Apache Airflow

What is Orchestration in BigData?

Grow Data Skills

Orchestration in big data refers to the automated configuration, coordination, and management of complex big data systems and services. Like an orchestra conductor ensures each section of the orchestra plays its part at the right time and the right way to create harmonious music, orchestration in big data ensures that each component of a big data system interacts in the correct manner at the right time to execute complex, multi-step processes efficiently and reliably.

In practical terms, orchestration involves:

- **Workflow management:** It defines, schedules, and manages workflows involving multiple tasks across disparate systems. These workflows can be simple linear sequences or complex directed acyclic graphs (DAGs) with branching and merging paths.

- **Task scheduling:** Orchestration tools schedule tasks based on their dependencies. This ensures that tasks are executed in the correct order and that tasks that can run in parallel do so, increasing overall system efficiency.

- **Failure handling:** Orchestration tools handle failures in the system, either by retrying failed tasks, skipping them, or alerting operators to the failure.

Need of Workflow/Dependency management while Designing Data Pipelines?

Grow Data Skills

Workflow or dependency management is a crucial component in the design of data pipelines, especially in the context of big data. Here's why:

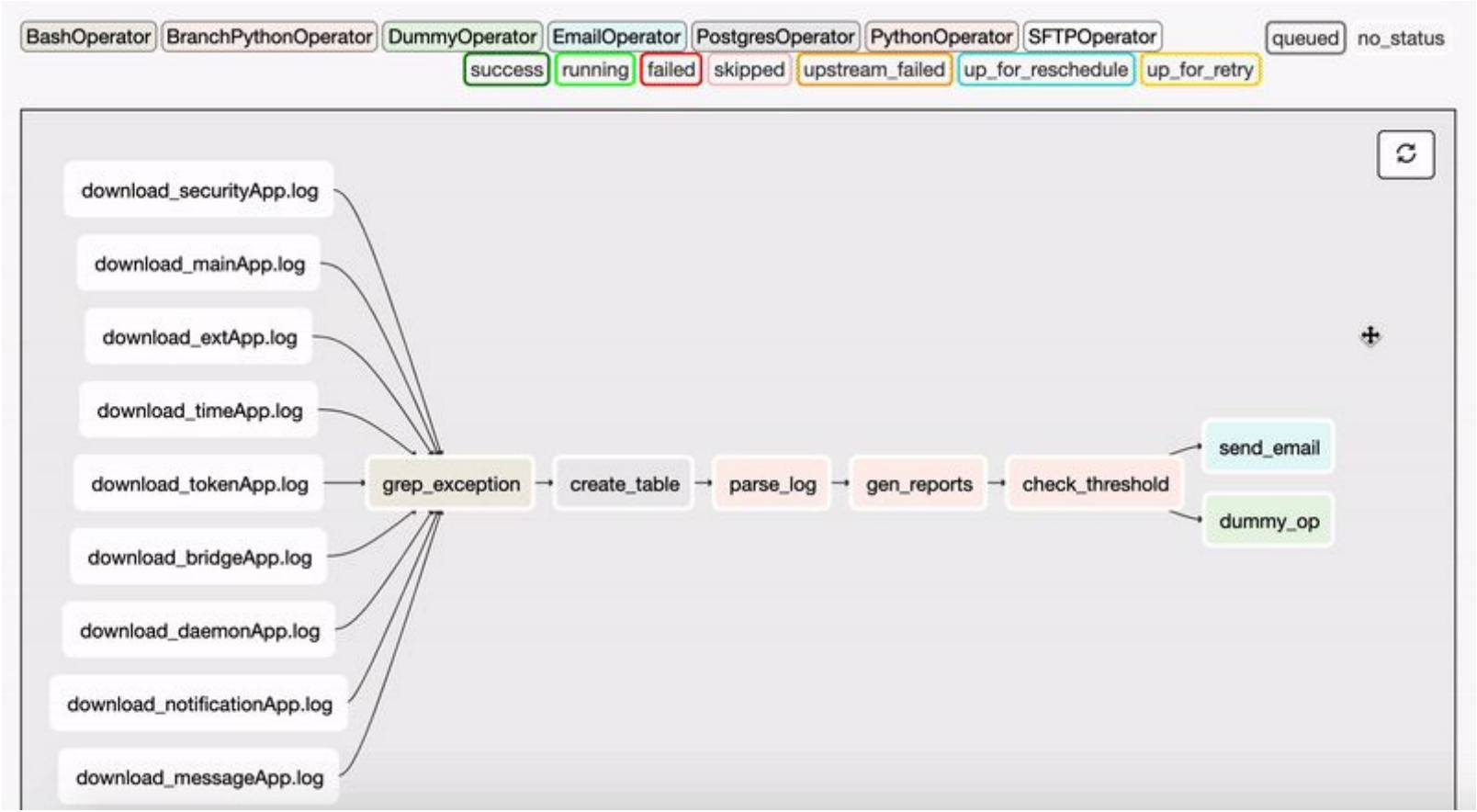
- **Ordering and Scheduling:** Data processing tasks often have dependencies, meaning one task needs to complete before another can begin. For example, a task that aggregates data may need to wait until the data has been extracted and cleaned. A workflow management system can keep track of these dependencies and ensure tasks are executed in the right order.

- **Parallelization**: When tasks don't have dependencies, they can often run in parallel. This can significantly speed up data processing. Workflow management systems can manage parallel execution, maximizing the use of computational resources and reducing overall processing time.

- **Error Handling**: If a task in a data pipeline fails, it can have a knock-on effect on other tasks. Workflow management systems can handle these situations, for instance by retrying failed tasks, skipping them, or stopping the pipeline and alerting operators.

- **Visibility and Monitoring**: Workflow management systems often provide tools for monitoring the progress of data pipelines and visualizing their structure. This can make it easier to spot bottlenecks or failures, and to understand the flow of data through the pipeline.

- **Resource Management:** Workflow management systems can allocate resources (like memory, CPU, etc.) depending on the requirements of different tasks. This helps in efficiently utilizing resources and ensures optimal performance of tasks.


BashOperator
BranchPythonOperator
DummyOperator
EmailOperator
PostgresOperator
PythonOperator
SFTPOperator
queued
no_status
success
running
failed
skipped
upstream_failed
up_for_reschedule
up_for_retry
download_securityApp.log
download_mainApp.log
download_extApp.log
download_timeApp.log
download_tokenApp.log
download_bridgeApp.log
download_daemonApp.log
download_notificationApp.log
download_messageApp.log
grep_exception
create_table
parse_log
gen_reports
check_threshold
send_email
dummy_op

What is AirFlow?

Grow Data Skills

Apache Airflow is an open-source platform that programmatically allows you to **author**, **schedule**, and **monitor** workflows. It was originally developed by Airbnb in 2014 and later became a part of the Apache Software Foundation's project catalog.

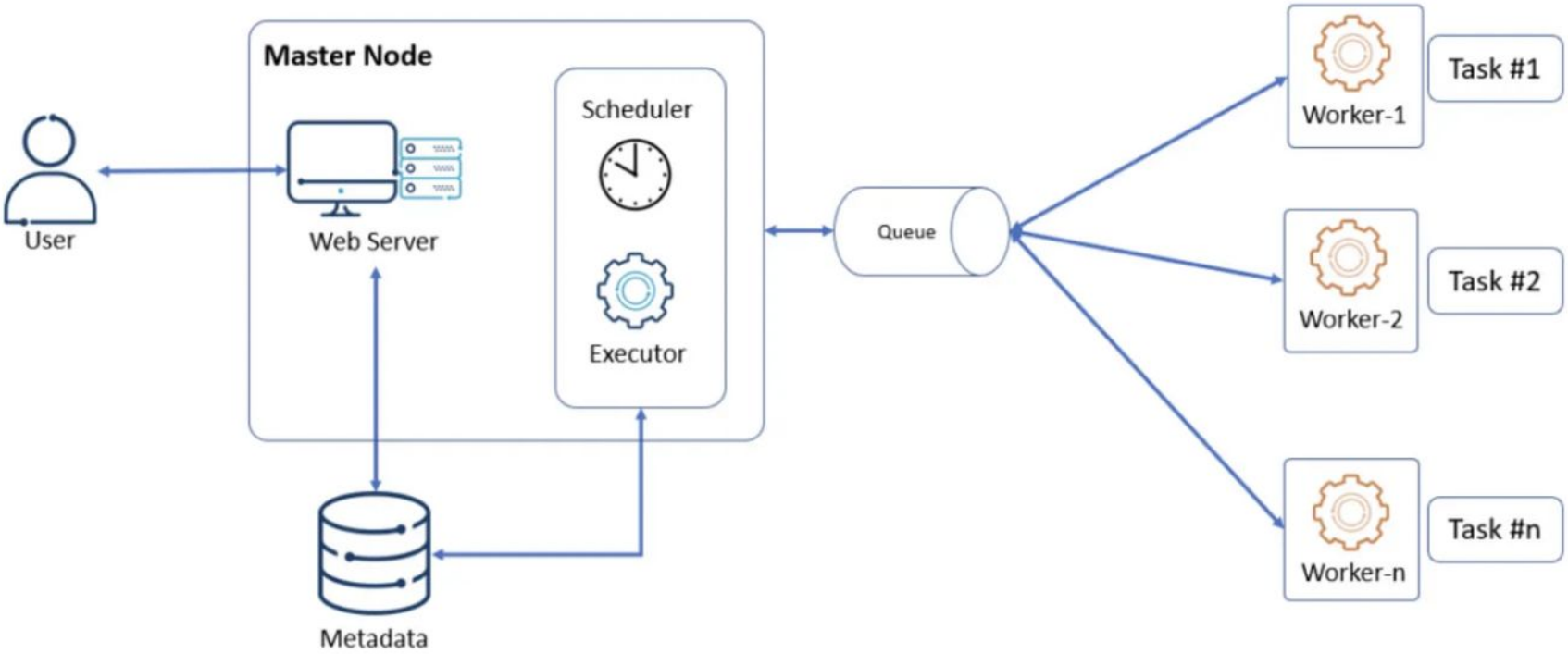
Airflow uses **directed acyclic graphs (DAGs)** to manage workflow orchestration. DAGs are a set of tasks with directional dependencies, where the tasks are the nodes in the graph, and the dependencies are the edges. In other words, each task in the workflow executes based on the completion status of its predecessors.

Key features of Apache Airflow include:

- **Dynamic Pipeline Creation:** Airflow allows you to create dynamic pipelines using Python. This provides flexibility and can be adapted to complex dependencies and operations.
- **Easy Scheduling:** Apache Airflow includes a scheduler to execute tasks at defined intervals. The scheduling syntax is quite flexible, allowing for complex scheduling.
- **Robust Monitoring and Logging:** Airflow provides detailed status and logging information about each task, facilitating debugging and monitoring. It also offers a user-friendly UI to monitor and manage the workflow.
- **Scalability**: Airflow can distribute tasks across a cluster of workers, meaning it can scale to handle large workloads.
- **Extensibility**: Airflow supports custom plugins and can integrate with several big data technologies. You can define your own operators and executors, extend the library, and even use the user interface API.
- **Failure Management**: In case of a task failure, Airflow sends alerts and allows for retries and catchup of past runs in a robust way.

AirFlow Architecture

Grow Data Skills



Airflow Architecture

Grow Data Skills

- **Scheduler**: The scheduler is a critical component of Airflow. Its primary function is to continuously scan the DAGs (Directed Acyclic Graphs) directory to identify and schedule tasks based on their dependencies and specified time intervals. The scheduler is responsible for determining which tasks to execute and when. It interacts with the metadata database to store and retrieve task state and execution information.

- **Metadata Database**: Airflow leverages a metadata database, such as PostgreSQL or MySQL, to store all the configuration details, task states, and execution metadata. The metadata database provides persistence and ensures that Airflow can recover from failures and resume tasks from their last known state. It also serves as a central repository for managing and monitoring task execution.

- **Web Server**: The web server component provides a user interface for interacting with Airflow. It enables users to monitor task execution, view the status of DAGs, and access logs and other operational information. The web server communicates with the metadata database to fetch relevant information and presents it in a user-friendly manner. Users can trigger manual task runs, monitor task progress, and configure Airflow settings through the web server interface.

- **Executors**: Airflow supports different executor types to execute tasks. The executor is responsible for allocating resources and running tasks on the specified worker nodes.

Airflow Architecture

- **Worker Nodes:** Worker nodes are responsible for executing the tasks assigned to them by the executor. They retrieve the task details, dependencies, and code from the metadata database and execute the tasks accordingly. The number of worker nodes can be scaled up or down based on the workload and resource requirements.

- **Message Queue:** Airflow relies on a message queue system, such as RabbitMQ, Apache Kafka, or Redis, to enable communication between the scheduler and the worker nodes. The scheduler places task execution requests in the message queue, and the worker nodes pick up these requests, execute the tasks, and update their status back to the metadata database. The message queue acts as a communication channel, ensuring reliable task distribution and coordination.

- **DAGs and Tasks:** DAGs are at the core of Airflow's architecture. A DAG is a directed graph consisting of interconnected tasks. Each task represents a unit of work within the data pipeline. Tasks can have dependencies on other tasks, defining the order in which they should be executed. Airflow uses the DAG structure to determine task dependencies, schedule task execution, and track their progress.

Each **task** within a DAG is associated with an **operator**, which defines the type of work to be performed. Airflow provides a rich set of built-in operators for common tasks like file operations, data processing, and database interactions. Additionally, custom operators can be created to cater to specific requirements.

Tasks within a DAG can be triggered based on various events, such as time-based schedules, the completion of other tasks, or the availability of specific data.

Operators in Airflow

Grow Data Skills

Apache Airflow has several types of operators that allow you to perform different types of tasks. Here are some of the most common ones:

- **BashOperator:** Executes a bash command.
- **PythonOperator:** Calls a Python function.
- **EmailOperator:** Sends an email.
- **SimpleHttpOperator:** Sends an HTTP request.
- **MySqlOperator, SqliteOperator, PostgresOperator, MsSqlOperator, OracleOperator**, etc.: Executes a SQL command.
- **DummyOperator:** A placeholder operator that does nothing.
- **Sensor:** Waits for a certain time, file, database row, S3 key, etc. There are many types of sensors, like HttpSensor, SqlSensor, S3KeySensor, TimeDeltaSensor, ExternalTaskSensor, etc.
- **SSHOperator:** Executes commands on a remote server using SSH.
- **DockerOperator:** Runs a Docker container.
- **SparkSubmitOperator:** Submits a Spark job.

Operators in Airflow

- **S3FileTransformOperator:** Copies data from a source S3 location to a temporary location on the local filesystem, transforms the data, and then uploads it to a destination S3 location.
- **S3ToRedshiftTransfer:** Transfers data from S3 to Redshift.
- **EmrAddStepsOperator:** Adds steps to an existing EMR (Elastic Map Reduce) job flow.
- **EmrCreateJobFlowOperator:** Creates an EMR JobFlow, i.e., a cluster.
- **AthenaOperator**: Executes a query on AWS Athena.
- **AwsGlueJobOperator:** Runs an AWS Glue Job.
- **S3DeleteObjectsOperator:** Deletes objects from an S3 bucket.
- **BigQueryOperator:** Executes a BigQuery SQL query.
- **BigQueryToBigQueryOperator:** Copies data from one BigQuery table to another.
- **DataprocClusterCreateOperator:** This operator is used to create a new cluster of machines on GCP's Dataproc service.
- **DataProcPySparkOperator:** This operator is used to submit PySpark jobs to a running Dataproc cluster.
- **DataProcSparkOperator:** This operator is used to submit Spark jobs written in Scala or Java to a running Dataproc cluster.
- **DataprocClusterDeleteOperator:** This operator is used to delete an existing cluster.

How to write AirFlow DAG Script?

Grow Data Skills

```
# Import necessary libraries
from datetime import datetime, timedelta
from airflow import DAG
from airflow.operators.dummy_operator import DummyOperator

# Define the default_args dictionary
default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'email_on_failure': False,
    'email_on_retry': False,
    'email': ['your-email@example.com'],
    'retries': 1,
    'retry_delay': timedelta(minutes=5)
}

# Instantiate a DAG
dag = DAG(
    'example_dag',
    default_args=default_args,
    description='An example DAG',
    schedule_interval=timedelta(days=1),
    start_date=datetime(2022, 1, 1),
    catchup=False,
    tags=['example']
)

# Define tasks and set their dependencies
start_task = DummyOperator(task_id='start', dag=dag)
end_task = DummyOperator(task_id='end', dag=dag)

start_task >> end_task
```

Attribute Description

Grow Data Skills

1. **'owner': 'airflow'** -> The owner of the task, using it can help identify the person who should be notified in case of task/job issues.
2. **'depends_on_past': False** -> If set to True, the task instance will fail if the previous task did not succeed.
3. **'email_on_failure': False** -> If set to True, Airflow will email the address specified in the 'email' key upon task failure.
4. **'email_on_retry': False** -> If set to True, Airflow will email the address specified in the 'email' key if the task needs to be retried.
5. **['your-email@example.com']** -> List of email addresses to notify if defined conditions such as failure or retries are met.
6. **'retries': 1** -> Number of retries in case of task failure.
7. **'Retry_delay': timedelta(minutes=5)** -> Time delay between retries.
8. **'example_dag'** -> Unique string identifier for the DAG.
9. **default_args=default_args** -> Dictionary of parameters & their default values.
10. **description='An example DAG'** -> String describing the purpose of the DAG.
11. **schedule_interval=timedelta(days=1)** -> How often the DAG should run. Uses cron-like string, or timedelta objects.
12. **start_date=datetime(2022, 1, 1)** -> The earliest logical date for starting the DAG.
13. **catchup=False** -> If set to True, then the DAG will catch up for all the missed runs since 'start_date'.
14. **tags=['example']** -> List of tags that can be used for filtering in the UI.

How to execute tasks parallelly

Grow Data Skills

```
start_task = DummyOperator(task_id='start_task', dag=dag)

parallel_task_1 = DummyOperator(task_id='parallel_task_1', dag=dag)
parallel_task_2 = DummyOperator(task_id='parallel_task_2', dag=dag)
parallel_task_3 = DummyOperator(task_id='parallel_task_3', dag=dag)

end_task = DummyOperator(task_id='end_task', dag=dag)

# Setting up the dependencies
start_task >> [parallel_task_1, parallel_task_2, parallel_task_3] >> end_task
```

In this example, the three parallel tasks **(parallel_task_1, parallel_task_2, parallel_task_3)** are specified as a list in the dependency chain. The **start_task** runs first. Once it completes, **all three parallel** tasks begin. When all three of them complete, the **end_task** starts.

How to overwrite depends_on_past property?

Grow Data Skills

The **depends_on_past** attribute in the **default_args** dictionary of a DAG **applies globally** to all tasks in the DAG when set. However, if you want to override this behavior for specific tasks, you can specify the depends_on_past attribute directly on those tasks.

For specific tasks where you want to override this behavior, set **depends_on_past** directly on the task:

```
task_with_custom_dep = DummyOperator(
    task_id='task_with_custom_dep',
    depends_on_past=False,
    dag=dag
)
```