Apache Airflow to Get Data

Apache Airflow is a platform to manage workflows that is a crucial role for data intensive applications. One can define, schedule, monitor and troubleshoot data workflows as code that makes maintenance, versioning, dependence management and testing more convenient. Being initiated by Airbnb, today it is an open-source tool backed by the Apache Software Foundation.

Airflow provides robust integrations with major cloud platforms (involving GCP, AWS, MS Azure, etc) as well as local resources. Moreover, it is written in Python that is also used for creating workflows. Accordingly and not surprisingly, it a well-accepted solution by the industry for applications in different scales. It is also important to note that it allows dynamically manage workflows (data pipelines) but workflows themselves are expected to be -almost- static. It is definitely not for streaming.

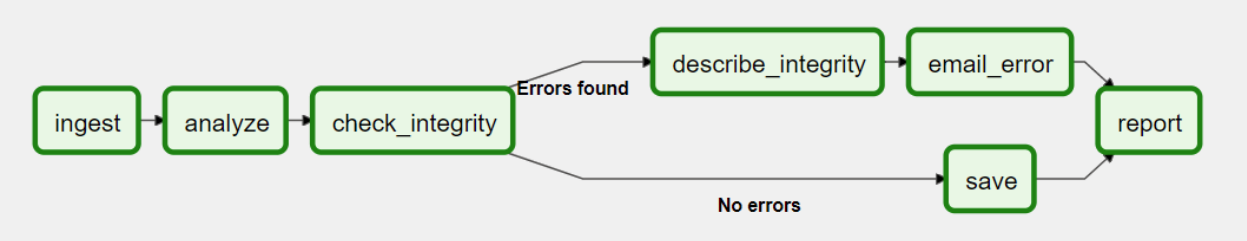
1. The Basic Architecture and Terminology

Task and Directed Acyclic Graph (DAG) are two fundamental concepts to understand how to use Airflow.

A **task** is an atomized and standalone piece of work (action). Airflow helps you define, run and monitor tasks in Python3, bash scripts, etc. It would be any operation on or with data such as transferring, analysis and storage. Tasks are defined using code templates called operators and the building block of all operators is the `BaseOperator`. Generic operators are used for variety of tasks that build DAGs. Moreover, there are specific versions of operators. One of them is sensors that are observing specific points of DAG to detect a specific event to happen. Other tasks with unique functionalities are defined with `@task` decorator that is handled by TaskFlow API. The code snippet given below shows the basic structure of defining tasks within DAG files with examples for BashOperator and PythonOperator.

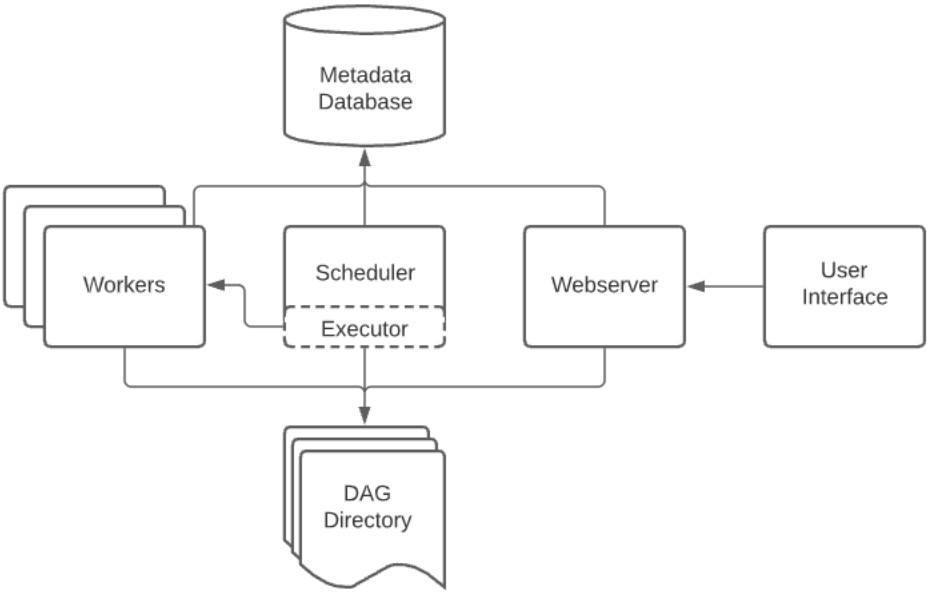
```
from airflow.operators.bash import BashOperator
from airflow.operators.python import PythonOperator

# ... We see complete DAG files below.
# Here is just an example for how to define tasks.

my_Bash_task = BashOperator(
        task_id="Bash_task_for_XX",
        bash_command="....bash....command....."
    )

my_Python_task = PythonOperator(
        task_id="Python_task_for_YY",
        python_callable=a_predefined_Python_function,
        op_kwargs={
            "src_file": "address_to_source_file",
        },
    )
```

A **DAG** represents the interdependence among tasks (see figure 1). Nodes of a DAG are individual tasks whereas the edges correspond to data transition among two tasks.



A basic DAG example (Source: https://airflow.apache.org)

Airflow helps you link tasks to compose DAGs for controlling flow. To do so, it brings together a variety of services as represented in Figure 2.

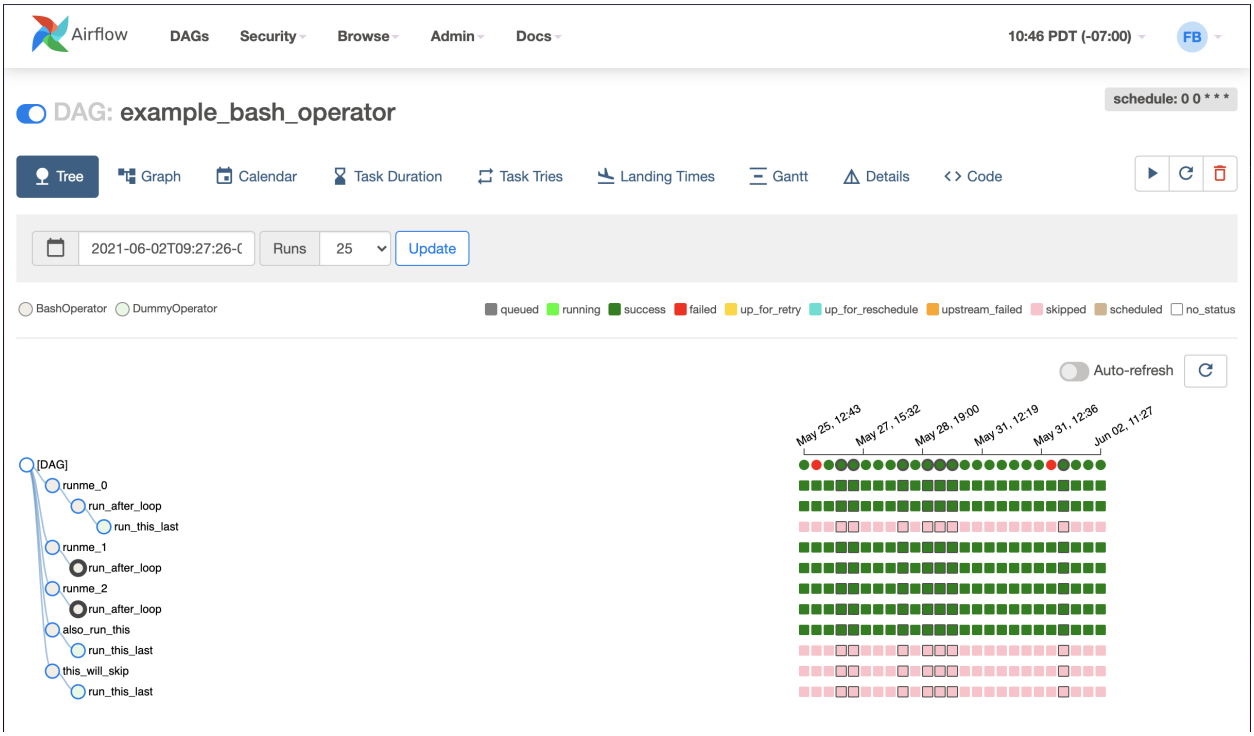


Components and Architecture of Apache Airflow (Source: https://airflow.apache.org)

In architecture of Apache Airflow,

- **Workers** are the components in which tasks are run in line with the commands received from executer.
- **Scheduler** follows dependencies defined for tasks and DAGs. Once these are met, scheduler triggers the tasks in accordance with the given timing policy.
- **Executor** runs tasks either inside the scheduler or by pushing corresponding workers.
- **DAG Directory** is the folder in which `.py` files for each DAG lives.
- **Metadata Database** stores the state of the scheduler, executer and webserver.
- **User interface** helps users to control and follow workflows with a intuitive graphical screen and to reach some outputs from the system (such as logs) easily.
- **Webserver** links the system with user interface for remote control with interactive GUI.

Once tasks and DAGS are defined and the system id activated (usually within containers), users get a screen as shown below where the workflow can be followed.



An example of Apache Airflow web interface (Source: https://airflow.apache.org)

Overviewing basic architecture and the terminology, let's see them in action.

2. Installing Airflow

Airflow is a highly configurable tool. Accordingly, it's installation can be customized due to the requirements of each specific application. Moreover, it is a common practice to host it in a container to isolate from system interactions and dependency conflicts. Brief guide presented here is based on the official guide and the show case is originally presented by DataTalks Club in during Data Engineering Zoomcamp (2022 cohort). The code base and the configurations files used in this tutorial are available here. The case in this tutorial aims to

- get large number of files (1000+) from a public dataset (OEDI photovoltaic systems dataset) to the local machine in `.csv` format
- convert them to `.parquet` format for more effective computation on cloud in following steps,
- upload data to a bucket on Google Cloud Platform (GCP),
- transfer them from bucket to BigQuery for further analysis.

Parquet is a free and open source columnar storage format backed by Apache Software Foundation. It allows efficient compression and encoding within Hadoop ecosystem independent of frameworks or programming languages. Hence it is a common format for public databases and actually OEDI PV dataset also has a version in `.parquet` format. However, in line with the corresponding DataTalks Club tutorials and videos, the conversion process is involved to present variety of operations. Otherwise, it is possible to directly transfer `.parquet` files to GCP using Airflow or any other tool.

2.1 Containerization

Following the best practices, the installation of the Airflow will be containerized. Accordingly, the `Dockerfile` and the `docker-compose.yaml` files have crucial role. Airflow documentation presents a typical docker-compose file for make life easier for newcomers. It uses the official airflow image: `apache/airflow:2.2.3`. Hence, the Dockerfile developed by DataTalks Club starts with it system requirements and settings. Then, the SDK for GCP is downloaded and installed for cloud integrations. The file concludes with

- setting a home directory for Airflow within container,
- including additional scripts if necessary
- setting a parameter for the user ID of Airflow.

(Note: it is a common practice to host the following files and folders within a folder, preferable named as ‘airflow’.

```
# First-time build can take upto 10 mins.

FROM apache/airflow:2.2.3

ENV AIRFLOW_HOME=/opt/airflow

USER root
RUN apt-get update -qq && apt-get install vim -qqq
# git gcc g++ -qqq

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Ref: https://airflow.apache.org/docs/docker-stack/recipes.html

SHELL ["/bin/bash", "-o", "pipefail", "-e", "-u", "-x", "-c"]

ARG CLOUD_SDK_VERSION=322.0.0
ENV GCLOUD_HOME=/home/google-cloud-sdk

ENV PATH="${GCLOUD_HOME}/bin/:${PATH}"

RUN DOWNLOAD_URL="https://dl.google.com/dl/cloudsdk/channels/rapid/downloads/google-cloud-sdk-${CLOUD_SDK_VERSION}-linux-x86_64.tar.gz" \
    && TMP_DIR="$(mktemp -d)" \
    && curl -fL "${DOWNLOAD_URL}" --output "${TMP_DIR}/google-cloud-sdk.tar.gz" \
    && mkdir -p "${GCLOUD_HOME}" \
    && tar xzf "${TMP_DIR}/google-cloud-sdk.tar.gz" -C "${GCLOUD_HOME}" --strip-components=1 \
    && "${GCLOUD_HOME}/install.sh" \
       --bash-completion=false \
       --path-update=false \
       --usage-reporting=false \
       --quiet \
    && rm -rf "${TMP_DIR}" \
    && gcloud --version

WORKDIR $AIRFLOW_HOME

COPY scripts scripts
RUN chmod +x scripts

USER $AIRFLOW_UID
```

The `docker-compose.yaml` file suggested by the Airflow documentation should be downloaded next to the Dockerfile:

```
curl -LfO 'https://airflow.apache.org/docs/apache-airflow/2.2.5/docker-compose.yaml'
```

It involves variety of functionalities that Airflow need and presents (descripition are from official guide):

- `airflow-scheduler` - The scheduler monitors all tasks and DAGs, then triggers the task instances once their dependencies are complete.
- `airflow-webserver` - The webserver is available at `http://localhost:8080`.
- `airflow-worker` - The worker that executes the tasks given by the scheduler.
- `airflow-init` - The initialization service.
- `flower` - The flower app for monitoring the environment. It is available at `http://localhost:5555`.
- `postgres` - The database.
- `redis` - The redis - broker that forwards messages from scheduler to worker.

The Airflow documentation also suggest to create folders to keep DAGs, log files and plugins outside the container:

```
mkdir -p ./dags ./logs ./plugins
```

Moreover, a `.env` file should be created to declare the user ID to the docker-compose:

```
echo -e "AIRFLOW_UID=$(id -u)" > .env
```

Having the base image, next step is to include GCP related components to the `docker-compose.yaml` and to set credentials. DataTalks Club template includes the following lines:

```
# (line 61 to 66)
GOOGLE_APPLICATION_CREDENTIALS: /.google/credentials/google_credentials.json
AIRFLOW_CONN_GOOGLE_CLOUD_DEFAULT: 'google-cloud-platform://?extra__google_cloud_platform__key_path=/.google/credentials/google_credentials

# TODO: Please change GCP_PROJECT_ID & GCP_GCS_BUCKET, as per your config
GCP_PROJECT_ID: 'YOUR-PROJECT-ID'
GCP_GCS_BUCKET: 'YOUR-BUCKET-NAME'

# line 72 >> link to the credentials file (at your host machine) for your GCP service account
- ~/.google/credentials/:/.google/credentials:ro
```

Also note that DataTalks template replaces the `image` tag in original document with the `build` of the Dockerfile (Line 47 to 49). The rest of the `docker-compose.yaml` file is ~300 line that is needless to display here. Please investigate the file downloaded (with `curl` command given above) and visit DataTalks Repository.

2.2 Running the Containers

Once gathering the necessary files and folders, it's time to build and up the service with the following commands:

```
docker-compose build  #would require 10-15 mins

docker-compose up airflow-init #requires ~1 min

docker-compose up #requires 2-3 mins
```

As mentioned above, Airflow has a webserver that provides an interactive GUI (`localhost:8080`) to monitor and control the processes declared by DAGs.

3. Composing DAGs to Use OEDI Data

Having an Airflow setup up-and-running, next step is to compose DAG files to execute tasks. The complete code for this part can be seen here. In this text, only the custom parts for OEDI PV dataset will be reviewed. Rest of the code is in line with DataTalks tutorial.

As like typical Python files, DAG files starts with imports followed by declarations. First part of the declarations involve the environmental parameters refer to Dockerfile and docker-compose files:

```
AIRFLOW_HOME = os.environ.get("AIRFLOW_HOME", "/opt/airflow/")
```

```
PROJECT_ID = os.environ.get("GCP_PROJECT_ID")
BUCKET = os.environ.get("GCP_GCS_BUCKET")
BIGQUERY_DATASET = os.environ.get("BIGQUERY_DATASET", pv_system_label)
```

Rest of the declarations are related to the OEDI data lake that is hosted on AWS-S3 buckets. An example URL for the files:

```
https://oedi-data-lake.s3.amazonaws.com/pvdaq/csv/pvdata/system_id=1199/year=2011/month=1/day=1/system_1199__date_2011_01_01.csv
```

It can be separated into following components:

- URL Core for PV dataset: `https://oedi-data-lake.s3.amazonaws.com/pvdaq/csv/pvdata/`
- System declaration with a numeric code: `system_id=1199/`
- Year declaration: `year=2011/`
- Month declaration: `month=1/`
- Day declaration: `day=1/`
- File name declaration: `system_1199__date_2011_01_01.csv`

Hence, we need a parameter to define system ID and independent year, month and day parameters to target specific files. The ID is just a string:

```
pv_system_ID = '1430'
```

To manipulate date parameters, we can embed Python codes within a Jinja template:

`{{ execution_date.strftime(\'%Y\') }}`

Hence, we can declare parametrized URL as:

```
URL_PREFIX='https://oedi-data-lake.s3.amazonaws.com/ \
      pvdaq/csv/pvdata/ \
      system_id='+pv_system_ID+ \
      '/year='+'{{ execution_date.strftime(\'%Y\') }}'+\
      '/month={{ execution_date.strftime(\'%-m\') }}' +\
      '/day='+'{{ execution_date.strftime(\'%-e\') }}'

URL_TEMPLATE= URL_PREFIX +\
      '/system_'+pv_system_ID+'__date_'+\
      '{{ execution_date.strftime(\'%Y\') }}'+\
      '_{{ execution_date.strftime(\'%m\') }}_'+\
      '{{ execution_date.strftime(\'%d\') }}'+'.csv'
```

In a similar manner, it is useful to rename downloaded files before conversion:

```
OUTPUT_FILE_TEMPLATE = AIRFLOW_HOME + \
'/pvsys'+pv_system_ID+\
'data_{{ execution_date.strftime(\'%Y\') }}\
{{ execution_date.strftime(\'%m\') }}\
{{ execution_date.strftime(\'%d\') }}.csv'

parquet_file = OUTPUT_FILE_TEMPLATE.\
replace('.csv', '.parquet')
```

DAG code continues with 2 function definitions (namely `format_to_parquet` and `upload_to_gcs` defined by DataTalk Club to be used in operators. In line with the 4 tasks given at the beginning of Section 2, the DAG involves 4 operators (remember the definition and role of operators given in Section 1).

The first operator gets data from the corresponding link by using the `curl` command with URL and output templates declared above:

```
download_task = BashOperator(
        task_id='get_data',
        bash_command=f'curl -sSL {URL_TEMPLATE} > {OUTPUT_FILE_TEMPLATE}'
    )
```

The second operator converts `.csv` file to `.parquet` file using the `format_to_parquet` funtion:

```
convert_task = PythonOperator(
        task_id="convert_csv_to_parquet",
        python_callable=format_to_parquet,
        op_kwargs={
            "src_file": OUTPUT_FILE_TEMPLATE,
        },
    )
```

The third operator sends the converted file to GCP bucket using the `upload_to_gcs` function with parametrized system and file names:

```
local_to_gcs_task = PythonOperator(
        task_id="local_to_gcs_task",
        python_callable=upload_to_gcs,
        op_kwargs={
            "bucket": BUCKET,
            "object_name": f"{pv_system_label}/{parquet_file_name}",
            "local_file": f"{parquet_file}",
        },
    )
```

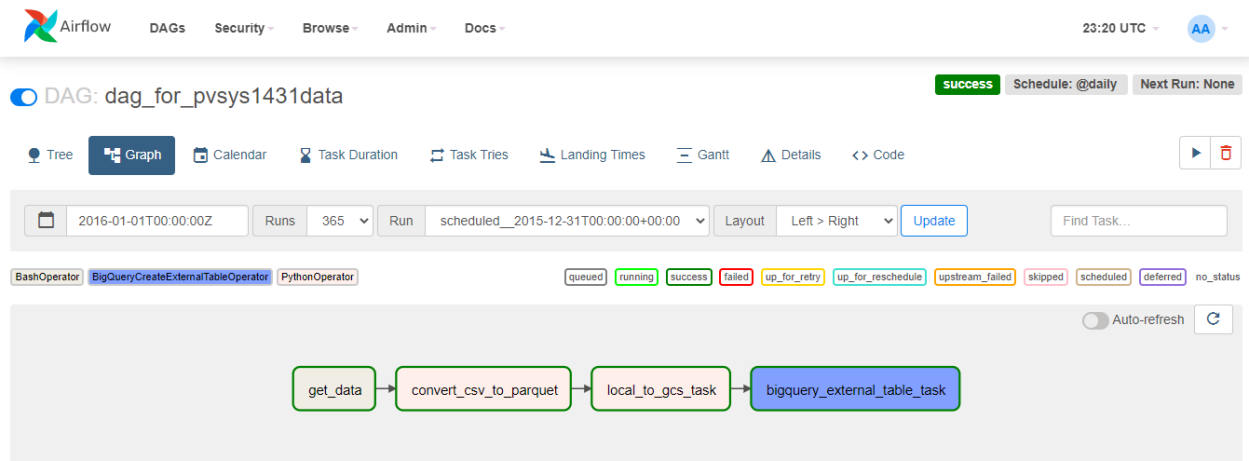
The last operator transfers files from bucket to BigQuery with a specific operator defined for this task:

```
bigquery_external_table_task = BigQueryCreateExternalTableOperator(
        task_id="bigquery_external_table_task",
        table_resource={
            "tableReference": {
                "projectId": PROJECT_ID,
                "datasetId": BIGQUERY_DATASET,
                "tableId": pv_system_label+"_"+"{{execution_date.strftime(\'%d\') }}{{execution_date.strftime(\'%m\') }}{{execution_date.st
            },
            "externalDataConfiguration": {
                "sourceFormat": "PARQUET",
                "sourceUris": [f"gs://{BUCKET}/{pv_system_label}/{parquet_file_name}"],
            },
        },
    )
```

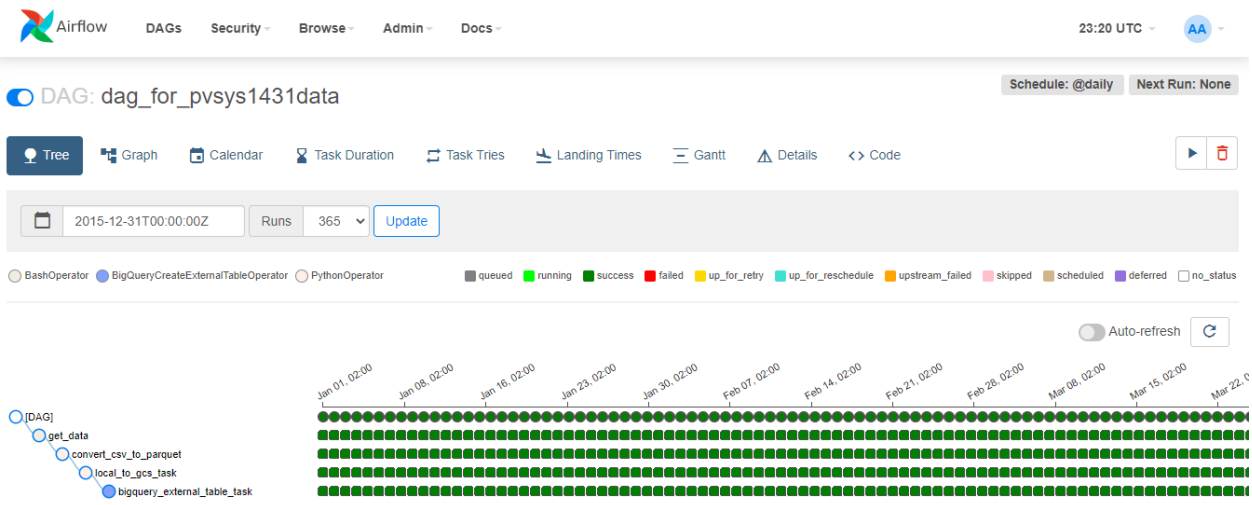
The last step is to ‘chain’ all these operators to build the ‘tree’ of tasks:

```
download_task >> convert_task \
    >> local_to_gcs_task >> bigquery_external_table_task
```

Initiating Airflow with such a DAG definition for 4 PV system (IDs with 1430 to 1433), one should get the following graph:



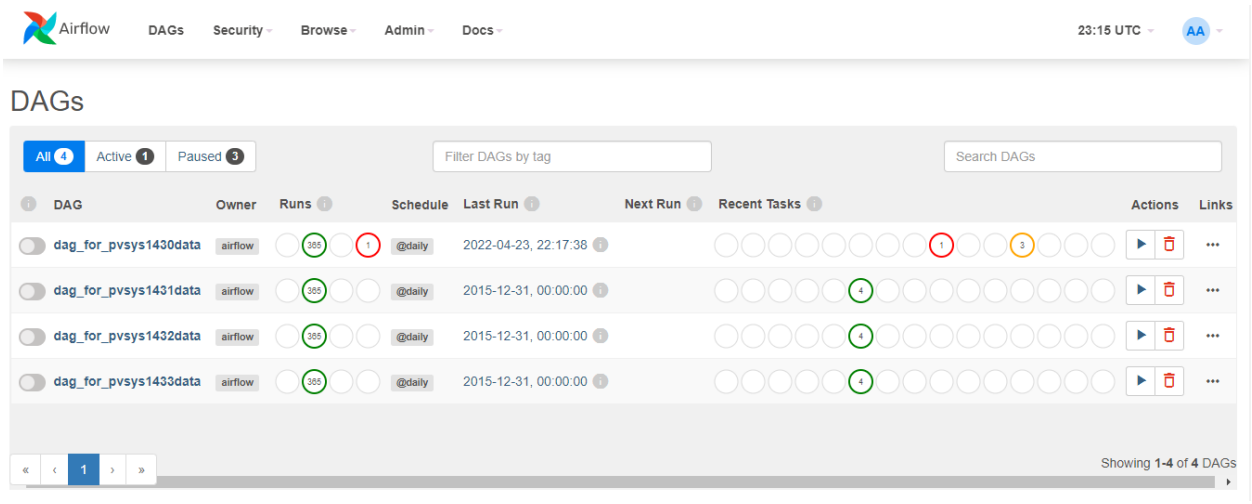
After triggering a DAG, the following tree visualizatio appears:

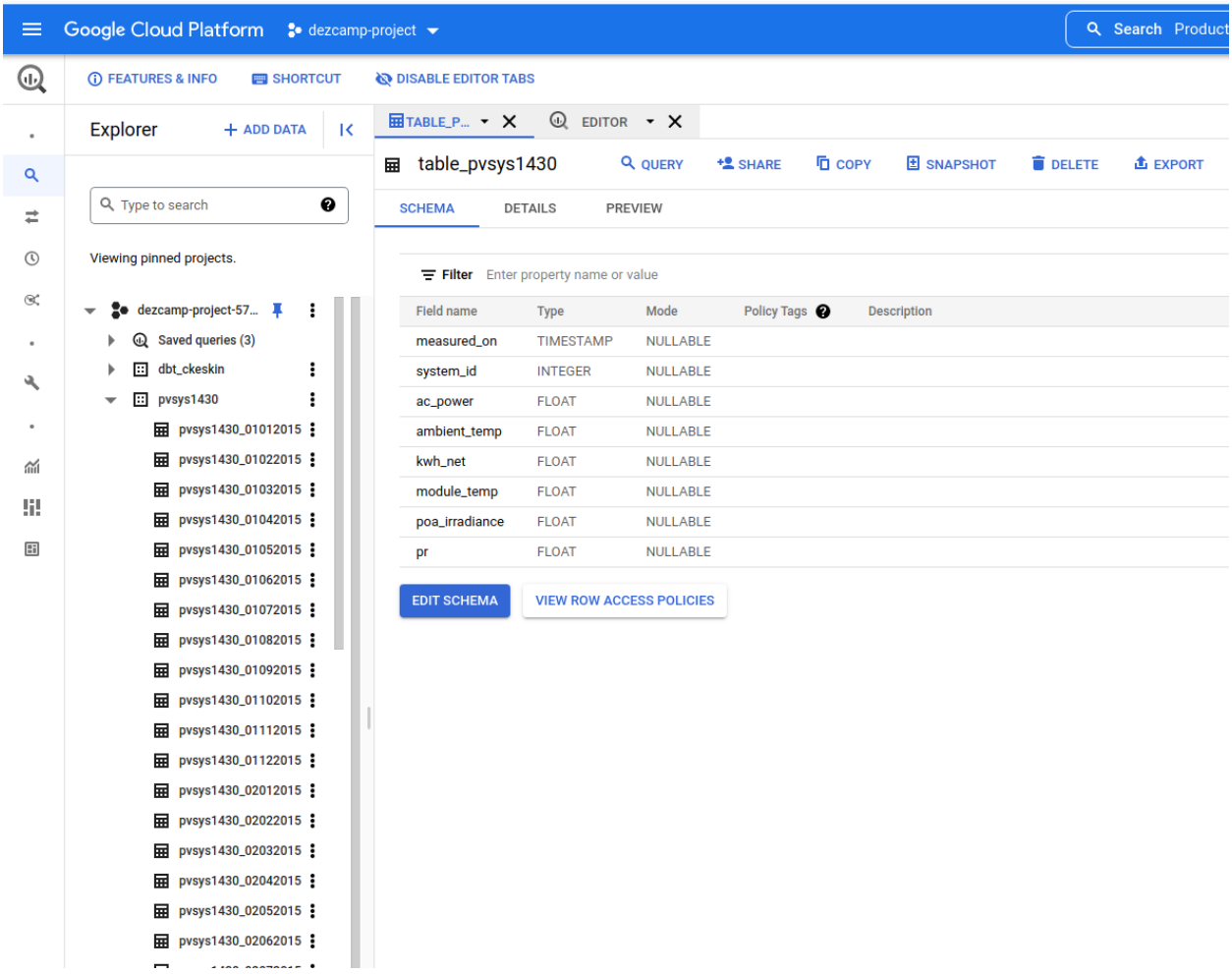


This also visualize how the DAG works. Referring to the date declarations given in DAT initiation:

```
with DAG(
    dag_id="dag_for_"+pv_system_label+"data",
    start_date=datetime(2015, 1, 1),
    end_date=datetime(2015, 12, 31),
    schedule_interval="@daily",

) as dag:
```

Airflow parametrizes year, month and day information to be used in DAG. These parameters are used in the URL template explained above to get the exact file for each iteration. Hence, beginning from `start date`, Airflow iterates the DAG for each day up to the `end_date`. After completing each run for 4 systems, the following creen appears on Airflow DAGs menu and BigQuery:


Airflow
DAGs
Security
Browse
Admin
Docs
23:15 UTC
AA
DAGs
All 4
Active 1
Paused 3
Filter DAGs by tag
Search DAGs
DAG
Owner
Runs
Schedule
Last Run
Next Run
Recent Tasks
Actions
Links
dag_for_pvsys1430data
dag_for_pvsys1431data
dag_for_pvsys1432data
dag_for_pvsys1433data
airflow
@daily
2022-04-23, 22:17:38
2015-12-31, 00:00:00
Showing 1-4 of 4 DAGs


Google Cloud Platform
dezcamp-project
Search
Product
FEATURES & INFO
SHORTCUT
DISABLE EDITOR TABS
Explorer
ADD DATA
Type to search
Viewing pinned projects.
dezcamp-project-57...
Saved queries (3)
dbt_ckeskin
pvsys1430
pvsys1430_01012015
pvsys1430_01022015
pvsys1430_01032015
pvsys1430_01042015
pvsys1430_01052015
pvsys1430_01062015
pvsys1430_01072015
pvsys1430_01082015
pvsys1430_01092015
pvsys1430_01102015
pvsys1430_01112015
pvsys1430_01122015
pvsys1430_02012015
pvsys1430_02022015
pvsys1430_02032015
pvsys1430_02042015
pvsys1430_02052015
pvsys1430_02062015
TABLE_P...
EDITOR
table_pvsys1430
QUERY
SHARE
COPY
SNAPSHOT
DELETE
EXPORT
SCHEMA
DETAILS
PREVIEW
Filter
Enter property name or value
Field name
Type
Mode
Policy Tags
Description
measured_on
TIMESTAMP
NULLABLE
system_id
INTEGER
NULLABLE
ac_power
FLOAT
NULLABLE
ambient_temp
FLOAT
NULLABLE
kwh_net
FLOAT
NULLABLE
module_temp
FLOAT
NULLABLE
poa_irradiance
FLOAT
NULLABLE
pr
FLOAT
NULLABLE
EDIT SCHEMA
VIEW ROW ACCESS POLICIES