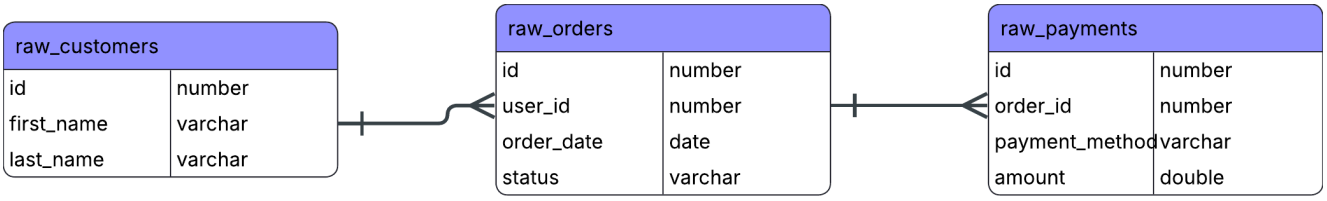
2025-12-03 Create a Customer Insights Pipeline with DBT and Airflow

🎯 Our Goal: Building a Production-Ready Data Pipeline

Welcome! In this notebook, we'll build a complete, production-ready ELT pipeline from scratch. Here's a brief overview of our project:

- **The Dataset:** We'll use the "Jaffle Shop," a fictional e-commerce store. Our raw data is split across three CSV files: `raw_customers`, `raw_orders`, and `raw_payments`. These tables are **logically linked** by shared `id` columns, which we'll use to join them, as shown in the schema diagram below.



- **The Tasks:** We will build an end-to-end pipeline. This includes **Loading** the data (using `dbt seed`), **Transforming** it with a 3-layer dbt model (`staging` → `intermediate` → `marts`), **Testing** our models for data quality (like uniqueness and relationships), and finally, **Orchestrating** the entire process into an automated, scheduled job with Airflow.
- **The Audience:** This pipeline is for any business that wants to answer the critical question, "Who are my most valuable customers?" Our final product will be a clean, reliable, and analytics-ready table (`dim_customers`) that a BI tool (like Tableau or Power BI) can connect to for analysis.

Task 0: Install Prerequisites

For this analysis, we need the **dbt-core**, **dbt-duckdb**, and **apache-airflow** Python packages in order to build our transformation models and orchestrate them.

```
# Install all our packages
# This cell will take about 1-2 minutes to run.
!pip install dbt-core dbt-duckdb apache-airflow

# Fix a known version conflict between dbt and airflow
!pip install --upgrade protobuf
```

Hidden output

```
# Let's peek at our raw data files before we begin

!echo "--- raw_customers.csv ---"
!head raw_customers.csv

!echo "--- raw_orders.csv ---"
!head raw_orders.csv

!echo "--- raw_payments.csv ---"
!head raw_payments.csv
```

```
--- raw_customers.csv ---
id,first_name,last_name
1,Michael,P.
2,Shawn,M.
3,Kathleen,P.
4,Jimmy,C.
5,Katherine,R.
6,Sarah,R.
7,Martin,M.
8,Frank,R.
9,Jennifer,F.
--- raw_orders.csv ---
id,user_id,order_date,status
1,1,2018-01-01,returned
2,3,2018-01-02,completed
3,94,2018-01-04,completed
4,50,2018-01-05,completed
5,64,2018-01-05,completed
6,54,2018-01-07,completed
7,88,2018-01-09,completed
8,2,2018-01-11,returned
9,53,2018-01-12,completed
--- raw_payments.csv ---
id,order_id,payment_method,amount
1,1,credit_card,1000
2,2,credit_card,2000
3,3,coupon,100
4,4,coupon,2500
5,5,bank_transfer,1700
```

Setting Up the ELT Environment (The Foundation)

Before we build our data pipeline, we must set up the necessary tools and connections. This initial setup phase involves three critical steps:

1. **Initialize the dbt Project:** Create the basic project folder structure (`dbt init`).
2. **Configure the Project:** Edit `dbt_project.yml` to define *how* our models should be built (e.g., views vs. tables).
3. **Configure the Connection Profile:** Create `profiles.yml` to define *where* the database (DuckDB) is located, allowing dbt to connect.

Once these steps are complete, we will have a fully configured environment ready for data loading and transformation.

Task 1.1: Initialize the dbt Project

1. Click the **"Open Terminal"** Under Run .
2. Type the following command and press **Enter**:

```
dbt init dbt_project
```

Copy

3. dbt will ask you a few questions:
   - `Which database adapter...` ... Type the number for `duckdb` and press **Enter**.
4. That's it! `dbt init` has created the full project structure. You can close the terminal and come back to the notebook.

Task 1.2: Configure the Project

Great. `dbt init` created a generic `dbt_project.yml` file for us. We just need to make one small edit to tell dbt *how* to build our models.

1. In the file explorer on the left, click on `dbt_project` to expand it.
2. Click on `dbt_project.yml` to open it in the editor.
3. Scroll all the way to the bottom to find the `models:` section. It looks like this:

```
models:
  dbt_project:
    # Config indicated by + and applies to all files under models/example/
    example:
      +materialized: view
```

4. **Replace** that *entire* `models:` block with our new rules for `staging`, `intermediate`, and `marts`:

```
models:
  dbt_project:
    staging:
      materialized: view
    intermediate:
      materialized: table
    marts:
      materialized: view
```

Task 1.3: Configure the Connection Profile

To tell dbt **how** and **where** to connect to our data (the DuckDB file), we need to create the global `profiles.yml` file. This is the **connection string** that dbt will use for every subsequent command.

We define a profile named `jaffle_shop` which points directly to our local DuckDB file path.

```
import os

# Ensure the .dbt directory exists
os.makedirs(os.path.expanduser("~/.dbt"), exist_ok=True)

# Now write the profiles.yml file
with open(os.path.expanduser("~/.dbt/profiles.yml"), "w") as f:
    f.write("""jaffle_shop:
  target: dev
  outputs:
    dev:
      type: duckdb
      path: /work/files/workspace/jaffle_shop.duckdb
      threads: 1
""")
```

Task 2: Load (Seed) Raw Data

Great! Our project is now fully configured.

Now we Copy our *real* data (the Jaffle Shop CSVs) into the `seeds` folder and run `dbt seed`.

Why We Use `dbt seed` Here:

While seeds are typically used in production for **small, static data** that rarely changes, we use them here for **convenience** and **replicability**. This step:

- **1. Creates Base Tables:** It quickly loads the raw CSV data into queryable tables within our local DuckDB database.
- **2. Guarantees Consistency:** It ensures that every participant starts with the **exact same raw data**, making our downstream tests and transformations reliable.

```
# 1. Copy our real data into the seeds folder
!cp raw_customers.csv dbt_project/seeds/
!cp raw_orders.csv dbt_project/seeds/
!cp raw_payments.csv dbt_project/seeds/

# 2. Run dbt seed!
# This loads the CSVs into our 'jaffle_shop.duckdb' file
!dbt seed --project-dir dbt_project --profile jaffle_shop
```

```
19:51:27  Running with dbt=1.9.2
19:51:27  Registered adapter: duckdb=1.9.2
19:51:28  Found 8 models, 3 seeds, 8 data tests, 3 sources, 424 macros
19:51:28
19:51:28  Concurrency: 1 threads (target='dev')
19:51:28
19:51:29
19:51:29  Finished running  in 0 hours 0 minutes and 0.23 seconds (0.23s).
19:51:29  Encountered an error:
Runtime Error
  IO Error: Could not set lock on file "/work/files/workspace/jaffle_shop.duckdb":
Conflicting lock is held in /usr/bin/python3.10 (PID 375). However, you would be
able to open this database in read-only mode, e.g. by using the -readonly parameter
in the CLI. See also https://duckdb.org/docs/connect/concurrency
```

Introducing the Transformation Layer (The "T" in ELT)

The next three sections—which contain our staging, intermediate, and marts models—represent the **Transformation Layer** of our pipeline. This is where we apply business logic to clean, combine, and reshape the raw data into valuable, analytics-ready tables.

This process follows a standard data modeling approach:

- **Sources:** Defines and documents the raw, external data tables (the entry point to the transformation pipeline).
- **Staging:** Cleans and renames raw source data (simple 1:1 views).
- **Intermediate:** Joins and aggregates data (performs complex logic).
- **Marts:** Creates final, clean tables ready for BI tools.

Task 3.1: Define Sources

This step creates the `source.yml` file, which is the necessary first step before writing any model. It officially tells dbt where to find the raw data tables (the seeds we loaded in Task 2) in the database.

By doing this, we create the **Sources** layer of our data pipeline.

```
# Create the staging folder
!mkdir -p dbt_project/models/staging
```

```
%%writefile dbt_project/models/staging/source.yml
version: 2

sources:
  - name: jaffle_shop
    schema: main # This is the schema DuckDB created for our seeds
    tables:
      - name: raw_customers
      - name: raw_orders
      - name: raw_payments
```

```
Overwriting dbt_project/models/staging/source.yml
```

Task 3.2: Build Staging Models

Now for the "T" (Transform)! We'll build our `staging` models. These are simple 1-to-1 **views** of our raw data. Their only job is to do light cleaning, like renaming columns.

We'll use the `{{ source(...) }}` macro to reference the raw tables we just loaded.

```
%%writefile dbt_project/models/staging/stg_customers.sql
select
    id as customer_id,
    first_name,
    last_name
from {{ source('jaffle_shop', 'raw_customers') }}
```

```
Overwriting dbt_project/models/staging/stg_customers.sql
```

```
%%writefile dbt_project/models/staging/stg_orders.sql
select
    id as order_id,
    user_id as customer_id,
    order_date,
    status
from {{ source('jaffle_shop', 'raw_orders') }}
```

```
Overwriting dbt_project/models/staging/stg_orders.sql
```

```
%%writefile dbt_project/models/staging/stg_payments.sql
select
    id as payment_id,
    order_id,
    amount,
    payment_method
from {{ source('jaffle_shop', 'raw_payments') }}
```

```
Overwriting dbt_project/models/staging/stg_payments.sql
```

```
# Run dbt run to execute all the models in our staging folder (stg_customers,
stg_orders, stg_payments)
!dbt run --select staging --project-dir dbt_project --profile jaffle_shop
```

```
20:14:14  Running with dbt=1.9.2
20:14:14  Registered adapter: duckdb=1.9.2
20:14:16  Found 8 models, 3 seeds, 8 data tests, 3 sources, 424 macros
20:14:16
20:14:16  Concurrency: 1 threads (target='dev')
20:14:16
20:14:17  1 of 3 START sql view model main.stg_customers
.............................. [RUN]
20:14:18  1 of 3 OK created sql view model main.stg_customers
.......................... [OK in 0.68s]
20:14:18  2 of 3 START sql view model main.stg_orders
................................... [RUN]
20:14:19  2 of 3 OK created sql view model main.stg_orders
............................... [OK in 0.59s]
20:14:19  3 of 3 START sql view model main.stg_payments
................................... [RUN]
20:14:19  3 of 3 OK created sql view model main.stg_payments
............................... [OK in 0.59s]
20:14:20
20:14:20  Finished running 3 view models in 0 hours 0 minutes and 3.78 seconds
(3.78s).
20:14:20
20:14:20  Completed successfully
20:14:20
20:14:20  Done. PASS=3 WARN=0 ERROR=0 SKIP=0 TOTAL=3
```

Task 3.3: Build Intermediate Models

Success! Our staging models are running.

Now we'll build our `intermediate` models. These models often join staging models together or perform heavier calculations. We'll create one that joins `stg_orders` and `stg_payments`.

```
# Create the intermediate folder
!mkdir -p dbt_project/models/intermediate
```

```
%%writefile dbt_project/models/intermediate/int_orders.sql
with orders as (
 select * from {{ ref('stg_orders') }}
),
payments as (
 select * from {{ ref('stg_payments') }}
),
order_payments as (
 select
 orders.order_id,
 orders.customer_id,
 orders.order_date,
 orders.status,
 payments.amount
 from orders
 left join payments
    on orders.order_id = payments.order_id
),
final_payments as (
 select
 order_id,
 customer_id,
 order_date,
 sum(case when trim(status) = 'completed' then amount else 0 end) as amount
 from order_payments
 group by 1, 2, 3
)
select * from final_payments
```

```
Overwriting dbt_project/models/intermediate/int_orders.sql
```

```
# Run the new intermediate model
!dbt run --select int_orders --project-dir dbt_project --profile jaffle_shop
```

```
20:18:52  Running with dbt=1.9.2
20:18:53  Registered adapter: duckdb=1.9.2
20:18:54  Found 8 models, 3 seeds, 8 data tests, 3 sources, 424 macros
20:18:54
20:18:54  Concurrency: 1 threads (target='dev')
20:18:54
20:18:55  1 of 1 START sql table model main.int_orders
.................................. [RUN]
20:18:56  1 of 1 OK created sql table model main.int_orders
.............................. [OK in 0.71s]
20:18:56
20:18:56  Finished running 1 table model in 0 hours 0 minutes and 2.01 seconds
(2.01s).
20:18:56
20:18:56  Completed successfully
20:18:56
20:18:56  Done. PASS=1 WARN=0 ERROR=0 SKIP=0 TOTAL=1
```

Task 3.4: Build Marts Models

We're at the final layer! **Marts** models are the clean, end-user-facing models that power dashboards and analysis. They are built from our `staging` and `intermediate` models.

We'll build two:

1. `fct_orders.sql`: A "fact" table that joins our `int_orders` model with `stg_customers`.
2. `dim_customers.sql`: A "dimension" table that is just a clean copy of `stg_customers`.

```
# Create the marts folder
!mkdir -p dbt_project/models/marts
```

First, the `fct_orders.sql` model. This joins our `int_orders` model with `stg_customers` to create a final "fact" table.

```
%%writefile dbt_project/models/marts/fct_orders.sql
with orders as (
    select * from {{ ref('int_orders') }}
),

customers as (
    select * from {{ ref('stg_customers') }}
),

final as (
    select
        orders.order_id,
        orders.customer_id,
        customers.first_name,
        customers.last_name,
        orders.order_date,
        orders.amount
    from orders
    left join customers using (customer_id)
)

select * from final
```

```
Overwriting dbt_project/models/marts/fct_orders.sql
```

Next, the `dim_customers.sql` model. This "dimension" table is just a clean version of our customers, ready for analysis.

```
%%writefile dbt_project/models/marts/dim_customers.sql
select
    customer_id,
    first_name,
    last_name
from {{ ref('stg_customers') }}
```

```
Overwriting dbt_project/models/marts/dim_customers.sql
```

Now, let's run *only* our new `marts` models.

```
!dbt run --select marts --project-dir dbt_project --profile jaffle_shop
```

```
20:23:16  Running with dbt=1.9.2
20:23:17  Registered adapter: duckdb=1.9.2
20:23:18  Found 8 models, 3 seeds, 8 data tests, 3 sources, 424 macros
20:23:18
20:23:18  Concurrency: 1 threads (target='dev')
20:23:18
20:23:19  1 of 2 START sql view model main.dim_customers
.............................. [RUN]
20:23:20  1 of 2 OK created sql view model main.dim_customers
.......................... [OK in 0.67s]
20:23:20  2 of 2 START sql view model main.fct_orders
................................. [RUN]
20:23:20  2 of 2 OK created sql view model main.fct_orders
.............................. [OK in 0.60s]
20:23:21
20:23:21  Finished running 2 view models in 0 hours 0 minutes and 2.69 seconds
(2.69s).
20:23:21
20:23:21  Completed successfully
20:23:21
20:23:21  Done. PASS=2 WARN=0 ERROR=0 SKIP=0 TOTAL=2
```

Task 4: Test Your Models

This is a critical step in any dbt project. We need to add **data tests** to ensure our logic is correct and the data is clean.

We'll add a new `schema.yml` file in our `marts` folder to define some tests:

- `dim_customers` should have a `customer_id` that is both **unique** and **never null**.
- `fct_orders` should have an `order_id` that is **unique** and **never null**.

Then, we'll run `dbt test`.

```
%%writefile dbt_project/models/marts/schema.yml
version: 2

models:
  - name: dim_customers
    columns:
      - name: customer_id
        tests:
          - unique
          - not_null

  - name: fct_orders
    columns:
      - name: order_id
        tests:
          - unique
          - not_null
```

```
Overwriting dbt_project/models/marts/schema.yml
```

```
# Run the tests!
!dbt test --project-dir dbt_project --profile jaffle_shop
```

```
20:27:31  Running with dbt=1.9.2
20:27:31  Registered adapter: duckdb=1.9.2
20:27:33  Found 8 models, 3 seeds, 8 data tests, 3 sources, 424 macros
20:27:33
20:27:33  Concurrency: 1 threads (target='dev')
20:27:33
20:27:33  1 of 8 START test not_null_dim_customers_customer_id
.......................... [RUN]
20:27:34  1 of 8 PASS not_null_dim_customers_customer_id
.............................. [PASS in 0.39s]
20:27:34  2 of 8 START test not_null_fct_orders_order_id
.............................. [RUN]
20:27:34  2 of 8 PASS not_null_fct_orders_order_id
.................................... [PASS in 0.36s]
20:27:34  3 of 8 START test not_null_my_first_dbt_model_id
............................... [RUN]
20:27:34  3 of 8 FAIL 1 not_null_my_first_dbt_model_id
.............................. [FAIL 1 in 0.40s]
20:27:35  4 of 8 START test not_null_my_second_dbt_model_id
............................... [RUN]
20:27:35  4 of 8 ERROR not_null_my_second_dbt_model_id
.............................. [ERROR in 0.40s]
20:27:35  5 of 8 START test unique_dim_customers_customer_id
............................ [RUN]
20:27:35  5 of 8 PASS unique_dim_customers_customer_id
.............................. [PASS in 0.37s]
20:27:36  6 of 8 START test unique_fct_orders_order_id
.............................. [RUN]
20:27:36  6 of 8 PASS unique_fct_orders_order_id
```

Task 5: Orchestrate the Pipeline with Airflow

We have successfully built and tested our dbt models. But so far, we've been running them manually.

The "Airflow" part of our pipeline is the **orchestrator**. Its job is to run these commands on a schedule.

Since Airflow is already installed, our first step is to initialize the Airflow database. This is a one-time command that creates the tables Airflow needs to track its own state.

```
# This command initializes the Airflow metadata database
!airflow db migrate
```

```
DB: sqlite:////home/repl/airflow/airflow.db
Performing upgrade to the metadata database sqlite:////home/repl/airflow/airflow.db
2025-12-04T20:30:45.744664Z [info     ] Context impl SQLiteImpl.
[alembic.runtime.migration] loc=migration.py:207
2025-12-04T20:30:45.744854Z [info     ] Will assume non-transactional DDL.
[alembic.runtime.migration] loc=migration.py:210
2025-12-04T20:30:45.747073Z [info     ] Migrating the Airflow database
[airflow.utils.db] loc=db.py:1129
2025-12-04T20:30:45.752290Z [info     ] Context impl SQLiteImpl.
[alembic.runtime.migration] loc=migration.py:207
2025-12-04T20:30:45.752428Z [info     ] Will assume non-transactional DDL.
[alembic.runtime.migration] loc=migration.py:210
2025-12-04T20:30:45.777368Z [info     ] Context impl SQLiteImpl.
[alembic.runtime.migration] loc=migration.py:207
2025-12-04T20:30:45.777534Z [info     ] Will assume non-transactional DDL.
[alembic.runtime.migration] loc=migration.py:210
Database migrating done!
```

Now that the database is initialized, let's create our Airflow **DAG file** (`dbt_pipeline_dag.py`). This is the Python script that defines our pipeline.

```
# Create a new, clean folder for our DAGs
!mkdir dags
```

```
mkdir: cannot create directory ‘dags’: File exists
```

```
%%writefile dags/dbt_pipeline_dag.py
from airflow.decorators import dag
from airflow.operators.bash import BashOperator
from pendulum import datetime
import os

WORKSPACE_ROOT = "/work/files/workspace"
DBT_PROJECT_DIR = "/work/files/workspace/dbt_project"
DBT_PROFILES_DIR = os.path.expanduser("~/.dbt")

@dag(
    dag_id="dbt_jaffle_shop_pipeline",
    start_date=datetime(2024, 1, 1),
    schedule="@daily",
    catchup=False,
    tags=["dbt", "jaffle_shop"],
)
def dbt_jaffle_shop_dag():
    """
    An Airflow DAG that runs the entire dbt project (seed, run, test) in one go.
    """

    dbt_build_task = BashOperator(
        task_id="dbt_build_pipeline",
        bash_command=f"""
        set -e
        cd {WORKSPACE_ROOT}

        cp ./raw_customers.csv ./dbt_project/seeds/
        cp ./raw_orders.csv ./dbt_project/seeds/
        cp ./raw_payments.csv ./dbt_project/seeds/

        dbt build --project-dir {DBT_PROJECT_DIR} --profile jaffle_shop
        """,
        env={"DBT_PROFILES_DIR": DBT_PROFILES_DIR}
    )

    dbt_build_task

# Instantiate the DAG
dbt_jaffle_shop_dag()
```

```
Overwriting dags/dbt_pipeline_dag.py
```

Run the Pipeline!

Now that the database is initialized and our DAG file exists, let's run the tasks.

```
# Close the previous Python connection to free the database lock ---
try:
    if 'conn' in locals() and conn:
        conn.close()
    print("Database connection closed successfully.")
except NameError:
    pass

# Tell Airflow to use *only* our new 'dags' folder
%env AIRFLOW__CORE__DAGS_FOLDER=dags
# Run the single dbt_build_pipeline task
print("--- RUNNING FULL PIPELINE: dbt_build_pipeline ---")
!airflow tasks test dbt_jaffle_shop_pipeline dbt_build_pipeline 2024-01-01
```

```
Database connection closed successfully.
env: AIRFLOW__CORE__DAGS_FOLDER=dags
--- RUNNING FULL PIPELINE: dbt_build_pipeline ---
2025-12-04T20:31:35.463146Z [info     ] DAG bundles loaded: dags-folder,
example_dags [airflow.dag_processing.bundles.manager.DagBundlesManager]
loc=manager.py:179
2025-12-04T20:31:35.469211Z [info     ] Filling up the DagBag from dags
[airflow.models.dagbag.DagBag] loc=dagbag.py:593
2025-12-04T20:31:35.574800Z [info     ] Sync 1 DAGs
[airflow.serialization.serialized_objects] loc=serialized_objects.py:2893
2025-12-04T20:31:36.025636Z [info     ] NumExpr defaulting to 16 threads.
[numexpr.utils] loc=utils.py:162
2025-12-04T20:31:36.194186Z [info     ] Setting next_dagrun for
dbt_jaffle_shop_pipeline to 2025-12-04 00:00:00+00:00, run_after=2025-12-04
00:00:00+00:00 [airflow.models.dag] loc=dag.py:688
2025-12-04T20:31:36.246844Z [info     ] Created dag run.
[airflow.models.dagrun] dagrun=<DagRun dbt_jaffle_shop_pipeline @ 2024-01-01
00:00:00+00:00: __airflow_temporary_run_2025-12-04T20:31:36.229558+00:00__,
state:running, queued_at: None. run_type: manual> loc=dagrun.py:2188
2025-12-04T20:31:36.258065Z [info     ] [DAG TEST] starting
task_id=dbt_build_pipeline map_index=-1 [airflow.sdk.definitions.dag]
loc=dag.py:1369
2025-12-04T20:31:36.258624Z [info     ] [DAG TEST] running task <TaskInstance:
dbt_jaffle_shop_pipeline.dbt_build_pipeline __airflow_temporary_run_2025-12-
04T20:31:36.229558+00:00__ [None]> [airflow.sdk.definitions.dag] loc=dag.py:1372
2025-12-04T20:31:39.146565Z [info     ] Task started
[airflow.api_fastapi.execution_api.routes.task_instances] hostname=2652358a-2371-
4178-b257-68ccce8b7751.sessions.sessions.svc.cluster.local loc=task_instances.py:199
previous_state=queued ti_id=019aeb10-2936-7811-996f-d802bb78a6ee
```

Task 6: View the Final Analysis

The entire pipeline—from raw data to tested transformations—was created to answer the question, "Who are my most valuable customers?" The answer is found in the final mart table, `dim_customers`.

We'll use a Python query to select and display the top 10 most valuable customers (based on order amount).

```
import duckdb

# Connect to the database file created by dbt
conn = duckdb.connect(database='/work/files/workspace/jaffle_shop.duckdb',
read_only=True)

# Query the final table, calculate customer value, and display the top 10
# Note: Since we don't know the exact columns in your dim_customers model,
# we'll perform a simple join with fct_orders to calculate total revenue per
customer.
top_customers_query = """
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    SUM(f.amount) AS total_lifetime_value
FROM dim_customers AS c
JOIN fct_orders AS f ON c.customer_id = f.customer_id
GROUP BY 1, 2, 3
ORDER BY total_lifetime_value DESC
LIMIT 10
"""

# Execute the query and display the results
df_final = conn.execute(top_customers_query).fetchdf()
print("Top 10 Most Valuable Customers (Final Output):")
display(df_final)
```

```
Top 10 Most Valuable Customers (Final Output):
```

	cus...	fi...	l..	total_lifetime_value
0	51	Howard	R.	9900
1	3	Kathleen	P.	6500
2	50	Billy	L.	4700
3	8	Frank	R.	4500
4	99	Mary	G.	4400
5	71	Gerald	C.	4400
6	54	Rose	M.	4100
7	53	Anne	B.	3900
8	69	Janet	P.	3200
9	32	Thomas	O.	3000

Rows: 10 Expand

🏁 Project Complete!

Congratulations! You have successfully built and orchestrated a full data pipeline. The pipeline successfully created the final analytical table, and the output directly answers the core business question: **"Who are our most valuable customers?"**.

What We Did:

- **Built Models (dbt):** We used **dbt** to load seed data, run transformations (staging → intermediate → marts), and test our data quality.
- **Orchestrated Pipeline (Airflow):** We wrote an **Airflow DAG** and used the **airflow** command to automatically run our entire dbt pipeline (`seed` , `run` , and `test`) in the correct, automated sequence.
- **The Answer:** The final output is the single, reliable `dim_customers` table, which a BI tool (like Tableau or Power BI) could connect to for analysis.

This is the core workflow of a modern data pipeline!