

Dynamic DAGs – The New Horizon

...

Ash Berlin-Taylor

Who is this Ash character anyway?



PMC member of
Apache Airflow project;
ASF Member

ASTRONOMER

Director of Airflow Engineering,
Astronomer.io



Have your say:
<https://bit.ly/AirflowSurvey22>



"Dynamic" DAGs in the past

Nothing good, all hacky in some way

The Airflow you know, but more powerful

Do what you want, natively

Architecture

How we built it, and its limitations



Hacks on top of hacks; or what we used to do



Create n tasks in your DAG

Might underuse slots when cluster idle

Have to pick an arbitrary "fixed" parallelism

```
def create_tasks(dag):
    for i in range(1, 100):
        task = Task(task_id=f'task_{i}',
                    func=lambda i=i: print(f'Task {i}'))
        dag.add_task(task)
```

Create *n* tasks in
your DAG

```
for i in range(NUM_PARALLEL_FILE_LOADERS):
    from_position = partial(get_from_position, i)

    download_task = MyDownloadFromS3Operator(
        task_id=f"download_file_{i}",
        xcom_callable=from_position,
```

Create n tasks in
your DAG



Dynamically at parse time

Easy to get wrong (make sure you cache locally to avoid slow dag file parsing!)

Only for slowly changing values

```

@task()
def update_experiment_list():
    s3_hook = CustomS3Hook(aws_conn_id='s3_default')
    ...
    Variable.set(key=experiment_var_name, value=experiments, serialize_json=True)

update_experiment_list_task= update_experiment_list()

ab_test_model_output_prefix= 'redshift/reporting/ab_testing.modeled_experiment_data'
experiment_name= None

with TaskGroup('run_model') as run_models:
    for experiment_name in Variable.get(key=experiment_var_name, default_val=[], deserialize_json=True):
        run_single_model= AbTestingModelOperator(
            task_id=experiment_name or 'none',
            output_key=f'{ab_test_model_output_prefix}/{experiment_name}.parquet',
            experiment_name=experiment_name,
        )
    if not experiment_name:
        run_single_model= DummyOperator(task_id='none')

redshift_load= RedshiftCopyOperator(
    database='reporting',
    schema='ab_testing',
    table='modeled_experiment_data',
    column_list=[
        Column('draw', 'BIGINT'),
        Column('variation', 'VARCHAR(255)'),
        Column('value', 'DOUBLE PRECISION'),
        Column('param', 'VARCHAR(255)'),
        Column('experiment', 'VARCHAR(255)'),
    ],
    bucket=s3_buckets.analytics_integration,
    key=f'{ab_test_model_output_prefix}/',
    copy_options=['PARQUET'],
    truncate_table=True,
)

chain_tasks(
    update_experiment_list_task,
    run_models,
    redshift_load,
)

```

"Dynamically" at parse time

```

@task()
def update_experiment_list():
    s3_hook = CustomS3Hook(aws_conn_id='s3_default')
    ...
    Variable.set(key=experiment_var_name, value=experiments, serialize_json=True)

update_experiment_list_task= update_experiment_list()

ab_test_model_output_prefix= 'redshift/reporting/ab_testing.modeled_experiment_data'
experiment_name= None
with TaskGroup('run_model') as run_models:
    for experiment_name in Variable.get(key=experiment_var_name, default_value=None, deserialize_json=True):
        run_single_model= AbTestingModelOperator(
            task_id=experiment_name or 'none',
            output_key=f"{ab_test_model_output_prefix}/{experiment_name}.parquet",
            experiment_name=experiment_name,
        )
    if not experiment_name:
        run_single_model= DummyOperator(task_id='none')

redshift_load= RedshiftCopyOperator(
    database='reporting',
    schema='ab_testing',
    table='modeled_experiment_data',
    column_list=[
        Column('draw', 'BIGINT'),
        Column('variation', 'VARCHAR(255)'),
        Column('value', 'DOUBLE PRECISION'),
        Column('param', 'VARCHAR(255)'),
        Column('experiment', 'VARCHAR(255)'),
    ],
    bucket=s3_buckets.analytics_integration,
    key=f"{ab_test_model_output_prefix}/",
    copy_options=['PARQUET'],
    truncate_table=True,
)

chain_tasks(
    update_experiment_list_task,
    run_models,
    redshift_load,
)

```

```

for experiment_name in Variable.get(key=experiment_var_name, deserialize_json=True):
    run_single_model = AbTestingModelOperator(
        task_id=experiment_name or 'none',

```

"Dynamically" at
parse time



TriggerDagRunOperator + sensor

Hard to view overall status at a glance



Just live with it being in one task

Slow

Not practical for high cardinality



Parallelism in external systems

(eg Spark) - can be overkill

\$\$\$

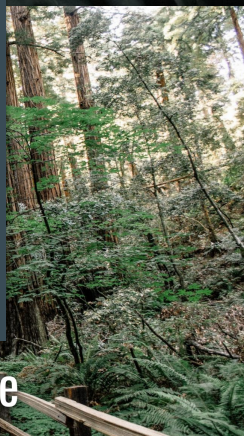


Your DAGs can now dynamically size themselves to fit your data!

Apache Airflow 2.2 and beyond

Airflow Summit 2021

Roadmap: A possible future



```
tor.partial(  
    )
```

```
    ).map(key=my_files)
```

```
data = ingest.map(markets)  
rois = calculate_roi.map(market, data)  
stats = aggregate_rois(market, rois)
```

Airflow Summit 2021

Airflow Summit 2022

Apache Airflow beyond

Roadmap

```
@task
def get_files_from_s3():
    return [...]
```

```
my_files = get_files_from_s3()
s3_delete_files = MyFileProcessOperator.partial(
    aws_conn_id="my-aws-conn-id",
    bucket="my-bucket"
).map(key=my_files)
```

```
data = ingest.map(markets)
rois = calculate_roi.map(market, data)
stats = aggregate_rois(market, rois)
```

Airflow Summit 2021

```
Operator.expand(arg_name=iterable, ...)
```

```
Operator.partial(fixed_arg=...)
    .expand(arg_name=iterable, ...)
```



CSV files \Rightarrow line-delimited JSON files


```
@task
def get_inputs(input_bucket, prefix):
    return S3Hook().list_keys(bucket_name=input_bucket, prefix=prefix)
```

```
@task
def csv_to_json(input_bucket, key, output_bucket):
    hook = S3Hook()
    output = io.BytesIO()

    csv_data = hook.read_key(key, input_bucket)
    reader = csv.DictReader(io.StringIO(csv_data))

    for row in reader:
        output.write(json.dumps(row, indent=None).encode('utf-8') + b'\n')

    output.seek(0)
    hook.load_file_obj(output, key=key, bucket_name=output_bucket)
```

```
files = get_inputs(input_bucket=input_bucket, prefix="data_provider_a/{{ data_interval_end | ds }}/")
```

```
csv_to_json.partial(input_bucket=input_bucket, output_bucket="airflow-summit-2022-processed") \
    .expand(key=files)
```

```
@task
def get_inputs(input_bucket, prefix):
    return S3Hook().list_keys(bucket_name=input_bucket, prefix=prefix)

@task
def csv_to_json(input_bucket, key, output_bucket):
    hook = S3Hook()
    output = io.BytesIO()

    csv_data = hook.read_key(key, input_bucket)
    reader = csv.DictReader(io.StringIO(csv_data))

    for row in reader:
        output.write(json.dumps(row, indent=None).encode('utf-8') + b'\n')

    output.seek(0)
    hook.load_file_obj(output, key=key, bucket_name=output_bucket)

files = get_inputs(input_bucket=input_bucket, prefix="data_provider_a/{{ data_interval_end | ds }}/")

csv_to_json.partial(input_bucket=input_bucket, output_bucket="airflow-summit-2022-processed") \
    .expand(key=files)
```

**Airflow 2.3 is vastly more
powerful and expressive
than Airflow 1.x**

Auto-refresh
DAG **dynamic_task_mapping_7** / Run **2022-05-19, 01:59:31 UTC** Task **mapped []** Hide Details Panel

List Instances, all runs Filter Upstream

Task Actions for all mapped tasks

128 Tasks Mapped
Overall Status: ■ success
■ success: 128

Task Id: [mapped \[\]](#)
 Run Id: [manual_2022-05-19T01:59:31.326095+00:00](#)
 Operator: PythonOperator

Overall Duration: 00:04:29
 Started: 2022-05-19, 01:59:34 UTC
 Ended: 2022-05-19, 02:04:04 UTC

Mapped Instances

	MAP INDEX	STATE	DURATION	START DATE	END DATE	
<input type="checkbox"/>	0	■ success	00:00:00	2022-05-19, 01:59:34 UTC	2022-05-19, 01:59:35 UTC	▲ <> ≡ ↔
<input type="checkbox"/>	1	■ success	00:00:00	2022-05-19, 01:59:36 UTC	2022-05-19, 01:59:37 UTC	▲ <> ≡ ↔
<input type="checkbox"/>	2	■ success	00:00:00	2022-05-19, 01:59:38 UTC	2022-05-19, 01:59:39 UTC	▲ <> ≡ ↔
<input type="checkbox"/>	3	■ success	00:00:00	2022-05-19, 01:59:41 UTC	2022-05-19, 01:59:41 UTC	▲ <> ≡ ↔
<input type="checkbox"/>	4	■ success	00:00:00	2022-05-19, 01:59:43 UTC	2022-05-19, 01:59:43 UTC	▲ <> ≡ ↔

@bbovenzi's amazing new UI

A person stands in the center of a vast, snow-covered mountain valley at night. The scene is illuminated by a single flashlight beam from the person, casting a bright glow on the snow and the surrounding rock faces. The sky is dark with a few stars visible. The overall atmosphere is cold and isolated.

**Toy example, real effect:
cross product**

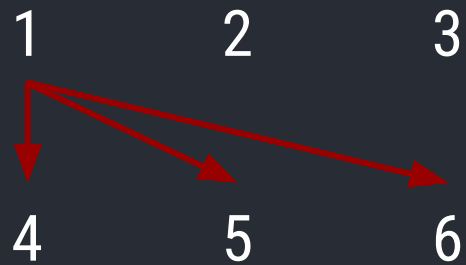
```
with DAG(dag_id='toy', start_date=datetime(2022, 5, 23)) as dag:
    @task
    def a():
        return [1,2,3]

    @task
    def b():
        return [4,5,6]

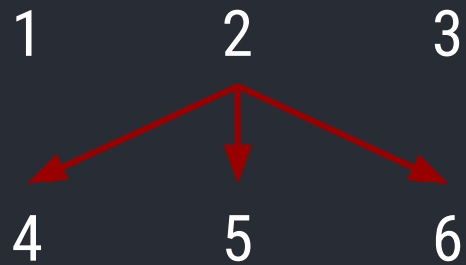
    @task
    def sum_it(vals):
        return sum(vals)

    @task
    def multiply(a, b):
        return a * b


combinations = multiply.expand(a=a(), b=b())
total = sum_it(combinations)
```



(a=1, b=4) (a=1, b=5) (a=1, b=6)



$(a=1, b=4)$ $(a=1, b=5)$ $(a=1, b=6)$ $(a=2, b=4)$ $(a=2, b=5)$ $(a=2, b=6)$ $(a=3, b=4) \dots$



**What? Where?
When? How?**

A 3D illustration of a wooden lattice structure, possibly a roof or a decorative element, rendered in a warm, golden-brown color. The structure consists of multiple vertical posts and horizontal beams that intersect to form a grid of diamond-shaped openings. The perspective is from a low angle, looking up at the structure. The background is a solid, muted blue color.

What?

List or a dict

Appearing directly in the dag file (or not as the result of a Task)

Result of a TaskFlow operator

"Generating"/Upstream Task must return a list or a dict, else it will fail

XComArg object created for "Classic" operator

XComArg(files) – yes, this is a bit clunky


```
@task
def cmds():
    return ["cmd1", "cmd2"]
```

```
BashOperator.expand(bash_command=["echo", cmds()]) # Wrong
```

```
@task
def cmds():
    return [["echo", "cmd1"], ["echo", "cmd2"]]
```

```
BashOperator.expand(bash_command=cmds()) # Only at top level
```

```
@task
def cmds():
    return ["cmd1", "cmd2"]

BashOperator.expand(bash_command=cmds().map(
    lambda v: ["echo", v])
)
```

Sneak peak - 2.4?



Where?

Usually in the Task Runner

Mapped TaskInstances are created in "mini scheduler" as upstreams finish

But fallback/"last resort" in Scheduler

"Static" expansions are expanded at DagRun creation

i.e. lists/dicts

A misty forest scene with a path leading up a hill, overlaid with the text "When?". The image is dark and atmospheric, with a path of stone steps leading up a hill in the foreground. The trees are dense and their trunks are visible through the mist. The overall tone is somber and mysterious.

When?

"Just in time"

Mapped TaskInstances are created as upstreams finish –
in worker (or scheduler)

An aerial photograph of a white commercial airplane flying through a dense, lush green forest. The plane is positioned in the center of the frame, moving from the bottom towards the top. The forest is composed of many tall, coniferous trees, creating a textured, green background. The word "How?" is written in large, white, sans-serif font across the middle of the image, partially overlapping the plane and the forest.

How?

Goal: Try to error at parse time, not run time

Make all errors as visible as early as possible

Goal: Small performance impact

Especially when if you aren't using mapped tasks

Goal: To feel pythonic



TaskMap table

Scheduler shouldn't pull (possibly large!) XCom rows, so pre-compute what we need

Add map_index to TaskInstance primary key

And all related tables (Fail, Reschedule etc). -1 = not mapped

(Configurable) limit on max number of expansions

Mostly just to prevent mistakenly creating millions of mapped Tasks

+16272 -6127

85 PRs from 7 authors (@ashb, @bbovenzi, @dstandish, @ephraimbuddy, @norm,
@tanelk, @uranusjr)

Special thanks to @MatrixManAtYrService for breaking it so often



Still more to add...

Airflow 2.4 and beyond

- "zip" (rather than cross-product) [#23803](#)
 - Expand TaskGroups (postponed in 2.3)
 - New expansion sources (Variables, DagParams)
 - Multiple args from a single source (**kwargs style)
-

We're hiring (of course)
<https://www.astronomer.io/careers>

ASTRONOMER