**Airflow:**

**Airflow is used for authoring,monitoring and scheduling a workflows.**

**Why we creating dag inside the task?**

In summary, the **dag=dag** part inside the task definition is used to link the task to a specific DAG, making it part of that workflow.

what is the use of the trigger rules?

ChatGPT

In Apache Airflow, Trigger Rules are used to define how a task should be triggered based on the status of its upstream (parent) tasks. When tasks are connected in a Directed Acyclic Graph (DAG), they form a dependency chain where the execution of one task depends on the successful completion or status of its parent task(s).

Trigger Rules allow you to control the flow of task execution based on the states of these dependencies. By default, if a task has multiple parent tasks, it will be triggered as soon as all its parent tasks have successfully completed. However, Trigger Rules provide more flexibility to customize this behavior.

Here are the available Trigger Rules and their purposes:

1. **all\_success**: (Default) The task will be triggered only if all its parent tasks have successfully completed. If any of the parent tasks fail, the task will not be executed.
2. **all\_failed**: The task will be triggered only if all its parent tasks have failed. If any of the parent tasks succeed, the task will not be executed.
3. **all\_done**: The task will be triggered only if all its parent tasks have completed, regardless of whether they succeeded or failed.
4. **one\_success**: The task will be triggered if at least one of its parent tasks has successfully completed.
5. **one\_failed**: The task will be triggered if at least one of its parent tasks has failed.
6. **none\_failed**: The task will be triggered if none of its parent tasks have failed. This means the task will run as long as all parent tasks have either succeeded or been skipped.
7. **none\_failed\_or\_skipped**: The task will be triggered if none of its parent tasks have failed or been skipped. This means the task will run as long as all parent tasks have either succeeded or are still running.

By using these Trigger Rules, you can design more complex workflows and control the task execution based on the desired conditions. For example, if you have multiple tasks that can run in parallel and one final task that depends on the success of all of them, you can use **all\_success** as the trigger rule for the final task. This ensures that the final task will only run when all its parallel dependencies are successful. Similarly, you can use other Trigger Rules to customize the behavior based on your workflow's requirements.

from airflow import DAG

from airflow.operators.python\_operator import PythonOperator

from airflow.operators.bash\_operator import BashOperator

from datetime import datetime

def task1\_function():

print("This is Task 1!")

def task2\_function():

print("This is Task 2!")

def task3\_function():

print("This is Task 3!")

dag = DAG('my\_dag', schedule\_interval='@daily', start\_date=datetime(2023, 7, 1), catchup=False)

# Define Task 1 using PythonOperator and set the trigger\_rule argument

task1 = PythonOperator(

task\_id='task1',

python\_callable=task1\_function,

dag=dag,

trigger\_rule='all\_success' # Here, we specify the trigger rule for Task 1

)

# Define Task 2 using BashOperator and set the trigger\_rule argument

task2 = BashOperator(

task\_id='task2',

bash\_command='echo "This is Task 2!"',

dag=dag,

trigger\_rule='one\_failed' # Here, we specify the trigger rule for Task 2

)

# Define Task 3 using PythonOperator and set the trigger\_rule argument

task3 = PythonOperator(

task\_id='task3',

python\_callable=task3\_function,

dag=dag,

trigger\_rule='all\_done' # Here, we specify the trigger rule for Task 3

)

# Set up task dependencies within the DAG

task1 >> task2

task2 >> task3

what is branching in airflow

ChatGPT

Branching in Apache Airflow refers to the capability of creating conditional workflows within a Directed Acyclic Graph (DAG). With branching, you can define different paths or branches in your DAG based on certain conditions or business logic, allowing tasks to take different routes depending on the outcomes of previous tasks or external factors.

In Airflow, you can use the **BranchPythonOperator** to implement branching. The **BranchPythonOperator** allows you to specify a Python function that determines the next task to execute based on some criteria. This function returns the task ID of the next task to run, effectively controlling the flow of the DAG dynamically.

Here's a simplified example of how branching works in Airflow:

from airflow import DAG

from airflow.operators.python\_operator import PythonOperator

from airflow.operators.dummy\_operator import DummyOperator

from airflow.operators.branch\_operator import BranchPythonOperator

from datetime import datetime

def decide\_branch(\*\*kwargs):

if some\_condition:

return 'task\_a'

else:

return 'task\_b'

def task\_a\_function():

# Your task\_a logic goes here

print("This is Task A!")

def task\_b\_function():

# Your task\_b logic goes here

print("This is Task B!")

dag = DAG('branching\_dag', schedule\_interval='@daily', start\_date=datetime(2023, 7, 1), catchup=False)

# Define a BranchPythonOperator to determine the next task based on some\_condition

branch\_operator = BranchPythonOperator(

task\_id='branch\_operator',

python\_callable=decide\_branch,

provide\_context=True,

dag=dag

)

# Define task\_a and task\_b as separate tasks

task\_a = PythonOperator(

task\_id='task\_a',

python\_callable=task\_a\_function,

dag=dag

)

task\_b = PythonOperator(

task\_id='task\_b',

python\_callable=task\_b\_function,

dag=dag

)

# Define a dummy task to represent the end of the branching paths

end\_task = DummyOperator(task\_id='end\_task', dag=dag)

# Set task dependencies and branching logic

branch\_operator >> [task\_a, task\_b]

task\_a >> end\_task

task\_b >> end\_task

In this example, the **decide\_branch** function is a Python function that returns the task ID of either **'task\_a'** or **'task\_b'** based on the value of **some\_condition**. The **BranchPythonOperator** uses this function to decide which path the DAG should follow.

When the DAG runs, the **decide\_branch** function is executed, and the returned task ID determines the next task to run. If **some\_condition** is true, **'task\_a'** will run, and if **some\_condition** is false, **'task\_b'** will run. After either **task\_a** or **task\_b** is completed, both paths merge into the **end\_task**.

With branching, you can create more flexible and dynamic workflows in Airflow that adapt to different conditions and scenarios during execution.

DAG Dependencies

ChatGPT

In Apache Airflow, DAGs (Directed Acyclic Graphs) represent workflows that consist of tasks and their dependencies. DAGs are used to define the sequence of tasks and the order in which they should be executed. The dependencies between tasks determine when each task will be triggered and how the workflow progresses.

Here are some important concepts related to DAG dependencies in Apache Airflow:

1. Task Dependencies: In a DAG, each task depends on one or more other tasks. These dependencies create a flow between the tasks, defining the order in which they should be executed. Task dependencies are set using the **>>** operator, where **task\_a >> task\_b** means that **task\_b** depends on **task\_a**, and **task\_a** should be executed before **task\_b**.
2. Downstream and Upstream Tasks: In a DAG, tasks can be categorized as downstream or upstream tasks based on their dependencies. Downstream tasks depend on other tasks and will be executed after their dependencies have successfully completed. Upstream tasks are those that other tasks depend on and will be executed before their dependent tasks.
3. Start and End Tasks: Every DAG should have a start and end task. The start task is the first task to be executed, and the end task represents the final task in the workflow. The start task has no dependencies, while the end task is not dependent on any other task.
4. Chaining Tasks: You can chain tasks together by using the **>>** operator to define their dependencies. For example, **task\_a >> task\_b >> task\_c** creates a chain of tasks, and **task\_a** will be executed first, followed by **task\_b**, and then **task\_c**.
5. Branching: Branching allows you to make conditional decisions in the workflow. The **BranchPythonOperator** is used to determine which task should be executed next based on a condition. The condition is evaluated in a Python function, and the returned task\_id specifies the next task to execute.
6. Trigger Rules: Trigger rules define the conditions under which a task should be triggered. For example, you can set a task to trigger only if all its upstream tasks have succeeded, or if any of them has failed, or if any of them is skipped, etc.
7. Parallelism: In some cases, you may have tasks that can be executed in parallel, meaning they do not have direct dependencies on each other and can be executed simultaneously. This can be achieved by setting their dependencies accordingly.

Overall, DAG dependencies are crucial for orchestrating the execution order of tasks in Apache Airflow. They provide a clear structure to your workflows and help in building complex data pipelines, ensuring tasks are executed in the correct order while handling failures and retries effectively.

Top of Form

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***Pausing/Deactivating a DAG**:

* Navigate to the Airflow Web UI (usually accessible at <http://localhost:8080/>).
* Go to the "DAGs" page to view a list of all available DAGs.
* Locate the DAG you want to pause or deactivate and click on the DAG name to access its details.
* On the DAG details page, click the "Pause" or "Deactivate" toggle button to pause or deactivate the DAG, respectively.
* Pausing the DAG will stop new DAG runs from being created, but existing DAG runs will continue to run.
* Deactivating the DAG will stop existing DAG runs and prevent new ones from being created.

If you prefer to use the **airflow** command-line interface (CLI), you can use the following commands:

* To pause a DAG: **airflow pause <DAG\_ID>**
* To activate (unpause) a DAG: **airflow unpause <DAG\_ID>**
* To deactivate a DAG: **airflow deactivate <DAG\_ID>**
* To delete a DAG: **airflow delete\_dag <DAG\_ID>**

Remember that these actions are irreversible, so exercise caution when pausing, deactivating, or deleting DAGs in your Airflow environment. It's a good practice to create backups or version control your DAG files to avoid accidental data loss.

DAG Runs

ChatGPT

In Apache Airflow, a DAG Run is an instance of a DAG that represents a specific execution of the DAG at a particular time. DAG Runs are created when the scheduler triggers the execution of a DAG based on its schedule\_interval or when a DAG is manually triggered.

Here are some key points about DAG Runs:

1. **Definition**: A DAG Run is a specific instantiation of a DAG. It represents a single run of the entire workflow defined by the DAG, and it captures the state and status of that particular run.
2. **Scheduled Runs**: DAG Runs are automatically created by the Airflow scheduler according to the defined schedule\_interval of the DAG. For example, if a DAG is scheduled to run daily, a new DAG Run will be created every day to trigger the DAG execution.
3. **Manually Triggered Runs**: Besides scheduled runs, you can manually trigger a DAG Run from the Airflow Web UI or using the **airflow** CLI. Manually triggered DAG Runs are useful when you want to run the DAG outside of its regular schedule or to rerun specific tasks within the DAG.
4. **DAG Run State**: Each DAG Run has a state that indicates its current status. The common states include "running," "success," "failed," "queued," "upstream\_failed," etc. The state reflects the overall status of the DAG Run and the individual task instances within it.
5. **Task Instances**: Within a DAG Run, each task defined in the DAG is executed as a Task Instance. Task Instances represent the actual executions of the tasks within a DAG Run and have their own status, such as "success," "failed," "skipped," etc.
6. **Retries**: If a task within a DAG Run fails, Airflow can automatically retry the task based on the **retry** settings defined for the task.
7. **Catchup**: When a DAG is first created or has its schedule\_interval changed, the scheduler can create multiple DAG Runs in quick succession to "catch up" on missed execution periods. This behavior can be controlled using the **catchup** parameter in the DAG definition.

DAG Runs and Task Instances are crucial for managing the execution of workflows in Airflow. They provide a historical record of DAG executions, allow you to track the status of individual tasks, and enable automatic retries in case of failures. This gives you better visibility and control over your data pipelines and ensures reliable data processing.

DAG Run Status

ChatGPT

In Apache Airflow, DAG Run Status refers to the current state of a specific execution instance (DAG Run) of a DAG. Each DAG Run can have one of several status values, which indicate the progress and outcome of the execution. Understanding the DAG Run Status is essential for monitoring and managing your data workflows.

The common DAG Run Status values in Apache Airflow include:

1. **running**: The DAG Run is currently in progress, and tasks are being executed.
2. **success**: All tasks in the DAG Run have completed successfully without any errors.
3. **failed**: One or more tasks in the DAG Run have failed during execution.
4. **up\_for\_retry**: A task has failed and is eligible for retries based on the **retry** settings defined for the task.
5. **up\_for\_reschedule**: The DAG Run has been rescheduled to run again due to changes in the DAG definition or schedule.
6. **queued**: The DAG Run is waiting in the queue to be executed.
7. **scheduled**: The DAG Run has been scheduled to run at a specific time in the future.
8. **none**: The DAG Run status is not yet defined or set.
9. **skipped**: A task has been skipped during execution, either due to a trigger rule or a condition.
10. **upstream\_failed**: One or more upstream tasks have failed, and the current task won't be executed due to trigger rules.
11. **up\_for\_retry**: A task has failed and is eligible for retries based on the **retry** settings defined for the task.
12. **up\_for\_reschedule**: The DAG Run has been rescheduled to run again due to changes in the DAG definition or schedule.

These status values provide information about the progress and outcome of the DAG Run, as well as any issues encountered during execution. Monitoring the DAG Run Status is crucial for identifying and addressing failures, ensuring data pipeline reliability, and maintaining data integrity.

You can view the DAG Run Status in the Airflow Web UI, where you can see the status of each DAG Run along with the status of individual Task Instances within that run. Additionally, you can use the **airflow** command-line interface (CLI) to check the status of DAG Runs and Task Instances.

1. **"failed" Status**: When a task fails, it will have the status "failed" to indicate that the task execution encountered an error or exception.
2. **"upstream\_failed" Status**: In addition to the "failed" status for the failed task, any downstream tasks that depend on the failed task will also have the status "upstream\_failed." This status is used to indicate that the downstream tasks were not executed due to the failure of their upstream dependencies.

A screenshot of a computer

Description automatically generated