1.Airflow

* Apache airflow is an open source platform to programmatically author, schedule and monitor workflows/ data pipelines.
* Airflow is an orchestrator. It allows to execute your tasks in the right way at the right time
* Benefits:
  + Dynamic: uses python which is good
  + Scalable: run as many tasks as you want
  + UI: provides a very nice UI to manage tasks
  + Extensible
* Airflow is an orchestrator, not a processing framework. Process your gigabytes of data outside of Airflow (i.e. You have a Spark cluster, you use an operator to execute a Spark job, and the data is processed in Spark).
* A DAG is a data pipeline, an Operator is a task.
* An Executor defines how your tasks are executed, whereas a worker is a process executing your task
* The Scheduler schedules your tasks, the web server serves the UI, and the database stores the metadata of Airflow.

2.Components

**DAG**: Directed Acyclic Graph

* a DAG is run at start\_date/last\_run + scheduled\_interval

**Operator**: one task in your data pipeline

3 types of operator:

1. Action operators - execute an action
2. Transfer operators - transfer data
3. sensors - wait or a condition to be met

**Providers**: They are not part of the default apache-airflow package

To install AWS, Dbt, Snowflake:

* pip install apache-airflow-providers-amazon
* pip install apache-airflow-providers-snowflake
* pip install apache-airflow-providers-dbt-cloud

**Sensors**: They wait for something to happen before executing the next task.

* ex. they poke a specific API until it gets some response before moving on to the next task

**Hook**: a hook allows you to easily interact with an external tool or an external service

**Catch-up**: this mechanism allows you to automatically run non trigger DAG runs between the last time your DAG was triggered and the date of now.

**Backfilling**: This mechanism allows you to run historical DAG runs. Ex. you want to run DAG prior to the start\_date, you can do that using *airflow dag backfill*

Here is the final code 👇

1. from airflow import DAG
2. from airflow.providers.postgres.operators.postgres import PostgresOperator
3. from airflow.providers.http.sensors.http import HttpSensor
4. from airflow.providers.http.operators.http import SimpleHttpOperator
5. from airflow.operators.python import PythonOperator
6. from airflow.providers.postgres.hooks.postgres import PostgresHook
8. import json
9. from pandas import json\_normalize
10. from datetime import datetime
12. def \_process\_user(ti):
13. user = ti.xcom\_pull(task\_ids="extract\_user")
14. user = user['results'][0]
15. processed\_user = json\_normalize({
16. 'firstname': user['name']['first'],
17. 'lastname': user['name']['last'],
18. 'country': user['location']['country'],
19. 'username': user['login']['username'],
20. 'password': user['login']['password'],
21. 'email': user['email'] })
22. processed\_user.to\_csv('/tmp/processed\_user.csv', index=None, header=False)
24. def \_store\_user():
25. hook = PostgresHook(postgres\_conn\_id='postgres')
26. hook.copy\_expert(
27. sql="COPY users FROM stdin WITH DELIMITER as ','",
28. filename='/tmp/processed\_user.csv'
29. )
31. with DAG('user\_processing', start\_date=datetime(2022, 1, 1),
32. schedule\_interval='@daily', catchup=False) as dag:
34. create\_table = PostgresOperator(
35. task\_id='create\_table',
36. postgres\_conn\_id='postgres',
37. sql='''
38. CREATE TABLE IF NOT EXISTS users (
39. firstname TEXT NOT NULL,
40. lastname TEXT NOT NULL,
41. country TEXT NOT NULL,
42. username TEXT NOT NULL,
43. password TEXT NOT NULL,
44. email TEXT NOT NULL
45. );
46. '''
47. )
49. is\_api\_available = HttpSensor(
50. task\_id='is\_api\_available',
51. http\_conn\_id='user\_api',
52. endpoint='api/'
53. )
55. extract\_user = SimpleHttpOperator(
56. task\_id='extract\_user',
57. http\_conn\_id='user\_api',
58. endpoint='api/',
59. method='GET',
60. response\_filter=lambda response: json.loads(response.text),
61. log\_response=True
62. )
64. process\_user = PythonOperator(
65. task\_id='process\_user',
66. python\_callable=\_process\_user
67. )
69. store\_user = PythonOperator(
70. task\_id='store\_user',
71. python\_callable=\_store\_user
72. )
74. create\_table >> is\_api\_available >> extract\_user >> process\_user >> store\_user

3.New way of Scheduling DAGs(v2.4+)

**Dataset:**it is just a group of data. Think of it as a file, as a SQL table, as anything that has data and you must define a URI

My\_file = Dataset(

"s3://dataset/file.csv”,

extra={‘owner’ : ‘james’},

)

*Note: canot pass the uri with airflow scheme ex: airflow://file.csv*

*Note: after 2.4 version, schedule\_interval is getting deprecated.*

Code:

**producer -> dataset -> consumer**

***consumer.py***

my\_file = Dataset(“/tmp/my\_file.txt”)

my\_file\_2 = Dataset(“/tmp/my\_file\_2.txt")

with DAG(

dag\_id=“consumer”,

schedule=[my\_file, my\_file\_2],

start\_date=datetime(2022, 1, 1),

catchup=False

):

@task

def read\_dataset():

with open(my\_file.uri, “r”) as f:

print(f.read())

@task

def read\_dataset\_2():

with open(my\_file\_2.uri, “r”) as f:

print(f.read())

read\_dataset() >> read\_dataset\_2()

***producer.py***

my\_file = Dataset(“/tmp/my\_file.txt”)

my\_file\_2 = Dataset(“/tmp/my\_file\_2.txt")

with DAG(

dag\_id=“producer”,

schedule=“@daily",

start\_date=datetime(2022, 1, 1),

catchup=False

):

@task(outlets=[my\_file])

def update\_dataset():

with open(my\_file.uri, “a+”) as f:

f.write(“producer update")

@task(outlets=[my\_file\_2])

def update\_dataset\_2():

with open(my\_file\_2.uri, “a+”) as f:

f.write(“producer update")

update\_dataset() >> update\_dataset\_2()

**Dataset limitations**

Datasets are amazing, but they have limitations as well:

* DAGs can only use Datasets in the same Airflow instance. A DAG cannot wait for a Dataset defined in another Airflow instance.
* Consumer DAGs are triggered every time a task that updates datasets completes successfully. **Airflow doesn't check whether the data has been effectively updated.**
* You can't combine different schedules like datasets with cron expressions.
* If two tasks update the same dataset, as soon as one is done, that triggers the Consumer DAG immediately without waiting for the second task to complete.
* Airflow monitors datasets only within the context of DAGs and Tasks. If an external tool updates the actual data represented by a Dataset, Airflow has no way of knowing that.

4.Database and Executors

**Executor:**It defines how to run your task, on which system. It does not execute the task.

Types of executors:

* **Sequential Executor:**It is the executor by default when you install airflow manually. It only runs 1 task at a time.
  + it uses SQLite
* **Local Executor:**It allows you to execute multiple tasks at he same time but on a single machine.
  + It uses PostgreSQL, mySQL, OracleDB
* **Celery Executor:**It allows to run multiple tasks at the same time. Also by using Celery cluster, we can execute tasks on multiple machines. It as airflow workers which run the tasks.
  + We need to install Celery queue which may be Redis

*from airflow import DAG*

*from airflow.operators.bash import BashOperator*

*from datetime import datetime*

*with DAG(‘parallel\_dag’, start\_date=datetime(2022, 1, 1),*

*schedule\_interval=‘@daily’, catchup=False) as dag:*

*extract\_a = BashOperator(*

*task\_id=‘extract\_a’,*

*bash\_command=‘sleep 1’*

*)*

*extract\_b = BashOperator(*

*task\_id=‘extract\_b’,*

*bash\_command=‘sleep 1’*

*)*

*load\_a = BashOperator(*

*task\_id=‘load\_a’,*

*bash\_command=‘sleep 1’*

*)*

*load\_b = BashOperator(*

*task\_id=‘load\_b’,*

*bash\_command=‘sleep 1’*

*)*

*transform = BashOperator(*

*task\_id=‘transform’,*

*bash\_command=‘sleep 1’*

*)*

*extract\_a >> load\_a*

*extract\_b >> load\_b*

*[load\_a, load\_b] >> transform*

5.Flower and Queue

**Flower:**It is a web based tool allowing you to monitor and administrate celery cluster.

It has a separate Ui from which you can see the number of tasks that are active, processed, failed, succeeded.

Remove DAG examples

To keep our Airflow instance nice and clean, we are going to remove the DAG examples from the UI:

* Replace the value 'true' by 'false' for the AIRFLOW\_\_CORE\_\_LOAD\_EXAMPLES environment variables

**Queue:** It is a queue of tasks. With Celery you have multiple workers, multiple machines.

With queues, you can define on which machine, on which worker you want to execute tasks according to their specificities.

* So you see with queues you are able to distribute your tasks among multiple machines according to the specificities of your tasks and your machines.

To create a new queue make this change in config:

*command: celery worker -q high\_cpu*

To send a task to a particular queue specific queue in the task*:*

*transform = BashOperato(*

*task\_id=’transform’,*

*queue=‘high\_cpu’,*

*bash\_command=’sleep 10’*

*)*

Concurrency, the parameters you must know!

Airflow has several parameters to tune your tasks and DAGs concurrency.

**Concurrency** defines the number of tasks and DAG Runs that you can execute at the same time (in parallel)

*Starting from the configuration settings*

**parallelism / AIRFLOW\_\_CORE\_\_PARALELISM**

This defines the maximum number of task instances that can run in Airflow per scheduler. By default, you can execute up to 32 tasks at the same time. If you have 2 schedulers: 2 x 32 = 64 tasks.

What value to define here depends on the resources you have and the number of schedulers running.

**max\_active\_tasks\_per\_dag / AIRFLOW\_\_CORE\_\_MAX\_ACTIVE\_TASKS\_PER\_DAG**

This defines the maximum number of task instances allowed to run concurrently in each DAG. By default, you can execute up to 16 tasks at the same time for a given DAG across all DAG Runs.

**max\_active\_runs\_per\_dag / AIRFLOW\_\_CORE\_\_MAX\_ACTIVE\_RUNS\_PER\_DAG**

This defines the maximum number of active DAG runs per DAG. By default, you can have up to 16 DAG runs per DAG running at the same time.

6.SubDAG(Deprecated since airflow v2.2)

**SubDAG:**it is used to group together multiple tasks

Code:

from airflow import DAG

from airflow.operators.bash import BashOperator

from airflow.operators.subdag import SubDagOperator

from subdags.subdag\_downloads import subdag\_downloads

from subdags.subdag\_transforms import subdag\_transforms

from datetime import datetime

with DAG('group\_dag', start\_date=datetime(2022, 1, 1),

schedule\_interval='@daily', catchup=False) as dag:

args = {'start\_date': dag.start\_date, 'schedule\_interval': dag.schedule\_interval, 'catchup': dag.catchup}

downloads = SubDagOperator(

task\_id='downloads',

subdag=subdag\_downloads(dag.dag\_id, 'downloads', args)

)

check\_files = BashOperator(

task\_id='check\_files',

bash\_command='sleep 10'

)

transforms = SubDagOperator(

task\_id='transforms',

subdag=subdag\_transforms(dag.dag\_id, 'transforms', args)

)

downloads >> check\_files >> transforms

from airflow import DAG

from airflow.operators.bash import BashOperator

def subdag\_transforms(parent\_dag\_id, child\_dag\_id, args):

with DAG(f"{parent\_dag\_id}.{child\_dag\_id}",

start\_date=args['start\_date'],

schedule\_interval=args['schedule\_interval'],

catchup=args['catchup']) as dag:

transform\_a = BashOperator(

task\_id='transform\_a',

bash\_command='sleep 10'

)

transform\_b = BashOperator(

task\_id='transform\_b',

bash\_command='sleep 10'

)

transform\_c = BashOperator(

task\_id='transform\_c',

bash\_command='sleep 10'

)

return dag

7.TaskGroups(Better way to group tasks than SubDAGs)

**TaskGroups:**They are also used to group together multiple tasks.

They are simple, straightforward and easier to use than SubDAGs

Code:

from airflow import DAG

from airflow.operators.bash import BashOperator

from airflow.utils.task\_group import TaskGroup

def transform\_tasks():

with TaskGroup("transforms", tooltip="Transform tasks") as group:

transform\_a = BashOperator(

task\_id='transform\_a',

bash\_command='sleep 10'

)

transform\_b = BashOperator(

task\_id='transform\_b',

bash\_command='sleep 10'

)

transform\_c = BashOperator(

task\_id='transform\_c',

bash\_command='sleep 10'

)

return group

from airflow import DAG

from airflow.operators.bash import BashOperator

from groups.group\_downloads import download\_tasks

from groups.group\_transforms import transform\_tasks

from datetime import datetime

with DAG('group\_dag', start\_date=datetime(2022, 1, 1),

schedule\_interval='@daily', catchup=False) as dag:

args = {'start\_date': dag.start\_date, 'schedule\_interval': dag.schedule\_interval, 'catchup': dag.catchup}

downloads = download\_tasks()

check\_files = BashOperator(

task\_id='check\_files',

bash\_command='sleep 10'

)

transforms = transform\_tasks()

downloads >> check\_files >> transforms

8.XCom and BranchPythonOperator

**Xcom**: It is mainly used to share data between tasks.

So the first task pushes data to XCom and the other task pulls data from XCom.

* + It contains information you want to share between your tasks and it is stored into the metadatabase of airflow.
  + It stands for cross communication and its nothing more than a little package that allows to exchange small amount of data
  + SQLite: 2GB
  + Postgres: 1GB
  + mySQL: 64KB

**BranchPythonOperator**: It allows you to execute a python function annd i that python function you return tha task id o the next task you want to execute. based on your condition.

Trigger Rule: defines why your task is triggered. You have many different trigger rules

* all\_success: (default) all parents have succeeded
* all\_failed: all parents are in a failed or upstream\_failed state
* all\_done: all parents are done with their execution
* one\_failed: fires as soon as at least one parent has failed, it does not wait for all parents to be done
* one\_success: fires as soon as at least one parent succeeds, it does not wait for all parents to be done
* none\_failed: all parents have not failed (failed or upstream\_failed) i.e. all parents have succeeded or been skipped
* none\_skipped: no parent is in a skipped state, i.e. all parents are in a success, failed, or upstream\_failed state
* dummy: dependencies are just for show, trigger at will

Code:

from airflow import DAG

from airflow.operators.python import PythonOperator, BranchPythonOperator

from airflow.operators.bash import BashOperator

from datetime import datetime

def \_t1(ti):

ti.xcom\_push(key='my\_key', value=42)

def \_t2(ti):

ti.xcom\_pull(key='my\_key', task\_ids='t1')

def \_branch(ti):

value = ti.xcom\_pull(key='my\_key', task\_ids='t1')

if (value == 42):

return 't2'

return 't3'

with DAG("xcom\_dag", start\_date=datetime(2022, 1, 1),

schedule\_interval='@daily', catchup=False) as dag:

t1 = PythonOperator(

task\_id='t1',

python\_callable=\_t1

)

branch = BranchPythonOperator(

task\_id='branch',

python\_callable=\_branch

)

t2 = PythonOperator(

task\_id='t2',

python\_callable=\_t2

)

t3 = BashOperator(

task\_id='t3',

bash\_command="echo ''"

)

t4 = BashOperator(

task\_id='t4',

bash\_command="echo ''",

trigger\_rule='none\_failed\_min\_one\_success'

)

t1 >> branch >> [t2, t3] >> t4