Airflow

Airflow

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

ETL

Machine Learning pipelines

Data warehousing

Orchestrating automated testing

Performing backups

Oct. 2014

Airflow was started at Airbnb

The project joined the Apache Software Foundation's Incubator program

Mar. 2016

Core concepts

- **Airflow DAG**

short for Directed Acyclic Graph. It's a collection of all the tasks you want to run, taking into account dependencies between them. The DAG doesn't actually care about what goes on in its tasks - it doesn't do any processing itself. Its job is to make sure that whatever they do happens at the right time and in the right order.

- **Airflow operators**

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

1. BashOperator - executes a bash command
2. PythonOperator - calls an arbitrary Python function (python_operator.py)
3. EmailOperator - sends an email
4. SimpleHttpOperator - sends an HTTP request
5. MySqlOperator, SqliteOperator, PostgresOperator, MsSqlOperator, OracleOperator, JdbcOperator, etc. - executes a SQL command
6. Sensor - waits for a certain time, file, database row, S3 key, etc… (filesensor.py)

Core concepts

- 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.

- TaskInstance

When the DAG is run, each Task spawns a TaskInstance - an instance of a task tied to a particular time of execution. All task instances in a DAG are grouped into a DagRun.

DAG Assignment

Operators do not have to be assigned to DAGs immediately. However, once an operator is assigned to a DAG, it can not be transferred or unassigned.

dag = DAG('my_dag', start_date=datetime(2019, 5, 1))

- sets the DAG explicitly

explicit_op = DummyOperator(task_id='op1', dag=dag)

- deferred DAG assignment

deferred_op = DummyOperator(task_id='op2’)

deferred_op.dag = dag

- inferred DAG assignment (linked operators must be in the same DAG)

inferred_op = DummyOperator(task_id='op3')
inferred_op.set_upstream(deferred_op)

Core concepts

- Scheduler

The Airflow scheduler monitors all tasks and all DAGs, and triggers the task instances whose dependencies have been met.

Note that if you run a DAG on a schedule_interval of one day, the run stamped 2016-01-01 will be trigger soon after 2016-01-01T23:59. In other words, the job instance is started once the period it covers has ended.

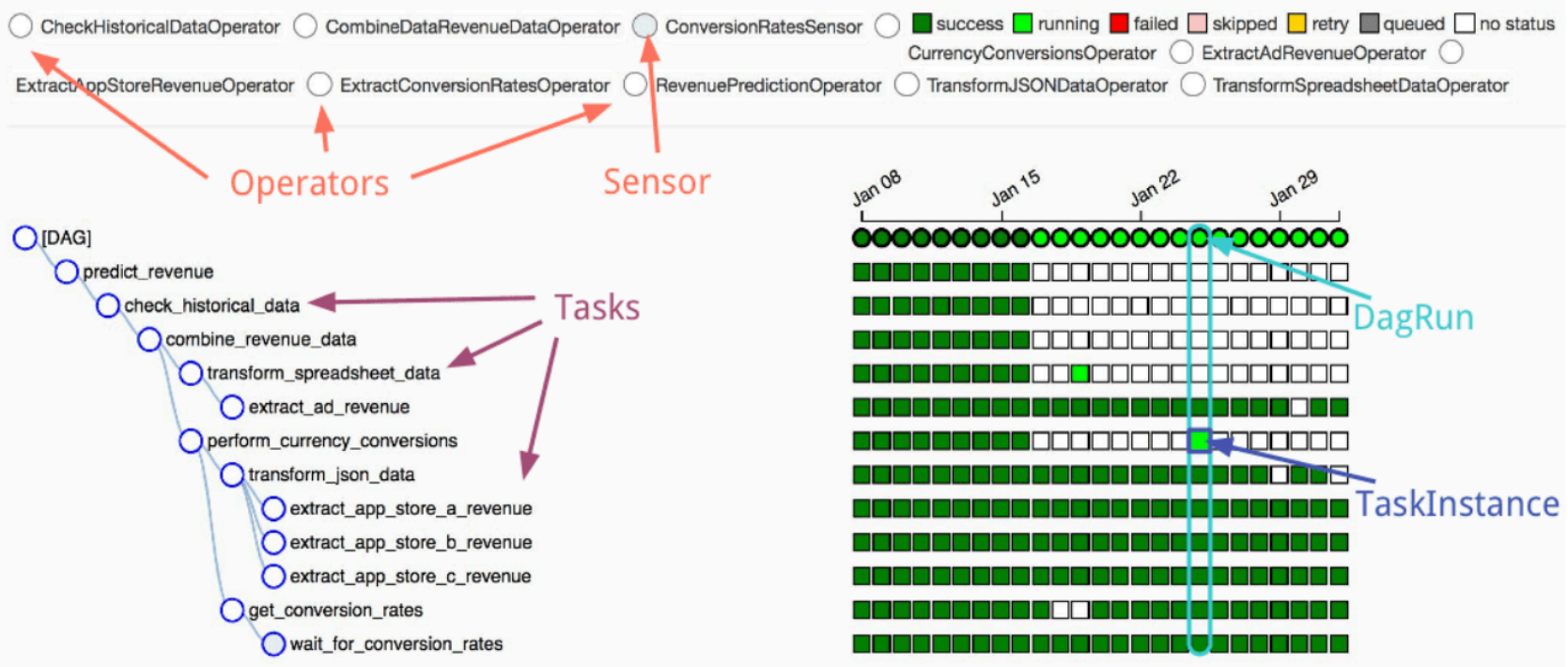


Image source: Understanding Apache Airflow's key concepts

- a DAG consists of tasks, which are parameterized representations of operators. Each time the DAG is executed a DagRun is created which holds all TaskInstances made from tasks for this run.

Prerequisites

 # install from pypi using pip

- pip install apache-airflow

 # initialize the database

- airflow initdb

 # start the web server, default port is 8080

- airflow webserver -p 8080

 # start the scheduler

- airflow scheduler

Command Line

- Pause a DAG

  airflow pause dag_id
- Resume a paused DAG

  airflow unpause dag_id
- Trigger a DAG run

  airflow trigger_dag dag_id
- Delete all DB records related to the specified DAG

  airflow delete_dag dag_id

Command Line

- Run a single task instane

  airflow run dag_id task_id execution_date
- Test a task instance. This will run a task without checking for dependencies or recording its state in the database.

  airflow test dag_id task_id execution_date
- List all the DAGs

  airflow list_dags
- Get the status of a dag run

  airflow dag_state dag_id execution_date
- Get the status of a task instance

  airflow task_state dag_id task_id execution_date

Command Line

- Run subsections of a DAG for a specified date range. If reset_dag_run option is used, backfill will first prompt users whether airflow should clear all the previous dag_run and task_instances within the backfill date range. If rerun_failed_tasks is used, backfill will auto re-run the previous failed task instances within the backfill date range.

  airflow backfill dag_id –s start_date –e end_date

- List the tasks within a DAG

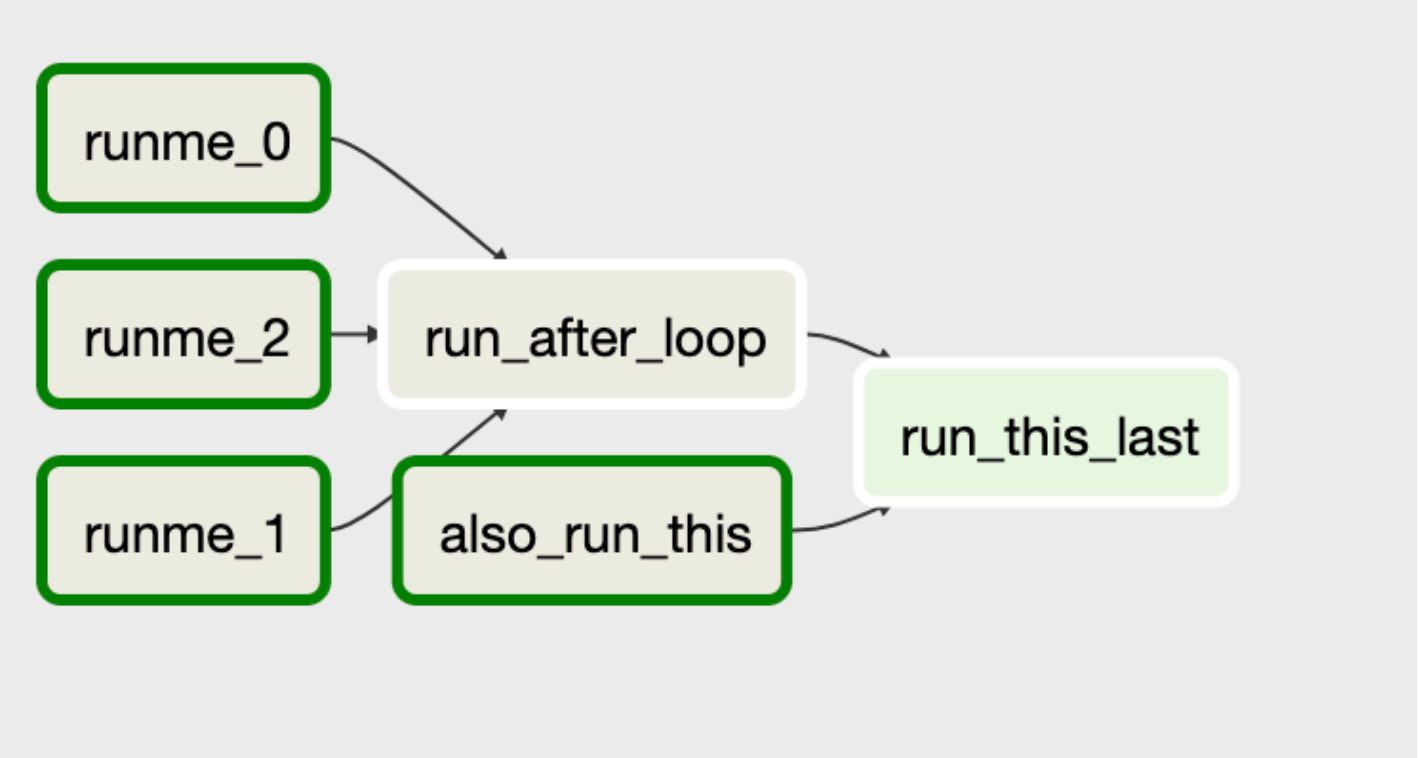
  airflow list_tasks dag_id

- Clear a set of task instance, as if they never ran

  airflow clear dag_id

Setting up Dependencies

- t1.set_downstream(t2)
- t2.set_upstream(t1)
- t1 >> t2
- t2 << t1
- t1 >> t2 >> t3
- t1.set_downstream([t2, t3])
- t1 >> [t2, t3]
- [t2, t3] << t1
- op1 >> op2 >> op3 << op4
  is equivalent to:
  op1.set_downstream(op2)
  op2.set_downstream(op3)
  op3.set_upstream(op4)
- dag >> op1 >> op2
  is equivalent to:
  op1.dag = dag
  op1.set_downstream(op2)



Example (example_bash_operator.py)

- Airflow list_dags
- airflow list_tasks testdag
- airflow test testdag run_after_loop 2019-5-1
- airflow backfill testing -s 2019-04-20 -e 2019-04-21

Core concepts

- **Hook**

Hooks are interfaces to external platforms and databases like Hive, S3, MySQL, Postgres, HDFS, and Pig.

```
from airflow.hooks.postgres_hook import PostgresHook

src = PostgresHook(postgres_conn_id='source', schema='source_schema’)

src_conn = src.get_conn()

cursor = src_conn.cursor()

cursor.execute("SELECT * FROM users;")

src_cursor.close()

src_conn.close()
```

Trigger Rules

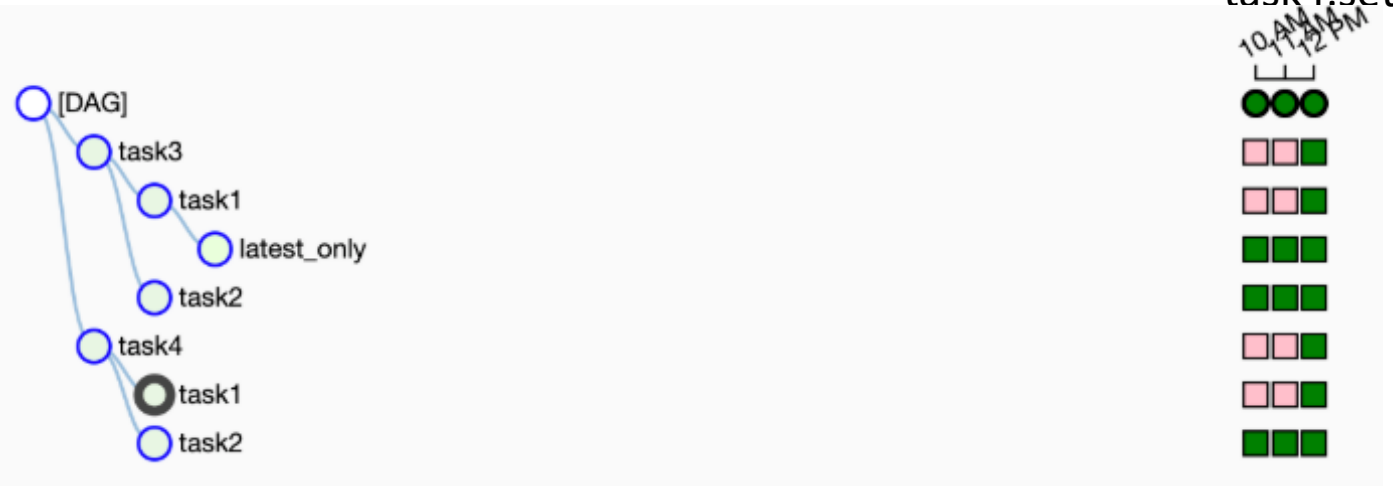
- all_success: (default) all parents have succeeded
- all_failed: all parents are in a failed or upstream_failed state
- all_done: all parents are done with their execution
- one_failed: fires as soon as at least one parent has failed, it does not wait for all parents to be done
- one_success: fires 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

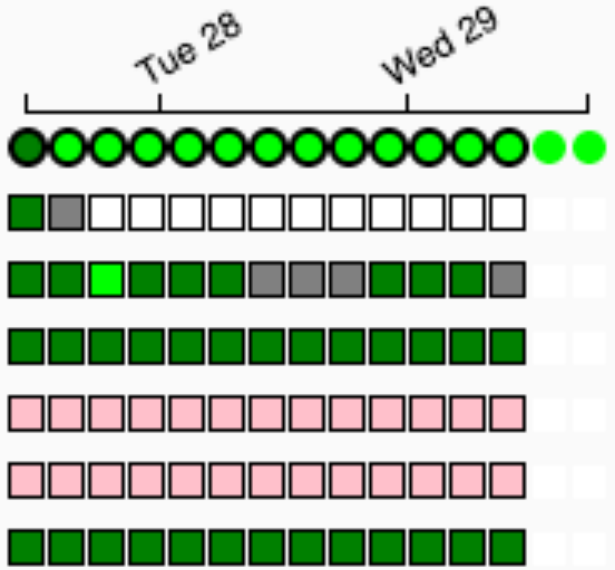
Eg. join = DummyOperator(task_id='join', dag=dag,trigger_rule='none_failed')

Latest Run Only

- task4 is downstream of task1 and task2. It will be first skipped directly by LatestOnlyOperator, even its trigger_rule is set to all_done.

```
dag = DAG( dag_id='latest_only_with_trigger',
schedule_interval=dt.timedelta(hours=1),
start_date=dt.datetime(2019, 2, 28), )

latest_only = LatestOnlyOperator(task_id='latest_only', dag=dag)

task1 = DummyOperator(task_id='task1', dag=dag)
task1.set_upstream(latest_only)

task2 = DummyOperator(task_id='task2', dag=dag)

task3 = DummyOperator(task_id='task3', dag=dag)
task3.set_upstream([task1, task2])

task4 = DummyOperator(task_id='task4', dag=dag,
trigger_rule=TriggerRule.ALL_DONE)
task4.set_upstream([task1, task2])
```



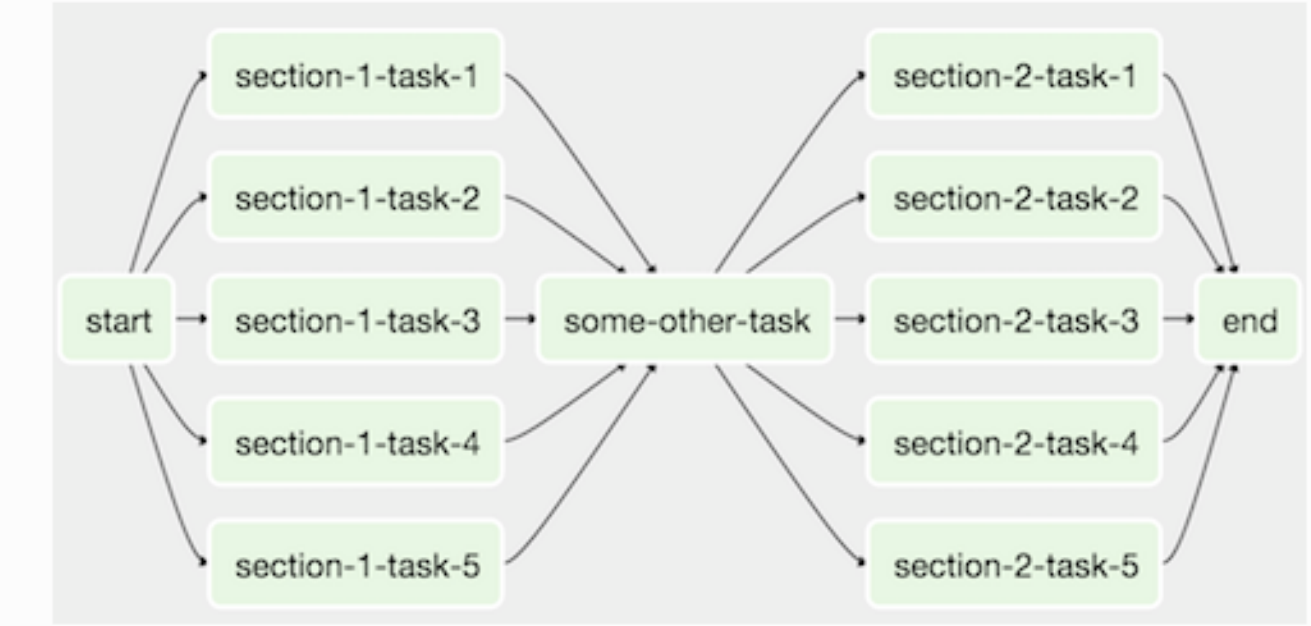


Branching

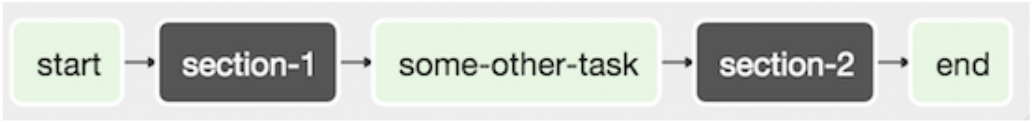
- Sometimes you need a workflow to branch, or only go down a certain path based on an arbitrary condition which is typically related to something that happened in an upstream task.
- The BranchPythonOperator is much like the PythonOperator except that it expects a python_callable that returns a task_id (or list of task_ids). The task_id returned is followed, and all of the other paths are skipped.

```
def branch_func(**kwargs):
    ti = kwargs['ti’]
    xcom_value = int(ti.xcom_pull(task_ids='start_task’))
    if xcom_value >= 5:
        return 'continue_task’
    else:
        return 'stop_task’
start_op = BashOperator( task_id='start_task',
bash_command="echo 5", xcom_push=True, dag=dag)
branch_op = BranchPythonOperator(
task_id='branch_task', provide_context=True,
python_callable=branch_func, dag=dag)
continue_op = DummyOperator(task_id='continue_task',
dag=dag)
stop_op = DummyOperator(task_id='stop_task', dag=dag)
start_op >> branch_op >> [continue_op, stop_op]
```

SubDAG (sub_dags.py)



- SubDAGs are perfect for repeating patterns. Defining a function that returns a DAG object is a nice design pattern when using Airflow.




```
Section1=SubDagOperator(
   task_id='section-1',
   subdag=sub_dag(PARENT_DAG_NAME, 'section-1',
            datetime.datetime(2019,11,22), '@daily'),
   dag=main_dag)
start >> section_1 >> some_other_task >> section_2 >> end
```

Xcom

- In general, if two operators need to share information, like a filename or small amount of data, you should consider combining them into a single operator. If it absolutely can’t be avoided, Airflow does have a feature for operator cross-communication called Xcom

- Push

Tasks can push XComs at any time by calling the xcom_push()method. In addition, if a task returns a value then an XCom containing that value is automatically pushed.

```
def push(**kwargs):
    kwargs['ti'].xcom_push(key='value from pusher 1’,value=value_1)
def push_by_returning(**kwargs):
    return value_2
```

- Pull

Tasks call xcom_pull() to retrieve XComs, optionally applying filters based on criteria like key, source task_ids, and source dag_id

```
def puller(**kwargs):
    ti = kwargs['ti‘]
    v1 = ti.xcom_pull(key=None, task_ids='push')
```