**1.basic understanding of airflow**

**The Airflow platform is a tool for describing, executing, and monitoring workflows.**

[**https://www.youtube.com/watch?v=5peQThvQmQk**](https://www.youtube.com/watch?v=5peQThvQmQk)

[**https://www.youtube.com/watch?v=1nk9pfbJM3Y**](https://www.youtube.com/watch?v=1nk9pfbJM3Y)

[**https://www.youtube.com/watch?v=5b2ZRswqSZc**](https://www.youtube.com/watch?v=5b2ZRswqSZc)

**2. spark understanding**

**The spark is introduced to process the big data 100 faster than hadoop**

[**https://www.youtube.com/watch?v=v\_uodKAywXA**](https://www.youtube.com/watch?v=v_uodKAywXA)

**1.pull operation/task in airflow dag**

In Apache Airflow, a "pull" operation usually refers to the process of fetching data from an external source (like a database, API, or file system) as part of a Directed Acyclic Graph (DAG) task. Here's a brief overview of how to implement a pull operation in an Airflow DAG:

**Steps to Implement a Pull Operation in an Airflow DAG**

1. **Set Up Your Environment**: Make sure you have Apache Airflow installed and configured. You can install it using pip if you haven't done so:

bash

Copy code

pip install apache-airflow

1. **Create a DAG File**: Create a new Python file in your Airflow DAGs folder (usually located at ~/airflow/dags/).
2. **Define Your DAG**: Here's a simple example of a DAG that pulls data from an API:

python

Copy code

from airflow import DAG

from airflow.operators.python import PythonOperator

from datetime import datetime

import requests

# Define your pull function

def pull\_data\_from\_api():

url = 'https://api.example.com/data' # Replace with your API URL

response = requests.get(url)

data = response.json() # or response.text for raw data

print(data) # Do something with the data, like saving it to a file or database

# Define the DAG

with DAG(

'pull\_data\_dag',

schedule\_interval='@daily',

start\_date=datetime(2023, 10, 1),

catchup=False,

) as dag:

# Define the task

pull\_task = PythonOperator(

task\_id='pull\_data',

python\_callable=pull\_data\_from\_api,

)

pull\_task

1. **Deploy the DAG**: Once your DAG is defined, place the Python file in the DAGs folder. Airflow will automatically detect the new DAG.
2. **Trigger the DAG**: You can trigger the DAG from the Airflow UI or use the command line:

bash

Copy code

airflow dags trigger pull\_data\_dag

1. **Monitor the Task**: After triggering, you can monitor the execution in the Airflow web UI, checking for logs and results.

**Additional Considerations**

* **Error Handling**: Implement error handling in your pull function to manage issues with API calls or data processing.
* **Data Storage**: Decide how to store the pulled data, whether in a database, file system, or other storage solutions.
* **Dependencies**: You can add more tasks and define dependencies between them if needed.

**3.push operation/task in ariflow dag**

In Apache Airflow, a "push" operation typically refers to the process of sending or storing data to a destination, such as a database, API, or data warehouse, as part of a Directed Acyclic Graph (DAG) task. Here’s how to implement a push operation in an Airflow DAG.

**Steps to Implement a Push Operation in an Airflow DAG**

1. **Set Up Your Environment**: Ensure you have Apache Airflow installed and configured. If you haven't set it up yet, you can do so using:

bash

Copy code

pip install apache-airflow

1. **Create a DAG File**: Create a new Python file in your Airflow DAGs folder (usually located at ~/airflow/dags/).
2. **Define Your DAG**: Here’s a simple example of a DAG that pushes data to a database:

python

Copy code

from airflow import DAG

from airflow.operators.python import PythonOperator

from datetime import datetime

import pandas as pd

from sqlalchemy import create\_engine

# Define your push function

def push\_data\_to\_database():

# Example data to be pushed

data = {

'name': ['Alice', 'Bob', 'Charlie'],

'age': [25, 30, 35],

}

df = pd.DataFrame(data)

# Database connection (update with your database URL)

engine = create\_engine('postgresql://username:password@localhost:5432/mydatabase')

# Push data to the database table

df.to\_sql('my\_table', engine, if\_exists='replace', index=False) # Adjust if\_exists as needed

# Define the DAG

with DAG(

'push\_data\_dag',

schedule\_interval='@daily',

start\_date=datetime(2023, 10, 1),

catchup=False,

) as dag:

# Define the task

push\_task = PythonOperator(

task\_id='push\_data',

python\_callable=push\_data\_to\_database,

)

push\_task

1. **Deploy the DAG**: Place the Python file in your DAGs folder. Airflow will automatically detect the new DAG.
2. **Trigger the DAG**: You can trigger the DAG from the Airflow UI or use the command line:

bash

Copy code

airflow dags trigger push\_data\_dag

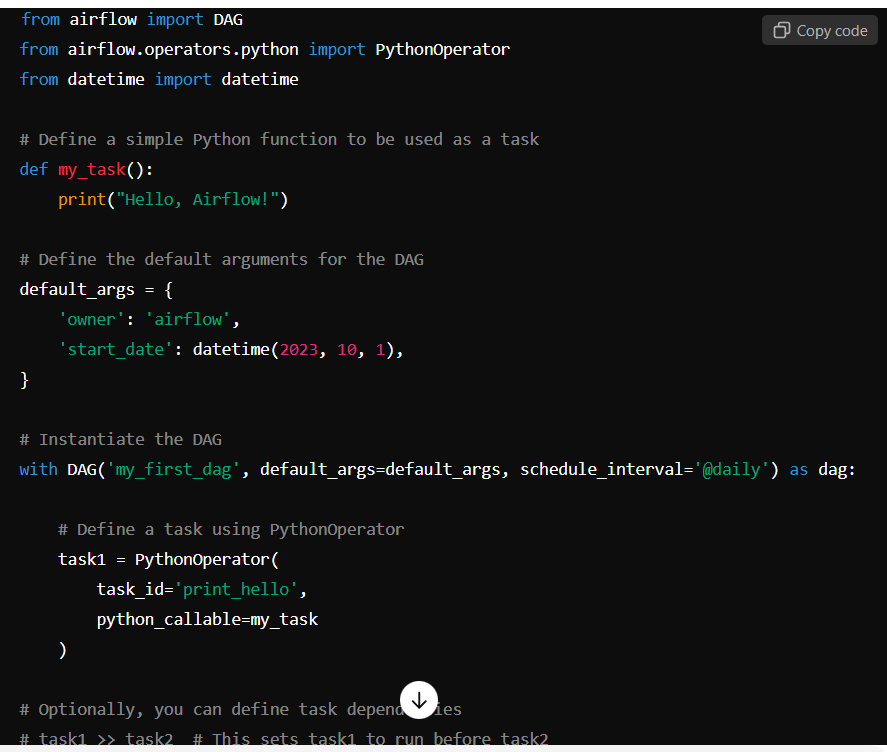
1. **Monitor the Task**: After triggering, monitor the execution in the Airflow web UI to check logs and results.

**Additional Considerations**

* **Database Configuration**: Make sure to replace the connection string with your actual database credentials and adjust the table name and schema as necessary.
* **Error Handling**: Include error handling in your push function to deal with issues that may arise during the data push operation.
* **Data Source**: You might want to integrate a pull operation beforehand to gather data to push, creating a more comprehensive workflow.
* **Task Dependencies**: If your push operation depends on other tasks, you can set dependencies using the >> operator.

**5.python with airflow dag library basic understanding**

Apache Airflow is a powerful tool for orchestrating complex workflows and managing data pipelines. It allows you to define workflows as Directed Acyclic Graphs (DAGs) using Python. Here’s a basic understanding of how to use Airflow with the DAG library.



**Key Concepts**

1. **DAG (Directed Acyclic Graph)**:
   * A DAG is a collection of tasks with defined relationships, meaning tasks can only move forward and never loop back. In Airflow, each workflow is represented as a DAG.
2. **Tasks**:
   * A task is a single unit of work in a DAG. Each task can perform operations like executing a script, running a SQL command, or moving data.
3. **Operators**:
   * Operators are predefined templates for various types of tasks, such as:
     + **PythonOperator**: Executes a Python function.
     + **BashOperator**: Executes a bash command.
     + **DummyOperator**: Used as a placeholder in a DAG.
4. **Scheduling**:
   * You can set up schedules for when your DAG should run using a cron-like syntax.

**Basic Structure of a DAG in Airflow**

Here’s a simple example of a DAG defined using Python and the Airflow library:

python

**Breakdown of the Code**

1. **Imports**:
   * Import necessary classes and functions from Airflow.
2. **Function Definition**:
   * Define a simple function (my\_task) that will be executed by the task.
3. **Default Arguments**:
   * Set default arguments for the DAG, such as the owner and the start date.
4. **DAG Definition**:
   * Create a DAG instance with a unique ID (my\_first\_dag), default arguments, and a schedule interval (e.g., @daily).
5. **Task Creation**:
   * Use the PythonOperator to define a task that calls the my\_task function.
6. **Task Dependencies**:
   * Optionally, define dependencies between tasks using the >> operator.

**Running Your DAG**

1. **Start Airflow**: Use the command line to start the Airflow web server and scheduler.

Copy code

airflow webserver --port 8080

airflow scheduler

1. **Access the Web UI**: Go to http://localhost:8080 to view and manage your DAGs.
2. **Trigger the DAG**: You can manually trigger the DAG from the UI or wait for it to run based on the defined schedule.

**Summary**

This is a basic introduction to using Python with the Apache Airflow DAG library. You can extend this by adding more tasks, managing dependencies, and incorporating different types of operators based on your workflow needs.

**6. main frame understanding**

**Slide 1: Introduction to Mainframes**



* **Definition**: Mainframes are powerful computers used for large-scale data processing and critical applications.
* **Key Industries**: Banking, healthcare, government, and large enterprises.

**Slide 2: Mainframe Architecture**

* **Components**:
  + **CPU**: Central Processing Unit, the brain of the mainframe.
  + **Memory**: Large capacity for handling vast amounts of data.
  + **Storage**: High-speed storage systems for data retention.
  + **I/O Devices**: Input/Output devices for data transfer.

**Slide 3: Mainframe Capabilities**

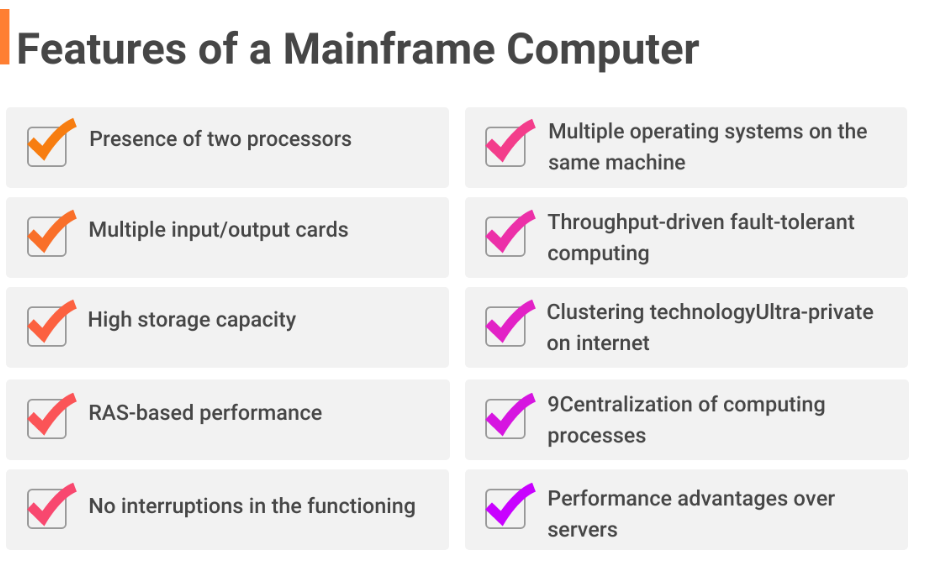
* **High Performance**: Handles billions of transactions in real-time.
* **Multi-user Support**: Supports thousands of users simultaneously.
* **Security**: Robust security features for data protection.
* **Reliability**: Designed for continuous operation without failure.

**Slide 4: Mainframe Applications**

* **Banking**: Transaction processing, account management.
* **Healthcare**: Patient records, billing systems.
* **Government**: Data management, citizen services.

**Advantages of Mainframes**

1. **Reliability**:
   * Minimal downtime and high fault tolerance.
2. **Scalability**:
   * Can handle increasing workloads without sacrificing performance.
3. **Security**:
   * Stronger security measures compared to many other systems.
4. **Cost-Effective for Large Operations**:
   * Economies of scale make them cost-effective for processing large volumes of transactions.



**Disadvantages of Mainframes**

1. **High Initial Cost**:
   * Expensive to purchase and maintain.
2. **Complexity**:
   * Requires specialized knowledge to manage and operate.
3. **Legacy Dependence**:
   * Many applications may be outdated and not easily integrable with modern technologies.
4. **Limited Programming Options**:
   * Primarily supports a few legacy languages, which may deter new development.

Example:

**IBM Mainframes**

1. **IBM zSeries**: A line of mainframe computers designed for high-performance transaction processing.
2. **IBM System z**: The successor to the zSeries, including models like z/OS and z14.
3. **IBM z15**: A recent model that enhances security and cloud capabilities.

**Unisys Mainframes**

1. **Unisys ClearPath**: A series of mainframes that provide a flexible platform for enterprise computing, supporting various operating environments.

**Fujitsu Mainframes**

1. **Fujitsu GS21**: A line of mainframe computers known for their reliability and performance.

**Hitachi Mainframes**

1. **Hitachi VOS3**: A mainframe designed for transaction processing and large-scale data management.

**Other Notable Mentions**

1. **Tandem Computers** (now part of HP): Known for fault-tolerant systems primarily used in banking and telecommunications.
2. **Bull (Atos)**: Offers the **Bullion** line of mainframes, designed for large-scale enterprise applications.

**Summary**

Mainframes play a crucial role in industries that require processing vast amounts of data reliably and securely. While they offer significant advantages, the initial costs and complexity can be barriers for some organizations.