

Python on Airflow pipelines

As we want to run pipelines many times to collect historical data, we need an scheduling mechanism. This is precisely what Airflow will do for us.

1. Add Airflow code to the pipeline

We will refactor the code of the previous section to run exactly the same pipeline as an Airflow DAG that can be scheduled for a few days to collect historical information.

1.1 Review the changes in the code

The necessary changes for enabling the python pipeline to run as an Airflow DAG are shown in the next cell

```

git > databand-workshop > dags > python dag_airflow.py > select_one_year
1 # File: pythondag_airflow.py
2 # Simple DAG for the Databand hands-on workshop
3
4 # This python file mirrors the previous pythondag.py but modifies
5 # some blocks to enable it as an Airflow DAG. Only those changes are
6 # commented. Refer to pythondag.py to follow the logic
7
8 import pandas as pd
9 import psycopg2
10 from sqlalchemy import create_engine
11 from dbnd import dbnd_tracking, task, dataset_op_logger
12 from datetime import datetime, timedelta
13
14 # These imports are required by Airflow
15 from airflow import DAG
16 from airflow.operators.python import PythonOperator
17
18 # The name of the dataset must be change so that
19 # Airflow can find it
20 motogp_file = '/opt/airflow/dags/sql/motogp.csv'
21
22 @task
23 def read_all_championships():
24     with dataset_op_logger(motogp_file,
25                             "read",
26                             with_schema=True,
27                             with_preview=True,
28                             with_stats=True,
29                             with_histograms=True
30                             ) as logger:
31         motogp_championships = pd.read_csv(motogp_file, sep=';')
32         logger.set(data=motogp_championships)
33
34     return(motogp_championships)
35
36 @task
37 def select_one_year(alldata):
38     with dataset_op_logger(motogp_file,
39                             "read",
40                             with_schema=True,
41                             with_preview=True,
42                             with_stats=True,
43                             with_histograms=True
44                             ) as logger:
45         oneyear = alldata[alldata.Season.eq(2021)]
46         logger.set(data=oneyear)
47
48     return(oneyear)
49
50 @task
51 def write_to_postgres(oneyear):
52
53     myconnntype = "postgresql+psycopg2"
54     mydatabase = "postgres"
55     myhost = "pg-nodeport-postgres.itzrks-110005cc8-21bzmg-6ccd7f378ae819553d37d5f2ee142bd6-0000.us-east.containers.appdomain.cloud"
56     myuser = "postgres"
57     mypassword = "postgres"
58     myport = "30208"
59     myconnstring = myconnntype+'://'+myuser+'@'+mypassword+'@'+myhost+':'+myport+'/'+mydatabase
60     myengine = create_engine(myconnstring)
61
62     with dataset_op_logger(motogp_file,
63                             "write",
64                             with_schema=True,
65                             with_preview=True,
66                             with_stats=True,
67                             with_histograms=True
68                             ) as logger:
69         oneyear.to_sql('motogp', myengine, if_exists='replace', index=False)
70         logger.set(data=oneyear)
71
72     conn = psycopg2.connect(database=mydatabase,
73                             host=myhost,
74                             user=myuser,
75                             password=mypassword,
76                             port=myport)
77     mysqlcount = "select count(*) from motogp"
78     cur = conn.cursor()
79     cur.execute(mysqlcount)
80     result = cur.fetchone()
81
82     return(result[0])
83

```

new imports for Airflow

this is not our local path
but the location where Airflow
will look for the file

a different season (just for fun)

Don't forget to edit this file and set
the right parameters for Postgres
(exactly the same as in the previous file)

[illegible]

As you can see the fundamental changes are located at the bottom of the file where we write a new header for Airflow

1.2. Edit the file `pythondag_airflow.py` to include Postgres and Data-band security parameters

All changes for Airflow are already done in `pythondag_airflow.py` but we need to enter the Postgres and Databand credentials for your particular environment. Please follow the same instructions as shown in the previous chapter under the paragraph 1.3. No more changes are necessary.

1.3. Transfer pythondag_airflow.py to Airflow

As we modified the file, we need to transfer it to Airflow to be registered as a DAG. We begin with the usual login to the cluster

```
# Replace the command with your own one inside the single quotes and run the cell
# Example OC_LOGIN_COMMAND='oc login --token=sha256~3bR5KXgwiUoaQipb2_kIXCDQnVfm_HQy3YwU2m-l
OC_LOGIN_COMMAND='oc login --token=sha256~6Xs6va20JZ2CFhS61HN6bpQC2z075XZbhIJt3tZ8L6w --serv
$OC_LOGIN_COMMAND
oc project airflow
```

We need to verify that the file `pythondag_airflow.py` is located in the DAGs directory.

```
# you may need to modify the cd command to place yourself in the DAGs directory
pwd
cd ../dags
ls -l
```

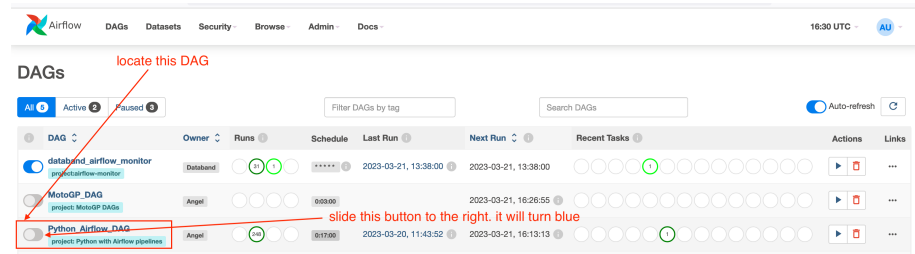
Look for a `pythondag_airflow.py` like this:

And then you can run this cell to transfer the file:

```
# Run this cell to copy the file to the openshift cluster
oc cp pythondag_airflow.py airflow-worker-0:dags/
```

1.4. Enable the run on Airflow

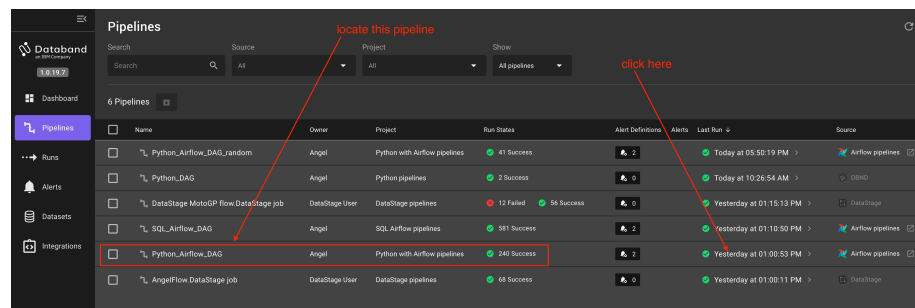
Once the file is transferred to Airflow you may need to wait about 5 minutes until the the DAG is visible. Then, you need to activate it:



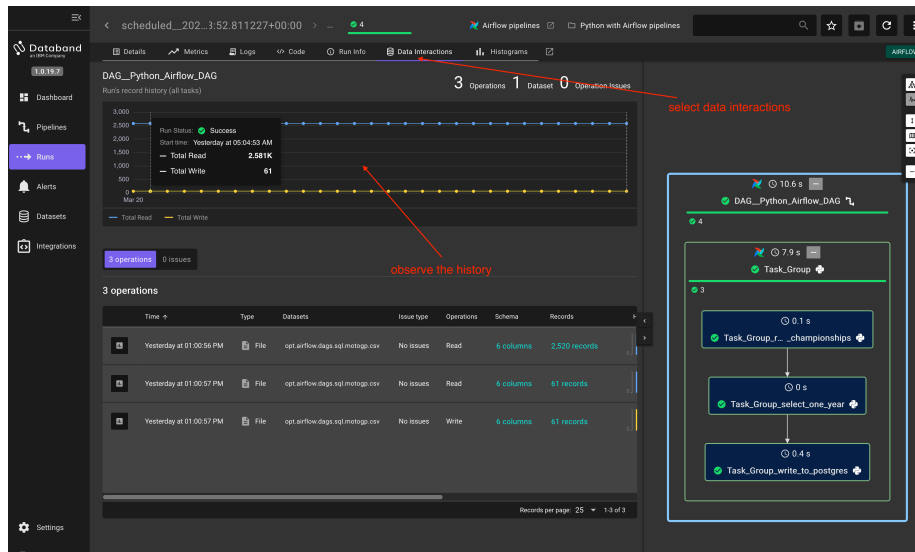
Now, the DAG is activated and will run every 17 minutes. Leave it running for a few days if you wish to see historical data or go to the next section where we see how it will look like.

2. Display performance data with Databand

The new created pipeline can will be shown as `Python_Airflow_DAG` as it is hardcoded in the header section of the python code

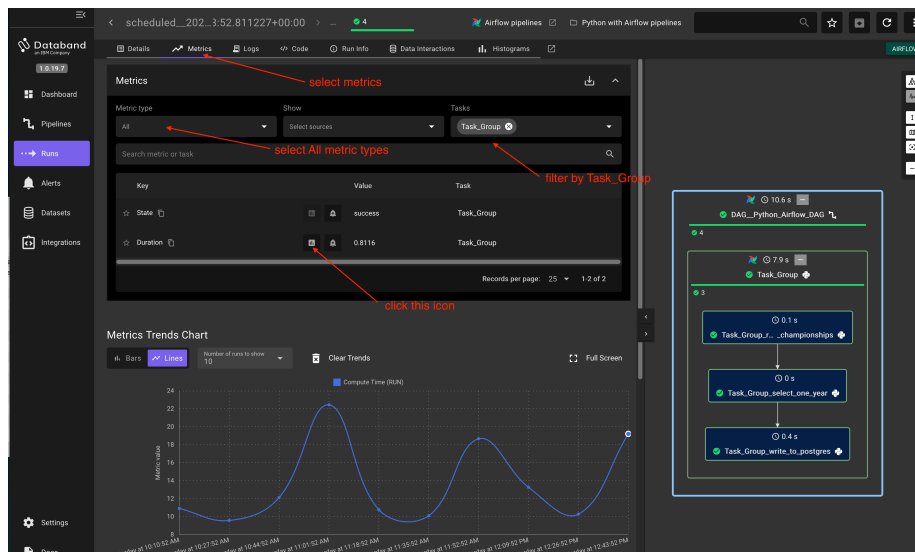


The pipeline ran 240 times so far and the historical data can be shown like follows:

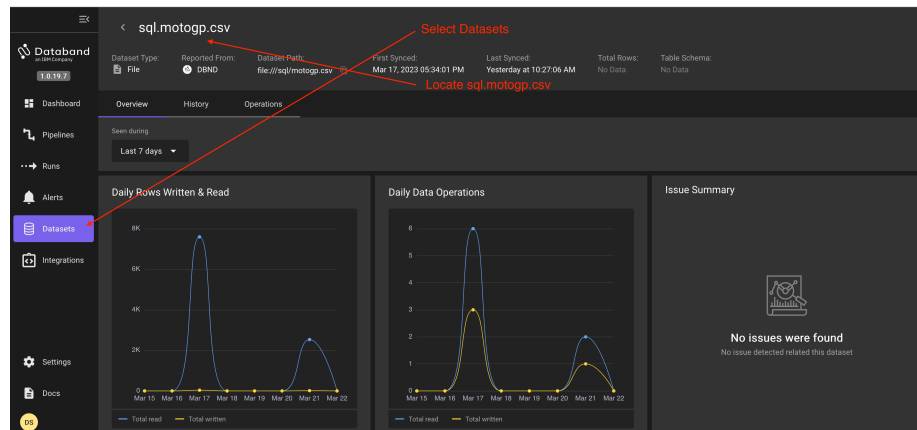


Unfortunately, when things run well, all graphics and trends look like very boring. In our case the number of rows written and read is the same during the whole history.

However, there are ways to see more exciting curves. Just proceed as instructed in the following picture:



Indeed, there are variations in the elapsed runtime caused by the concurrency of several jobs while the performance data was collected. You can now switch to the Datasets view and see the cumulated traffic of records in the last days:



Next Section: DataStage pipelines
Previous Section: Python pipelines
Return to main