Who am I?

- Airflow PMC
- co-creator of the K8sExecutor
- Strategy Engineer @ Astronomer.io
- Excited to be in Australia (virtually!)
The Team

Tatiana Al-Chueyr
Kaxil Naik
Utkarsh Sharma
Mike Shwe
Vikram Koka
What is DAG Authoring?
What is DAG Authoring?

- Airflow = engine, DAG authoring = chassis
- Airflow 2.0: Massive improvements to the engine of Airflow (scheduler HA, improved k8sexec, etc.)
- “There’s a lot here for the Airflow admin, but what about for the DAG writer?”
Why is DAG authoring difficult?

- Current DAG writing involves obscure Airflow knowledge + python expertise → lengthy onboarding, challenging maintenance

- DAG author needs to keep track of how data is passed between tasks- more difficult to write and debug, especially collaboratively
Why is DAG Authoring Difficult?

```python
from airflow.providers.snowflake.operators.snowflake import SnowflakeOperator
from airflow.providers.snowflake.transfers.s3_to_snowflake import S3ToSnowflakeOperator

create_table = SnowflakeOperator(
    task_id="create_table",
    sql=f"""CREATE OR REPLACE TABLE {SNOWFLAKE_ORDERS} 
    (order_id char(10), customer_id char(10), purchase_date DATE, amount FLOAT)""",
    snowflake_conn_id=SNOWFLAKE_CONN_ID,
)

extract_data = S3ToSnowflakeOperator(
    task_id='extract_data',
    s3_keys=['orders_data.csv'],
    snowflake_conn_id=SNOWFLAKE_CONN_ID,
    stage=SNOWFLAKE_STAGE,
    table=SNOWFLAKE_ORDERS,
    file_format="(type = 'CSV', field_delimiter = ',')",
)
create_table >> extract_data
```
Attempt 1: Taskflow API

- simplified python task writing
- didn’t really do much in terms of data awareness
- Only works with python tasks
- Still dependent on XCom
- Ultimately you still need traditional operators

```python
from airflow.decorators import task

task
def count_url(url):
    import pandas as pd
    c = pd.read_csv(url)
    return c.count()
```
extract_data = S3ToSnowflakeOperator(
    task_id='extract_data',
    s3_keys=['orders_data.csv'],
    snowflake_conn_id=SNOWFLAKE_CONN_ID,
    stage=SNOWFLAKE_STAGE,
    table=SNOWFLAKE_ORDERS,
    file_format=('type = 'CSV',field_delimiter = ','),
)

from airflow.decorators import task
from airflow.providers.amazon.aws.hooks.s3 import S3Hook
from airflow.providers.snowflake.hooks.snowflake import SnowflakeHook
import io
import requests

@task
def s3_to_snowflake(key, bucket_name):
    snow_hook = SnowflakeHook(
        snowflake_conn_id=SNOWFLAKE_CONN_ID,
    )
    snow_hook.run(
        "COPY INTO testtable FROM s3://<s3_bucket>/data/
        STORAGE_INTEGRATION = myint
        FILE_FORMAT=(field_delimiter='),'"
        .format(aws_access_key_id=AWS_ACCESS_KEY_ID,
            aws_secret_access_key=AWS_SECRET_ACCESS_KEY))

with dag:
    loaded_data = s3_to_snowflake(KEY, BUCKET_NAME)
So... we started from scratch
Rewriting the Dag Authoring Story

- Airflow DAG code should be almost indistinguishable from standard python
- Data Engineers should be able to treat SQL tables as first-class citizens in their python environment
- Moving data between SQL databases and python environments should be seamless
- Airflow users should be able to run the same DAG across different flavors of SQL, datastores, and data warehouses
Introducing Astro Python SDK

DAG authoring for data engineers reinvented:

Write self-documenting, Pythonic code

Move data between relational stores and python data structures without temp tables

Validate your data with built-in operators

Benefits: 50% fewer lines of code, simplified maintenance
When to use Astro Python SDK

For data engineering teams who are:

• Writing new pipelines
• Re-factoring existing pipelines
• Creating DAGs with multiple-object stores or databases
• Augmenting their pipeline authoring team with data engineers lacking detailed Airflow knowledge and Python expertise
Let’s take a look!
ETL conceptual steps

A very simple, common ETL workflow

- **Extract** (from S3 to Snowflake)
- **Filter** data in Snowflake
- **Join** filtered data with another Snowflake table
- **Load**: Merge joined data into a reporting table in Snowflake
ETL with standard Airflow

Use temp tables to move data from one place to another

**Extract** (from S3 to Snowflake)
- Create temp table

**Transform**
- **Filter** data in Snowflake
  - Create 2nd temp table for filtered results

- **Join** filtered data with another Snowflake table
  - Create 3rd temp table for joined results

**Load: Merge**
- joined data into a reporting table in Snowflake
  - Hand-craft a database-specific merge statement on 3rd temp table

**Extract** (from S3 to Snowflake)
  Create temp table

**Filter** data in Snowflake
  Create 2nd temp table for filtered results

**Join** filtered data with another Snowflake table
  Create 3rd temp table for joined results

**Load: Merge**
  joined data into a reporting table in Snowflake
  Hand-craft a database-specific merge statement on 3rd temp table
ETL with the Astro Python SDK

Extract (from S3 to Snowflake) → Filter data in Snowflake → Join filtered data with another Snowflake table → Load: Merge joined data into a reporting table in Snowflake

(Astro SDK does all this extra work for you! Focus on the business logic, not the mechanisms to implement the logic)
**Extract**

- Extract (from S3 to Snowflake)
- Transform:
  - Filter data in Snowflake
  - Join filtered data with another Snowflake table
- Load: Merge joined data into a reporting table in Snowflake
Extract

`load_file` is the same for GCS→BQ as it is for S3→Snowflake, but:

### GCS→BQ

```python
eject_data = gcs_to_bq.GoogleCloudStorageToBigQueryOperator(
    task_id='extract_data',
    bucket='BQ_BUCKET',
    source_objects=['bigquery/orders_data.csv'],
    destination_project_dataset_table='BQ_TABLE',
    schema_fields=[
        {'name': 'order_id', 'type': 'STRING', 'mode': 'NULLABLE'},
        {'name': 'customer_id', 'type': 'STRING', 'mode': 'NULLABLE'},
        {'name': 'purchase_date', 'type': 'DATE', 'mode': 'NULLABLE'},
        {'name': 'amount', 'type': 'NUMERIC', 'mode': 'NULLABLE'},
    ],
    write_disposition='WRITE_TRUNCATE',
    dag=dag)
```

### S3→Snowflake

```python
create_table = SnowflakeOperator(
    task_id = "create_table",
    sql = f""
      CREATE OR REPLACE TABLE {SNOWFLAKE_ORDERS} 
      (order_id char(10), customer_id char(10), purchase_date DATE, amount FLOAT)"
    ,
    snowflake_conn_id = SNOWFLAKE_CONN_ID,
)

eject_data = S3ToSnowflakeOperator(
    task_id = 'extract_data',
    s3_keys = ['orders_data.csv'],
    snowflake_conn_id=SNOWFLAKE_CONN_ID,
    stage = SNOWFLAKE_STAGE,
    table = SNOWFLAKE_ORDERS,
    file_format = "(type = 'CSV',field_delimiter = ',')"),
)
create_table >> eject_data
```
Extract

Simplified with a new load_file operator

```python
orders_data = aql.load_file(
    file = File('s3://.../orders_data_header.csv',S3_CONN_ID),
    output_table=Table(conn_id=SNOWFLAKE_CONN_ID)
)

orders_data = aql.load_file(
    file = File('gs://.../orders_data_header.csv',GCS_CONN_ID),
    output_table=Table(conn_id=BQ_CONN_ID)
)
```
Extract

Simplified with a new load_file operator

### Standard Airflow

```python
create_table = SnowflakeOperator(
    task_id = "create_table",
    sql = """CREATE OR REPLACE TABLE {SNOWFLAKE_ORDERS} (order_id char(10),customer_id char(10), purchase_date DATE, amount FLOAT)"
     ,
    snowflake_conn_id = SNOWFLAKE_CONN_ID,
)

extract_data = S3ToSnowflakeOperator(
    task_id = 'extract_data',
    s3_keys = ['orders_data.csv'],
    snowflake_conn_id=SNOWFLAKE_CONN_ID,
    stage = SNOWFLAKE_STAGE,
    table = SNOWFLAKE_ORDERS,
    file_format = "(type = 'CSV',field_delimiter = ',')",
)

create_table >> extract_data
```

### Astro Python SDK

```python
orders_data = aql.load_file(
    file = File('/orders_data_header.csv',S3_CONN_ID),
    output_table=Table(conn_id=SNOWFLAKE_CONN_ID)
)
```

- **Create temp table**
- **Needs a staging table**
- **Fill the temp table**
- **Specify task dependencies**
Extract

Simplified with a new, datastore- and database-agnostic load_file operator

Standard Airflow

```python
create_table = SnowflakeOperator(  
    task_id = "create_table",  
    sql = f"""CREATE OR REPLACE TABLE {SNOWFLAKE_ORDERS}  
        (order_id char(10),customer_id char(10), purchase_date DATE, amount  
        FLOAT)""'',  
    snowflake_conn_id = SNOWFLAKE_CONN_ID,
)

extract_data = S3ToSnowflakeOperator(  
    task_id = "extract_data",  
    s3_keys = ["orders_data.csv"],  
    snowflake_conn_id=SNOWFLAKE_CONN_ID,  
    stage = SNOWFLAKE_STAGE,  
    table = SNOWFLAKE_ORDERS,  
    file_format = "(type = 'CSV',field_delimiter = ',')",  
)

create_table >> extract_data
```

Astro Python SDK

```python
orders_data = aql.load_file(  
    file = File("/orders_data_header.csv",S3_CONN_ID),  
    output_table=Table(conn_id=SNOWFLAKE_CONN_ID)
)
```
Transform

**Extract** (from S3 to Snowflake)

**Filter** data in Snowflake

**Join** filtered data with another Snowflake table

**Load**: Merge joined data into a reporting table in Snowflake
Transform

Simplified with the new transform operator

Standard Airflow

```python
# Assumes there's already a populated SNOWFLAKE_CUSTOMERS table
join_data = SnowflakeOperator(
    task_id = "join_data",
    sql = f"""CREATE OR REPLACE TABLE {SNOWFLAKE_JOINED} AS
      (SELECT c.customer_id, customer_name, order_id, purchase_date, amount, type FROM
       {SNOWFLAKE_FILTERED_ORDERS} fo JOIN {SNOWFLAKE_CUSTOMERS} c
       ON fo.customer_id = c.customer_id)""
)

create_table >> extract_data >> filter_data >> join_data
```

Astro Python SDK

```python
@aqi.transform
def filter_orders (input_table: Table):
    return "SELECT * FROM {{input_table}} WHERE amount > 150"

@aqi.transform
def join_orders_customers (filtered_orders_table: Table, customers_table : Table):
    return """SELECT c.customer_id, customer_name, order_id, purchase_date, amount, type FROM {{filtered_orders_table}} fo JOIN {{customers_table}} c ON fo.customer_id = c.customer_id""

with dag:
    customers_table=Table(name=SNOWFLAKE_CUSTOMERS,conn_id=SNOWFLAKE_CONN_ID)
    filtered_data = filter_orders(orders_data)
    joined_data = join_orders_customers(filtered_data, customers_table)
```

Simplified with the new `transform` operator
Transform

Simplified with the new transform operator

Standard Airflow

```python
def filter_orders(input_table: Table):
    return "SELECT * FROM {{input_table}} WHERE amount > 150"
```

```sql
CREATE OR REPLACE TABLE {SNOWFLAKE_FILTERED_ORDERS} AS
(SELECT * FROM {SNOWFLAKE_ORDERS} WHERE amount > 150),
snowflake_conn_id = SNOWFLAKE_CONN_ID,
```

create_table >> extract_data >> filter_data >> join_data

Create temp table to hold filtered orders

Create temp table to hold joined results

Specify task dependencies

Astro Python SDK

```python
@aql.transform
def filter_orders (input_table: Table):
    return "SELECT * FROM {{input_table}} WHERE amount > 150"
```

```python
@aql.transform
def join_orders_customers (filtered_orders_table: Table, customers_table : Table):
    return "SELECT c.customer_id, customer_name, order_id, purchase_date,
amount, type FROM {{filtered_orders_table}} fo JOIN {{customers_table}} c ON
fo.customer_id = c.customer_id"
```

```python
with dag:
    customers_table=Table(name=SNOWFLAKE_CUSTOMERS,conn_id=SNOWFLAKE_CONN_ID)
    filtered_data = filter_orders(orders_data)
    joined_data = join_orders_customers(filtered_data, customers_table)
```

# assumes there’s already a populated SNOWFLAKE_CUSTOMERS table

join_data = SnowflakeOperator(
    task_id = "join_data",
    sql = f"""CREATE OR REPLACE TABLE {SNOWFLAKE_JOINED} AS
(SELECT c.customer_id, customer_name, order_id, purchase_date,
amount, type FROM
{SNOWFLAKE_FILTERED_ORDERS} fo JOIN {SNOWFLAKE_CUSTOMERS} c
ON fo.customer_id = c.customer_id)""
)
```
Transform
Simplified with the new transform operator

Standard Airflow

```
filter_data = SnowflakeOperator(
    task_id = "filter_data",
    sql = f"""CREATE OR REPLACE TABLE {SNOWFLAKE_FILTERED_ORDERS} AS
         (SELECT * FROM {SNOWFLAKE_ORDERS} WHERE amount > 150)""",
    snowflake_conn_id = SNOWFLAKE_CONN_ID)
```

Create temp table to hold filtered orders

```
join_data = SnowflakeOperator(
    task_id = "join_data",
    sql = f"""CREATE OR REPLACE TABLE {SNOWFLAKE_JOINED} AS
         (SELECT c.customer_id, customer_name, order_id, purchase_date,
          amount, type FROM {SNOWFLAKE_FILTERED_ORDERS} fo JOIN {SNOWFLAKE_CUSTOMERS} c
          ON fo.customer_id = c.customer_id)"""
)
```

Create temp table to hold joined results

```
create_table >> extract_data >> filter_data >> join_data
```

Specify task dependencies

Astro Python SDK

```
@aql.transform
def filter_orders (input_table: Table):
    return "SELECT * FROM {{input_table}} WHERE amount > 150"

@aql.transform
def join_orders_customers (filtered_orders_table: Table, customers_table : Table):
    return """SELECT c.customer_id, customer_name, order_id, purchase_date,
    amount, type FROM {{filtered_orders_table}} fo JOIN {{customers_table}} c ON
    fo.customer_id = c.customer_id"""

with dag:
customers_table=Table(name=SNOWFLAKE_CUSTOMERS,conn_id=SNOWFLAKE_CONN_ID)
filtered_data = filter_orders(orders_data)
joined_data = join_orders_customers(filtered_data, customers_table)
```

Inputs and outputs are Table objects
from my.library import orders_data, filter_orders, join_orders_customers

customers_table=Table(name=SNOWFLAKE_CUSTOMERS, conn_id=SNOWFLAKE_CONN_ID)

with dag:
    joined_data = join_orders_customers(filter_orders(orders_data), customers_table=customers_table)

@aql.transform
def filter_orders(input_table: Table):
    return "SELECT * FROM {{input_table}} WHERE amount > 150"

@aql.transform
def join_orders_customers(filtered_orders_table: Table, customers_table : Table):
    return """SELECT c.customer_id, customer_name, order_id, purchase_date, amount, type FROM {{filtered_orders_table}} fo JOIN {{customers_table}} c ON fo.customer_id = c.customer_id"""
Load

Extract (from S3 to Snowflake) → Transform → Load: Merge joined data into a reporting table in Snowflake

Filter data in Snowflake → Join filtered data with another Snowflake table
merge_data = SnowflakeOperator(
    task_id="merge_data",
    sql=f"""MERGE INTO {SNOWFLAKE_REPORTING} r using {SNOWFLAKE_JOINED} j
        ON r.order_id = j.order_id WHEN MATCHED THEN
            UPDATE SET r.customer_id = j.customer_id, r.customer_name = j.customer_name"
"""
)

Merge

Postgres

```sql
INSERT INTO {main_table} ("list","sell","taxes")
SELECT "list","sell","taxes" FROM {merge_table}
ON CONFLICT ("list","sell")
DO UPDATE SET
"list"=EXCLUDED."list","sell"=EXCLUDED."sell","taxes"=EXCLUDED."taxes"
```

Sqlite

```sql
INSERT INTO {main_table} (list,sell,taxes)
SELECT list,sell,taxes FROM {merge_table}
WHERE true
ON CONFLICT (list,sell)
DO UPDATE SET
list=EXCLUDED.list,sell=EXCLUDED.sell,taxes=EXCLUDED.taxes
```

Snowflake

```sql
merge into {{main_table}} using {{merge_table}} on
Identifier(taxes)=Identifier(taxes) AND
Identifier(age)=Identifier(age)
when matched then
  UPDATE SET {{main_table}}.list=merge_table.list,
  {{main_table}}.sell=merge_table.sell,
  {{main_table}}.taxes=merge_table.taxes
when not matched then
  insert({{main_table}}.list,{{main_table}}.sell,{{main_table}}.taxes)
  values (merge_table.list,merge_table.sell,merge_table.taxes)
```

BigQuery

```sql
MERGE {table} T USING {merge_table} S
ON T.list= S.list AND T.sell= S.sell
WHEN NOT MATCHED BY TARGET THEN
  INSERT (list,sell,taxes) VALUES (list,sell,taxes)
  WHEN MATCHED THEN
    UPDATE SET T.list=S.list, T.sell=S.sell, T.taxes=S.taxes
```
Merge
Simplified with a database agnostic merge operator

**Standard Airflow**

```python
merge_data = SnowflakeOperator(
    task_id="merge_data",
    sql=""""MERGE INTO {SNOWFLAKE_REPORTING} r using {SNOWFLAKE_JOINED} j
        ON r.order_id = j.order_id WHEN MATCHED THEN
            UPDATE SET r.customer_id = j.customer_id, r.customer_name =
            j.customer_name"
)"
```

**Astro Python SDK**

```python
aql.merge(target_table =
    Table(name=SNOWFLAKE_REPORTING, conn_id=SNOWFLAKE_CONN_ID),
    merge_table=joined_data,
    merge_columns=["customer_id", "customer_name"],
    target_columns=["customer_id", "customer_name"],
    merge_keys=["order_id", "order_id"],
    conflict_strategy="update")
```

**MERGE syntax is database-specific**

**MERGE syntax is database-agnostic**
But What About Python?
ETL with only SQL
A very simple, common ETL workflow

Extract (from S3 to Snowflake) → Filter data in Snowflake → Join filtered data with another Snowflake table → Load: Merge joined data into a reporting table in Snowflake
ETL with SQL and Dataframes

A slightly less simple, common ETL workflow

- **Extract** (from S3 to Snowflake)
- **Extract** (from S3 to pandas)

**Transform**

- **Filter** data in Snowflake
- **Analyze** relevant features using pandas/numpy
- **Join** filtered data with result of dataframe function

**Load:** Merge joined data into a reporting table in Snowflake
Moving Data Between Python and SQL

Traditional way: APIs, hooks, etc.

```python
from airflow.decorators import task
from airflow.providers.snowflake.hooks.snowflake import SnowflakeHook
@task
def add_one_to_column(conn_id: str, warehouse: str, table_name: str, output_table_name: str):
    table_hook = SnowflakeHook(
        snowflake_conn_id=conn_id,
        warehouse=warehouse,
        database=database,
        role=role,
        schema=schema,
        authenticator=authenticator,
        session_parameters=session_parameters,
    )
    table_df = table_hook.get_pandas_df(f"SELECT * FROM {table_name}"
    table_df["column_name"] = table_df["column_name"] + 1
    table_df.to_sql(
        name=output_table_name,
        con=table_hook.get_sqlalchemy_engine().connect(),
    )
```
The Dataframe Decorator
Move data between pandas dataframes and tables with ease!

```python
@aql.dataframe
def my_df_func(input_df: DataFrame):
    input_df['column_name'] = input_df['column_name'] + 1

with dag:
    my_homes_df = aql.load_file(
        file = File('path/to/orders_data_header.csv', S3_CONN_ID),
    )
    my_df_func(my_homes_df)
```
The Dataframe Decorator
Table -> Dataframe

```python
@aql.dataframe
def my_df_func(input_df: DataFrame):
    input_df["column_name"] = input_df["column_name"] + 1

with dag:
    my_homes_table = aql.load_file(file = File('/orders_data_header.csv',S3_CONN_ID),
                                output_table=Table(conn_id="postgres_conn"),
                                )
    my_df_func(my_homes_table)
```
The Dataframe Decorator
Table -> Dataframe -> Table

```python
@aql.dataframe
def my_df_func(input_df: DataFrame):
    input_df["column_name"] = input_df["column_name"] + 1

with dag:
    my_homes_table = aql.load_file(
        file = File('../orders_data_header.csv',S3_CONN_ID),
        output_table=Table(
            conn_id="postgres_conn",
        ),
    )
    my_df_func(my_homes_table, output_table=Table(...))
```
Comparing the Two
Comparing the Two

Traditional: 93 lines of code
Comparing the Two
Astro SDK: 66 lines of code
Coming Soon
@aql.transform
def join_orders_customers(filtered_orders_table: Table, customers_table: Table):
    return """SELECT c.customer_id, customer_name, order_id, purchase_date, amount, type FROM
    {{filtered_orders_table}} fo JOIN {{customers_table}} c
    ON fo.customer_id = c.customer_id"
""

checks = [Check("customer_id_not_null", "customer_id != null"), Check("order_has_cost", "cost >= 0")]

@dag(start_date=datetime(2021, 12, 1), schedule_interval="@daily", catchup=False)
def example_with_validation():
    transformed_data = join_orders_customers(
        df=extracted_data, output_table=Table(name="homes_data_long"),
        input_data_checks = checks)
Data Validation + Lineage
Distributed Dataframes

Move **bigger** data between dataframes and tables with ease!

```python
@snowpark
def my_df_func(input_df: DataFrame):
    input_df["column_name"] = input_df["column_name"] + 1

with dag:
    my_homes_table = aql.load_file(
        path=f"{s3_bucket}/homes.csv",
        output_table=Table(
            conn_id="snowflake_conn",
        ),
    )
    my_df_func(my_homes_table, output_table=Table(...))

@spark
def my_df_func(input_df: DataFrame):
    input_df["column_name"] = input_df["column_name"] + 1

with dag:
    my_homes_table = aql.load_file(
        path=f"{s3_bucket}/homes.csv",
        output_table=Table(
            conn_id="snowflake_conn",
        ),
    )
    my_df_func(my_homes_table, output_table=Table(...))
```
Dynamic Task Templates

```python
@aq1.transform
def filter_snow_data(input_table: Table) -> Table:
    return ""
    SELECT * FROM {{input_table}} WHERE amount > 150
    ""

@task
def print_value(val: str):
    print(f"the value is {val}"

with dag:
    filtered_data = filter_snow_data(input_table=Table(...))
aql.run_per_row(table=filtered_data, task=print_value)
```
And Much More!

Some possibilities:

● Optimized file loading
● Support for asynchronous operators
● Support more databases: RedShift, Azure
● etc.
How to get involved

- Astro Python SDK is available in a **preview** release. Not ready for production use!
- Write some DAGS with it: give us your feedback on the interfaces so we can improve for your needs
- Improvements in progress:
  - Increase speed of file loads
  - Support larger data sets
- Find the code: [https://pypi.org/project/astro-sdk-python/](https://pypi.org/project/astro-sdk-python/)
Thank you

Tw: @danimberman

GH: @dimberman

Source: github.com/astronomer/astro-sdk/tutorial.md

Pypi: https://pypi.org/project/astro-sdk-python/
Appendix
Introducing Astronomer Open Source

What: Add-ons and code overlays complementary to Airflow core. Starting with the Astro Python SDK, Astro CLI, async providers

Why: Shorter development cycles

How: Apache 2.0 license

Who: Open for community contribution, maintained by Astronomers

Where: github.com/astronomer