Expanded Data Awareness in Airflow

Authors: Constance Martineau & Tzu-ping Chung

Warning: We are in the process of migrating this document to Confluence here. This will be updated once the migration is completed

Introduction

This document aims to facilitate a formal discussion around a critical evolution within Airflow: The integration of enhanced data awareness. We will follow up with a set of formal AIPs.

Motivation

Airflow has become the standard for orchestrating complex data workflows. However, it operates with limited visibility into the actual data it processes or produces. While it understands task execution order and attributes like operators and parameters in use, it lacks insight into the nature of data inputs and outputs. This link between data and tasks is fundamental to data engineering and is vital for providing insights into the state and health of data as it moves through the workflow. Orchestrators are the heart of data platforms, and if they can understand this link, they can make orchestration decisions based on data quality and freshness, while also providing data engineers with insights about system and data reliability in one place.

The current proposal aims to incentivize users to provide context about processed data when designing pipelines and provide data insights within Airflow. An enhanced understanding of the relationship between tasks and the data that was processed will also allow Airflow to more reliably emit this information to third-party systems via OpenLineage.

Addressing the Knowledge Gap

Primary Concerns of Data Engineers

In the context of data engineering, assets are the data entities that move through pipelines, undergo transformations, and ultimately drive business insights and power data products. Although data engineers spend considerable time building and maintaining data pipelines and are important, they are a means to an end. The data asset itself is what drives business value.

Most data tools like Fivetran, Snowflake, dbt, Monte Carlo, Feast, and Tableau have adopted data assets as their central concept. Airflow, on the other hand, prioritizes task execution and

treats the data asset as a byproduct. This conscious decision has allowed Airflow to be flexible enough to be used for a myriad of use-cases outside of data engineering and is a key factor to Airflow's viral adoption. However, with changes to the data landscape and the advent of Gen AI, stakeholders' access to high quality data is more important than ever, making it critical for data orchestration tools to provide data engineers with actionable insights about the data's state and health as it moves through the workflow.

The current disconnect hampers Airflow's ability to make informed orchestration decisions based on the actual data and creates a gap between task management and data management, resulting in clunky integrations, limited understanding of how data flows from point A to point B, and forces data engineers to translate between the workflow-oriented world of Airflow and the data asset-oriented world of the rest of the data platform.

Progressive Adoptability

Enhancing data awareness in Airflow requires a strategy that supports progressive adoptability to bridge the gap between task-centric and asset-oriented approaches. This strategy is crucial as it is impractical to rewrite existing pipelines entirely. Evolution, not revolution, is the key.

- **Resource and Effort Considerations**: Defining and maintaining detailed data semantics introduces complexity. Users need the flexibility to adopt these features gradually, implementing them as their needs evolve without an overwhelming burden.
- **Legacy Support**: Airflow is widely used, with many critical existing DAGs. Progressive adoptability ensures that enhancements in data awareness can be integrated without disrupting or rendering these workflows obsolete, thus protecting users' investments.
- **Selective Enforcement**: Similar to CI tools with customizable rulesets, Airflow can allow users to incrementally adopt data-awareness features. This approach enables users to apply specific aspects of data awareness to their workflows, aligning with their requirements and maturity levels.
- **Leveraging Existing Practices**: Drawing from software development practices, such as Python type annotations, Airflow can enable users to gradually enhance their DAGs with data-related semantics and validations, improving data handling and pipeline reliability at their own pace.

By facilitating a smooth, incremental transition, Airflow can address the gap effectively, allowing it to evolve into a more data-aware platform while ensuring continuity and minimizing disruption.

Handling Incremental Load Strategies

A critical aspect of enhancing data awareness in Airflow involves improving how the framework manages incremental jobs and data processing strategies – critical for executing backfills effectively and ascertaining data freshness. Although the existing model is flexible with logical dates and data intervals that are calculated based on the DAG schedule, there is a longstanding need for first-class support for incremental processes within Airflow. This approach should

incorporate other strategies that are commonly used, such as leveraging watermarks or processing data for rolling windows. Establishing proper methodology for incrementally loading data and re-processing data chunks ensures reliability, performance, and lays the groundwork for executing “intelligent” backfills.

Drawing inspiration from DBT, Airflow could offer alternative methods for executing and tracking incremental job runs:

- **Full-refresh**: Rebuilds the dataset at every run. This is suitable for small environments or in the early stages of a DAG.
- **Every Instance**: Each task execution is independent and can be run in parallel, potentially with different parameters. This is useful for ML experimentation.
- **Only-Run-Latest/Catch-up**: Executes only the most recent task instance to update the dataset, streamlining operations and focusing resources on the most current data. This aligns well with the high-watermark strategy and may be appropriate for event-driven workloads.
- **Date Range**: Executes tasks based on a specified date or time range and would include windows, allowing for targeted updates of a dataset. This method is similar to leveraging data intervals today.
- **Segments**: Executes tasks based on segments such as region, team, or other categories. This approach allows for targeted processing of specific segments, enabling efficient handling of large datasets divided by logical partitions.

Incorporating these strategies requires Airflow to have a nuanced understanding of the data it manages, specifically identifying which blocks of a dataset need updates. This enhancement promises to streamline workflow execution and align Airflow more closely with modern data engineering practices, facilitating a more intuitive and efficient handling of incremental data processing.

Proposed Enhancements

Introducing Assets

⚠️ All code examples in this document may evolve and is subject to change.

A data asset is, fundamentally, a collection of logically related data. This can be one of (but not limited to):

- A table in a relational database
- An persisted ML model
- An embedded dashboard or report
- A bunch of tables grouped by a naming convention sharing a near-identical structure
- A directory containing CSVs

Assets are **written** at some point. From a programmatic standpoint, we will define assets as entities within a function that are generated when that function is called and executed. Assets can be created as one-off events, but more commonly are updated somewhat periodically, either indefinitely or until a set point of time. Each write can either:

- Replace data in its entirety (for example by deleting and recreating a table)
- Incrementally append data to an existing asset (for example by inserting or upserting DB records into an existing table or by creating a new file in a directory), or
- Publish a new iteration of the asset (for example by publishing a new ML model version or creating and sending a new iteration of regulatory report).

While it's not necessary for a pipeline to understand this distinction, similar to how version control systems consider diffs, it is important for pipelines to understand what each task execution changes within an asset in order for Airflow to highlight impacts on other dependent assets and recreating those writes retroactively (i.e backfilling, discussed later).

Rename `Datasets` to `Assets`

Taking inspiration from other tooling like Great Expectations, Atlan and Dagster, we propose to rename *Datasets* to *Assets*, and potentially introduce subtypes. The term "asset" is more generalizable, and can be used to represent other data products like serialized ML models.

- This opens the door to us adding different types of assets beyond datasets, like ML models. ML practitioners use datasets AND serialized models as part of their ML pipelines, and while there is nothing that is stopping ML practitioners from representing both as "datasets" in Airflow today, it is uncomfortable to refer to models as datasets when they are distinct concept, and makes it hard to distinguish between tasks updating ML models and tasks updating datasets in the UI and in the code.
- We can add specific attributes for different asset types. For example, most datasets are semi-structured and in a tabular format, so it is reasonable to assume that datasets have schemas with column names and types.

Asset Annotations

We propose to formalize support for manually annotating data inputs and outputs via task inlets and outlets and showcase this relationship in the UI, allowing for the ability to progressively adopt the features. Inlets and outlets are used by OpenLineage today to allow operators to track lineage. By setting data inputs as task inlets and data outputs as task outlets, we ensure that each task's data dependencies and contributions are clearly articulated, establishing a contract for data flow within the workflow.

Using Asset Annotations, you could then do something like this:

Python

```
raw_sales = Asset('raw_sales', uri='s3://...') # URI is optional
aggregated_sales = Asset(
    'aggregated_sales',
    uri='snowflake://...',
    column=['sales'],
)

task_aggregate_sales = SqlOperator(
    inlets=[raw_sales],
    outlets=[aggregated_sales],
    sql = """
        INSERT INTO {{ outlets.aggregated_sales }}
        SELECT SUM({{ inlets.raw_sales.sales }})
        FROM {{ inlets.raw_sales }}
    """,
)
```

New Asset-Centric Syntax

Asset Function

If creating new workflows for tools that are asset-centric, we propose extending the Taskflow API to include primitives that are asset-centric. In addition to Asset Annotations, Assets can be defined in a file by a decorated python function:

Python

```
# at is optional. human readable name defaults to function
# name if none is given
@asset(at="s3://aws_conn_id@bucket/bus_trips.parquet")
def bus_trips():
    # Write bus trips data to asset...
```

The asset would be at the same level as a DAG, and would be defined by the function. Unlike `@tasks` and `@dags`, you would not need to call it.

The `at` argument specifies the asset's location, similar to adding a Dataset to `outlets` today. It allows simply passing in a plain URI string or an `ObjectStoragePath` to be automatically coerced.

Multi-Assets

There are cases where the same function may generate multiple assets, such as when generating training and testing datasets for ML. This isn't necessary for most situations, but is a valid use case (like an upstream asset needing to be split into two). To accommodate, we will allow users to pass in parameters for multiple Assets. Some parameters will be unique per asset and some, like schedules or partitions (see below) will be shared.

Python

```
@asset.multi(
    outlets=[
        {"name":"bus_trip_1","at":"s3://../bus_trip_1.parquet"},
        {"name":"bus_trip_2","at":"s3://../bus_trip_2.parquet"},
    ],
)
def split_trips():
    ...
```

Asset Dependencies

Upstream assets can be set as function arguments for the decorated function.

Python

```
@asset(at="s3://aws_conn_id@bucket/raw_sales.parquet")
def raw_sales():
    # Write raw sales data to asset...

@asset(at="snowflake://...")
def aggregated_sales(raw_sales):
    # Write aggregated sales data to asset...
```

All assets in an Airflow deployment share one single namespace. This is also the same namespace for DAGs to avoid confusion. Different assets can reference the same URI to support the use case of writing to the same data target from two functions—not a best but unfortunately common practice.

Asset Validations

To increase data reliability, we propose adding capabilities for data consumers and data producers to define and verify expectations around input and output assets via Operators and/or Asset definitions. Data Engineers can state things like expected column names and data types, update frequency, or even what the contents of a field should look like, allowing them to define their expectations.

There would be two types of validations:

- Pre-runtime validations for data consumers to verify that established data flow contracts are valid
- Post-runtime validations for data producers that publicize that the data that was processed upholds an established contract

As data consumers, you would be able to define pre-runtime validation rules that are executed prior to the task, such as for example checks to ensure that certain columns exist. If discrepancies are found, Airflow could send a notification, preventing potential runtime errors and removing the need to perform cleanup actions. These could be set as part of the task definition (“inlet_validations” in the example) or as part of setting asset dependencies if using an asset-centric syntax.

As data producers, you could define validation rules that “certify” that the data was processed correctly, for example checks to ensure that values in a specific column follow a pattern, like phone numbers. This could be set as part of the actual asset definition.

Lastly, we should have built-in validations that are accessible via the asset and also allow for custom user-defined validations.

Python

```
raw_sales = Asset(
    'raw_sales',
    uri='s3://...',
    outlet_validations=[AssetValidation.has_columns('sales')]
    column=['sales'],
)
aggregated_sales = Asset(
    'aggregated_sales',
    uri='snowflake://...',
    column=[...],
    type='dataset',
)
```

```
task_aggregate_sales = SqlOperator(
    inlets=[raw_sales],
    outlets=[aggregated_sales],
    inlet_validations=AssetValidation.has_columns(
        raw_sales.sales
    ),
    sql = """
        INSERT INTO {{ outlets.aggregated_sales }}
        SELECT SUM({{ inlets.raw_sales.sales}})
        FROM {{ inlets.raw_sales }}
    """,
)
```

Asset Partitions and Schedules

This section is meant to be high-level and introduce some core concepts. The actual AIPs will go in much more detail.

As part of this proposal, we would like to decouple the concept of when an asset is scheduled, and the slice of data that was processed (data interval today). Using a simple example, an asset may be scheduled to be generated once a day, but includes data for the last three days.

Today, Airflow calculates the data interval based on the schedule and assumes that the data interval or partition is the time in between runs. Users may leverage the data interval or logical dates to calculate the proper data interval range, but that makes it difficult to highlight the impact on other downstream assets whenever something has been reprocessed. Furthermore, this only works for time-based intervals. We will overcome this by introducing the concept of a partition.

Schedules

Time and Asset-based Scheduling

The schedule of an asset denotes when the asset is written to. If the writing is done by a process managed by the pipeline, the schedule reflects when the pipeline kicks off the process to do the writing. Similar to today, you will be able to set time-based schedules and/or schedule assets to be generated whenever upstream assets are generated.

```python
@asset(
    start_date=datetime(1976, 6, 4),  # When writes start. (Optional)
    schedule="@yearly",  # When should a write happen.
)
def name():
    # Code to concretely write data to this asset...
```

Future Work: Scheduling Tolerations and Thresholds

Some upstream assets may be more important than others. While ideally it may be best to wait until all upstream dependencies have been met before generating an asset, there are cases where it is better to generate assets using what is available vs waiting, for example when generating regulatory reports that are fined when late. In addition, there are also cases where it's acceptable to generate an asset provided the freshness is within a certain threshold, for example leveraging an ML model that has been trained within the last month. We propose to introduce the concept of thresholds and tolerations to account for these types of use-cases. These can be used by data consumers as needed, and may be different across dependent assets.

One example use case is an asset that depends on another asset that is written daily, but sometimes a run is missed. You can mark the downstream asset's tolerance to *daily* so a materialization is triggered whenever the upstream is materialized, *or when a day has passed without the upstream being updated*. There are more than one way to do this, and the feature itself is not considered essential, so we are delaying this further into the future until the basic features are done.

Partitions

Understanding Partitions

At a high-level, partitions are "slices" of data assets that can be tracked and computed independently. They typically correspond to separate files or slices of tables, and are especially useful for modeling data that is appended or modified when running incremental load processes.

The idea behind partitions is to provide a way to model and organize data that is closely related, but still distinct, and decouple it from the task. For example, a daily-partitioned table for sales orders could be seen as a collection of partitions that correspond to a sales order data asset, with each partition corresponding to sales orders that were made on a particular day. These partitions are computed using the same code and derived from the same upstream sources. They can be processed and visualized in bulk, and also individually or as a subset.

Partitions in Airflow are meant to be a logical concept and don't necessarily need to correspond to a partition at the storage layer. Some systems like Hive have an explicit notion of partitions with table partitions corresponding to a subdirectory in the directory where the table is stored, whereas other systems like Snowflake abstract storage away entirely. Even without a direct link, the concept is incredibly useful for data orchestration. When you overwrite data in a Snowflake table for a specific date range, you're deleting all the rows that have that date and then inserting a new set of rows for that date — i.e working with a logical partition.

While Airflow has a basic notion of logical partitioning today via the grid view and ability to parameterize DAG runs with a `logical_date`, partitions are fundamentally a property of the data, not the task. Airflow's current implementation works well in certain cases, but comes with constraints:

- It's automatically calculated based on the schedule and requires for all data updated by the DAG to be partitioned in the same way.
- It requires for partitions to be one-to-one with task runs.

For data flows that don't fit these parameters, our current implementation can actually increase confusion by misrepresenting what's going on.

A partition describes how an asset is partially generated whenever its function executes. The function itself does not strictly require this information — the asset's function body can technically do whatever it wants and is inherently the canonical source of truth to what gets written — but there are clear benefits to using them:

- The partition tells other parts of the system how an asset is generated, **without needing to analyze data outputs**. If we know that the raw sales order asset uses hourly partitions, and we know that the daily sales order asset is dependent on the raw sales order asset, then we also know that 24 partitions from the raw sales order asset that were generated on a specific day correspond to 1 partition in the daily sales order asset for that same day. We can more granularly highlight the flow of data in the UI, and showcase how issues impact downstream assets.
- A single task can generate a range of partitions ad-hoc, for example, as part of backfills. Let's say an asset is scheduled to run once a day and normally processes a day's worth of data. Instead of launching one task per day in a date range, it may be more performant to delegate the parallelization to the compute engine like Spark and kick off a single task that covers multiple partitions. If using partitions, we can provide this capability.

Using Partitions

We will introduce two categories of partitions:

1. Time-based partitions: These are similar to data intervals today, and are meant to represent time or date ranges. Like today, they can automatically be derived from the schedule or can be manually set.

```python
@asset(
    ...,
    schedule="@hourly",
    # defining partition for illustrative purposes. default is 'auto'
    # if 'auto' and schedule is time-based, will derive partition from schedule
    # if 'auto' and schedule is not timebased, partition will resolve to none
    # @hourly is shorthand for PartitionByInterval("@hourly")
    partition="@hourly",
)
def hourly_data():
    ...
```

2. Segment-based partitions: These are meant to represent other types of logical groupings, like teams or cloud regions. For example, if an asset is generated every day to evaluate the spend of multiple data warehouses, each data warehouse can be represented as a partition.

```python
@asset(
    ...,
    schedule="@daily",
    partition=PartitionBySequence(["marketing-dwh", "engineering-dwh"]),
)
def dwh_daily_cloud_spend():
    ...
```

Similar to logical dates and data intervals, partitions are parameterizable and will be available via the task context.

```python
@asset(...partition=PartitionBySequence(["marketing-dwh", "engineering-dwh"]))
def dwh_daily_cloud_spend():
    context = get_current_context()
    # Accesses partition information. Can retrieve the original definition,
    # the current and past partitions, etc. (Not fully hashed out yet.)
```

```
    dwh_partitions = context["partitions"]
    for dwh_partition in dwh_partions:
        # do something
```

Partitions can be static or dynamic. When partitions are known up front, they will be declared as part of the asset definition, as shown in the two previous examples. Sometimes partitions are not known until runtime. For example, to predict energy consumption, an energy company trains an ML model per customer. Customers are frequently added and removed, so it is simpler to get an accurate list of customers at runtime.

Python
```
active_customers = PartitionAtRuntime(name="active_customers")

@asset(schedule="@hourly", partition=active_customers)
def energy_consumption_model():
    """get customer list and train model"""
    customers = ... # get customer list
    for customer in customers:
        train_model = ...
        active_customers.add_partition(key=customer)
```

You will also be able to combine time-based and segment-based partitions.

Python
```
@asset(
    schedule=dwh_daily_cloud_spend,
    partition=PartitionByProduct(
        ["@hourly", PartitionBySequence(["marketing-dwh", "engineering-dwh"]
    ),
)
def dwh_spend_analysis():
    ...
```

After the upstream asset is generated each day, 48 (24 hours x 2 data warehouses) runs are triggered. To simplify logic, a combined partition definition can only contain *at most one*

time-based partition definition. We'll explore more complex combinations if there are concrete real-world use cases after this feature is fully implemented and rolled out.