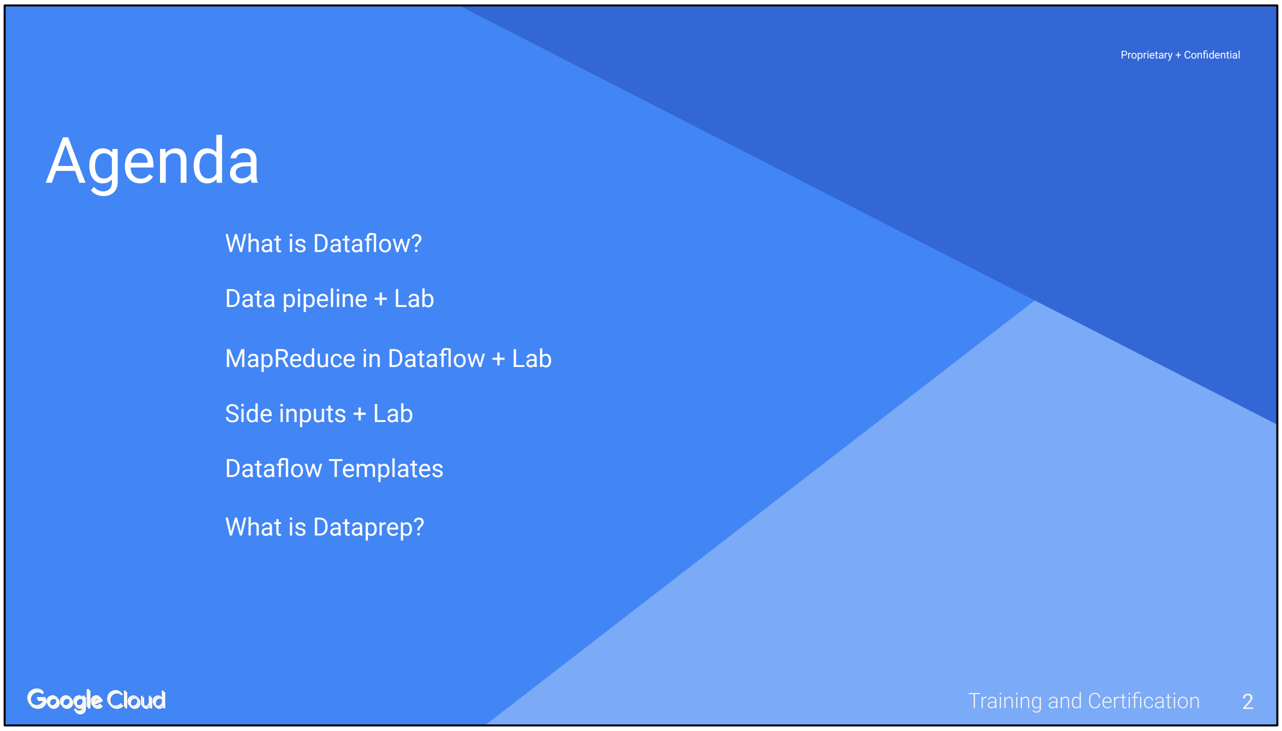
Google Cloud

Autoscaling data processing pipelines

Data Engineering on Google Cloud Platform

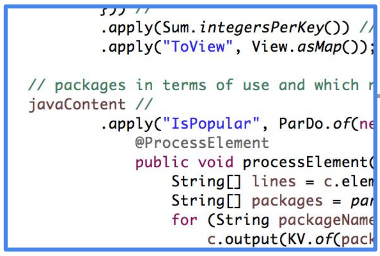
Notes:

30 slides + 3 labs ~ 3 hours

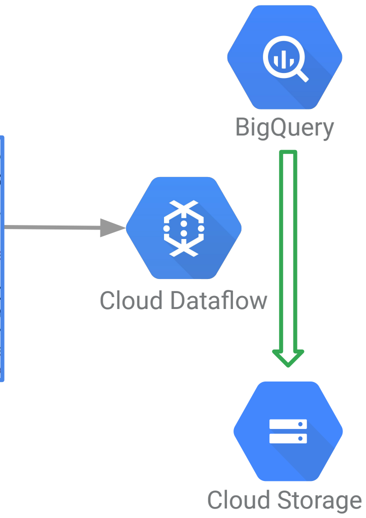


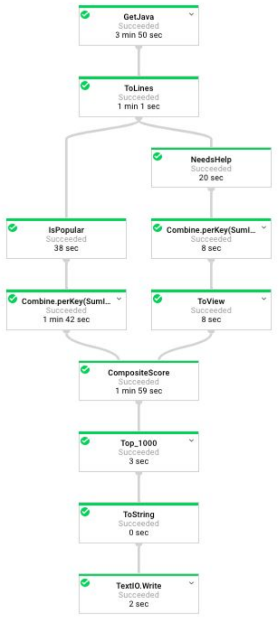
Agenda

Elastic data processing pipeline



```
        }}) //
        .apply(Sum.integersPerKey()) //
        .apply("ToView", View.asMap());

// packages in terms of use and which r
javaContent //
        .apply("IsPopular", ParDo.of(ne
            @ProcessElement
            public void processElement(
                String[] lines = c.elem
                String[] packages = par
                for (String packageName
                    c.output(KV.of(pack
```





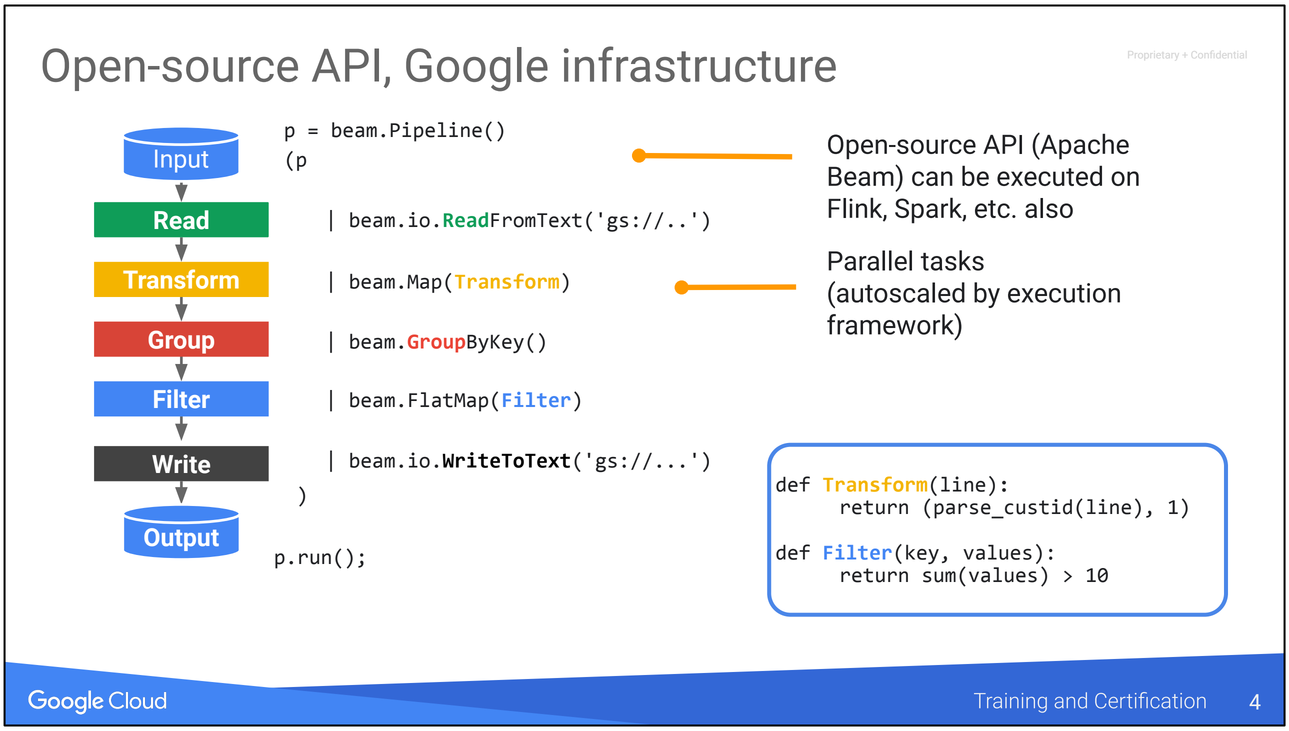
GetJava Succeeded 3 min 50 sec
ToLines Succeeded 1 min 1 sec
NeedsHelp Succeeded 20 sec
IsPopular Succeeded 38 sec
Combine.perKey(Suml... Succeeded 8 sec
Combine.perKey(Suml... Succeeded 1 min 42 sec
ToView Succeeded 8 sec
CompositeScore Succeeded 1 min 59 sec
Top_1000 Succeeded 3 sec
ToString Succeeded 0 sec
TextIO.Write Succeeded 2 sec

Notes:

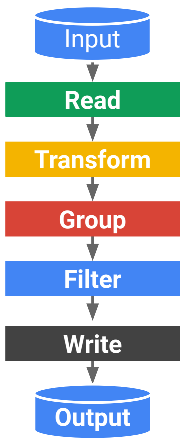
The idea is to write Java code, deploy it to Dataflow which then executes the pipeline.

The pipeline here reads data from BigQuery, does a bunch of processing and writes its output to CloudStorage.

Elastic: unlike Dataproc, there is no need to launch a cluster. Like BigQuery in that respect.



Open-source API, Google infrastructure



```
p = beam.Pipeline()
(p

   | beam.io.ReadFromText('gs://..')

   | beam.Map(Transform)

   | beam.GroupByKey()

   | beam.FlatMap(Filter)

   | beam.io.WriteToText('gs://...')
 )

p.run();
```

Open-source API (Apache Beam) can be executed on Flink, Spark, etc. also

Parallel tasks (autoscaled by execution framework)

```
def Transform(line):
     return (parse_custid(line), 1)

def Filter(key, values):
     return sum(values) > 10
```

Notes:

Distinguish between the API (Apache Beam) and the implementation/execution framework (Dataflow)

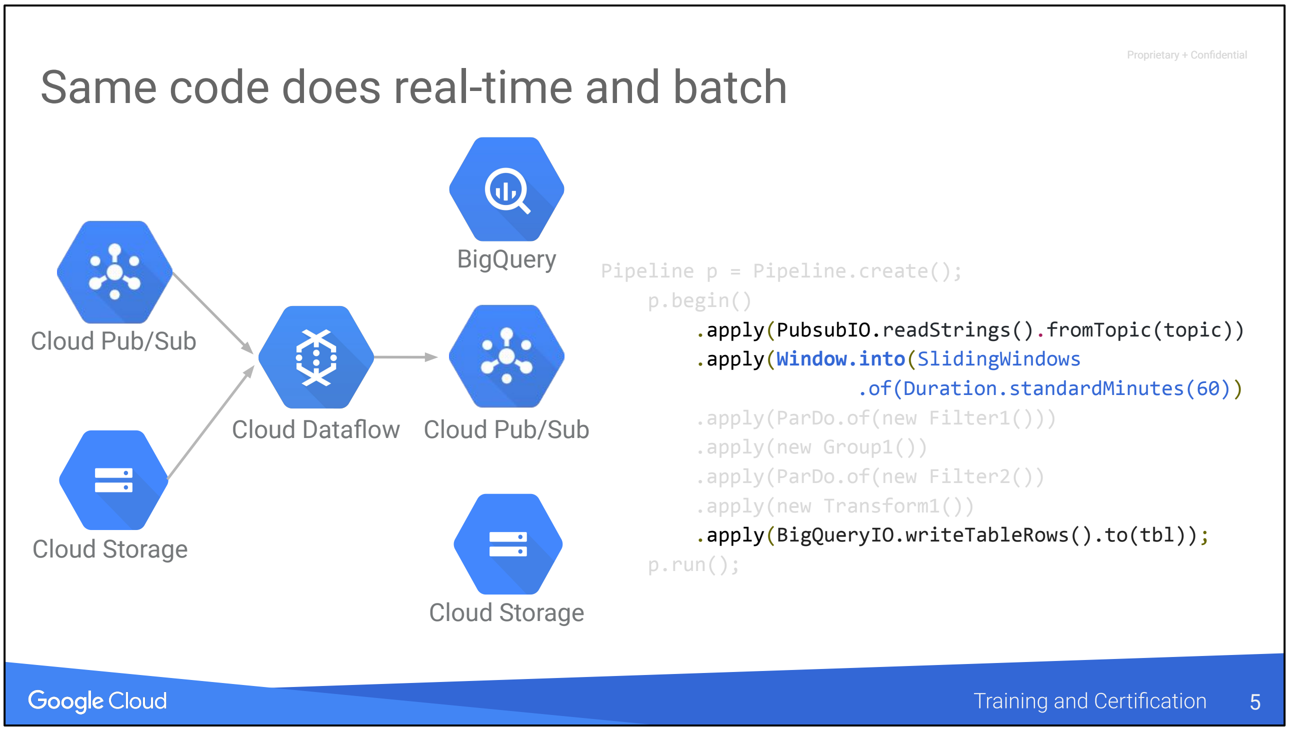
Each step of the pipeline does a filter, group, transform, compare, join, and so on. Transforms can be done in parallel.
c.element() gets the input. c.output() sends the output to the next step of the pipeline.

The example code will write out customers whose IDs appear at least 10 times.

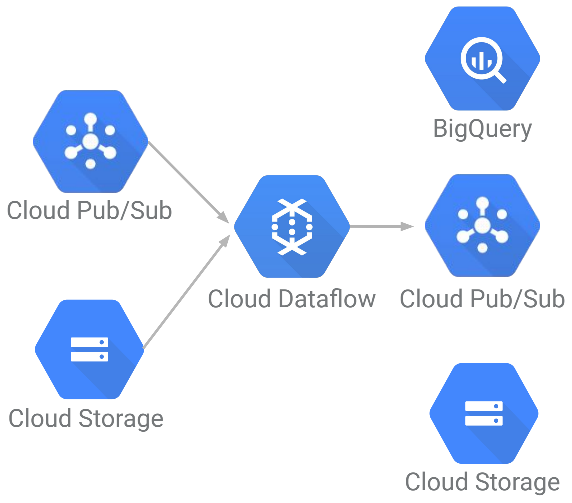
One thing that makes writing Apache Beam easy is that the code written for Beam is similar to how people think of data processing pipelines. Take a look at the pipeline in the center of the slide. This sample Python code analyzes the number of words in lines of text in documents. So, as an input to the pipeline you may want to read text files from Google Cloud Storage.

Then, you transform the data, figure out the number of words in each line of text. This kind of a transformation can be automatically scaled by Dataflow to run in parallel. Next, In your pipeline, you can group lines by the number of words using grouping and other aggregation operations. You can also filter out values, for example to ignore lines with fewer than 10 words. Once all the transformation, grouping, and filtering operations are done, the pipeline writes the result to Google Cloud Storage.

Notice that this implementation separates the pipeline definition from the pipeline execution. All the steps that you see before call to the p.run() method are just defining what the pipeline should do. The pipeline actually gets executed only when you call the run method.



Same code does real-time and batch

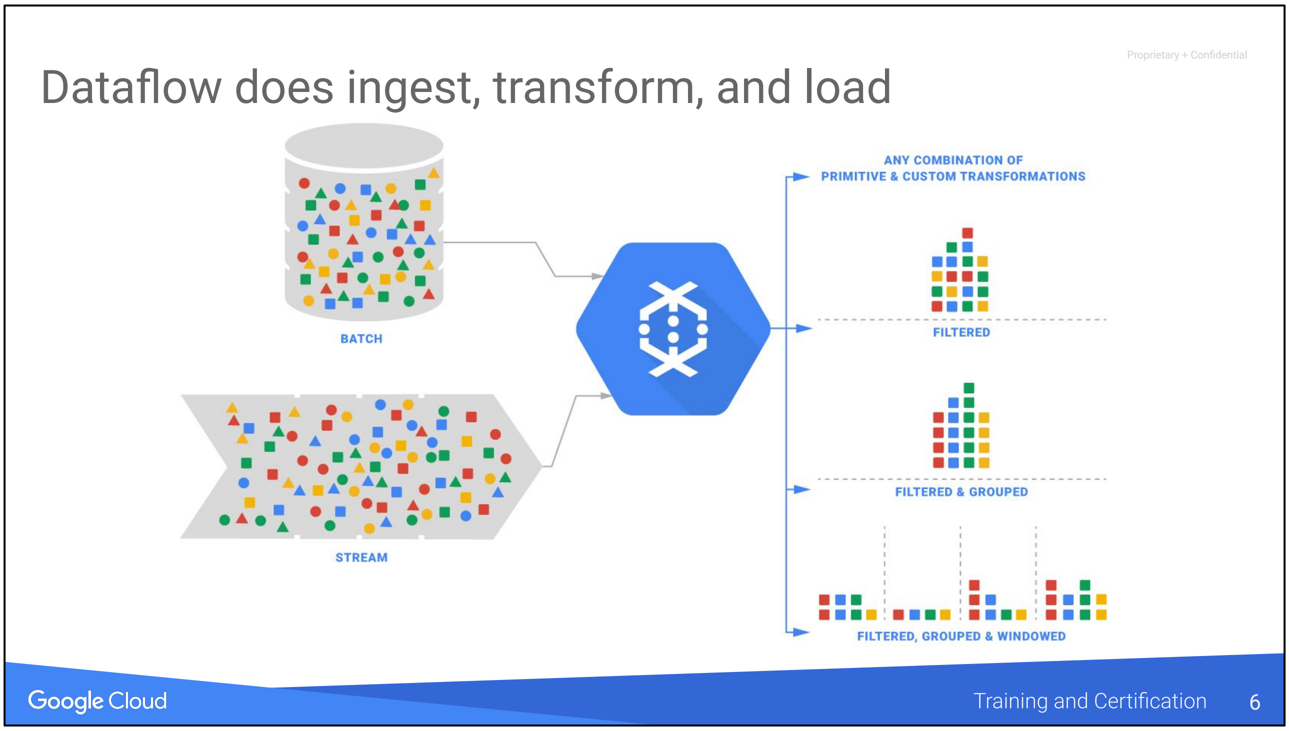


```
Pipeline p = Pipeline.create();
    p.begin()
        .apply(PubsubIO.readStrings().fromTopic(topic))
        .apply(Window.into(SlidingWindows
                    .of(Duration.standardMinutes(60))
        .apply(ParDo.of(new Filter1()))
        .apply(new Group1())
        .apply(ParDo.of(new Filter2())
        .apply(new Transform1())
        .apply(BigQueryIO.writeTableRows().to(tbl));
    p.run();
```

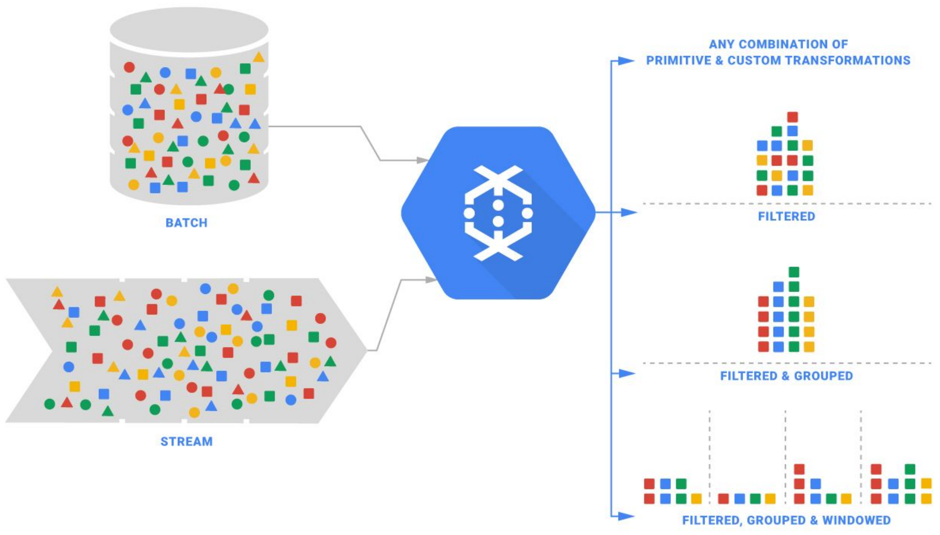
Notes:

You can get input from any of several sources, and you can write output to any of several sinks. The pipeline code remains the same.

You can put this code inside a servlet, deploy it to App Engine, and schedule a cron task queue in App Engine to execute the pipeline periodically.



Dataflow does ingest, transform, and load

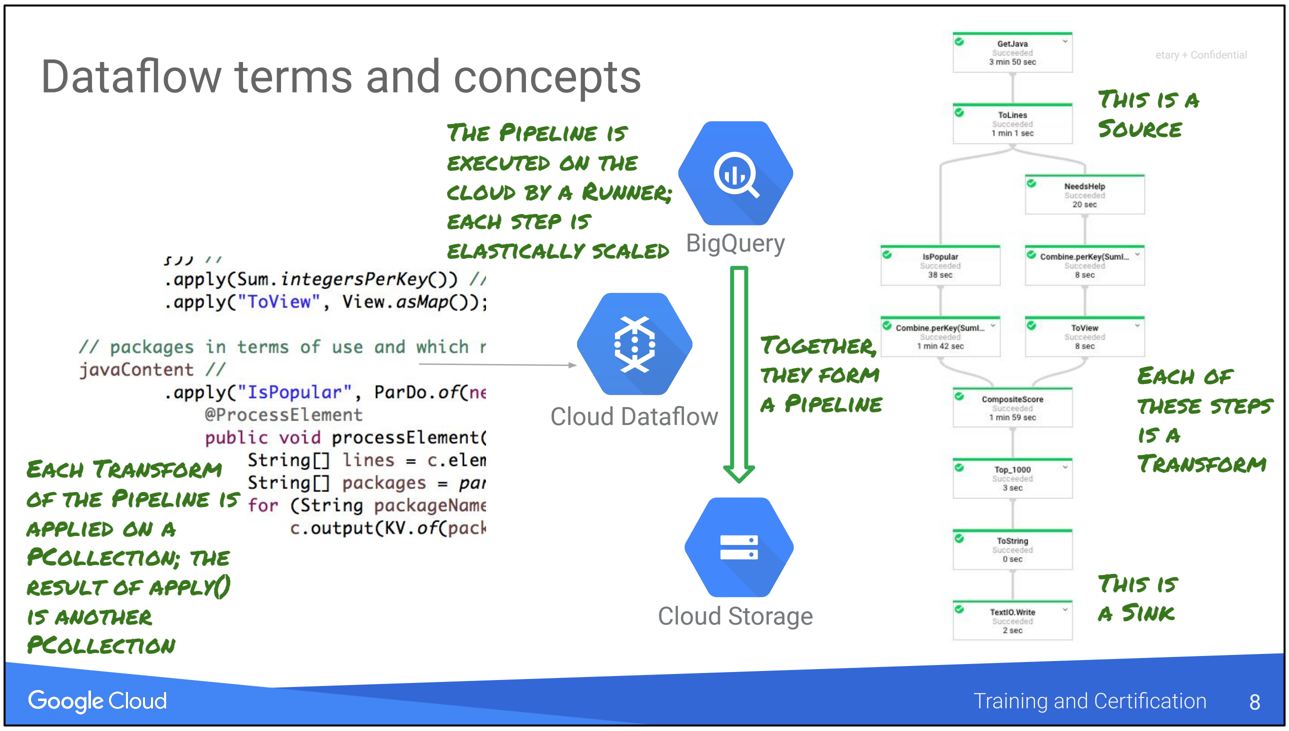

BATCH
STREAM
ANY COMBINATION OF PRIMITIVE & CUSTOM TRANSFORMATIONS
FILTERED
FILTERED & GROUPED
FILTERED, GROUPED & WINDOWED

Notes:

You can replace all the various data handling tools with just Dataflow. Many Hadoop workloads can be done easily and more maintainably with Dataflow. Plus, Dataflow is NoOps.

Agenda

Data pipeline + Lab



Dataflow terms and concepts

```
        ;)) //
        .apply(Sum.integersPerKey()) //
        .apply("ToView", View.asMap());

// packages in terms of use and which r
javaContent //
        .apply("IsPopular", ParDo.of(ne
            @ProcessElement
            public void processElement(
                String[] lines = c.elem
                String[] packages = par
                for (String packageName
                    c.output(KV.of(pack
```

THE PIPELINE IS EXECUTED ON THE CLOUD BY A RUNNER; EACH STEP IS ELASTICALLY SCALED

TOGETHER, THEY FORM A PIPELINE

THIS IS A SOURCE

EACH OF THESE STEPS IS A TRANSFORM

THIS IS A SINK

EACH TRANSFORM OF THE PIPELINE IS APPLIED ON A PCOLLECTION; THE RESULT OF APPLY() IS ANOTHER PCOLLECTION

BigQuery
Cloud Dataflow
Cloud Storage
GetJava Succeeded 3 min 50 sec
ToLines Succeeded 1 min 1 sec
NeedsHelp Succeeded 20 sec
IsPopular Succeeded 38 sec
Combine.perKey(SumI... Succeeded 8 sec
Combine.perKey(SumI... Succeeded 1 min 42 sec
ToView Succeeded 8 sec
CompositeScore Succeeded 1 min 59 sec
Top_1000 Succeeded 3 sec
ToString Succeeded 0 sec
TextIO.Write Succeeded 2 sec
etary + Confidential

Notes:

The idea is to write Java (or Python code), deploy it to Dataflow which then executes the pipeline.

The pipeline here reads data from BigQuery, does a bunch of processing and writes its output to CloudStorage.

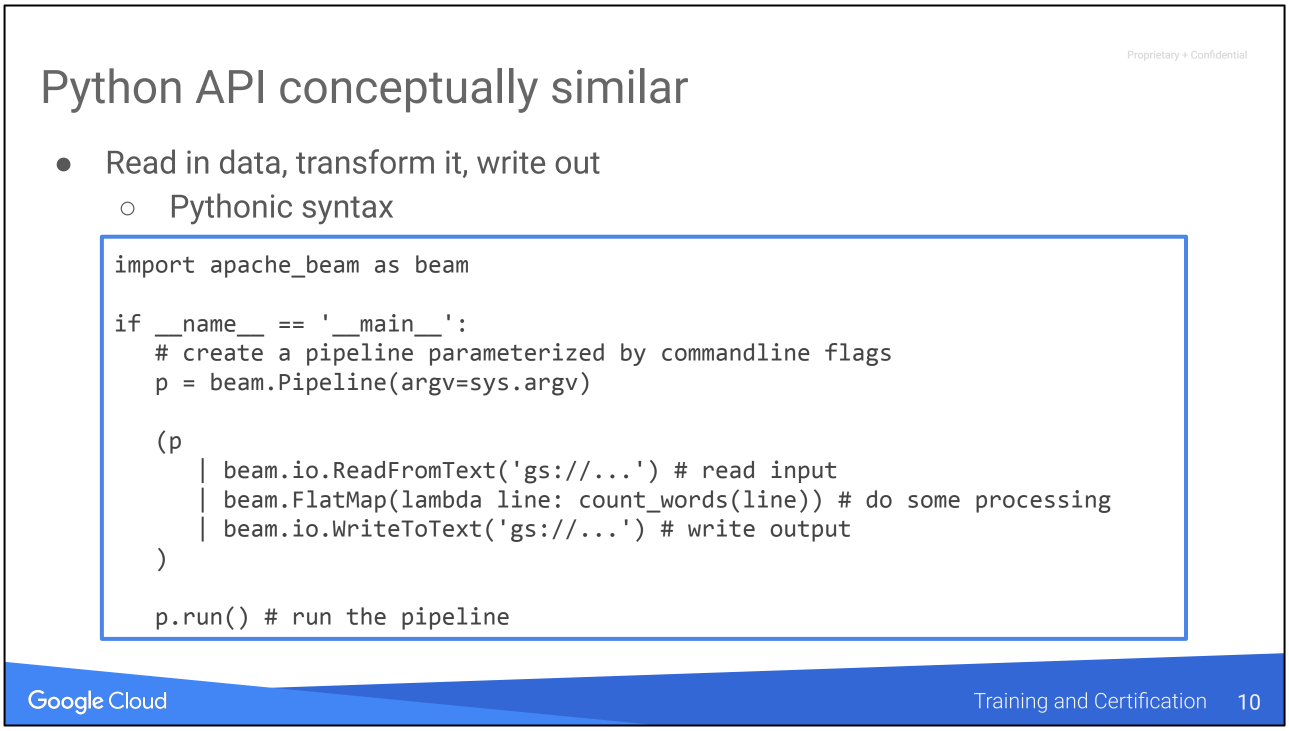
Elastic: unlike Dataproc, there is no need to launch a cluster. Like BigQuery in that respect

Key concepts to be familiar with in Dataflow are highlighted in bold. Start at top-right and work your way clockwise through the callouts.

A Pipeline is a directed graph of steps

- Read in data, transform it, write out
  - Can branch, merge, use if-then statements, etc.

```
import org.apache.beam.sdk.Pipeline;  // etc.

public static void main(String[] args) {
    // Create a pipeline parameterized by commandline flags.
    Pipeline p = Pipeline.create(PipelineOptionsFactory.fromArgs(args));

    p.apply(TextIO.read().from("gs://..."))   // Read input.
     .apply(new CountWords())               // Do some processing.
     .apply(TextIO.write().to("gs://..."));   // Write output.

    // Run the pipeline.
    p.run();
}
```



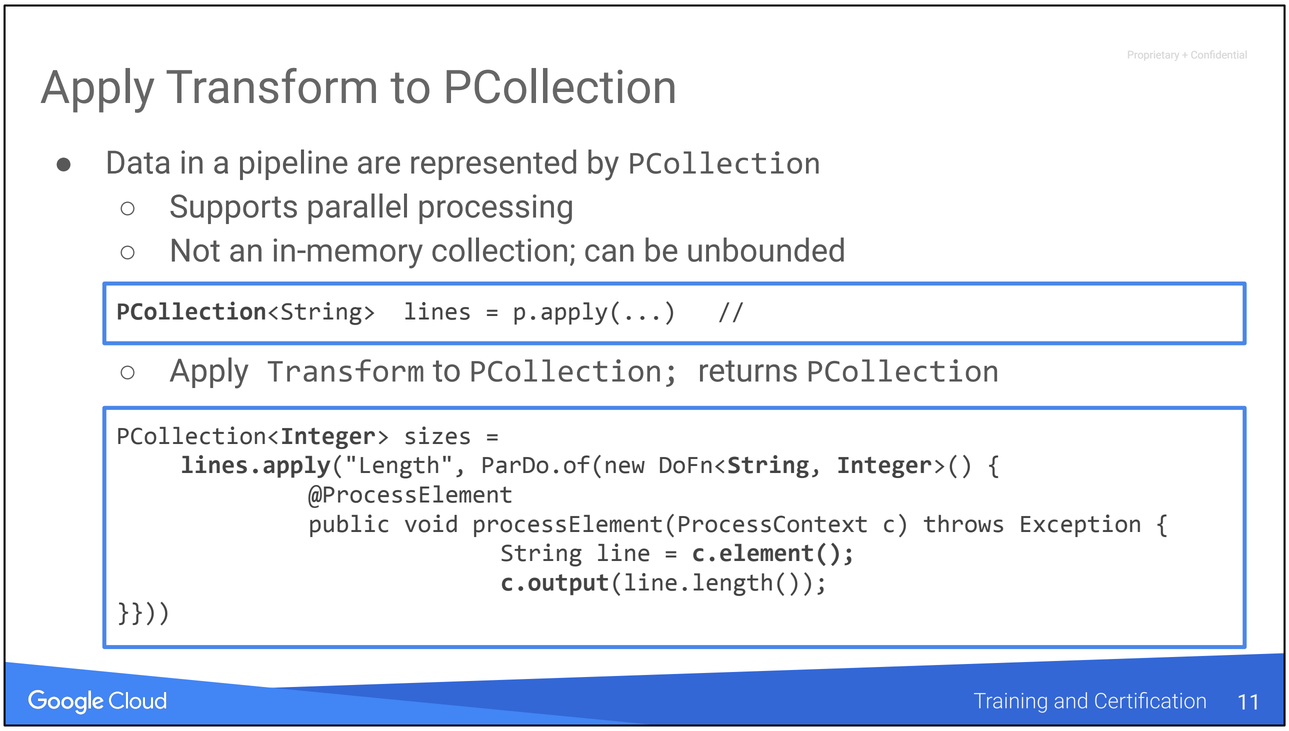
Python API conceptually similar

- Read in data, transform it, write out
  - Pythonic syntax

```
import apache_beam as beam

if __name__ == '__main__':
   # create a pipeline parameterized by commandline flags
   p = beam.Pipeline(argv=sys.argv)

   (p
      | beam.io.ReadFromText('gs://...') # read input
      | beam.FlatMap(lambda line: count_words(line)) # do some processing
      | beam.io.WriteToText('gs://...') # write output
   )

   p.run() # run the pipeline
```

Notes:

| operator overloaded to mean .apply()
>> overload to mean “assign-this-name” to this PTransform is omitted here and introduced on next slide.



Proprietary + Confidential

Apply Transform to PCollection

- Data in a pipeline are represented by `PCollection`
  - Supports parallel processing
  - Not an in-memory collection; can be unbounded

```
PCollection<String>  lines = p.apply(...)   //
```

  - Apply `Transform` to `PCollection;` returns `PCollection`

```
PCollection<Integer> sizes =
     lines.apply("Length", ParDo.of(new DoFn<String, Integer>() {
              @ProcessElement
              public void processElement(ProcessContext c) throws Exception {
                            String line = c.element();
                            c.output(line.length());
}}))
```

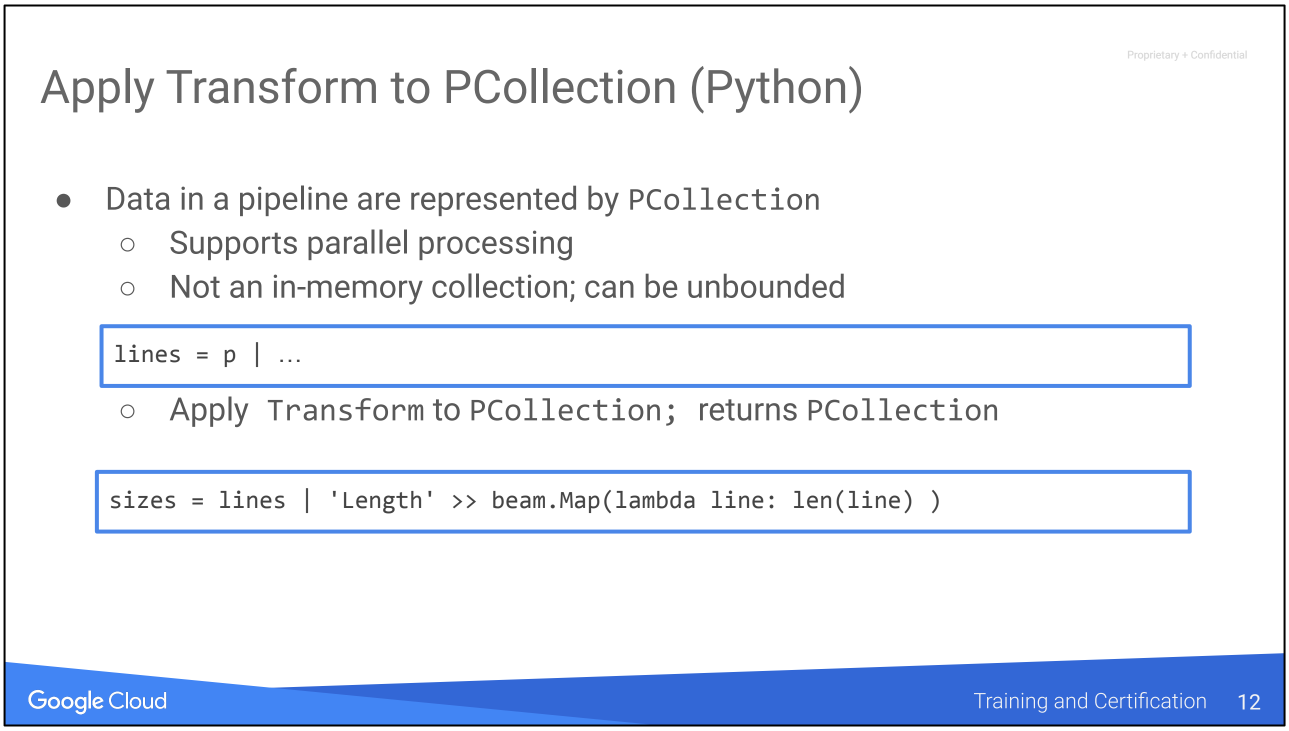
Google Cloud   Training and Certification   11

Notes:

The key thing is that PCollection is not an in-memory collection (it can even be unbounded)

In this case, we take in a String (c.element()) and return a Integer (c.output()) that are then provided to next step in the pipeline one by one

PCollections belong to the pipeline in which they are created (can not be shared)



Apply Transform to PCollection (Python)

- Data in a pipeline are represented by `PCollection`
  - Supports parallel processing
  - Not an in-memory collection; can be unbounded

```
lines = p | …
```

  - Apply `Transform` to `PCollection;` returns `PCollection`

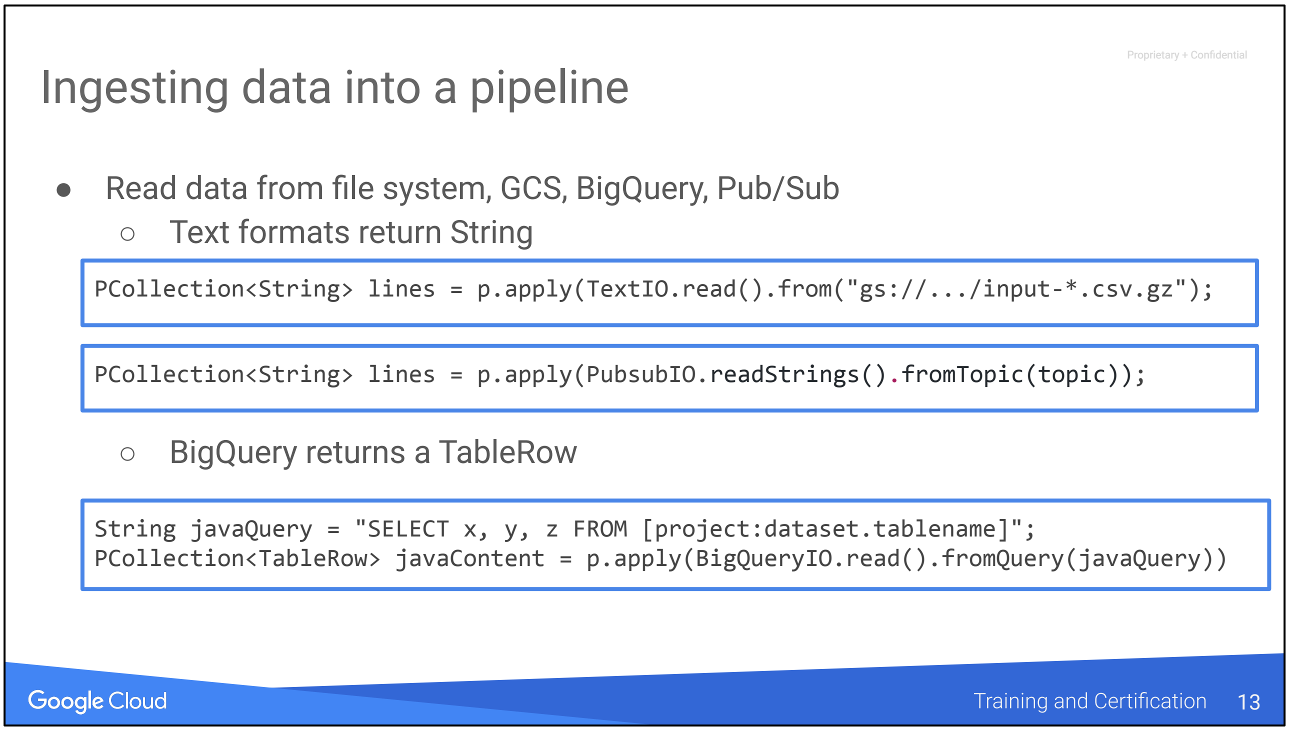
```
sizes = lines | 'Length' >> beam.Map(lambda line: len(line) )
```

Notes:

Java on previous slide; Python on this slide.

In Python, do not use apply(). Best practice is to use the pipe operator.

Notice how we supply the name ‘Length’ to the bottom transform.



Ingesting data into a pipeline

- Read data from file system, GCS, BigQuery, Pub/Sub
  - Text formats return String

```
PCollection<String> lines = p.apply(TextIO.read().from("gs://.../input-*.csv.gz");
```

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

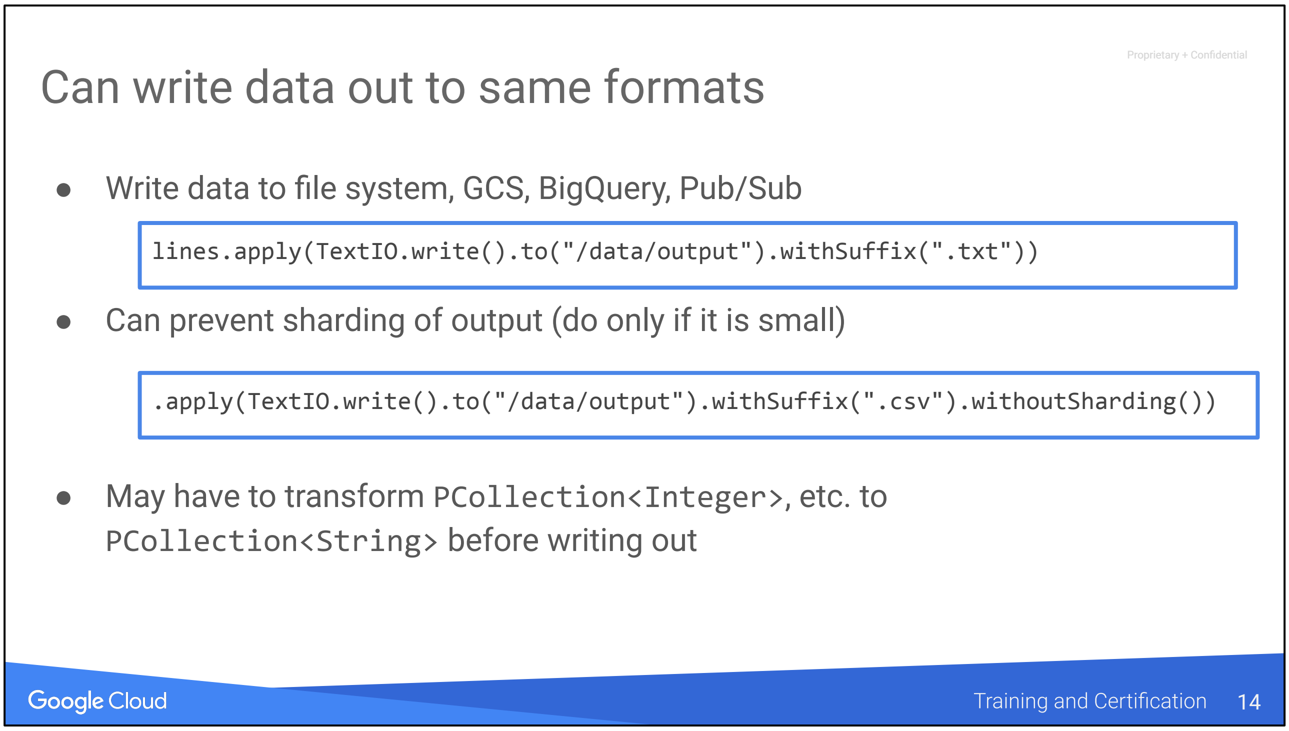
  - BigQuery returns a TableRow

```
String javaQuery = "SELECT x, y, z FROM [project:dataset.tablename]";
PCollection<TableRow> javaContent = p.apply(BigQueryIO.read().fromQuery(javaQuery))
```

Notes:

Notice the wildcards and .gz extension -- both are supported.

There is also I/O to Bigtable, but it's not part of the Dataflow SDK



Can write data out to same formats

- Write data to file system, GCS, BigQuery, Pub/Sub

```
lines.apply(TextIO.write().to("/data/output").withSuffix(".txt"))
```

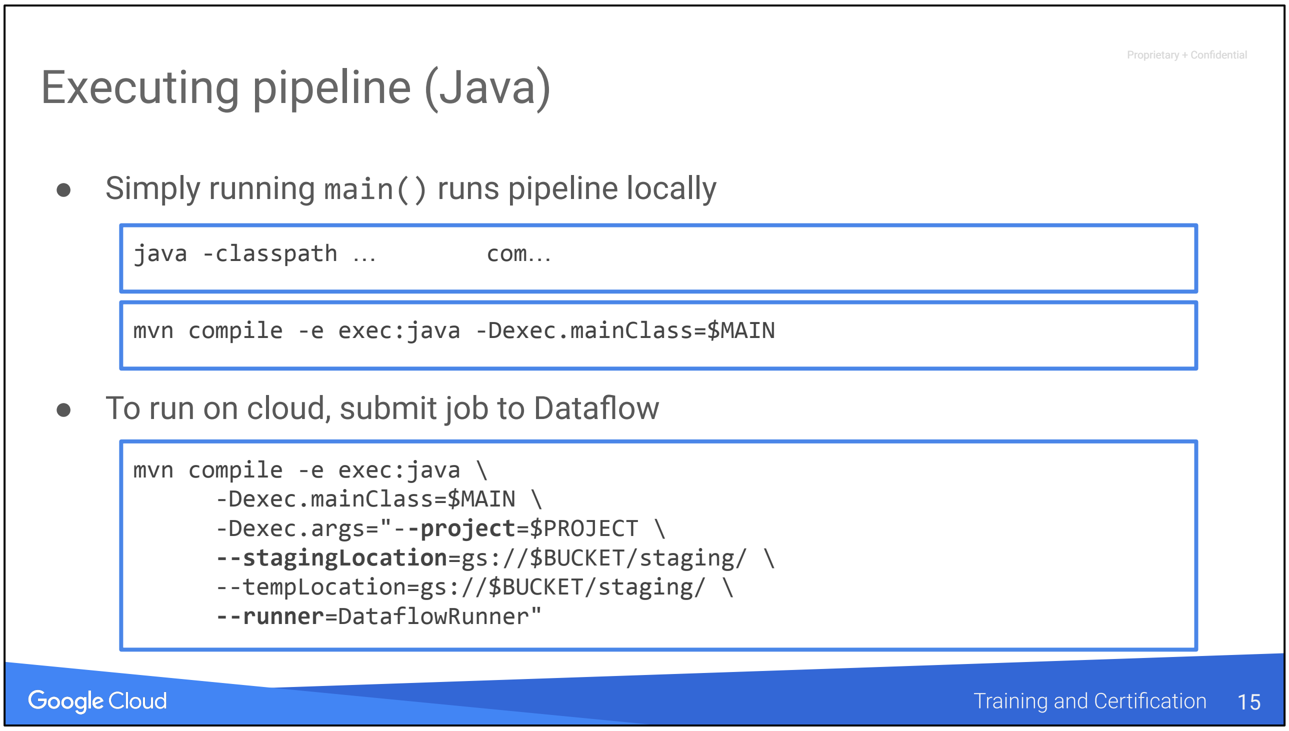
- Can prevent sharding of output (do only if it is small)

```
.apply(TextIO.write().to("/data/output").withSuffix(".csv").withoutSharding())
```

- May have to transform `PCollection<Integer>`, etc. to `PCollection<String>` before writing out

Notes:

Normally, you'll get /data/output-0000-of-0010.txt



Executing pipeline (Java)

- Simply running `main()` runs pipeline locally

```
java -classpath ...        com...
```

```
mvn compile -e exec:java -Dexec.mainClass=$MAIN
```

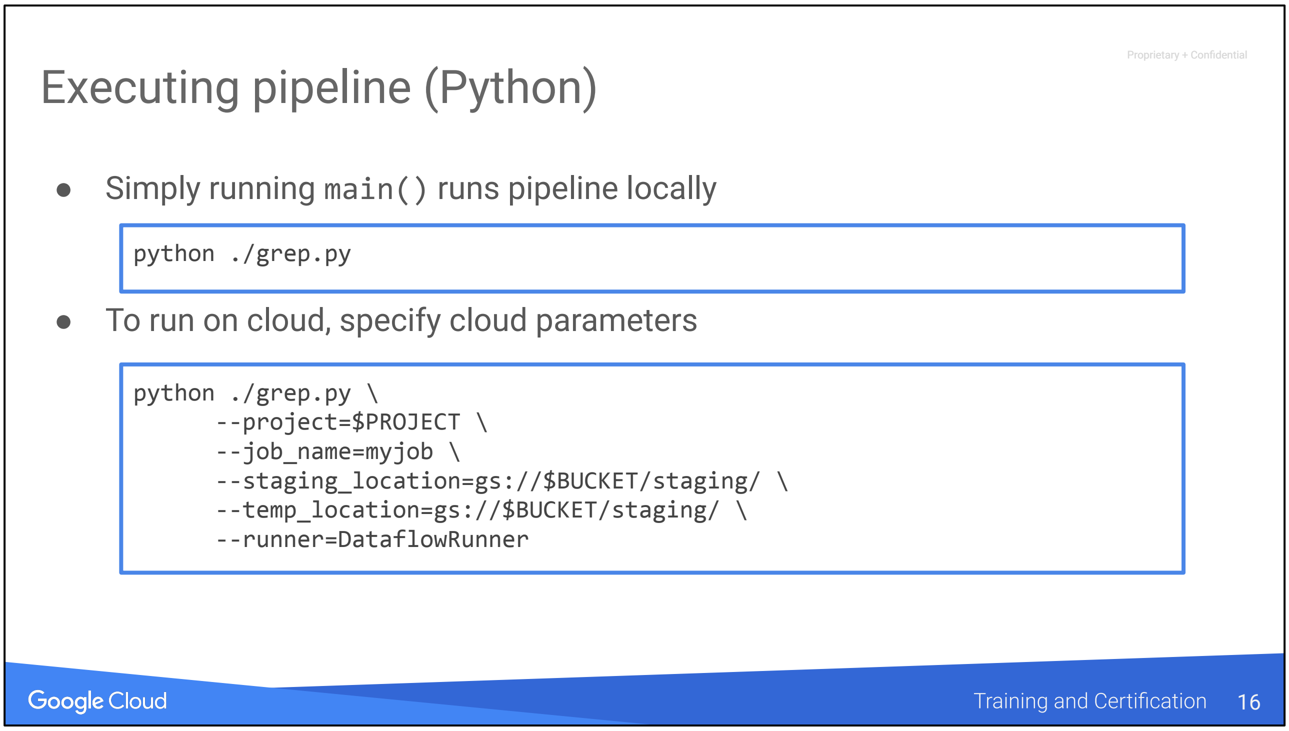
- To run on cloud, submit job to Dataflow

```
mvn compile -e exec:java \
      -Dexec.mainClass=$MAIN \
      -Dexec.args="--project=$PROJECT \
      --stagingLocation=gs://$BUCKET/staging/ \
      --tempLocation=gs://$BUCKET/staging/ \
      --runner=DataflowRunner"
```

Notes:

Run using java and specifying classpath etc. or use mvn

Specify project for billing and staging, temporary locations to store intermediate output, and runner as Dataflow.



Executing pipeline (Python)

- Simply running `main()` runs pipeline locally

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

- To run on cloud, specify cloud parameters

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

Notes:

Conceptually similar to Java.

Syntax is pythonic: --staging_location instead of --stagingLocation etc.

Serverless Data Analysis with Dataflow

Lab 1: A Simple Dataflow Pipeline

In this lab, you will learn how to:

- Set up a Dataflow project
- Write a simple pipeline
- Execute the pipeline on the local machine
- Execute the pipeline on the cloud

```
p //
                .apply("GetJava", TextIO.
                .apply("Grep", ParDo.of(n
                        @ProcessElement
                        public void proce
                                String li
                                if (line.
                                        c
                                }
                        }
                })) //
                .apply(TextIO.write().to(
```

Notes:

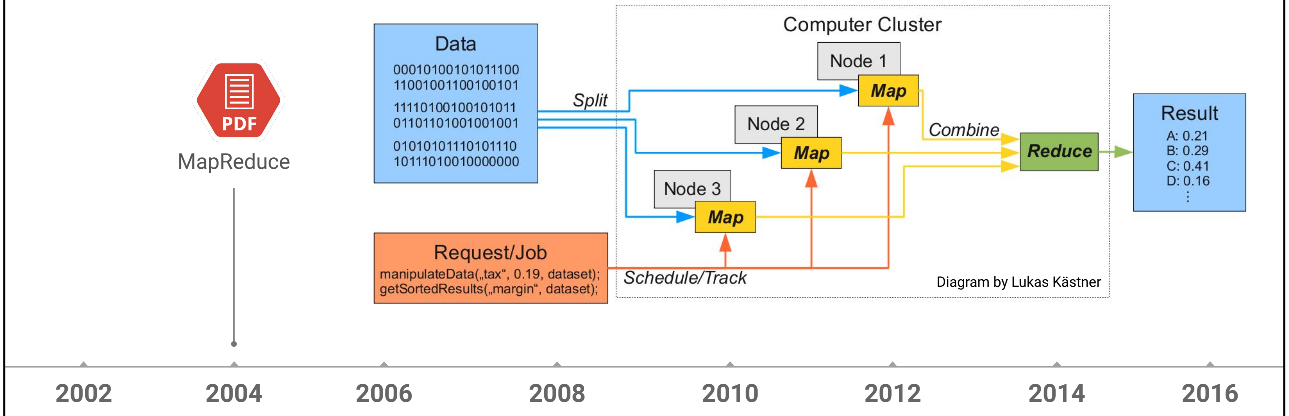
The pipeline they will build does a Grep -- looks for lines in Java files that have the keyword “import” in them.

Image (cc0) https://pixabay.com/en/sieve-icing-sugar-kitchen-help-1262922/

Agenda

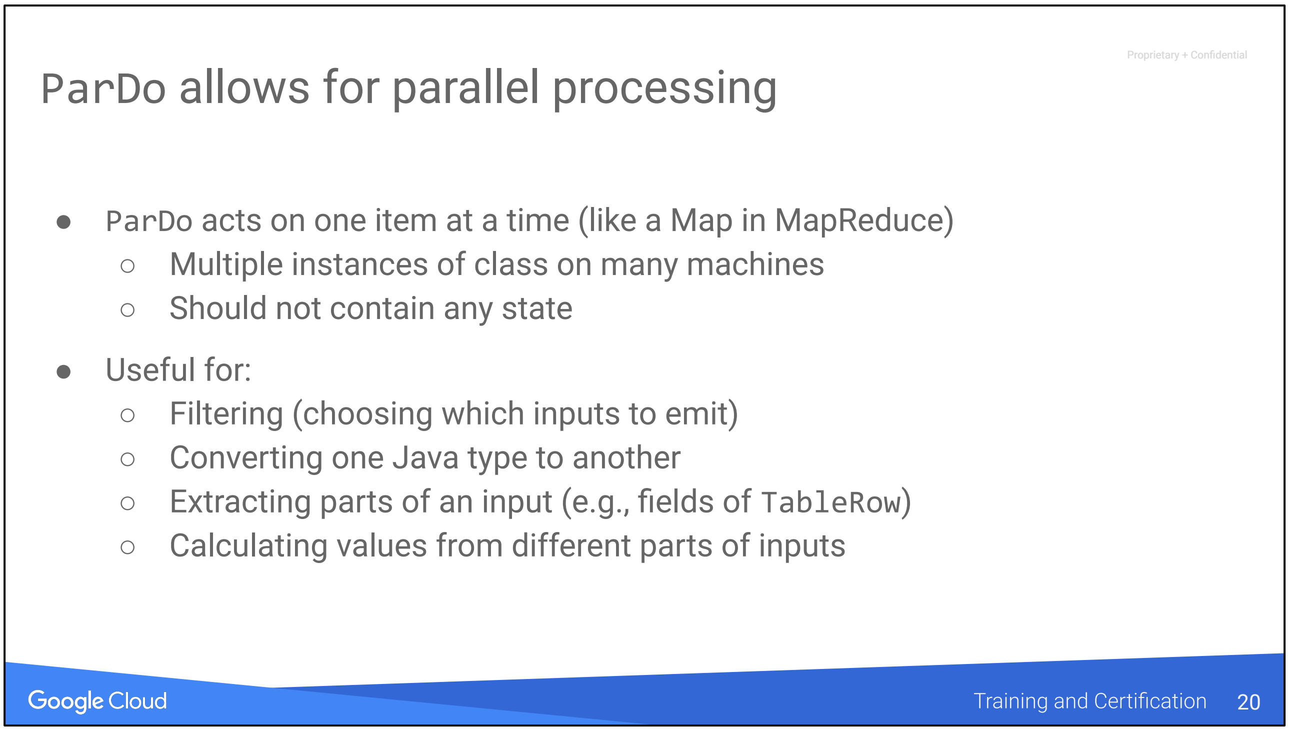
MapReduce in Dataflow + Lab

MapReduce approach splits Big Data so that each compute node processes data local to it



Notes:

Diagram source: https://www.flickr.com/photos/lkaestner/4861146813
cc-by-saLukas Kastner



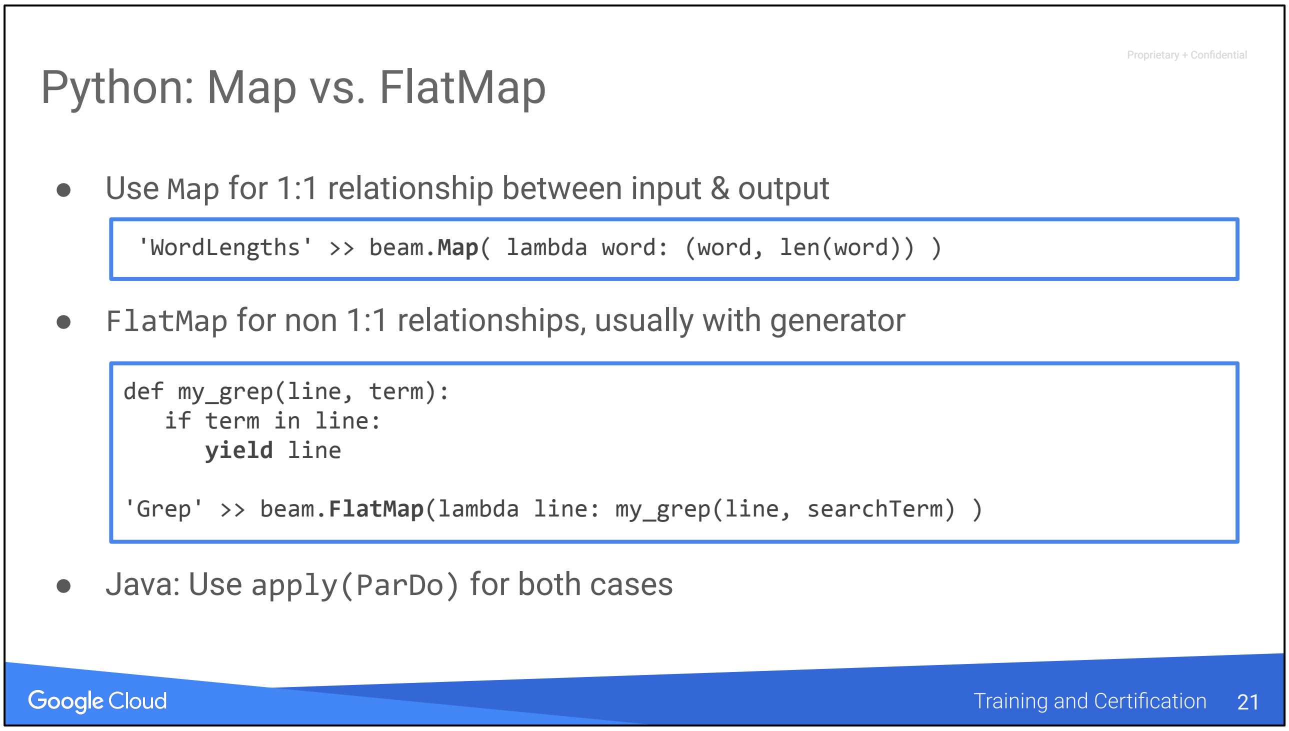
ParDo allows for parallel processing

- `ParDo` acts on one item at a time (like a Map in MapReduce)
  - Multiple instances of class on many machines
  - Should not contain any state
- Useful for:
  - Filtering (choosing which inputs to emit)
  - Converting one Java type to another
  - Extracting parts of an input (e.g., fields of `TableRow`)
  - Calculating values from different parts of inputs

Notes:

You may want to start off by saying that a MapReduce framework consists of Map, followed by shuffle, followed by Reduce. Here, the ParDo does the map operations. This way, there is no confusion between the use of Map on this slide and the Python class `Map` on the next slide.

You can do anything with a single input “row”, but no combination (no persistent state!)
ParDo is not quite a `Map` as in Python, since it can output 0-N elements.



Python: Map vs. FlatMap

- Use `Map` for 1:1 relationship between input & output

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

- `FlatMap` for non 1:1 relationships, usually with generator

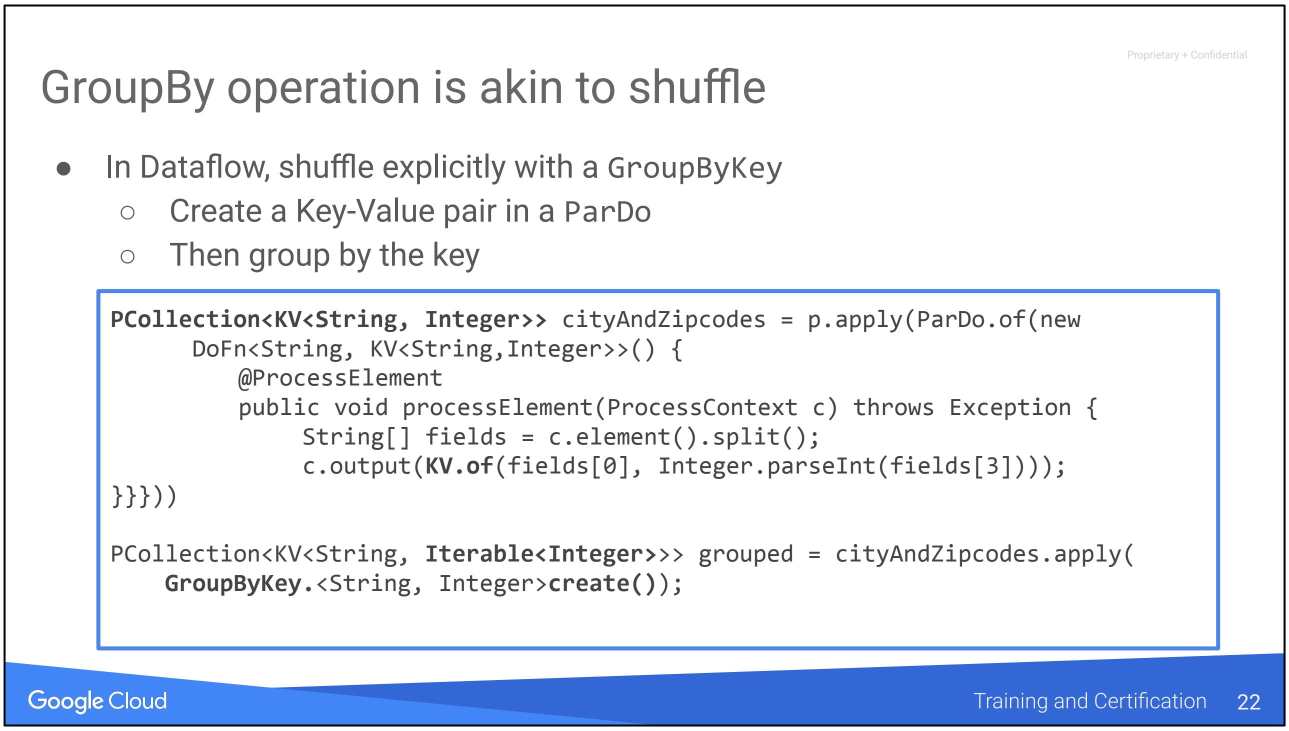
```
def my_grep(line, term):
   if term in line:
      yield line

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

- Java: Use `apply(ParDo)` for both cases

Notes:

The Map example returns a key-value pair (in Python this is simply a 2-tuple) for each word.

The FlatMap example yields the line only for lines that contain the searchTerm.



GroupBy operation is akin to shuffle

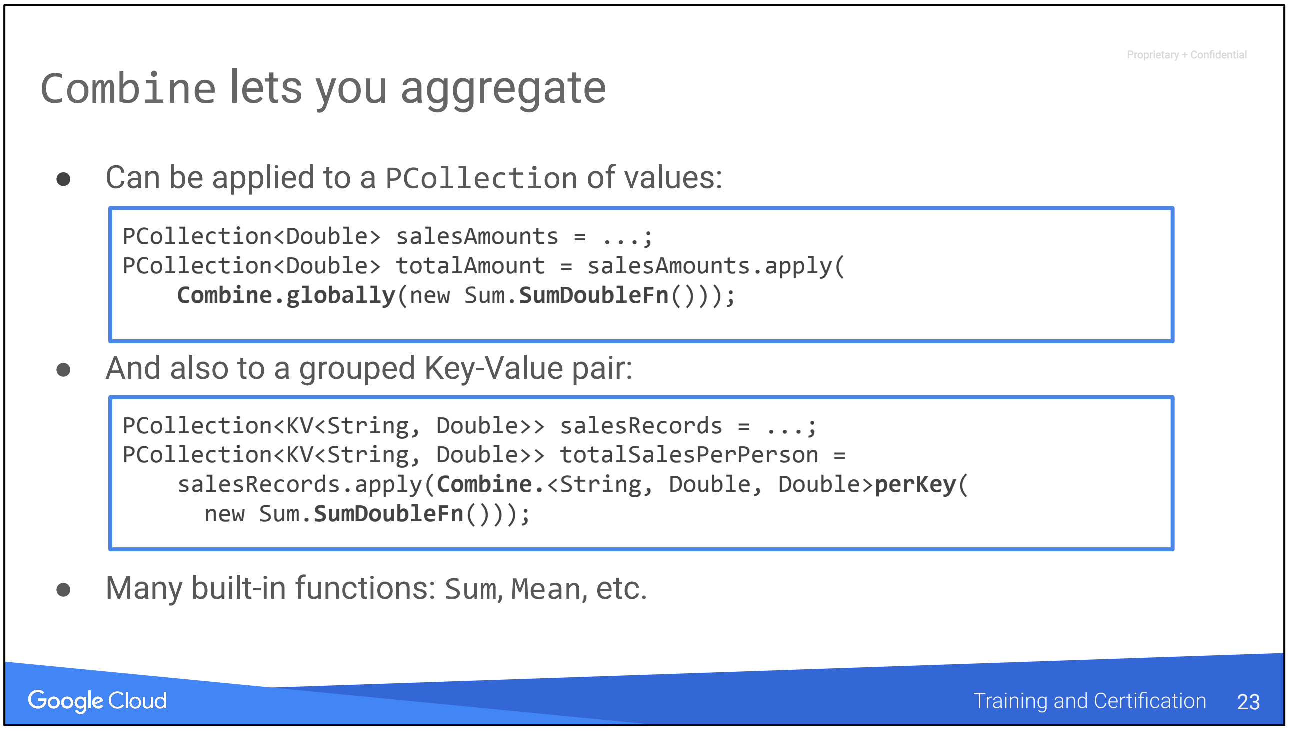
- In Dataflow, shuffle explicitly with a `GroupByKey`
  - Create a Key-Value pair in a `ParDo`
  - Then group by the key

```
PCollection<KV<String, Integer>> cityAndZipcodes = p.apply(ParDo.of(new
      DoFn<String, KV<String,Integer>>() {
          @ProcessElement
          public void processElement(ProcessContext c) throws Exception {
               String[] fields = c.element().split();
               c.output(KV.of(fields[0], Integer.parseInt(fields[3])));
}}}))

PCollection<KV<String, Iterable<Integer>>> grouped = cityAndZipcodes.apply(
    GroupByKey.<String, Integer>create());
```

Notes:

The idea is here is that we want to find all the zipcodes associated with a city

E.g., NewYork is the city and it may have 10001 10002 etc.



`Combine` lets you aggregate

- Can be applied to a `PCollection` of values:

```
PCollection<Double> salesAmounts = ...;
PCollection<Double> totalAmount = salesAmounts.apply(
    Combine.globally(new Sum.SumDoubleFn()));
```

- And also to a grouped Key-Value pair:

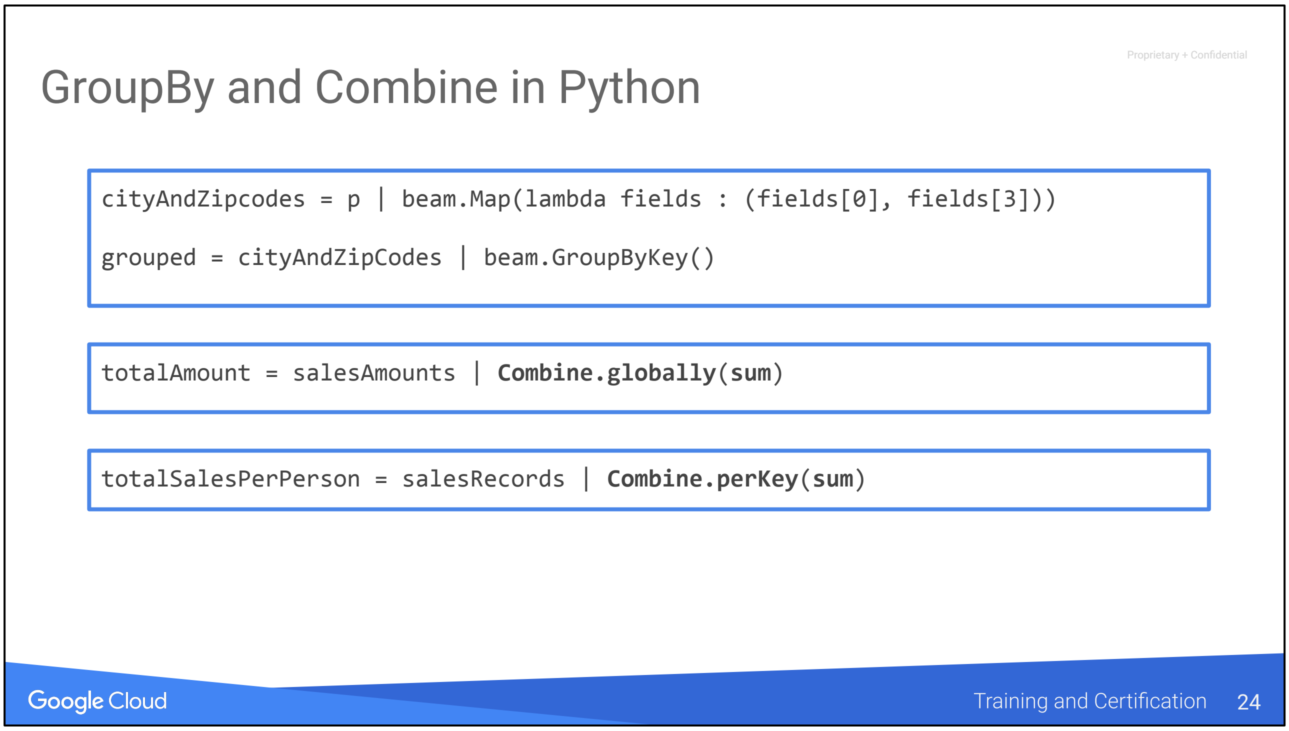
```
PCollection<KV<String, Double>> salesRecords = ...;
PCollection<KV<String, Double>> totalSalesPerPerson =
    salesRecords.apply(Combine.<String, Double, Double>perKey(
      new Sum.SumDoubleFn()));
```

- Many built-in functions: `Sum`, `Mean`, etc.

Notes:

With Java 8, some of these generics are optional.

Can write a custom Combine function by extending CombineFn, so not limited to the built-in ones.



GroupBy and Combine in Python

```
cityAndZipcodes = p | beam.Map(lambda fields : (fields[0], fields[3]))

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

```
totalAmount = salesAmounts | Combine.globally(sum)
```

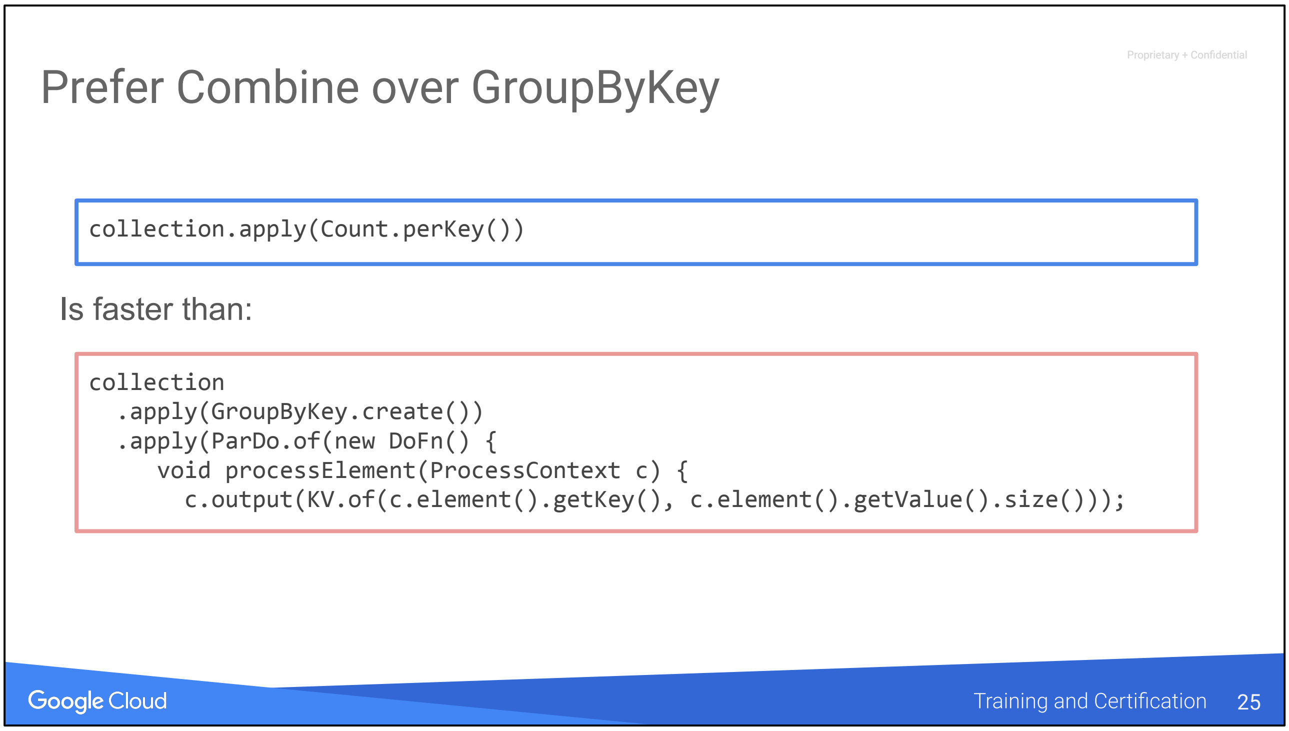
```
totalSalesPerPerson = salesRecords | Combine.perKey(sum)
```

Notes:

Conceptually similar, pythonic syntax

Key-value pairs are simply 2-tuples

Group-by-key is simply GroupByKey() -- none of the generic overload as in Java



Prefer Combine over GroupByKey

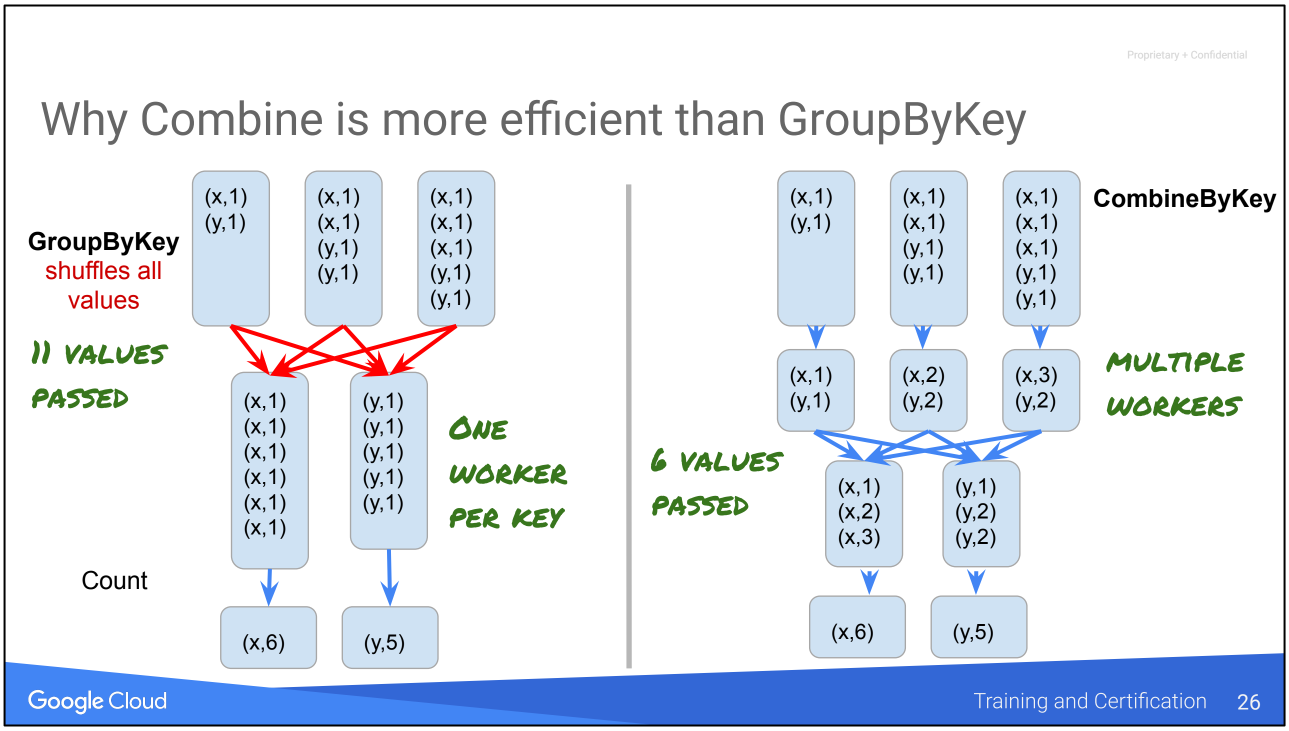
```
collection.apply(Count.perKey())
```

Is faster than:

```
collection
  .apply(GroupByKey.create())
  .apply(ParDo.of(new DoFn() {
     void processElement(ProcessContext c) {
       c.output(KV.of(c.element().getKey(), c.element().getValue().size()));
```

Notes:

In cases where you can use either a GroupBy or a Combine, use a Combine.
The version using Count is orders of magnitude faster, because Dataflow can parallelize the operation on multiple machines, which is impossible in the GroupByKey version.
This is because Dataflow knows that a Combine can be done in stages -- it can aggregate locally, then aggregate again overall.
In the GroupByKey version, Dataflow does not know this, so it will wait for the GroupByKey to fully finish before it can do the ParDo.
You can write a custom Combine function by extending CombineFn, so you are not limited to the built-in ones.

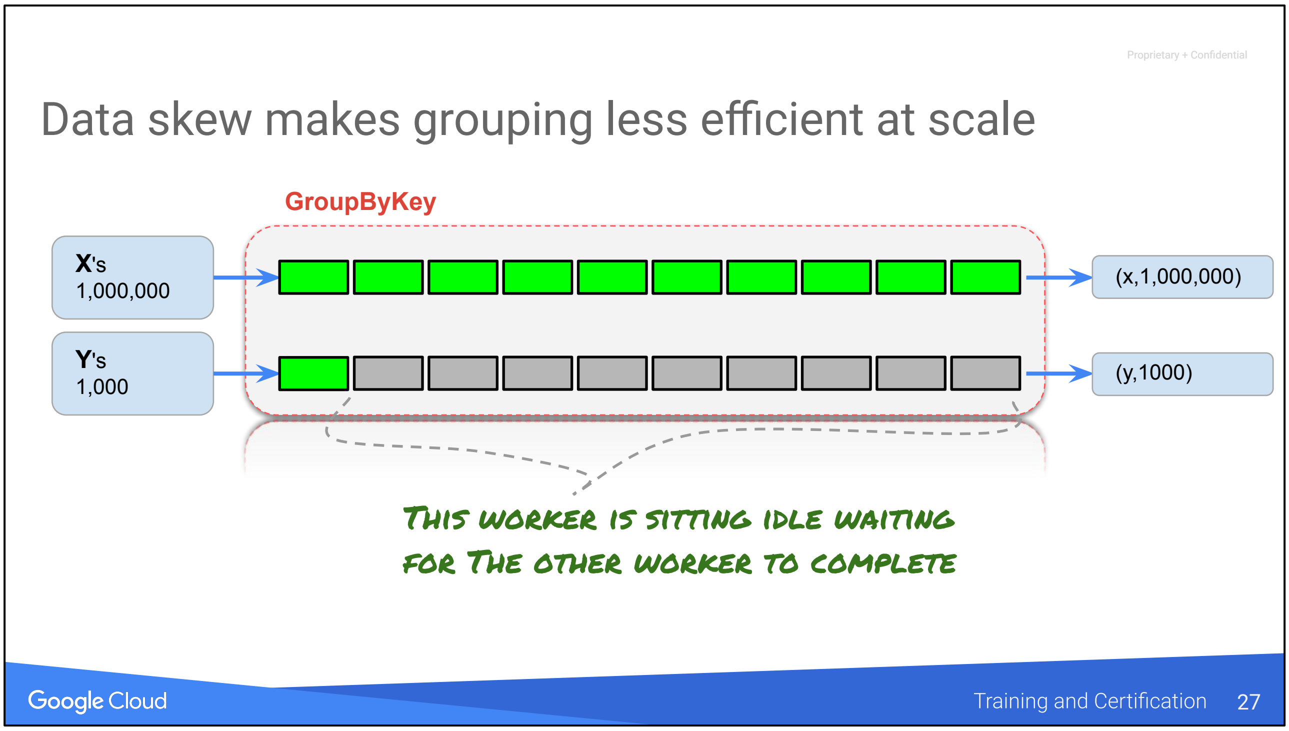


Why Combine is more efficient than GroupByKey

The way that **GroupByKey** works, Datalfow 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.



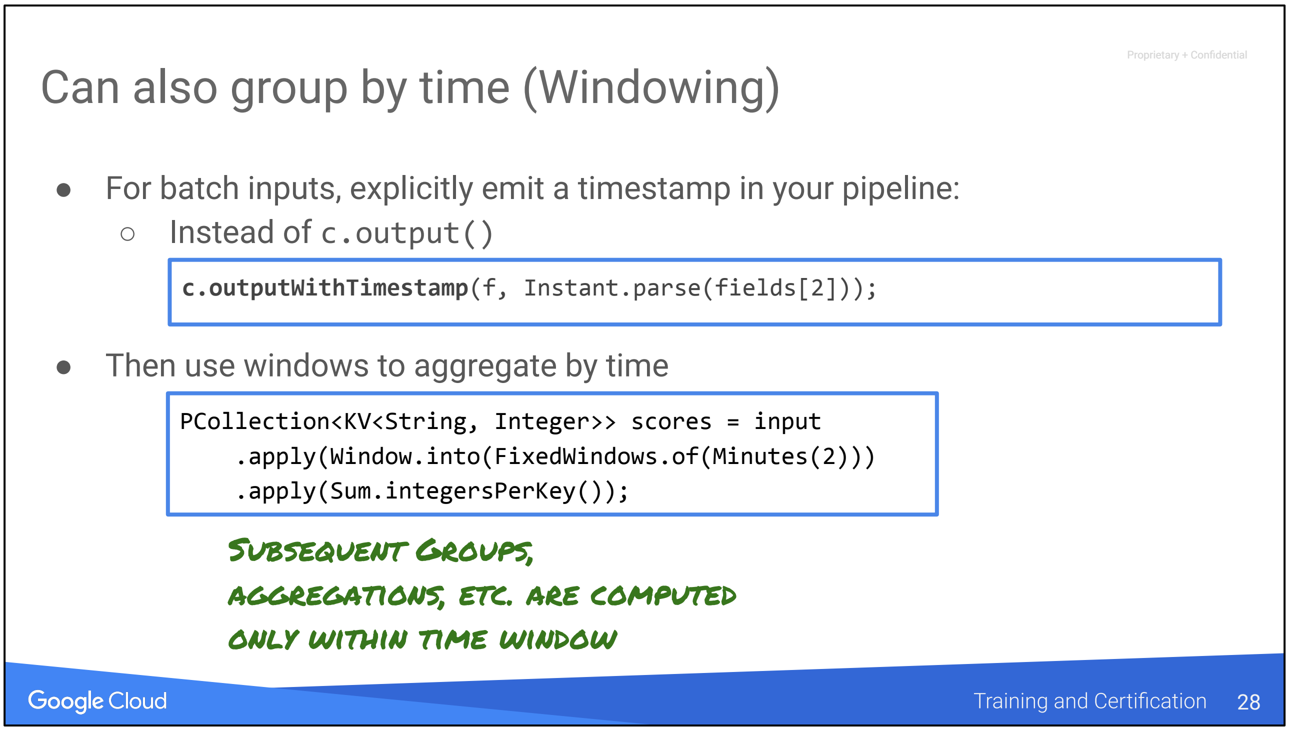
Data skew makes grouping less efficient at scale

GroupByKey
X's 1,000,000
Y's 1,000
(x,1,000,000)
(y,1000)
THIS WORKER IS SITTING IDLE WAITING FOR THE OTHER WORKER TO COMPLETE

When the same example is scaled up in the presence of skewed data, the situation becomes much worse.

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 are paying for the worker that sits idle waiting for the other worker to complete.

Dataflow is designed to avoid inefficiencies by keeping the data balanced. 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.



Can also group by time (Windowing)

- For batch inputs, explicitly emit a timestamp in your pipeline:
  - Instead of `c.output()`

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

- Then use windows to aggregate by time

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

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

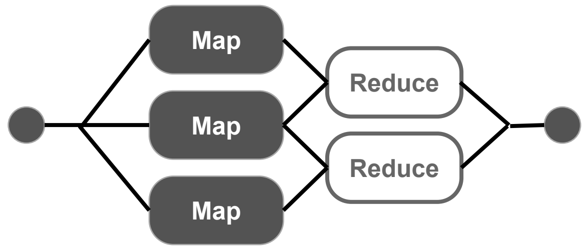
Notes:

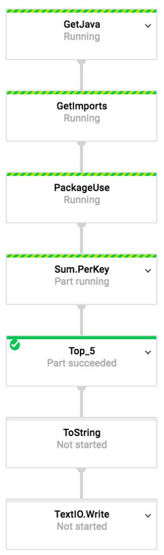
This timestamp will be the time at which the element was published to PubSub. If you want to use a custom timestamp, it must be published as a PubSub attribute, and you tell Dataflow about it using the `timestampLabel` setter.

Lab 2: MapReduce in Dataflow

In this lab, you will learn how to:

- Specify and use command-line options
- Carry out Map and Reduce operations

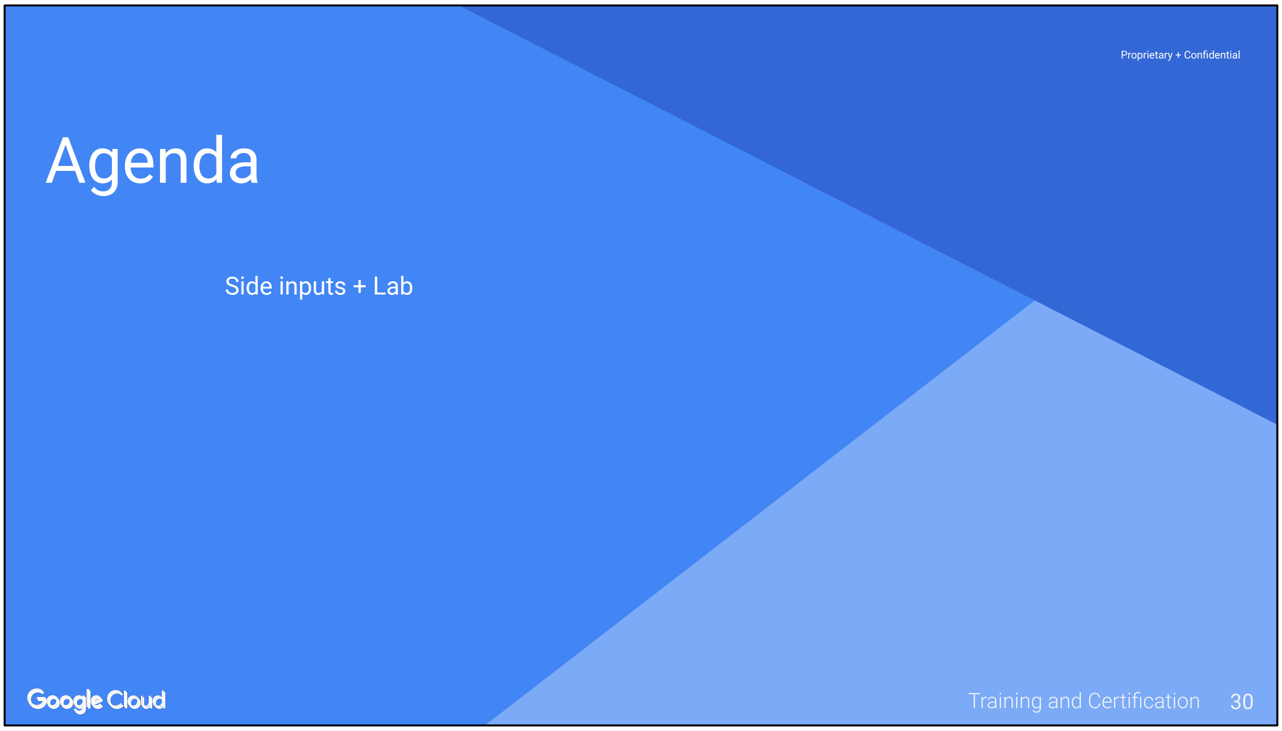




Notes:

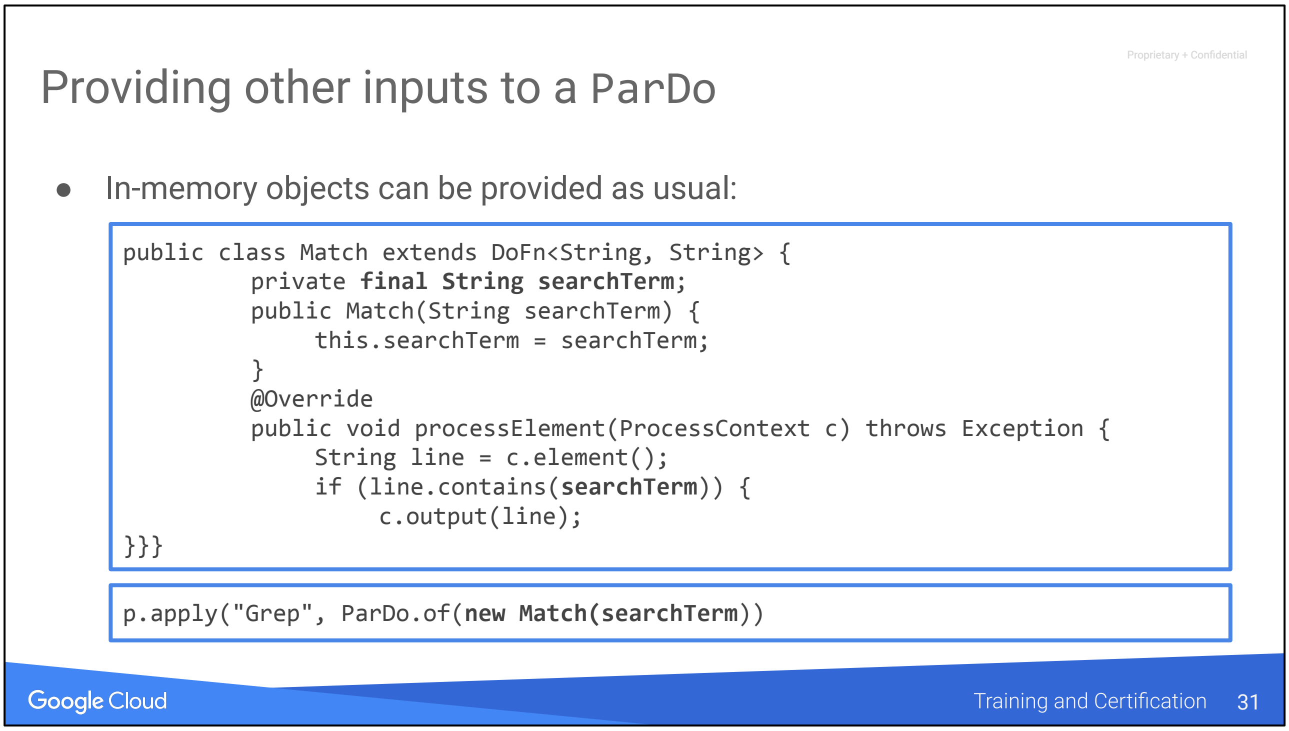
The pipeline identifies the 5 most popular imported packages.

All the ParDo() operations are Maps; the Sum and Top5 are reduces.



Agenda

Side inputs + Lab



Providing other inputs to a ParDo

- In-memory objects can be provided as usual:

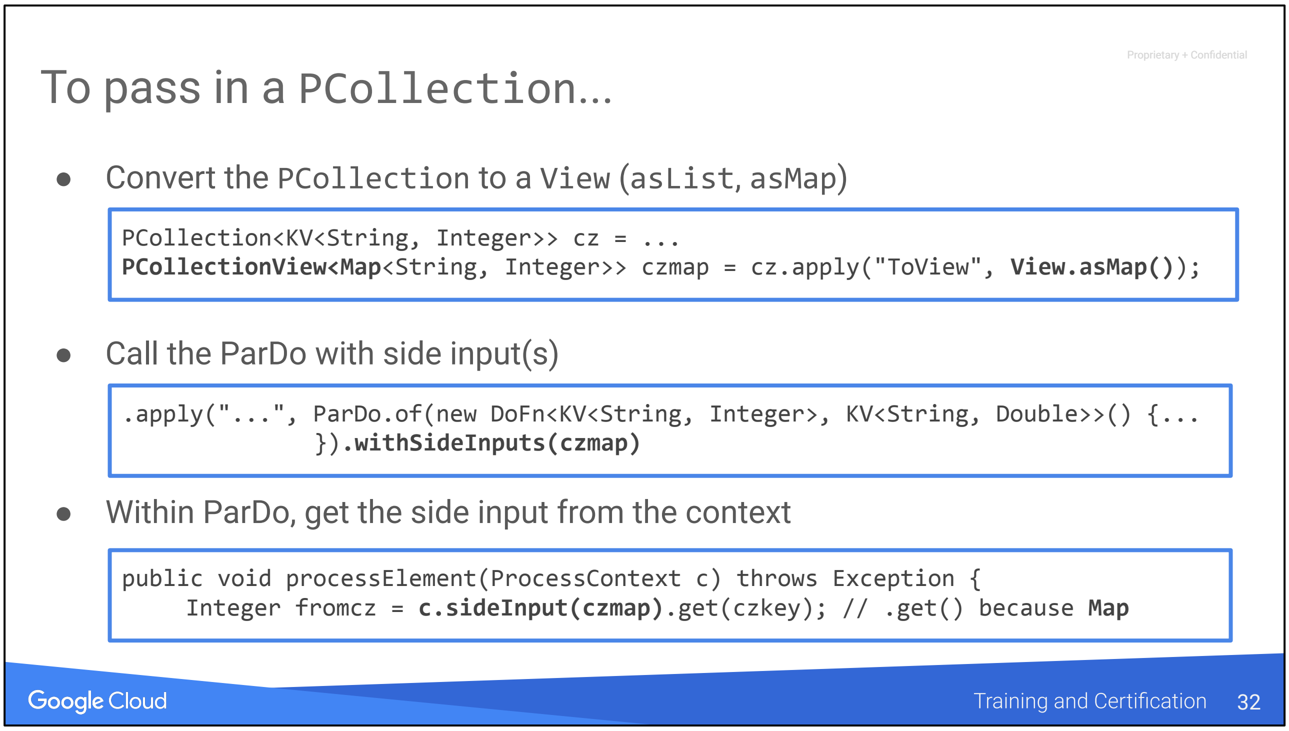
```
public class Match extends DoFn<String, String> {
         private final String searchTerm;
         public Match(String searchTerm) {
              this.searchTerm = searchTerm;
         }
         @Override
         public void processElement(ProcessContext c) throws Exception {
              String line = c.element();
              if (line.contains(searchTerm)) {
                   c.output(line);
}}}
```

```
p.apply("Grep", ParDo.of(new Match(searchTerm))
```

Notes:

Make such objects final to ensure that they are not mistakenly used as mutable state.

But PCollections are not in-memory (the batch runner supports large values as a special case, but it is not a good practice to do it. A Co-Group-By-Key would be a better choice).



To pass in a `PCollection`...

- Convert the `PCollection` to a `View` (`asList`, `asMap`)

```
PCollection<KV<String, Integer>> cz = ...
PCollectionView<Map<String, Integer>> czmap = cz.apply("ToView", View.asMap());
```

- Call the ParDo with side input(s)

```
.apply("...", ParDo.of(new DoFn<KV<String, Integer>, KV<String, Double>>() {...
              }).withSideInputs(czmap)
```

- Within ParDo, get the side input from the context

```
public void processElement(ProcessContext c) throws Exception {
     Integer fromcz = c.sideInput(czmap).get(czkey); // .get() because Map
```

Notes:

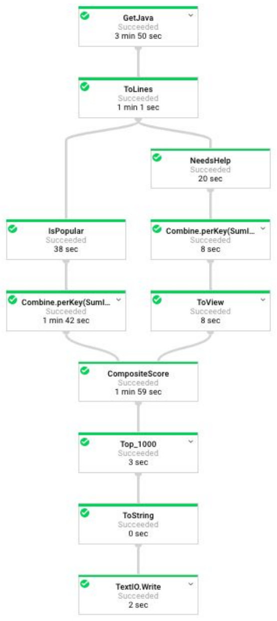
c.sideInput() returns a java.util.Map

Serverless Data Analysis with Dataflow

Lab 3: Side Inputs

In this lab, you will learn how to:

- Get data from BigQuery
- Use side inputs in an `apply()`



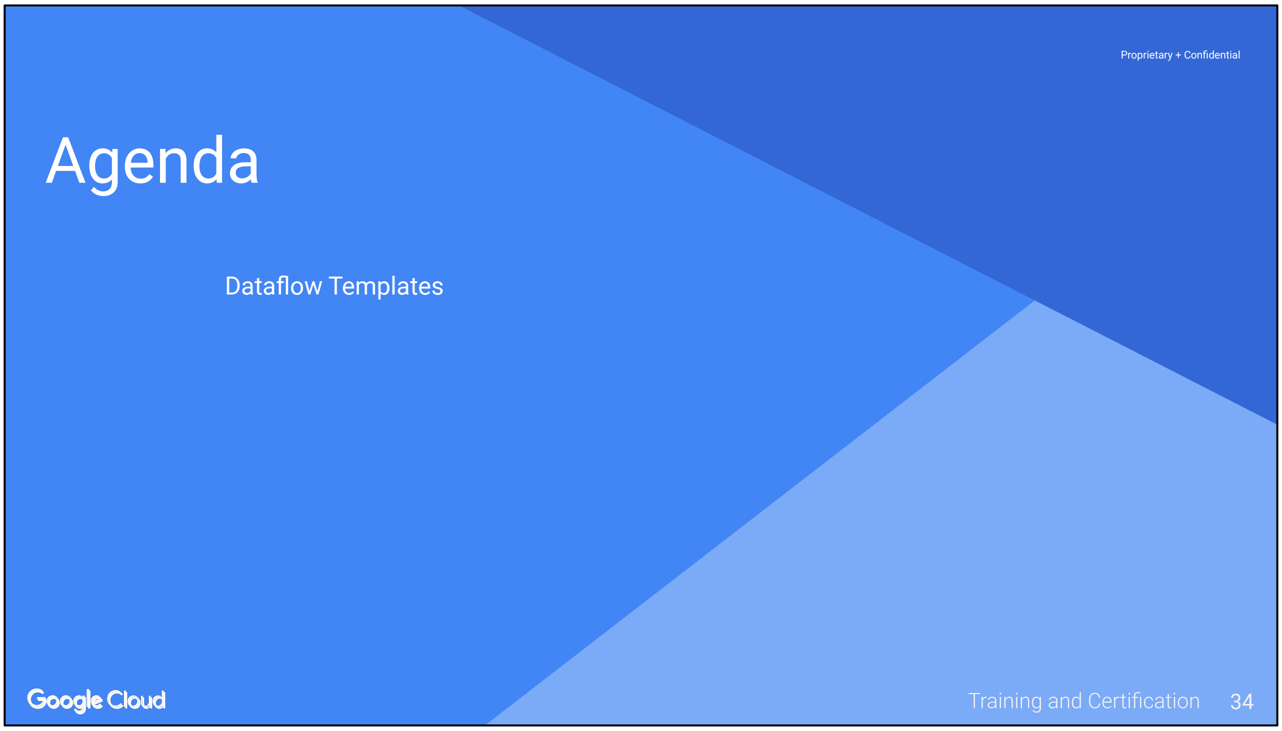
Notes:

The pipeline identifies popular Java packages that need volunteers to do small tasks on the code base.

Depending on time constraints and class interest, your instructor might choose to make this lab a demo and walk through the code.

See https://medium.com/google-cloud/popular-java-projects-on-github-that-could-use-some-help-analyzed-using-bigquery-and-dataflow-dbd5753827f4#.v646zq4xp for a description of what this pipeline does.

Image (cc0)
https://pixabay.com/en/volunteer-volunteerism-volunteering-652383/



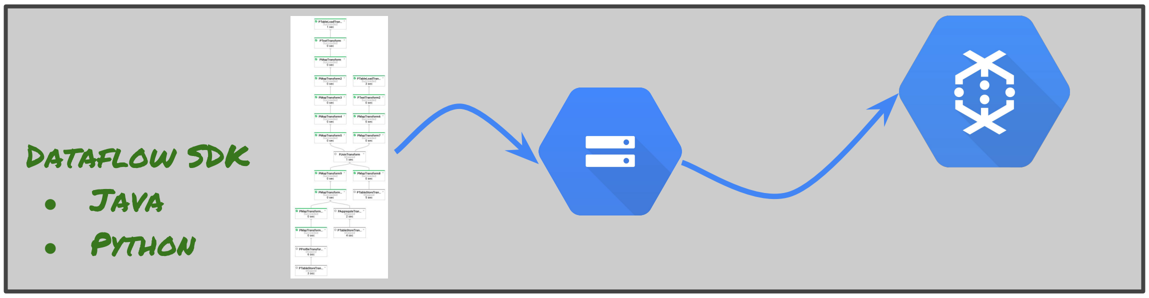
Agenda

Dataflow Templates

Google Cloud

Traditional workflow all happens in one environment

DEVELOPMENT ENVIRONMENT

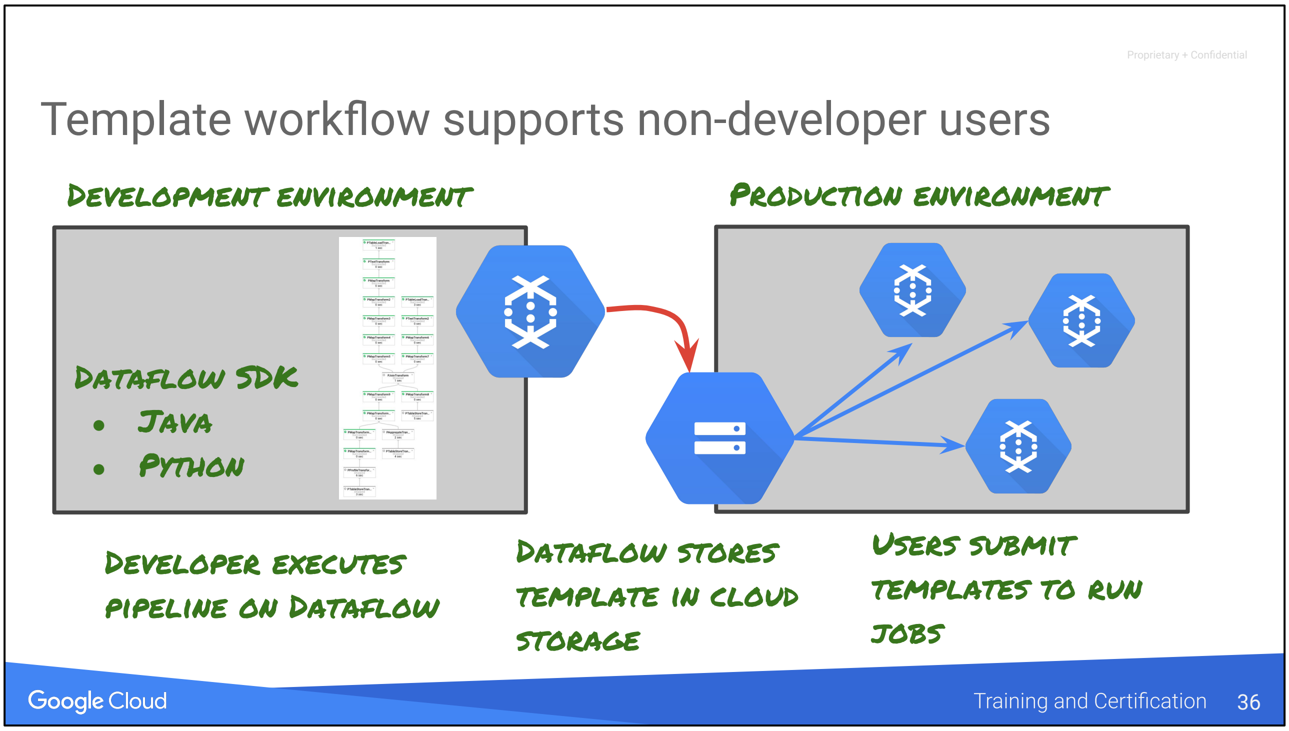


DEVELOPER CREATES PIPELINE IN THE DEVELOPMENT ENVIRONMENT

SDK STAGES FILES IN CLOUD STORAGE

DEVELOPER OR USER SUBMITS SOURCE CODE TO RUN DATAFLOW JOBS

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.

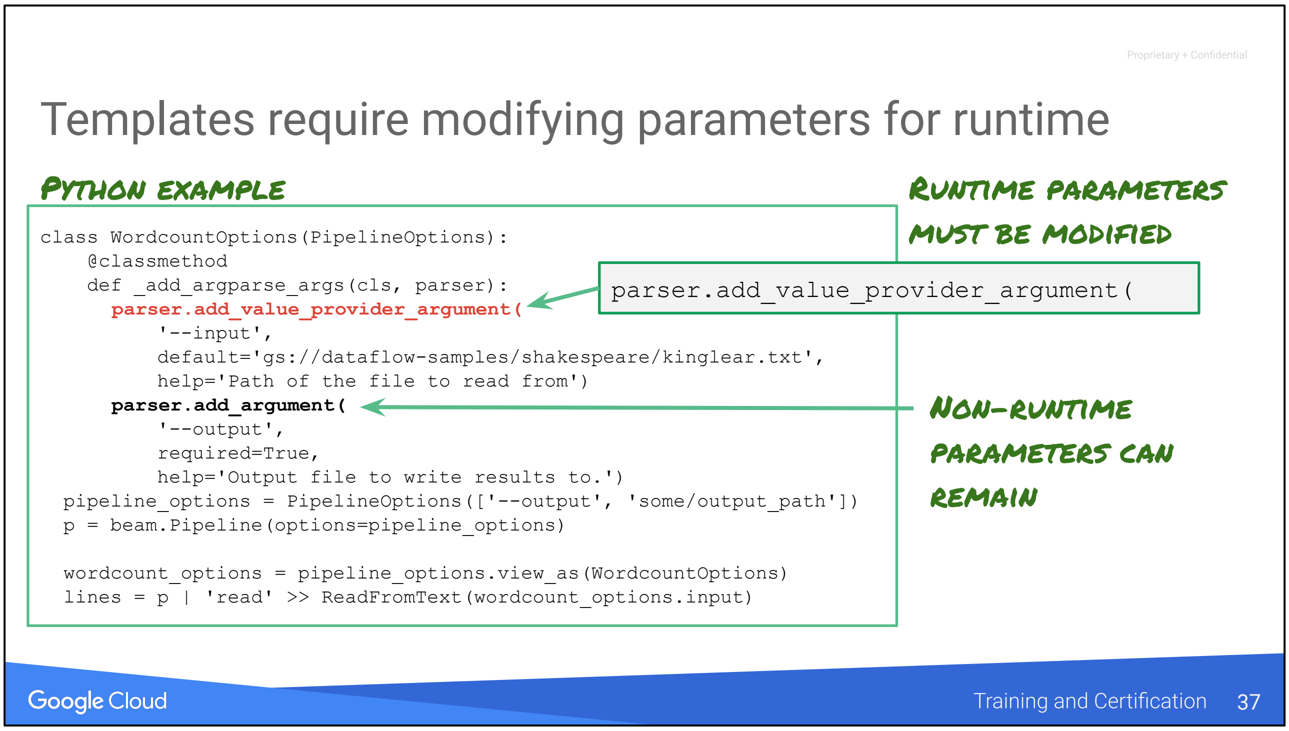


Template workflow supports non-developer users

Development environment
Production environment
Dataflow SDK
• Java
• Python
Developer executes pipeline on Dataflow
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.

https://cloud.google.com/dataflow/docs/templates/overview



Templates require modifying parameters for runtime

PYTHON EXAMPLE

```
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)
```

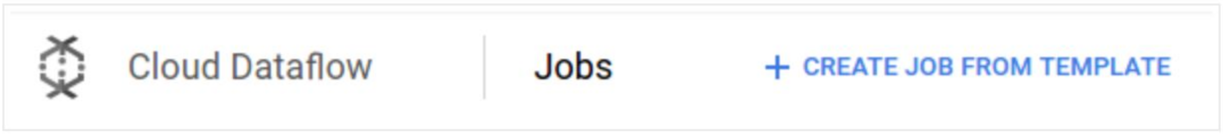
RUNTIME PARAMETERS MUST BE MODIFIED

```
parser.add_value_provider_argument(
```

NON-RUNTIME PARAMETERS CAN REMAIN

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. Theser 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 DoFn (your functions). And there are Static and Nested versions of ValueProvider for more complex cases.

Create jobs from template via SDK, gcloud, or Console

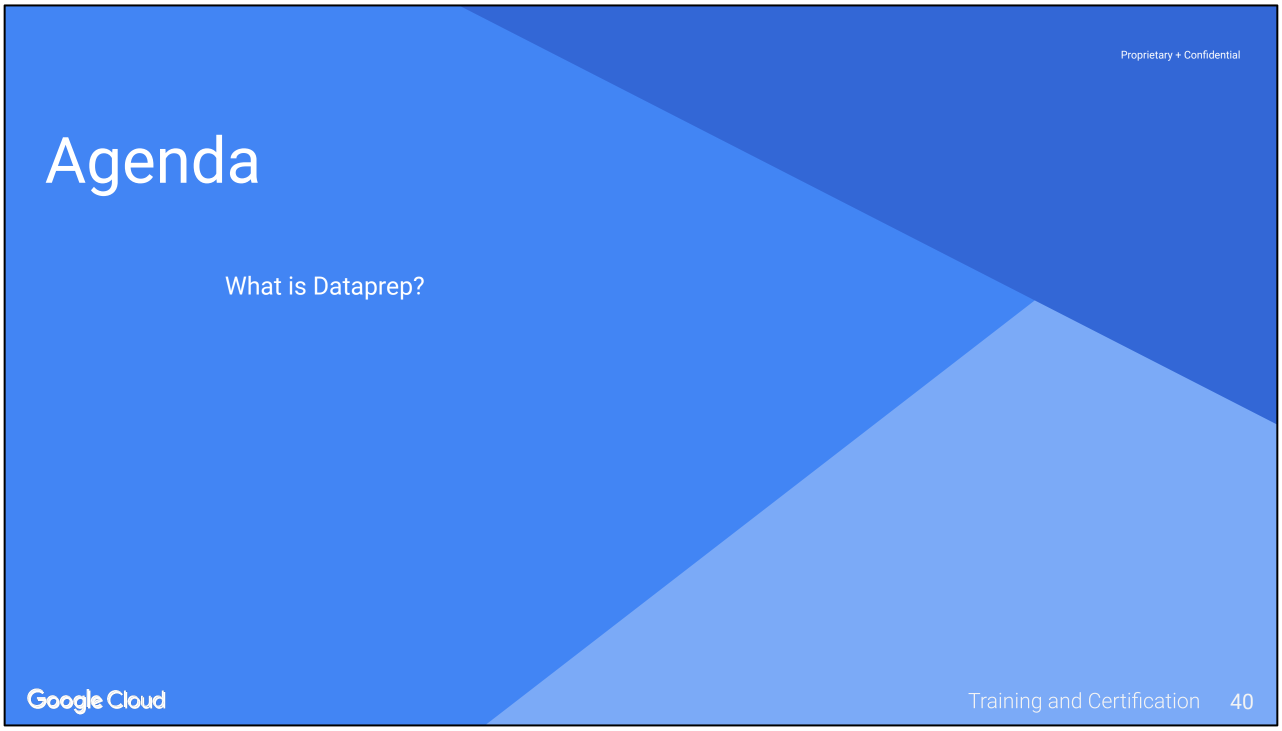


YOU SPECIFY:

- THE LOCATION OF THE TEMPLATE IN CLOUD STORAGE
- AN OUTPUT LOCATION IN CLOUD STORAGE
- NAME : VALUE PARAMETERS (THAT MAP TO THE VALUEPROVIDER INTERFACE)

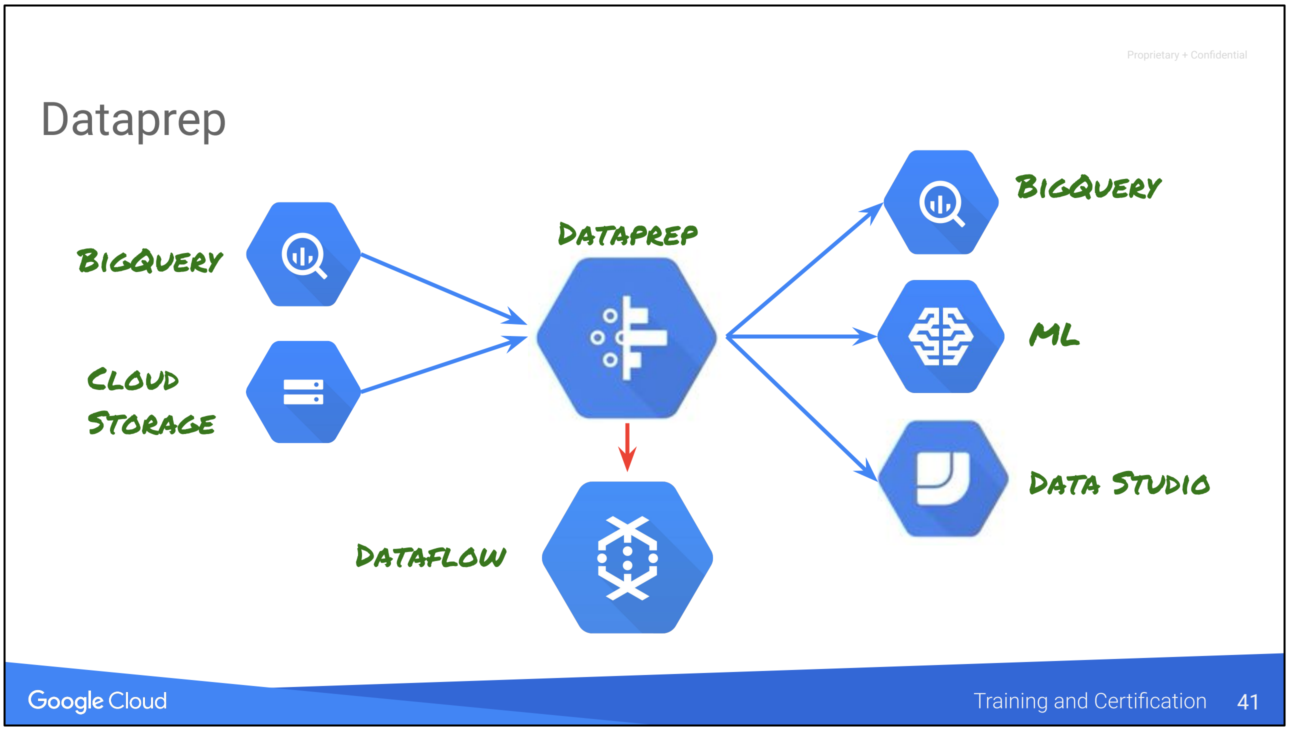
Example templates for basic tasks are provided

- WordCount
- Cloud Pub/Sub to BigQuery
- Cloud Storage Text to Cloud Pub/Sub
- Cloud Pub/Sub to Cloud Storage Text
- Cloud Datastore to Cloud Storage Text
- Cloud Storage Text to BigQuery
- Cloud Storage Text to Cloud Datastore
- Bulk Decompress Cloud Storage Files

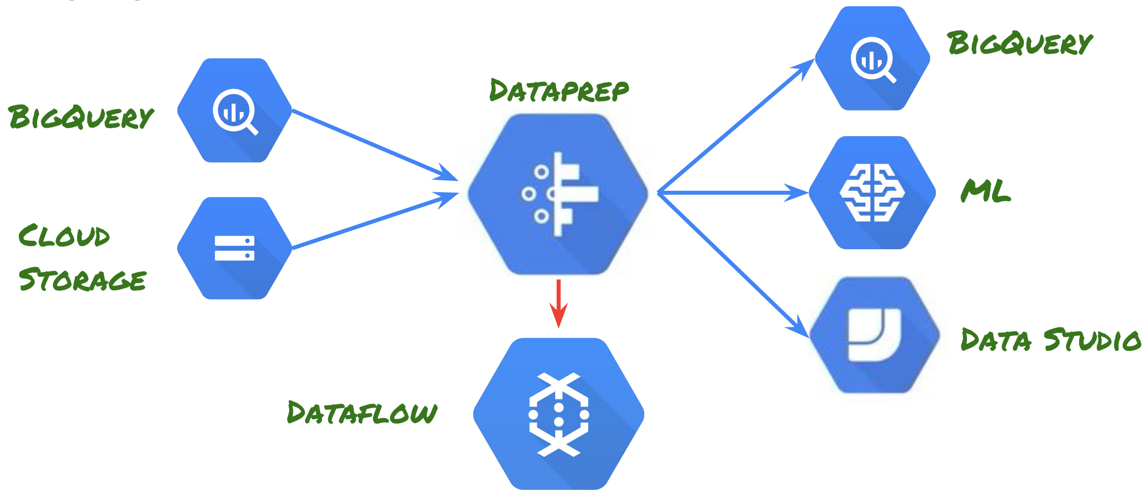


Agenda

What is Dataprep?



Dataprep

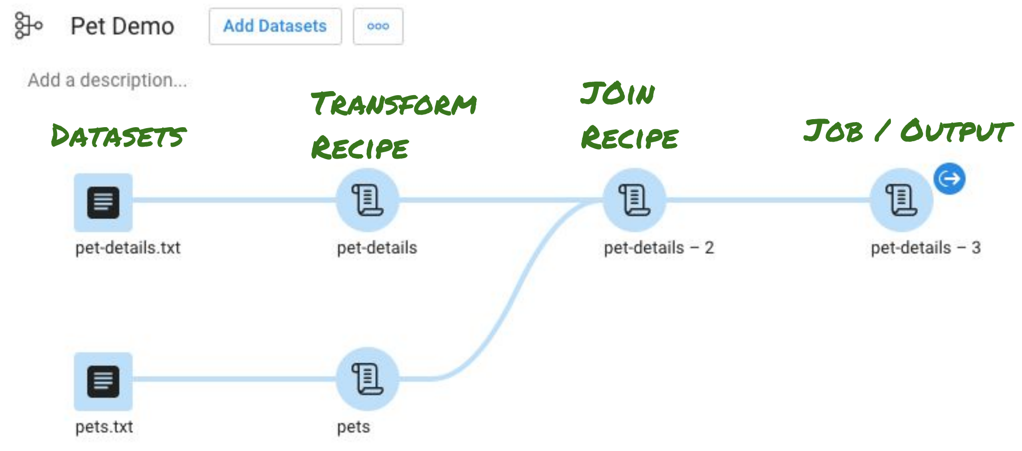


Google Cloud

Dataprep is an interactive graphical system for preparing structured or unstructured data for use in analytics (BigQuery), Visualization (Data Studio), and to train Machine Learning models. Input integration with Cloud Storage, BigQuery, and Files.

Walk through at Next'17. https://www.youtube.com/watch?v=Q5GuTIgmt98

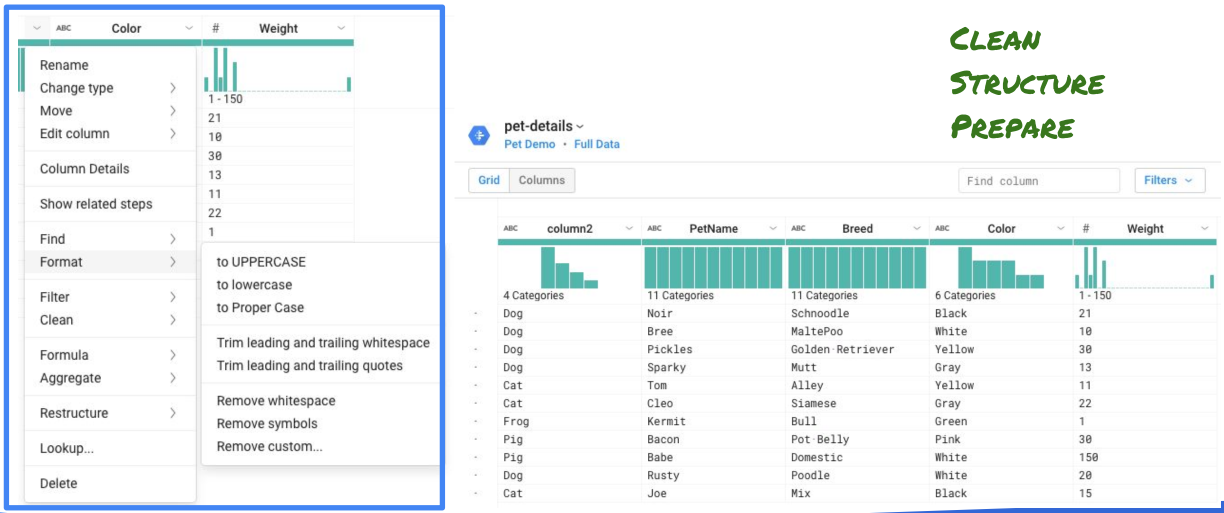
Create pipelines in Dataprep Flows



Dataprep provides a graphical interface for interactively designing a pipeline. The elements are divided into datasets, recipes, and output. A dataset roughly translates into a Dataflow pipeline read. A recipe usually translates into multiple pipeline transformations. And an output translates into a pipeline action.

Tools for data cleanup, structuring, and transformation

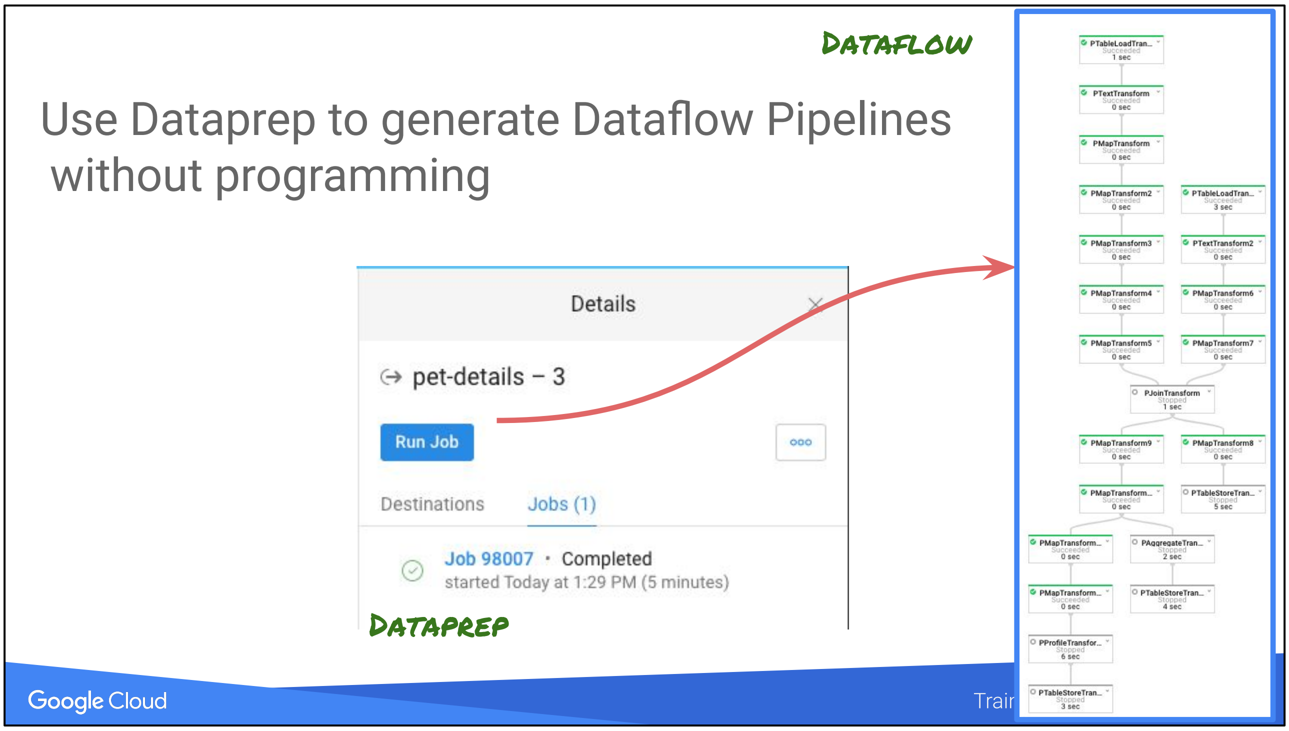
CLEAN
STRUCTURE
PREPARE



Weight
1 - 150
21
10
30
13
11
22
1

column2	PetName	Breed	Color	Weight
4 Categories	11 Categories	11 Categories	6 Categories	1 - 150
Dog	Noir	Schnoodle	Black	21
Dog	Bree	MaltePoo	White	10
Dog	Pickles	Golden Retriever	Yellow	30
Dog	Sparky	Mutt	Gray	13
Cat	Tom	Alley	Yellow	11
Cat	Cleo	Siamese	Gray	22
Frog	Kermit	Bull	Green	1
Pig	Bacon	Pot Belly	Pink	30
Pig	Babe	Domestic	White	150
Dog	Rusty	Poodle	White	20
Cat	Joe	Mix	Black	15

Dataprep provides a rich set of tools for working with data. In this example, the format of a string field can have transformations applied to change to uppercase, to proper case (initial uppercase letters), to trim leading and trailing whitespace, and to remove whitespace altogether. These are the kinds of transformations commonly needed to improving the quality of data produced by a native system in preparation for big data processing.



Use Dataprep to generate Dataflow Pipelines without programming

Details

pet-details – 3

Run Job

Destinations Jobs (1)

Job 98007 · Completed
started Today at 1:29 PM (5 minutes)

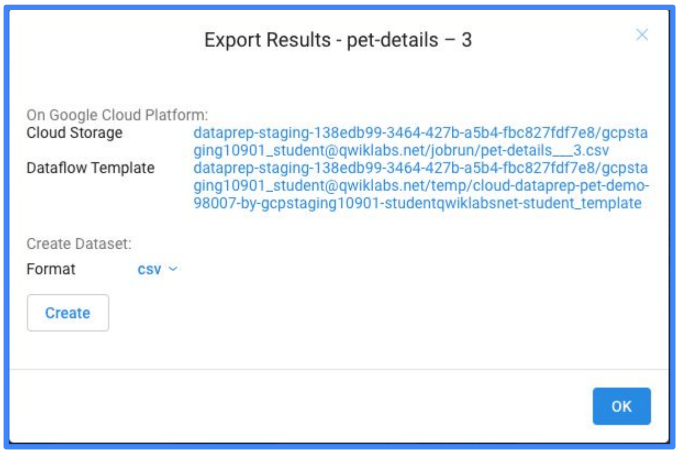
DATAFLOW
DATAPREP
PTableLoadTran... Succeeded 1 sec
PTextTransform Succeeded 0 sec
PMapTransform Succeeded 0 sec
PMapTransform2 Succeeded 0 sec
PTableLoadTran... Succeeded 3 sec
PMapTransform3 Succeeded 0 sec
PTextTransform2 Succeeded 0 sec
PMapTransform4 Succeeded 0 sec
PMapTransform6 Succeeded 0 sec
PMapTransform5 Succeeded 0 sec
PMapTransform7 Succeeded 0 sec
PJoinTransform Stopped 1 sec
PMapTransform9 Succeeded 0 sec
PMapTransform8 Succeeded 0 sec
PMapTransform... Succeeded 0 sec
PTableStoreTran... Stopped 5 sec
PMapTransform... Succeeded 0 sec
PAggregateTran... Stopped 2 sec
PMapTransform... Succeeded 0 sec
PTableStoreTran... Stopped 4 sec
PProfileTransfor... Stopped 6 sec
PTableStoreTran... Stopped 3 sec
Google Cloud
Trair

Dataprep provides a high-leverage method to quickly create dataflow pipelines without coding. This is especially useful for data quality tasks and for master data tasks (combining data from multiple sources), where programming may not be required.

Download Dataflow Templates

Output to Cloud Storage (CSV)

Download Dataflow Template



Export Results - pet-details – 3

On Google Cloud Platform:

Cloud Storage	dataprep-staging-138edb99-3464-427b-a5b4-fbc827fdf7e8/gcpstaging10901_student@qwiklabs.net/jobrun/pet-details___3.csv
Dataflow Template	dataprep-staging-138edb99-3464-427b-a5b4-fbc827fdf7e8/gcpstaging10901_student@qwiklabs.net/temp/cloud-dataprep-pet-demo-98007-by-gcpstaging10901-studentqwiklabsnet-student_template

Create Dataset:

Format csv

The pipeline can be output as a Dataflow Template for continued use in Dataflow.

For example, you could set up a data quality job to clean up source data provided by a native system that is destined for data analysis. Then this template can be used by the administrative staff periodically to submit clean data for the analysis task.

Resources

- Dataflow: https://cloud.google.com/dataflow/
- Dataprep: https://cloud.google.com/dataprep/
- Which Java projects need help? https://medium.com/google-cloud/popular-java-projects-on-github-that-could-use-some-help-analyzed-using-bigquery-and-dataflow-dbd5753827f4#.t82wsxd2c
- Processing logs at scale using Cloud Dataflow https://cloud.google.com/solutions/processing-logs-at-scale-using-dataflow
- Beam resources:

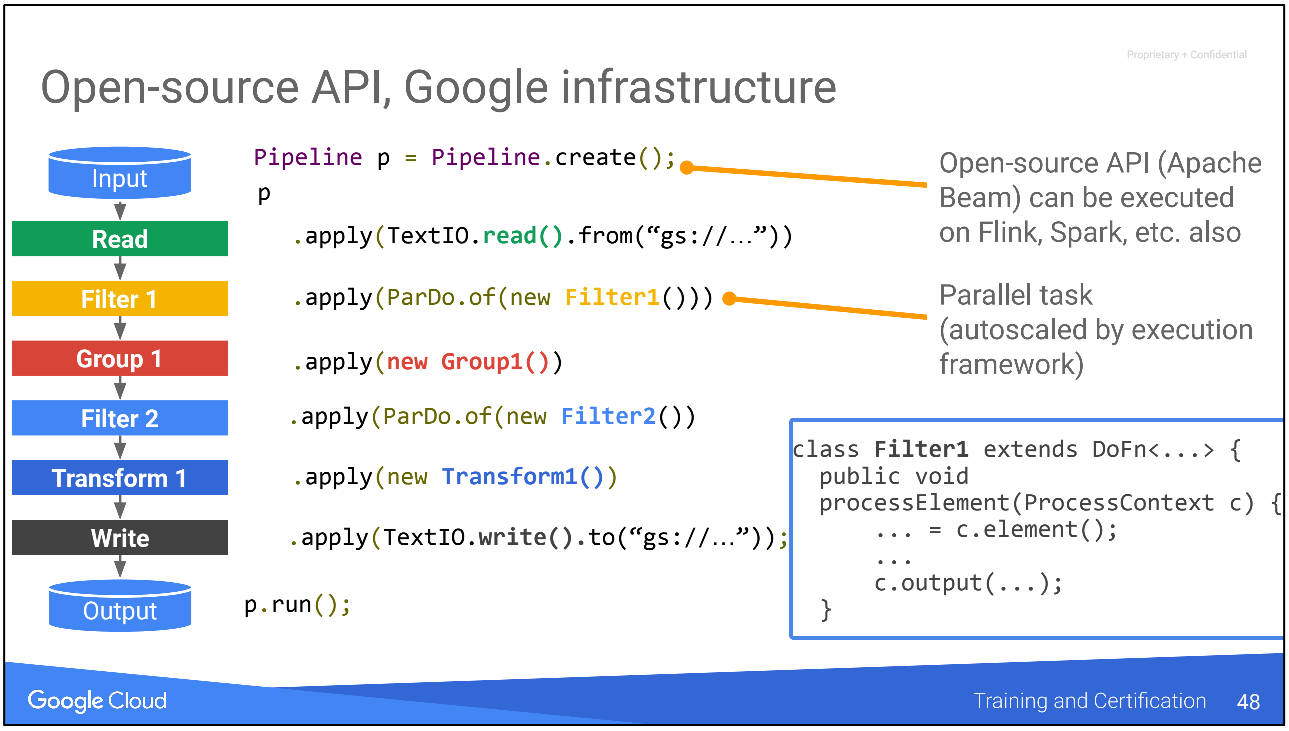
https://beam.apache.org/contribute/presentation-materials/

https://beam.apache.org/documentation/resources/

cloud.google.com

Images by Connie Zhou

Notes:



Open-source API, Google infrastructure

Input
Read
Filter 1
Group 1
Filter 2
Transform 1
Write
Output

```
Pipeline p = Pipeline.create();
p
   .apply(TextIO.read().from(“gs://…”))

   .apply(ParDo.of(new Filter1()))

   .apply(new Group1())

   .apply(ParDo.of(new Filter2())

   .apply(new Transform1())

   .apply(TextIO.write().to(“gs://…”));

p.run();
```

Open-source API (Apache Beam) can be executed on Flink, Spark, etc. also

Parallel task (autoscaled by execution framework)

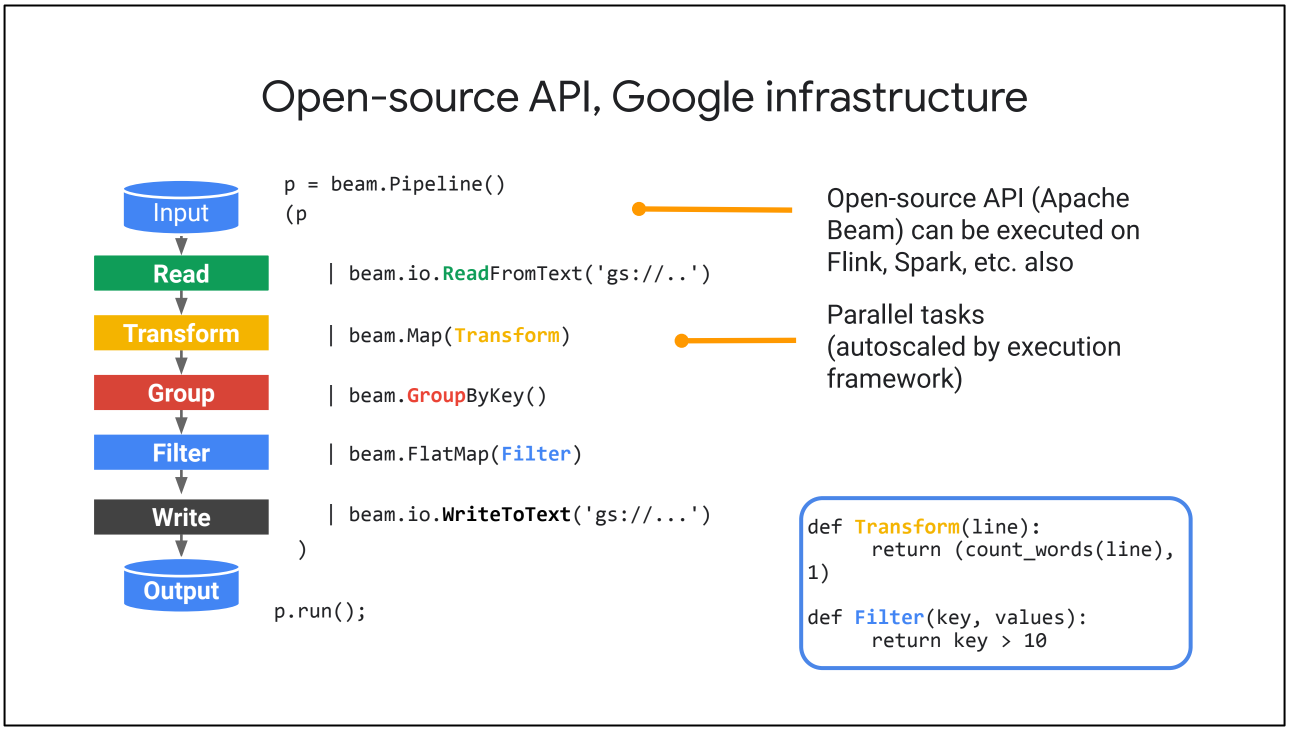
```
class Filter1 extends DoFn<...> {
  public void
  processElement(ProcessContext c) {
      ... = c.element();
      ...
      c.output(...);
  }
```

Notes:

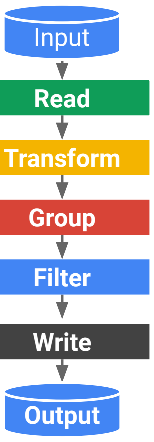
Distinguish between the API (Apache Beam) and the implementation/execution framework (Dataflow)

Each step of the pipeline does a filter, group, transform, compare, join, and so on. Transforms can be done in parallel.

c.element() gets the input. c.output() sends the output to the next step of the pipeline.



Open-source API, Google infrastructure



```
p = beam.Pipeline()
(p

     | beam.io.ReadFromText('gs://..')

     | beam.Map(Transform)

     | beam.GroupByKey()

     | beam.FlatMap(Filter)

     | beam.io.WriteToText('gs://...')
 )

p.run();
```

Open-source API (Apache Beam) can be executed on Flink, Spark, etc. also

Parallel tasks (autoscaled by execution framework)

```
def Transform(line):
     return (count_words(line),
1)

def Filter(key, values):
     return key > 10
```

One thing that makes writing Apache Beam easy is that the code written for Beam is similar to how people think of data processing pipelines. Take a look at the pipeline in the center of the slide. This sample Python code analyzes the number of words in lines of text in documents. So, as an input to the pipeline you may want to read text files from Google Cloud Storage.

Then, you transform the data, figure out the number of words in each line of text. This kind of a transformation can be automatically scaled by Dataflow to run in parallel. Next, In your pipeline, you can group lines by the number of words using grouping and other aggregation operations. You can also filter out values, for example to ignore lines with fewer than 10 words. Once all the transformation, grouping, and filtering operations are done, the pipeline writes the result to Google Cloud Storage.

Notice that this implementation separates the pipeline definition from the pipeline execution. All the steps that you see before call to the p.run() method are just defining what the pipeline should do. The pipeline actually gets executed only when you call the run method.