How To - AirFlow

1.1 What is Airflow?

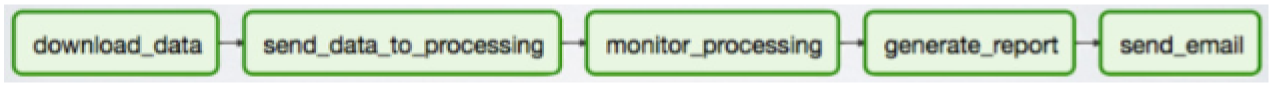
Airflow was developed by engineers at AirBnB to provide a standardized way to handle multiple ETL processes around an Enterprise Data Warehouse system.Airflow is written in Python but is language agnostic. It utilizes rabbitMQ, Jinja

- Airflow is a platform to programmaticaly author, schedule and monitor workflows or data pipelines.
- Use Airflow to author workflows as Directed Acyclic Graphs (DAGs) of tasks. The Airflow scheduler executes your tasks on an array of workers while following the specified dependencies.
- Easy to visualize pipelines running in production, monitor progress, and troubleshoot issues when needed.

1.2 What is a Workflow?

- a sequence of tasks
- started on a schedule or triggered by an event
- frequently used to handle big data processing pipelines

A typical workflows



- Download data from source

- send data somewhere else to process

- Monitor when the process is completed

-Get the result and generate the report

- Send the report out by email

1.3 How Apache Airflow Works

Apache Airflow accomplishes the tasks by taking DAG(Directed Acyclic Graphs) as an array of the workers, some of these workers have particularized contingencies.

- A workflow (data-pipeline) management system developed by Airbnb
- A framework to define tasks & dependencies in python
- Executing, scheduling, distributing tasks across worker nodes.
- View of present and past runs, logging feature
- Extensible through plugins
- Nice UI, possibility to define REST interface
- Interact well with database

Principles

- **Dynamic**: Airflow pipelines are configuration as code (Python), allowing for dynamic pipeline generation. This allows for writing code that instantiates pipelines dynamically.
- **Extensible**: Easy to initiate the operators, executors and extend the library so that it fits the level of abstraction that suits your environment.
- **Elegant**: Airflow pipelines are lean and explicit. Parameterizing your scripts is built into the core of Airflow using the powerful **Jinja** templa ting engine.
- **Scalable**: The architecture of Airflow composed of standardized units which also use messaging technique for queuing the number of workers and moreover it is scalable to infinity.

Use cases

Airflow as an orchestration tool to coordinate work done by other services. It’s not a data streaming solution—even though tasks can exchange some metadata, they do not move data among themselves. Example types of use cases suitable for Airflow:

- ETL (extract, transform, load) jobs - extracting data from multiple sources, transforming for analysis and loading it into a data store

- Machine Learning pipelines
- Data warehousing
- Orchestrating automated testing
- Performing backups

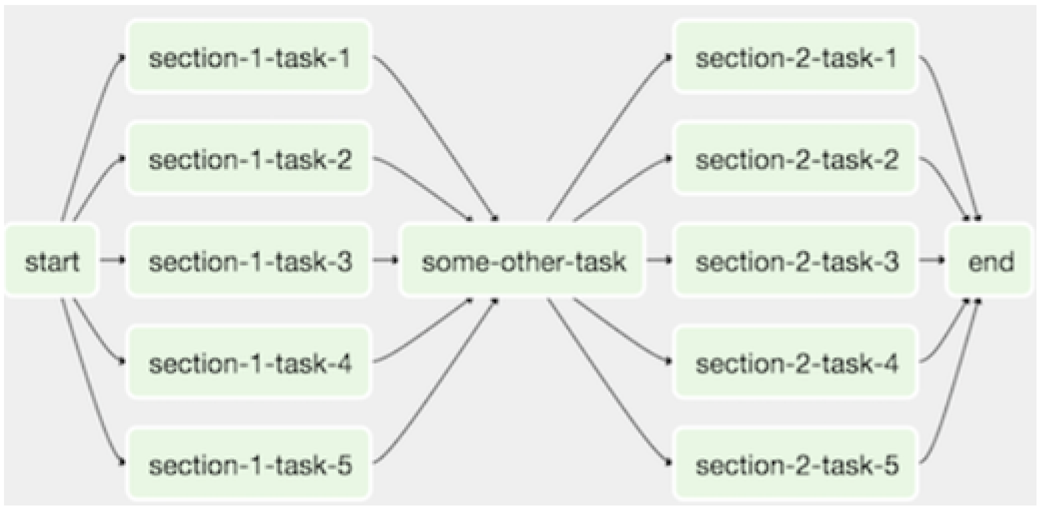
It is generally best suited for regular operations which can be scheduled to run at specific times.

2. Core concepts

2.1 Airflow DAG

First of them is the DAG - short for Directed Acyclic Graph. It’s a collection of all the tasks you want to run, taking into account dependencies between them.

- Workflow as a Directed Acyclic Graph (DAG) with multiple tasks which can be executed independently.
- Airflow DAGs are composed of Tasks.



2.2 Airflow operators

While DAGs describe how to run a workflow, Airflow operators determine what actually gets done. There are several types of operators:

OPERATOR : Action operators which perform a single operation and return e.g. `BashOperator`, `GoogleCloudStorageDownloadOperator`

Sensor : sensors which pause the execution until certain criteria are met, such as until a certain key appears in `S3` e.g. `GoogleCloudStorageObjectSensor`

Executor : transfer operators which connect 2 services and enable sending data between them e.g. `GoogleCloudStorageToS3Operator`

An operator is simply a Python class with an “execute()” method, which gets called when it is being run.

```
class ExampleOperator(BaseOperator):
    def execute(self, context):
        # Do something
        pass
```

In the same vein a sensor operator is a Python class with a “poke()” method, which gets called repeatedly until “True” is returned.

```
class ExampleSensorOperator(BaseSensorOperator):
    def poke(self, context):
        # Check if the condition occurred
        return True
```

2.3 Task

In order to execute an operator we need to create a task, which is a representation of the operator with a particular set of input arguments.

example: we have one BashOperator, but we can create three different “bash tasks” in a DAG, where each task is passed a different bash command to execute. So far we have the DAG, operators and tasks.

Source

https://airflow.apache.org/

https://www.xenonstack.com/insights/apache-airflow/

https://www.applydatascience.com/airflow/airflow-tutorial-introduction/

https://www.polidea.com/blog/apache-airflow-tutorial-and-beginners-guide/