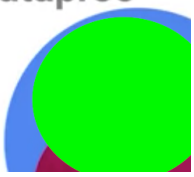
On-premises Hadoop clusters have a number of limitations

- ✗ Not elastic
- ✗ Hard to scale fast
- ✗ Have capacity limits
- ✗ Have no separation between storage and compute resources

Cloud Dataproc simplifies Hadoop workloads on GCP

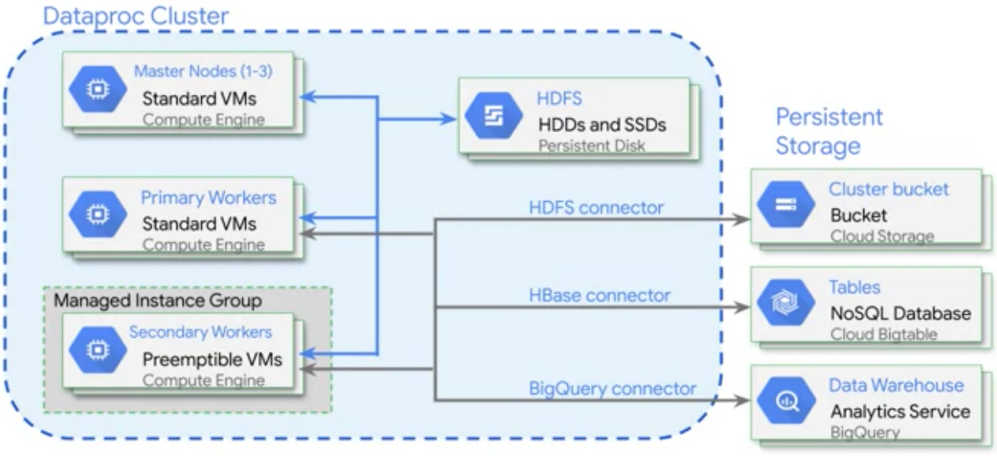
- ✓ Built-in support for Hadoop
- ✓ Managed hardware and configuration
- ✓ Simplified version management
- ✓ Flexible job configuration

Cloud Dataproc

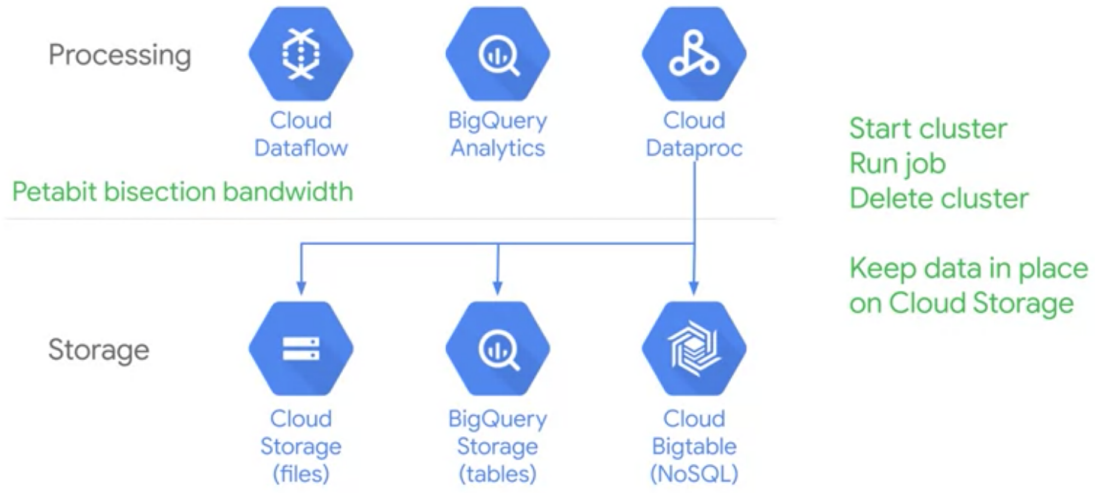


Cloud Dataproc is a managed service for running Hadoop and Spark data processing workloads

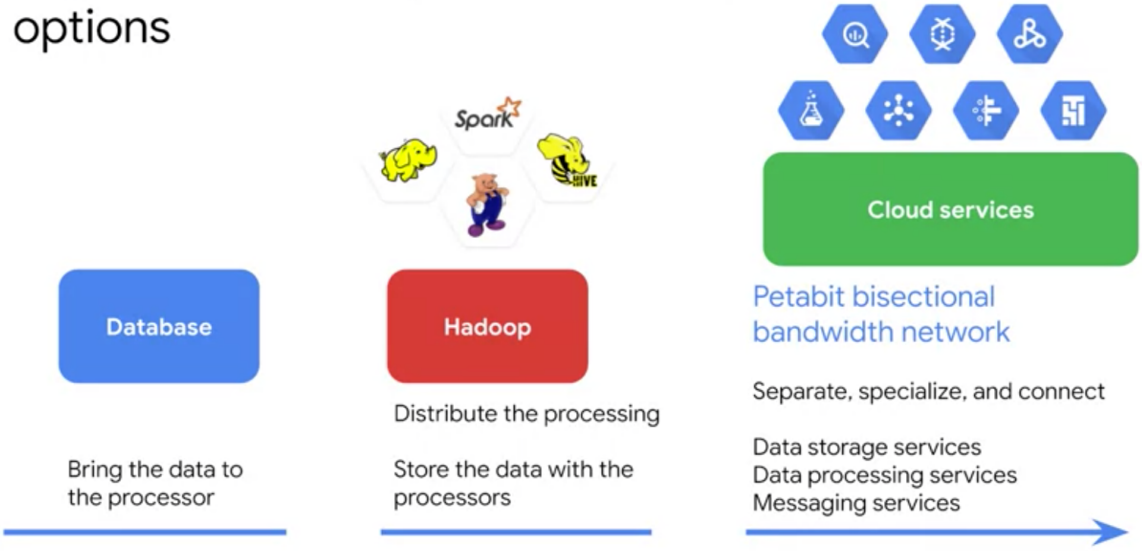
Dataproc cluster can read/write to GCP storage products



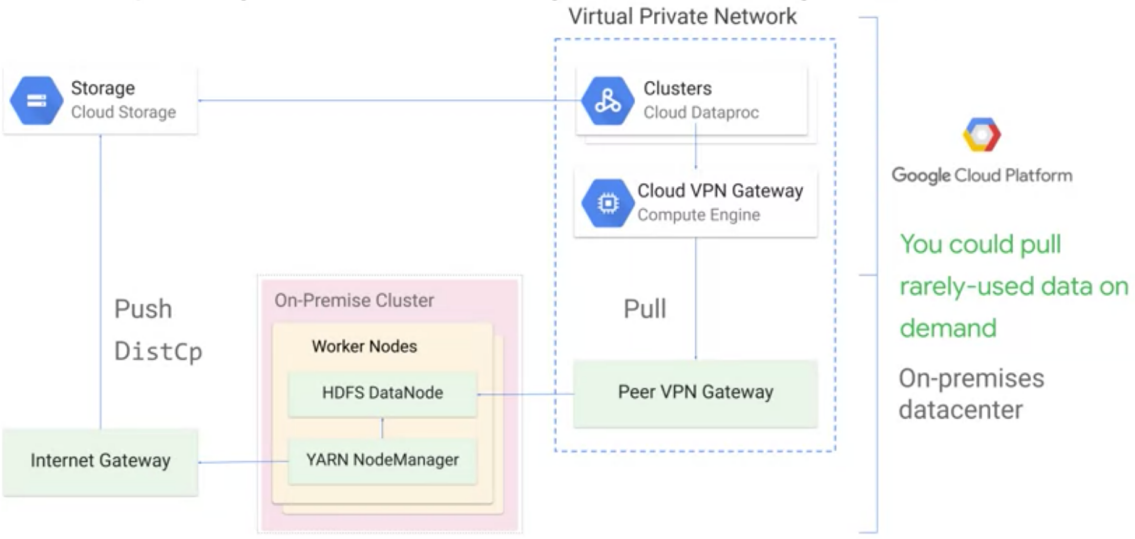
On GCP, Jupiter, and Colossus make separation of compute and storage possible



Separation of compute and storage enables better options



DistCp on-prem data that you will always need



https://hadoop.apache.org/docs/current/hadoop-distcp/DistCp.html

Local HDFS is necessary at times

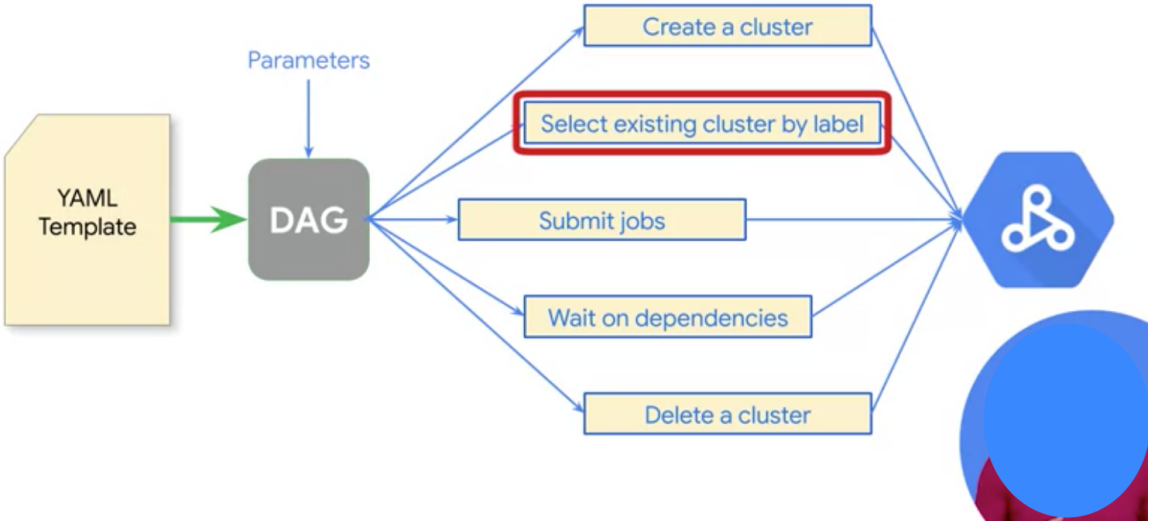
Local HDFS is a good option if:

- Your jobs require a lot of metadata operations--for example, you have thousands of partitions and directories, and each file size is relatively small
- You modify the HDFS data frequently or you rename directories (Cloud Storage objects are immutable, so renaming a directory is an expensive operation because it consists of copying all objects to a new key and deleting them afterwards)
- You heavily use the append operation on the HDFS files
- You have workloads that involve heavy I/O. For example, you have a lot of partitioned writes, such as the following

  ```
  spark.read().write.partitionBy(...).parquet("gs://")
  ```

- You have I/O workloads that are especially sensitive to latency. For example, you require single-digit millisecond latency per storage operation

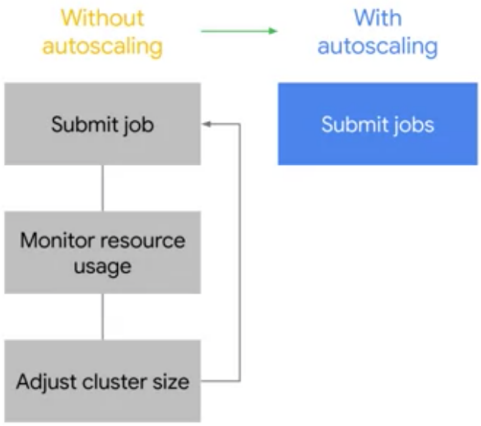
Cloud Dataproc Workflow Template

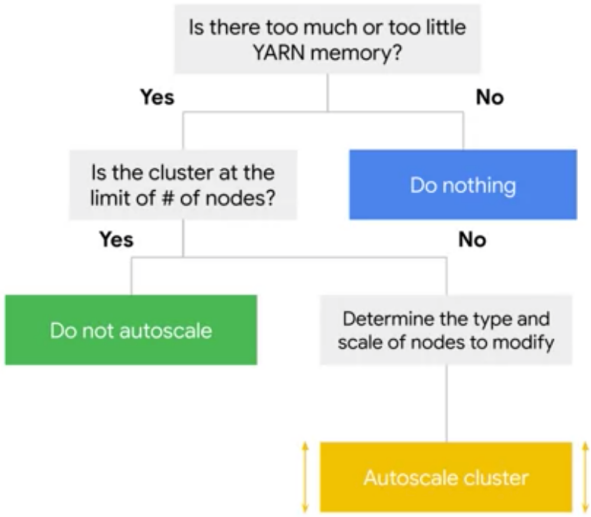


Cloud Dataproc workflow templates

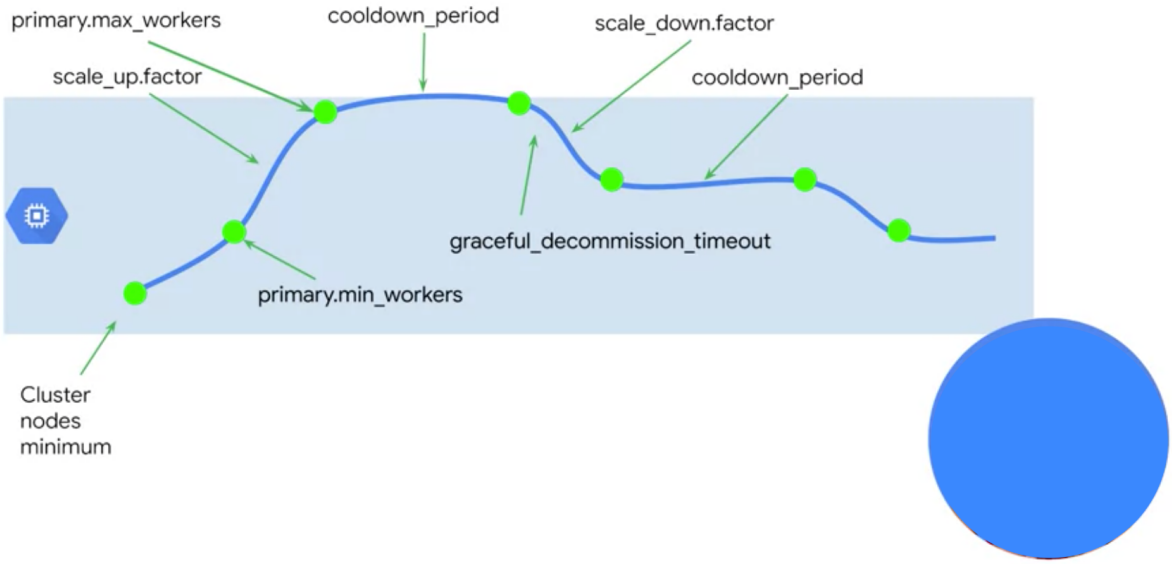
```
# the things we need pip-installed on the cluster
STARTUP_SCRIPT=gs://${BUCKET}/sparktobq/startup_script.sh
echo "pip install --upgrade --quiet google-compute-engine google-cloud-storage matplotlib" >
/tmp/startup_script.sh
gsutil cp /tmp/startup_script.sh $STARTUP_SCRIPT

# create new cluster for job
gcloud dataproc workflow-templates set-managed-cluster $TEMPLATE \
    --master-machine-type $MACHINE_TYPE \
    --worker-machine-type $MACHINE_TYPE \
    --initialization-actions $STARTUP_SCRIPT \
    --num-workers 2 \
    --image-version 1.4 \
    --cluster-name $CLUSTER

# steps in job
gcloud dataproc workflow-templates add-job \
  pyspark gs://$BUCKET/spark_analysis.py \
  --step-id create-report \
  --workflow-template $TEMPLATE \
  -- --bucket=$BUCKET


# submit workflow template
gcloud beta dataproc workflow-templates instantiate $TEMPLATE
```

Cloud Dataproc autoscaling workflow





How Cloud Dataproc Autoscaling works



Cloud Fusion (Beta)

Cloud Data Fusion is a fully-managed, cloud native, enterprise data integration service for quickly building and managing data pipelines.

Developer, Data Scientist, and Business Analyst

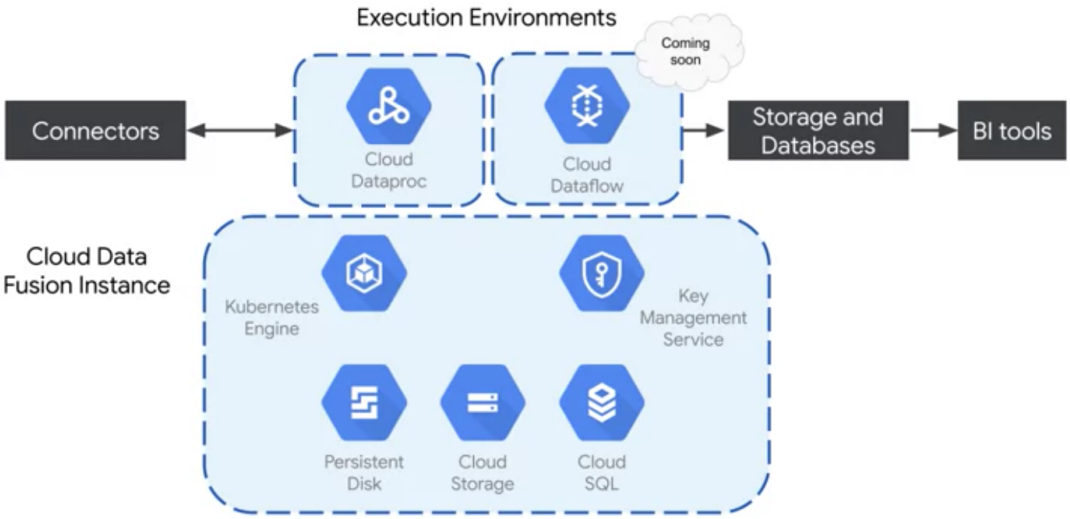
1. Need to cleanse, match, de-dupe, blend, transform, partition, transfer, standardize, automate, & monitor
2. Use Cloud Data Fusion to visually build integration pipeline, test, debug and deploy
3. Run it at scale on GCP, operationalize (monitor, report) pipelines, inspect rich integration metadata

Data Fusion creates ephemeral execution environments to run pipelines

Cloud Dataproc

Cloud Data Fusion translates your visually built pipeline into an Apache Spark or MapReduce program that executes transformations on an ephemeral Cloud Dataproc cluster in parallel. This enables you to easily execute complex transformations over vast quantities of data in a scalable, reliable manner, without having to wrestle with infrastructure and technology.

How does Cloud Data Fusion work?



Build data pipelines with a friendly UI

Rich graphical interface

100+ plugins - connectors, transforms, & actions

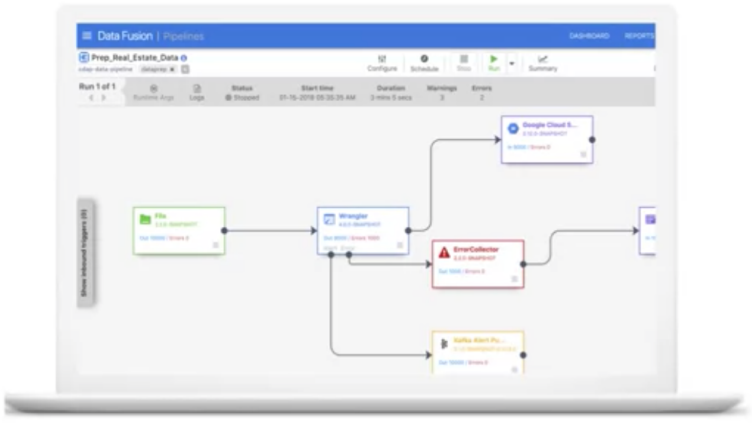
Code free visual transformations

1000+ transforms, data quality

Test and debug pipeline

Pre-built pipelines

Developer SDK



Integration Metadata

Tags and Properties support

Pipeline

Dataset

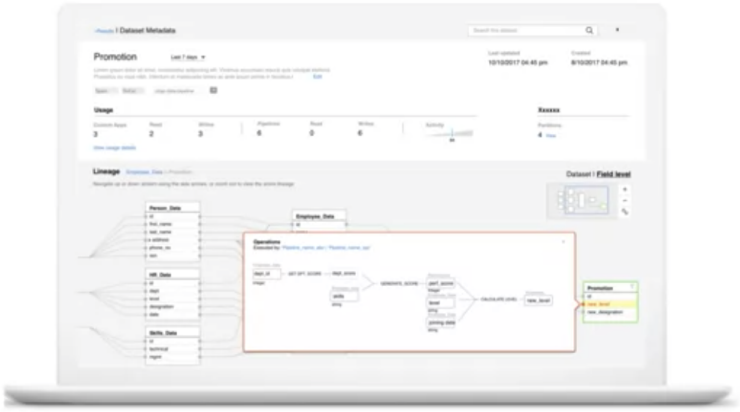
Schema

Search integrated entities by

Keyword

Schema name and type

Dataset level and Field level Lineage



Components of Cloud Data Fusion

Wrangler -- Framework

- Data Preparation for on-boarding new sources and datasets
- Perform Data Transformations, Data Quality checks with visual feedback
- Extend the Wrangler by building new user defined directives
- Integrates with Data Pipeline for operationalizing transformations

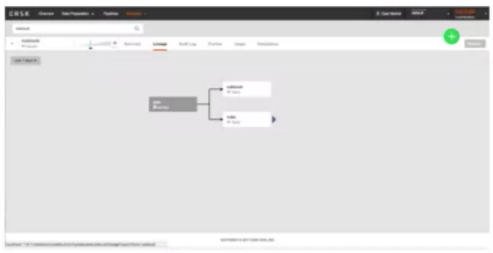
Data Pipeline -- Framework

- User interface for building complex data workflows
- Join, Lookup, Aggregate, Filtering data in-flight
- Building complex workflows with 100s of connectors
- Extend Data Pipeline using simple APIs
- Integrates with Dataprep, Rule Engine, and Metadata Aggregator

Components of Cloud Data Fusion

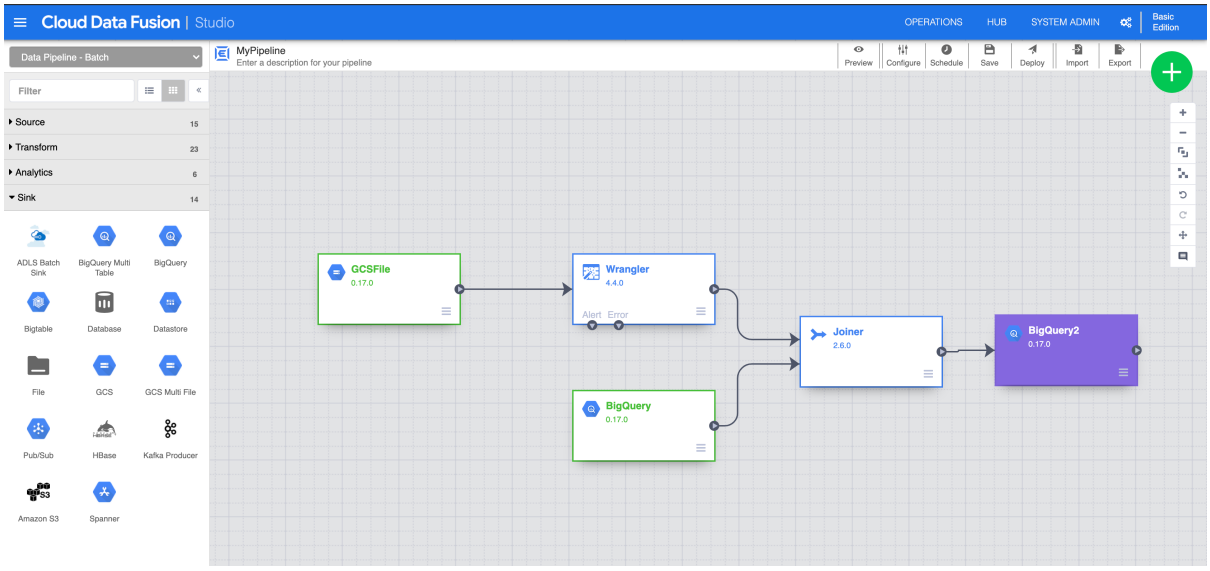
Rules Engine -- Tool*

- Business Data Transformations and checks codified for business users
- Define Complex rules using intuitive and simple to use user interface
- Logically group Rules in Rulebook and trigger or schedule processing
- Integrates with Data Pipeline for operationalizing Rules

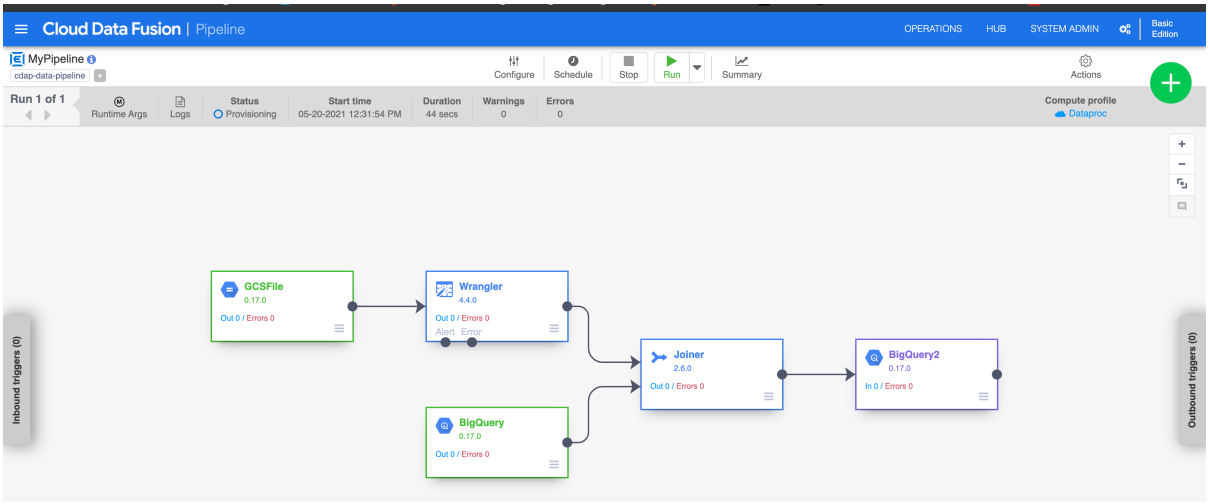


Metadata Aggregator -- Tool

- Aggregate Business, Technical, and Operational Metadata
- Track the flow of data (Lineage) for richer data needed for governance
- Create Data Dictionary and Metadata Repository
- Integrate with enterprise MDM solutions
- Integrates with Data Pipeline, Rules Engine



Deploy and Run Pipeline –

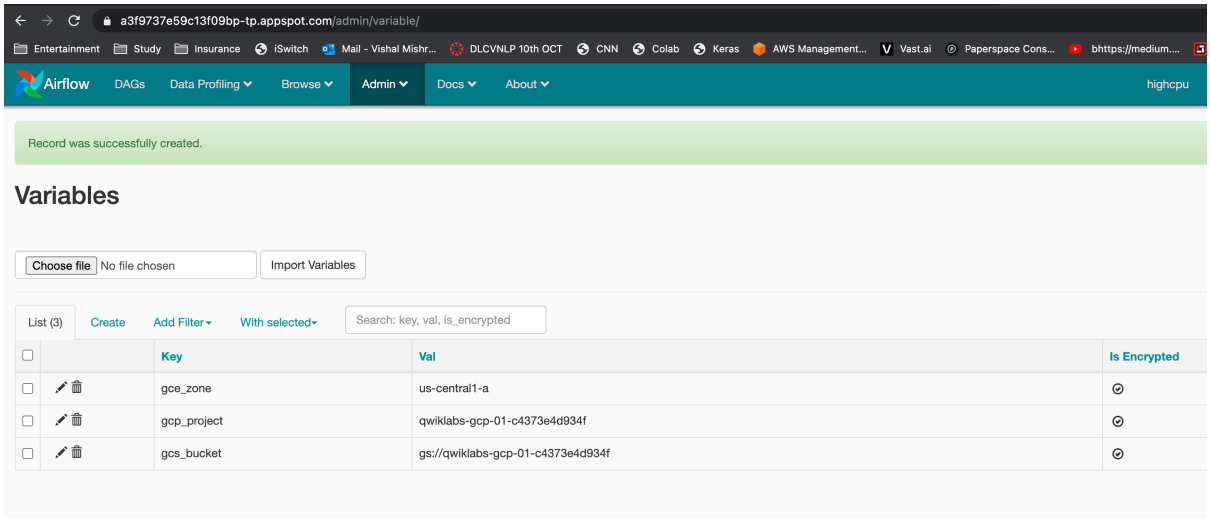


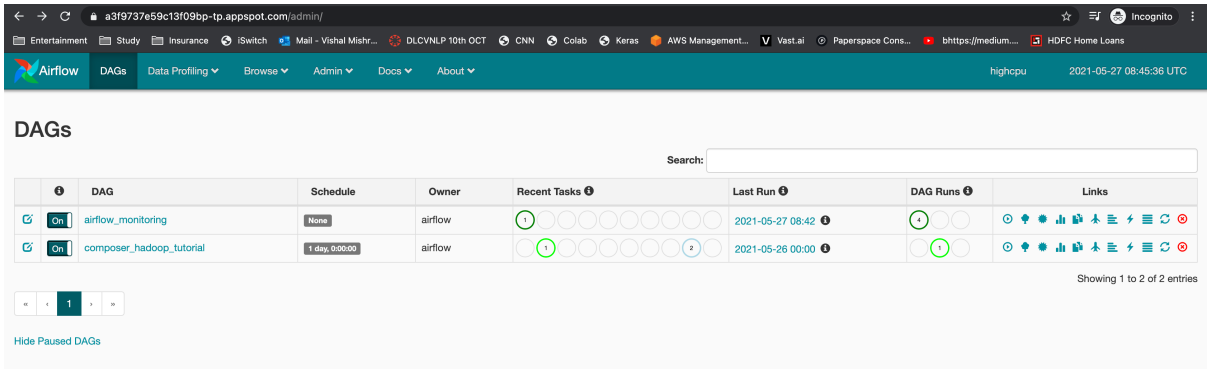
Cloud Composer –

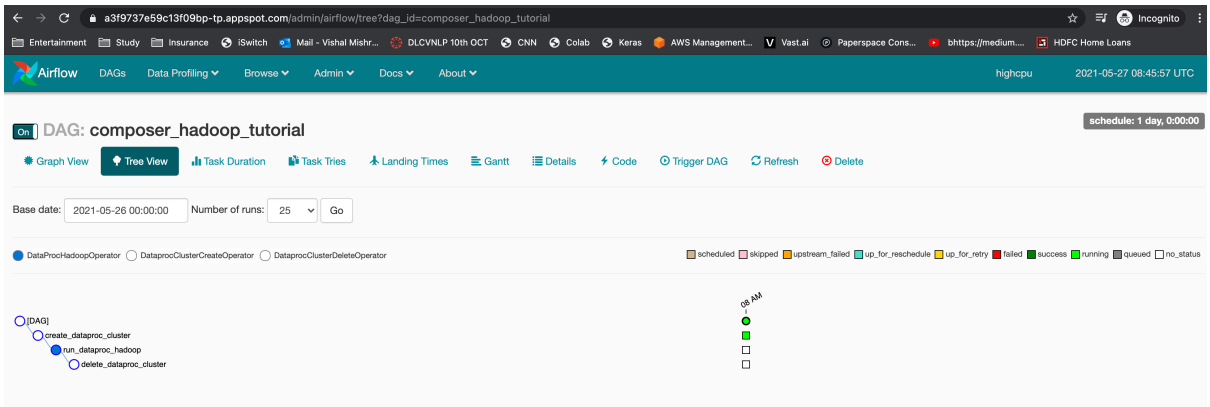
https://airflow.apache.org/docs/apache-airflow/stable/concepts/index.html

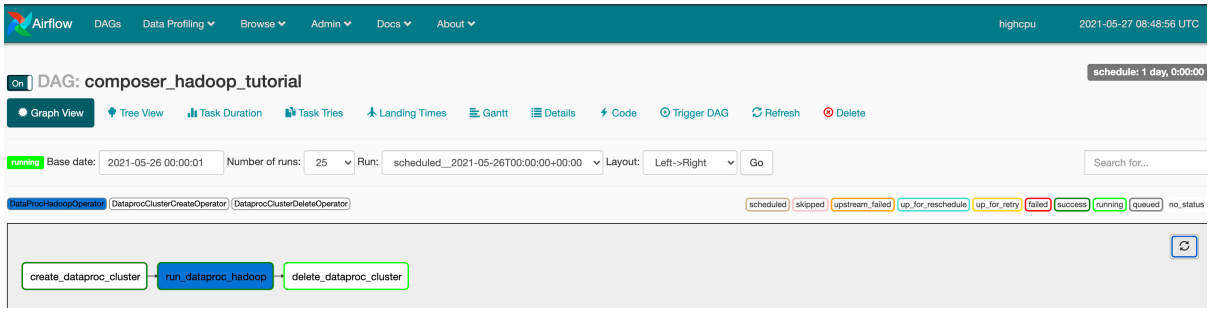
Defining the workflow

Now let's discuss the workflow you'll be using. Cloud Composer workflows are comprised of DAGs (Directed Acyclic Graphs). DAGs are defined in standard Python files that are placed in Airflow's `DAG_FOLDER`. Airflow will execute the code in each file to dynamically build the `DAG` objects. You can have as many DAGs as you want, each describing an arbitrary number of tasks. In general, each one should correspond to a single logical workflow.


a3f9737e59c13f09bp-tp.appspot.com/admin/variable/
Airflow
DAGs
Data Profiling
Browse
Admin
Docs
About
highcpu
Record was successfully created.
Variables
Choose file
No file chosen
Import Variables
List (3)
Create
Add Filter
With selected
Search: key, val, is_encrypted
Key
Val
Is Encrypted
gce_zone
us-central1-a
gcp_project
qwiklabs-gcp-01-c4373e4d934f
gcs_bucket
gs://qwiklabs-gcp-01-c4373e4d934f


a3f9737e59c13f09bp-tp.appspot.com/admin/
Airflow
DAGs
Data Profiling
Browse
Admin
Docs
About
highcpu
2021-05-27 08:45:36 UTC
DAGs
Search:
DAG
Schedule
Owner
Recent Tasks
Last Run
DAG Runs
Links
airflow_monitoring
None
airflow
2021-05-27 08:42
composer_hadoop_tutorial
1 day, 0:00:00
airflow
2021-05-26 00:00
Showing 1 to 2 of 2 entries
Hide Paused DAGs

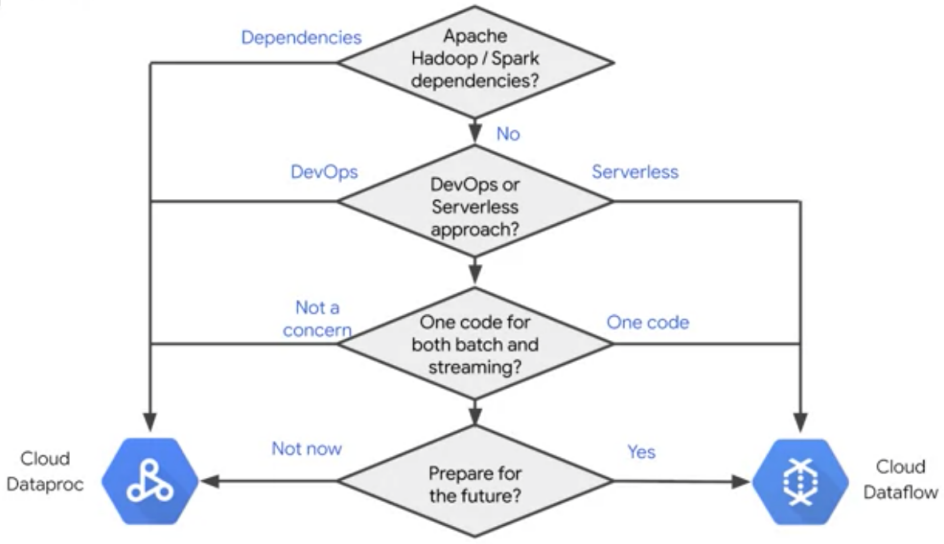

a3f9737e59c13f09bp-tp.appspot.com/admin/airflow/tree?dag_id=composer_hadoop_tutorial
Airflow
DAGs
Data Profiling
Browse
Admin
Docs
About
highcpu
2021-05-27 08:45:57 UTC
DAG: composer_hadoop_tutorial
schedule: 1 day, 0:00:00
Graph View
Tree View
Task Duration
Task Tries
Landing Times
Gantt
Details
Code
Trigger DAG
Refresh
Delete
Base date:
2021-05-26 00:00:00
Number of runs:
25
Go
DataProcHadoopOperator
DataprocClusterCreateOperator
DataprocClusterDeleteOperator
scheduled
skipped
upstream_failed
up_for_reschedule
up_for_retry
failed
success
running
queued
no_status
[DAG]
create_dataproc_cluster
run_dataproc_hadoop
delete_dataproc_cluster
08 AM


Airflow
DAGs
Data Profiling
Browse
Admin
Docs
About
highcpu
2021-05-27 08:48:56 UTC
DAG: composer_hadoop_tutorial
schedule: 1 day, 0:00:00
Graph View
Tree View
Task Duration
Task Tries
Landing Times
Gantt
Details
Code
Trigger DAG
Refresh
Delete
running
Base date:
2021-05-26 00:00:01
Number of runs:
25
Run:
scheduled__2021-05-26T00:00:00+00:00
Layout:
Left->Right
Go
Search for...
DataProcHadoopOperator
DataprocClusterCreateOperator
DataprocClusterDeleteOperator
scheduled
skipped
upstream_failed
up_for_reschedule
up_for_retry
failed
success
running
queued
no_status
create_dataproc_cluster
run_dataproc_hadoop
delete_dataproc_cluster

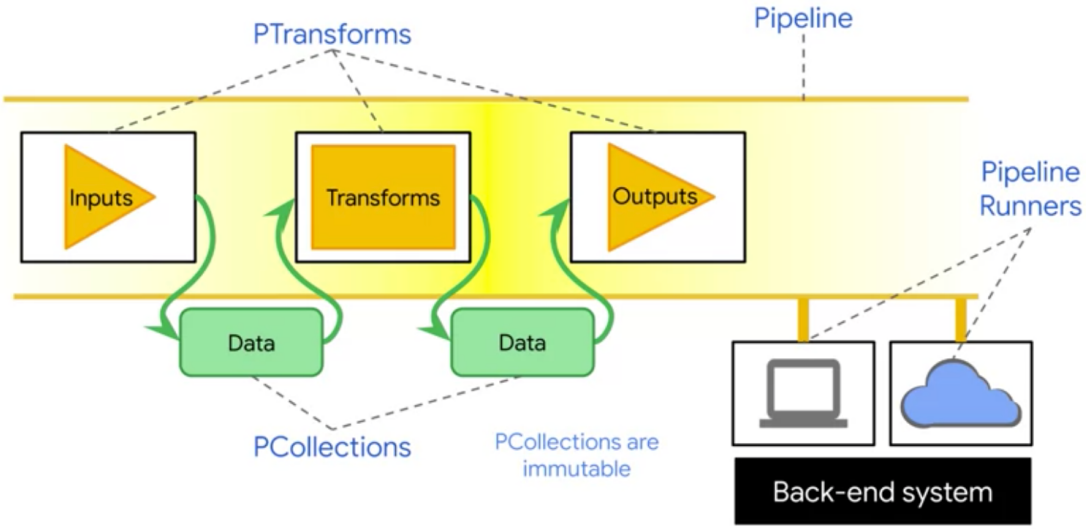
Google Cloud data pipeline processing options

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

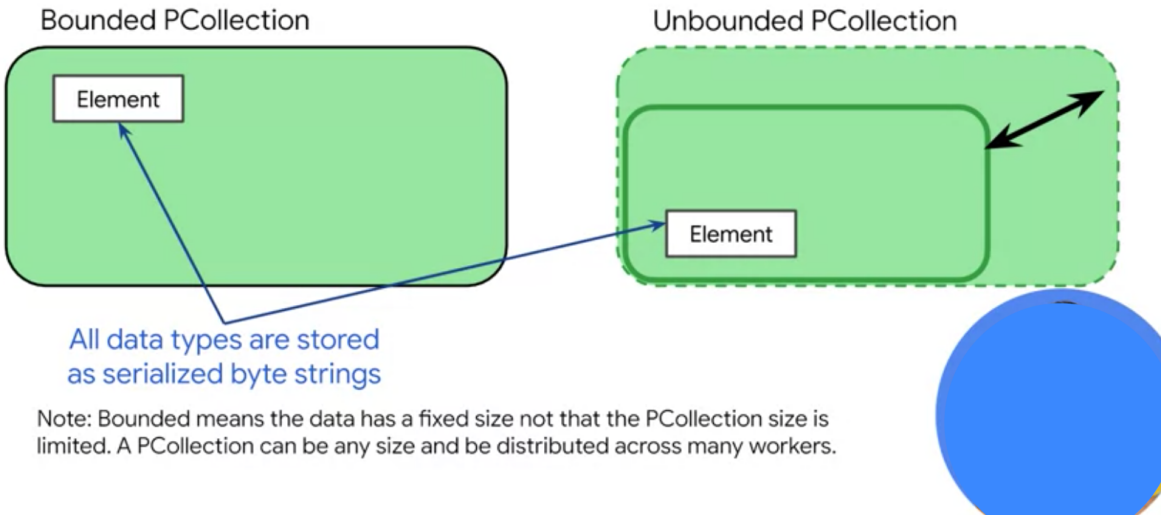
Choosing between Cloud Dataflow and Cloud Dataproc



Apache BEAM = Batch + strEAM

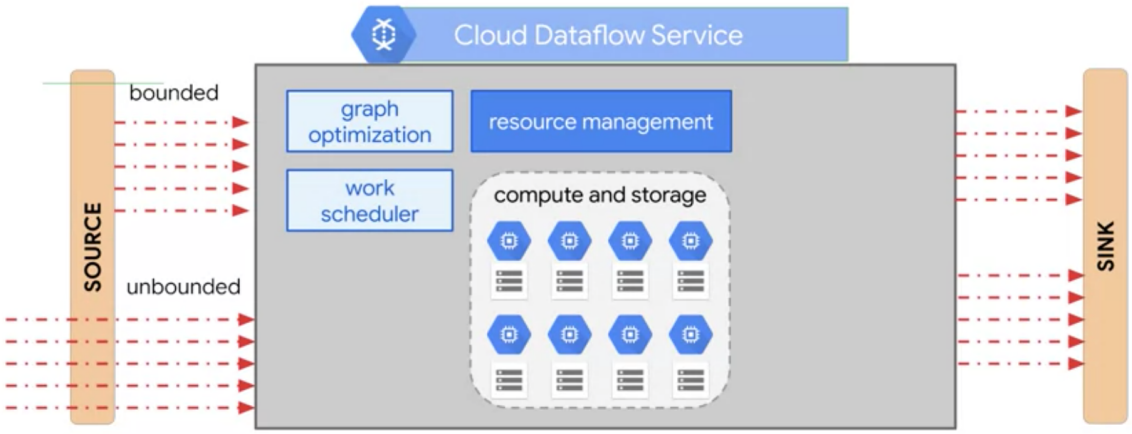


A PCollection represents batch or stream data



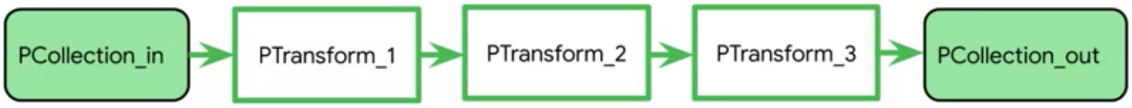
Note: Bounded means the data has a fixed size not that the PCollection size is limited. A PCollection can be any size and be distributed across many workers.

How does Cloud Dataflow work?



Cloud Dataflow constantly rebalances the work.

How to construct a simple pipeline




```
PCollection_out = (PCollection_in | PTransform_1
                  | PTransform_2
                  | PTransform_3 )
```

Python

Python overloads the pipe operator

Java

Java uses the .apply method

```
PCollection_out = PCollection_in.apply(PTransform_1)
.apply(PTransform_2)
.apply(PTransform_3)
```

How to construct a branching pipeline

Python

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

Java

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

A Pipeline is a directed graph of steps

Python

```
import apache_beam as beam

if __name__ == '__main__':

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

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

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

Create a pipeline parameterized by command line flags

Read input

Apply transform

Write output

Run a pipeline on Cloud Dataflow

Python

```
import apache_beam as beam

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

Where to run

This creates the pipeline

Pipeline Execution using DataflowRunner

Run local

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

Run on cloud

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

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

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

Read from Cloud Storage (returns a string)

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

Read from Cloud Pub/Sub (returns a string)

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

Read from BigQuery (returns rows)

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

Setup

Read

Write to a BigQuery table

Establish reference to BigQuery table

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

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

Write to BigQuery table

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

Create a PCollection in memory

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

This is the display name of the pipeline step

PCollection

Transforming data with PTransforms

Map and FlatMap

Use Map for 1:1 relationship between input and output

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

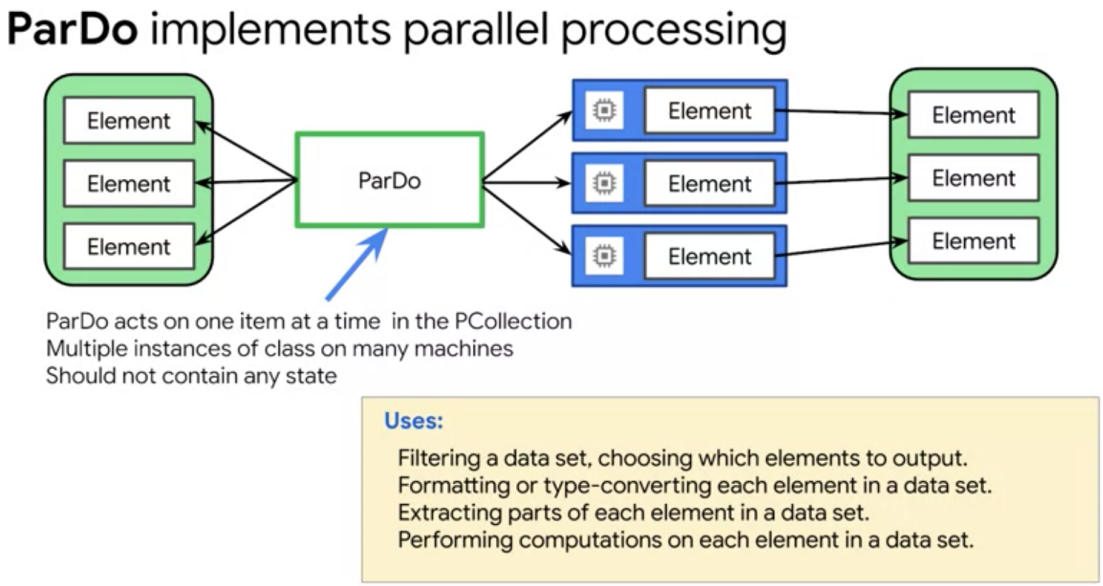
Map (fn) uses a callable fn to do a one-to-one transformation.

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

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

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

Generator

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

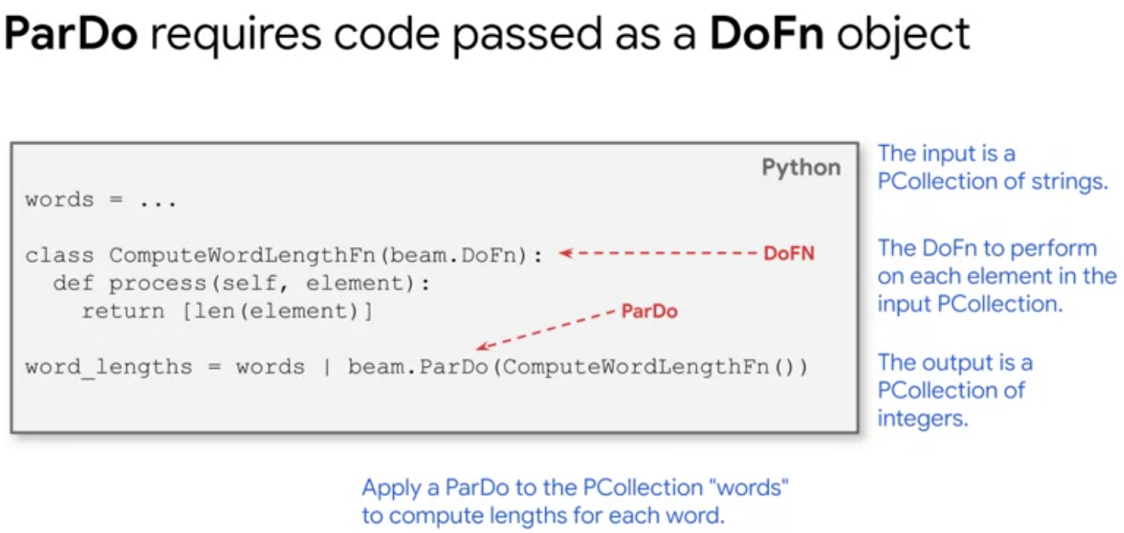


ParDo or parallel do is a common intermediate step in a pipeline.
You might use it to extract certain fields from a set of raw input records or convert raw input into a different format.

You might also use ParDo to convert process data into an output format like table rows for big query or strings for printing. You can use ParDo to filter a dataset. So consider each elements in a PCollection, and either output that elements to a new collection or discard it or. Or you can use it for formatting or type converting each elements in a dataset. If your input PCollection contains elements that are of a different type or format that you want. You can use pardho to perform a conversion of each elements and output the results to a new PCollection.

You can use it for extracting parts of each element in a dataset as well. If you have a PCollection of records with multiple fields, for example, you can use a ParDo to parse out just the fields you want to consider into a new PCollection. And you can use ParDo for performing computations on each elements in a dataset. You can use ParDo to perform simple or complex computations on every elements or certain elements of a PCollection and output the results as a new PCollection. When you apply a ParDo transform, you need to provide user code in the form of a DoFn object. Where do function is a beam SDK class that defines a distributed processing function. Your do function code must be fully serializable, item potent and thread safe.

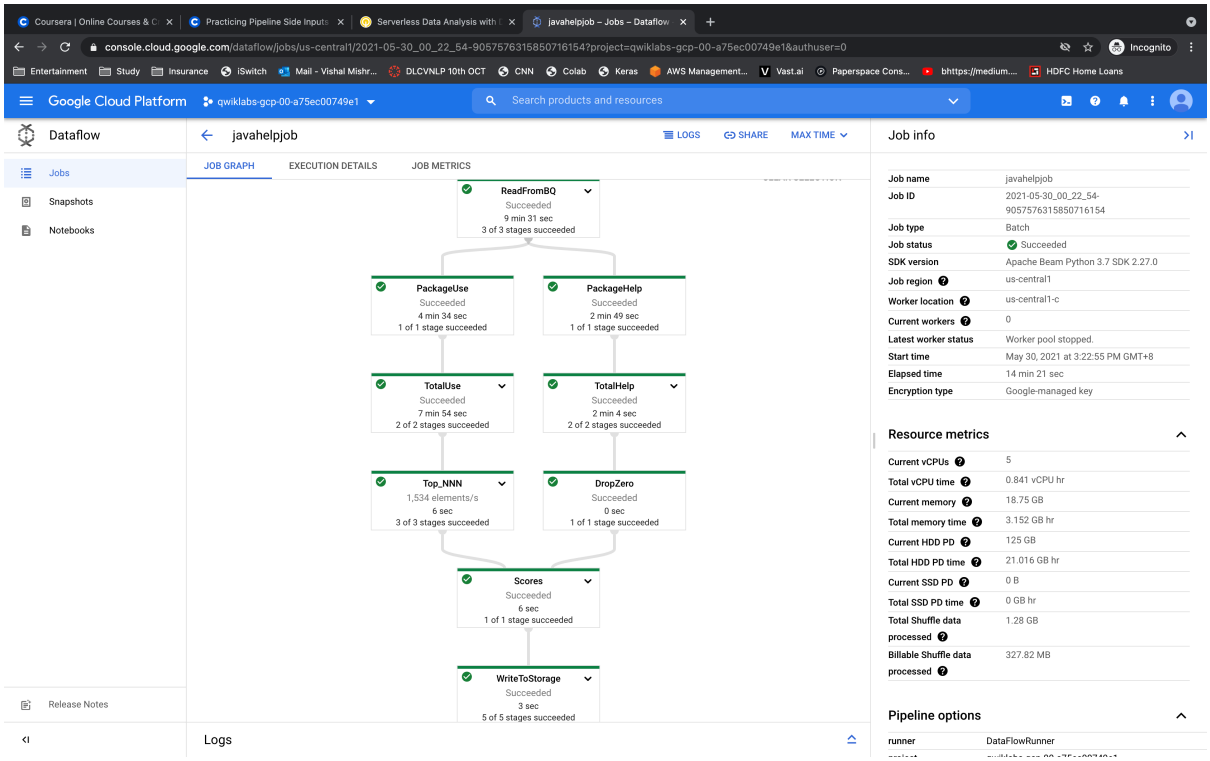
In this example, we're just counting the number of words in a line, and returning the length of the line. Transformations are always going to work on one element at a time here. Here we have an example from Python which can return multiple variables.



DataFlow Hands on =>
==========

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

```
cd ~/training-data-analyst/courses/data_analysis/lab2/python
```


Coursera | Online Courses & C
Practicing Pipeline Side Inputs
Serverless Data Analysis with
javahelpjob – Jobs – Dataflow
console.cloud.google.com/dataflow/jobs/us-central1/2021-05-30_00_22_54-9057576315850716154?project=qwiklabs-gcp-00-a75ec00749e1&authuser=0
Incognito
Entertainment
Study
Insurance
iSwitch
Mail - Vishal Mishr...
DLCVNLP 10th OCT
CNN
Colab
Keras
AWS Management...
Vast.ai
Paperspace Cons...
bhttps://medium....
HDFC Home Loans
Google Cloud Platform
qwiklabs-gcp-00-a75ec00749e1
Search products and resources
Dataflow
Jobs
Snapshots
Notebooks
Release Notes
javahelpjob
LOGS
SHARE
MAX TIME
JOB GRAPH
EXECUTION DETAILS
JOB METRICS
ReadFromBQ
Succeeded
9 min 31 sec
3 of 3 stages succeeded
PackageUse
Succeeded
4 min 34 sec
1 of 1 stage succeeded
PackageHelp
Succeeded
2 min 49 sec
1 of 1 stage succeeded
TotalUse
Succeeded
7 min 54 sec
2 of 2 stages succeeded
TotalHelp
Succeeded
2 min 4 sec
2 of 2 stages succeeded
Top_NNN
1,534 elements/s
6 sec
3 of 3 stages succeeded
DropZero
Succeeded
0 sec
1 of 1 stage succeeded
Scores
Succeeded
6 sec
1 of 1 stage succeeded
WriteToStorage
Succeeded
3 sec
5 of 5 stages succeeded
Logs
Job info
Job name
javahelpjob
Job ID
2021-05-30_00_22_54-9057576315850716154
Job type
Batch
Job status
Succeeded
SDK version
Apache Beam Python 3.7 SDK 2.27.0
Job region
us-central1
Worker location
us-central1-c
Current workers
0
Latest worker status
Worker pool stopped.
Start time
May 30, 2021 at 3:22:55 PM GMT+8
Elapsed time
14 min 21 sec
Encryption type
Google-managed key
Resource metrics
Current vCPUs
5
Total vCPU time
0.841 vCPU hr
Current memory
18.75 GB
Total memory time
3.152 GB hr
Current HDD PD
125 GB
Total HDD PD time
21.016 GB hr
Current SSD PD
0 B
Total SSD PD time
0 GB hr
Total Shuffle data processed
1.28 GB
Billable Shuffle data processed
327.82 MB
Pipeline options
runner
DataFlowRunner