ISSN: 2754-6705

Journal of Mathematical & Computer Applications

SCIENTIFIC Research and Community

Review Article

Open Access

Establishing Dependencies between Multiple DAGs in Apache Airflow: Coordinating Complex Workflows

Pankaj Dureja

USA

ABSTRACT

It covers a guiding light on approaches and resolutions for managing dependencies amongst multiple Directed Acyclic Graphs (DAGs) in Apache Airflow (a trending work automation tool). Cross-DAG dependencies is one of the crucial requirement for to manage the very complex workflow which can span in different processes and different systems. Through this research we implement basic operators including ExternalTaskSensor and TriggerDagRunOperator, custom solutions like using MySQL DB table to keep track of running DAGs. This study demonstrates the pros, possible bespoke solutions and implications of these solutions for optimizing workflow automation through Apache Airflow. The paper concludes by providing an analysis of how well DAG dependency management is done in Airflow and provides possible roadmap.

***Corresponding author**

Pankaj Dureja, USA.

Received: February 12, 2024; **Accepted:** February 19, 2024, **Published:** February 26, 2024

Keywords: Apache Airflow, DAG Dependencies, Cross-DAG Dependencies, Workflow Automation, External Task Sensor, Trigger Dag Run Operator, My SQL Tracking, Data Pipelines, Task Scheduling

Introduction

With the rise of data-driven organizations, Apache Airflow has made it a go-to tool to orchestrate complex workflows. This is where Airflow comes in, with the capability to script, schedule, and visualize all parts of workflow it lends itself to orchestrating arduously detailed data processing pipelines. In Airflow, every workflow is a Directed Acyclic Graph (DAG) where every node is a task, and the edges define the dependencies between tasks. Although Airflow is good at managing tasks inside a single DAG, in reality, need dependencies between multiple DAGs to deal with more complex and intertwined processes.

Many data-driven environment of works are not sitting as silos. Rather, they are a part of a bigger system where completion of one workflow would kick off another. So, with the above pattern, there are separate DAGs for extract, transform and load in an ETL process. In order to get these DAGs working together it is also needed to set up dependencies across these DAGs to make sure each step is executed in the right order. It does information co-ordination is essential to maintain data integrity for fast access to data.

To enforce this global scheduling constraint, we need to rely on cross-DAG dependencies to make sure that the DAGs get executed in the correct order, cooperate in a proper way and, thus, optimize the entire operations and reliability. Workflows may break up when dependencies in the software are not maintained, this may cause errors, delays, and other inefficiencies. E.g., if a data loading DAG starts before the data transformation DAG completes, it might also load incomplete/incorrect data into the target system. Therefore, handling these dependencies is critical to data processing with ease and without errors.

In this paper, we explore methods, and operators that can be used to handle dependencies among multiple DAGs in Airflow. From common methods like the ExternalTaskSensor and TriggerDagRunOperator, to more ad hoc approaches such as databases to monitor dependencies and control running workflows. In this article, we hope to cover these approaches, and compile a guide on how to orchestrate complex workflows that span multiple DAGs from running seamlessly and efficiently.

Problem Statement

Managing dependencies within a single DAG in Airflow is straightforward, but coordinating tasks across multiple DAGs presents several challenges. Without proper mechanisms to establish and manage these dependencies, workflows can become disjointed, leading to errors, delays, and inefficiencies. Organizations need a comprehensive solution that allows for seamless integration and synchronization of tasks across different DAGs. The absence of such a solution can result in fragmented workflows, increased manual intervention, and difficulties in troubleshooting and maintaining the data pipelines.

Solution Implemented

To address these challenges, this paper implements a solution using various operators and customized approaches in Apache Airflow:

1. **ExternalTaskSensor:** This operator waits for a task in an external DAG to complete before proceeding.

Following library needs to imported in the DAG:

from airflow.sensors.external_task_sensor import ExternalTaskSensor

In the below python implementation of DAG, **wait_for_**

transform_data task will run when transorm_data task will be completed in external **data_extraction_transformation_dag**.

```
# Define DAG B
with DAG(
    dag_id='data_loading_dag',
    default_args=default_args,
    description='DAG B: Data Loading',
    schedule_interval='@daily',
    catchup=False,
) as dag_b:

    start = DummyOperator(
        task_id='start'
    )

    wait_for_transform_data = ExternalTaskSensor(
        task_id='wait_for_transform_data',
        external_dag_id='data_extraction_transformation_dag',  # DAG A's ID
        external_task_id='transform_data',  # Task ID to wait for in DAG A
        mode='poke',
        poke_interval=60,  # Check every 60 seconds
        timeout=3600  # Timeout after 1 hour
    )

    end = DummyOperator(
        task_id='end'
    )

    start >> wait_for_transform_data >> end
```

Use Case: Ensuring that a data processing task in DAG A only starts after a data extraction task in DAG B is complete.

Drawback of External Task Sensor Operator
The ExternalTaskSensor Operator may encounter issues in a distributed setup with multiple Celery nodes, as it relies on the metadata database for task status, leading to potential synchronization problems if different nodes are not properly synced.

2. **TriggerDagRunOperator:** This operator triggers another DAG run in the chain.
Following library needs to imported in the DAG:
from airflow.operators.dagrun_operator import TriggerDagRunOperator

In the below python implementation of DAG, external dag **trigger_production_load_complete_dag** will be executed once **combined_count_email_task** completes within the DAG.

```
trigger_production_load_complete_dag = TriggerDagRunOperator(
    task_id='trigger_production_load_complete',
    trigger_dag_id='production_load_complete',  # Specify the ID of the DAG to trigger
    dag=dag,
)

last_task >> combined_count_and_email_task >> trigger_production_load_complete_dag >> end
```

Use Case: Automatically starting DAG B once a critical task in DAG A is completed.

Drawback of External Task Sensor Operator

The TriggerDagRunOperator does not wait for the triggered DAG to complete before moving on to the next task. For instance, in the following line of Python code, There are two external DAGs to be triggered: `**trigger_production_load_complete_dag` and `trigger_production_load_aggregate_dag**`.

last_task >> combined_count_and_email_task >> send_email_task >> **trigger_production_load_complete_dag** >> **trigger_production_load_aggregate_dag** >> end

The TriggerDagRunOperator triggers `trigger_production_load_complete_dag` and then immediately proceeds to trigger `trigger_production_load_aggregate_dag` without waiting for the first DAG to complete.

3. **Custom MySQL Table & Procedure for DAG Status**: In order to solve the problem of ExternalTaskSensor and TriggerDagRunOperator, a custom solution where a MySQL table is used to update the running status of a DAG. Other DAGs query this table to check if a specific DAG is running, using stored procedures to manage the checks.

Following library needs to imported in the DAG:

from airflow.providers.mysql.operators.mysql import MySqlOperator

MySQL Table

```
CREATE TABLE `airflow_dag_status` (
  `dag_id` varchar(2000) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL,
  `status` varchar(200) CHARACTER SET utf8 COLLATE utf8_general_ci DEFAULT 'NOT RUNNING',
  `create_ts` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `update_ts` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (`dag_id`)
) AUTOSTATS_CARDINALITY_MODE=PERIODIC AUTOSTATS_HISTOGRAM_MODE=CREATE SQL_MODE='PIPES_AS_CONCAT,STRICT_ALL_TABLES'

SELECT * FROM airflow_dag_status;
```

ow_dag_status 1 ×

CT * FROM airflow_dag_status | Enter a SQL expression to filter results (use Ctrl+Space)

dag_id	status	create_ts	update_ts
DAG 1	NOT RUNNING	2024-01-07 16:09:43.0	2024-01-09 16:09:43.0
DAG 2	RUNNING	2024-01-11 01:18:42.0	2024-01-29 09:37:04.0
DAG 3	NOT RUNNING	2024-01-07 16:09:43.0	2024-01-09 16:31:56.0

MySQL Proceudure

```
CREATE OR REPLACE PROCEDURE `update_airflow_dag_status`(pn_update_id bigint(20) NULL
, pv_dag_name varchar(2000) CHARACTER SET utf8 COLLATE utf8_general_ci NULL
, pv_dag_status varchar(255) CHARACTER SET utf8 COLLATE utf8_general_ci NULL) RETURNS void AS

DECLARE

ln_app_id            INT = 170;
lv_entry             VARCHAR(20000);
lv_subentry          VARCHAR(20000);
lv_comment       TEXT;
lv_error_msg         TEXT;
ln_row_count     BIGINT;
lv_dag_name VARCHAR(2000);
lv_dag_status VARCHAR(255) ;

BEGIN

    lv_entry = 'update_airflow_dag_status';
    lv_subentry = pn_update_id;

    lv_dag_name = pv_dag_name;
    lv_dag_status = pv_dag_status;

    START TRANSACTION;

    INSERT IGNORE INTO airflow_dag_status(dag_id)
    SELECT UPPER(lv_dag_name) dag_id;

    UPDATE airflow_dag_status
    SET status = lv_dag_status
    WHERE dag_id = UPPER(lv_dag_name);

    ln_row_count = row_count();

    COMMIT;


EXCEPTION
    WHEN OTHERS
    THEN
        ROLLBACK;

        lv_error_msg = exception_message();
        CALL save_log_prc(ln_app_id, lv_entry, lv_subentry, CONCAT('ERROR:MSG:',lv_error_msg,lv_comment), 1);

END;
```

Python Implementation

Using MySQL Hook, dependencies can be checked, and if the DAG is already running, the waiting DAG can enter sleep mode for 2 minutes until the condition is satisfied.

```
def check_dag_running(**kwargs):
    ti = kwargs['ti']
    hook = MySqlHook(mysql_conn_id='ms_ie_dba_conn')
    countsql = "SELECT count(1) dag_running_count FROM airflow_dag_status WHERE dag_id IN ('DAG 1', 'DAG 2','DAG 3') AND status = 'RUNNING'"
    while True:
        dag_run_count = hook.get_first(sql=countsql)
        # dag_run_count = kwargs['ti'].xcom_pull(task_ids='get_dag_running_status_count_task', key='return_value')
        print("Dag Run Count:", dag_run_count[0])  # Print the value of dag_run_count
        if dag_run_count[0] == 0:
            break  # Exit the loop if the condition is met
        else:
            print("Waiting for DAG running count to be zero. Sleeping for 2 minutes...")
            sleep(2 * 60)  # Sleep for 2 minutes before rechecking the condition
```

Use Case: A DAG updates its status in a MySQL table upon start and completion. Other DAGs query this table using a stored procedure to check the status, sleeping for 2 minutes before rechecking if the desired status is not yet achieved.

Potential Extended Use Cases

The solutions for managing dependencies between multiple DAGs in Airflow can be extended to various other scenarios:

1. **Complex ETL Pipelines:** Coordinating different stages of ETL processes across multiple DAGs, where each stage is a separate DAG.
2. **Microservices Orchestration:** Managing workflows in a microservices architecture where each service is represented by a separate DAG.
3. **Batch Processing and Real-time Processing Integration:** Ensuring that batch processing tasks are aligned with real-time data processing tasks.

Impact

The implementation of these solutions has a significant impact on the efficiency and reliability of workflows involving multiple DAGs. Airflow takes care of this coordination instead of saying "run this workflow, then this workflow, etc," using Airflow can establish cross-DAG dependencies, ensuring that workflows are run in the correct order with minimal manual intervention. As a result, workflow get minimal error, better task synchronization, and resource utilization. Increased visibility and control of the entire workflow allows for greater operational efficiency and ease of troubleshooting and maintaining the data pipelines.

Scope

The purpose of this paper is to dig into creating and managing DAG Dependencies between multiple DAGs in Apache Airflow. While the article does not go much into the advanced topics like Airflow architecture optimization or security considerations, it covers the implementation details, benefits and some potential use cases. In any event, future research will likely explore these other dimensions as well, in order to arrive at a more complete elasticity picture of how Airflow handles sophisticated workflows [1-5].

Conclusion

DAG dependencies are important to manage workflows across different processes as we can create multiple DAGs to represent different steps in our process. The operators, as well as the custom solutions presented in this paper, for example, ExternalTaskSensor, TriggerDagRunOperator, the DAG status tracking in MySQL table, supports easy DAG coordination, and guarantees workflow execution. Through these solutions, enterprises can establish deep automation, enhance task scheduling and streamline workflows. This aspect signifies the talent and applicability of these techniques across diverse domains and the promise of future extended use cases that might be uncovered.

References

1. Maxime Beauchemin (2021) The Apache Airflow Book, O'Reilly Media 45-70.
2. Anirudh Kala (2020) Apache Airflow: A Real-World Guide to Data Pipelines. Packt Publishing 115-140.
3. Wes McKinney (2017) Python for Data Analysis. O'Reilly Media 220-245.
4. Apache Airflow Operators https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/operators.html.
5. Airflow Operators https://www.astronomer.io/docs/learn/what-is-an-operator.