Apache Airflow

Post Reads - airflow1.2

Environment Setup - setup_airflow

Table of contents

Why this lecture?

- The traditional RDBMS systems, like MySQL, PostgreSQL etc., aren't optimised to run analytics-heavy queries.
- To support analytics-heavy queries in a distributed environment, we have already built a pipeline from MySQL to Apache HIVE. [Hopefully, you would have done that in previous lectures]

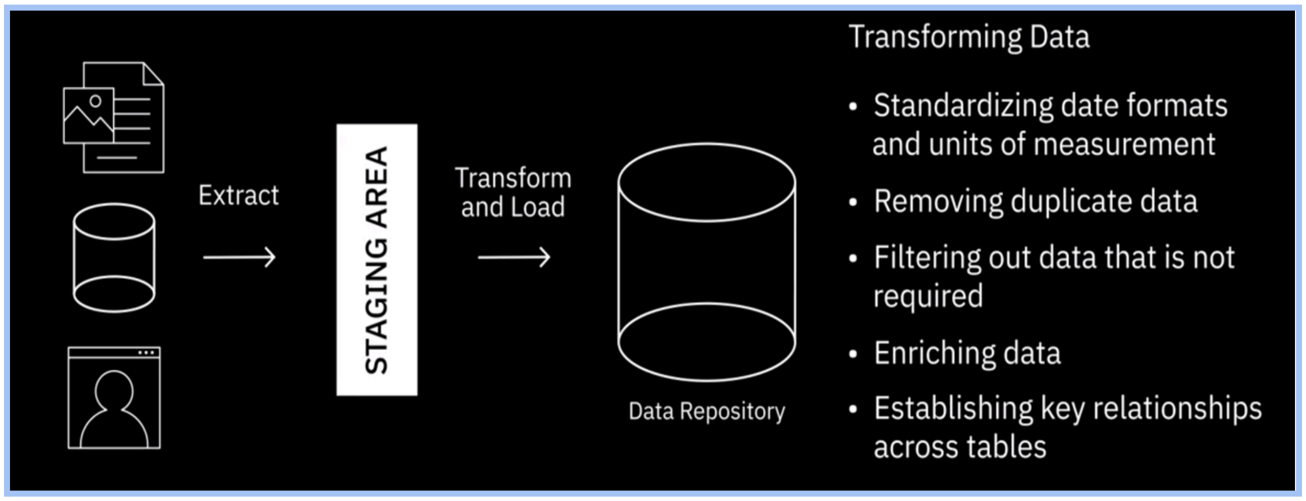
- Q - If we want the warehouse data to be updated every hour of the day, would it be feasible for us to run it manually?

- Ans - We can schedule those pipelines in any workflow orchestration platform for orchestrating distributed applications.

- But what is orchestration?
    a. Data orchestration is the process of taking siloed data from multiple data storage locations, combining and organising it, and making it available for data analysis tools.
    b. Data orchestration enables businesses to automate and streamline data-driven decision-making.

- **A few orchestration tools widely used in the industry are** - [details here]
    a. Apache Airflow (open-source, free)
    b. Luigi
    c. Apache NiFi
    d. Prefect
    e. AWS Step Functions
    f. Apache Oozie
    g. Dagster
    h. Kedro

- **Objective - To schedule/orchestrate a robust batch Data Pipeline {using Apache Airflow} that extracts, transforms and loads the data on an ongoing basis.**


Extract
STAGING AREA
Transform and Load
Data Repository
Transforming Data
• Standardizing date formats and units of measurement
• Removing duplicate data
• Filtering out data that is not required
• Enriching data
• Establishing key relationships across tables

- Why Airflow? Why not anything else?
    a. Airflow is versatile, expressive, and capable of creating highly complex workflows. Airflow is highly functional, and everything can be tracked from one GUI.
    b. Airflow has a **vast community** contributing to it, making it easy to find integration solutions for every major service/cloud provider.
    c. A key differentiating feature is templating to replace variables or expressions with values when a template is rendered.
    d. Features such as backfilling enable you to quickly (re)process historical data and recompute any derived data sets after making changes to your code.

Question - Let us say our pipeline is failing for the last 6 hours because of an error in the transformation logic. How do we ensure that we don't lose that data?

Ans - We can simply change the `start_time` of the pipeline, and it will also pick data for those hours. It is this simple in Airflow; otherwise, it would be tedious, and we might also lose data.

Avoid for Streaming pipelines: However, one should avoid using Airflow if the objective is, handling streaming pipelines. Airflow is primarily designed to run recurring or batch-oriented tasks rather than streaming workloads.

Problem Statement

- **For best recommendations, one of the services at Netflix fetches data (new movies, ratings, comments, etc.) from IMDb and stores it in their internal database. They do it every 45 minutes.**
- **The objective is to use this data to observe/analyse the best recommendations to their subscribers.**
- **To run analytics-heavy queries and train Machine Learning algorithms, they would need the data in warehouses instead of databases.**
- **ToDo: Build a pipeline to fetch data from the database and dump it into the warehouse.**
- **We will take MySQL as the database and HIVE as the warehouse in this lecture for illustrative purposes.**

Solution

We can write a cronjob for the same.

What is cron?

- The cron command-line utility is a job scheduler on Unix-like operating systems.
- Users who set up and maintain software environments use cron to schedule jobs (commands or shell scripts), also known as cron jobs, to run periodically at fixed times, dates, or intervals.
- General Example -
    - Fetch tweets with #scaler every hour
    - Send the newsletter at 10:30 AM daily
    - Figure out what was trending on Twitter yesterday!
- This is how we mention the schedule

```
# ┌───────────── minute (0 - 59)
# │ ┌───────────── hour (0 - 23)
# │ │ ┌───────────── day of the month (1 - 31)
# │ │ │ ┌───────────── month (1 - 12)
# │ │ │ │ ┌───────────── day of the week (0 - 6) (Sunday to Saturday;
# │ │ │ │ │                                   7 is also Sunday on some systems)
# │ │ │ │ │
# │ │ │ │ │
# * * * * * <command to execute>
```

How to build this pipeline using cron?

- We can write Python scripts to fetch data from MySQL and dump it into HIVE.
- We can schedule that script to run every hour on cron in any `ec2` instance.

There would be many limitations to this implementation.

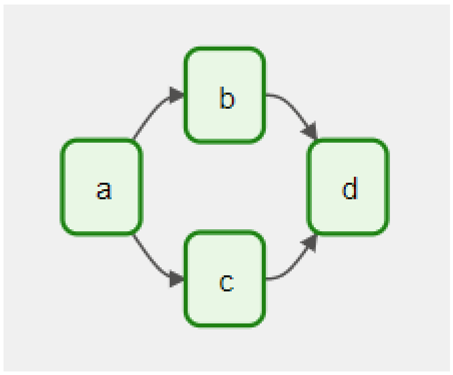
How to build this pipeline using Airflow?

Requirements -

- MySQL DB (source)
  - MySQL Connector Java
- HIVE (sink)
  - Hadoop Stack [HDFS, MapReduce]
- Airflow (orchestrator)
  - apache-airflow-providers-mysql
  - apache-airflow-providers-apache-hive
- Assumption - MySQL and HIVE both are correctly installed and working fine.
- **Link - Airflow setup**

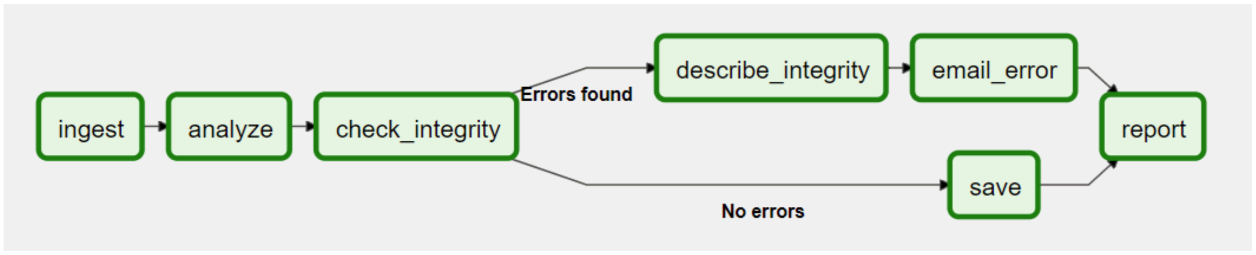
Steps:

- In Airflow, we can call a pipeline a Workflow and Airflow lets us build and run *workflows*.


$$\begin{tikzcd} & b \arrow[dr] & \\ a \arrow[ur] \arrow[dr] & & d \\ & c \arrow[ur] & \end{tikzcd}$$

- **Workflow** - It is represented as a *DAG (a Directed Acyclic Graph)* and contains individual pieces of work called *Tasks*, arranged with dependencies and data flows taken into account.
- **DAG (Directed Acyclic Graph)** -
  - It is the core concept of Airflow.

    - It collects Tasks together, organised with dependencies and relationships to say how they should run.
- In this image, we have a DAG:
  It defines four Tasks - A, B, C, and D - and dictates the order in which they have to run and which tasks depend on what others.
- It will also say how often to run the DAG - maybe "every 5 minutes starting tomorrow" or "every day since November 2 1st, 2022". We call it a **cron schedule.** An awesome place for all your cron queries - **crontab.guru**
- **But what is a task in itself?**
    - A Task is the most basic unit of execution in Airflow.
    - Tasks are arranged into DAGs, with upstream and downstream dependencies set between them to express the order they should run.
    - The Tasks describe what to do, be it fetching data, running analysis, triggering other systems, or more.
- Example DAG -



How to declare a DAG?

- The two main ways of declaring a DAG are:
    a. Using a **context manager**, which will implicitly add the DAG to anything inside it [Recommended]
    b. Using the **@dag decorator** to turn a function into a DAG generator

- The best and most widely used method is using a context manager -

```
with DAG(
    "my_dag_name",  # DAG name/identifier
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    schedule="@daily",  # cron format supported
    catchup=False
) as dag:
    task_name = OperatorName(task_id="task_id", key="val")
```

- **In case of multiple tasks, how can we define execution orders of the tasks?**
    - This is popularly known as Dependency/Relationship/Flow control among tasks.
    - A Task/Operator does not usually live alone; it depends on other tasks (those upstream of it), and other tasks depend on it (those downstream of it).
    - Declaring these dependencies between tasks is what makes up the DAG structure (the edges of the directed acyclic graph)
    - **Example**
        - Let's say a DAG has three tasks (extraction, transformation, loading). We can say, "Execute transformation only after extraction has executed, but loading can be independently executed at any time."
    - **We can also specify other constraints, such as timeout for a task, the number of retries to perform for a failing task, when to start a task, etc.**
    - **How do we declare individual task dependencies?**
        - use the >> and << operators [recommended]
        - or more explicitly use set_upstream and set_downstream methods:

```
first_task.set_downstream(second_task, third_task)
third_task.set_upstream(fourth_task)
```

        - The other two significant complex ways of defining dependencies are
            - cross_downstream dependency
            - Example - Let us say we have four tasks `t1, t2, t3, t4` and the requirement is t1 and t2 should execute before t3 and t4 and the order among `[t1, t2]` and `[t3, t4]` doesn’t matter. Then we can use something like this -

```
from airflow.models.baseoperator import cross_downstream

# Replaces
# [op1, op2] >> op3
# [op1, op2] >> op4
cross_downstream([op1, op2], [op3, op4])
```

            - chain dependency
            - Example - In case of lot many tasks, we can programmatically mention their execution order with the help of chain scheduling.

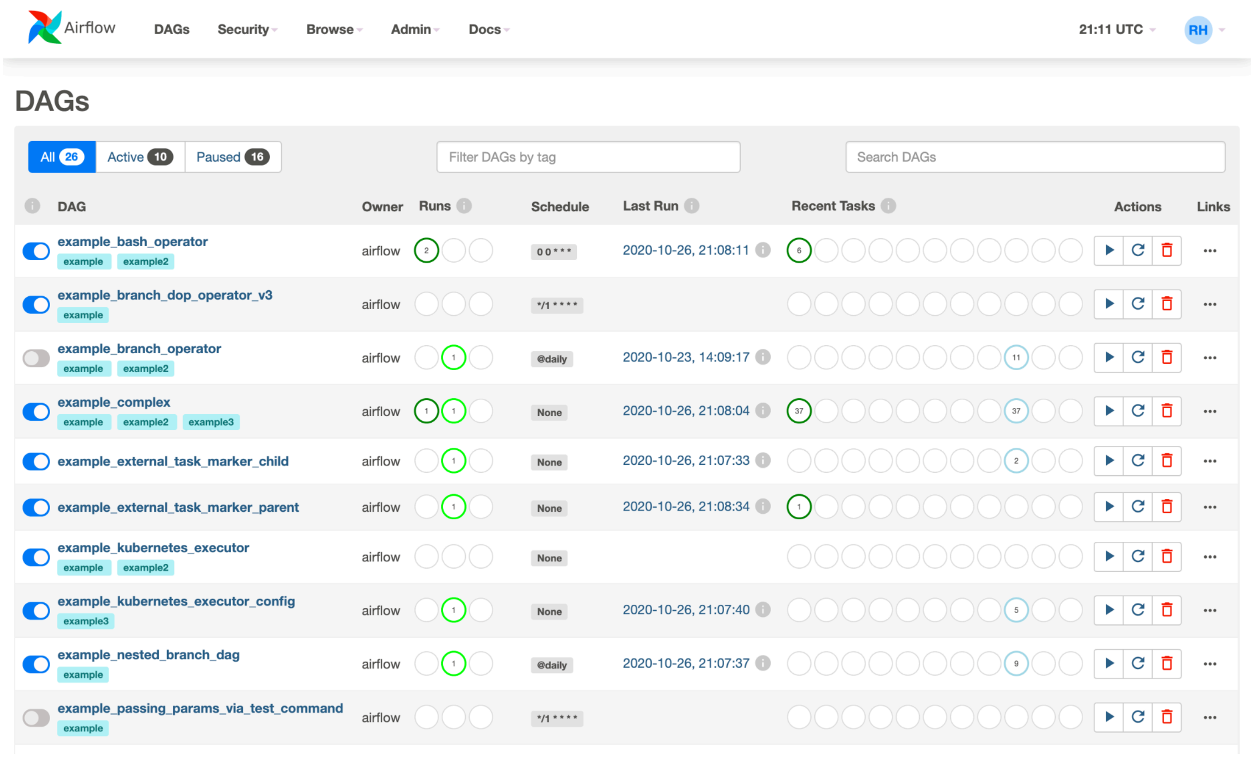
```python
from airflow.models.baseoperator import chain

# Replaces op1 >> op2 >> op3 >> op4
chain(op1, op2, op3, op4)

# You can also do it dynamically
chain(*[EmptyOperator(task_id='op' + i) for i in
range(1,5)])
```

Where can we see these DAGs?

User interface

Airflow has a user interface that lets you see

- What DAGs and their tasks are doing?
- Trigger runs of DAGs
- Logs, and do some limited debugging and resolution of problems with your DAGs



DAGs

DAG	Owner	Runs	Schedule	Last Run	Recent Tasks	Actions	Links
example_bash_operator example example2	airflow	2	0 0 * * *	2020-10-26, 21:08:11	6		...
example_branch_dop_operator_v3 example	airflow		*/1 * * * *				...
example_branch_operator example example2	airflow	1	@daily	2020-10-23, 14:09:17	11		...
example_complex example example2 example3	airflow	1 1	None	2020-10-26, 21:08:04	37 37		...
example_external_task_marker_child	airflow	1	None	2020-10-26, 21:07:33	2		...
example_external_task_marker_parent	airflow	1	None	2020-10-26, 21:08:34	1		...
example_kubernetes_executor example example2	airflow		None				...
example_kubernetes_executor_config example3	airflow	1	None	2020-10-26, 21:07:40	5		...
example_nested_branch_dag example	airflow	1	@daily	2020-10-26, 21:07:37	9		...
example_passing_params_via_test_command example	airflow		*/1 * * * *				...

How can we get this UI?

After proper installation, we can initialise a database to store the metadata (like DAG logs etc.) and then start the webserver and scheduler.
Here is how we do that -

```
airflow db init
```

```
airflow users create \
    --username admin \
    --firstname prashant \
    --lastname tiwari \
    --role Admin \
    --email prashant.tiwari_1@scaler.com
```

```
airflow webserver --port 8080
```

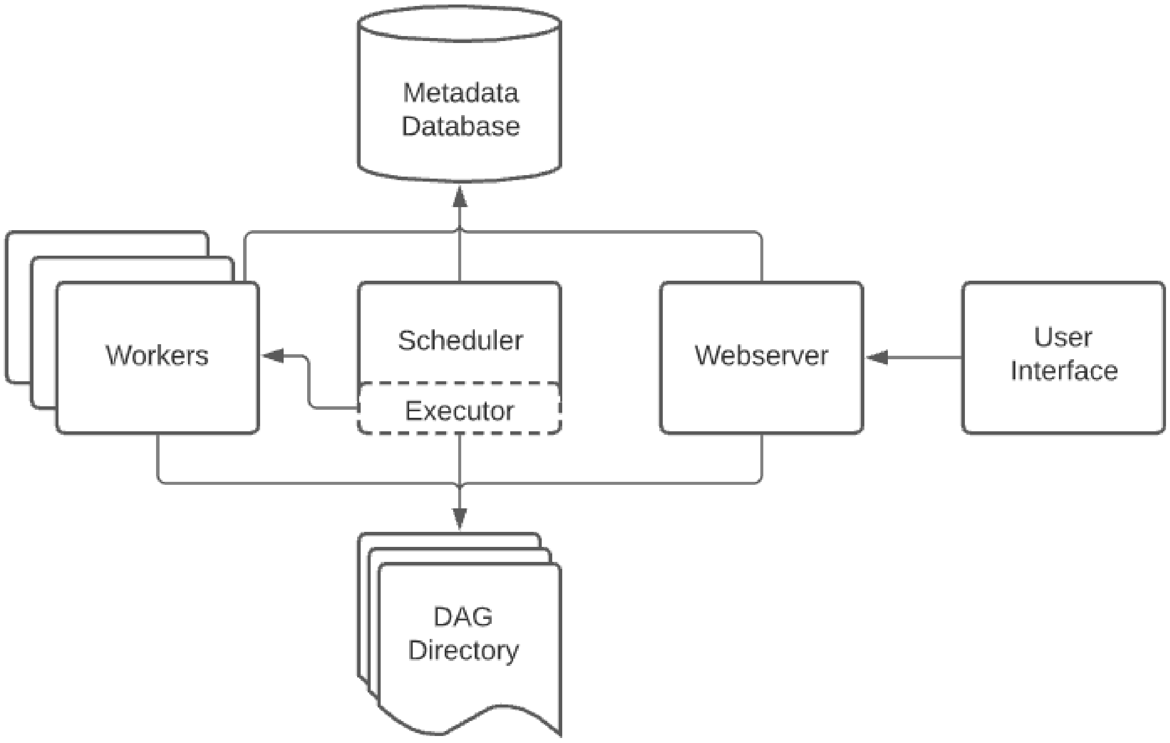
```
airflow scheduler
```

We can now head over to http://localhost:8080/ for the above shown Airflow UI.

How do these databases, webserver and scheduler interact?

[Very important for Interviews]
An Airflow installation generally consists of the following components:

- A scheduler, which handles both triggering scheduled workflows and submitting Tasks to the executor to run.
- An executor which handles running tasks. In the default Airflow installation, this runs everything inside the scheduler, but most production-suitable executors actually push task execution out to workers.
- A web server presents a handy user interface to inspect, trigger and debug the behaviour of DAGs and tasks.
- A folder of DAG files, read by the scheduler and executor (and any workers the executor has)
- A metadata database, used by the scheduler, executor and webserver to store state.
- This is what the interaction among these looks like



A quick recap!

What have we learnt till now?

1. Why is Airflow more helpful than other orchestration tools?
2. What are the general steps in building a pipeline?
   a. Workloads or DAGs
   b. Tasks
   c. Schedule/Cron
3. How do we declare a DAG?
   a. Basic Skeleton
   b. Task Dependencies
4. Internals of Airflow
   a. UI
   b. Architecture Overview

[Interview] Are we clear about the basic components of a pipeline and HOW

- [ ] It gets scheduled on a scheduler?
- [ ] Can be seen on the web server?
- [ ] Executed with the worker nodes?

How can we build the actual pipeline with these fundamental concepts/functionalities?

- Like we implement Classes and Methods while developing any API or backend microservice, we follow a similar fashion here.
- We club a class Blueprint and methods. Instead of the "**main"** method, we call those methods into a special function called "**execute"**.
- We give them a special term - **Operators**
    - **Operators are Python classes that encapsulate logic to do a unit of work. They can be viewed as a wrapper around each unit of work that defines the actions that will be completed and abstract the majority of code you would typically need to write.**
    - When you create an instance of an operator in a DAG and provide it with its required parameters, it becomes a task (typically only requires a few parameters)
    - All operators inherit from the abstract BaseOperator class, which contains the logic to execute the operator's work within the context of a DAG. The following are some of the most frequently used Airflow operators:
        - PythonOperator: Executes a Python function.
        - BashOperator: Executes a bash script.
    - If the operator you need isn't installed with Airflow by default, you can probably find it as part of our huge set of community provider packages. Some popular operators from here include
        - SimpleHttpOperator
        - MySqlOperator
        - PostgresOperator
        - JdbcOperator
        - DockerOperator
        - HiveOperator,
        - S3FileTransformOperator

- Already existing Operators are good for some standard use cases. For a use case that can fulfil our requirements, we need to write a **CustomAirflowOperator**.

Example 1 - Write a simple operator to onboard a pipeline from MySQL to HIVE.

Algorithm -

1. Connect to the MySQL table and return a Cursor
2. Execute SQL query using that cursor
3. Write the output of the SQL query in a Temp file.
    a. Q - Why can't we store it in Dataframes?
4. Close the MySQL cursor
5. Create a HIVE connection
6. Read data from the Temp file
    a. Maps MySQL data type to Hive data type
    b. Example - Decimal in MySQL is DOUBLE in HIVE
7. Insert it into the HIVE table
8. Close the HIVE connection
9. Do Data Quality Checks

When translated into code it looks like this and this in Airflow terms is called a custom operator -

```
"""This module contains an operator to move data from MySQL to
Hive."""
from __future__ import annotations

from collections import OrderedDict
from tempfile import NamedTemporaryFile
from typing import TYPE_CHECKING, Sequence

import MySQLdb
import unicodecsv as csv
```

```
from airflow.models import BaseOperator
from airflow.providers.apache.hive.hooks.hive import HiveCliHook
from airflow.providers.mysql.hooks.mysql import MySqlHook

if TYPE_CHECKING:
   from airflow.utils.context import Context


class MySQLToHiveOperator(BaseOperator):
   """
   Moves data from MySql to Hive. The operator runs your query
against
   MySQL, stores the file locally before loading it into a Hive
table.
   If the ``create`` or ``recreate`` arguments are set to ``True``,
   a ``CREATE TABLE`` and ``DROP TABLE`` statements are generated.
   Hive data types are inferred from the cursor's metadata. Note that
the
   table generated in Hive uses ``STORED AS textfile``
   which isn't the most efficient serialization format. If a
   large amount of data is loaded and/or if the table gets
   queried considerably, you may want to use this operator only to
   stage the data into a temporary table before loading it into its
   final destination using a ``HiveOperator``.

   :param sql: SQL query to execute against the MySQL database.
(templated)
   :param hive_table: target Hive table, use dot notation to target a
       specific database. (templated)
   :param mysql_conn_id: source mysql connection
   :param hive_cli_conn_id: Reference to the
       :ref:`Hive CLI connection id <howto/connection:hive_cli>`.
```

```
    """
    template_fields: Sequence[str] = ("sql", "partition",
"hive_table")
    template_ext: Sequence[str] = (".sql",)
    template_fields_renderers = {"sql": "mysql"}

    def __init__(
        self,
        *,
        sql: str,
        hive_table: str,
        mysql_conn_id: str = "mysql_default",
        hive_cli_conn_id: str = "hive_cli_default",

        **kwargs,
    ) -> None:
        super().__init__(**kwargs)
        self.sql = sql
        self.hive_table = hive_table
        self.mysql_conn_id = mysql_conn_id
        self.hive_cli_conn_id = hive_cli_conn_id


    @classmethod
    def type_map(cls, mysql_type: int) -> str:
        """Maps MySQL type to Hive type."""
        types = MySQLdb.constants.FIELD_TYPE
        type_map = {
            types.BIT: "INT",
            types.DECIMAL: "DOUBLE",
            types.NEWDECIMAL: "DOUBLE",
            types.DOUBLE: "DOUBLE",
            types.FLOAT: "DOUBLE",
            types.INT24: "INT",
            types.LONG: "BIGINT",
            types.LONGLONG: "DECIMAL(38,0)",
            types.SHORT: "INT",
            types.TINY: "SMALLINT",
            types.YEAR: "INT",
            types.TIMESTAMP: "TIMESTAMP",
        }
        return type_map.get(mysql_type, "STRING")

    def execute(self, context: Context):
        mysql = MySqlHook(mysql_conn_id=self.mysql_conn_id)
        self.log.info("Dumping MySQL query results to local file")
        conn = mysql.get_conn()
        cursor = conn.cursor()
        cursor.execute(self.sql)
      # write output into a temp CSV File
      with NamedTemporaryFile("wb") as f:
            csv_writer = csv.writer(
                f,
                encoding="utf-8",
            )
            field_dict = OrderedDict()
            for field in cursor.description:
                field_dict[field[0]] = self.type_map(field[1])
            csv_writer.writerows(cursor)
            f.flush()
            cursor.close()
            conn.close()

            # load that CSV file into HIVE

            self.log.info("Loading file into Hive")
            # create HIVE connection
        hive=HiveCliHook(hive_cli_conn_id=self.hive_cli_conn_id)

            hive.load_file(
                f.name,
                self.hive_table,
                field_dict=field_dict
            )
        self.log.info('\nSuccessfully loaded into Hive!')
```

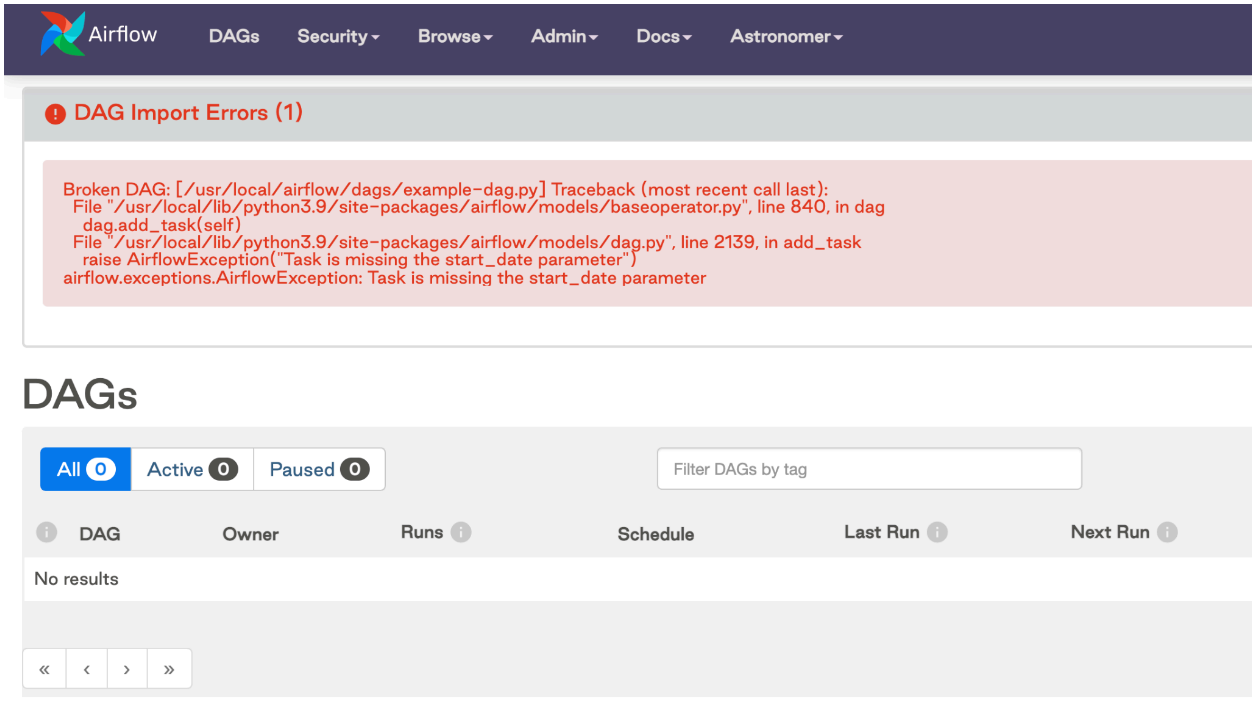
Example 2 - Use the MySQLToHiveOperator in a DAG to create the actual pipeline we set out for!

```
from datetime import timedelta

from airflow import DAG
from operators.mysql_to_hive_op import MySQLToHiveOperator
from airflow.utils.dates import days_ago

mysqlquery = """select * from ml_25m.genome_tags limit 10;"""
hive_table = 'ml_25m.genome_tags'  # schema_name.table_name
mysql_conn_id = 'mysql_conn'
hive_cli_conn_id = 'hive_local'
schedule_interval='0 3 * * *'  # https://crontab.guru/
dagrun_timeout=timedelta(minutes=10)

default_args = {
    'owner': 'prashant.tiwari_1@scaler.com',
    'start_date': airflow.utils.dates.days_ago(1),
    'depends_on_past': False,
    'catchup': False,
    'email': ['prashant.tiwari_1@sclaer.com', 'amit.singh_1@scaler.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}
```

```
with DAG(
    dag_id='mysql_to_hive',
    default_args=default_args,
    dagrun_timeout=dagrun_timeout,
    schedule_interval=schedule_interval,
    description='MySQL to Hive Data Transfer',
    tags=['mysql', 'hive']
    )as dag:
    load_mysql_to_hive = MySQLToHiveOperator(
        task_id="load_mysql_to_hive",
        sql= mysqlquery,
        hive_table=hive_table,
        mysql_conn_id=mysql_conn_id,
        hive_cli_conn_id=hive_cli_conn_id
    )



load_mysql_to_hive

if __name__ == '__main__ ':
  dag_mysql_to_hive.cli()
```

Where and how does this code execute?

- With these codes in the DAG folder, the webserver configurations are such that the corresponding DAGs will automatically appear in Airflow UI in the browser (**provided there is no import errors**)
  [DAGs don't appear in the Airflow UI]()



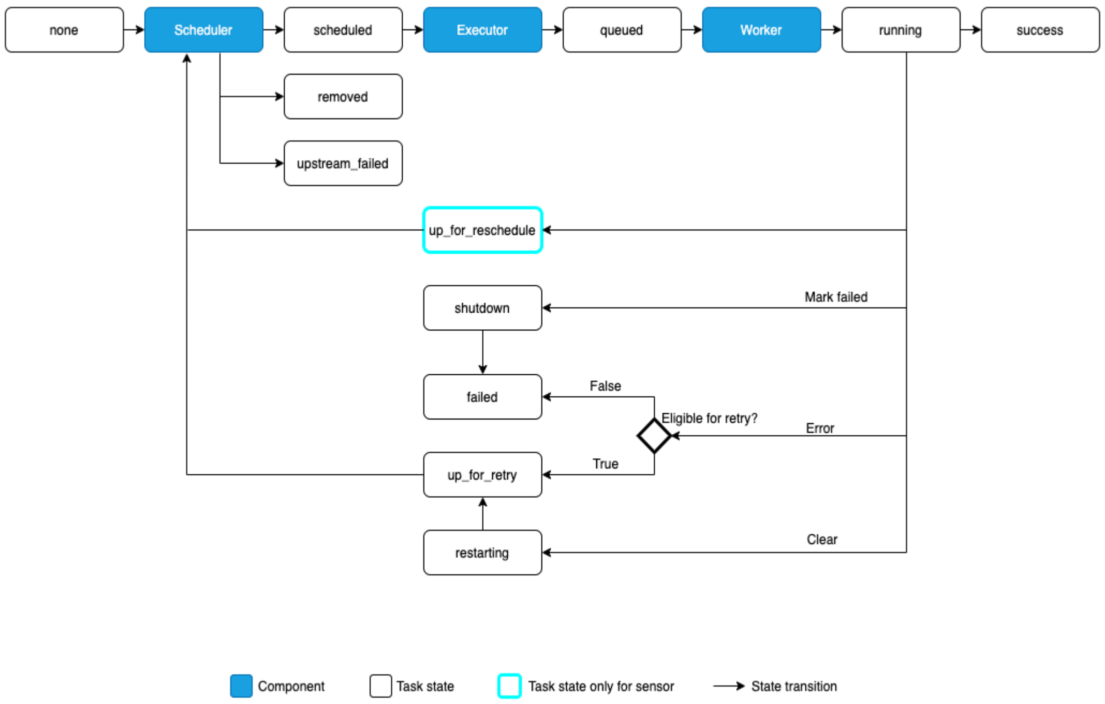
```
Broken DAG: [/usr/local/airflow/dags/example-dag.py] Traceback (most recent call last):
  File "/usr/local/lib/python3.9/site-packages/airflow/models/baseoperator.py", line 840, in dag
    dag.add_task(self)
  File "/usr/local/lib/python3.9/site-packages/airflow/models/dag.py", line 2139, in add_task
    raise AirflowException("Task is missing the start_date parameter")
airflow.exceptions.AirflowException: Task is missing the start_date parameter
```

- From the UI, we can enable the DAG and test its run. We can look for logs in case it **fails**.

What all other states a DAG can be in?

- When we trigger the DAG, it goes into multiple stages. An instance of a Task is a specific run of that task for a given DAG (and thus for a given data interval).
- They are also the representation of a Task that has state, representing what stage of the lifecycle it is in.
- The possible states for a Task Instance are:
  - none: The Task has not yet been queued for execution (its dependencies are not yet met)
  - scheduled: The scheduler has determined the Task's dependencies are met and it should run
  - queued: The task has been assigned to an Executor and is awaiting a worker
  - running: The task is running on a worker (or on a local/synchronous executor)
  - success: The task finished running without errors
  - shutdown: The task was externally requested to shut down when it was running
  - restarting: The task was externally requested to restart when it was running
  - failed: The task had an error during execution and failed to run

- skipped: The task was skipped due to branching, LatestOnly, or similar.
- upstream_failed: An upstream task failed and the Trigger Rule says we needed it
- up_for_retry: The task failed, but has retry attempts left and will be rescheduled.
- up_for_reschedule: The task is a Sensor that is in reschedule mode



- **Ideally, a task should flow from none, to scheduled, to queued, to running, and finally to success.**

For a few advanced concepts. Please refer to the Post Reads airflow1.2

Interview Questions -

- **You want to schedule a task to run every weekday at 9am, except on public holidays. How would you accomplish this using Airflow?**

    a. Define a schedule interval with weekdays and exclude public holidays in the DAG definition
    b. Use a Python function to dynamically calculate the next execution date

c. Use an external calendar service to specify the holidays and update the DAG definition accordingly
d. None of the above

- **How would you handle a situation where a task in a DAG takes much longer than expected to complete, causing delays in downstream tasks?**

  a. Increase the number of retries for the task
  b. Set up parallelism for the DAG to allow multiple tasks to run simultaneously
  c. Use a timeout for the task to prevent it from running indefinitely
  d. Implement a quality check for the output of the task and only trigger downstream tasks if the output meets certain criteria
- **You have a cron job scheduled to run every day at midnight, but you need to temporarily change the schedule to run every hour for the next 24 hours. How would you accomplish this?**

  a. Edit the cron job file to update the schedule and restart the cron daemon
  b. Use the crontab command to modify the cron job schedule for the next 24 hours
  c. Use a cron job scheduler tool to modify the schedule
  d. Create a new cron job with the updated schedule and delete the original job after 24 hours
- **What happens to a task's state if it times out while waiting for a resource?**

  a. The state remains unchanged.
  b. The state changes to "skipped".
  c. The state changes to "failed".
  d. The state changes to "upstream_failed".
- **How does the scheduler in Apache Airflow interact with the web server and the database?**

  a. The scheduler reads metadata and state information from the database and updates the web server.
  b. The scheduler reads task definitions from the web server and updates the database.
  c. The scheduler reads task definitions from the database and updates the web server.
  d. The scheduler reads metadata and state information from the web server and updates the database.