Data pipelines for PySpark

Preprocessing data for data modeling

Data pipeline summary

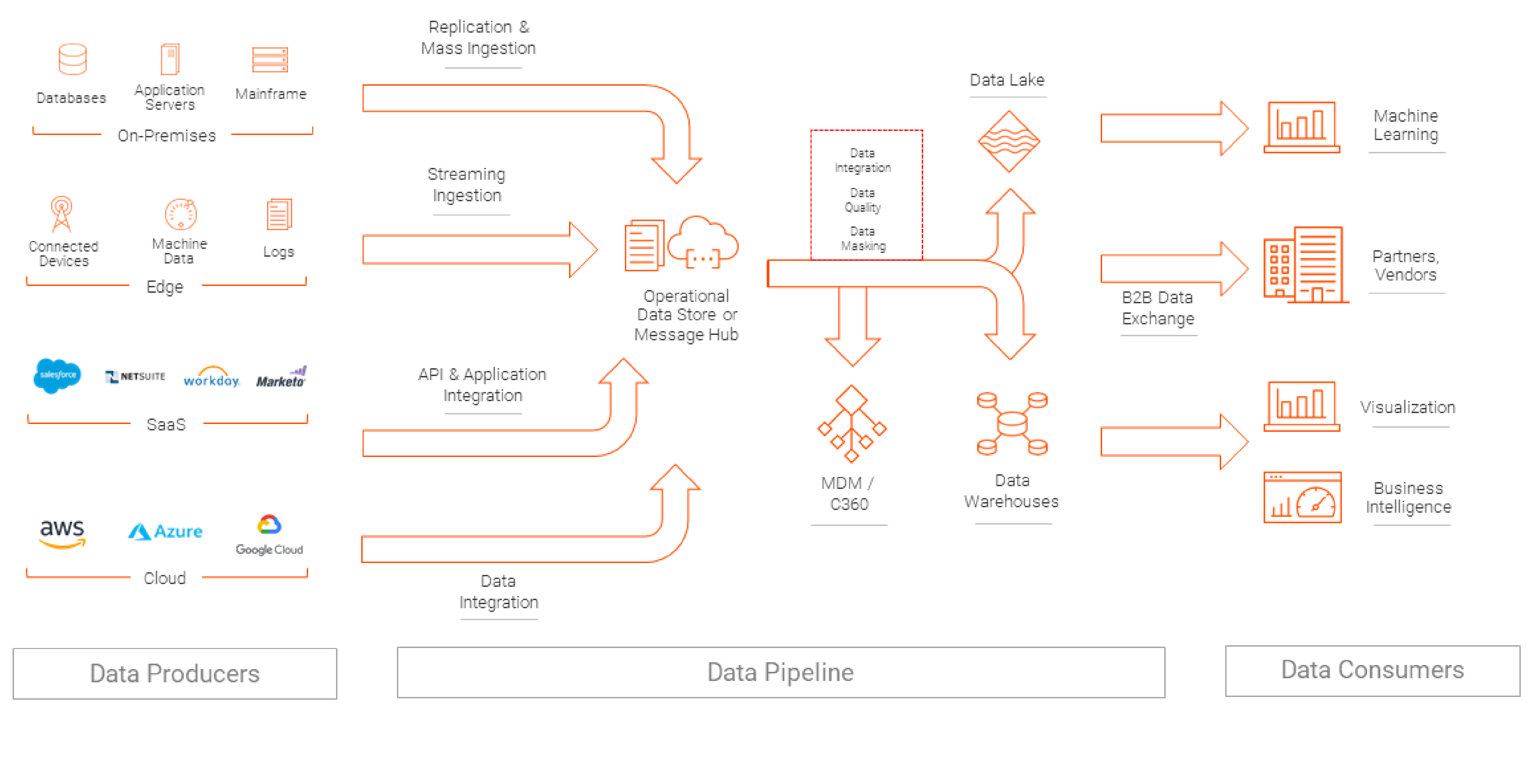
- A data pipeline processes elements connected in series, where the output of one element is the input of the next one
- Common to
  - Ingest data using distributed data logging programs (Kafka)
  - Process data using PySpark
    - Data cleaning and eventually ML pipeline
  - Manage the ingest to modeling via a workflow framework
    - E.g., Apache Airflow
- Notebook workflow is a common for PySpark pipelines

Data pipeline

- is a set of data processing elements connected in series, where the output of one element is the input of the next one.
  - The elements of a pipeline are often executed in parallel or in time-sliced fashion
- Examples
  - Machine learning pipeline
    - Data is prepared for input into ML library functions
  - Data pipeline
    - Data is acquired and processed in preparation for input into another process, such as an ML pipeline

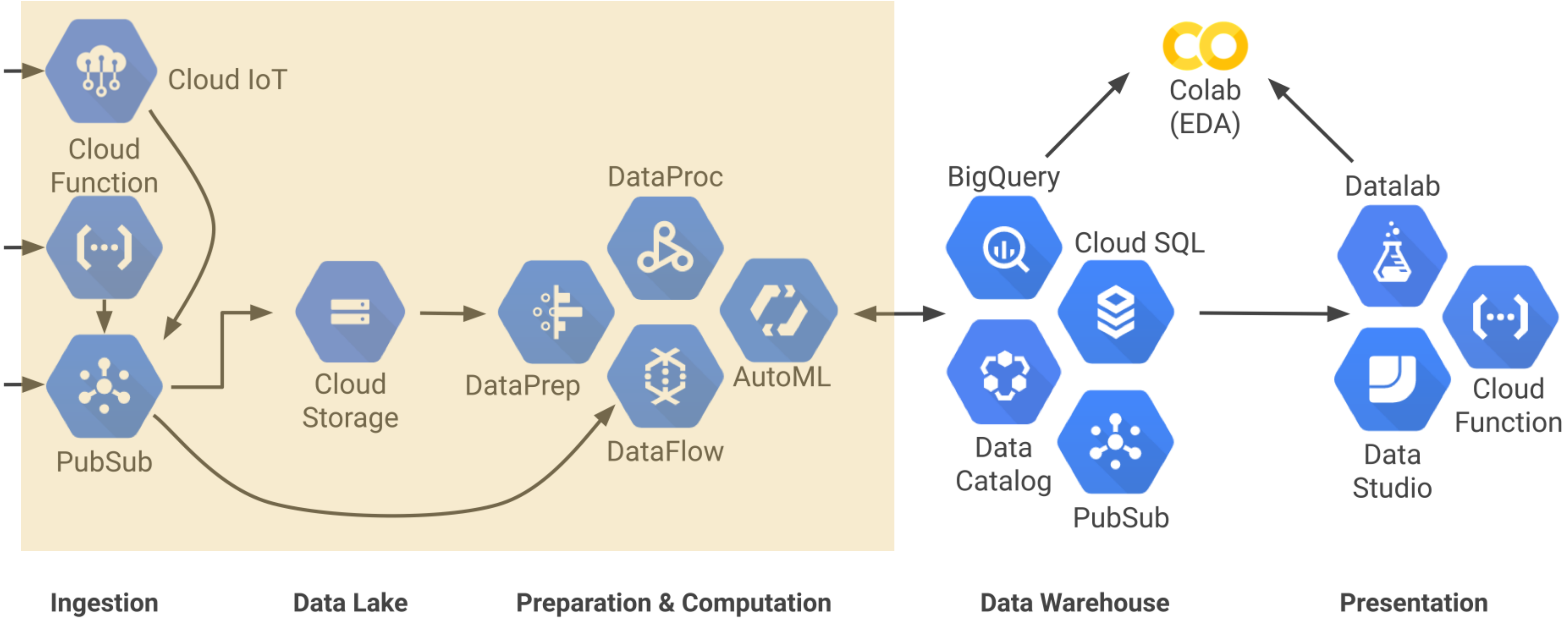
https://en.wikipedia.org/wiki/Pipeline_(computing)

Data pipeline patterns



https://blogs.informatica.com/2019/08/20/data-processing-pipeline-patterns/

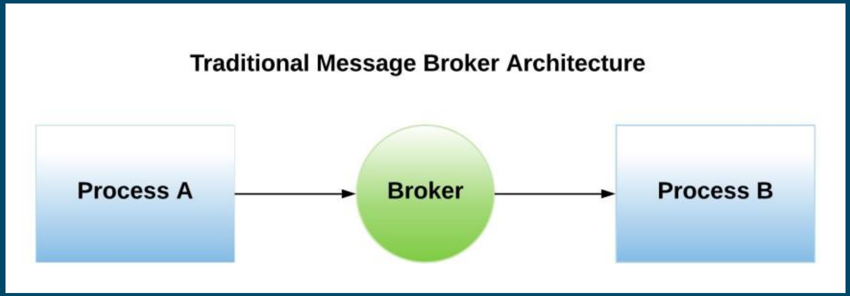
Data pipeline in Google Cloud Platform

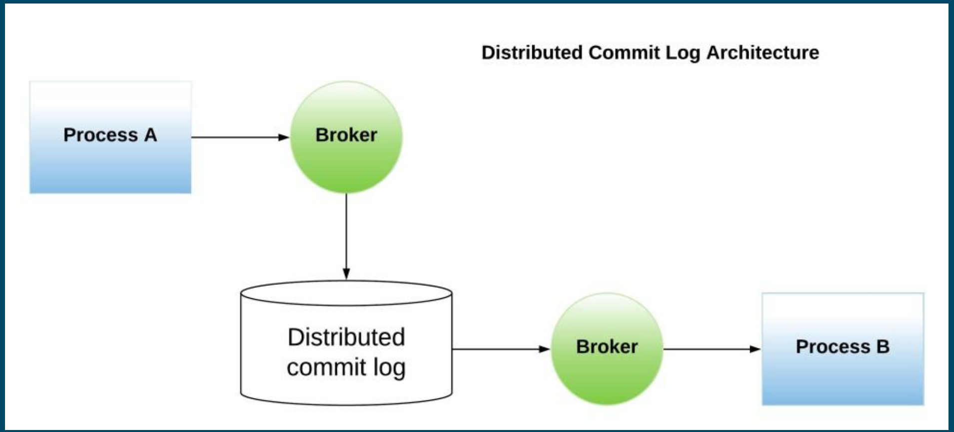


https://towardsdatascience.com/scalable-efficient-big-data-analytics-machine-learning-pipeline-architecture-on-cloud-4d59efc092b5

Step 1: Ingest the data

- Traditional message broker systems (JMS) connect direct to brokers, and brokers which connect direct to processes.
  - These solutions tend are optimized towards flexibility and configurable delivery guarantees.
- Distributed commit log technologies are similar role to the traditional broker message systems.
  - They are optimized towards concentrated on scalability and throughput, which lesser guarantees on event ordering or arrival
  - Apache Kafka
  - Amazon Kinesis
  - Microsoft Azure Event Hubs
  - Google Pub/Sub





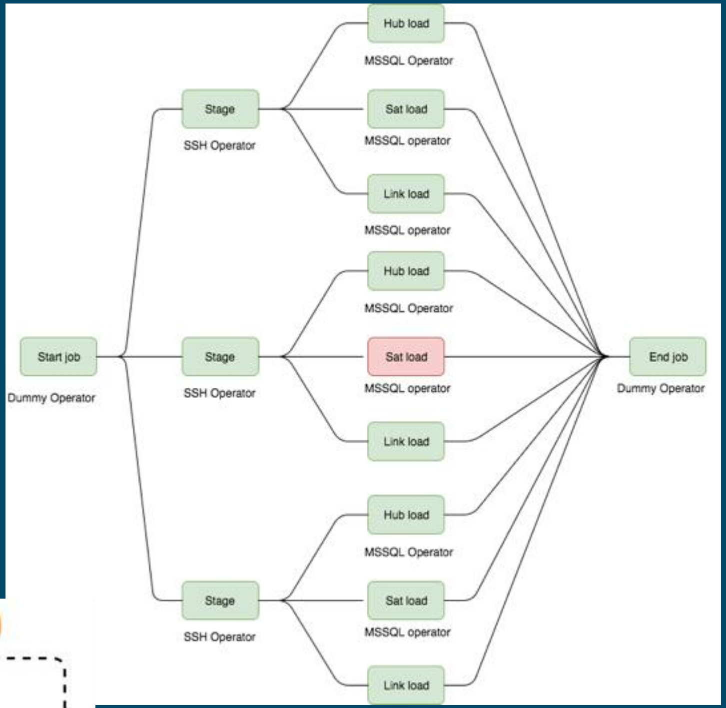
https://blog.scottlogic.com/2018/04/17/comparing-big-data-messaging.html

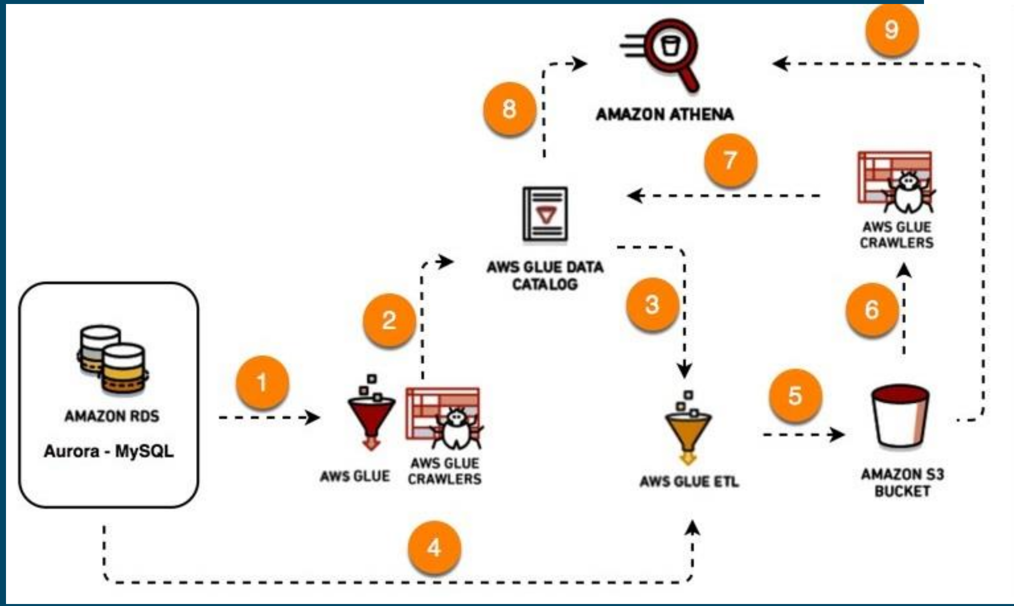
Step 2: process the data

- Transformation using
  - Python
    - Not for big data (parallelism)
  - PySpark
  - Spark notebook workflow
  - Apache Beam
  - Google Dataflow
  - AWS Data Pipeline

Step 3: schedule the ingest, data process, etc.

- Apache airflow
  - Open source, run anywhere, task scheduler
- AWS Glue
  - AWS serverless task scheduler



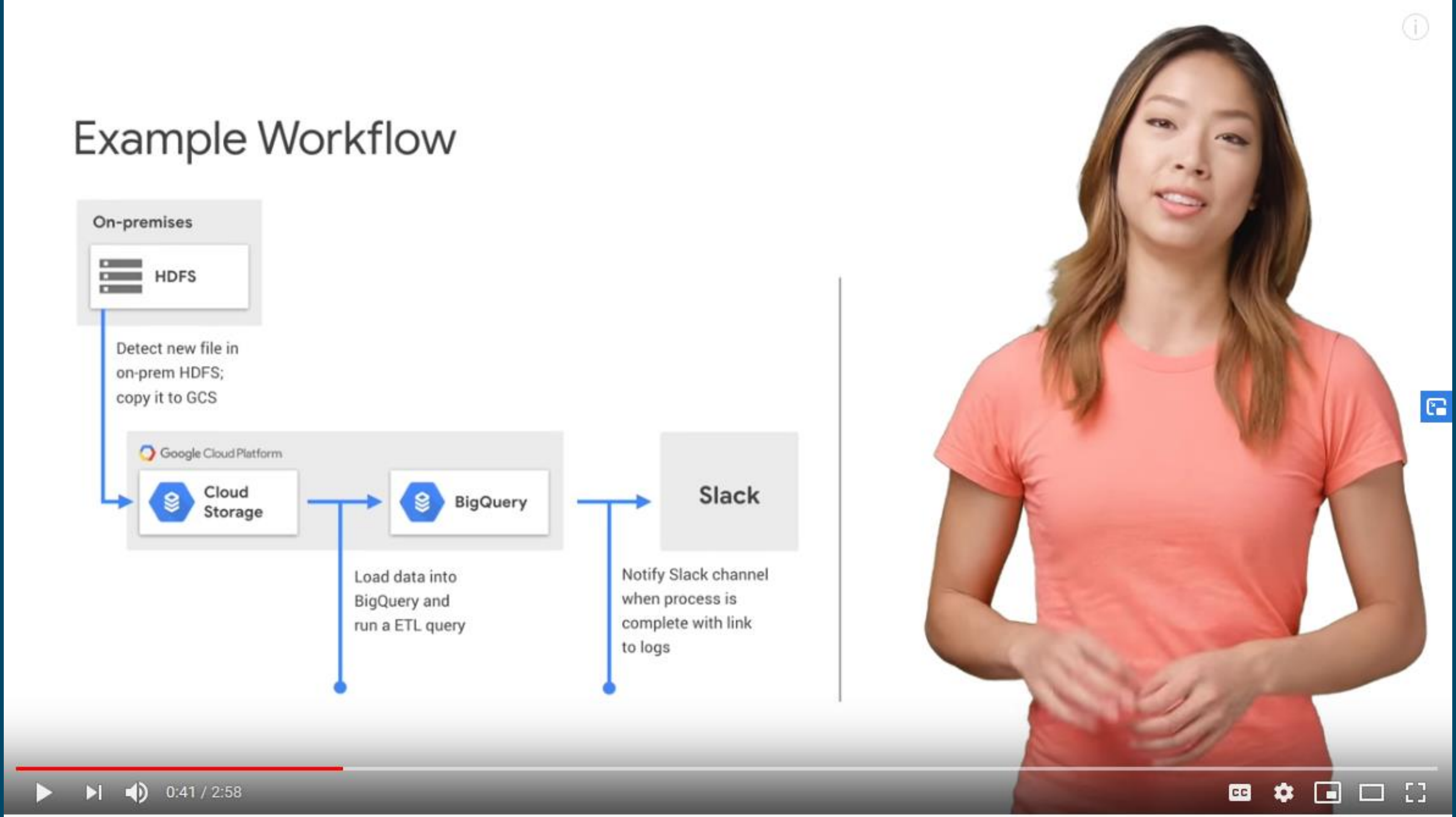


NEXT '18

REWIND

Flexible, Easy Data Pipelines on Google Cloud with Cloud Composer

Illustrative summary with GCP Cloud Composer



https://www.youtube.com/watch?v=Lh51TET7u5Y

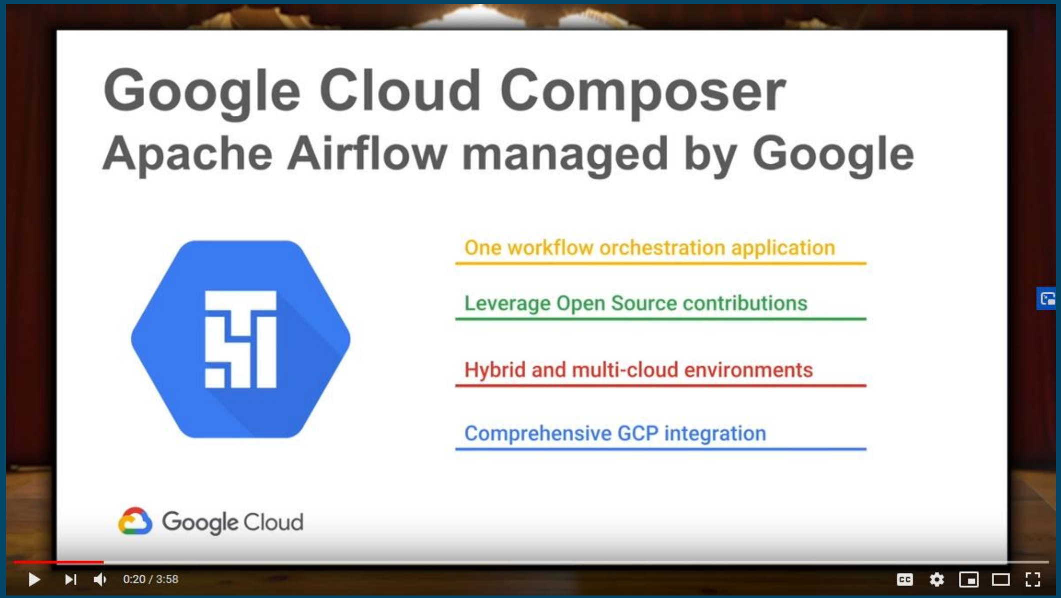
TAKE5

Cloud Composer

Google Cloud Platform

Example of running Cloud Composer

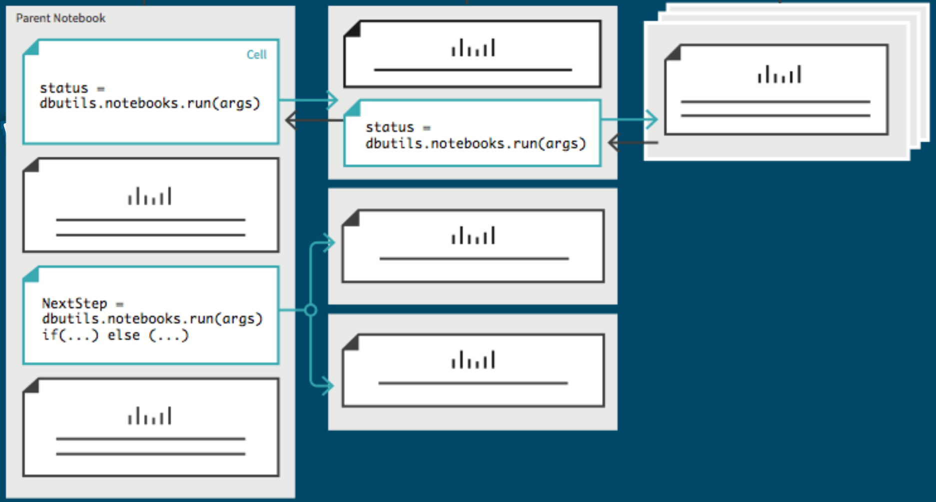
- In the example, tasks are defined as Python statement
- Tasks can also be calls to run Jupyter notebooks



https://www.youtube.com/watch?v=bwZOAXnCMf8

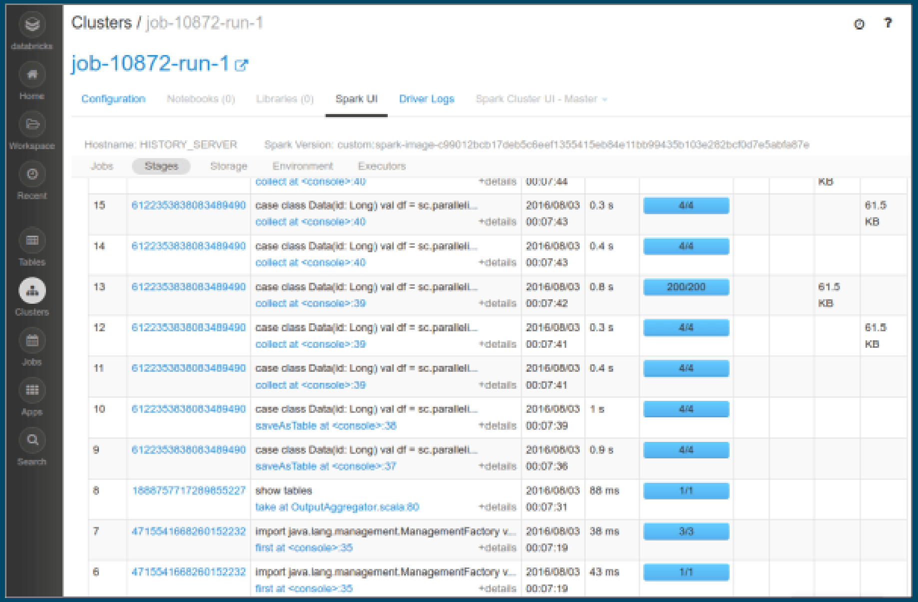
Notebook workflow

Databricks Notebook Workflow



- Each notebook becomes a job to be scheduled
    - Notebooks typically do data ingest, processing, modeling, etc.
- One notebook starts the process
    - Write the workflow in PySpark
    - Can also a workflow product (Airflow, Glue)

```
dbutils.notebook.run(
  "../path/to/my/notebook",
  timeout_seconds = 60,
  arguments = {"x": "value1", "y": "value2", ...})
```



https://docs.databricks.com/notebooks/notebook-workflows.html

Google Notebook Workflows

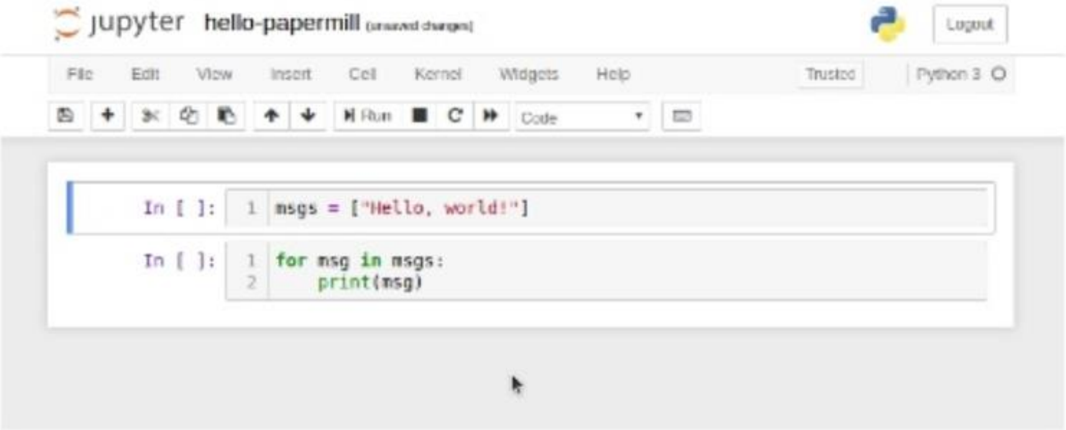
Automation

Papermill: parametrized notebooks

papermill

https://github.com/nteract/papermill

1. Declare some cells as a parameters cells.



2. Declare parameters values in YAML file.

```
x:
    - 0.0
    - 1.0
    - 2.0
    - 3.0
linear_function:
    slope: 3.0
    intercept: 1.0
```

3. Run papermill with Python API.

```
import papermill as pm

pm.execute_notebook(
   'path/to/input.ipynb',
   'path/to/output.ipynb',
   parameters = dict(alpha=0.6, ratio=0.1)
)
```

3. Run papermill with CLI.

```
$ papermill local/input.ipynb s3://bkt/output.ipynb -f parameters.yaml
```

Notebook workflow pipeline illustrated

Call notebook

```
dbutils.notebook.run(
  "../path/to/my/notebook",
  timeout_seconds = 60,
  arguments = {"x": "value1", "y": "value2", ...})
```

Notebook parameters

1. WorldData.Pipeline (Python)

s6.4 File View: Code Permissions Run All Clear Publish

Cmd 4

Notebook parameters, which can be overriden by notebook workflow

```
params = {
  "file_prefix" : "WPP2019",
  "files" : ["_TotalPopulationBySex.csv", "_Period_Indicators_Medium.csv"],
  "CSV_directory": "https://population.un.org/wpp/Download/Files/1_Indicators%20(Standard)/CSV_FILES"
}
```

Cmd 6

Download files

```
import requests
for fname in params["files"]:
  name = params["file_prefix"] + fname
  url = "{}/{}".format(params["CSV_directory"], name)
  print("Downloading file: ", url)
  r = requests.get(url)
  open(name , 'wb').write(r.content)
  print("Saved file", name)
```

```
Downloading file:  https://population.un.org/wpp/Download/File
```

Data pipeline summary

- A data pipeline processes elements connected in series, where the output of one element is the input of the next one
- Common to
  - Ingest data using distributed data logging programs (Kafka)
  - Process data using PySpark
    - Data cleaning and eventually ML pipeline
  - Manage the ingest to modeling via a workflow framework
    - E.g., Apache Airflow
- Notebook workflow is a common for PySpark pipelines

Important to remember

Simplifying PySpark data transformations

Consider these kinds of DataFrame transformations

- Dropping columns
- Changing column types
- Creating a new column
  - By combining columns
  - Running a UDF

Drop duplicate columns and rename Time

```
dfInd = indMed.drop('Time').drop('LocID').drop('VarID').drop('Variant').withColumnRenamed('MidPeriod','Time'
```

Convert types of columns

```
# Just an illustation here because types are OK
dfInd = dfInd.withColumn('Births', dfInd['Births'].cast('int'))
```

Instead of in-line code, apply transformation functions

- The result looks like this

Cmd 10

```
dfInd = indMed.transform(ind_drop_cols())\
              .transform(cast_to_type('Births','int'))
bySex = bySex.transform(cast_to_type('Time','int'))
```

- Define the transform function, on DataFrame
  - It applies your given function to the DataFrame and returns the update data frame
    - df = function(df)

Monkey patch

- Allows the definition of new functions on existing classes (in Python)

```python
from pyspark.sql.dataframe import DataFrame

######################################################################
# DataFrame transformations monkey patches
# See https://medium.com/@mrpowers/chaining-custom-pyspark-transformations-4f38a8c7ae55
######################################################################

def transform(self, f):
    return f(self)
DataFrame.transform = transform
```

Two example functions to apply to a DataFrame

- Function specific to this problem
  - Dropping specific columns
- Generic function
  - Can be reused in any data pipeline

```python
# Used to simplify the code
def ind_drop_cols():
  def inner(df):
    df = df.drop('Time').drop('LocID').drop('VarID').drop('Variant')\
              .withColumnRenamed('MidPeriod','Time').where("Time < 2019 and Variant = 'Medium'")
    return df
  return inner


def cast_to_type(col,type):
  def inner(df):
    return df.withColumn(col,df[col].cast(type))
  return inner
```

Before and after using transform

```
dfInd = indMed.drop('Time').drop('LocID').drop('VarID').drop('Variant').withColumnRenamed('MidPeriod','Time').where("Time < 2019
and Variant = 'Medium'")
```

```
# Just an illustation here because types are OK
dfInd = dfInd.withColumn('Births', dfInd['Births'].cast('int'))
bySex = bySex.withColumn('Time', bySex['Time'].cast('int'))
```

```
# Used to simplify the code
def ind_drop_cols():
  def inner(df):
    df = df.drop('Time').drop('LocID').drop('VarID').drop('Variant')\
              .withColumnRenamed('MidPeriod','Time').where("Time < 2019 and Variant = 'Medium'")
    return df
  return inner


def cast_to_type(col,type):
  def inner(df):
    return df.withColumn(col,df[col].cast(type))
  return inner
```

```
dfInd = indMed.transform(ind_drop_cols())\
           .transform(cast_to_type('Births','int'))
bySex = bySex.transform(cast_to_type('Time','int'))
```

A function that returns a function

- Input
  - Two parameters
- Process
  - Defines a new function (inner)
  - Casts the given column to the given type
  - Return that function
- How to call it

```
def cast_to_type(col,type):
  def inner(df):
    return df.withColumn(col,df[col].cast(type))
  return inner
```

```
dfInd = indMed.transform(ind_drop_cols())\
              .transform(cast_to_type('Births','int'))
```

Monkey patch for
DataFrame transform
can simplify (read, write, maintain) PySpark
data pipelines