## **Advanced dbt Project: E-commerce Analytics on Trino-Nessie Lakehouse**

Current Date: Tuesday, June 3, 2025

Location Context: Chennai, Tamil Nadu, India

### **Introduction**

This guide builds upon our previous "Comprehensive Guide" for setting up a Trino-Nessie-MinIO lakehouse with dbt Core. Here, we will create a more complex dbt project focusing on sample e-commerce analytics. This will involve ingesting four new data sources and building a multi-layered dbt transformation pipeline with more intricate logic.

**Assumptions:**

* You have successfully completed Phases 1 and 2 of the "Comprehensive Guide," meaning your MinIO, Nessie, and Trino services are running and configured.
* You have dbt Core and dbt-trino adapter installed.
* Your profiles.yml is configured as per the previous guide.

### **Phase 1: New Sample Data & Enhanced Ingestion**

We'll define four new CSV files for our e-commerce data and create a new Spark script to ingest them.

**1.1 New Sample Data Files**

Create the following CSV files in your main my\_lakehouse\_project/ directory:

**a) raw\_customers.csv**

customer\_id,first\_name,last\_name,email,registration\_date,city  
101,Ravi,Kumar,ravi.k@example.com,2023-01-15,Chennai  
102,Priya,Sharma,priya.s@example.com,2023-02-20,Bangalore  
103,Amit,Patel,amit.p@example.com,2023-01-25,Mumbai  
104,Sunita,Das,sunita.d@example.com,2023-03-10,Chennai  
105,Vijay,Rao,vijay.r@example.com,2023-04-01,Hyderabad

**b) raw\_products.csv**

product\_id,product\_name,category,unit\_price  
201,Laptop Pro,Electronics,75000  
202,Wireless Mouse,Electronics,1200  
203,Organic Tea,Groceries,350  
204,Running Shoes,Apparel,4500  
205,Smartphone X,Electronics,55000

**c) raw\_orders.csv**

order\_id,customer\_id,order\_date,order\_status  
301,101,2024-05-01,completed  
302,102,2024-05-03,completed  
303,101,2024-05-10,shipped  
304,103,2024-05-12,pending  
305,104,2024-05-15,completed  
306,102,2024-05-20,completed  
307,105,2024-05-22,shipped

**d) raw\_order\_items.csv**

order\_item\_id,order\_id,product\_id,quantity,price\_per\_unit\_at\_order  
401,301,201,1,75000  
402,301,202,2,1200  
403,302,203,3,350  
404,302,204,1,4500  
405,303,205,1,55000  
406,304,201,1,75000  
407,305,202,1,1200  
408,306,203,2,350  
409,306,205,1,55000  
410,307,204,1,4500

**1.2 New Spark Ingestion Script (ingest\_ecommerce.py)**

Create a new Python file named ingest\_ecommerce.py in your my\_lakehouse\_project/ directory. This script will load all four CSVs.

from pyspark.sql import SparkSession  
from pyspark.sql.types import StructType, StructField, StringType, IntegerType, DateType, DoubleType  
  
# Define versions for required Spark packages. Ensure compatibility.  
ICEBERG\_VERSION = "1.5.0"   
NESSIE\_VERSION = "0.91.0"   
AWS\_SDK\_VERSION = "2.17.230" # Check for a version compatible with your Spark and Hadoop versions  
  
packages = [  
 f"org.apache.iceberg:iceberg-spark-runtime-3.4\_2.12:{ICEBERG\_VERSION}",  
 f"org.projectnessie.nessie-integrations:nessie-spark-extensions-3.4\_2.12:{NESSIE\_VERSION}",  
 f"software.amazon.awssdk:bundle:{AWS\_SDK\_VERSION}"  
]  
  
spark = SparkSession.builder \  
 .appName("EcommerceDataIngestionToNessie") \  
 .config("spark.jars.packages", ",".join(packages)) \  
 .config("spark.sql.extensions", "org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions,org.projectnessie.spark.extensions.NessieSparkSessionExtensions") \  
 .config("spark.sql.catalog.nessie", "org.apache.iceberg.spark.SparkCatalog") \  
 .config("spark.sql.catalog.nessie.catalog-impl", "org.apache.iceberg.nessie.NessieCatalog") \  
 .config("spark.sql.catalog.nessie.uri", "http://localhost:19120/api/v2") \  
 .config("spark.sql.catalog.nessie.ref", "main") \  
 .config("spark.sql.catalog.nessie.warehouse", "s3a://lakehouse/warehouse") \  
 .config("spark.hadoop.fs.s3a.endpoint", "http://localhost:9000") \  
 .config("spark.hadoop.fs.s3a.access.key", "minio") \  
 .config("spark.hadoop.fs.s3a.secret.key", "minio123") \  
 .config("spark.hadoop.fs.s3a.path.style.access", "true") \  
 .getOrCreate()  
  
print("Spark session created successfully for e-commerce data ingestion.")  
  
# Schemas for more robust ingestion (instead of inferSchema)  
customer\_schema = StructType([  
 StructField("customer\_id", IntegerType(), True),  
 StructField("first\_name", StringType(), True),  
 StructField("last\_name", StringType(), True),  
 StructField("email", StringType(), True),  
 StructField("registration\_date", DateType(), True),  
 StructField("city", StringType(), True)  
])  
  
product\_schema = StructType([  
 StructField("product\_id", IntegerType(), True),  
 StructField("product\_name", StringType(), True),  
 StructField("category", StringType(), True),  
 StructField("unit\_price", DoubleType(), True) # Changed to Double for price  
])  
  
order\_schema = StructType([  
 StructField("order\_id", IntegerType(), True),  
 StructField("customer\_id", IntegerType(), True),  
 StructField("order\_date", DateType(), True),  
 StructField("order\_status", StringType(), True)  
])  
  
order\_item\_schema = StructType([  
 StructField("order\_item\_id", IntegerType(), True),  
 StructField("order\_id", IntegerType(), True),  
 StructField("product\_id", IntegerType(), True),  
 StructField("quantity", IntegerType(), True),  
 StructField("price\_per\_unit\_at\_order", DoubleType(), True) # Changed to Double  
])  
  
# Data to ingest: (csv\_filename, schema\_object, table\_name)  
datasets\_to\_ingest = [  
 ("raw\_customers.csv", customer\_schema, "customers"),  
 ("raw\_products.csv", product\_schema, "products"),  
 ("raw\_orders.csv", order\_schema, "orders"),  
 ("raw\_order\_items.csv", order\_item\_schema, "order\_items")  
]  
  
for csv\_file, schema, table\_name in datasets\_to\_ingest:  
 print(f"\nIngesting {csv\_file} into nessie.raw\_data.{table\_name}...")  
 df = spark.read.csv(csv\_file, header=True, schema=schema)  
   
 print(f"Schema for {table\_name}:")  
 df.printSchema()  
 print(f"Sample data for {table\_name}:")  
 df.show(3)  
   
 table\_identifier = f"nessie.raw\_data.{table\_name}"  
 df.write.format("iceberg").mode("overwrite").save(table\_identifier)  
 print(f"Successfully ingested data into Iceberg table: {table\_identifier} on Nessie branch 'main'.")  
  
spark.stop()  
print("\nAll e-commerce data ingested. Spark session stopped.")

*Description:* This script is similar to the previous one but defines explicit schemas for each CSV for better data integrity and ingests all four new datasets into the nessie.raw\_data schema.

1.3 Running the New Ingestion Script

From your my\_lakehouse\_project/ directory, run:

spark-submit ingest\_ecommerce.py

*Description:* This will load all four e-commerce tables into your lakehouse.

1.4 Verifying Ingestion in Trino

Access the Trino CLI (docker exec -it trino trino-cli) and check if the tables exist:

SHOW TABLES IN nessie.raw\_data;  
-- Expected: customers, products, orders, order\_items (and the old 'users' table if not cleared)  
  
SELECT \* FROM nessie.raw\_data.customers LIMIT 2;  
SELECT \* FROM nessie.raw\_data.products LIMIT 2;  
SELECT \* FROM nessie.raw\_data.orders LIMIT 2;  
SELECT \* FROM nessie.raw\_data.order\_items LIMIT 2;

### **Phase 2: Expanding Your dbt Project**

Now, let's update your dbt project (my\_lakehouse\_dbt\_project/) to transform this new e-commerce data.

2.1 Update models/staging/sources.yml

Add the new source tables:

version: 2  
  
sources:  
 - name: raw\_lakehouse\_data # Existing group  
 schema: raw\_data   
 tables:  
 - name: users # From previous guide  
 description: "Raw user data ingested from CSV via Spark."  
 # New E-commerce tables  
 - name: customers  
 description: "Raw customer data for the e-commerce platform."  
 - name: products  
 description: "Raw product catalog information."  
 - name: orders  
 description: "Raw order header data."  
 - name: order\_items  
 description: "Raw order line item data, linking orders to products."

2.2 Create New Staging Models (models/staging/)

These will all be ephemeral as per our dbt\_project.yml configuration.

**a) models/staging/stg\_customers.sql**

-- Staging model for customers  
-- Renames columns for consistency and performs basic type casting if needed.  
select  
 customer\_id,  
 first\_name,  
 last\_name,  
 email,  
 registration\_date,  
 city  
from {{ source('raw\_lakehouse\_data', 'customers') }}

**b) models/staging/stg\_products.sql**

-- Staging model for products  
select  
 product\_id,  
 product\_name,  
 category,  
 unit\_price  
from {{ source('raw\_lakehouse\_data', 'products') }}

**c) models/staging/stg\_orders.sql**

-- Staging model for orders  
select  
 order\_id,  
 customer\_id,  
 order\_date,  
 order\_status  
from {{ source('raw\_lakehouse\_data', 'orders') }}

**d) models/staging/stg\_order\_items.sql**

-- Staging model for order items  
select  
 order\_item\_id,  
 order\_id,  
 product\_id,  
 quantity,  
 price\_per\_unit\_at\_order  
from {{ source('raw\_lakehouse\_data', 'order\_items') }}

2.3 Create Intermediate Models (models/intermediate/)

Intermediate models are useful for complex joins or pre-aggregations that might be used by multiple mart models. Create a new folder intermediate inside models.

a) models/intermediate/int\_order\_items\_with\_products.sql

This model joins order items with product details. Let's make this one ephemeral too for now, or it could be a table if heavily reused.

-- Enriches order items with product information like name and category.  
select  
 oi.order\_item\_id,  
 oi.order\_id,  
 oi.product\_id,  
 p.product\_name,  
 p.category as product\_category,  
 oi.quantity,  
 oi.price\_per\_unit\_at\_order,  
 (oi.quantity \* oi.price\_per\_unit\_at\_order) as line\_item\_total  
from {{ ref('stg\_order\_items') }} oi  
left join {{ ref('stg\_products') }} p   
 on oi.product\_id = p.product\_id

2.4 Create New Mart Models (models/marts/)

These are the final, analytics-ready tables.

a) models/marts/fct\_orders.sql

A fact table summarizing order details, including customer information and total amounts.

-- Fact table for orders, enriched with customer details and order totals.  
with order\_totals as (  
 select  
 order\_id,  
 sum(line\_item\_total) as order\_total\_amount,  
 sum(quantity) as total\_items\_in\_order  
 from {{ ref('int\_order\_items\_with\_products') }}  
 group by 1  
)  
select  
 o.order\_id,  
 o.customer\_id,  
 c.first\_name as customer\_first\_name,  
 c.city as customer\_city,  
 o.order\_date,  
 o.order\_status,  
 ot.order\_total\_amount,  
 ot.total\_items\_in\_order,  
 -- Example of a window function: Rank orders by total amount for each customer  
 rank() over (partition by o.customer\_id order by ot.order\_total\_amount desc) as customer\_order\_rank\_by\_value  
from {{ ref('stg\_orders') }} o  
left join {{ ref('stg\_customers') }} c   
 on o.customer\_id = c.customer\_id  
left join order\_totals ot   
 on o.order\_id = ot.order\_id  
where o.order\_status in ('completed', 'shipped') -- Only considering fulfilled or in-progress orders

b) models/marts/dim\_customers\_summary.sql

A dimension table summarizing customer activity.

-- Dimension table for customers, summarizing their order history.  
select  
 c.customer\_id,  
 c.first\_name,  
 c.last\_name,  
 c.email,  
 c.registration\_date,  
 c.city,  
 count(distinct fo.order\_id) as total\_orders,  
 sum(fo.order\_total\_amount) as total\_lifetime\_value,  
 avg(fo.order\_total\_amount) as average\_order\_value,  
 min(fo.order\_date) as first\_order\_date,  
 max(fo.order\_date) as latest\_order\_date  
from {{ ref('stg\_customers') }} c  
left join {{ ref('fct\_orders') }} fo -- Joins to the fact table we just defined  
 on c.customer\_id = fo.customer\_id  
group by 1,2,3,4,5,6

2.5 (Optional) Example of an Incremental Mart Model

Let's imagine fct\_orders could become very large. Here's a conceptual sketch of how it might look as an incremental model (this would replace the fct\_orders.sql above if you choose to implement it).

**models/marts/fct\_orders\_incremental.sql (Conceptual)**

{{  
 config(  
 materialized='incremental',  
 incremental\_strategy='merge',  
 unique\_key='order\_id'   
 -- For Iceberg with Trino, often a timestamp-based merge condition is better  
 -- or a more complex merge strategy might be needed if order\_status can change.  
 )  
}}  
  
with order\_totals as (  
 select  
 order\_id,  
 sum(line\_item\_total) as order\_total\_amount,  
 sum(quantity) as total\_items\_in\_order  
 from {{ ref('int\_order\_items\_with\_products') }}  
 {% if is\_incremental() %}  
 -- Filter order\_items based on the order\_date of orders we need to process  
 where exists (  
 select 1 from {{ ref('stg\_orders') }} so  
 where so.order\_id = int\_order\_items\_with\_products.order\_id   
 and so.order\_date >= (select date\_add('day', -3, max(order\_date)) from {{ this }})  
 -- The above filter is a simple example; a robust one would track processed order\_ids or modification timestamps.  
 )  
 {% endif %}  
 group by 1  
)  
select  
 o.order\_id,  
 o.customer\_id,  
 c.first\_name as customer\_first\_name,  
 c.city as customer\_city,  
 o.order\_date,  
 o.order\_status,  
 ot.order\_total\_amount,  
 ot.total\_items\_in\_order,  
 rank() over (partition by o.customer\_id order by ot.order\_total\_amount desc) as customer\_order\_rank\_by\_value  
from {{ ref('stg\_orders') }} o  
left join {{ ref('stg\_customers') }} c   
 on o.customer\_id = c.customer\_id  
left join order\_totals ot   
 on o.order\_id = ot.order\_id  
where o.order\_status in ('completed', 'shipped')  
  
{% if is\_incremental() %}  
 -- This condition ensures we only process orders that are new or recently updated.  
 -- A more robust approach might use a dedicated 'last\_modified\_at' timestamp.  
 and o.order\_date >= (select date\_add('day', -3, max(order\_date)) from {{ this }})  
{% endif %}

*Note: Implementing robust incremental logic, especially for MERGE strategies with potential updates to existing rows (like order\_status changing), requires careful consideration of your data's update patterns and often involves using modification timestamps.*

### **Phase 3: Running the Expanded dbt Project**

1. Create Dev Branch (if not already done for your user):  
   In Trino CLI: CALL nessie.system.create\_branch('dev\_your\_user', 'main');
2. Run dbt:  
   From your my\_lakehouse\_dbt\_project directory:  
   dbt run --target dev   
   # Or just 'dbt run' if dev is your default and the branch exists
3. Verify in Trino:  
   Check your new mart tables in the analytics\_dev\_your\_user schema on your dev branch.  
   -- In Trino CLI, replace 'your\_user'  
   SELECT \* FROM nessie.analytics\_dev\_your\_user.fct\_orders AT BRANCH "dev\_your\_user" LIMIT 5;  
   SELECT \* FROM nessie.analytics\_dev\_your\_user.dim\_customers\_summary AT BRANCH "dev\_your\_user" LIMIT 5;

### **Conclusion**

This expanded project demonstrates a more realistic multi-layered dbt transformation pipeline. It includes:

* Multiple data sources.
* Staging models for basic cleanup.
* Intermediate models for common joins/pre-aggregations.
* Mart models with business logic, including aggregations and window functions.
* A conceptual example of an incremental model.

This provides a solid foundation for building out more complex analytics on your Trino-Nessie-Iceberg lakehouse. Remember to adapt the logic and incremental strategies to the specific characteristics of your actual data.