- snowflake-ml-python version: 1.4.0
- Last updated on: 4/10/2024

DBT External Feature Pipeline Demo

This notebook showcases the interoperation between DBT and Snowflake Feature Store. The source data is managed in Snowflake database, while the feature pipelines are managed and executed from DBT. The output is stored as feature tables in Snowflake. Then We read from the feature tables and register as Feature View.

This demo requires DBT account.

Setup Snowflake connection

In [ ]:
```
from snowflake.snowpark import Session
from snowflake.ml.utils.connection_params import SnowflakeLoginOptions

session = Session.builder.configs(SnowflakeLoginOptions()).create()
```

Create test database, schema and warehouse.

In [ ]:
```
# database name where test data and feature store lives.
FS_DEMO_DB = f"SNOWML_FEATURE_STORE_DBT_DEMO"
# schema where test data lives.
TEST_DATASET_SCHEMA = 'DBT_DATA'
# feature store name.
FS_DEMO_SCHEMA = "FS_DBT_DEMO"

session.sql(f"DROP DATABASE IF EXISTS {FS_DEMO_DB}").collect()
session.sql(f"CREATE DATABASE IF NOT EXISTS {FS_DEMO_DB}").collect()
session.sql(f"""
    CREATE SCHEMA IF NOT EXISTS
    {FS_DEMO_DB}.{TEST_DATASET_SCHEMA}
""").collect()
```

Prepare source data

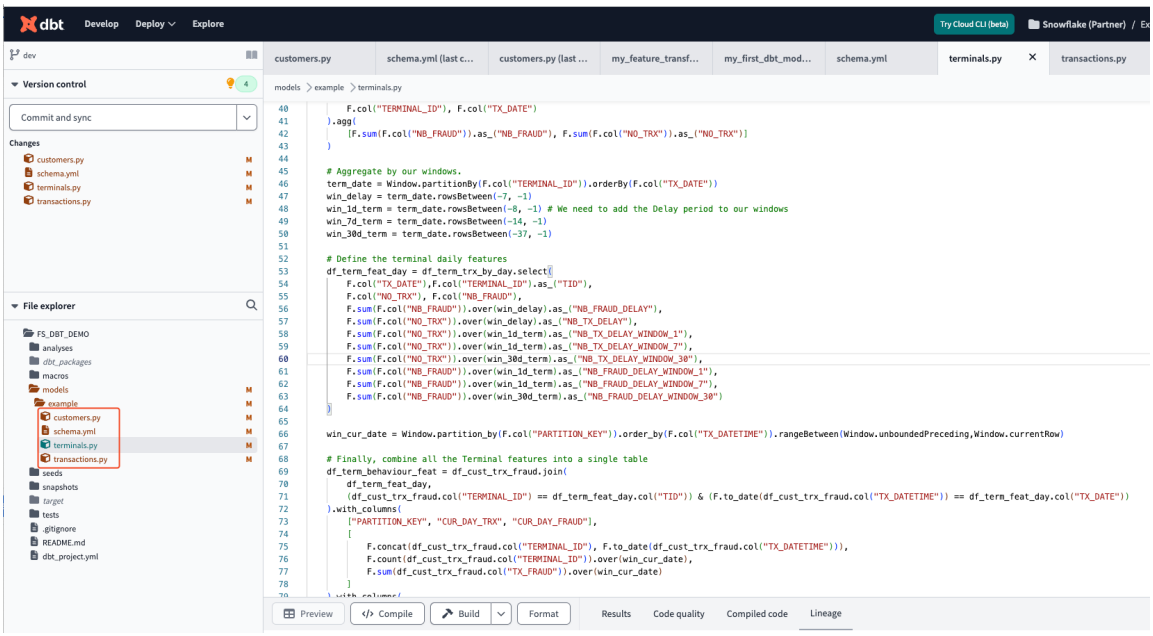
This notebook will use public `fraud_transactions` data as source. It contains transaction data range between [2019-04-01 00:00:00.000, 2019-09-01 00:00:00.000). We will split this dataset into two parts based on its timestamp. The first part includes rows before 2019-07-01, the second part includes rows after 2019-07-01. We copy the first part into `CUSTOMER_TRANSACTIONS_FRAUD` table now. And will copy second part into same table later.

In [ ]:
```
# Replace with your local path
LOCAL_PATH_CUSTOMER_TRANSACTIONS_FRAUD = "./fraud_transactions.csv.gz"

raw_data_path = f"{FS_DEMO_DB}.{TEST_DATASET_SCHEMA}.RAW_FRAUD_TRANSACTIONS"
session.sql(
    f"""create or replace TABLE {raw_data_path} (
        TRANSACTION_ID NUMBER,
        TX_DATETIME TIMESTAMP_NTZ,
        CUSTOMER_ID NUMBER,
        TERMINAL_ID NUMBER,
        TX_AMOUNT FLOAT,
        TX_TIME_SECONDS NUMBER,
        TX_TIME_DAYS NUMBER,
        TX_FRAUD NUMBER,
        TX_FRAUD_SCENARIO NUMBER)
    """).collect()

session.file.put(
    LOCAL_PATH_CUSTOMER_TRANSACTIONS_FRAUD,
    f"@{FS_DEMO_DB}.{TEST_DATASET_SCHEMA}.%RAW_FRAUD_TRANSACTIONS",
    auto_compress=False)
session.sql(f"""
    copy into {raw_data_path} file_format = (type = csv)""").collect()
session.sql(f"SELECT COUNT(*) FROM {raw_data_path}").show()
```

In [ ]:
```
fraud_data_path = f"{FS_DEMO_DB}.{TEST_DATASET_SCHEMA}.CUSTOMER_TRANSACTIONS
session.sql(f"""
    CREATE OR REPLACE TABLE {fraud_data_path} AS
    SELECT *
    FROM {FS_DEMO_DB}.{TEST_DATASET_SCHEMA}.RAW_FRAUD_TRANSACTIONS
    WHERE TX_DATETIME < '2019-07-01'
""").collect()
session.sql(f"SELECT COUNT(*) FROM {fraud_data_path}").show()
```

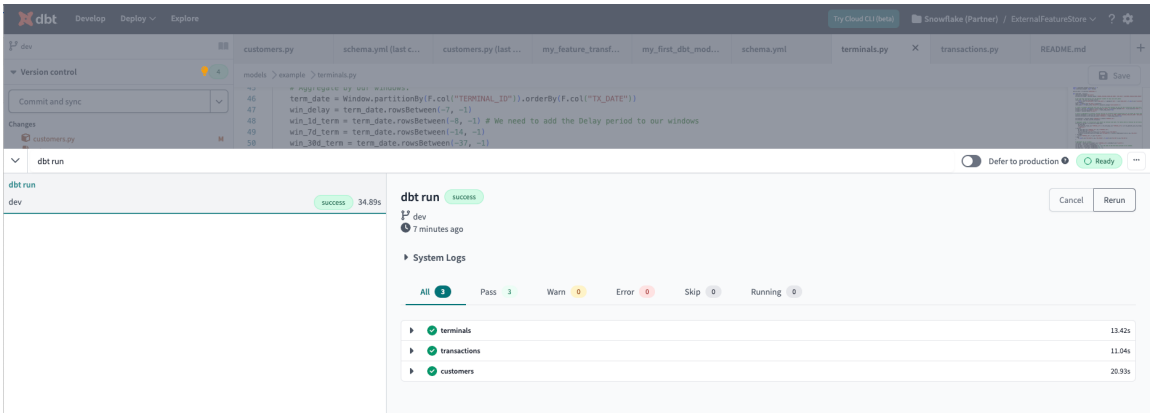
Define models in DBT

Now lets switch to DBT IDE(this link will not work for you, you will need to create your own project) for a while. You will need a DBT account beforehand. Once you have DBT account, then you can clone the demo code from here (Snowflake repo). Below screenshot shows how DBT IDE looks like. In the file explorer section, you can see the code structure. Our DBT models defined under models/example folder. We have 3 models: customers, terminals and transactions. These 3 models will later output 3 Snowflake DataFrame object. Lastly, Feature Store will register these DataFrames and make them FeatureViews.



Run models in DBT

After we defined models, now we can run and generate our feature tables. Simple exeucte `dbt run` in the terminal and it will do all the work.



After the run success, lets check whether the feature tables are populated.

(TODO, the output schema has a werid "FS_DBT_" prefix that comes from nowhere)

In [ ]:
```
# replace 'transactions' with 'customers' or 'terminals' to show respective
session.sql(f"SELECT * FROM {FS_DEMO_DB}.FS_DBT_{TEST_DATASET_SCHEMA}.transa
```

Register feature tables as Feature Views

Now lets create Feature Views with Feature Store. Since DBT is responsible for executing the pipeline, the feature tables will be registered as external pipeline. Underlying, it creates views, instead of dynamic tables, from the feature tables.

In [ ]:
```
from snowflake.ml.feature_store import (
    FeatureStore,
    FeatureView,
    Entity,
    CreationMode
)

fs = FeatureStore(
    session=session,
    database=FS_DEMO_DB,
    name=FS_DEMO_SCHEMA,
    default_warehouse='PUBLIC',
    creation_mode=CreationMode.CREATE_IF_NOT_EXIST,
)
```

Register entities for features.

In [ ]:
```
customer = Entity(name="CUSTOMER", join_keys=["CUSTOMER_ID"])
terminal = Entity(name="TERMINAL", join_keys=["TERMINAL_ID"])
transaction = Entity(name="TRANSACTION", join_keys=["TRANSACTION_ID"])
fs.register_entity(customer)
fs.register_entity(terminal)
fs.register_entity(transaction)
fs.list_entities().show()
```

Define feature views. `feature_df` is a dataframe object that selects from a subset of columns of feature tables. `refresh_freq` is None indicates it is static and won't be refreshed. Underlying it will create views on the feature tables.

In [ ]:
```
# terminal features
terminals_df = session.sql(f"""
    SELECT
        TERMINAL_ID,
        EVENT_TIMESTAMP,
        TERM_RISK_1,
        TERM_RISK_7,
        TERM_RISK_30
    FROM {FS_DEMO_DB}.FS_DBT_{TEST_DATASET_SCHEMA}.terminals
    """)
terminals_fv = FeatureView(
    name="terminal_features",
    entities=[terminal],
    feature_df=terminals_df,
    timestamp_col="EVENT_TIMESTAMP",
    refresh_freq=None,
    desc="A bunch of terminal related features")

# customer features
customers_df = session.sql(f"""
    SELECT
        CUSTOMER_ID,
        EVENT_TIMESTAMP,
        CUST_AVG_AMOUNT_1,
        CUST_AVG_AMOUNT_7,
```

```
        CUST_AVG_AMOUNT_30
    FROM {FS_DEMO_DB}.FS_DBT_{TEST_DATASET_SCHEMA}.customers
    """)
customers_fv = FeatureView(
    name="customers_features",
    entities=[customer],
    feature_df=customers_df,
    timestamp_col="EVENT_TIMESTAMP",
    refresh_freq=None,
    desc="A bunch of customer related features")

# transaction features
transactions_df = session.sql(f"""
    SELECT
        TRANSACTION_ID,
        EVENT_TIMESTAMP,
        TX_AMOUNT,
        TX_FRAUD
    FROM {FS_DEMO_DB}.FS_DBT_{TEST_DATASET_SCHEMA}.transactions
    """)
transactions_fv = FeatureView(
    name="transactions_features",
    entities=[transaction],
    feature_df=transactions_df,
    timestamp_col="EVENT_TIMESTAMP",
    refresh_freq=None,
    desc="A bunch of transaction related features")
```

Register these feature views in feature store so you can retrieve them back later even after notebook session is destroyed.

In [ ]:
```
terminals_fv = fs.register_feature_view(
    feature_view=terminals_fv,
    version="1",
    block=True)

customers_fv = fs.register_feature_view(
    feature_view=customers_fv,
    version="1",
    block=True)

transactions_fv = fs.register_feature_view(
    feature_view=transactions_fv,
    version="1",
    block=True)
```

Lets check whether feature views are reigstered successfully in feature store. You will see 3 registerd feature views and their status is "static".

In [ ]:
```
fs.list_feature_views().select([
    "NAME",
    "VERSION",
    "ENTITIES",
    "REFRESH_FREQ",
```

```
    "STATUS",
    "PHYSICAL_NAME"]).show()
```

Generate training dataset with point-in-time correctness

We can now generate training dataset with feature views. Firstly, we create a mock spine dataframe which has 3 columns: instance_id, customer_id and event_timestamp. Note the event_timestamp of 3 rows are same: "2019-09-01 00:00:00.000". Later, we will update the source table (`CUSTOMER_TRANSACTIONS_FRAUD`) and feature tables with newer events. We will still use this `spine_df` with same timestamp to generate dataset but it is expected to output a different training data. The new training data will join spine_df with latest feature values from newer events.

In [ ]:
```
spine_df = session.create_dataframe(
    [
        (1, 2443, "2019-09-01 00:00:00.000"),
        (2, 1889, "2019-09-01 00:00:00.000"),
        (3, 1309, "2019-09-01 00:00:00.000")
    ],
    schema=["INSTANCE_ID", "CUSTOMER_ID", "EVENT_TIMESTAMP"])

old_training_data = fs.generate_dataset(
    spine_df=spine_df,
    features=[customers_fv],
    materialized_table="customer_fraud_training_data",
    spine_timestamp_col="EVENT_TIMESTAMP",
    spine_label_cols = []
)
old_training_data.df.show()
```

Update features from DBT

Now we are injecting newer events into source, then refresh the pipeline and generate new feature values. We firstly check how many rows the source table currently has.

In [ ]:
```
session.sql(f"SELECT COUNT(*) FROM {fraud_data_path}").show()
```

We inject new events with timestamp later than '2019-07-01'. Then check how many rows in the source table after the injection.

In [ ]:
```
session.sql(f"""
    INSERT INTO {fraud_data_path}
    SELECT *
    FROM {raw_data_path}
    WHERE TX_DATETIME >= '2019-07-01'
""").collect()
session.sql(f"SELECT COUNT(*) FROM {fraud_data_path}").show()
```

Then, we go back to DBT and the pipelines again.

Generate new training dataset

We don't need to update feature views because the underlying tables are updated by DBT. We only need to generate dataset again with same timestamp and it will join with newer feature values.

In [ ]:
```
new_training_data = fs.generate_dataset(
    spine_df=spine_df,
    features=[customers_fv],
    materialized_table="customer_fraud_training_data",
    spine_timestamp_col="EVENT_TIMESTAMP",
    spine_label_cols = [],
    save_mode="merge",
)
new_training_data.df.show()
```

Cleanup notebook

In [ ]:
```
session.sql(f"DROP DATABASE IF EXISTS {FS_DEMO_DB}").collect()
```