**Airflow - Theoretical Assessment**

**Section A - Basics**

**1. What is Apache Airflow and why is it used?**

Apache Airflow is an open-source platform that helps you create, schedule, and monitor workflows programmatically. Think of it as a way to define complex data pipelines using Directed Acyclic Graphs (DAGs) of tasks. People use Airflow to automate and orchestrate all sorts of jobs, from simple data moves to more involved tasks like training ML models, making sure everything runs reliably, repeatably, and with good visibility.

**2. Define a DAG. What does each part of the acronym stand for?**

A DAG (Directed Acyclic Graph) is basically a collection of all the tasks you want to run, laid out to show how they relate to each other and what depends on what. It's the structured blueprint for your workflow.Here's what the acronym means:**D**irected: Tasks have clear relationships, meaning the flow of execution is defined.**A**cyclic: There are no loops. A task can't depend on itself or a task that comes before it in a circular way, which stops things from running forever.**G**raph: It's a collection of nodes (tasks) connected by edges (dependencies).

**3. Explain the difference between a DAG and a Task.**

A **DAG** is the overall workflow or pipeline - it defines the structure, the dependencies between steps, and when it should run. It's the big picture.A **Task**, on the other hand, is the smallest unit of work inside a DAG. It's a single operation, like running a script, executing a query, or moving a file. A DAG is made up of many of these tasks linked together.

**4. Why should workflows be "Directed Acyclic Graphs" in Airflow?**

Workflows are designed as DAGs in Airflow for some key reasons:

**Directed:** This ensures data and control flow in a predictable, one-way path. Dependencies are clear, and tasks run in the specific order you intend.

**Acyclic:** This guarantees the workflow will finish. If there was a cycle (like Task A needs Task B, which needs Task C, which somehow loops back to needing Task A), the workflow would never end. The acyclic nature means every path eventually terminates.

**Section B - Core Concepts**

**1. Describe the role of the following Airflow components:**

* **Webserver:** This is the user interface (UI) where you can see, trigger, and debug your DAGs and tasks. It's your dashboard for visualizing workflows and checking logs.
* **Scheduler:** This is the engine room of Airflow. It constantly watches all your DAGs and tasks, checks their dependencies and schedules, and kicks off tasks when they're ready to run.
* **Metadata Database:** This is where Airflow keeps track of everything - the status of DAGs, tasks, runs, connections, variables, and more. It's crucial for the scheduler and webserver to know what's going on.

**2. What is the purpose of the airflow db init command?**

The airflow db init command (or more commonly now, airflow db migrate) is used to set up or update the Airflow metadata database. It creates all the necessary tables and structures that Airflow needs to store its state. It's a foundational step when you're first setting up Airflow or when upgrading it.

**3. What is the significance of start\_date and schedule\_interval in a DAG?**

**start\_date:** This is the earliest date from which Airflow will consider running your DAG. It's essential for defining when your workflow should begin and is key for backfilling historical runs.

**schedule\_interval:** This dictates how often your DAG should run. You can set this using cron expressions, time differences (timedelta), or preset options like '@daily' or '@hourly'. It tells the scheduler when to create a new run for the DAG.

**4. What does catchup=False do, and when would you use it?**

Setting catchup=False for a DAG tells Airflow to skip any past DAG runs that might have been missed between the start\_date and the current time. Instead of running all those historical intervals, Airflow will only schedule the \*current\* interval. This is really handy for DAGs that don't need to process historical data or where running past runs would be redundant or costly. For example, a DAG generating a daily summary might only need to run for the latest day, not every single day since its start date.

**Section C - Operators & Execution**

**1. What is an Operator? Give two examples.**

An **Operator** is essentially a template for a single task in Airflow. It defines a specific unit of work. Operators encapsulate actions, whether that's running a Python function, executing a command-line tool, or interacting with a cloud service.Here are two examples:

* **BashOperator:** Used to run Bash commands.
* **PythonOperator:** Used to run Python functions.

**2. How does Airflow handle task failures and retries?**

Airflow has built-in ways to manage task failures:

**Retries:** You can configure tasks to automatically retry if they fail. You can specify how many times to retry (retries), how long to wait between retries (retry\_delay), and even use exponential backoff. If a task fails, Airflow will try it again based on these settings.

**Failure Callbacks:** You can set up custom Python functions that run when a task fails (on\_failure\_callback) or when it's retrying (on\_retry\_callback).**Alerting:** Airflow can be configured to send notifications (like emails) when tasks fail after exhausting all retry attempts.

**3. What is XCom and how is it useful?**

XCom stands for Cross-communication. It's Airflow's way for tasks to pass small bits of data between each other. One task can 'push' some information, and another task can 'pull' it. This is super useful for sending simple messages or results - for instance, a task might push the ID of a record it just created, and a later task could use that ID to do something with that specific record. Just remember, XCom isn't for large amounts of data; for that, you'd want to use persistent storage like S3 or a database.

**4. Explain the difference between BashOperator and PythonOperator.**

**BashOperator:**- Executes a bash command or script in a separate process.- Great for simple shell tasks, file management, or using command-line tools.- Less suited for complex logic or managing state.

**PythonOperator:**- Executes a Python function.- Offers a lot more flexibility for complex logic, conditional operations, and using Python libraries.- Ideal for more involved data processing tasks.

**Section D - Real-World Use**

**1. Give one real-world example where Airflow can be used for ETL.**

**ETL Example:** Daily Sales Data Aggregation Imagine an e-commerce company needing to process daily sales data. Airflow can handle this ETL process:

1. **Extract:** A task (e.g., using `PythonOperator` or `PostgresOperator`) pulls sales transaction data from various places like a live PostgreSQL database or CSV files from S3, covering the previous day's sales.

2. **Transform:** Another task (perhaps using `SparkSubmitOperator` or `PythonOperator`) cleans the data, combines information from different sources, calculates summary metrics, and aggregates key sales figures.3. **Load:** Finally, a task (like `PostgresOperator` or `RedshiftOperator`) loads this processed, aggregated data into a data warehouse table, ready for business intelligence tools.

**2. Why is it recommended to keep DAG scripts lightweight and avoid heavy computations inside them?**

It's best to keep DAG scripts lightweight because:

**Scheduler Performance:** The Airflow scheduler has to read and parse all DAG files. If a DAG file has heavy computations, it can slow down the scheduler, impacting its ability to manage tasks across your entire Airflow instance.

**Task Execution Isolation:** The DAG script should define the workflow structure, but the actual heavy lifting should be done by operators and external systems (like Spark or Kubernetes). This keeps Airflow components focused and allows for scalable task execution without blocking the scheduler or other critical processes.

**Easier Maintenance:** Simpler DAG files are much easier to read, debug, and maintain. Complex logic tucked away in the DAG file can be a headache to troubleshoot.

**3. Why should every DAG have a unique dag\_id ?**

Every DAG needs a unique dag\_id because it's the main identifier Airflow uses to recognize and manage that specific workflow. It's how Airflow:

* Tells different workflows apart.
* Keeps track of the history and status for each DAG run.
* Prevents conflicts, especially in teams where multiple people might be managing DAGs.
* Ensures the scheduler and webserver can load and reference each DAG correctly without confusion.

If two DAGs had the same dag\_id, Airflow wouldn't know which one you meant, leading to errors or unpredictable behavior.

**4. How does Airflow ensure workflows run in the correct order?**

Airflow ensures tasks run in the correct order using **dependencies**. When you define tasks in a DAG, you can specify that one task must complete successfully before another can start. This is usually done with operators like >> (downstream) and << (upstream), or methods like set\_upstream(). The Airflow scheduler then looks at these dependencies and only triggers a task when all its preceding tasks have finished successfully. This guarantees that your workflow executes exactly as you've defined it.