Autoscaling data processing pipelines

Data Engineering on Google Cloud Platform

Notes:

30 slides + 3 labs ~ 3 hours

Agenda

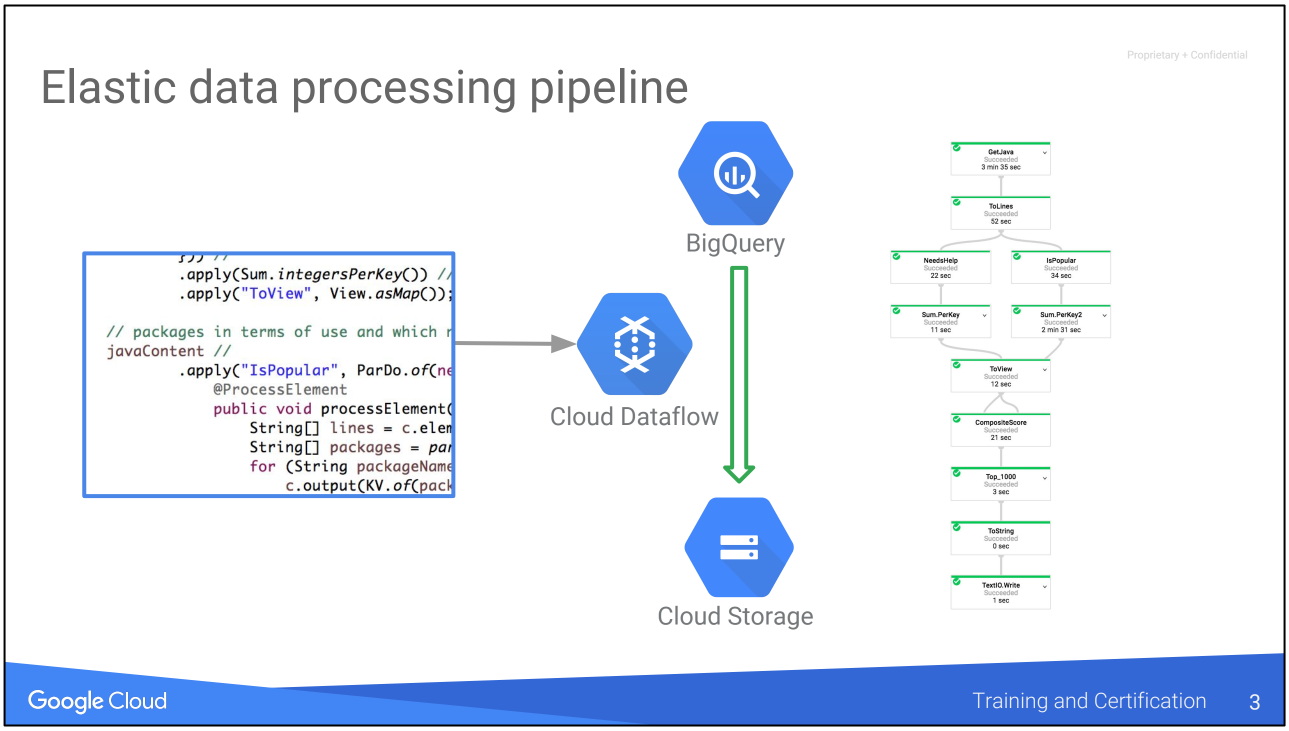
What is Dataflow?

Data pipeline + Lab

MapReduce in Dataflow + Lab

Side inputs + Lab

Streaming



Proprietary + Confidential

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
```

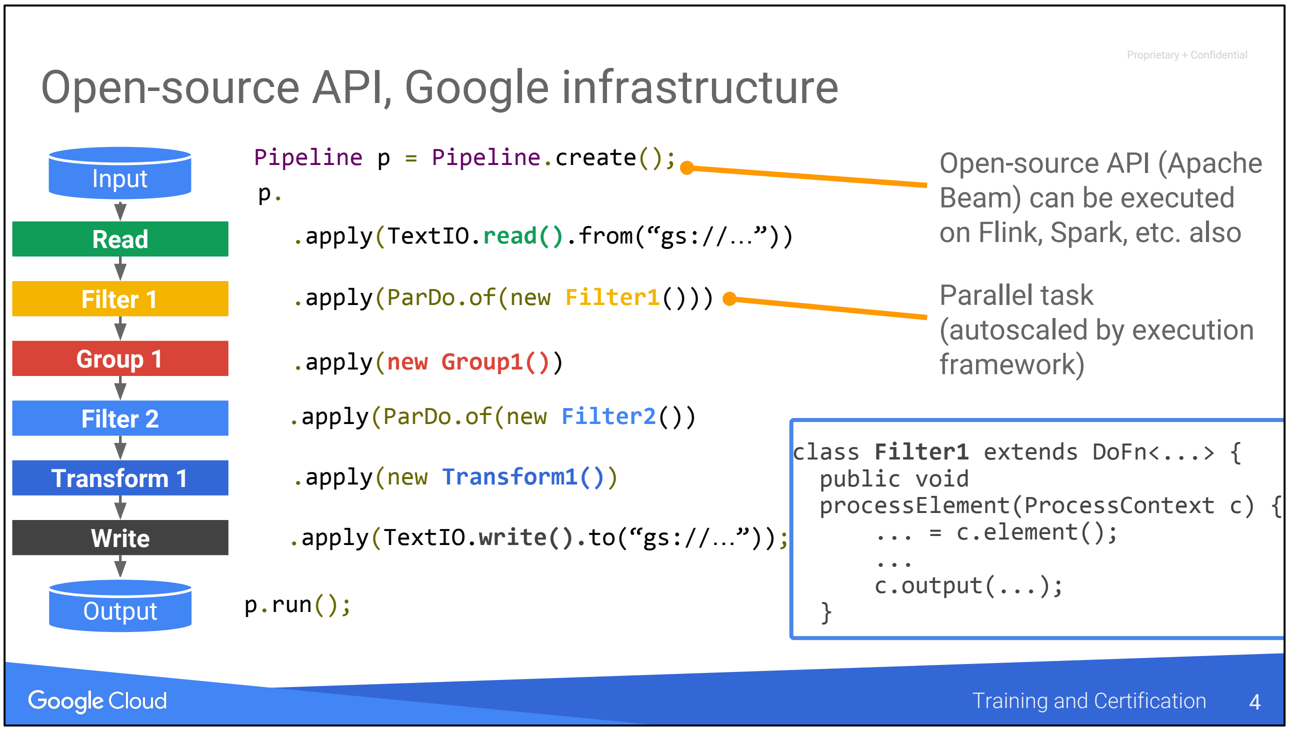
Google Cloud

Notes:

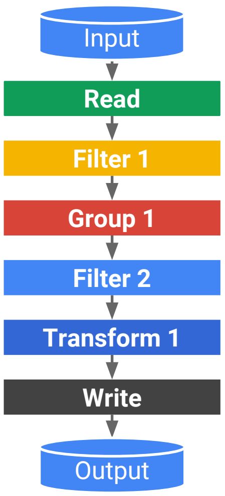
The idea is to write Java code, deploy it to Dataflow which then executes the pipeline.

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



```
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(...);
  }
```

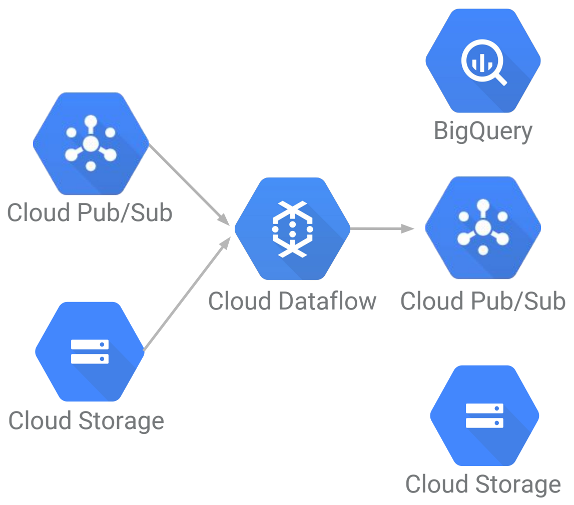
Notes:

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.

Same code does real-time and batch

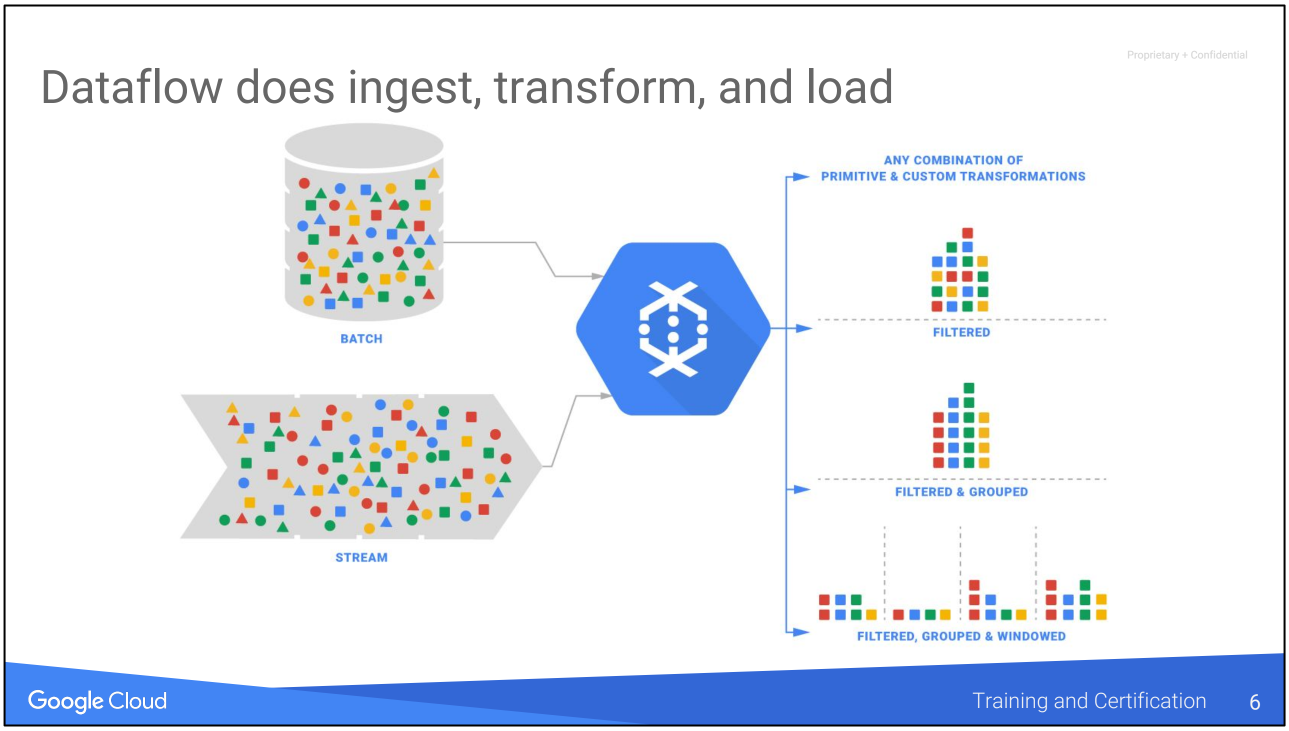


```
Pipeline p = Pipeline.create();
    p.begin()
        .apply(PubsubIO.readStrings().fromTopic(topic))
        .apply(Window.into(SlidingWindows
                     .of(Duration.standardMinutes(60))
        .apply(ParDo.of(new Filter1()))
        .apply(new Group1())
        .apply(ParDo.of(new Filter2())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

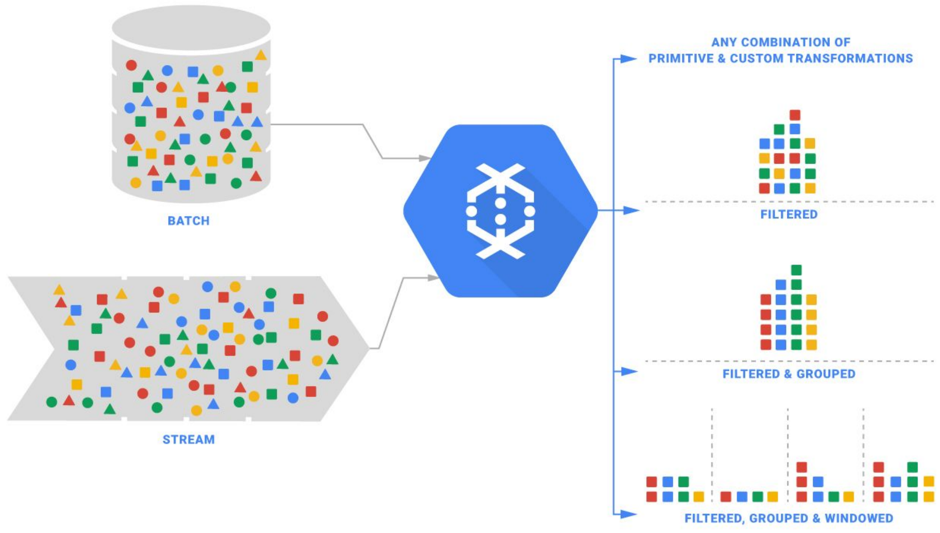
Notes:

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

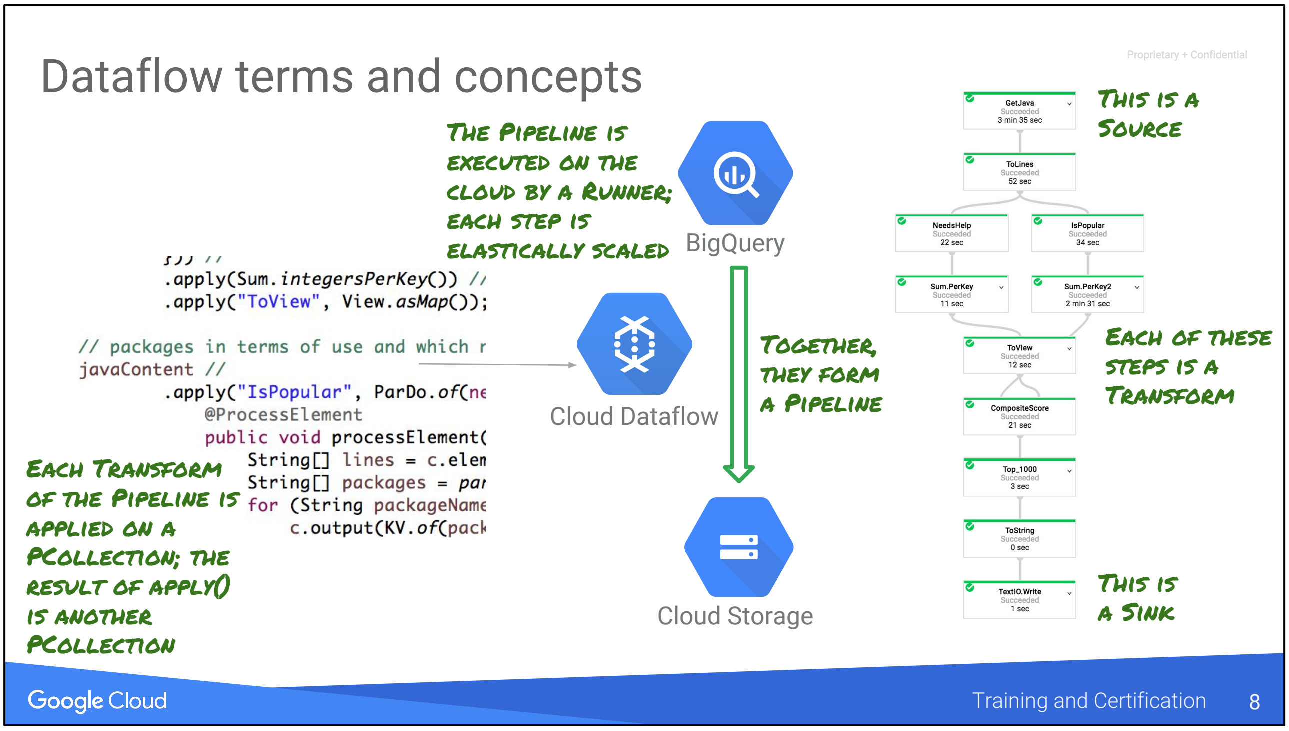

BATCH
STREAM
ANY COMBINATION OF PRIMITIVE & CUSTOM TRANSFORMATIONS
FILTERED
FILTERED & GROUPED
FILTERED, GROUPED & WINDOWED

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

THIS IS A SOURCE

TOGETHER, THEY FORM A PIPELINE

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 35 sec
ToLines Succeeded 52 sec
NeedsHelp Succeeded 22 sec
IsPopular Succeeded 34 sec
Sum.PerKey Succeeded 11 sec
Sum.PerKey2 Succeeded 2 min 31 sec
ToView Succeeded 12 sec
CompositeScore Succeeded 21 sec
Top_1000 Succeeded 3 sec
ToString Succeeded 0 sec
TextIO.Write Succeeded 1 sec

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.

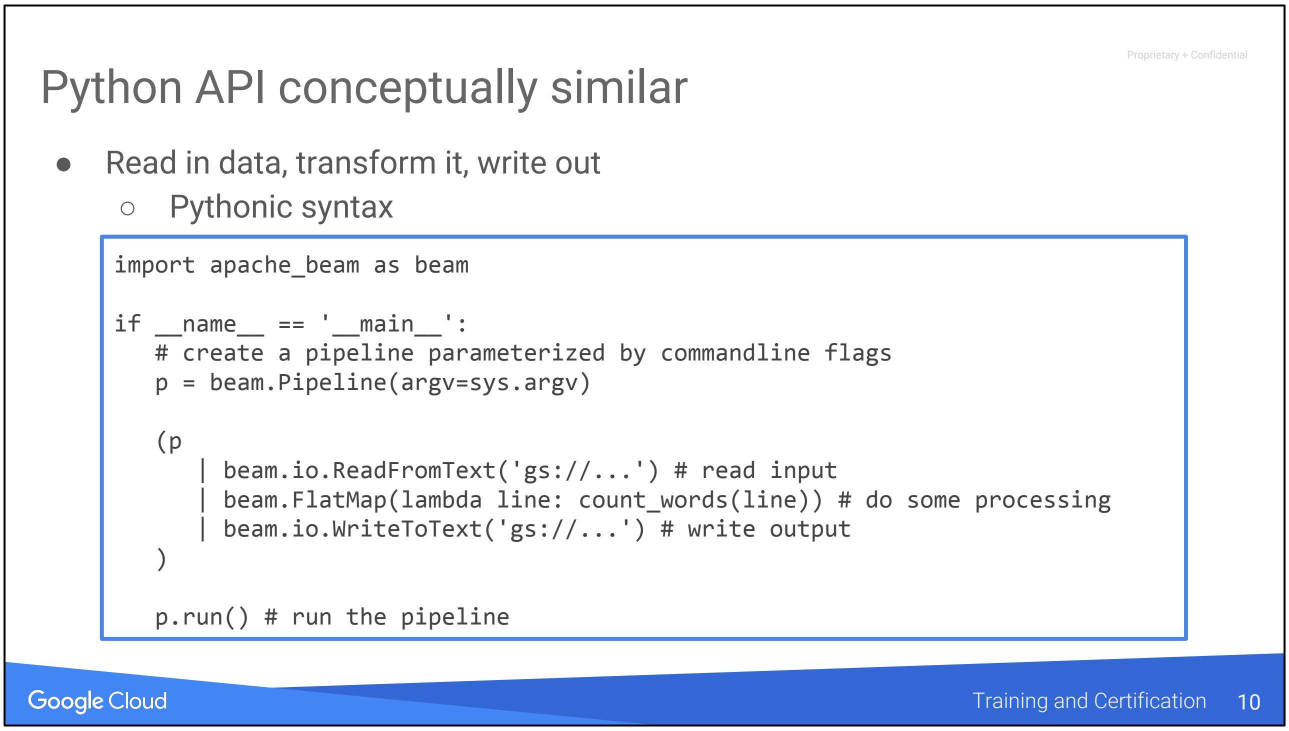
Elastic: unlike Dataproc, there is no need to launch a cluster. Like BigQuery in that respect

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();
}
```



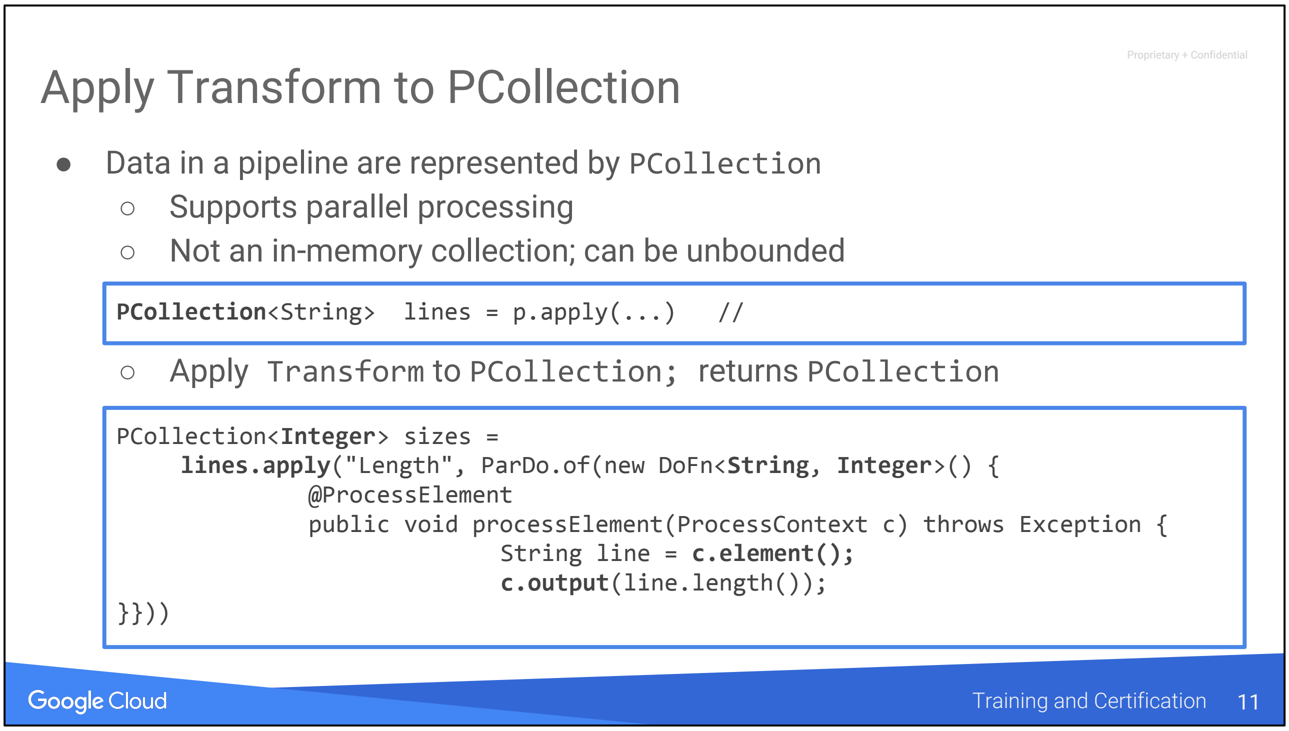
Python API conceptually similar

- 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.



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`

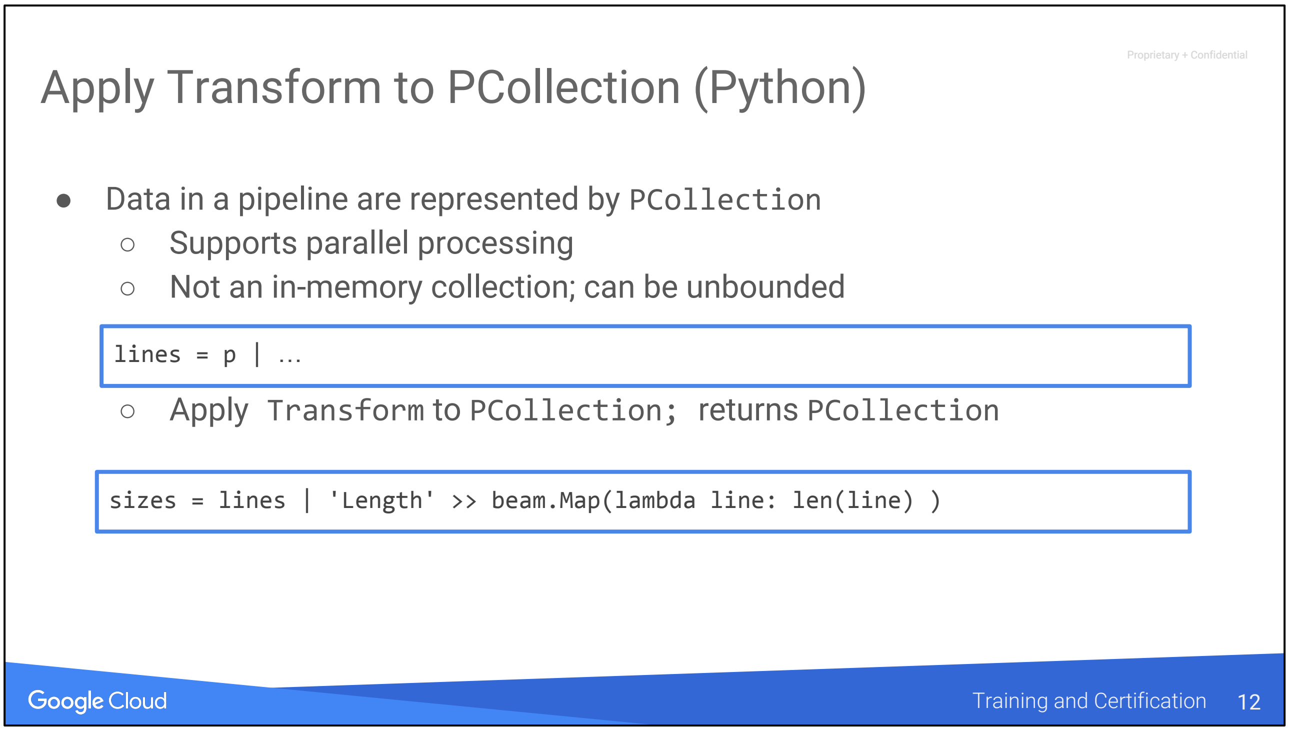
```
PCollection<Integer> sizes =
     lines.apply("Length", ParDo.of(new DoFn<String, Integer>() {
              @ProcessElement
              public void processElement(ProcessContext c) throws Exception {
                            String line = c.element();
                            c.output(line.length());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

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`

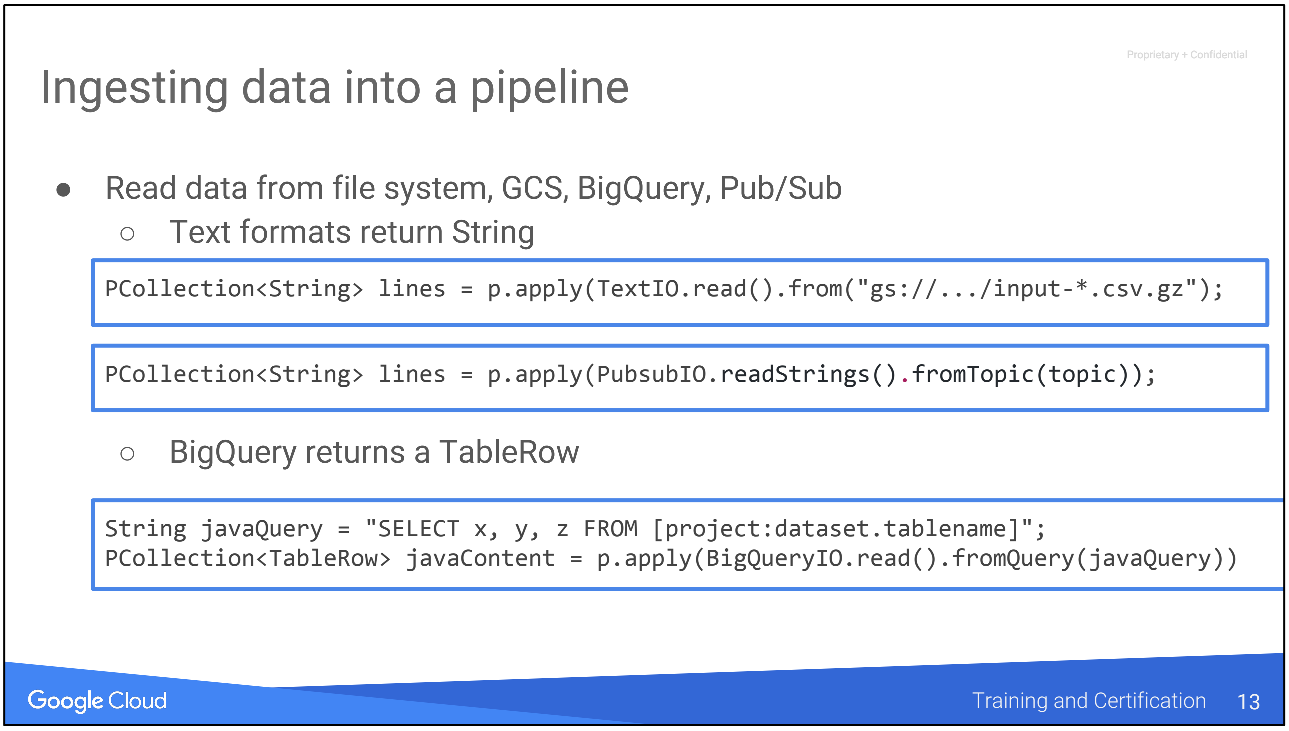
```
sizes = lines | 'Length' >> beam.Map(lambda line: len(line) )
```

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));
```

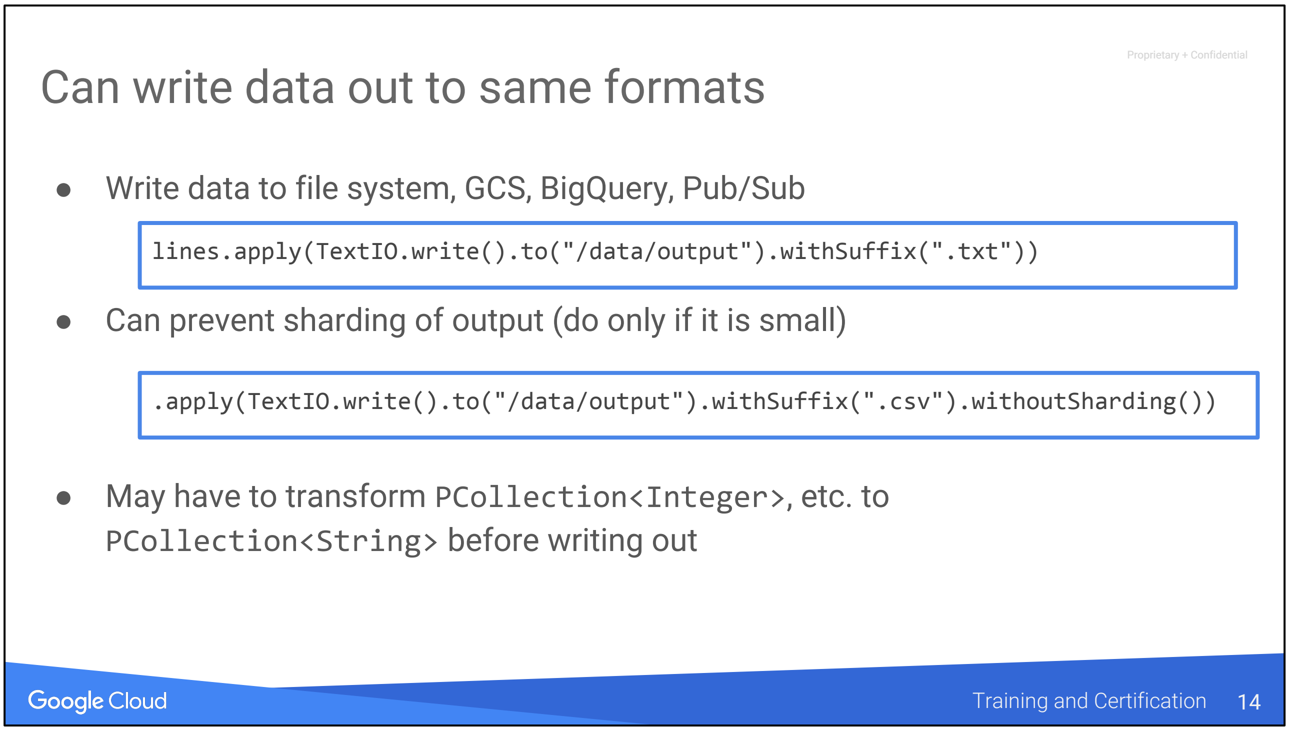
  - BigQuery returns a TableRow

```
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"))
```

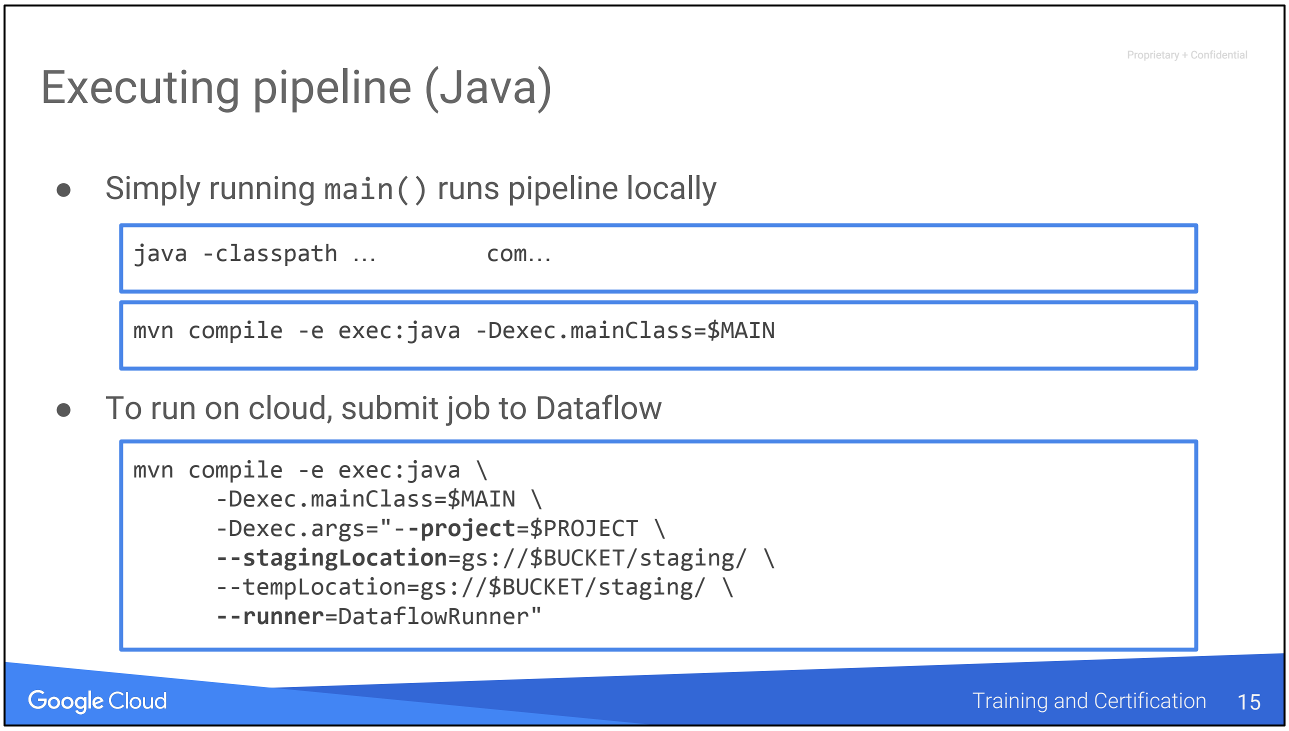
- Can prevent sharding of output (do only if it is small)

```
.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



Proprietary + Confidential

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

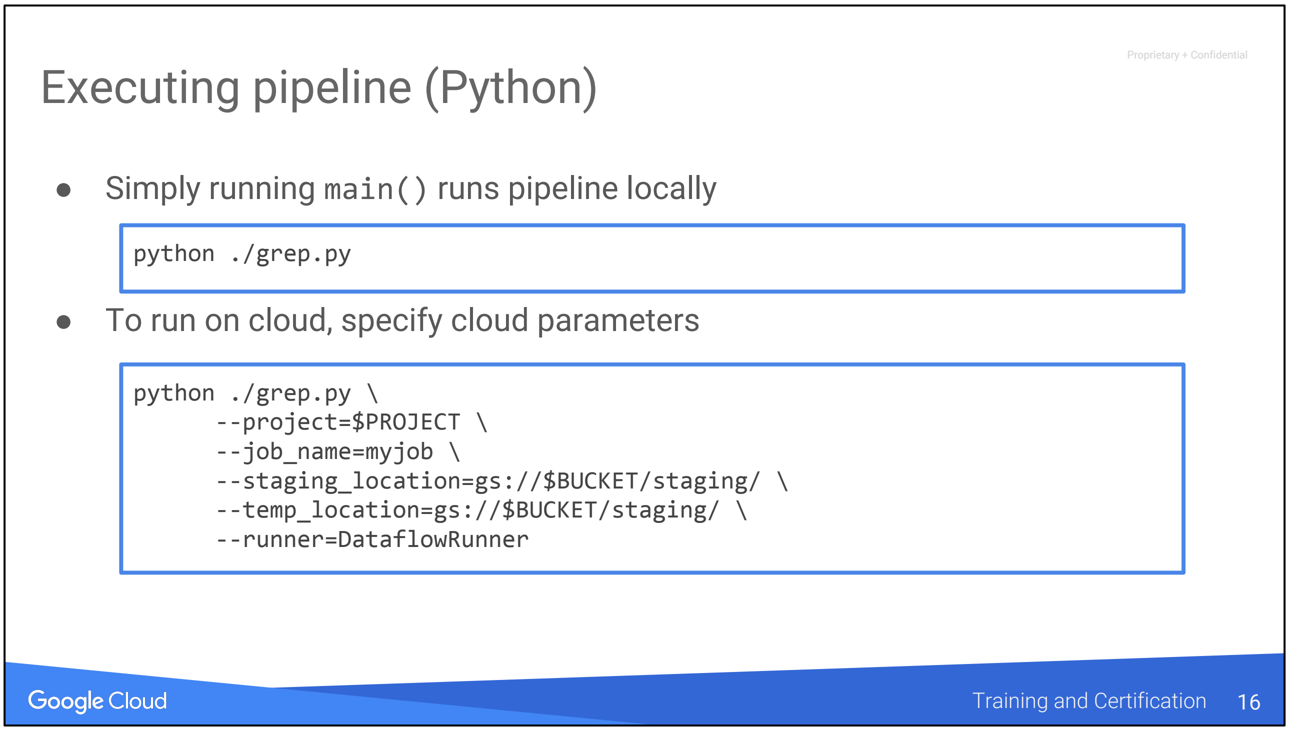
```
mvn compile -e exec:java \
      -Dexec.mainClass=$MAIN \
      -Dexec.args="--project=$PROJECT \
      --stagingLocation=gs://$BUCKET/staging/ \
      --tempLocation=gs://$BUCKET/staging/ \
      --runner=DataflowRunner"
```

Google Cloud

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.

Lab - Serverless Data Analysis (Java/Python) : Part 4

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:

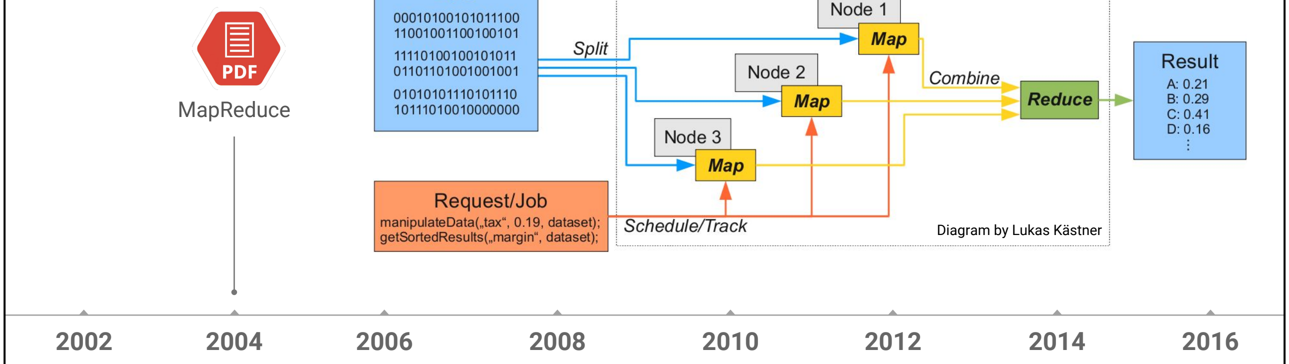
The pipeline they will build does a Grep -- looks for lines in Java files that have the keyword “import” in them.

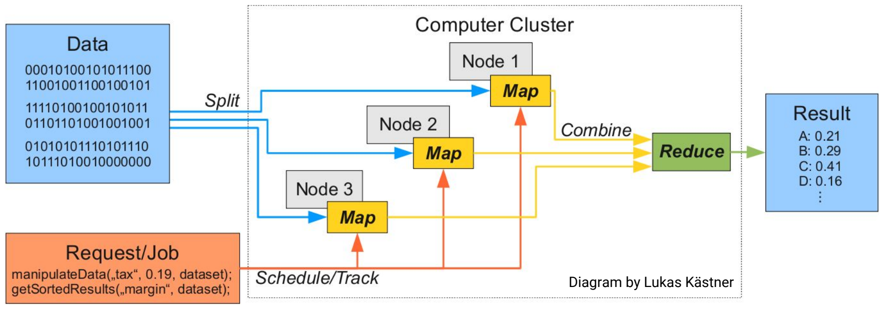
Image (cc0) https://pixabay.com/en/sieve-icing-sugar-kitchen-help-1262922/

Agenda

MapReduce in Dataflow + Lab

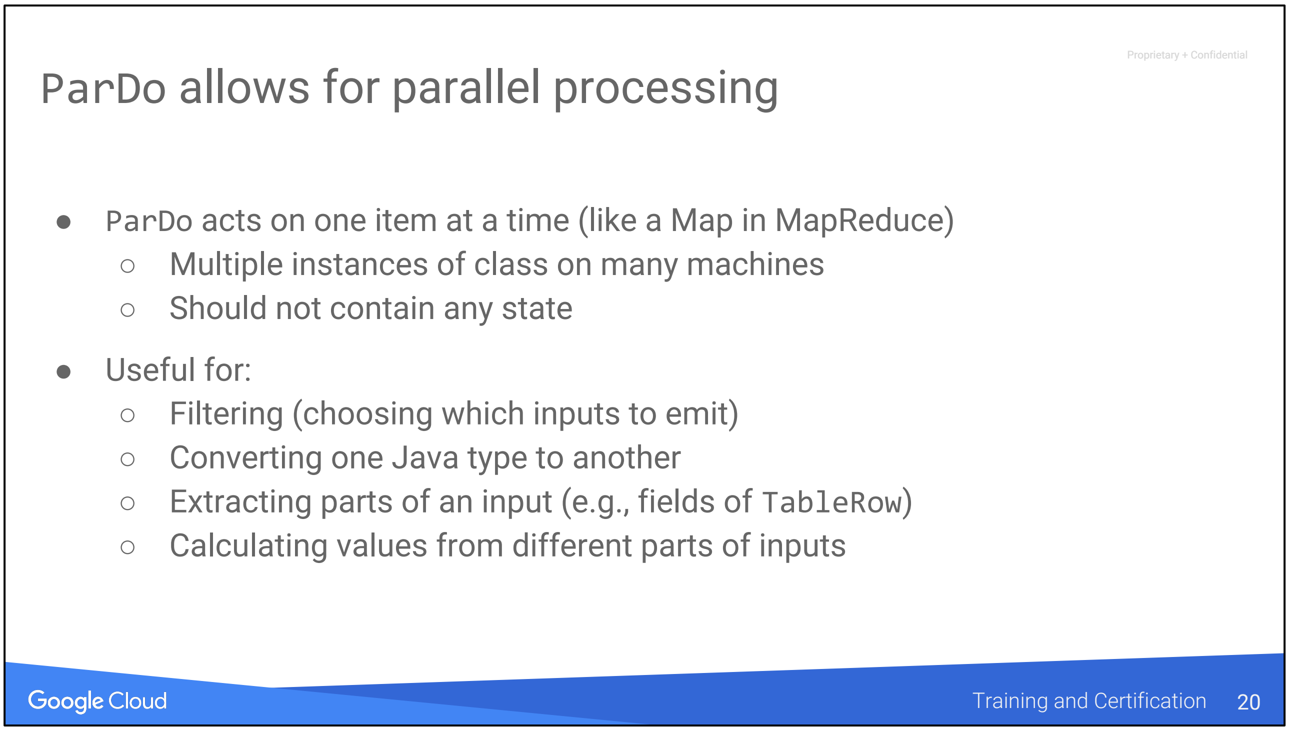
MapReduce approach splits Big Data so that each compute node processes data local to it





Notes:

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



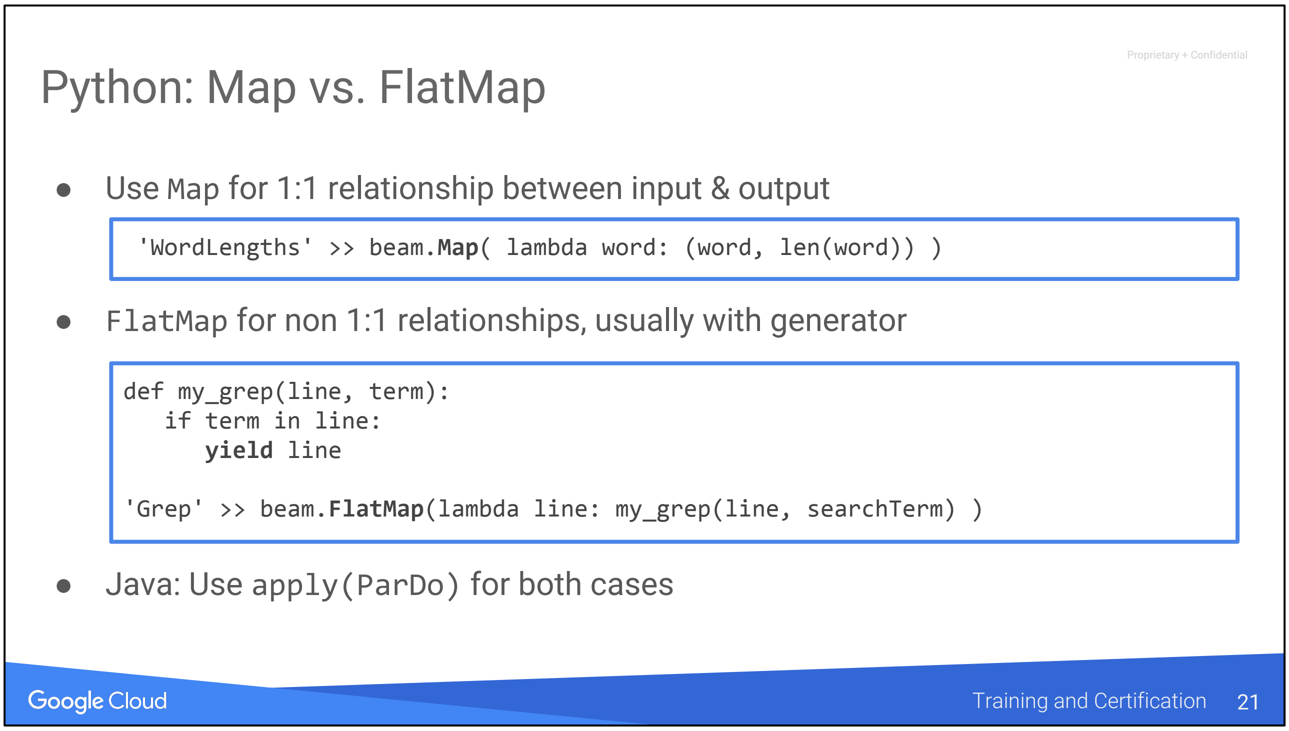
`ParDo` allows for parallel processing

- `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

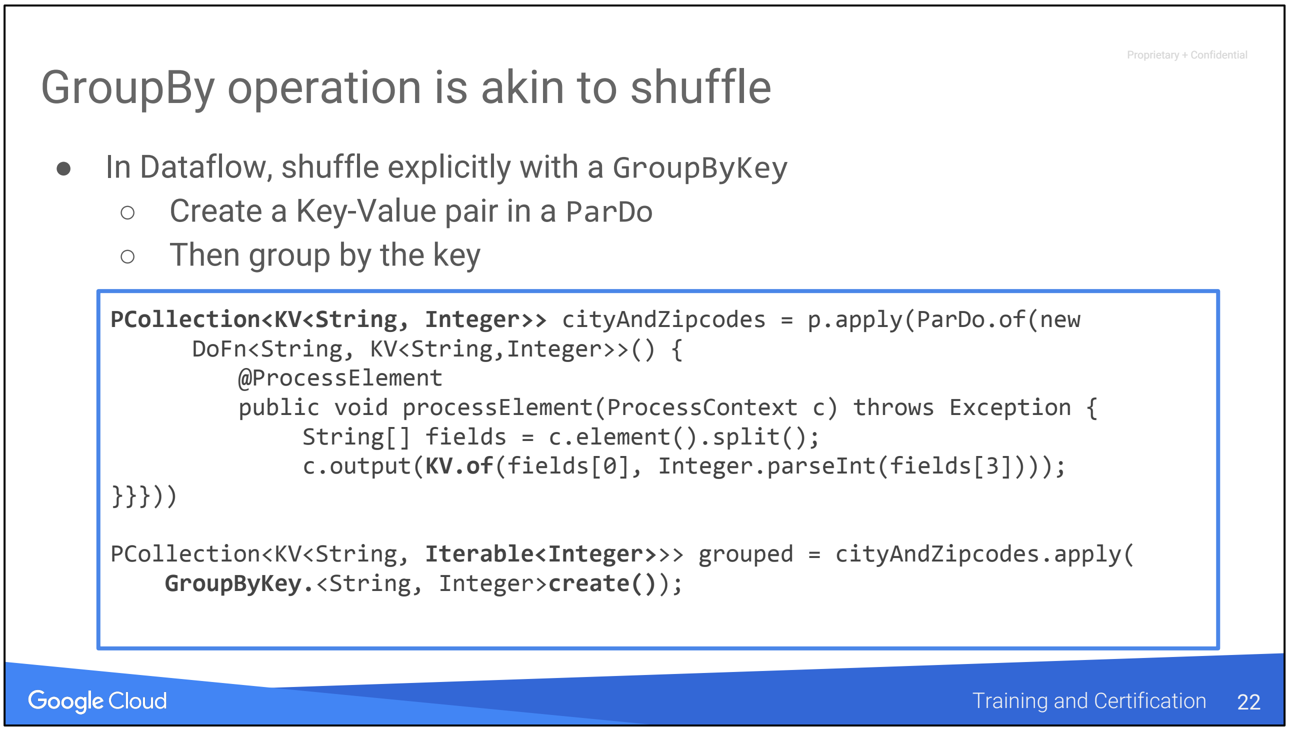
```
def my_grep(line, term):
   if term in line:
      yield line

'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

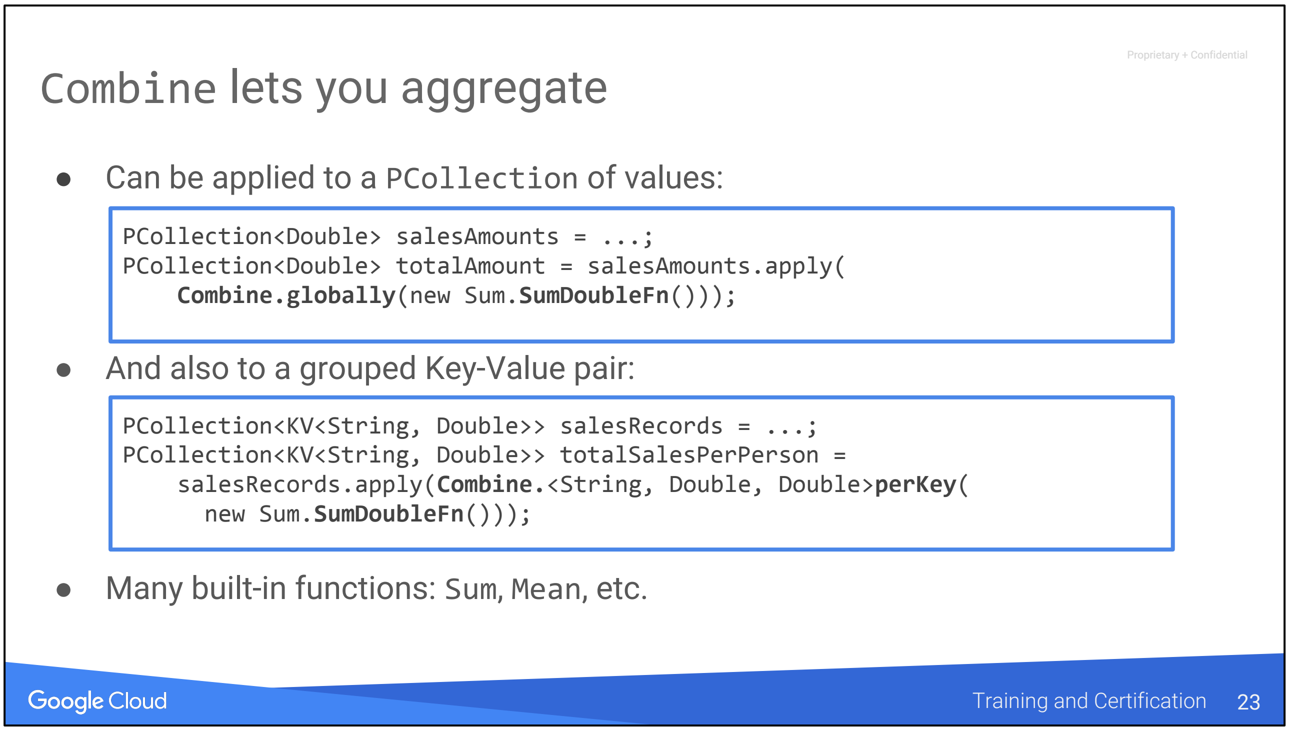
- In Dataflow, shuffle explicitly with a `GroupByKey`
  - Create a Key-Value pair in a `ParDo`
  - Then group by the key

```
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:

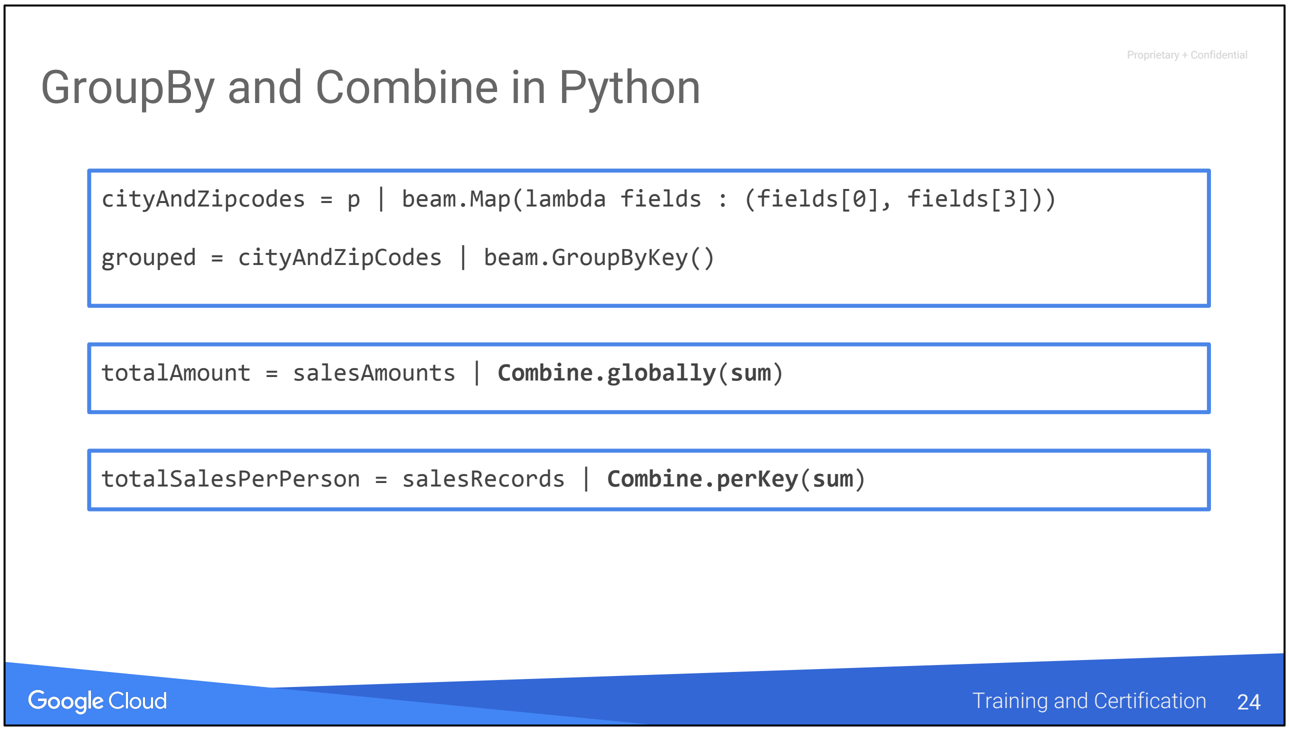
```
PCollection<KV<String, Double>> salesRecords = ...;
PCollection<KV<String, Double>> totalSalesPerPerson =
    salesRecords.apply(Combine.<String, Double, Double>perKey(
      new Sum.SumDoubleFn()));
```

- 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)
```

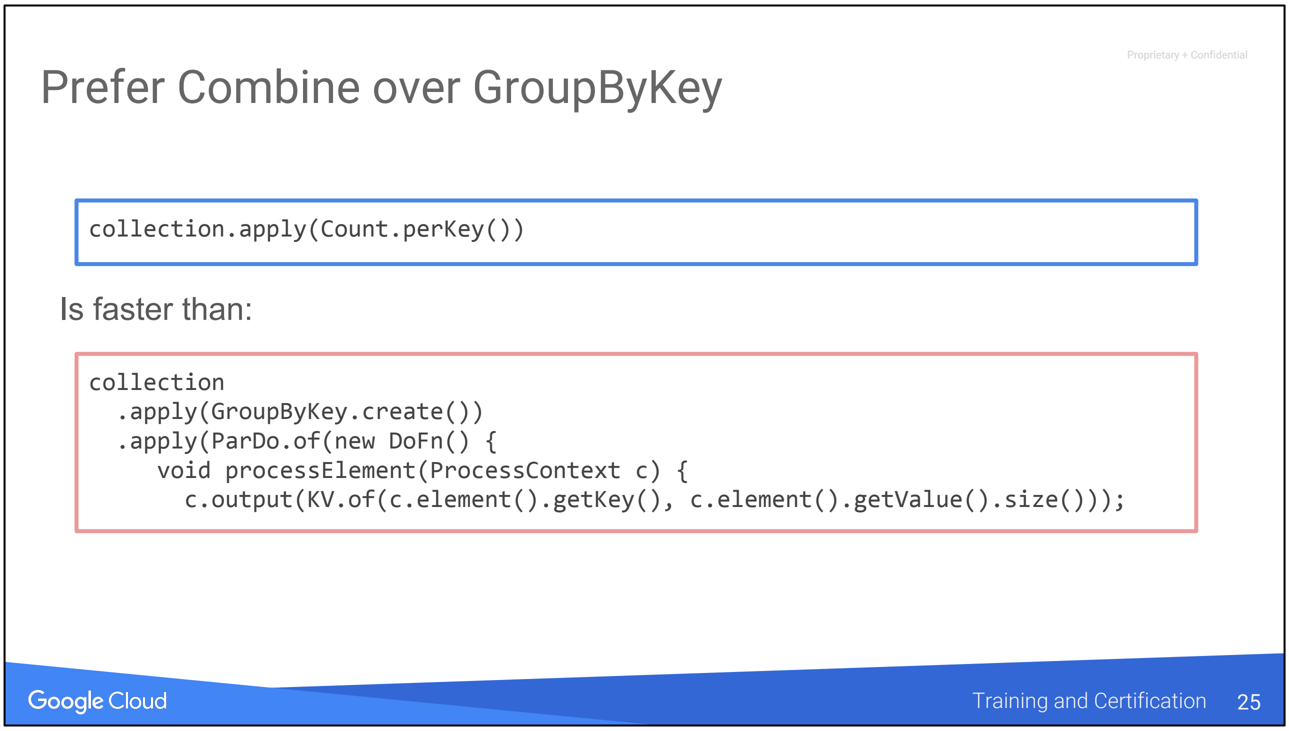
```
totalSalesPerPerson = salesRecords | Combine.perKey(sum)
```

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:

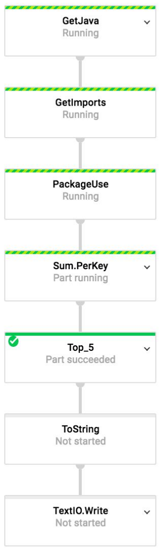
In cases where you can use either a GroupBy or a Combine, use a Combine. The version using Count is orders of magnitude faster, because Dataflow can parallelize the operation on multiple machines, which is impossible in the GroupByKey version.

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

Lab - Serverless Data Analysis (Java/Python) : Part 5

In this lab, you will learn how to:

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



Notes:

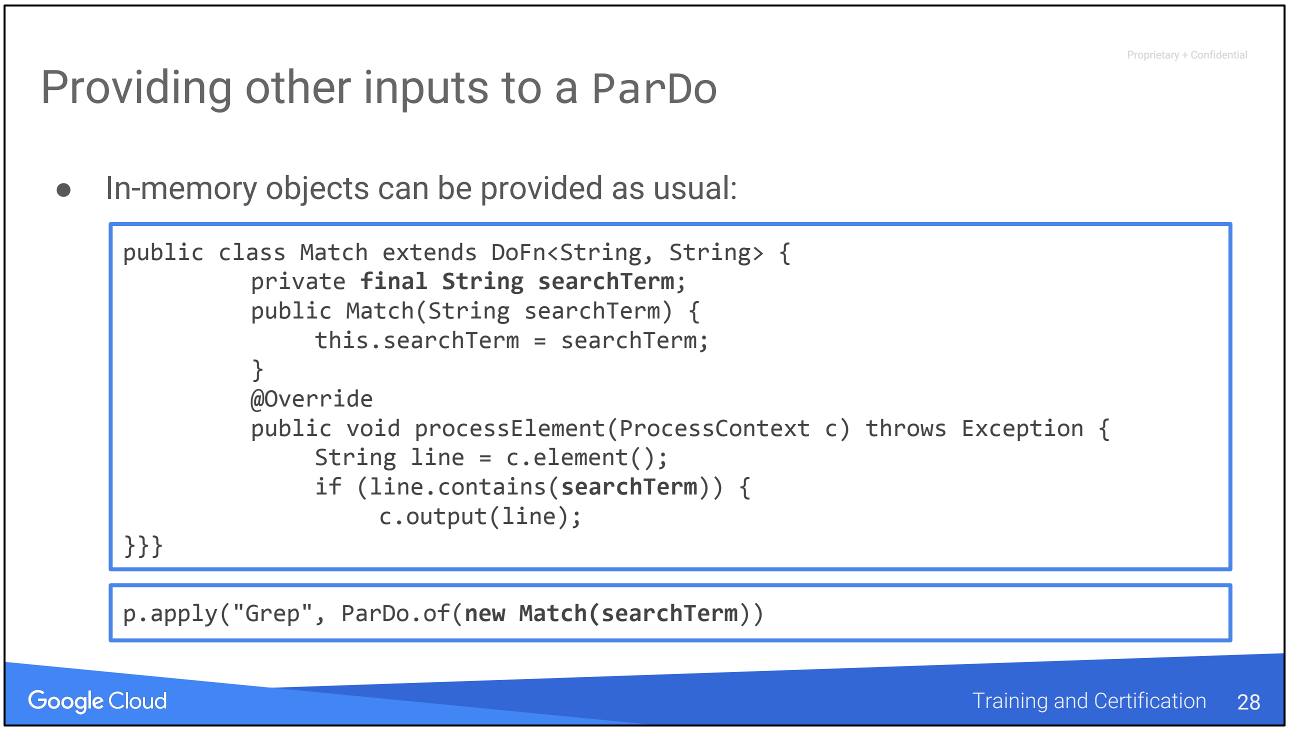
The pipeline identifies the 5 most popular imported packages.

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

Image (cc0) https://pixabay.com/en/bullring-bullfighters-arena-314361/

Agenda

Side inputs + Lab



Providing other inputs to a ParDo

- In-memory objects can be provided as usual:

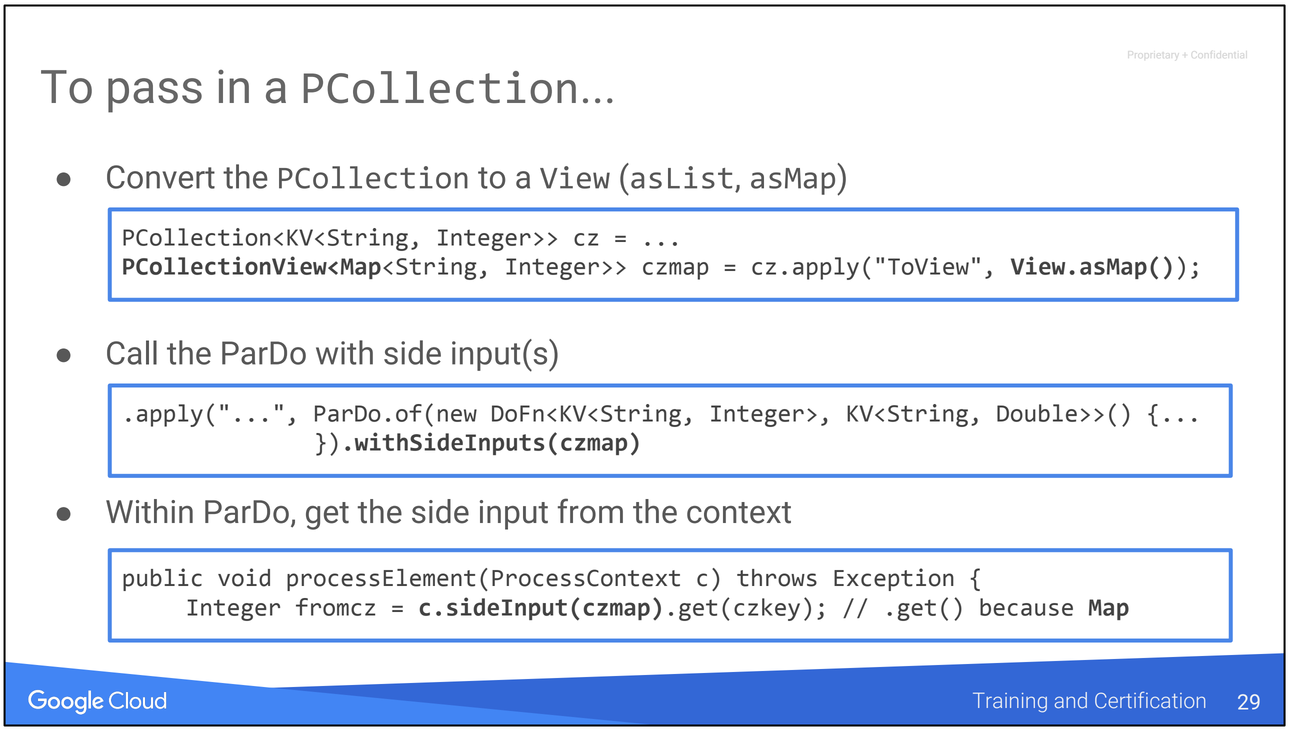
```
public class Match extends DoFn<String, String> {
         private final String searchTerm;
         public Match(String searchTerm) {
              this.searchTerm = searchTerm;
         }
         @Override
         public void processElement(ProcessContext c) throws Exception {
              String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

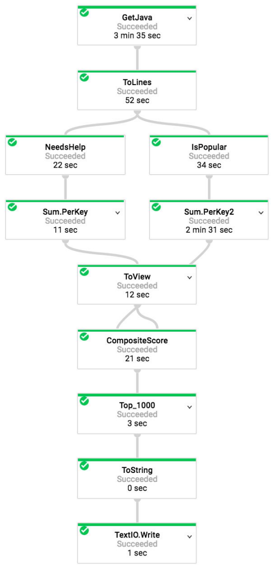
Notes:

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

Lab - Serverless Data Analysis (Java/Python) : Part 6

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.

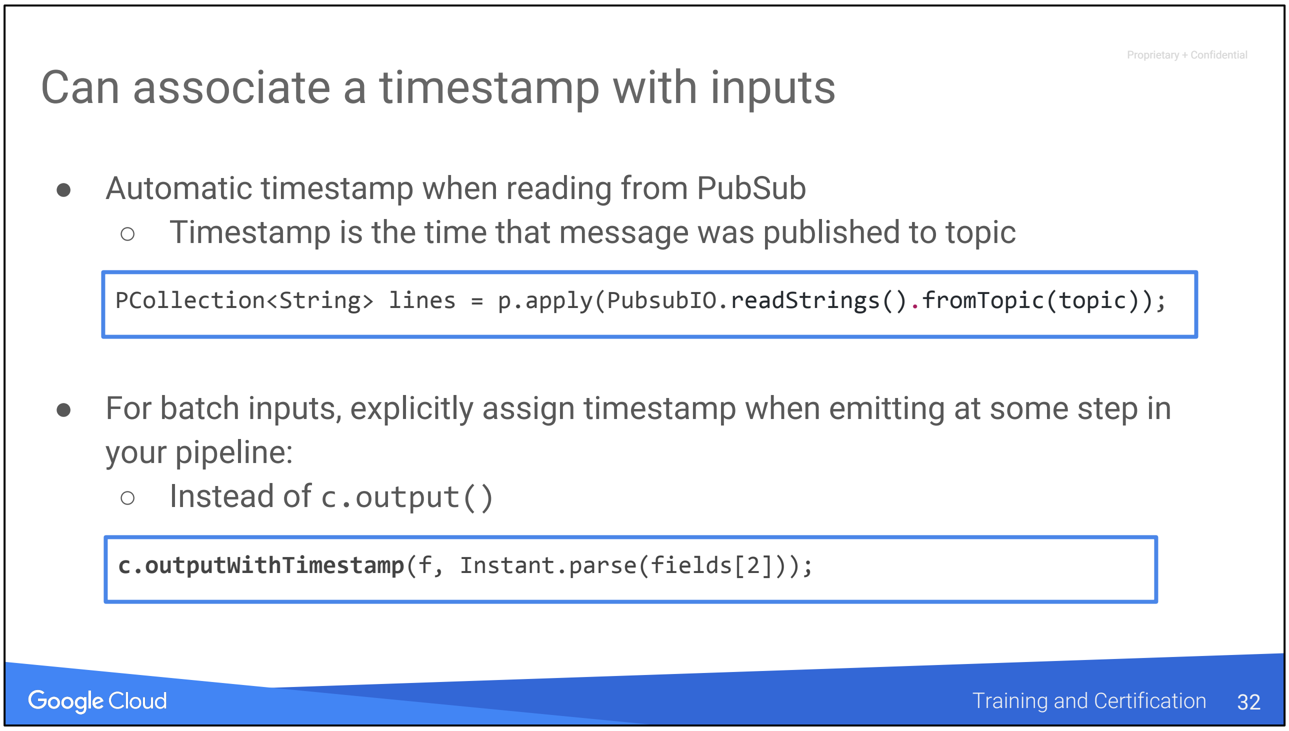
Depending on time constraints and class interest, your instructor might choose to make this lab a demo and walk through the code.

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

Streaming



Can associate a timestamp with inputs

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

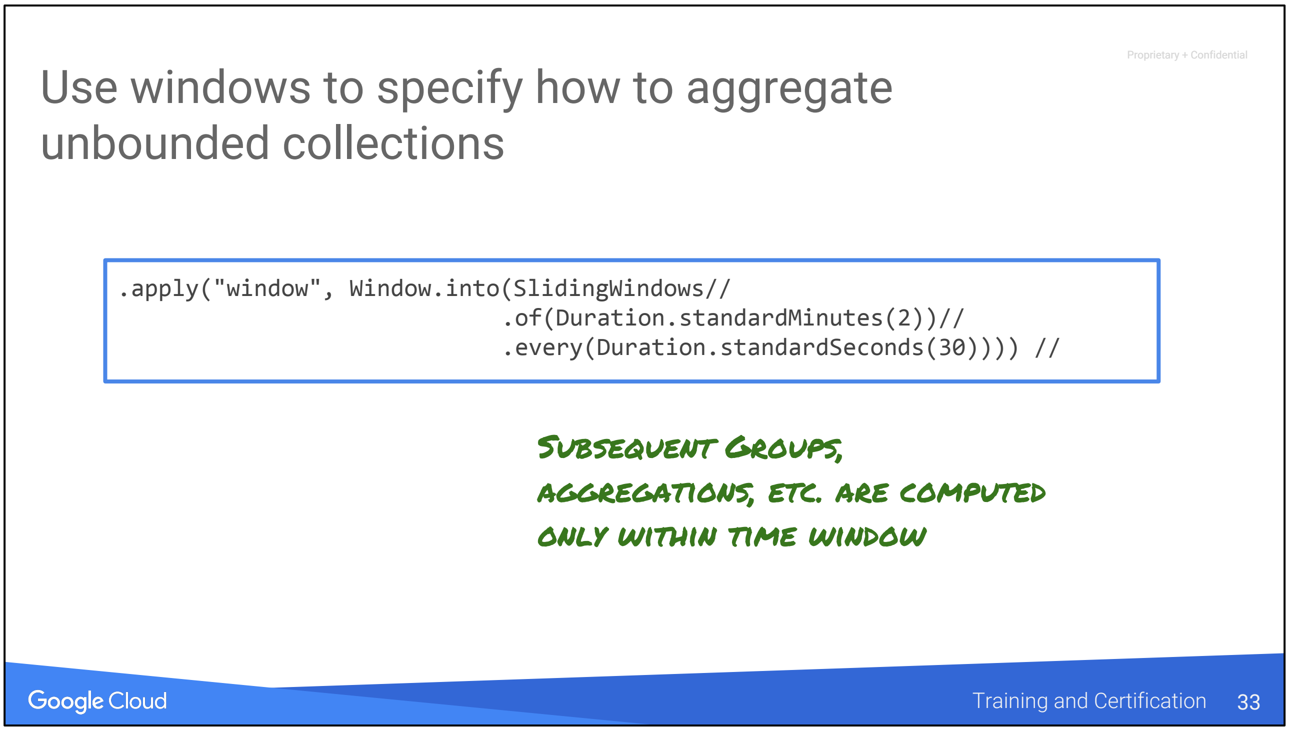
```
PCollection<String> lines = p.apply(PubsubIO.readStrings().fromTopic(topic));
```

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

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

Notes:

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



Use windows to specify how to aggregate unbounded collections

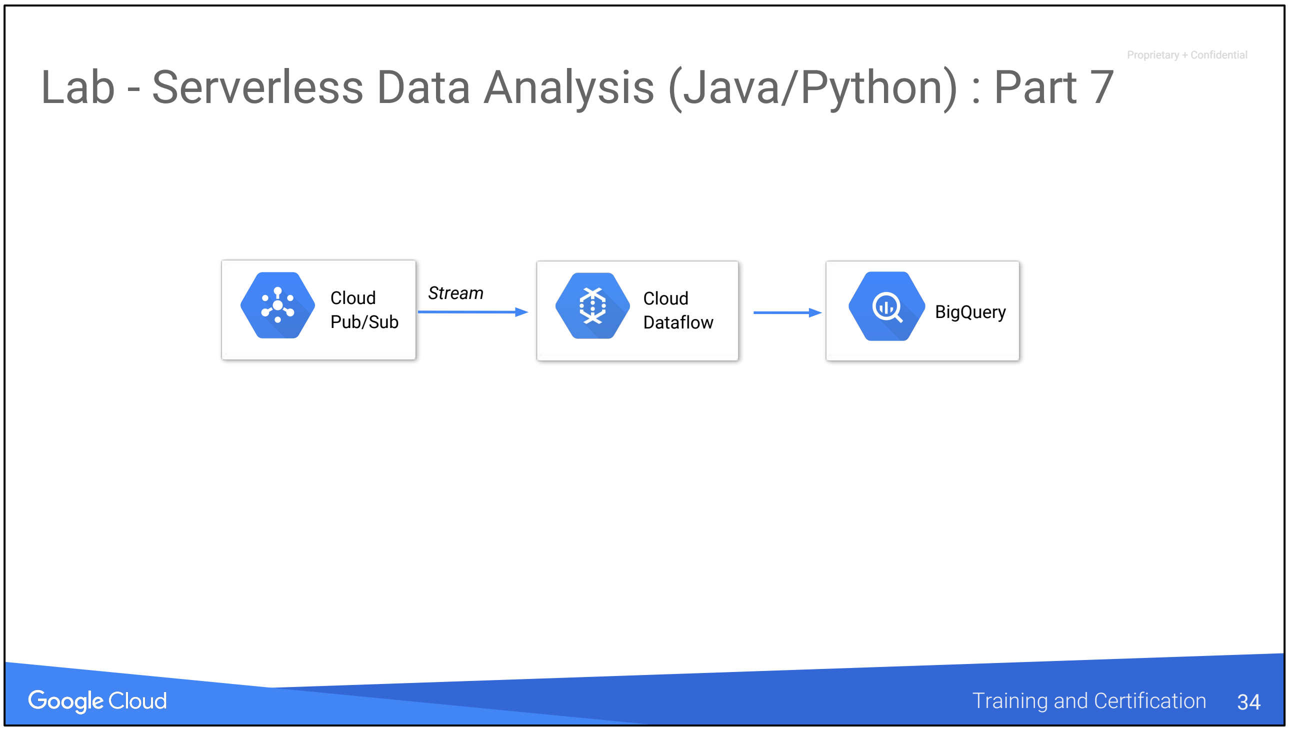
```
.apply("window", Window.into(SlidingWindows//
                              .of(Duration.standardMinutes(2))//
                              .every(Duration.standardSeconds(30)))) //
```

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

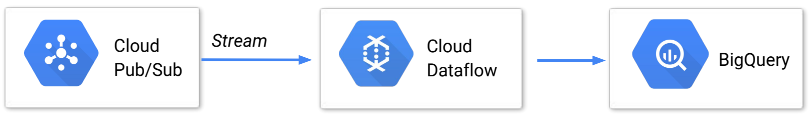
Notes:

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

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



Lab - Serverless Data Analysis (Java/Python) : Part 7



Notes:

This lab shows the top part of the reference architecture for GCP.

On GCP console, go to Pub/Sub and create a topic named streamdemo.

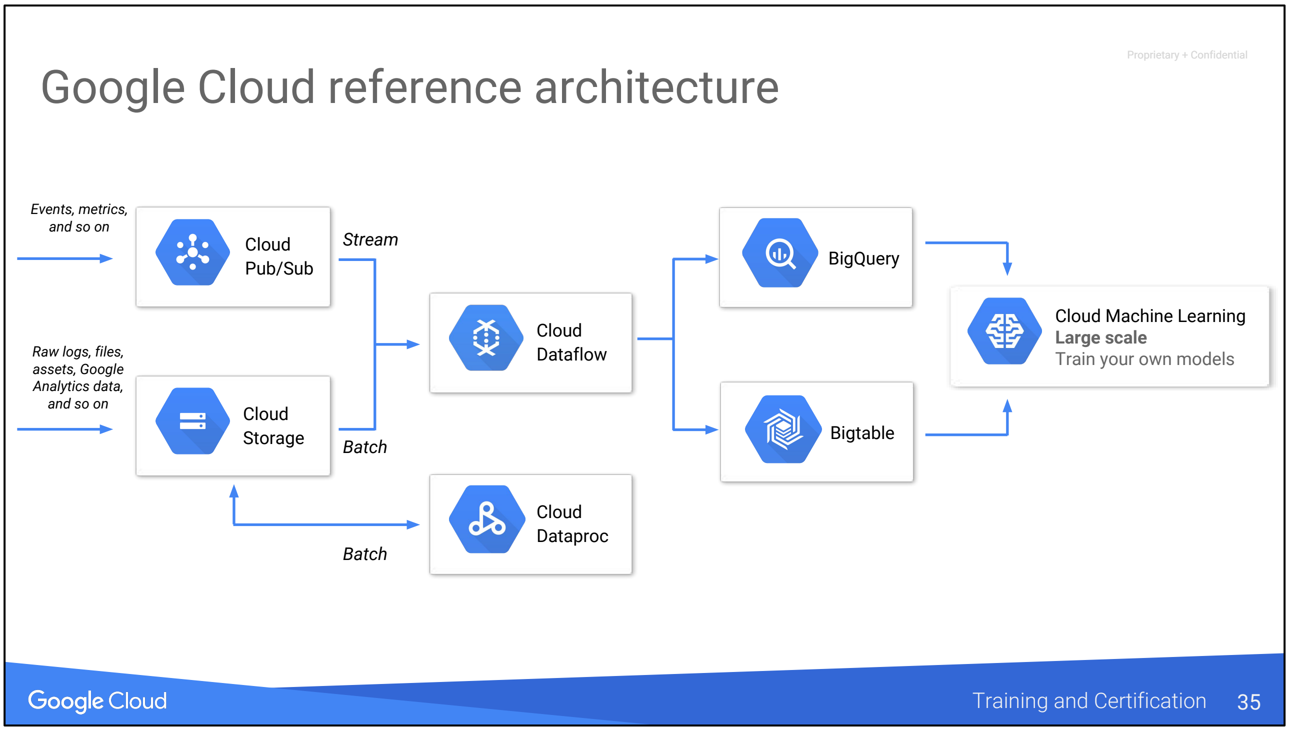
Also, go to BigQuery console and create a dataset named demos.

The Dataflow client is in lab2 of the Java side. Use ./run_oncloud4.sh

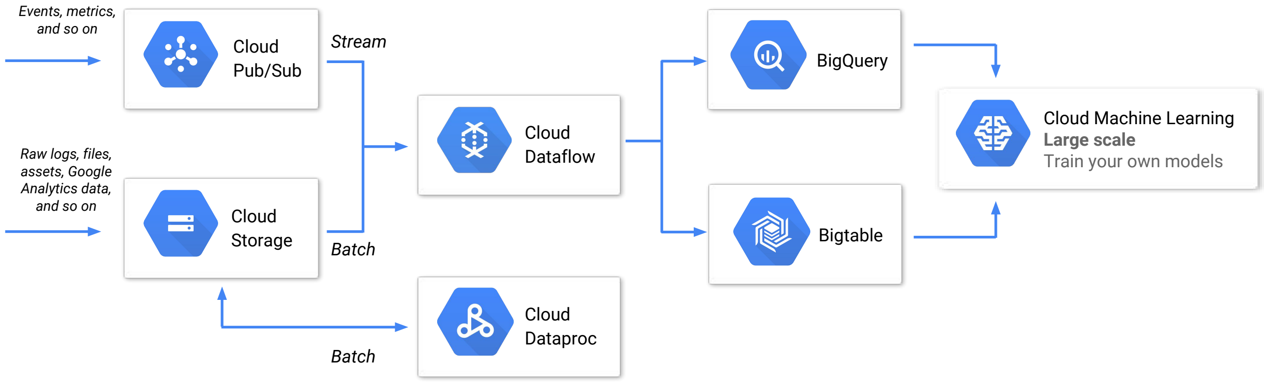
Dataflow automatically creates a subscription is to ensure that you don’t miss any records

Use the GCP console to publish plain-text messages into Pub/Sub.

After 30s, you should be analyze the data in BQ even as it is streaming in.



Google Cloud reference architecture

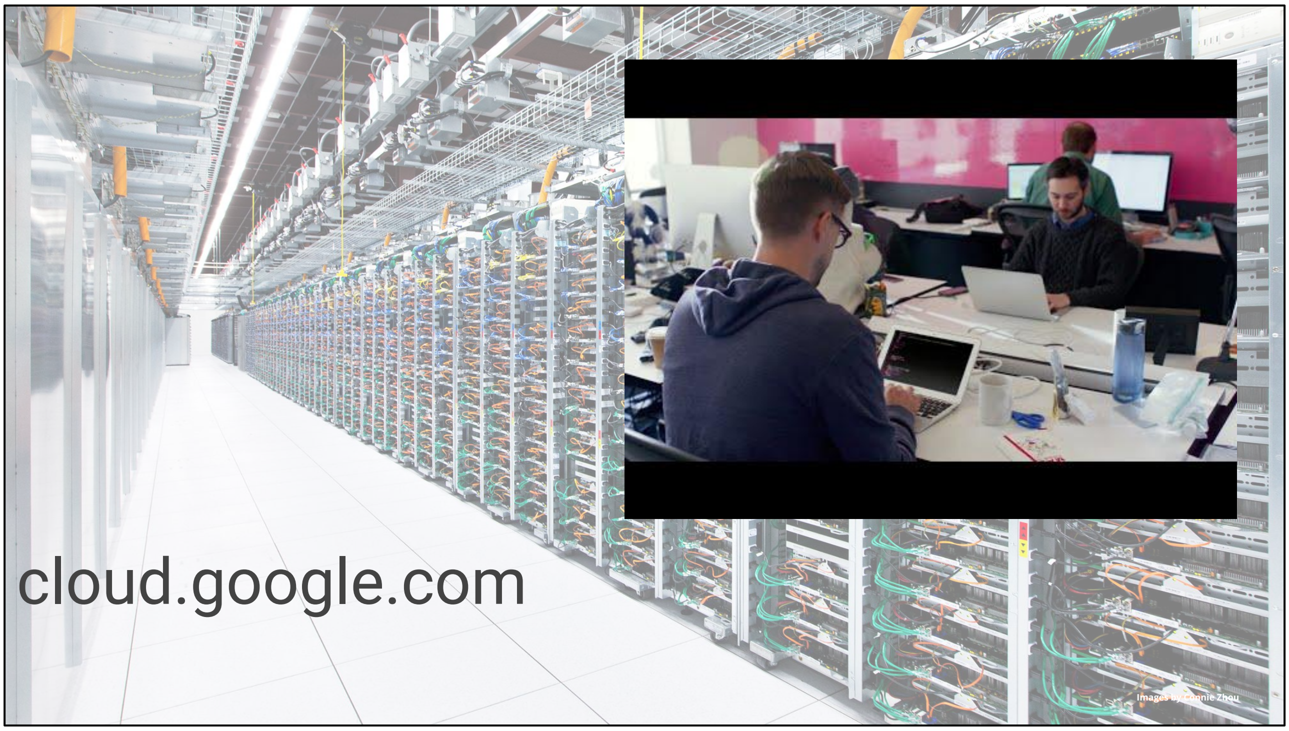


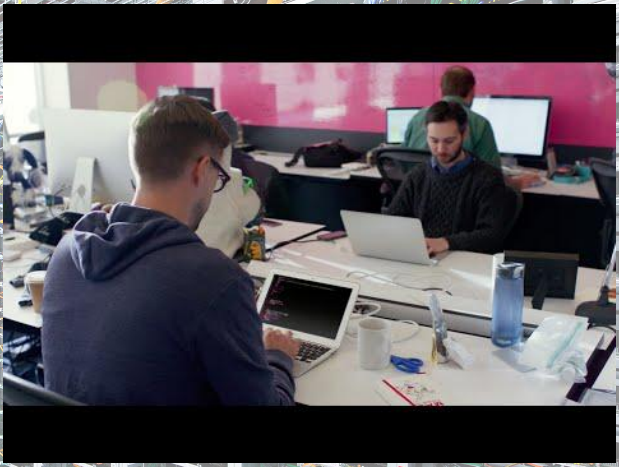
Notes:

The previous lab showed the top part of the reference architecture for GCP. We'll look at Cloud ML in the next part of this track.

Resources

- Cloud Dataflow
  https://cloud.google.com/dataflow/
- 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





cloud.google.com

Notes:

How Spotify uses Dataflow. 1 minute, 47 seconds.