■ Snowpark RFM Project — Enterprise Data Engineering & Feature Creation Case Study

■ Executive Summary

This document presents a complete, end-to-end implementation of a **Snowpark for Python project** focused on building **customer churn prediction features (RFM metrics)** using Snowflake's native computation engine.
It combines architectural insights, data ingestion workflows, Snowpark DataFrame transformations, and hands-on feature engineering aligned with the **SnowPro Advanced Data Engineer** principles.

■■ 1. Project Overview

Objective

Design and implement a **data preprocessing pipeline** that leverages Snowpark for Python to transform raw JSON sales data into structured analytical features for a **machine learning churn model**.

Scope

- Load JSON/NDJSON data into Snowflake.
- Perform programmatic transformations via Snowpark DataFrames.
- Compute **Top 10 Selling Products** and **Customer RFM Metrics**.
- Persist engineered features for downstream analytics and ML.

Alignment to SnowPro Advanced Data Engineer

Certification Domain	Coverage
----------------------	-----------
Snowpark Architecture (Client vs Server)	■
Snowpark DataFrame API (Select, Filter, Join)	■
Aggregations with Snowpark	■
Lazy vs Eager Execution	■
Data Engineering for ML Feature Creation	■

■ 2. Snowpark Architecture — Client vs Server

Client (Notebook / Python Environment)

- The **client** (in this case, a Snowsight Python worksheet) constructs logical execution plans using the Snowpark API.
- These logical plans are serialized and **sent to Snowflake** for execution.

Server (Snowflake Compute Engine)

- Snowflake **executes the entire plan server-side**, ensuring scalability, security, and reduced data movement.
- Only the results are returned to the client.

Execution Modes

Type	Method	Description
------	---------	-------------
Lazy	DataFrame operations build logical plans only.	No data fetched until `.show()` or `.collect()` is called.
Eager	Explicit execution using `.show()`, `.collect()`, `.count()`.	Executes SQL inside Snowflake and returns results.

■■ 3. Environment Setup (Snowflake + Snowpark)

Environment Requirements

Component	Description
------------	-------------
Snowflake Account	With Anaconda Integration Enabled
Python	Snowpark Runtime in Snowsight
Warehouse	`COMPUTE_WH`
Database	`POS_DEMO`
Schemas	`RAW`, `PROD`
Stage	`RAW.JSON_STAGE`

Setup Steps (as implemented in notebook)

```
CREATE DATABASE IF NOT EXISTS POS_DEMO;
CREATE SCHEMA IF NOT EXISTS POS_DEMO.RAW;
CREATE SCHEMA IF NOT EXISTS POS_DEMO.PROD;
CREATE OR REPLACE STAGE POS_DEMO.RAW.JSON_STAGE;
```

The notebook automatically ensures environment setup using Snowpark's `session.sql()` calls.

■ 4. Data Ingestion & Normalization

File Format Creation

CREATE OR REPLACE FILE FORMAT POS_DEMO.RAW.FF_JSON_NDJSON
TYPE = 'JSON'
STRIP_OUTER_ARRAY = FALSE;
This allows ingestion of newline-delimited JSON (NDJSON).

Load Raw JSON

The files `FCT_SALES_10000.ndjson` and `DIM_PRODUCT.json` are uploaded to the stage `@POS_DEMO.RAW.JSON_STAGE`.

COPY INTO POS_DEMO.RAW.RAW_JSON
FROM @POS_DEMO.RAW.JSON_STAGE/FCT_SALES_10000.ndjson
FILE_FORMAT = (FORMAT_NAME = 'POS_DEMO.RAW.FF_JSON_NDJSON')
ON_ERROR = 'CONTINUE';

Normalize into Structured Tables

The notebook parses `RAW_JSON` into two normalized tables:
- `POS_DEMO.RAW.FCT_SALES`
- `POS_DEMO.RAW.DIM_PRODUCT`

Example transformation snippet:
INSERT INTO POS_DEMO.RAW.FCT_SALES (SALE_ID, CUSTOMER_ID, PRODUCT_ID, SALE_DATE, QUANTITY, UNIT_PRICE, TOTAL_AMOUNT, REGION)
SELECT
raw:"SALE_ID"::STRING,
raw:"CUSTOMER_ID"::STRING,
raw:"PRODUCT_ID"::STRING,
TO_DATE(raw:"SALE_DATE"::STRING,'YYYY-MM-DD'),
raw:"QUANTITY"::NUMBER,
raw:"UNIT_PRICE"::FLOAT,
raw:"TOTAL_AMOUNT"::FLOAT,
raw:"REGION"::STRING
FROM POS_DEMO.RAW.RAW_JSON
WHERE raw:"SALE_ID" IS NOT NULL;

■ 5. Snowpark DataFrame Operations

Session Initialization

from snowflake.snowpark import Session
session = Session.builder.configs({}).create()

DataFrame Loading

```
sales_df = session.table("POS_DEMO.RAW.FCT_SALES")
prod_df = session.table("POS_DEMO.RAW.DIM_PRODUCT")
```

Lazy vs Eager Example

```
expr = sales_df.select("SALE_ID", "CUSTOMER_ID",
"PRODUCT_ID").filter(sales_df["TOTAL_AMOUNT"] > 0)
expr.show(5) # Eager execution
expr.explain() # Server-side SQL plan
```

■ 6. Aggregations — Top 10 Selling Products (October 2025)

Snowpark Implementation

```
from snowflake.snowpark.functions import col, sum as ssum, lit

top10_oct = (
sales_df
.filter((col("SALE_DATE") >= lit("2025-10-01")) & (col("SALE_DATE") <= lit("2025-10-31")))
.group_by("PRODUCT_ID")
.agg(
ssum(col("QUANTITY")).alias("TOTAL_QTY"),
ssum(col("TOTAL_AMOUNT")).alias("TOTAL_SALES")
)
.order_by(col("TOTAL_SALES").desc())
.limit(10)
)

top10_with_name = top10_oct.join(prod_df, top10_oct["PRODUCT_ID"] ==
prod_df["PRODUCT_ID"], how="left") .select(top10_oct["PRODUCT_ID"],
prod_df["PRODUCT_NAME"], col("TOTAL_QTY"), col("TOTAL_SALES"))

top10_with_name.show()
```

Outcome

PRODUCT_ID	PRODUCT_NAME	TOTAL_QTY	TOTAL_SALES
------------	--------------	----------	-------------
P003	Noise Cancelling Headphones	1250	124,560
P004	Smartwatch	980	110,240
P001	Wireless Mouse	915	103,850

■ 7. Feature Engineering — RFM (Recency, Frequency, Monetary)

Snowpark Computation

```
from snowflake.snowpark.functions import max as smax, count as scount, sum as ssum,
datediff, lit

analysis_date = "2025-11-04"
rfm_df = (
sales_df.group_by("CUSTOMER_ID")
.agg(
smax(col("SALE_DATE")).alias("LAST_PURCHASE_DATE"),
scount(col("SALE_ID")).alias("FREQUENCY"),
ssum(col("TOTAL_AMOUNT")).alias("MONETARY")
)
.with_column("RECENCY_DAYS", datediff(lit("day"), col("LAST_PURCHASE_DATE"),
lit(analysis_date)))
.select("CUSTOMER_ID", "LAST_PURCHASE_DATE", "RECENCY_DAYS", "FREQUENCY",
"MONETARY")
)
rfm_df.show(10)
```

Adding Churn Label

```
from snowflake.snowpark.functions import when
rfm_labeled = rfm_df.with_column("CHURN_LABEL", when(col("RECENCY_DAYS") > 30,
1).otherwise(0))
rfm_labeled.show(5)
```

■ 8. Persisting Results

```
top10_with_name.write.mode("overwrite").save_as_table("POS_DEMO.PROD.TOP10_PROD
UCTS_OCT2025")
rfm_labeled.write.mode("overwrite").save_as_table("POS_DEMO.PROD.RFM_CUSTOMER_
FEATURES_LABELED")
```

■ These tables are now accessible to data scientists for feature selection and modeling.

■ 9. Validation & Explain Plan

Record Checks

```
SELECT COUNT(*) FROM POS_DEMO.RAW.FCT_SALES;
SELECT COUNT(*) FROM POS_DEMO.PROD.RFM_CUSTOMER_FEATURES_LABELED;
SELECT CHURN_LABEL, COUNT(*) FROM
POS_DEMO.PROD.RFM_CUSTOMER_FEATURES_LABELED GROUP BY
CHURN_LABEL;
```

Explain Plan Example

print(top10_with_name.explain())
Shows the server-side SQL translation confirming pushdown execution.

■ 10. Key Learnings & Best Practices

Concept	Description
----------	--------------
Server-side computation	All logic is executed in Snowflake, minimizing data transfer.
Lazy evaluation	Operations compile to SQL; execution triggered only when needed.
Scalability	Snowpark leverages Snowflake’s compute scalability.
Security	Data never leaves Snowflake.
Reusability	Modular code design enables quick adaptation for other data marts.

■ 11. Extensions & Future Enhancements

1. Add **Average Order Value** feature (`MONETARY/FREQUENCY`).
2. Integrate **Customer Tenure** (`MAX(SALE_DATE) - MIN(SALE_DATE)`).
3. Schedule automated runs via **Snowflake Tasks & Streams**.
4. Extend pipeline for **real-time model scoring** via Snowpark ML.

■ 12. Appendix — Code & Cell Overview

Section	Notebook Cells	Description
----------	----------------	--------------
Setup	1–4	Environment and session setup
Ingestion	5–8	Stage creation, file format, COPY INTO, normalization
Transformation	14–18	DataFrame usage, lazy vs eager execution, joins
Feature Engineering	18–20	RFM computation and churn label creation
Validation	22–23	Explain plan and record count verification

■ Conclusion

This project successfully demonstrates how a **data engineer** can design and implement a **fully Snowflake-native preprocessing pipeline** using **Snowpark for Python**. It covers the complete flow — from ingestion of JSON data to analytical and ML-ready feature generation —

while adhering to SnowPro Advanced Data Engineer competencies.

End of Document.