May 23–27, 2022

AIRFLOW SUMMIT
Apache Airflow at Scale

John Jackson
Principal Product Manager
Amazon Web Services (AWS)
What We’ll Cover Today

• Introduction
• What is Apache Airflow at Scale
  • Understanding Considerations
  • Scheduler Loops and Configurations
• Scaling Workloads
  • Containers
  • Pools and priority
• Scaling DAGs
  • Dynamic DAGs/DAG Factories
  • CI/CD
  • DAG Access Control
• Multiple Environments
  • How to split up workloads (users/downstream access/priority)
  • Central Governance: Creation and Monitoring
  • Example: Distributing workloads across Airflow clusters
• Q&A
Introduction

John Jackson

- Product Manager for Amazon Managed Workflows for Apache Airflow (MWAA)
- 2+ years with Amazon Web Services
- Part of the Airflow Summit 2022 Organizing Committee
- Software Developer/Solution Architect/Product Manager for over 25 years
- Based in Vancouver, Canada
- https://github.com/john-jac
What is Apache Airflow at Scale?

- Large number of workflow definitions
- Large or complicated workflows
- Long running tasks
- Large number of concurrent tasks

DAGs

- All 112
- Active 112
- Paused 0
Considerations

What things affect your ability to scale

DAGs are parsed continuously, whether active or not

DAG objects are analyzed by the Scheduler to see which tasks should be queued next

Typically there is a fixed amount of compute for these operations (number ofSchedulers, Workers, Web Servers, plus DB size, network capacity, etc.)
Scheduler Loops and Configurations

1. Check for new files
2. Exclude recently processed files
3. Queue file paths
4. Process files
5. Collect results
6. Log statistics (ie Total Parse Time)
Scheduler Loops and Configurations

DagFileProcessorManager:
1. Check for new files
2. Exclude recently processed files
3. Queue file paths
4. Process files
5. Collect results
6. Log statistics (ie Total Parse Time)

DagFileProcessorProcess:
1. Process file
2. Load modules from file
3. Process modules
4. Return DagBag
Scheduler Loops and Configurations

TaskProcessor:
- Load Serialized DagBag
- Queue Dag Runs
- Schedule DAG run, check DAG structure for change, dependencies
- Queue/Execute tasks instances
- Clean up completed/failed tasks
- Emit Heartbeat
- Run timed events (ie SLA check)

DagFileProcessorManager:
- Check for new files
- Exclude recently processed files
- Queue file paths
- Process files
- Collect results
- Log statistics (ie Total Parse Time)

DagFileProcessorProcess:
- Process file
- Load modules from file
- Process modules
- Return DagBag
Scheduler Loops and Configurations

TaskProcessor
- Load Serialized DagBag
- Queue Dag Runs
- Schedule DAG run, check DAG structure for change, dependencies
- Queue/Execute tasks instances
- Clean up completed/failed tasks
- Emit Heartbeat
- Run timed events (ie SLA check)

DagFileProcessorManager
- Check for new files
- Exclude recently processed files
- Queue file paths
- Process files
- Collect results
- Log statistics (ie Total Parse Time)

DagFileProcessorProcess
- Process file
- Load modules from file
- Process modules
- Return DagBag

Executor
- Sends Message over Broker for worker to pick up
- Updates state based on worker response
Scheduler Loops and Configurations

TaskProcessor
- Load Serialized DagBag
- Queue Dag Runs
- Schedule DAG run, check DAG structure for change, dependencies
- Queue/Execute tasks instances
- Clean up completed/failed tasks
- Emit Heartbeat
- Run timed events (ie SLA check)

DagFileProcessorManager
- Check for new files
- Exclude recently processed files
- Queue file paths
- Process files
- Collect results
- Log statistics (ie Total Parse Time)

DagFileProcessorProcess
- Process file
- Load modules from file
- Process modules
- Return DagBag

Executor
- Sends Message over Broker for worker to pick up
- Updates state based on worker response
Configuration Options
Some key options when running at scale

**dag_dir_list_interval** - How often to scan the DAGs directory for new files. Default is 5 minutes (300 seconds).

**min_file_process_interval** - Number of seconds after which a DAG file is re-parsed. The DAG file is parsed every min_file_process_interval number of seconds. Default is 30 seconds.

**parsing_processes** - The scheduler can run multiple processes in parallel to parse DAG files. This defines how many processes will run. Default is 2.

**dag_file_processor_timeout** - How long before timing out the processing of a dag file. Default is 50 seconds.

**dagbag_import_timeout** - How long before timing out a python file import. Default is 30 seconds.

Scaling Workloads
Getting the most work out of your Airflow cluster
Containers

Offloading work

- Using Kubernetes, Docker, ECS, EKS, EMR, Batch, etc Operators

- Using Airflow as an Orchestrator, not for doing the actual processing

- Similar philosophy around ETL—use Airflow to orchestrate the overall ETL set of jobs, but use a dedicated ETL service (Spark, Kafka, Hive) to execute the actual ETL or ELT (Snowflake, Redshift, other analytics databases) to perform the transform after load
Pools, Priority, and Parallelism
Control which tasks run, when, and how many

- **Airflow pools** can be used to limit the execution parallelism on arbitrary sets of tasks. Typically this is done to limit downstream impact, for example putting all database tasks in an “RDS” pool that has a limit based upon the connection limit of the DB.

- The **priority_weight** of a task defines priorities in the executor queue. In a given pool, as slots free up, queued tasks start running based on the Priority Weights of the task and its descendants.

- **Parallelism** at the system level defines the maximum number of task instances that can run concurrently in Airflow regardless of scheduler count and worker count. Generally, this value is reflective of the number of task instances with the running state in the metadata database.

- **Concurrency** is the maximum number of task instances allowed to run concurrently in each DAG, and is configurable at the DAG level with `max_active_tasks`, which is defaulted as `max_active_tasks_per_dag`.

- A **deferrable operator** is able to suspend itself and free up the worker when it knows it has to wait, and hand off the job of resuming it to a Trigger.
Scaling DAGs
More workflows, less code
Dynamic DAGs/DAG Factories

DAG Factory Configuration → Python file in DAG folder → DAG object → DAG object → DAG object → DAG object → DAG object → DAG object
Dynamic DAG Example 1

```python
@dag(
    dag_id=f"{DAG_ID}_listing",
    schedule_interval="@hourly",
    start_date=datetime(2022, 1, 1),
    catchup=False,
)
def update_table_listing():
    t = PostgresOperator(task_id="query_listing",
                         sql="select * from listing;",
                         postgres_conn_id=POSTGRES_CONN_ID)
update_table_listing_dag = update_table_listing()

@dag(
    dag_id=f"{DAG_ID}_sales",
    schedule_interval="@hourly",
    start_date=datetime(2022, 1, 1),
    catchup=False,
)
def update_table_sales():
    t = PostgresOperator(task_id="query_sales",
                         sql="select * from sales;",
                         postgres_conn_id=POSTGRES_CONN_ID)
update_table_sales_dag = update_table_sales()

@dag(
    dag_id=f"{DAG_ID}_accounts",
    schedule_interval="@hourly",
    start_date=datetime(2022, 1, 1),
    catchup=False,
)
def update_table_accounts():
    t = PostgresOperator(task_id="query_accounts",
                         sql="select * from accounts;",
                         postgres_conn_id=POSTGRES_CONN_ID)
update_table_accounts_dag = update_table_accounts()
```
def get_sources():
    pg_request = "SELECT * FROM information_schema.tables \\n    WHERE table_schema = 'public'"
    pg_hook = PostgresHook(postgres_conn_id=POSTGRES_CONN_ID, schema="dev")
    connection = pg_hook.get_conn()
    cursor = connection.cursor()
    cursor.execute(pg_request)
    sources = cursor.fetchall()
    for source in sources:
        print("Source: {0}".format(source))
    return sources

SOURCES = get_sources()

for source in SOURCES:
    dag_id=f"{DAG_ID}_{source[2]}"
    @dag(
        dag_id=dag_id,
        schedule_interval="@hourly",
        start_date=datetime(2022, 1, 1),
        catchup=False,
    )
    def update_table_dag(sql=""):
        t = PostgresOperator(task_id="query_table",
            sql=f"select * from {source[2]}",
            postgres_conn_id=POSTGRES_CONN_ID)
        globals()[dag_id] = update_table_dag()

@dag(
    dag_id=f"{DAG_ID}_listing",
    schedule_interval="@hourly",
    start_date=datetime(2022, 1, 1),
    catchup=False,
)
def update_table_listing():
    t = PostgresOperator(task_id="query_listing",
        sql="select * from listing;",
        postgres_conn_id=POSTGRES_CONN_ID)
update_table_listing_dag = update_table_listing()

@dag(
    dag_id=f"{DAG_ID}_sales",
    schedule_interval="@hourly",
    start_date=datetime(2022, 1, 1),
    catchup=False,
)
def update_table_sales():
    t = PostgresOperator(task_id="query_sales",
        sql="select * from sales;",
        postgres_conn_id=POSTGRES_CONN_ID)
update_table_sales_dag = update_table_sales()

@dag(
    dag_id=f"{DAG_ID}_accounts",
    schedule_interval="@hourly",
    start_date=datetime(2022, 1, 1),
    catchup=False,
)
def update_table_accounts():
    t = PostgresOperator(task_id="query_accounts",
        sql="select * from accounts;",
        postgres_conn_id=POSTGRES_CONN_ID)
update_table_accounts_dag = update_table_accounts()
Dynamic DAG Example 2
Less Parsing Overhead

```python
@task()
def get_sources():
    pg_request = "SELECT * FROM information_schema.tables \n    WHERE table_schema = 'public'"
    pg_hook = PostgresHook(
        postgres_conn_id=POSTGRES_CONN_ID,schema="dev")
    connection = pg_hook.get_conn()
    cursor = connection.cursor()
    cursor.execute(pg_request)
    sources = cursor.fetchall()
    jsonStr = json.dumps(sources)
    with open(TABLE_LIST_FILE_PATH, 'w') as f:
        f.write(jsonStr)
    return sources

dag(
    dag_id=f"{DAG_ID}_{source[2]}",
    schedule_interval="0 * * * *",
    start_date=datetime(2022, 1, 1),
    catchup=False,
)
def update_table_dag(sql=""):  
t = PostgresOperator(task_id="query_table",
    sql=f"select * from {source[2]}",
    postgres_conn_id=POSTGRES_CONN_ID)
globals()[dag_id] = update_table_dag()
```

with open(TABLE_LIST_FILE_PATH) as f:
    jsonStr = f.readlines()
    sources = json.loads(jsonStr)
    for source in sources:
        dag_id=f"{DAG_ID}_{source[2]}"
        @dag(
            dag_id=dag_id,
            schedule_interval="0 * * * *",
            start_date=datetime(2022, 1, 1),
            catchup=False,
        )
        def update_table_dag(sql=""):  
t = PostgresOperator(task_id="query_table",
    sql=f"select * from {source[2]}",
    postgres_conn_id=POSTGRES_CONN_ID)
globals()[dag_id] = update_table_dag()
```
Dynamic DAG Example 2
Less Parsing Overhead

```python
TABLE_LIST_FILE_PATH="/usr/local/airflow/dags"

@task()
def get_sources():
    pg_request = "SELECT * FROM information_schema.tables \n    WHERE table_schema = 'public'"
    pg_hook = PostgresHook( 
        postgres_conn_id=POSTGRES_CONN_ID,schema="dev")
    connection = pg_hook.get_conn()
    cursor = connection.cursor()
    cursor.execute(pg_request)
    sources = cursor.fetchall()
    jsonStr = json.dumps(sources)
    with open(TABLE_LIST_FILE_PATH, 'w') as f:
        f.write(jsonStr)
    return sources

@dag(
    dag_id="{DAG_ID}_get_sources",
    schedule_interval="55 * * * *",
    start_date=datetime(2022, 1, 1),
    catchup=False,
)
def update_table_list():
    TI = get_sources()
    update_table_list_dag = update_table_list_dag()
```

```python
with open(TABLE_LIST_FILE_PATH) as f:
    jsonStr = f.readlines()
    sources = json.loads(jsonStr)
    for source in sources:
        dag_id="{DAG_ID}_{{source[2]}}"
        @dag(
            dag_id=dag_id,
            schedule_interval="0 * * * *",
            start_date=datetime(2022, 1, 1),
            catchup=False,
        )
        def update_table_dag(sql):
            t = PostgresOperator(task_id="query_table",
                sql="select * from {{source[2]}},
                postgres_conn_id=POSTGRES_CONN_ID)
            globals()[dag_id] = update_table_dag()
```

```
Dynamic DAG Example 2
Less Parsing Overhead

TABLE_LIST_FILE_PATH="/usr/local/airflow/dags"

@task()
def get_sources():
    pg_request = "SELECT * FROM information_schema.tables \n    WHERE table_schema = 'public'"
    pg_hook = PostgresHook( 
        postgres_conn_id=POSTGRES_CONN_ID,schema="dev")
    connection = pg_hook.get_conn()
    cursor = connection.cursor()
    cursor.execute(pg_request)
    sources = cursor.fetchall()
    jsonStr = json.dumps(sources)
    with open(TABLE_LIST_FILE_PATH, 'w') as f:
        f.write(jsonStr)
    return sources

@dag(
    dag_id="{DAG_ID}_get_sources",
    schedule_interval="55 * * * *",
    start_date=datetime(2022, 1, 1),
    catchup=False,
)
def update_table_list():
    TI = get_sources()
    update_table_list_dag = update_table_list_dag()
```
Dynamic DAG Example 2
Less Parsing Overhead

```python
@task()
def get_sources():
    pg_request = "SELECT * FROM information_schema.tables \n    WHERE table_schema = 'public'"

    pg_hook = PostgresHook(  
        postgres_conn_id=POSTGRES_CONN_ID,schema="dev")
    connection = pg_hook.get_conn()  
    cursor = connection.cursor()  
    cursor.execute(pg_request)  
    sources = cursor.fetchall()  
    jsonStr = json.dumps(sources)  
    with open(TABLE_LIST_FILE_PATH, 'w') as f:
        f.write(jsonStr)

    return sources

@dag(
    dag_id="{DAG_ID}_{source[2]}"
    schedule_interval="0 * * * *",
    start_date=datetime(2022, 1, 1),
    catchup=False,
)
def update_table_dag(sql):
    t = PostgresOperator(task_id="query_table",
        sql="select * from {source[2]}",
        postgres_conn_id=POSTGRES_CONN_ID)
    return t

def update_table_list():
    ti = get_sources()
    update_table_list_dag = update_table_list_dag()
```
CI/CD
Continuous Integration and Deployment

• Add controls to verify integrity, scope, and usage of DAGs before deploying

• Update configurations

• Test using staging environment

• Automate anything that has to be done more than once
DAG Access Control
"Cooperative Multitenancy"

- Using multiple RBAC roles
- Airflow is not (yet) multi-tenant (AIP-1)
- DAG Factory can limit what a DAG can do
- Multiple clusters may be a better alternative for true multitenancy
Multiple Airflow Clusters
Level-up your Isolation and Resilience
Splitting Workloads Across Environments

How to decide what runs where

Stage

Airflow Downstream Access

Cross-environment

User Groups/Teams

Workload Priority/Schedule/Pattern
Centralized Creation and Monitoring

Create
- Terraform, Helm, CloudFormation, GitLab, Kubernetes, CDK, Docker Compose, ...

Log
- Datadog, S3, Prometheus, CloudWatch, ...

Monitor
- StatsD, Grafana, CloudWatch, DataDog, ...

Alarm
- SLAs, Callbacks, Email, Prometheus, EventBridge, ...

Airflow Summit 2022
A new CSV file is added to S3 that needs to be added to Redshift. The Destination DB executes transfer SQL from S3.
Distributing Dags Cross-Environment

Airflow scans files and creates one dag/task for each

A new CSV file is added to S3 that needs to be added to Redshift

Destination DB executes transfer SQL from S3
**Distributing Dags Cross-Environment**

Airflow reads from DB, and if within min and max ID will create DAG of SQL

Source DB stores ID, schedule, and list of SQL commands to run

Lambda triggered on add/remove

A new CSV file is added to S3 that needs to be added to Redshift

Destination DB executes transfer SQL from S3
Airflow creates JSON from DB, and creates DAGs/tasks from JSON.

Source DB stores ID, schedule, and list of SQL commands to run.

Lambda triggered on add/remove.

A new CSV file is added to S3 that needs to be added to Redshift.

Destination DB executes transfer SQL from S3.
Distributing Dags Cross-Environment

- Source DB stores ID, schedule, and list of SQL commands to run
- Lambda triggered on add/remove
- A new CSV file is added to S3 that needs to be added to Redshift
- Airflow reads from JSON, and if within min and max ID will create DAG of SQL
  - custom.range=1,400
  - custom.range=401,800
  - custom.range=801,1200
- Destination DB executes transfer SQL from S3
with open(TABLE_LIST_FILE_PATH) as f:
    jsonStr = f.readlines()
    sources = json.loads(jsonStr)

    range = os.getenv('AIRFLOW__CUSTOM__RANGE', default='0,0').split(',')
    min = int(range[0])
    max = int(range[1])

    for i in range(min, max+1):
        source = sources[i]
        dag_id=f"{DAG_ID}_{source[2]}"
        @dag(
            dag_id=dag_id,
            schedule_interval="0 * * * *",
            start_date=datetime(2022, 1, 1),
            catchup=False,
        )
        def update_table_dag(sql=""):
            t = PostgresOperator(task_id="query_table",
                sql=f"select * from {source[2]}",
                postgres_conn_id=POSTGRES_CONN_ID)
            globals()[dag_id] = update_table_dag()
Resources
For more information

• Airflow Slack Group: https://apache-airflow.slack.com/


• Airflow GitHub: https://github.com/apache/airflow

• AWS Blogs: https://aws.amazon.com/managed-workflows-for-apache-airflow/resources/

Thank you!