End-to-End Data Engineering Project: Building a Complete Analytics Pipeline with PySpark and DBT

Alissa Khan | Alissa@AIDataPro.onmicrosoft.com | www.linkedin.com/in/alissakhan-data

Project Overview

This project demonstrates the construction of a complete data engineering pipeline using a pseudo-Uber dataset. The pipeline follows the Medallion Architecture (Bronze for raw data, Silver for cleaned and transformed data, and Gold for modeled analytics-ready data). It handles incremental data ingestion, data cleaning, deduplication, upserts, and dimensional modeling with slowly changing dimensions (SCD Type 2). The dataset includes CSV files for entities like trips, customers, drivers, vehicles, payments, and locations with completed Views for BI Analytics use at the end.

The project is built on Databricks (free tier) and emphasizes production-grade practices such as idempotency, cost optimization, and modular code. It showcases real-world data engineering skills applicable to cloud platforms like Azure, GCP, and AWS.

Tools and Technologies Used

- **PySpark**: For structured streaming, data ingestion, transformations, deduplication, and upserts.
- **Databricks**: Platform for compute, storage (Delta Lake), catalogs, schemas, volumes, and notebooks.
- **DBT (Data Build Tool)**: For data modeling, incremental builds, snapshots, sources, and orchestration (using DBT Cloud free tier).
- **Jinja**: Templating engine within DBT for dynamic SQL generation (e.g., loops and conditionals).
- **Python**: For modular utilities, classes, and reusable transformation logic.
- **Delta Lake**: Storage format for reliable, ACID-compliant tables with merge capabilities.
- **YAML**: For DBT configurations, sources, and snapshots.
- **SQL**: Core language for DBT models and PySpark transformations.
- **Git**: Version control for DBT projects (integrated in DBT Cloud).
- **CSV Files**: Source data format for the pseudo-Uber dataset.

Relevant Skills Demonstrated

- PySpark Structured Streaming for incremental ingestion.

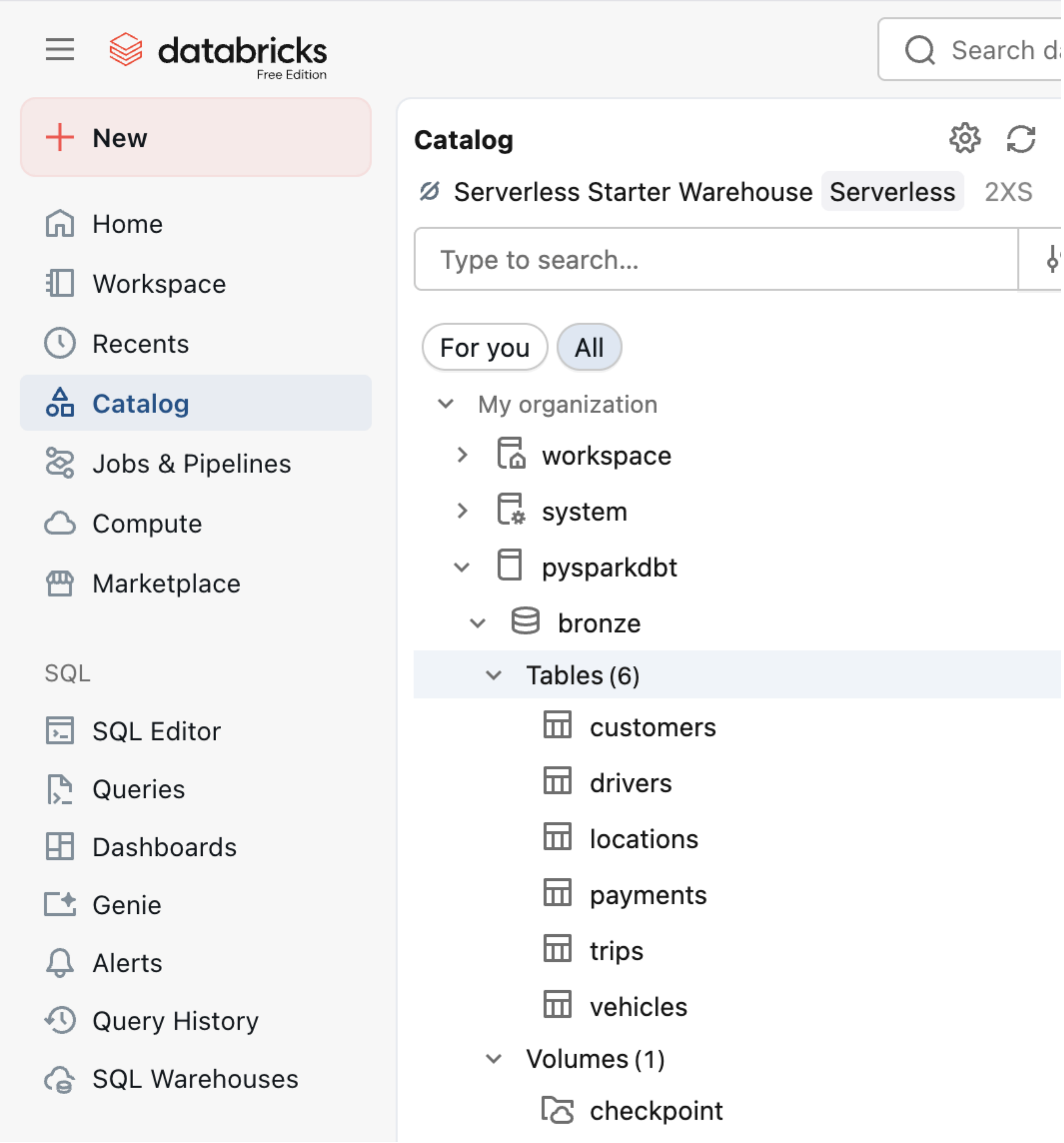
- Dynamic schema handling and notebook automation.
- Data cleaning techniques (e.g., regex, concatenation, conditional columns).
- Object-oriented programming in Python for reusable transformations.
- Deduplication using window functions and hashing.
- Upsert operations with Delta Tables and conditional merges.
- DBT modeling including incremental materializations, SCD Type 2 via snapshots, and lineage tracking.
- Jinja templating for dynamic SQL.
- Databricks environment management (catalogs, schemas, volumes, checkpoints).
- Data quality and governance (audit timestamps, idempotent processing).
- Modular and scalable pipeline design for production readiness.

Step-by-Step Project Build Process

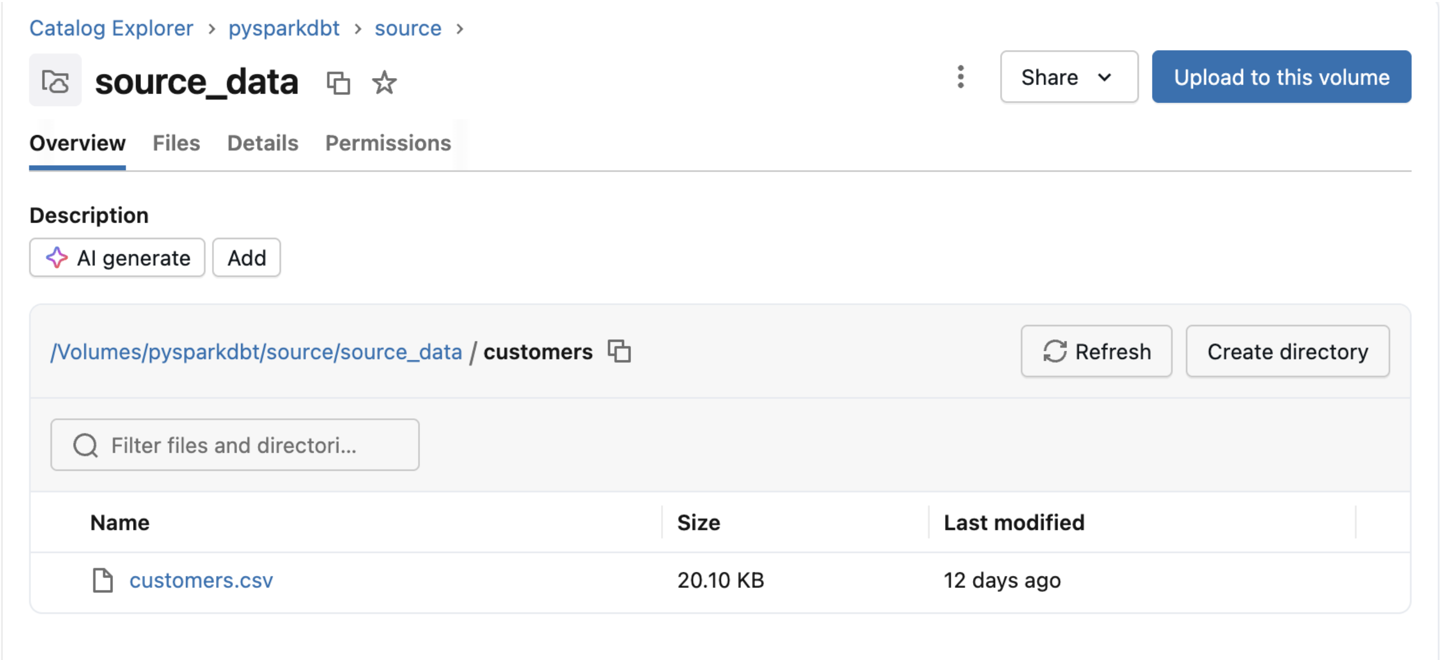
Step 1: Environment Setup on Databricks

I started by setting up the environment in Databricks using the free community edition. This involved creating a serverless compute cluster with autoscaling to handle processing efficiently without manual VM management.

- Created a catalog named pyspark_dbt.
- Defined schemas: source for raw data, bronze for ingested data, silver for transformed data, and gold for modeled data.
- Set up a volume source_data under the source schema, with subdirectories for each entity (e.g., trips, customers, drivers, vehicles, payments, locations).
- Uploaded CSV files into the respective subdirectories.
- Created a checkpoint volume under the bronze schema to store metadata for incremental processing.



Databricks catalog and schemas setup



Catalog Explorer › pysparkdbt › source ›

source_data

Description

/Volumes/pysparkdbt/source/source_data / customers

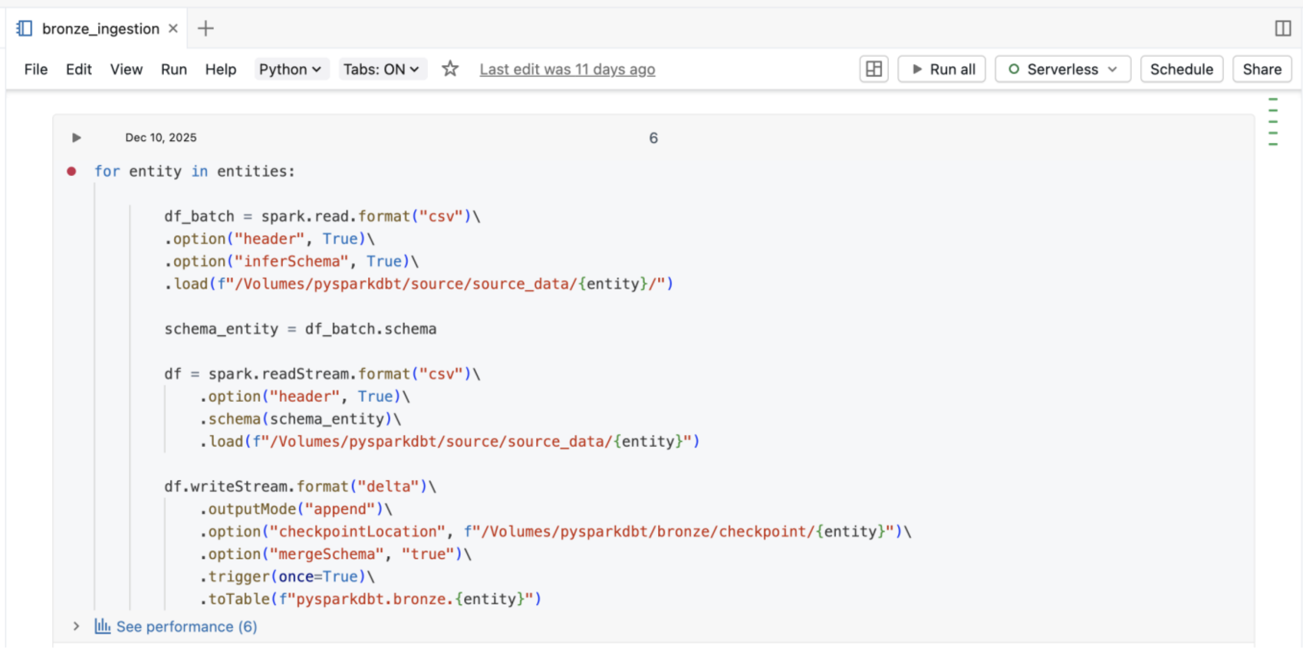
Name	Size	Last modified
customers.csv	20.10 KB	12 days ago

Source data volume with uploaded CSV files

Step 2: Bronze Layer - Incremental Data Ingestion with PySpark

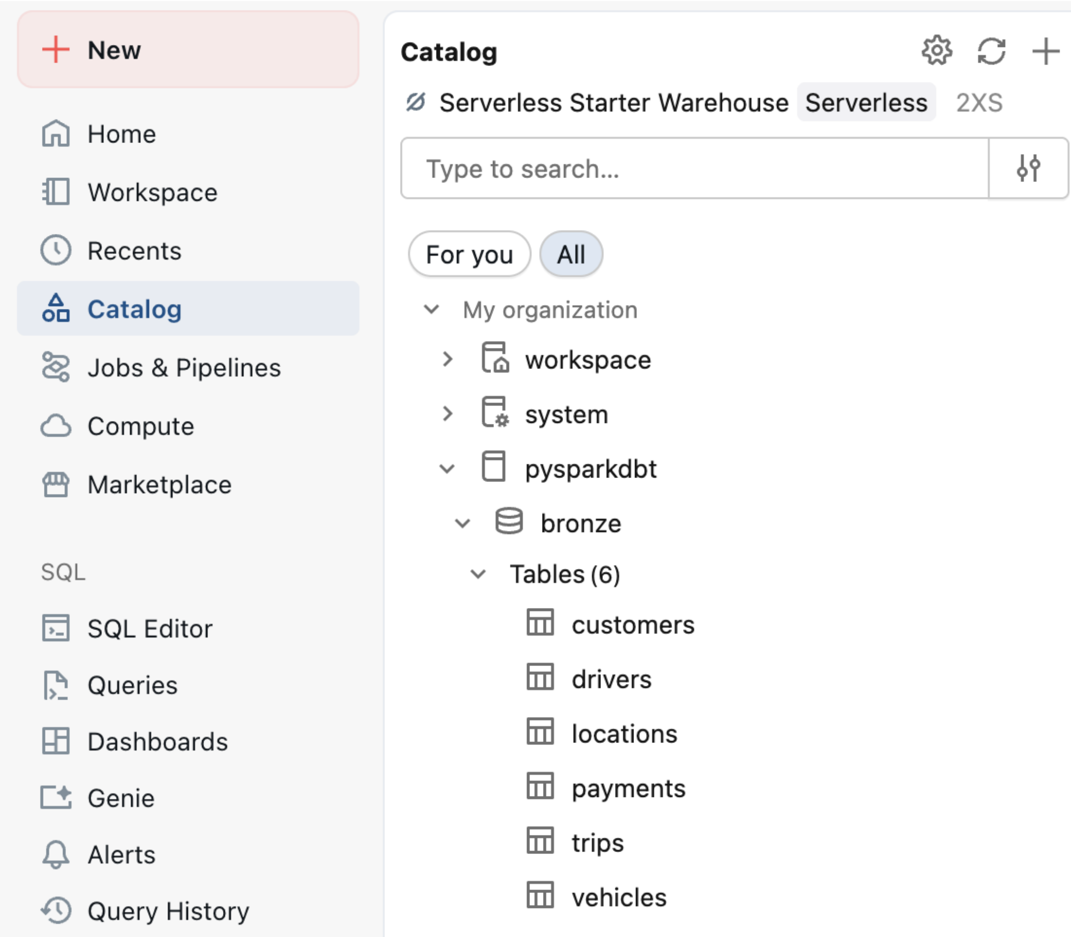
In this step, I used PySpark Structured Streaming to ingest raw CSV data incrementally into Delta tables in the bronze layer. This ensures only new files are processed, making the pipeline efficient.

- Defined a list of entities: ["customers", "trips", "vehicles", "drivers", "payments", "locations"].
- In a Databricks notebook, used a loop to dynamically process each entity:
  - Read streaming data with spark.readStream.format("csv").option("header", "true").option("inferSchema", "true").
  - Extracted schema dynamically from a batch DataFrame to avoid hardcoding.
  - Wrote to bronze tables using format("delta").outputMode("append").option("checkpointLocation", "<path>").trigger(once=True).start().
- This created 6 Delta tables in the bronze schema, preserving raw data with checkpoints for fault tolerance.



```
for entity in entities:

        df_batch = spark.read.format("csv")\
        .option("header", True)\
        .option("inferSchema", True)\
        .load(f"/Volumes/pysparkdbt/source/source_data/{entity}/")

        schema_entity = df_batch.schema

        df = spark.readStream.format("csv")\
            .option("header", True)\
            .schema(schema_entity)\
            .load(f"/Volumes/pysparkdbt/source/source_data/{entity}")

        df.writeStream.format("delta")\
            .outputMode("append")\
            .option("checkpointLocation", f"/Volumes/pysparkdbt/bronze/checkpoint/{entity}")\
            .option("mergeSchema", "true")\
            .trigger(once=True)\
            .toTable(f"pysparkdbt.bronze.{entity}")
```

PySpark code for dynamic ingestion loop

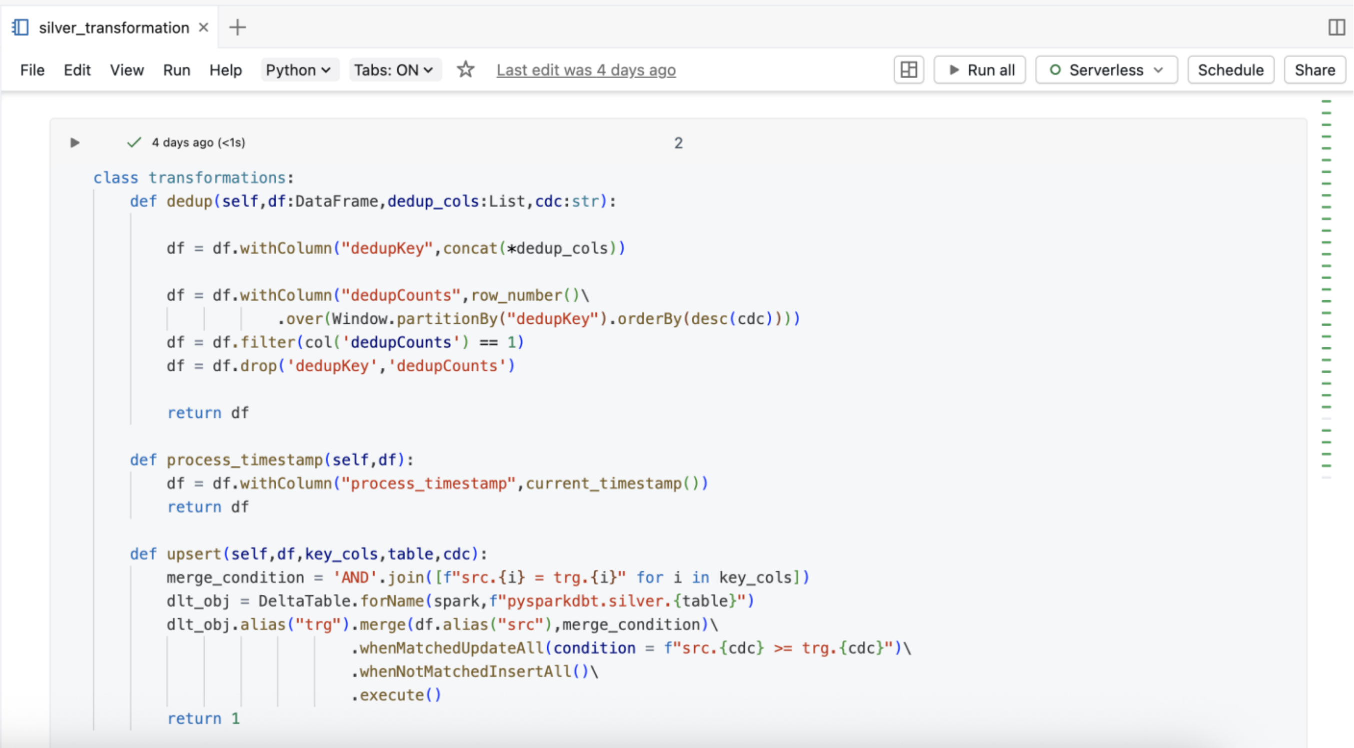


Bronze layer Delta tables in Databricks catalog

Step 3: Silver Layer - Data Transformation with PySpark

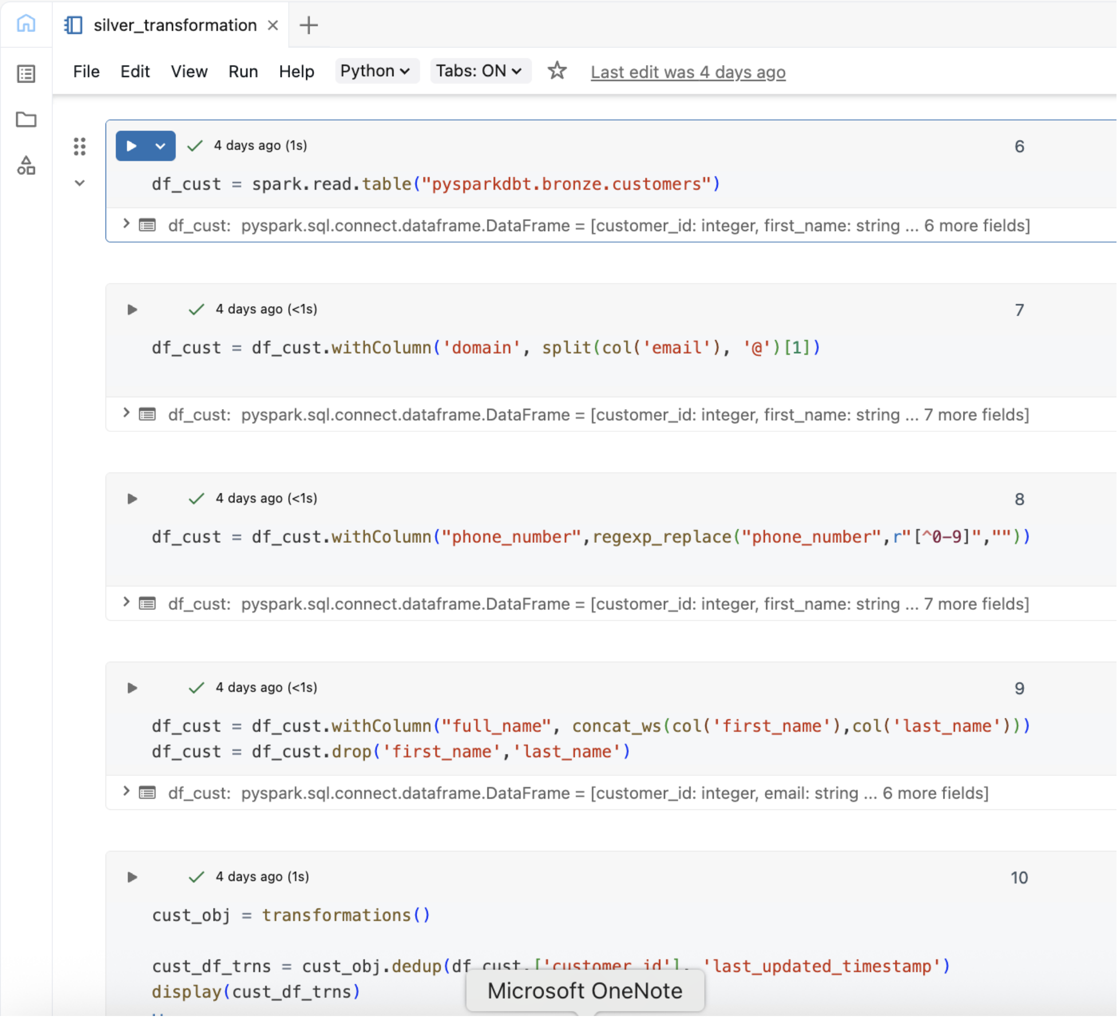
Here, I focused on cleaning and enriching the data using modular Python code. I created a utility file custom_utils.py with a Transformations class to encapsulate reusable logic.

- Read data from bronze tables: spark.read.table("bronze.<entity>").
- Applied generic transformations:
  - Deduplication: Used window functions (row_number over partitioned hash keys) to keep the latest records based on a change data capture (CDC) timestamp.
  - Added process_timestamp using current_timestamp() for auditing.
  - Upserts: Checked if silver table exists; if yes, performed Delta merge with dynamic conditions (e.g., update if source timestamp is newer).
- Entity-specific cleaning (examples):
  - Customers: Extracted email domains with split, cleaned phone numbers with regexp_replace, concatenated full names with concat_ws.
  - Payments: Added conditional online_payment_status using when().otherwise().
  - Vehicles: Converted make to uppercase.
- Looped over entities to apply transformations and upsert into silver.<entity> tables.



```
class transformations:
    def dedup(self,df:DataFrame,dedup_cols:List,cdc:str):

        df = df.withColumn("dedupKey",concat(*dedup_cols))

        df = df.withColumn("dedupCounts",row_number()\
                  .over(Window.partitionBy("dedupKey").orderBy(desc(cdc))))
        df = df.filter(col('dedupCounts') == 1)
        df = df.drop('dedupKey','dedupCounts')

        return df

    def process_timestamp(self,df):
        df = df.withColumn("process_timestamp",current_timestamp())
        return df

    def upsert(self,df,key_cols,table,cdc):
        merge_condition = 'AND'.join([f"src.{i} = trg.{i}" for i in key_cols])
        dlt_obj = DeltaTable.forName(spark,f"pysparkdbt.silver.{table}")
        dlt_obj.alias("trg").merge(df.alias("src"),merge_condition)\
                        .whenMatchedUpdateAll(condition = f"src.{cdc} >= trg.{cdc}")\
                        .whenNotMatchedInsertAll()\
                        .execute()
        return 1
```

Transformations class in custom_utils.py



4 days ago (1s) 6

```
df_cust = spark.read.table("pysparkdbt.bronze.customers")
```

```
df_cust:  pyspark.sql.connect.dataframe.DataFrame = [customer_id: integer, first_name: string ... 6 more fields]
```

4 days ago (<1s) 7

```
df_cust = df_cust.withColumn('domain', split(col('email'), '@')[1])
```

```
df_cust:  pyspark.sql.connect.dataframe.DataFrame = [customer_id: integer, first_name: string ... 7 more fields]
```

4 days ago (<1s) 8

```
df_cust = df_cust.withColumn("phone_number",regexp_replace("phone_number",r"[^0-9]",""))
```

```
df_cust:  pyspark.sql.connect.dataframe.DataFrame = [customer_id: integer, first_name: string ... 7 more fields]
```

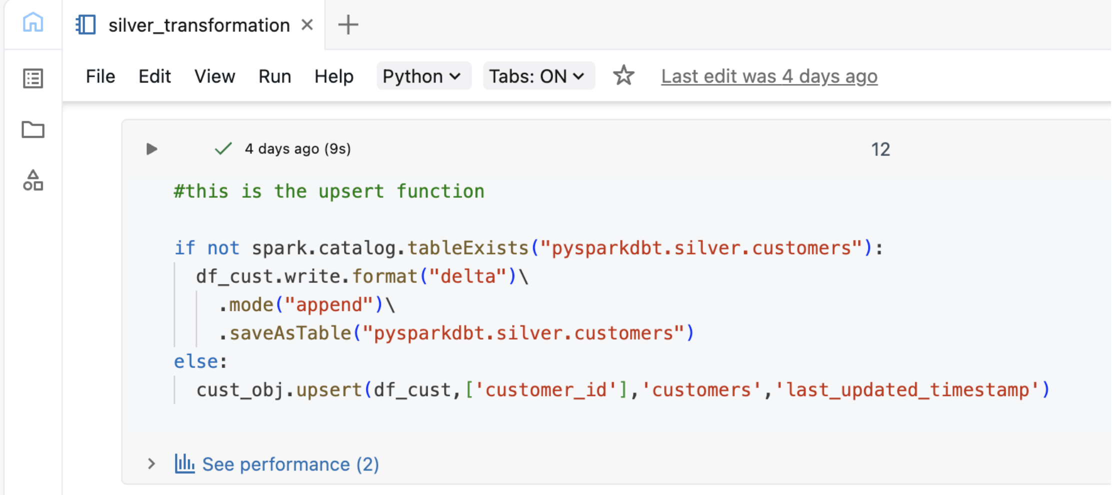
4 days ago (<1s) 9

```
df_cust = df_cust.withColumn("full_name", concat_ws(col('first_name'),col('last_name')))
df_cust = df_cust.drop('first_name','last_name')
```

```
df_cust:  pyspark.sql.connect.dataframe.DataFrame = [customer_id: integer, email: string ... 6 more fields]
```

4 days ago (1s) 10

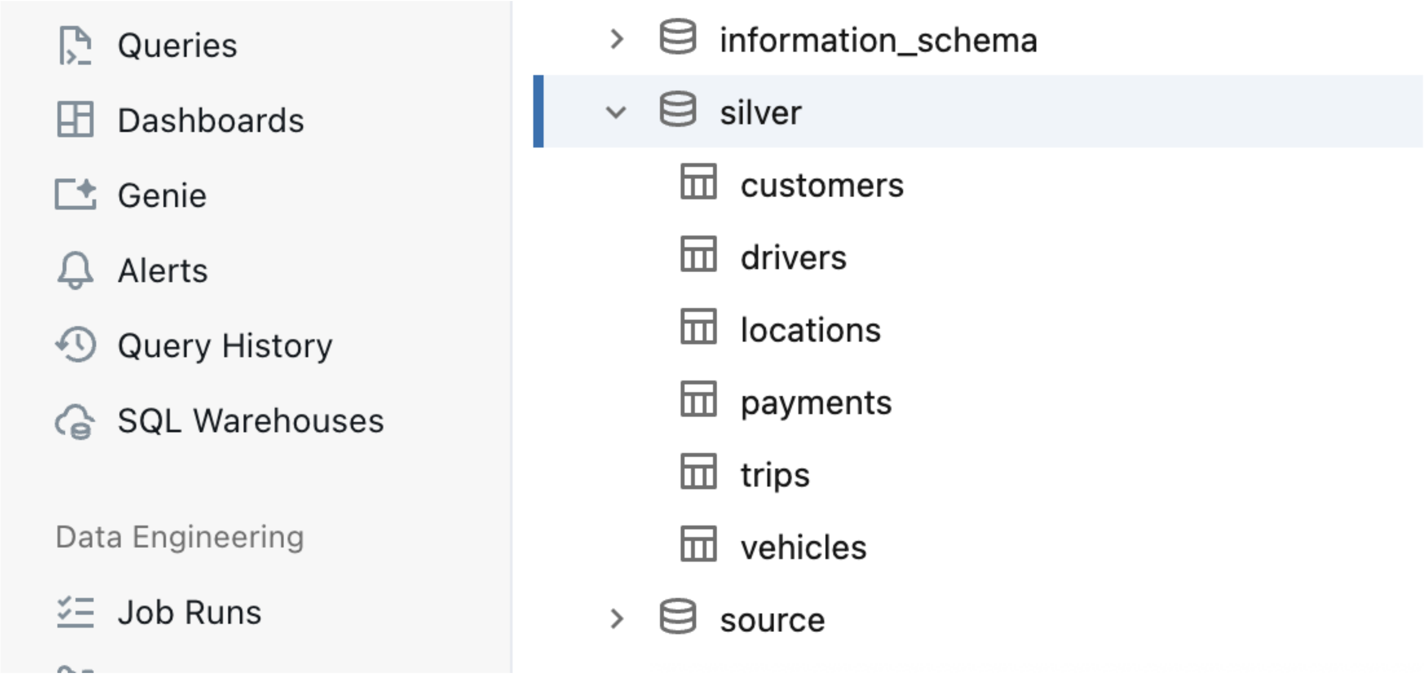
```
cust_obj = transformations()

cust_df_trns = cust_obj.dedup(df_cust,['customer_id'], 'last_updated_timestamp')
display(cust_df_trns)
```



4 days ago (9s) 12

```
#this is the upsert function

if not spark.catalog.tableExists("pysparkdbt.silver.customers"):
    df_cust.write.format("delta")\
      .mode("append")\
      .saveAsTable("pysparkdbt.silver.customers")
else:
    cust_obj.upsert(df_cust,['customer_id'],'customers','last_updated_timestamp')
```

Example PySpark code for customer transformations and upsert



information_schema
silver
customers
drivers
locations
payments
trips
vehicles
source

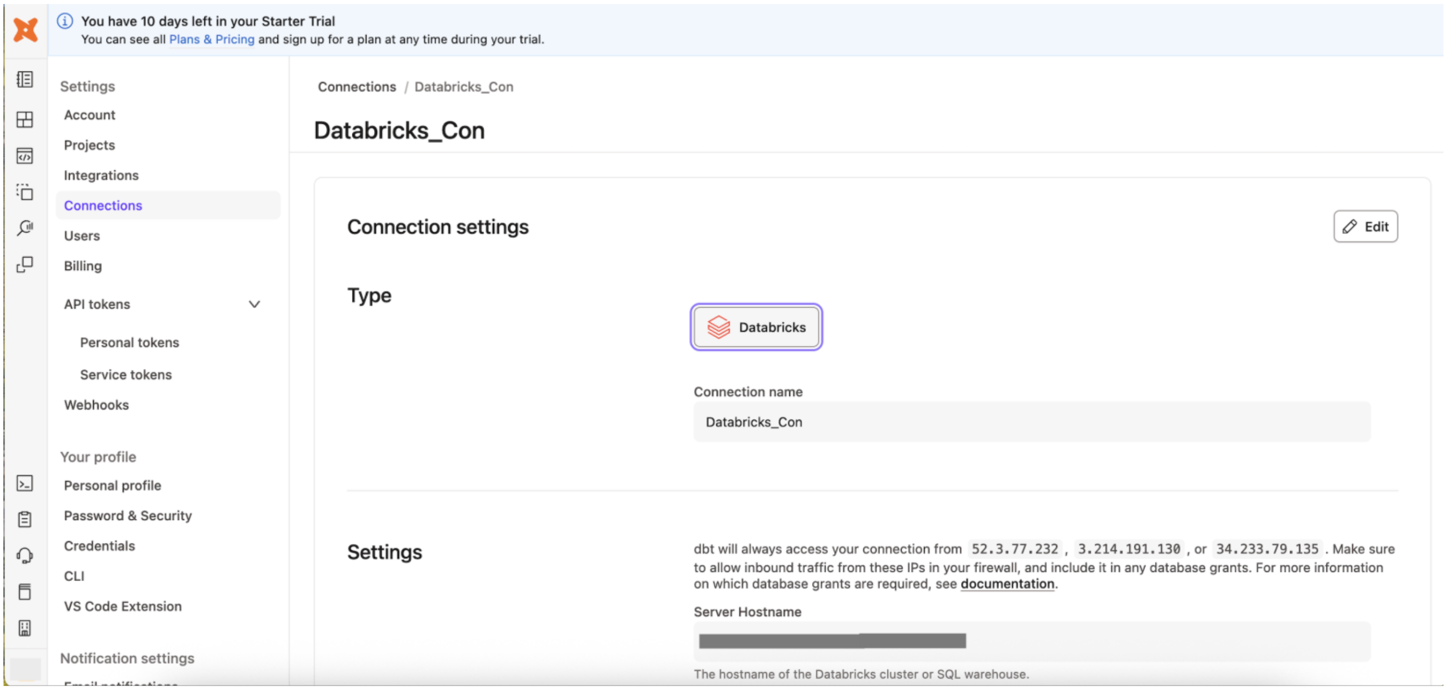
Silver layer tables after processing

Step 4: DBT Setup and Integration

I shifted to DBT Cloud (free tier) for the modeling phase. Connected DBT to Databricks using hostname, HTTP path, access token, catalog, and schema details.

- Initialized a DBT project in DBT Cloud Studio.
- Created a development branch for safe experimentation.
- Defined project structure: models/silver/, models/gold/, sources/, snapshots/, macros/.

- Customized schema generation with a macro generate_schema_name.sql to use clean names like silver without prefixes.



You have 10 days left in your Starter Trial
You can see all Plans & Pricing and sign up for a plan at any time during your trial.

Connections / Databricks_Con

Databricks_Con

Connection settings

Type: Databricks

Connection name: Databricks_Con

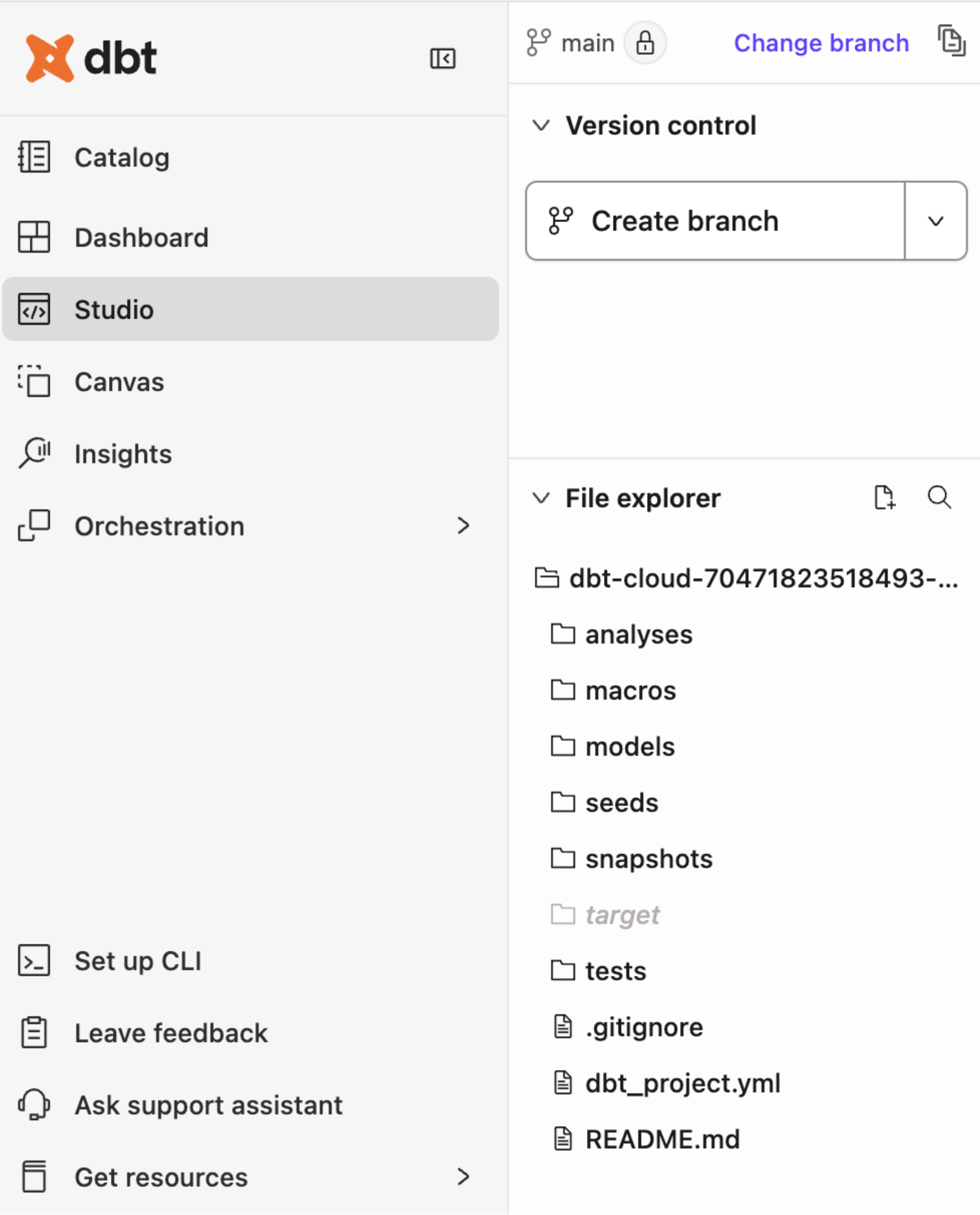
Settings

dbt will always access your connection from 52.3.77.232 , 3.214.191.130 , or 34.233.79.135 . Make sure to allow inbound traffic from these IPs in your firewall, and include it in any database grants. For more information on which database grants are required, see documentation.

Server Hostname: [redacted]

The hostname of the Databricks cluster or SQL warehouse.

DBT Cloud connection setup to Databricks

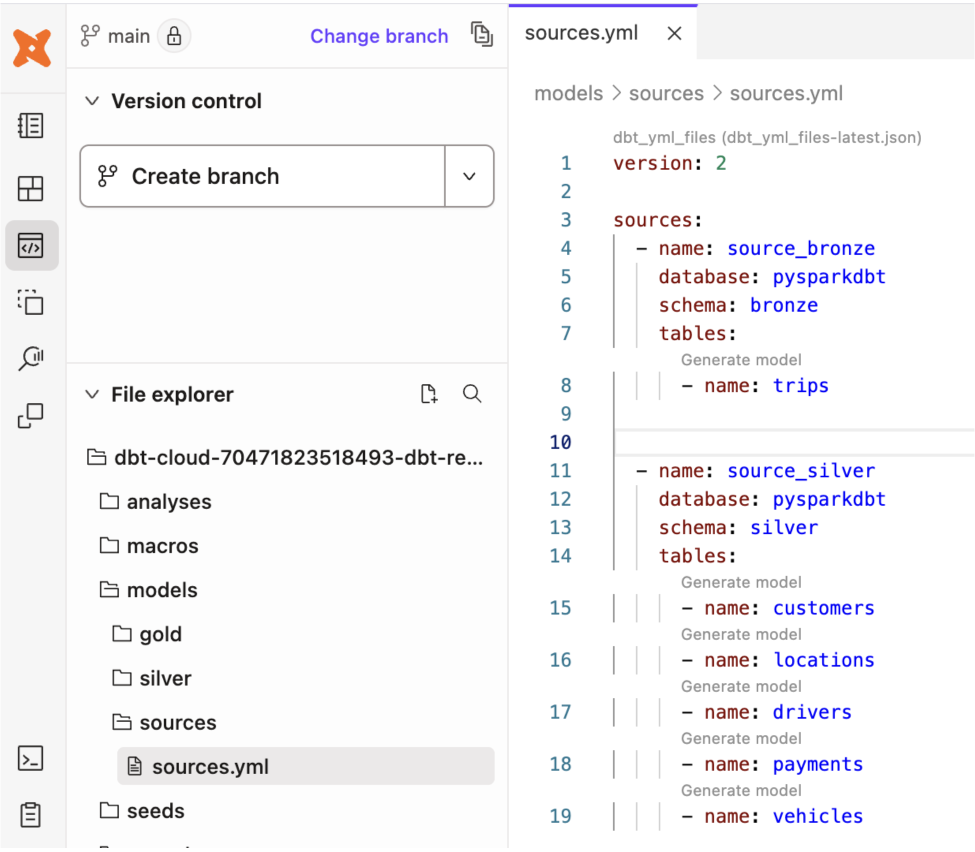


DBT project structure in Studio

Step 5: Gold Layer - Dimensional Modeling with DBT

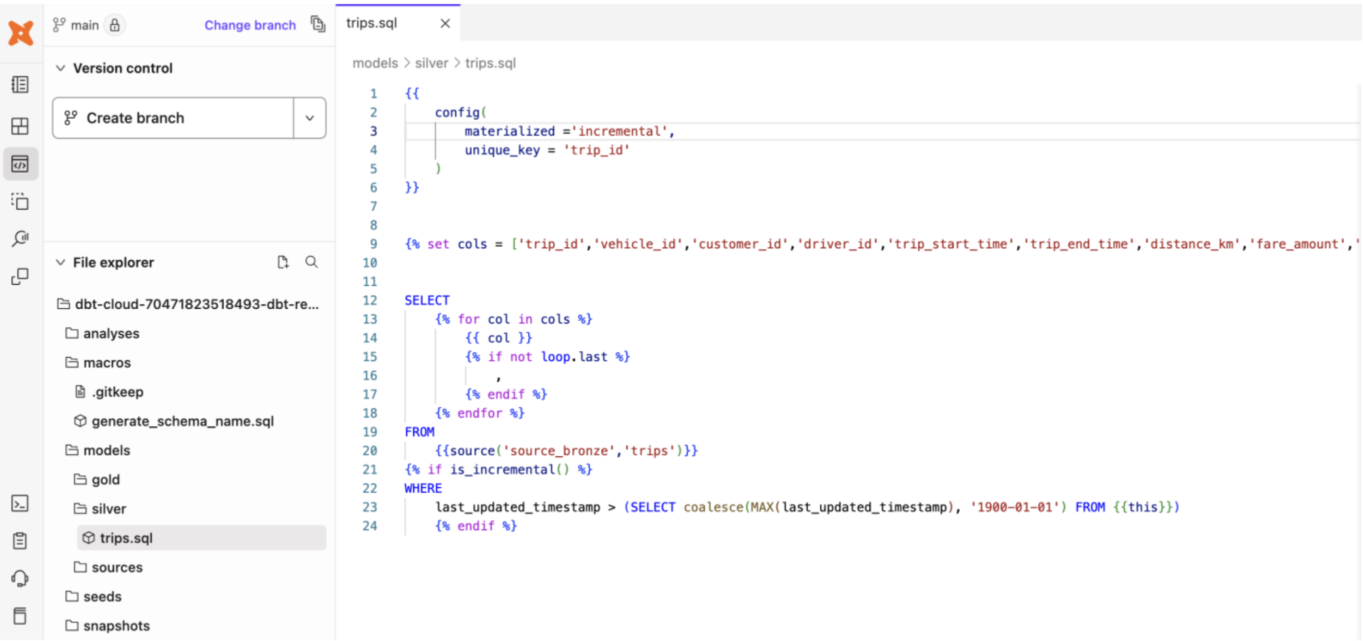
This step involved building a star schema with dimension and fact tables using DBT models, sources, and snapshots.

- Defined sources in sources.yml to register silver tables for lineage.
- Created incremental models (e.g., silver/trips.sql):
  - Used config(materialized='incremental', unique_key='trip_id').
  - Applied Jinja for dynamic SELECT: Looped over column lists with {% for %}...{% if not loop.last %},{% endif %}.
  - Filtered incremental data with is_incremental() macro and last_updated_timestamp.
- Implemented SCD Type 2 for dimensions via snapshots (e.g., dim_customers.yml):
  - Strategy: timestamp with unique_key and updated_at.
  - Added dbt_valid_from and dbt_valid_to for versioning.
- Created fact table fact_trips referencing silver models with ref('trips') and joining dimensions.
- Ran dbt run for models and dbt snapshot for SCDs.
- Viewed lineage and documentation in DBT.



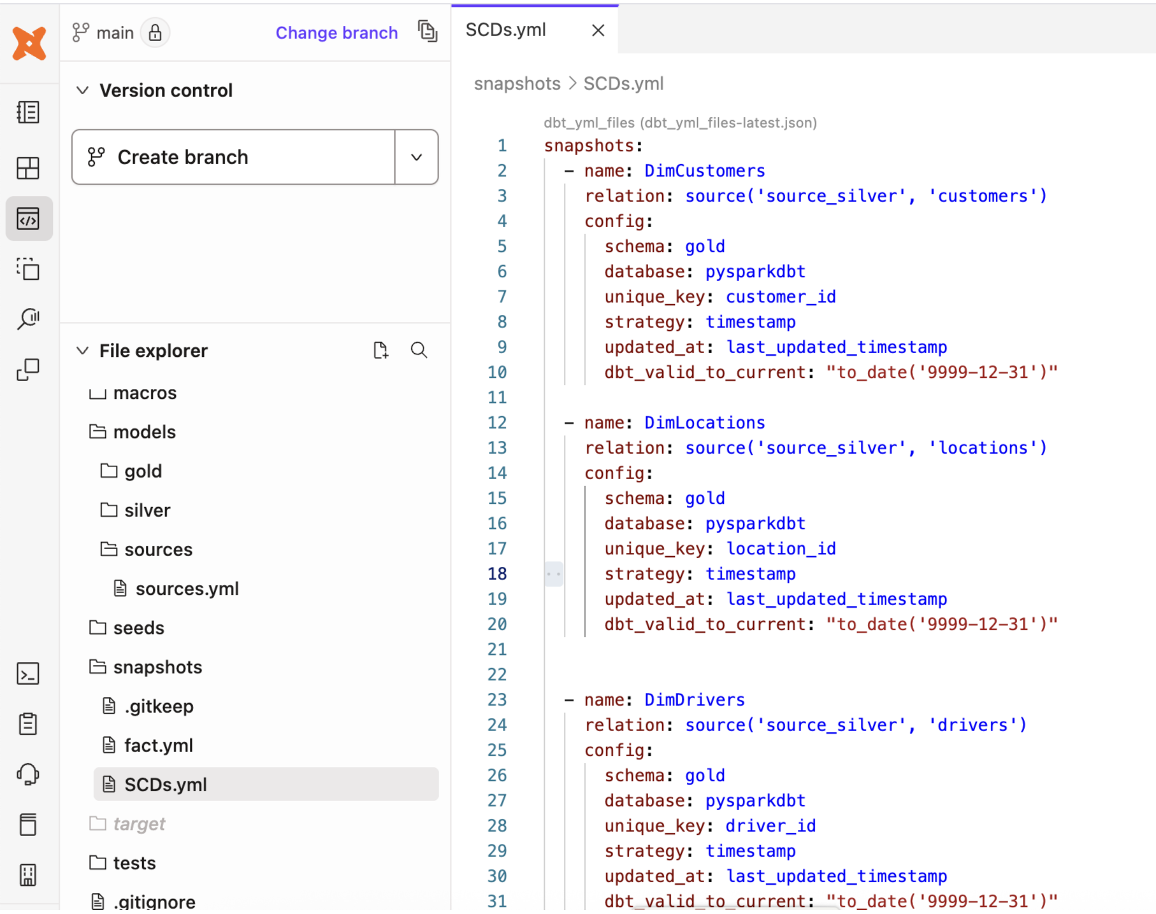
```
version: 2

sources:
  - name: source_bronze
    database: pysparkdbt
    schema: bronze
    tables:
      - name: trips


  - name: source_silver
    database: pysparkdbt
    schema: silver
    tables:
      - name: customers
      - name: locations
      - name: drivers
      - name: payments
      - name: vehicles
```

sources.yml file



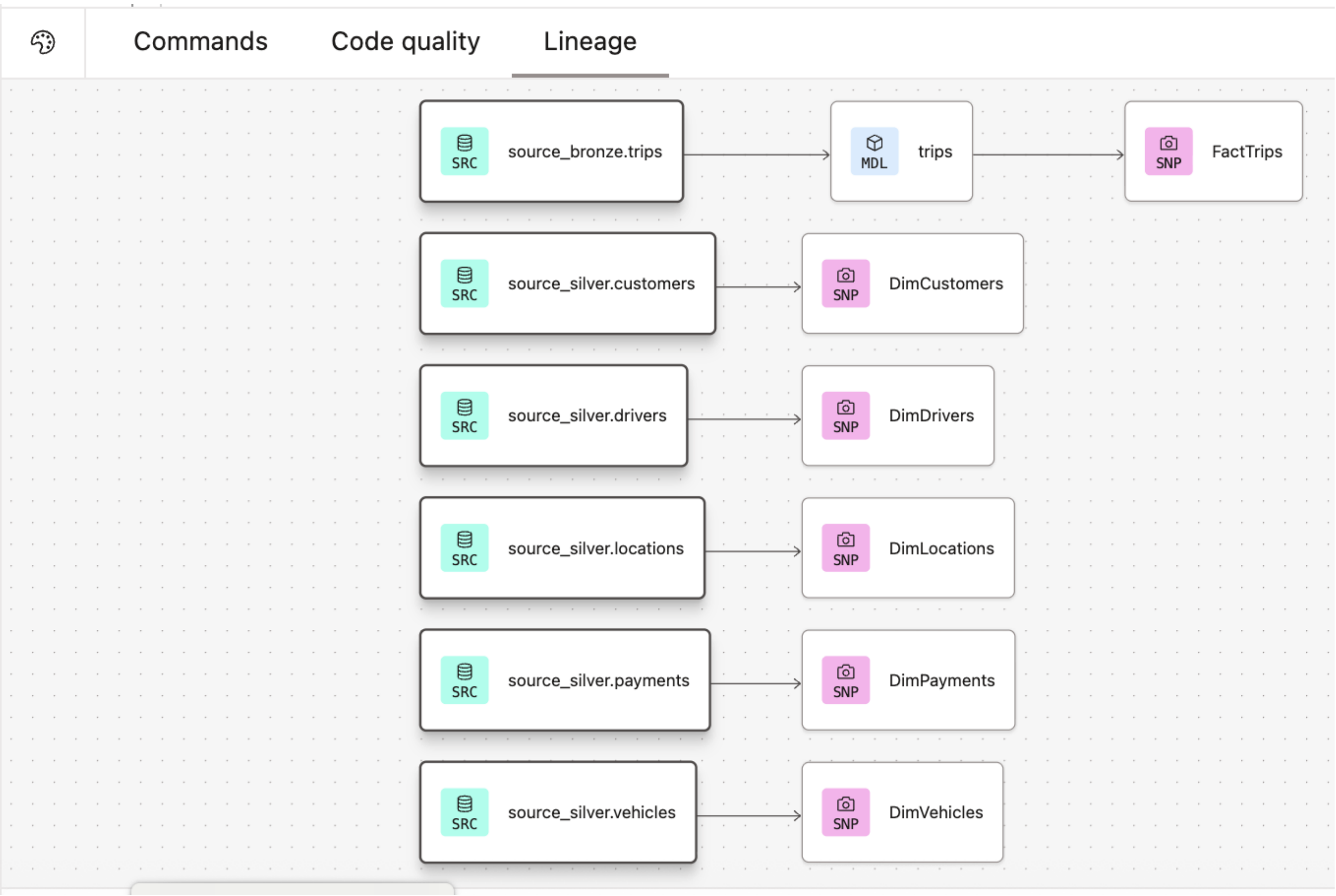
```sql
{{
    config(
        materialized ='incremental',
        unique_key = 'trip_id'
    )
}}


{% set cols = ['trip_id','vehicle_id','customer_id','driver_id','trip_start_time','trip_end_time','distance_km','fare_amount','


SELECT
    {% for col in cols %}
        {{ col }}
        {% if not loop.last %}
            ,
        {% endif %}
    {% endfor %}
FROM
    {{source('source_bronze','trips')}}
{% if is_incremental() %}
WHERE
    last_updated_timestamp > (SELECT coalesce(MAX(last_updated_timestamp), '1900-01-01') FROM {{this}})
    {% endif %}
```

Example DBT model with Jinja templating

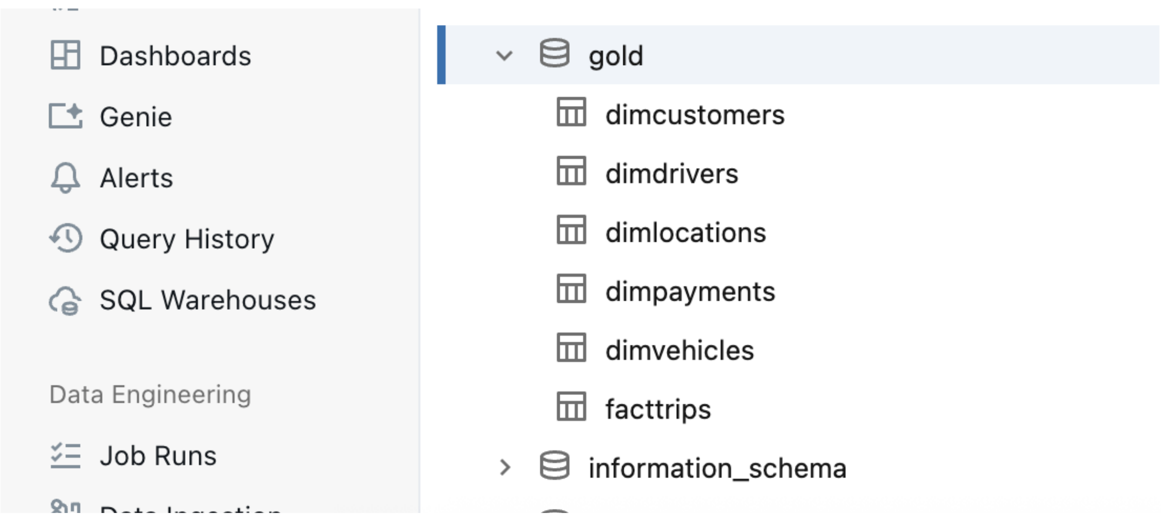


```yaml
snapshots:
  - name: DimCustomers
    relation: source('source_silver', 'customers')
    config:
      schema: gold
      database: pysparkdbt
      unique_key: customer_id
      strategy: timestamp
      updated_at: last_updated_timestamp
      dbt_valid_to_current: "to_date('9999-12-31')"

  - name: DimLocations
    relation: source('source_silver', 'locations')
    config:
      schema: gold
      database: pysparkdbt
      unique_key: location_id
      strategy: timestamp
      updated_at: last_updated_timestamp
      dbt_valid_to_current: "to_date('9999-12-31')"


  - name: DimDrivers
    relation: source('source_silver', 'drivers')
    config:
      schema: gold
      database: pysparkdbt
      unique_key: driver_id
      strategy: timestamp
      updated_at: last_updated_timestamp
      dbt_valid_to_current: "to_date('9999-12-31')"
```

DBT snapshot configuration for SCD



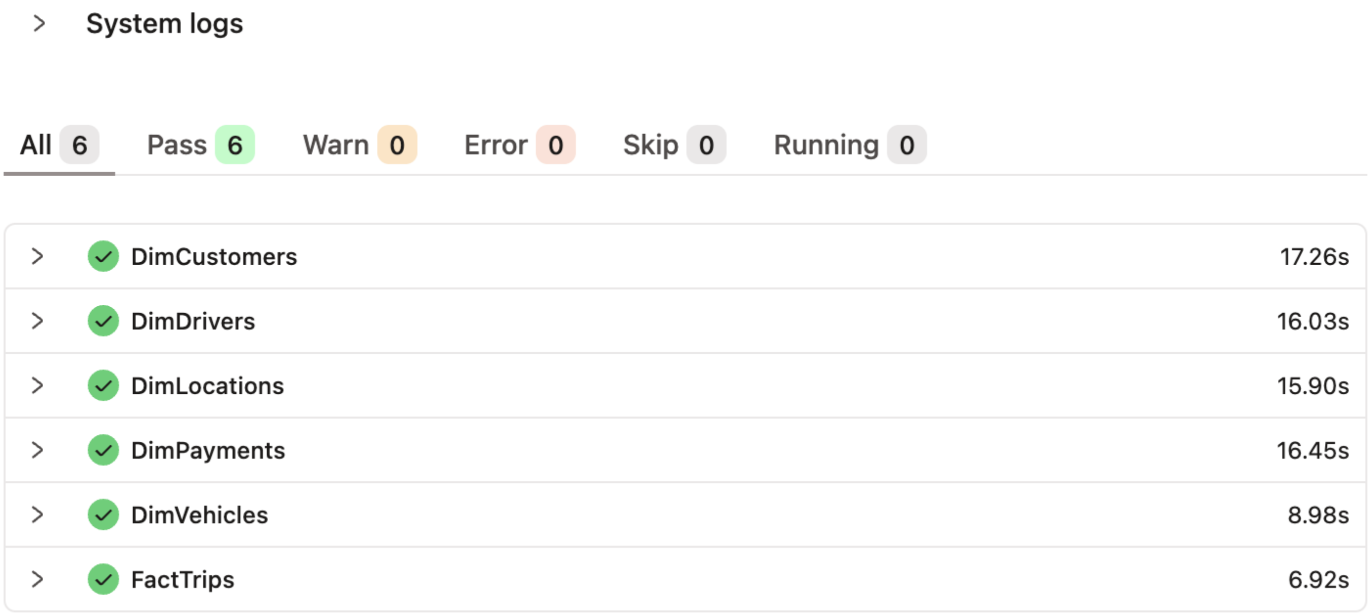
DBT lineage graph



Gold layer tables in Databricks (dimensions and facts)

Step 6: Final Deployment and Testing

- Committed changes to Git in DBT Cloud and merged to main.
- Ran full pipeline: Ingested new data in PySpark, then executed DBT jobs.
- Verified data in Databricks catalog and shared connection details for BI tools (e.g., Power BI).
- Ensured idempotency by re-running with no duplicates or errors.



> System logs

All 6 | Pass 6 | Warn 0 | Error 0 | Skip 0 | Running 0

✓	DimCustomers	17.26s
✓	DimDrivers	16.03s
✓	DimLocations	15.90s
✓	DimPayments	16.45s
✓	DimVehicles	8.98s
✓	FactTrips	6.92s

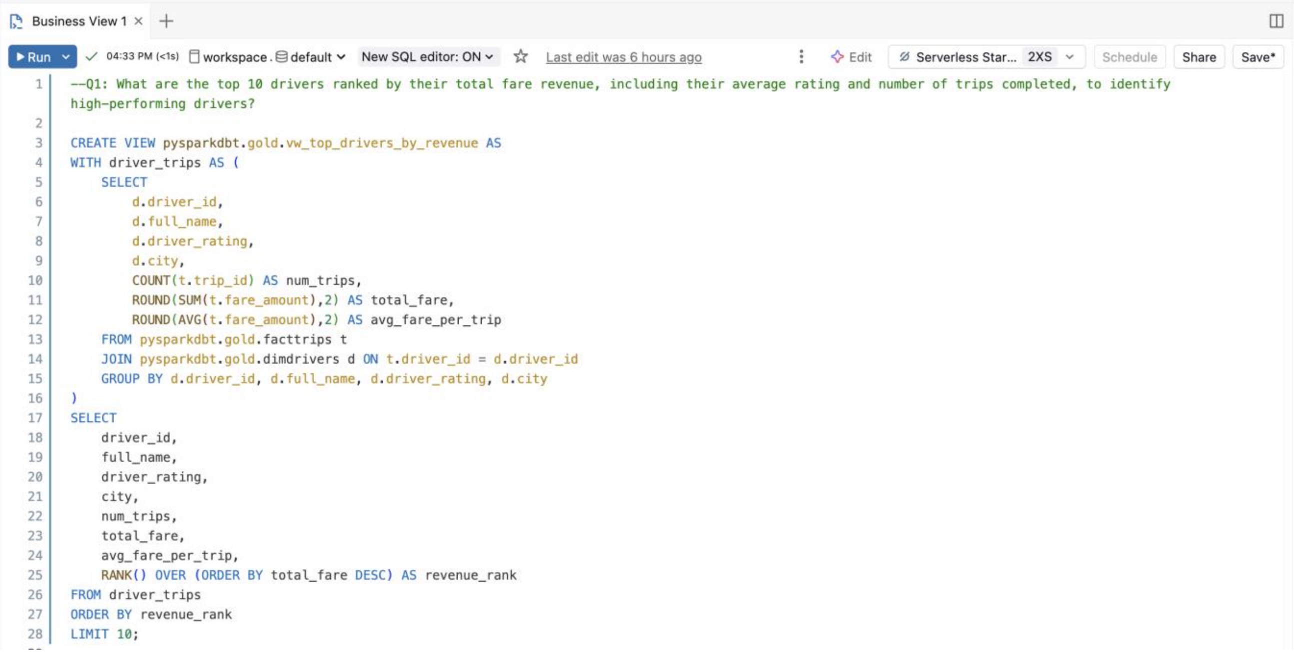
DBT run output and success logs

Step 7: Created Analytical Views for Business Intelligence

- Developed 3 SQL views in Databricks gold schema to support BI and analytics.
- Utilized **CTEs** for modular query structure and **window functions** (RANK(), ROW_NUMBER(), AVG() OVER) for rankings and rolling averages.

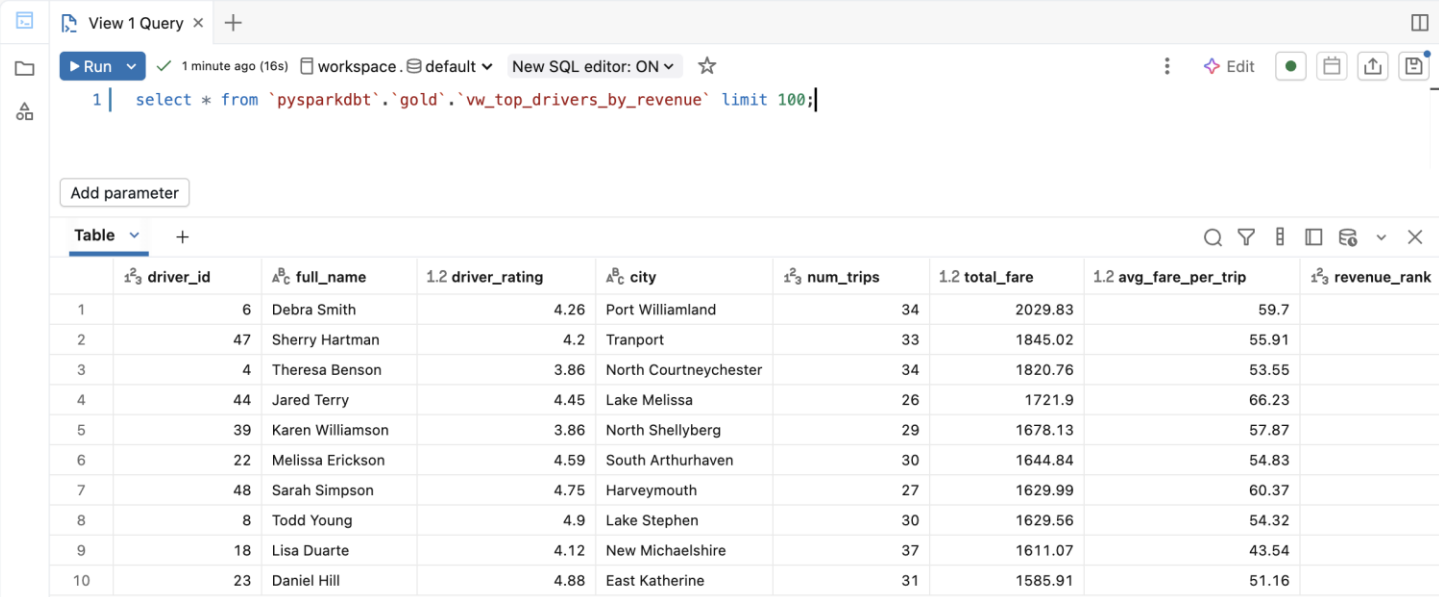
- Views answer key business questions:

1. **vw_top_drivers_by_revenue**: Top 10 drivers by total fare, trip counts, and ratings for performance analysis.



```sql
--Q1: What are the top 10 drivers ranked by their total fare revenue, including their average rating and number of trips completed, to identify
high-performing drivers?

CREATE VIEW pysparkdbt.gold.vw_top_drivers_by_revenue AS
WITH driver_trips AS (
    SELECT
        d.driver_id,
        d.full_name,
        d.driver_rating,
        d.city,
        COUNT(t.trip_id) AS num_trips,
        ROUND(SUM(t.fare_amount),2) AS total_fare,
        ROUND(AVG(t.fare_amount),2) AS avg_fare_per_trip
    FROM pysparkdbt.gold.facttrips t
    JOIN pysparkdbt.gold.dimdrivers d ON t.driver_id = d.driver_id
    GROUP BY d.driver_id, d.full_name, d.driver_rating, d.city
)
SELECT
    driver_id,
    full_name,
    driver_rating,
    city,
    num_trips,
    total_fare,
    avg_fare_per_trip,
    RANK() OVER (ORDER BY total_fare DESC) AS revenue_rank
FROM driver_trips
ORDER BY revenue_rank
LIMIT 10;
```

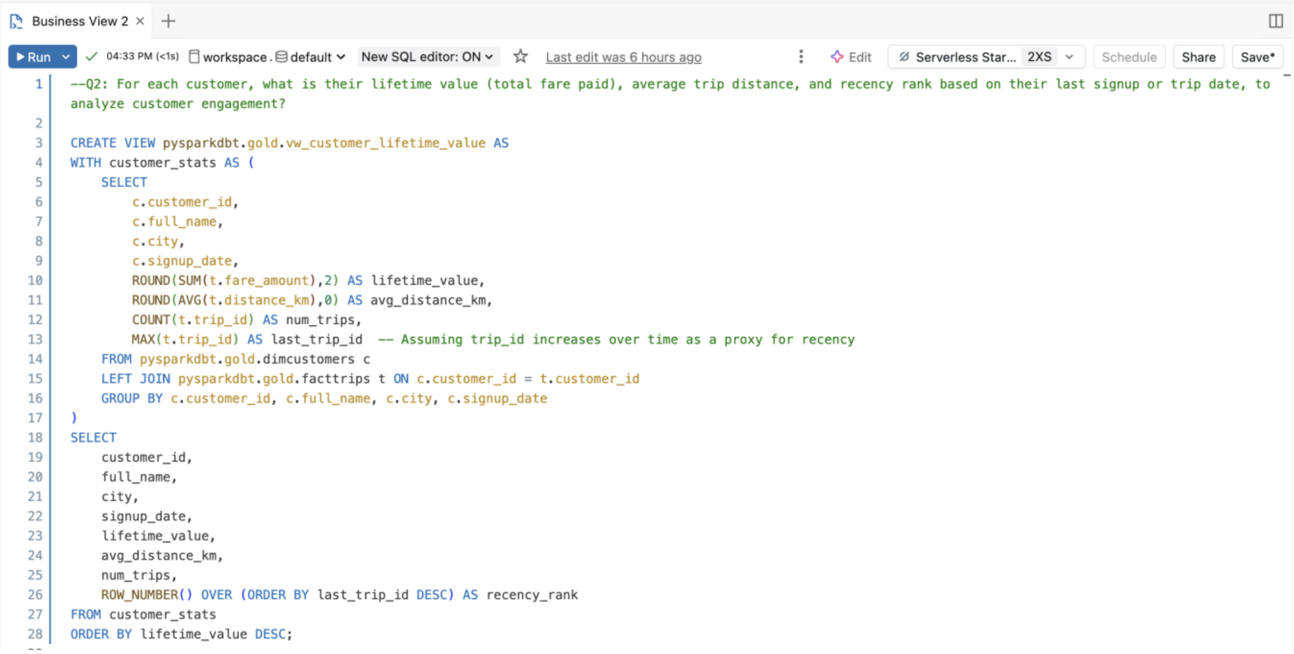
View 1 Query Results



```sql
select * from `pysparkdbt`.`gold`.`vw_top_drivers_by_revenue` limit 100;
```

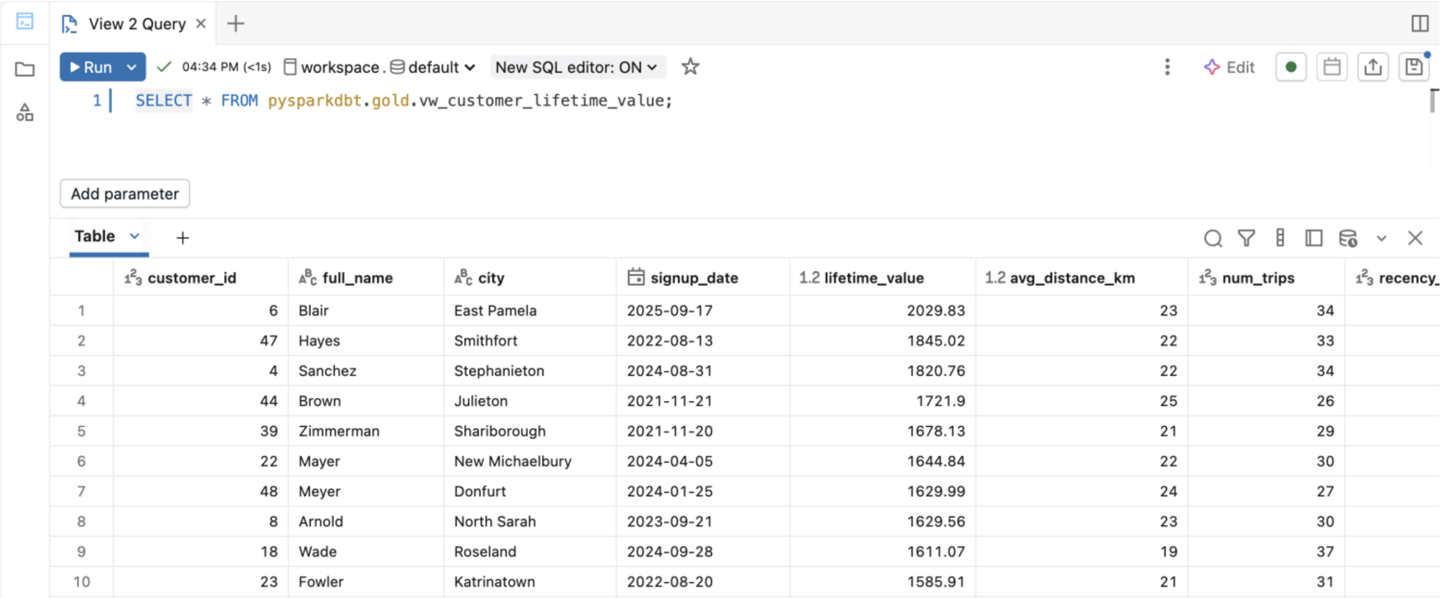
	driver_id	full_name	driver_rating	city	num_trips	total_fare	avg_fare_per_trip	revenue_rank
1	6	Debra Smith	4.26	Port Williamland	34	2029.83	59.7	
2	47	Sherry Hartman	4.2	Tranport	33	1845.02	55.91	
3	4	Theresa Benson	3.86	North Courtneychester	34	1820.76	53.55	
4	44	Jared Terry	4.45	Lake Melissa	26	1721.9	66.23	
5	39	Karen Williamson	3.86	North Shellyberg	29	1678.13	57.87	
6	22	Melissa Erickson	4.59	South Arthurhaven	30	1644.84	54.83	
7	48	Sarah Simpson	4.75	Harveymouth	27	1629.99	60.37	
8	8	Todd Young	4.9	Lake Stephen	30	1629.56	54.32	
9	18	Lisa Duarte	4.12	New Michaelshire	37	1611.07	43.54	
10	23	Daniel Hill	4.88	East Katherine	31	1585.91	51.16	

2. **vw_customer_lifetime_value**: Customer LTV, trip frequency, and recency for segmentation and retention strategies.



```sql
--Q2: For each customer, what is their lifetime value (total fare paid), average trip distance, and recency rank based on their last signup or trip date, to analyze customer engagement?

CREATE VIEW pysparkdbt.gold.vw_customer_lifetime_value AS
WITH customer_stats AS (
    SELECT
        c.customer_id,
        c.full_name,
        c.city,
        c.signup_date,
        ROUND(SUM(t.fare_amount),2) AS lifetime_value,
        ROUND(AVG(t.distance_km),0) AS avg_distance_km,
        COUNT(t.trip_id) AS num_trips,
        MAX(t.trip_id) AS last_trip_id  -- Assuming trip_id increases over time as a proxy for recency
    FROM pysparkdbt.gold.dimcustomers c
    LEFT JOIN pysparkdbt.gold.facttrips t ON c.customer_id = t.customer_id
    GROUP BY c.customer_id, c.full_name, c.city, c.signup_date
)
SELECT
    customer_id,
    full_name,
    city,
    signup_date,
    lifetime_value,
    avg_distance_km,
    num_trips,
    ROW_NUMBER() OVER (ORDER BY last_trip_id DESC) AS recency_rank
FROM customer_stats
ORDER BY lifetime_value DESC;
```

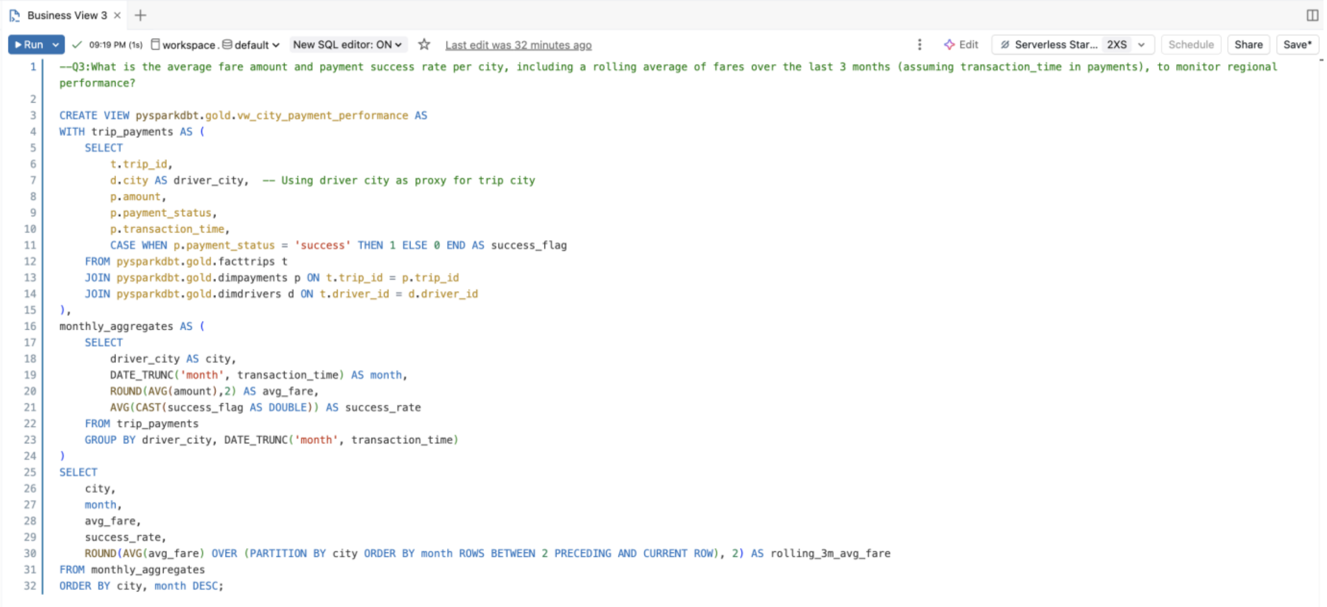
View 2 Query Results



```sql
SELECT * FROM pysparkdbt.gold.vw_customer_lifetime_value;
```

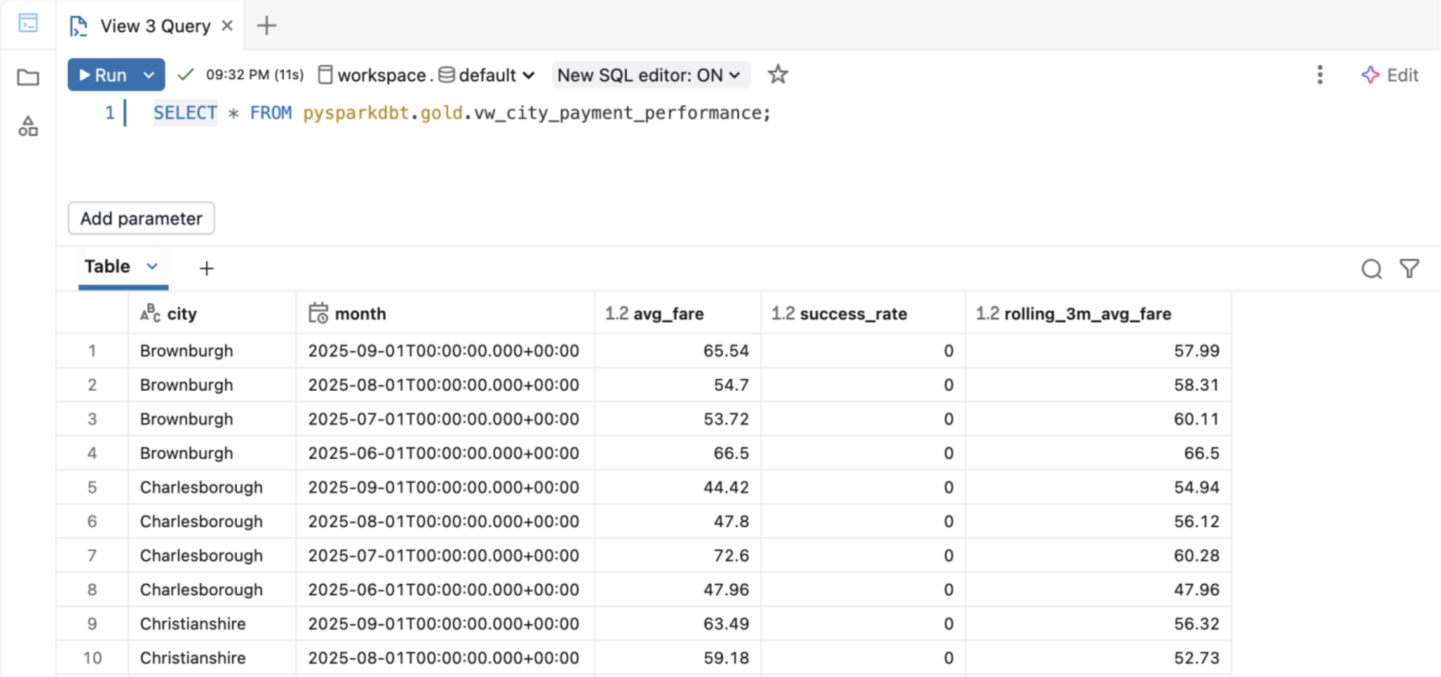
	customer_id	full_name	city	signup_date	lifetime_value	avg_distance_km	num_trips	recency_
1	6	Blair	East Pamela	2025-09-17	2029.83	23	34	
2	47	Hayes	Smithfort	2022-08-13	1845.02	22	33	
3	4	Sanchez	Stephanieton	2024-08-31	1820.76	22	34	
4	44	Brown	Julieton	2021-11-21	1721.9	25	26	
5	39	Zimmerman	Shariborough	2021-11-20	1678.13	21	29	
6	22	Mayer	New Michaelbury	2024-04-05	1644.84	22	30	
7	48	Meyer	Donfurt	2024-01-25	1629.99	24	27	
8	8	Arnold	North Sarah	2023-09-21	1629.56	23	30	
9	18	Wade	Roseland	2024-09-28	1611.07	19	37	
10	23	Fowler	Katrinatown	2022-08-20	1585.91	21	31	

3. **vw_city_payment_performance**: Regional payment success rates and 3-month rolling fare averages for operational insights.



```sql
--Q3:What is the average fare amount and payment success rate per city, including a rolling average of fares over the last 3 months (assuming transaction_time in payments), to monitor regional performance?

CREATE VIEW pysparkdbt.gold.vw_city_payment_performance AS
WITH trip_payments AS (
    SELECT
        t.trip_id,
        d.city AS driver_city,  -- Using driver city as proxy for trip city
        p.amount,
        p.payment_status,
        p.transaction_time,
        CASE WHEN p.payment_status = 'success' THEN 1 ELSE 0 END AS success_flag
    FROM pysparkdbt.gold.facttrips t
    JOIN pysparkdbt.gold.dimpayments p ON t.trip_id = p.trip_id
    JOIN pysparkdbt.gold.dimdrivers d ON t.driver_id = d.driver_id
),
monthly_aggregates AS (
    SELECT
        driver_city AS city,
        DATE_TRUNC('month', transaction_time) AS month,
        ROUND(AVG(amount),2) AS avg_fare,
        AVG(CAST(success_flag AS DOUBLE)) AS success_rate
    FROM trip_payments
    GROUP BY driver_city, DATE_TRUNC('month', transaction_time)
)
SELECT
    city,
    month,
    avg_fare,
    success_rate,
    ROUND(AVG(avg_fare) OVER (PARTITION BY city ORDER BY month ROWS BETWEEN 2 PRECEDING AND CURRENT ROW), 2) AS rolling_3m_avg_fare
FROM monthly_aggregates
ORDER BY city, month DESC;
```

View 3 Query Results



```sql
SELECT * FROM pysparkdbt.gold.vw_city_payment_performance;
```

	city	month	avg_fare	success_rate	rolling_3m_avg_fare
1	Brownburgh	2025-09-01T00:00:00.000+00:00	65.54	0	57.99
2	Brownburgh	2025-08-01T00:00:00.000+00:00	54.7	0	58.31
3	Brownburgh	2025-07-01T00:00:00.000+00:00	53.72	0	60.11
4	Brownburgh	2025-06-01T00:00:00.000+00:00	66.5	0	66.5
5	Charlesborough	2025-09-01T00:00:00.000+00:00	44.42	0	54.94
6	Charlesborough	2025-08-01T00:00:00.000+00:00	47.8	0	56.12
7	Charlesborough	2025-07-01T00:00:00.000+00:00	72.6	0	60.28
8	Charlesborough	2025-06-01T00:00:00.000+00:00	47.96	0	47.96
9	Christianshire	2025-09-01T00:00:00.000+00:00	63.49	0	56.32
10	Christianshire	2025-08-01T00:00:00.000+00:00	59.18	0	52.73

- Views consume the star schema models, providing analysts clean, aggregated data without complex joins.

Project Summary

This end-to-end data engineering project transforms raw pseudo-Uber CSV data into a fully modeled analytics pipeline using PySpark for ingestion and transformation, and DBT for advanced modeling and orchestration on Databricks. It demonstrates proficiency in handling incremental loads, data quality, SCD Type 2, and dynamic SQL with Jinja, resulting in a scalable, production-ready data warehouse. Key outcomes include 6 bronze tables, cleaned silver layers with upserts, and a gold star schema with historical tracking. This portfolio project highlights my ability to build efficient, modular data pipelines, making it ideal for data engineering roles requiring expertise in big data tools, cloud platforms, and modern data stacks. The entire codebase is available in the repository for review.