Snowflake Feature Store Best Practices Guide

Author : Simon Field : Technical Director. SnowCat
simon.field@snowflake.com
linkedin.com/in/fieldy6961

Date : 2025/05/22
Version : v1

CONTENTS

INTRODUCTION

This document captures best-practices and guidance for the application of Snowflake Feature Store.

The information herein is provided as a guide for using the Snowflake Feature Store across the different aspects of feature management to supplement information provided in the product documentation. The topics covered reflect coomon questions from users implementing a Feature Store. We will continue to update this document with additional topics to cover questions as we receive them. The document currently relates to Snowflake-ml 1.8.4, and will be updated as new versions and changes in Feature Store and the database features it uses require it.

Topics

The document is organised into five key topic/areas.

1. Concepts

2. Design & Organisational Considerations
    a. Entity
        i. Management
        ii. Hierarchies
    b. Feature Life-Cycle
        i. Evolution & Version management

3. Feature Store Creation/Development
    a. Feature Engineering & Transformations
        i. Model-generic vs Model-specific
        ii. Data Transformation API
        iii. Snowflake ML Transformers
    b. ‘Backfill’

4. Feature Store Usage & Application
    a. Metadata
    b. Generating training/inference datasets
    c. Model-Registry integration and usage

5. Feature Store Operations, Management & Monitoring
    a. Monitoring **dynamic tables** / Tasks
    b. Monitoring Data Quality
    c. Cost monitoring/management

CONCEPTS

Key Objects and Components of Snowflake Feature Store

In Snowflake ***Feature Store*** we have the following object types and taxonomy for describing features and their relationships:

- A ***Feature Store*** is a container for a collection of features you want to store and manage together. You can have as many Feature Stores as needed, for example per business-unit, per environment etc. Within Snowflake a Feature Store is a `schema`, plus `object tags` where the database objects for that Feature Store are created.
- ***Entities*** - entities define the business-entity and hierarchies that we want to gather features and develop ML models for. (e.g. customer, store, product etc). They define the key column(s) that uniquely identify an entity (e.g. `customer_id`). Within Snowflake, ***Entities*** are represented by `object-tags` that are attached to the other database objects used within the ***Feature Store***
- Features are defined and grouped within a ***Feature View***. Within Snowflake a ***Feature View*** is created as either a `Dynamic Table` or `View.` Features are typically grouped within the ***Feature View*** based on physical and logical characteristics:
    - source(s) of data they are derived from
    - temporality (e.g. hourly, daily, time-intervals)
    - organisation (e.g. department, business unit)
    - application/use (e.g. per ML model).
- ***Feature Views*** are associated with one or more ***Entities***. A ***Feature View*** can be defined with 1:n ***Entities***, but typically one.
- ***Feature Views*** are comprised of
    - Entity key columns
    - 1:n Features
    - Timestamp (optional) - the point-in-time from which the feature-values apply, allowing a history of values to be maintained.
- Several (many) ***Feature Views*** will contain Features for the same ***Entity***.
- The ***Entity*** (key columns) are used to join ***Feature Views*** together to gather features from multiple ***Feature Views*** within a single training or inference dataset***.*** They are

joined by ***Entity*** to retrieve a combined set of features, using a temporal (`ASOF`) join, using an optional `timestamp` column. This ensures retrieved feature values reference the correct consistent point-in-time required for training, or inference.

- The entity-keys, timestamps we want to return in the training/inference dataset are provided via a ***Spine*** dataframe, which is typically materialised or sourced from table(s) in Snowflake. The ***Spine*** may also include additional columns required in the training dataset, e.g. label/target column for training.
- Pipelines of ***Feature Views*** are used to derive features from raw data sources through to model ready features forming feature-engineering pipelines managed within the ***Feature Store***.
- A ***Feature Slice*** provides a way of creating a transient subset of the ***Features*** from a single ***Feature View*** if needed. It can be used within the Feature Store API, mostly anywhere that a ***Feature View*** can be used. The ***Feature Slice*** is a python only transient object, and is not persisted within Snowflake database. If you need a permanent database object representing a subset of features you can instead create a ***Feature View*** derived from another ***Feature View***

Creating and ***registering Feature Store*** objects with the Python API, results in the creation of database objects within Snowflake that store and manage feature data. The documentation briefly describes here the mapping between ***Feature Store*** objects in Python and your database objects in your Snowflake account.

Whilst the ***Feature Store*** API aims to abstract the complexity of declaring, managing and combining data-sources as Features, through the python and Snowflake database objects, we can optimise storage and performance through understanding these relationships, and applying sound principles and best practices.

DESIGN & ORGANISATIONAL CONSIDERATIONS

You can create as many ***Feature Stores*** within an account as needed to organise/collate and manage your features as required. This provides flexibility in how users can structure and organise data for machine-learning. With this flexibility, comes responsibility!

One of the advantages of using a ***Feature Store*** is the centralised management and visibility of Features for re-use across many use-cases. Therefore some consideration of the organisation and structure of your Feature Stores will go a long way towards successful adoption and collaboration, whilst minimising fragmentation and undue process.

Caution:
The larger the number of ***Feature Stores***, the harder it will be to discover and manage features across the organisation.

Feature Store Organisation Structure

Examples of how multiple ***Feature Stores*** are used by organisations today:

Environments : Support different environments and model-life cycle into production :

Model-Development → Model-Test → Production

This typically aligns to the same structure used to manage development, test and production environments within an account. e.g. multiple databases.

Regional Federation : Organisations that need to separate data by geography for legal jurisdiction and compliance reasons. e.g. customer data in EMEA and US.

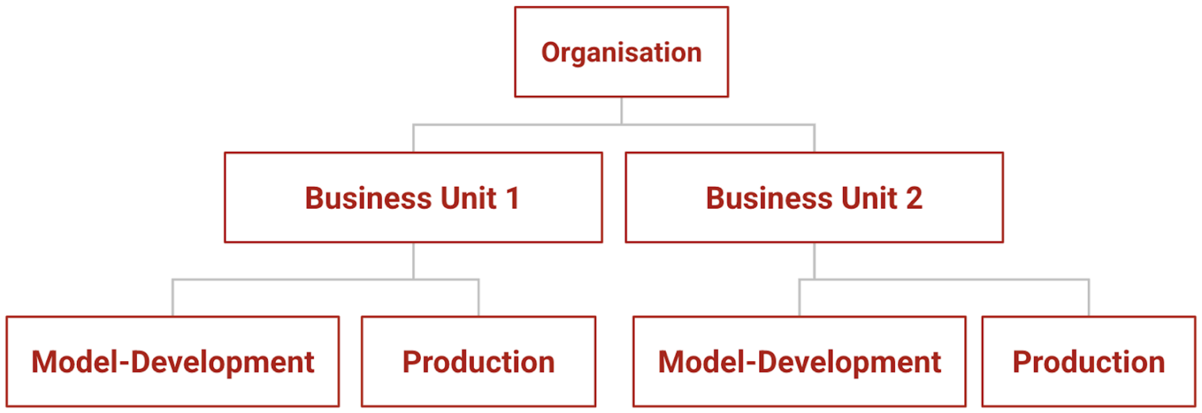
Business Group : Split by different business organisation/department, normally to align with Data Science teams that are also organised similarly

Access : segregating highly sensitive features into their own ***Feature Store***

Feature Store Hierarchy : Combinations of the above examples are also valid and seen regularly. For example an organisation might

- split by Business Group, and within that each group has multiple ***Feature Stores*** for environment life-cycle.
- Have a company store for global features applicable across all Business Units and Business Unit specific stores for each Business Unit

Below is an example of using multiple ***Feature Stores*** in a hierarchy:



Working with multiple Feature Stores

Snowflake Feature Store supports using and retrieving features from multiple ***Feature Store*** at a time. The retrieval functions (e.g. `generate_training_set`) can join ***Feature Views*** from across multiple ***Feature Stores***, as long as they share a common entity definition. The metadata stored for each ***Feature View***, fully-qualifies the database object name.

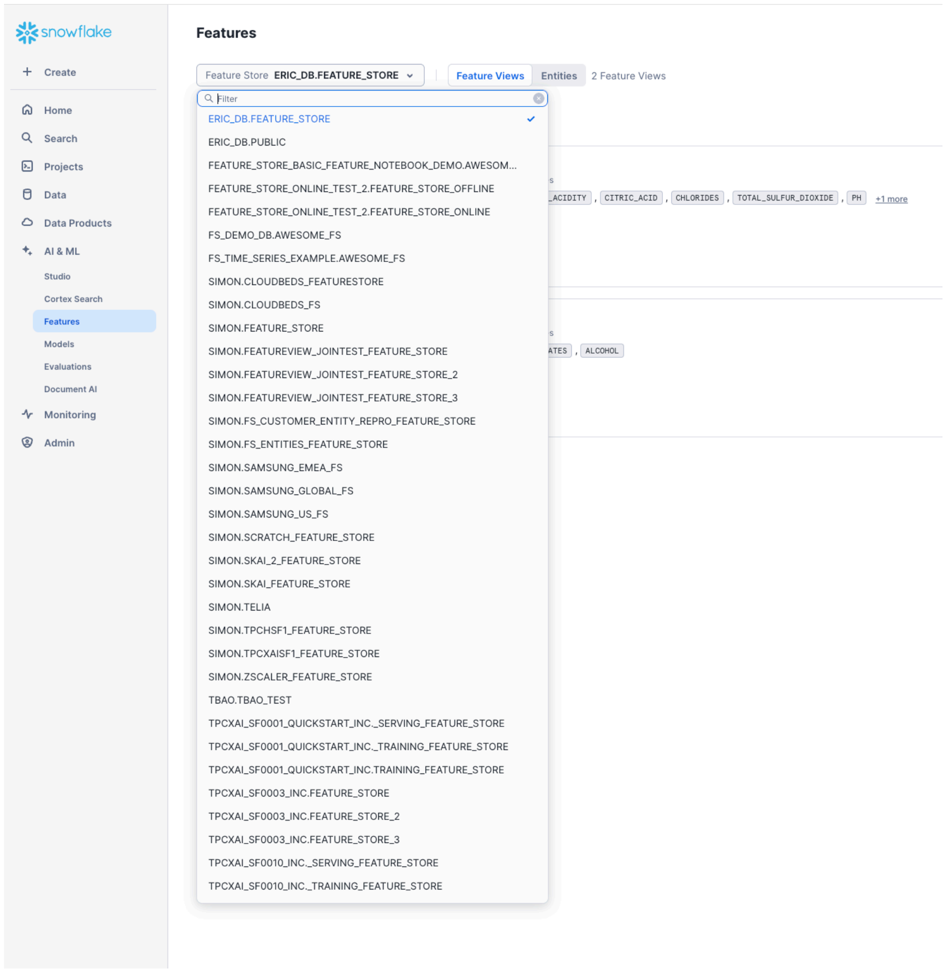
Tip:
Use the retrieval functions across more than one ***Feature Store*** to combine features that span multiple stores. To do so ensure that ***Entities*** are conformed and common across ***Feature Stores***.

The code sample below shows how we can combine the ***Feature Views*** from two ***Feature Stores*** to retrieve features that span across them. In this example, we might have a production model and versioned customer-spend features, and want to experiment with some new customer-spend features we have created in development to test if they add lift to an existing production model.

```python
# Business Unit 1 - Model Development Feature Store
bu1_dev_fs =  FeatureStore(
        session = session,
        database = 'bu1_dev_db',
        name = 'bu1_dev_fs',
        default_warehouse="dev_wh",
        creation_mode=CreationMode.CREATE_IF_NOT_EXIST,
)
# Retrieve the experimental feature view we have previously defined
cust_spend_fv_v2 = bu1_dev_fs.get_feature_view("CUSTOMER_SPEND_FV", "v2")
```

```
# Business Unit 1 - Production Feature Store
bu1_prod_fs =  FeatureStore(
        session = session,
        database = 'bu1_prod_db',
        name = 'bu1_prod_fs',
        default_warehouse="prod_wh",
        creation_mode=CreationMode.CREATE_IF_NOT_EXIST,
)
# Retrieve the production feature view we have previously defined
cust_spend_fv_v1 = bu1_prod_fs.get_feature_view("CUSTOMER_SPEND_FV", "v1")

# Generate training data that combines production and experimental
# customer-spend features
training_set = fs.generate_training_set(
     customer_spine_df,
     [cust_spend_fv_v1, cust_spend_fv_v2],
     save_as="experimental_training_data",
 )
```

You can use the Snowsight UI to easily select any of the ***Feature Stores*** in your account to discover and inspect the objects within.



ERIC_DB.FEATURE_STORE ✓
ERIC_DB.PUBLIC
FEATURE_STORE_BASIC_FEATURE_NOTEBOOK_DEMO.AWESOM...
FEATURE_STORE_ONLINE_TEST_2.FEATURE_STORE_OFFLINE
FEATURE_STORE_ONLINE_TEST_2.FEATURE_STORE_ONLINE
FS_DEMO_DB.AWESOME_FS
FS_TIME_SERIES_EXAMPLE.AWESOME_FS
SIMON.CLOUDBEDS_FEATURESTORE
SIMON.CLOUDBEDS_FS
SIMON.FEATURE_STORE
SIMON.FEATUREVIEW_JOINTEST_FEATURE_STORE
SIMON.FEATUREVIEW_JOINTEST_FEATURE_STORE_2
SIMON.FEATUREVIEW_JOINTEST_FEATURE_STORE_3
SIMON.FS_CUSTOMER_ENTITY_REPRO_FEATURE_STORE
SIMON.FS_ENTITIES_FEATURE_STORE
SIMON.SAMSUNG_EMEA_FS
SIMON.SAMSUNG_GLOBAL_FS
SIMON.SAMSUNG_US_FS
SIMON.SCRATCH_FEATURE_STORE
SIMON.SKAI_2_FEATURE_STORE
SIMON.SKAI_FEATURE_STORE
SIMON.TELIA
SIMON.TPCHSF1_FEATURE_STORE
SIMON.TPCXAISF1_FEATURE_STORE
SIMON.ZSCALER_FEATURE_STORE
TBAO.TBAO_TEST
TPCXAI_SF0001_QUICKSTART_INC._SERVING_FEATURE_STORE
TPCXAI_SF0001_QUICKSTART_INC._TRAINING_FEATURE_STORE
TPCXAI_SF0001_QUICKSTART_INC.TRAINING_FEATURE_STORE
TPCXAI_SF0003_INC.FEATURE_STORE
TPCXAI_SF0003_INC.FEATURE_STORE_2
TPCXAI_SF0003_INC.FEATURE_STORE_3
TPCXAI_SF0010_INC._SERVING_FEATURE_STORE
TPCXAI_SF0010_INC._TRAINING_FEATURE_STORE

The **`list_`** functions are currently scoped to a single ***Feature Store*** (schema).

Entities

This approach relies on consistent ***Entity*** definitions across ***Feature Stores***. Define your Business-Entity relational model, and ***entity*** hierarchy, and use the same model and definitions for ***Feature Store Entities*** across all the ***Feature Stores***. This ensures consistency and will enable ***Feature Views*** to be combined across ***Feature Stores***. Most organisations should already have a logical or business-entity data model in place that will be a good starting point for this.

For the code example above that illustrates retrieving features from two Feature Stores we would have defined a common customer ***Entity***, and then used **`register_entity`** to create that same ***Entity*** (*object-tag*) within each ***Feature Store*** (*schema*) :

```python
customer_ent = Entity(
     name="customer",
     join_keys=['customer_id'],
     desc='Customers of our organisation, uniquely identified via customer_id'
 )
# Create the entity in Business Unit 1 Development Feature Store
bu1_dev_fs.register_entity(customer_ent)
# Create the entity in Business Unit 1 Production Feature Store
bu1_prod_fs.register_entity(customer_ent)
```

Common Feature Views

We have seen how we can use multiple ***Feature Views*** from different ***Feature Stores***. There is another approach that can also be used to support this, adding a Feature View to a ***Feature Store*** that is a *pointer* to a ***Feature View*** in another. Lets illustrate that referencing the code in the example above, where we want to use the production customer-spend features ***Feature View*** directly within our development ***Feature Store***.

```python

# Define a dataframe using fully-qualified name of FeatureView we want to use
cust_spend_fv_v1_df =
  bu1_prod_fs.read_feature_view("CUSTOMER_SPEND_FV", "v1")

# Use the dataframe to create a feature view that is a database view
cust_spend_fv_v1 = FeatureView(
    name="CUSTOMER_SPEND_FV",
    entities=[customer_ent],
    feature_df=cust_spend_fv_v1_df,
    timestamp_col='TS',
    refresh_freq = None, # <- creates Feature-view as database view
    desc='View based feature view pointing to a production feature-view Dynamic Table',
)
# Creates the view in the development Feature Store
bu1_dev_fs.register_feature_view("CUSTOMER_SPEND_FV", "v1")
```

We:

- We use read_feature_view to *point* to a ***Feature View*** in another ***Feature Store*** which returns a dataframe.
- We use that dataframe to create a ***Feature View***, using **`refresh_freq = None`** (the default).
- register the ***Feature View*** to the ***Feature Store(s)*** that we want to use it in.

Within Snowflake this creates a ***view*** in our development ***Feature Store*** schema, pointing to the ***Feature View*** (***dynamic table*** or ***view***) in the production ***Feature Store***. In the case of a ***dynamic table*** it avoids materialising the data physically in both ***Feature Stores***, and duplicating the **dynamic table** feature computation twice, saving cost. We can now work with both ***Feature Views*** from within the development ***Feature Store***, and will also obtain lineage information back to the production ***Feature View*** when we generate training data and fit our models. The development ***Feature View*** is persisted, in the schema via the ***view***, so anyone working within the Development ***Feature Store*** will be able to see and use it without having to reference and work with ***Feature Views*** across multiple ***Feature Stores***.

If your organisation uses multiple Snowflake accounts, a similar technique can be used using Snowflake data-shares to share ***Feature Views*** across accounts, referencing them in local ***Feature Stores*** with a ***view*** based ***Feature View***.

In summary, the techniques we have shown here can be useful where you want to use multiple ***Feature Stores*** to separate and organise features by life-cycle, environment, organisation, data-governance etc. This enables collaboration and the benefits of a 'common' ***Feature Store*** to be realised, whilst maintaining any required separation of concerns, and avoiding duplicate compute and storage cost.

Managing Snowflake Feature Store(s); Environments & Releases

Typically organisations manage development using releases (versioning) and environments to promote code and its output. For example some feature-engineering code and the features that it generates are created or modified in Development (Dev), tested in Test, and released into Production (Prod). Often there are more environments, and distinctions between them to support business processes and control.

Feature Stores will typically require similar concepts and approaches to manage the code and features they control, together with their dependencies to other items; models in model-registry, upstream data-sources and their changes etc.

There are a number of different ways organisations might choose to setup their ***Feature Store***, typically strongly correlated to the way they manage other aspects of their Snowflake system.

Here are some examples to illustrate when managing within a single account via database/schema partitioning, naming convention, and privileges.

Single Account Approach Examples [**DATABASE**, **SCHEMA**]

1. Same database per FeatureStore, separate schema per environment e.g.;
   - `PROCUREMENT_FEATURE_STORE`
     - `.DEV`
     - `.TEST`
     - `.PROD`
   - `MARKETING_FEATURE_STORE`
     - `.DEV`
     - Etc....
2. Different database per environment, schema per BU; e.g.
   - `DEV`
     - `.PROCUREMENT_FEATURE_STORE`
     - `.MARKETING_FEATURE_STORE`
   - `TEST`
     - `.PROCUREMENT_FEATURE_STORE`
     - Etc....
3. Different database per [Business group]_[Environment], same schema; e.g.
   - `PROCUREMENT_DEV`
     - `.FEATURE_STORE`
   - `MARKETING_DEV`
     - `.FEATURE_STORE`

- `PROCUREMENT_TEST`
  - `.FEATURE_STORE`
- `MARKETING_TEST`
  - `.FEATURE_STORE`

Multiple Account Approach Examples [ACCOUNT, DATABASE, SCHEMA]

1. Account per Environment. Database to store all ***Feature Stores***. Schemas per business-unit
   - `DEV ACCOUNT : account1.snowflakecomputing.com`
     - `FEATURE_STORE`
       - `.MARKETING`
       - `.PROCUREMENT`
   - `TEST ACCOUNT : account2.snowflakecomputing.com`
     - `FEATURE_STORE`
       - `.MARKETING`
       - etc....
2. Account per Environment. Database for business-unit, Schema for FeatureStore.
   - `DEV ACCOUNT : account1.snowflakecomputing.com`
     - `MARKETING`
       - `.FEATURE_STORE`
     - `PROCUREMENT`
       - `.FEATURE_STORE`
   - `TEST ACCOUNT : account2.snowflakecomputing.com`
     - `MARKETING`
       - `.FEATURE_STORE`
     - `PROCUREMENT`
       - `.FEATURE_STORE`
       - etc

Cross Environment Feature promotion (e.g. Dev → Test → Prod)

Maintaining the feature(s) across multiple environments. In the majority of cases **schemas** (object-definitions) will be aligned across each environment. Therefore 'promoting' a ***Feature View*** between environments is as trivial as re-creating the ***Feature View*** in the new environment with the appropriate namespace.

For example using the following code example we can define the namespace for our ***Feature View*** source tables based on an environment variable (or similar mechanism) that assigns the value in the DevOps process, and avoid hard-coding it within the definition of the objects.

```python
# Database
fs_database            = 'TPCXAI_SF0001'


# Get the Schema for a specific (Environment)
fs_schema = os.environ['ENVIRONMENT']


# Source Data to Snowflake Dataframe Objects
# Snowpark Dataframe
line_item_sdf      = session.table([fs_database, fs_schema, LINEITEM])
order_sdf          = session.table([fs_database, fs_schema, ORDERS])
order_returns_sdf  = session.table([fs_database, fs_schema, ORDER_RETURNS])


fs = FeatureStore(session, fs_database,
                           f'{fs_schema}_feature_store',
                           fs_warehouse, CreationMode.FAIL_IF_NOT_EXIST)
```

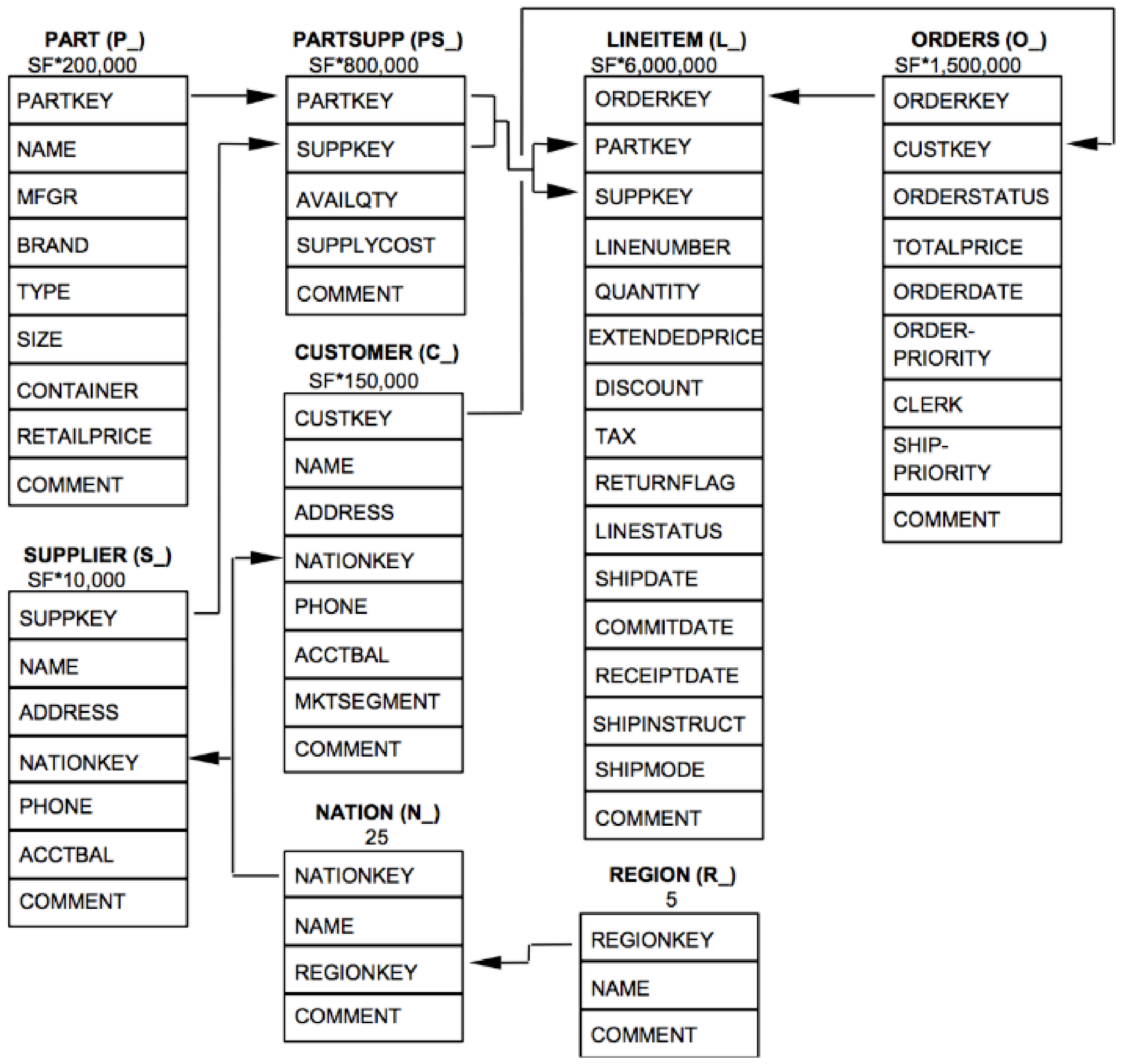
When we now define feature-engineering dataframes, ***Feature Views*** etc, we reference the appropriate environment for the objects ensuring that our code is portable across environments without hard-coded references.

Entities and Entity Hierarchies : the relational model

Entities are the glue that enable us to work and combine features across ***Feature Views*** and ***Feature Stores***. Entities and their relationships will most likely strongly conform to an organisations logical or business-entity data model and therefore this will be a good starting point for determining the entities in your ***Feature Store(s)***.

To provide an example of working and combining multiple entities we will use the TPC-H data model and the database that is provided in the snowflake_sample_data as it is readily available in most snowflake accounts.

Figure 2: The TPC-H Schema



This section highlights the key best practices for working with entity hierarchies. For more in-depth study of this you can import this Snowflake Notebook to your account to walk through detailed examples. The first few cells contain parameters related to the environment that you want to run the notebook in. e.g. notebook schema, name, warehouse etc. Change these as required and the Notebook is then intended to run as-is using the sample data. You will also need to select **snowflake-ml** package from the Notebook package selector.

As a feature store matures and grows, we will most likely:
- have multiple ***Entities*** that have relationships with each other
- have many ***Feature Views*** for the same ***Entity***
- want to combine features from multiple entities to generate training/inference data

The Entities in most organizations are likely to be well defined and relatively static, but the relations between them may change through time. For example:
- the department or group that a product is managed under may change following a product or business reorganisation
- the suppliers that we source a product from may change over time.

You may want to use the current (as-is) relations, or those from an earlier point-in-time, when retrieving features.

Feature Views can have multiple ***Entities*** assigned to them via the **`entities`** list argument. However, we recommend you only assign the entities required to form the primary-key that uniquely identify their features. Additional ***Entities*** (e.g. foreign-keys) can and should be defined within your ***Spine*** at time of retrieval, and relate to the relationships you want to infer for that point-in-time for your training data.

Defining non primary key relationships in Entities fixes the relationship to that in the source data defined in the ***Feature View dataframe***. If those relationships change within the source data, the data in ***Feature View's*** will reflect that. In the case of **dynamic table** based ***Feature Views***, any updates to those relationships (primary - foreign key) in source data would be applied via the next refresh of the **dynamic table**. In the case of view based ***Feature Views***, the relationships will be inferred at the time of query (feature retrieval).

Illustrating possible approaches by example from the TPC-H data model:

Customer

Creating a customer entity, the primary-key (`join_keys`) for the entity should clearly be `CUSTKEY.` The `NATIONKEY` is a foreign-key in the `CUSTOMER` table, giving us some design choices if we know that a customer could relocate to another Nation in time. The table does not currently contain a timestamp (e.g. `VALID_FROM_TS`) that would support historisation of value changes to customer features, including the `NATIONKEY` foreign-key, over time. Therefore with the current data-source, we can only reflect the values as of the last source data changes in the `CUSTOMER` table. We might consider adding this functionality to the source data, maybe creating a new **`CUSTOMER_HISTORY`** table that maintains this. Alternatively, some new features in Dynamic Tables (immutability constraints, insert-only) can be used to maintain this historisation

Given the following table and ***Entity*** definitions:

```python
# CUSTOMER SOURCE TABLE
customer_table = 'CUSTOMER'
customer_sdf = (session.table(f'{src_database}.{src_schema}.{customer_table}')
                 .with_columns(['CUSTKEY','NATIONKEY'],
                               [F.col('C_CUSTKEY'), F.col('C_NATIONKEY')])
                 .drop('C_CUSTKEY', 'C_NATIONKEY') )
customer_nosk_sdf = customer_sdf.drop('NATIONKEY')


# NATION SOURCE TABLE
nation_table = 'NATION'
nation_sdf = session.table(f'{src_database}.{src_schema}.{nation_table}') \

.with_columns(['NATIONKEY','REGIONKEY'],[F.col('N_NATIONKEY'),F.col('N_REGIONKEY')]) \
                        .drop('N_NATIONKEY', 'N_REGIONKEY')
nation_nosk_sdf = nation_sdf.drop('REGIONKEY')


# Customer Entity Definition
customer_entity = Entity(
    name="CUSTOMER",
    join_keys=["CUSTKEY"],
    desc="Customer entity")
fs.register_entity(customer_entity)


# Nation Entity Definition
nation_entity = Entity(
    name="NATION",
    join_keys=["NATIONKEY"],
    desc="Nation entity"
    )
fs.register_entity(nation_entity)
```

Some options;

1. Exclude **NATIONKEY** in the ***Feature View dataframe***. Include the **NATIONKEY** to the ***Spine dataframe***, where the spine may be derived from other historical sources that maintain the temporal relationship between **CUSTOMER** & **NATION**.

```python
# CUSTOMER FEATUREVIEW

# Create Customer FeatureView in Feature Store
customer_fv = FeatureView(
    name = f"FV_CUSTOMER",
    entities = [customer_entity],
    feature_df = customer_nosk_sdf, # Using dataframe that excludes NATIONKEY
    desc = f"Customer feature view"
)
# Register the Customer FeatureView in the schema, and add the python
# featureview to our list
customer_fv_v1 = fs.register_feature_view(
    feature_view = customer_fv,
    version = "V1",
)

# Persist a spine table & include the NATIONKEY within the SPINE
customer_nation_spine_tbl = [sess_db,org,'CUSTOMER_SPINE']

customer_sdf.select('CUSTKEY','NATIONKEY') \
    .distinct() \
    .sample(n=num_spine_rows) \
    .write.saveAsTable(customer_nation_spine_tbl,
                       mode = 'overwrite', table_type = 'temp')

customer_nation_sdf = session.table(customer_nation_spine_tbl)

# Generate training data
customer_training_df = fs.generate_training_set( customer_nation_sdf,
    features = [customer_fv_v1] )
```

The **NATIONKEY** column derived in the ***Spine*** will be attached to the returned training dataset joined on **CUSTKEY**.

2. Include **NATIONKEY** in the ***Feature View dataframe***, which will in effect only ever represent the current-state value maintained in the ***Feature View***.

```python
# CUSTOMER FEATUREVIEW

# Create Customer FeatureView in Feature Store
customer_fv = FeatureView(
    name = f"FV_CUSTOMER",
    entities = [customer_entity],
    feature_df = customer_sdf, # Using dataframe that includes NATIONKEY
    desc = f"Customer feature view"
)
# Register the Customer FeatureView in the schema, and add the python
# featureview to our list
customer_fv_v1 = fs.register_feature_view(
    feature_view = customer_fv,
    version = "V1",
)

# Persist a spine table & include the NATIONKEY within the SPINE
customer_spine_tbl = [sess_db,org,'CUSTOMER_SPINE']

customer_sdf.select('CUSTKEY') \
    .distinct() \
    .sample(n=num_spine_rows) \
    .write.saveAsTable(customer_nation_spine_tbl,
                       mode = 'overwrite', table_type = 'temp')

customer_sdf = session.table(customer_spine_tbl)

# Generate training data
customer_training_df = fs.generate_training_set( customer_sdf,
    features = [customer_fv_v1] )
```

3. Include the **`NATIONKEY`** in the ***Feature View dataframe***, providing the current-state value as maintained in the ***Feature View***. Provide any alternative **`NATIONKEY`** values, derived in the ***Spine***.

```python
customer_nation_spine_tbl = [sess_db,org,'CUSTOMER_SPINE']

( customer_sdf.select(F.col('CUSTKEY'),
                      F.col('NATIONKEY').as_("ALTERNATIVE_NATIONKEY"))
              .distinct().sample(n=num_spine_rows)
              .write.saveAsTable(customer_nation_spine_tbl,
                                 mode = 'overwrite', table_type = 'temp')
)

customer_nation_sdf = session.table(customer_nation_spine_tbl)
customer_nation_sdf.sort('CUSTKEY').show()

# Generate training data
customer_training_df = fs.generate_training_set(
    customer_nation_sdf,
    features = [customer_nation_fv_v1]
)
```

```
customer_training_df.sort('CUSTKEY').show()
```

Part & Supplier Entities

We have a few tables with the **PARTKEY** and **SUPPKEY** keys in them : **PART**, **SUPPLIER**, **PARTSUPP** and **LINEITEM**. We will clearly need **PART** and **SUPPLIER** entities, and optionally a combined **PARTSUPP** entity.

It is straightforward to use these in **PART** and **SUPPLIER** tables for their respective primary keys.

PARTKEY and **SUPPKEY** are foreign-keys in the **LINEITEM** table so we can choose to treat them similarly to the options outlined previously for the **CUSTOMER** table **NATIONKEY** column.

The **PARTSUPP** table uses **PARTKEY** and **SUPPKEY** as a compound Primary Key.

Python
```python
# Part Entity Definition
part_entity = Entity(
    name="PART",
    join_keys=["PARTKEY"],
    desc="Part entity")

fs.register_entity(part_entity)

# Supplier Entity Definition
supplier_entity = Entity(
    name="SUPPLIER",
    join_keys=["SUPPKEY"],
    desc="Supplier entity")

fs.register_entity(supplier_entity)

# PART SUPPLIER FEATUREVIEW

# Create Part Supplier FeatureView in Feature Store
part_supplier_fv = FeatureView(
    name = f"FV_PART_SUPPLIER",
    entities = [part_entity, supplier_entity],
    feature_df = partsupp_sdf,
    desc = f"Part Supplier feature view")

# Register the Part Supplier FeatureView in the schema
part_supplier_fv_v1 = fs.register_feature_view(
    feature_view = part_supplier_fv,
    version = "V1")
```

Alternatively, we could create a combined `part_supplier_entity`, and use that instead of, or in addition to, the part and supplier entities.

```python
# Part Supplier Entity Definition
part_supplier_entity = Entity(
    name="PART_SUPPLIER",
    join_keys=["PARTKEY", "SUPPKEY"],
    desc="Part Supplier entity"
    )
fs.register_entity(part_supplier_entity)

# PART SUPPLIER FEATUREVIEW

# Create Part Supplier FeatureView in Feature Store
part_supplier_fv = FeatureView(
    name = f"FV_PART_SUPPLIER",
    entities = [part_entity, supplier_entity, part_supplier_entity],
#   entities = [part_supplier_entity],
    feature_df = partsupp_sdf,
    desc = f"Part Supplier feature view")

# Register the Part Supplier FeatureView in the schema
part_supplier_fv_v1 = fs.register_feature_view(
    feature_view = part_supplier_fv,
    version = "V1")
```

Including `part_supplier_entity` in the ***Feature View*** definition, effectively enforces and requires any spine used for data-retrieval to include both those key columns within the spine. This may, or may not be a desirable behaviour. Supplying only a **PARTKEY** or a **SUPPKEY** would likely return additional records i.e. if we have multiple suppliers for a part, or vice-versa, and we may want to prevent users from inadvertently returning additional training records in this case. Likewise we might want to return all the training records for a given part, regardless of supplier, without having to include every part/supplier combination within our **Spine**.

FEATURE STORE CREATION & DEVELOPMENT

Feature Life-Cycle Evolution

Feature View & Feature Versioning

How do we handle schema evolution for features? If for example a feature definition changes, do we backfill/update the previous instances of the feature or just the latest one?

Here we are referring only to changes/evolution of the features definition, as opposed to evolution of the underlying schema that the features are sourced from. There are several ways this can be managed within a ***Feature Store***. The most appropriate will depend on specific requirements and preferences for the way of working, but here we outline some of the principles.

In ***Feature Store*** we really version a ***Feature View*** rather than the features within. The features are implicitly versioned as a bundle within. A ***Feature View*** at a given version (when registered) should really be considered an immutable object. When we register (**`register_feature_view`**) we provide a new **`version`** of the ***Feature View*,** typically by incrementing a version-numbering scheme of our own choosing. This will create a new physical database object (**dynamic table** or **view**) in the Feature Store schema using the dataframe/SQL used in its definition. In the case of a **dynamic table**, it creates a completely new **dynamic table** re-computing the backfill and updates for all the features as defined in the ***Feature Views*** dataframe. This includes any prior features you carry over from the prior ***Feature View*** dataframe version(s) into the definition of the new. i.e. under the covers we don't do any differencing between versions, and only create the new or changed features in the new ***Feature View*** version.

Tip:
You cannot have duplicate column (feature) names within a ***Feature View***. Similarly to the explicit version numbering of a ***Feature View***, you can also create a versioning scheme/approach for features. This can then be incremented if modifying the definition/derivation of features. e.g.

```
f_customer_30day_visits$v01
f_customer_30day_visits$v02
```

It is also good practice to avoid duplicate feature names within a Feature Store. Thie can be avoided by using a abbreviation for the ***Feature View*** as a prefix/suffix in each ***Feature View***

Reducing feature duplication/computation.
How to avoid storing multiple copies of the same feature across different ***Feature View*** versions requiring additional computation and storage? Instead of creating a new version of the ***Feature View*** with both the prior and new features in it, we alternatively create a new supplemental ***Feature View***, for the same ***Entity***, but only containing the modified or new Features within it. In this case the new ***Feature View*** object, when registered, only needs to backfill/update and store the new features values. Now we have two ***Feature Views***, rather than one, and our features are split across these with potentially different definitions across both, for features with the same name, at different version levels. What are our options for combining them when needed.

Option 1 - Combine at point of use
We can easily find all the features in the ***Feature Store*** for a particular ***Entity*** using `list_feature_views`. We can then select all, or subsets, of them for use in our model and combine them at the point of use when they are needed. E.g. for training or inference.

Tip:
We recommend considering a naming convention and a standard agreed taxonomy for your ***Feature Views*** that enables users to search easily and identify key characteristics, by name.

Option 2 - Combine into a master Entity-level Feature View (or new Feature View version)
Should we want to gather and see all the features for the same entity combined (logically) as one within the ***Feature Store***, we can :
Either
- Create a dataframe that joins the ***Feature Views*** and use it to create a new combined master ***Feature View.***

Or
- Create a new version of the original ***Feature View***, to ‘merge’ back the ‘branch’, containing the new/revised features. I.e. increment the ***Feature View*** version.

If these are created as a **view** based ***Feature Views***, these joins occur on-the-fly at retrieval time, and avoid needing to compute and store the data in a ***dynamic table***.

Lifting the Hood - A bit more detail for those interested.

Providing a bit of background and explanation on internals of the ***Snowflake Feature Store***, which hopefully the majority of end-users don't need to concern themselves with too much.

A ***Feature View*** defines a collection of features, associated to a given entity.

```
Python
entity = Entity(name="WINE", join_keys=["WINE_ID"])
fs.register_entity(entity)
```

The features in ***Feature Views*** are described via a ***Snowpark Dataframe***, (which could also be defined using SQL via `session.sql()`).

```
Python
fv = Feature View(
    name="WINE_FEATURES",
    entities=[entity],
    feature_df=feature_df,
    refresh_freq="1 minute", # creates a Dynamic Table
    desc="wine features"
)
```

Within Snowflake, this gets materialised as a **dynamic table** when the ***Feature View*** is registered with `register_feature_view`, and when a `refresh_freq` is specified in the ***Feature View*** definition. Assuming Snowflake is able to, the ***Feature View*** (**dynamic table**) will be initially populated on creation, based on the definition and then incrementally populated with changes subsequently. Snowflake **dynamic table** restrictions apply.

If `refresh_freq` is not provided in the ***Feature View*** definition, it is created as a **view**, and so will be fully computed on-the-fly when those features/values are requested. Depending on the ***Feature View*** definition, and the information scope being requested when queried, this might be more or less computationally intensive, than pre-computing the features in a **dynamic table**.

```
Python
fv_v1 = fs.register_feature_view(
    feature_view=fv,
    version="V1",
    block=True
)
```

Note the `version` above. The corresponding **dynamic table** (or **view**) name created will be `WINE_FEATURES$V1`. As a given ***Feature View***/Version is immutable you can't create another ***Feature View*** with the same name and version as an existing, unless you drop it first (`delete_feature_view()`). Or, you can specify `overwrite=True` when you `register_feature_view()`.

Tip: *`overwrite`* is provided in the API primarily to support externally managed feature tables, that have a full refresh/replacement strategy. For example in a DBT pipeline where full-refresh is being performed. Or within an ETL process that does a full table rebuild via CTAS. It is not intended for users to use this to manage versioning of Feature Store managed ***Feature Views***.

Option 1 - Update 'MAIN'
Create new versions of the same ***Feature View*** (e.g. `"WINE_FEATURES$V2"`) when you need new versions of the features within. The definition of the **`feature_df`** is edited to include the new features, and any revisions to the derivation of the existing features. We then `register` the new ***Feature View*** version. This creates a new **dynamic table** (`WINE_FEATURES$V2)` which will require duplicate initial and ongoing incremental computation for any features that have not changed between `"WINE_FEATURES$V1"` and `"WINE_FEATURES$V2"`. They are two distinct and independently executing **dynamic table**. In addition, any changes to the underlying source-data will result in changes passed to both the **dynamic table** and its applicable computation.

Models can be trained on data from a specific version(s) of the ***Feature View***, and this data lineage is captured in the ***Snowflake Model Registry***, when the model is registered.

When a version of the ***Feature View*** is no longer required (e.g. because it is not actively being used in any models, or a new version has superseded it and is now being used across all models, it can be suspended (**`suspend_feature_view`**) which will PAUSE the **dynamic table**. The ***Feature View*** version will continue to exist and take up storage, but no longer be maintained, refreshed from source data changes. Thus the definition and historical data is available should it be needed. When no longer needed it can be dropped with **`delete_feature_view()`** which will avoid table storage cost.

Option 2 - Branch (and optionally Merge)
Think of this a little like version management in GIT!.

Branch
We copy the existing ***Feature View*** and dataframe definition and modify it to only include the new/revised features that we want. We create a new ***Feature View*** (e.g. `EXTRA_WINE_FEATURES$V1`) with only the new (redefined) features in it using the same ***Entity*** as the original. Assuming a **`refresh_freq`** is specified a new **dynamic table** will be created. Only the new and revised features will be computed; via initial and then ongoing incremental computation.

Maybe these new features are experimental, at least for a while, and we want to use and keep them in the ***'branch'*** until we are sure of their long-term value. We can combine the **Feature**

Views at the point of use when the data is retrieved. E.g. for model-fitting or inference using either `generate_dataset` or `retrieve_feature_values` which both support a list of ***Feature Views***, provided as arguments.

Merge

We have now decided the new or revised features, should supercede the prior versions in use.

We create a new dataframe that joins the two ***Feature Views*** together, and use this **Feature View** defined via a new dataframe that joins the two ***Feature Views*** into one. In most cases this should be very straightforward join

```
Python
fv_v2 = fv_v1.join(fv_extra_v1, fv_v1.WINE_ID == fv_extra_v1.WINE_ID )
```

Note: the join above uses an INNER_JOIN between the two underlying tables so rows from either side may get 'dropped' from the join if there are entity-key/timestamp differences between them due to significant differences in the definitions of the dataframes. You will possibly want to use a left, or full outer join, or alternative techniques to avoid this on the entity key. If the ***Feature Views*** contain a timestamp column you will also need to include that in the join columns, and may need to consider the right join approach if the timestamp columns are not exactly aligned. E.g. use an ASOF join to match rows.

The new ***Feature View*** version can be registered as either a **view** or **dynamic table** depending on whether `refresh_freq` is provided. It is recommended that a **view** is used in this instance, as in effect we are just performing a join of two existing ***feature views***, where no additional feature-derivation computation is needed. Within Snowflake this creates a SQL view that joins the two together.

If we reuse the original ***Feature View*** name and increment the version, we effectively create a new version of that ***Feature View*** with the combined set of features from the prior version and the branch, akin to a MERGED PULL request.

```
Python
fv = Feature View(
    name="WINE_FEATURES",
    entities=[entity],
    feature_df=fv_v2,
    desc="wine features"
)

fv_v2 = fs.register_feature_view(
    feature_view=fv,
    version="V2",
    block=True
)
```

We will also be able to keep track of the mapping between the underlying **dynamic table** (`WINE_FEATURES$V1, EXTRA_WINE_FEATURES$V1`) and the new view with the built-in lineage supported within Feature Store.

Source Object life-cycle impact on Feature Views

This section covers the impact of changes to underlying feature-store source objects (i.e. **tables**, **views**, …) used as a source for ***Feature Views*** (**dynamic tables** & **views**) and outlines options for handling and working around pottential issues. It’s possible that a source object change might break the and predevent the **dynamic tables** from refreshing. These breaking changes require the **dynamic tables** to be recreated, requireing full refresh, which could be pottentially expensive. If there are downstream ***Feature Views*** dependancies, these will also require refreshing.

Using this single source table.

```SQL
create or replace TABLE FS_SOURCE_TABLE (
        TID VARCHAR(32),
        EID VARCHAR(32),
        FEAT_VECT VARIANT, -- contains a number of features embedded within the field
        CTS TIMESTAMP_LTZ(0)
);
```

Dynamic Table based Feature Views

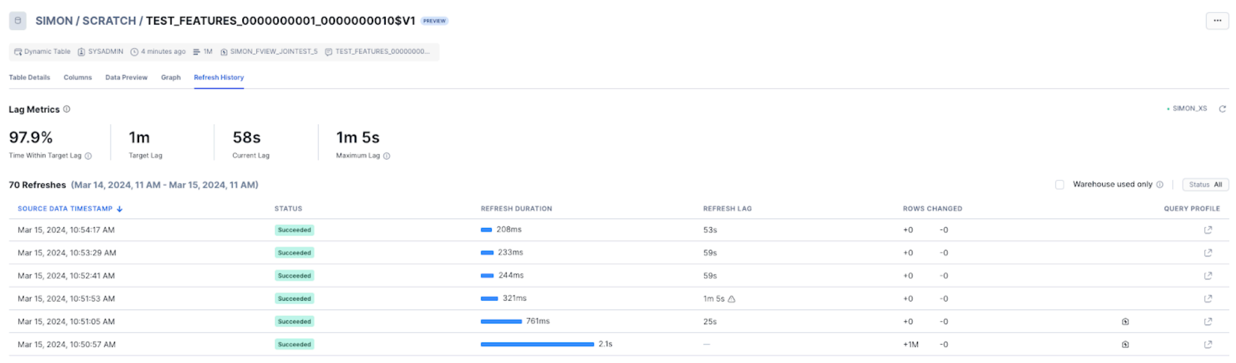
We have the following ***Feature View*** defined over the source table

```Python
Feature_df = session.sql(‘’’
  SELECT "TID", "EID", "CTS",
   "FEAT_VECT":F00001::FLOAT F00001,   "FEAT_VECT":F00002::FLOAT F00002,
   "FEAT_VECT":F00003::FLOAT F00003,   "FEAT_VECT":F00004::FLOAT F00004,
   "FEAT_VECT":F00005::FLOAT F00005,   "FEAT_VECT":F00006::FLOAT F00006,
   "FEAT_VECT":F00007::FLOAT F00007,   "FEAT_VECT":F00008::FLOAT F00008,
   "FEAT_VECT":F00009::FLOAT F00009,   "FEAT_VECT":F00010::FLOAT F00010,
   FROM FS_SOURCE_TABLE;
‘’’)

managed_fv = FeatureView(
    name="TEST_FEATURES_0000000001_0000000010",
    entities=[entity],
    feature_df=feature_df,
    timestamp_col="CTS",
    refresh_freq="1 minutes",
    desc="A subset of 10 features (1-10) selected from the 1000 feature source"
)

registered_fv_0: FeatureView = fs.register_feature_view(
    feature_view=managed_fv,
    version="1",
    block=False,
    override=False,
)
```

Note: the features in our **FS_SOURCE_TABLE** are stored within a **VARIANT** column, which we are extracting from using **:** syntax and casting using **::** E.g **"FEAT_VECT":F00001::FLOAT F00001**. Which results in the creation of the following **dynamic table**

```
SQL
create or replace dynamic table TEST_FEATURES_0000000001_0000000010$V1(
        TID,
        EID,
        CTS,
        F00001,  F00002,  F00003,  F00004,  F00005,  F00006,  F00007,  F00008,  F00009,  F00010
) lag = '1 minute' refresh_mode = AUTO initialize = ON_CREATE warehouse = SIMON_FVIEW_JOINTEST_5
 comment='A subset of 10 features (1-10) selected from the 1000 feature source'
 as
 select "TID", "EID", "CTS",
 "FEAT_VECT":F00001::FLOAT F00001,   "FEAT_VECT":F00002::FLOAT F00002,
 "FEAT_VECT":F00003::FLOAT F00003,   "FEAT_VECT":F00004::FLOAT F00004,
 "FEAT_VECT":F00005::FLOAT F00005,   "FEAT_VECT":F00006::FLOAT F00006,
 "FEAT_VECT":F00007::FLOAT F00007,   "FEAT_VECT":F00008::FLOAT F00008,
 "FEAT_VECT":F00009::FLOAT F00009,   "FEAT_VECT":F00010::FLOAT F00010,
 from FS_SOURCE_TABLE;
```

Once created, we leave it running for a few refresh-cycles to reach a steady state.


SIMON / SCRATCH / TEST_FEATURES_0000000001_0000000010$V1 PREVIEW

Dynamic Table · SYSADMIN · 4 minutes ago · 1M · SIMON_FVIEW_JOINTEST_5 · TEST_FEATURES_00000000...

Lag Metrics

97.9% Time Within Target Lag · 1m Target Lag · 58s Current Lag · 1m 5s Maximum Lag

70 Refreshes (Mar 14, 2024, 11 AM - Mar 15, 2024, 11 AM)

Warehouse used only · Status All

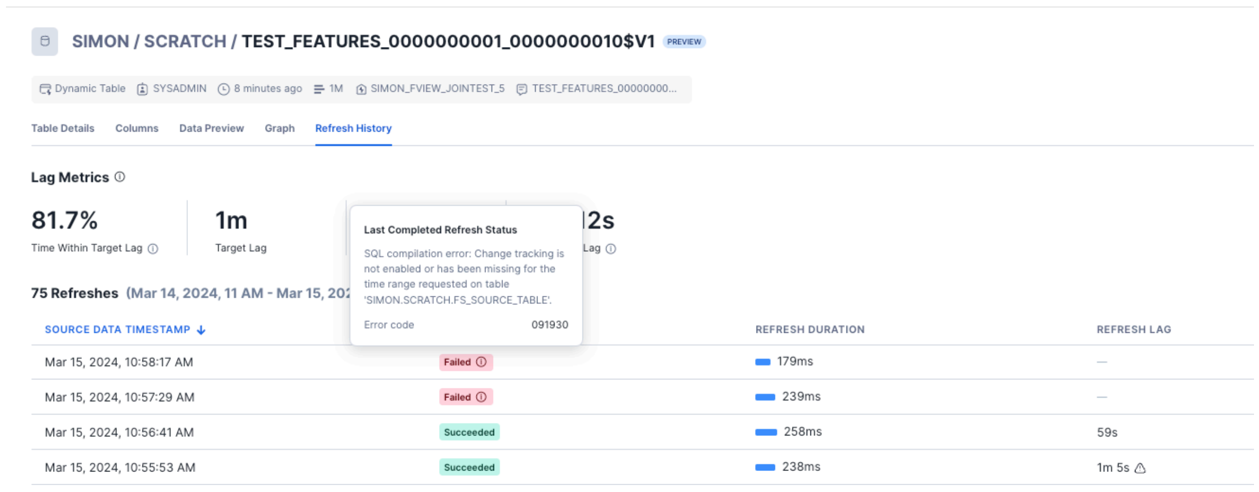
SOURCE DATA TIMESTAMP	STATUS	REFRESH DURATION	REFRESH LAG	ROWS CHANGED		QUERY PROFILE
Mar 15, 2024, 10:54:17 AM	Succeeded	208ms	53s	+0	-0	
Mar 15, 2024, 10:53:29 AM	Succeeded	233ms	59s	+0	-0	
Mar 15, 2024, 10:52:41 AM	Succeeded	244ms	59s	+0	-0	
Mar 15, 2024, 10:51:53 AM	Succeeded	321ms	1m 5s ⚠	+0	-0	
Mar 15, 2024, 10:51:05 AM	Succeeded	761ms	25s	+0	-0	
Mar 15, 2024, 10:50:57 AM	Succeeded	2.1s	—	+1M	-0	

Issue : Underlying Source Table is recreated (e.g. Create table as Select)

We now re-create (create replace) the source table for the ***Feature View***.

```
SQL
create or replace table FS_SOURCE_TABLE as
select * from  TEST_TABLE_1000_FIDS_X_1000000_EIDS_FVECT_CTS_W_D1;
```

Rechecking Snowsight to see the status of the **dynamic table**


SIMON / SCRATCH / TEST_FEATURES_0000000001_0000000010$V1 PREVIEW

Dynamic Table · SYSADMIN · 8 minutes ago · 1M · SIMON_FVIEW_JOINTEST_5 · TEST_FEATURES_00000000...

Table Details · Columns · Data Preview · Graph · Refresh History

Lag Metrics

81.7% Time Within Target Lag · 1m Target Lag · 2s Lag

Last Completed Refresh Status

SQL compilation error: Change tracking is not enabled or has been missing for the time range requested on table 'SIMON.SCRATCH.FS_SOURCE_TABLE'.

Error code 091930

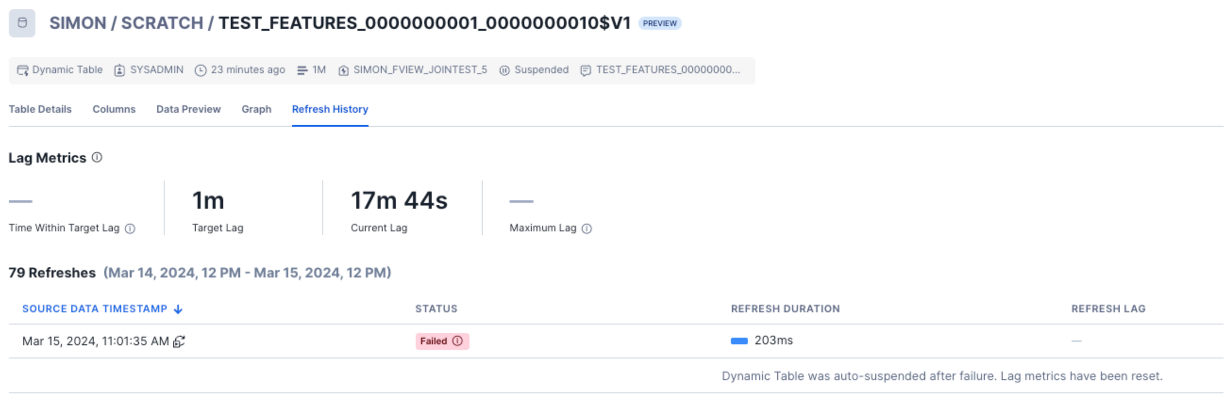
75 Refreshes (Mar 14, 2024, 11 AM - Mar 15, 202

SOURCE DATA TIMESTAMP ↓		REFRESH DURATION	REFRESH LAG
Mar 15, 2024, 10:58:17 AM	Failed	179ms	—
Mar 15, 2024, 10:57:29 AM	Failed	239ms	—
Mar 15, 2024, 10:56:41 AM	Succeeded	258ms	59s
Mar 15, 2024, 10:55:53 AM	Succeeded	238ms	1m 5s

As expected it is now failing with :

```
Last Completed Refresh Status

SQL compilation error: Change tracking is not enabled or has been missing for the time range
requested on table 'FS_SOURCE_TABLE'.

Error code : 091930
```

If we leave it running for five-cycles the **dynamic table** will auto-suspend


SIMON / SCRATCH / TEST_FEATURES_0000000001_0000000010$V1 PREVIEW

Dynamic Table · SYSADMIN · 23 minutes ago · 1M · SIMON_FVIEW_JOINTEST_5 · Suspended · TEST_FEATURES_00000000...

Table Details · Columns · Data Preview · Graph · Refresh History

Lag Metrics

— Time Within Target Lag · 1m Target Lag · 17m 44s Current Lag · — Maximum Lag

79 Refreshes (Mar 14, 2024, 12 PM - Mar 15, 2024, 12 PM)

SOURCE DATA TIMESTAMP ↓	STATUS	REFRESH DURATION	REFRESH LAG
Mar 15, 2024, 11:01:35 AM	Failed	203ms	—

Dynamic Table was auto-suspended after failure. Lag metrics have been reset.

And if we perform a manual refresh:

```
SQL
alter dynamic table TEST_FEATURES_0000000001_0000000010$V1 refresh;
```

We get the following similar error.

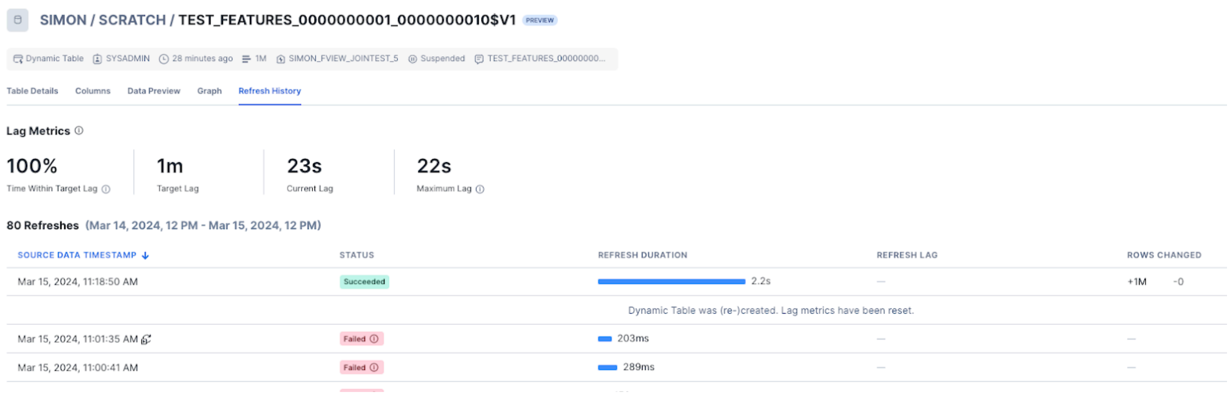
```
Failed to refresh dynamic table with refresh_trigger MANUAL at data_timestamp 1710500495360
because of the error: SQL compilation error: Target table failed to refresh: SQL compilation
error: Change tracking is not enabled or has been missing for the time range requested on table
'FS_SOURCE_TABLE'.
```

Solution
To fix this problem, we need to re-create the ***Feature View*** (or **dynamic table**).

```python
registered_fv_0: FeatureView = fs.register_feature_view(
    feature_view='TEST_FEATURES_0000000001_0000000010',
    version="1",
    block=False,
    override=True,
)
```

```sql


create or replace dynamic table TEST_FEATURES_0000000001_0000000010$V1(
        TID,
        EID,
        CTS,
        F00001,  F00002,  F00003,  F00004,  F00005,  F00006,  F00007,  F00008,  F00009,  F00010
) lag = '1 minute' refresh_mode = AUTO initialize = ON_CREATE warehouse = SIMON_FVIEW_JOINTEST_5
 Comment ='TEST_FEATURES_0000000001_0000000010'
 as
 select "TID", "EID", "CTS",
 "FEAT_VECT":F00001::FLOAT F00001,   "FEAT_VECT":F00002::FLOAT F00002,
 "FEAT_VECT":F00003::FLOAT F00003,   "FEAT_VECT":F00004::FLOAT F00004,
 "FEAT_VECT":F00005::FLOAT F00005,   "FEAT_VECT":F00006::FLOAT F00006,
 "FEAT_VECT":F00007::FLOAT F00007,   "FEAT_VECT":F00008::FLOAT F00008,
 "FEAT_VECT":F00009::FLOAT F00009,   "FEAT_VECT":F00010::FLOAT F00010,
 From FS_SOURCE_TABLE;
```

And we can then see that it has been recreated, and it runs a full-refresh to get the original 1M records back.



SIMON / SCRATCH / TEST_FEATURES_0000000001_0000000010$V1 PREVIEW

Dynamic Table · SYSADMIN · 28 minutes ago · 1M · SIMON_FVIEW_JOINTEST_5 · Suspended · TEST_FEATURES_00000000...

Table Details · Columns · Data Preview · Graph · Refresh History

Lag Metrics

100%	1m	23s	22s
Time Within Target Lag	Target Lag	Current Lag	Maximum Lag

80 Refreshes (Mar 14, 2024, 12 PM - Mar 15, 2024, 12 PM)

SOURCE DATA TIMESTAMP ↓	STATUS	REFRESH DURATION	REFRESH LAG	ROWS CHANGED
Mar 15, 2024, 11:18:50 AM	Succeeded	2.2s	—	+1M -0
		Dynamic Table was (re-)created. Lag metrics have been reset.		
Mar 15, 2024, 11:01:35 AM	Failed	203ms	—	—
Mar 15, 2024, 11:00:41 AM	Failed	289ms	—	—

This full-refresh unavoidably re-computes the entire backfill history scope of the Feature View so could be costly, particularly if the ***Feature View*** (**dynamic table**) has been running incrementally for a long-time. There is no way around this. E.g. If we suspend the **dynamic table** before we re-create the source table, or create a new version of the **dynamic table** and swap it we get similar errors. Ongoing enhancements to **dynamic tables** will improve the resilience to underlying source object changes over time.

Issue : Add column to source table.

What if we add a new column to the source table.

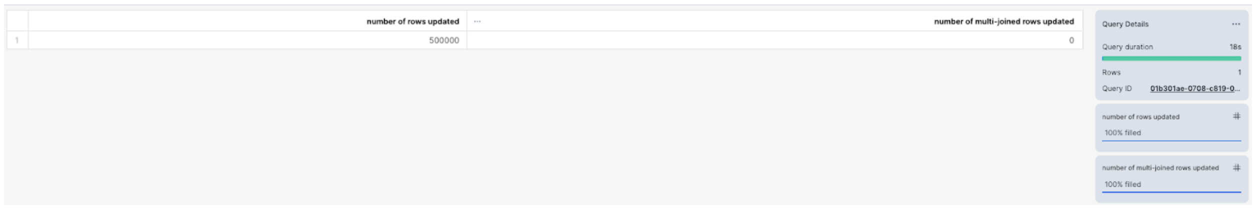
```sql
alter table FS_SOURCE_TABLE add column F01001 float default 0;
```

This should have no impact on the **dynamic table** provided we have not used wild-cards to select column names in the SQL used to define it.

Issue : Add/Update VARIANT column referenced by Dynamic Table

Firstly, what if we add a new KEY/VALUE pair into the variant, **`FEAT_VECT`** column. E.g. we add a new KEY (**`'F99999'`**) that is not referenced by the **dynamic table** select statement.

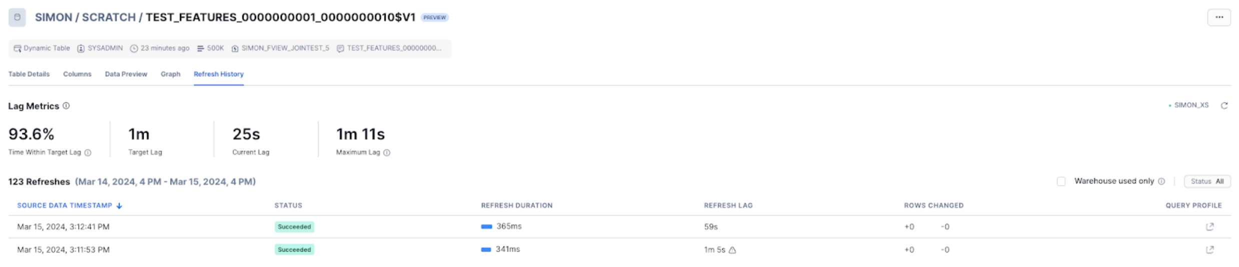
```sql


update FS_SOURCE_TABLE t set FEAT_VECT = o.FEAT_VECT
  from ( select TID, EID, CTS,
                object_insert(FEAT_VECT, 'F99999',
                              uniform(0::float, 1::float, random())) FEAT_VECT
         from FS_SOURCE_TABLE) o
where t.TID = o.TID and
      t.EID = o.EID and
      t.CTS = o.CTS ;
```



	number of rows updated	··· number of multi-joined rows updated
1	500000	0

Query Details
Query duration 18s
Rows 1
Query ID 01b301ae-0708-c819-0...
number of rows updated
100% filled
number of multi-joined rows updated
100% filled

As this created a new Feature Value that is not extracted in the **dynamic table** SQL it does not need to perform an update, even though the `FEAT_VECT` column has been effectively over-written by this Update!



SIMON / SCRATCH / TEST_FEATURES_0000000001_0000000010$V1 PREVIEW

Dynamic Table SYSADMIN 23 minutes ago 500K SIMON_FVIEW_JOINTEST_5 TEST_FEATURES_00000000...

Table Details Columns Data Preview Graph Refresh History

Lag Metrics

SIMON_XS

93.6% Time Within Target Lag
1m Target Lag
25s Current Lag
1m 11s Maximum Lag

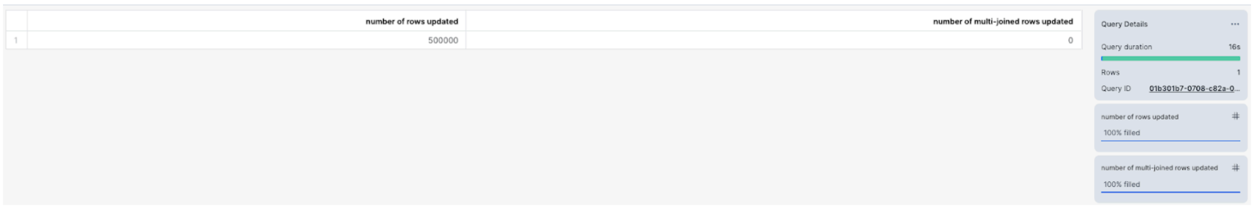
123 Refreshes (Mar 14, 2024, 4 PM - Mar 15, 2024, 4 PM)

Warehouse used only Status All

SOURCE DATA TIMESTAMP ↓	STATUS	REFRESH DURATION	REFRESH LAG	ROWS CHANGED	QUERY PROFILE
Mar 15, 2024, 3:12:41 PM	Succeeded	365ms	59s	+0 -0	
Mar 15, 2024, 3:11:53 PM	Succeeded	341ms	1m 5s ⚠	+0 -0	

Let’s try the same thing, but this time update the existing `F00010` Feature, that is referenced in the **dynamic table**.

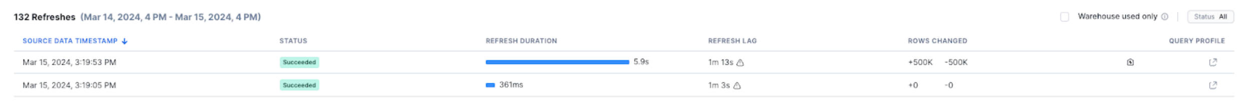
```
SQL
update FS_SOURCE_TABLE t set FEAT_VECT = o.FEAT_VECT
  from ( select TID, EID, CTS,
                object_insert(FEAT_VECT, 'F00010',
                              uniform(0::float, 1::float, random()), true) FEAT_VECT
         from FS_SOURCE_TABLE) o
where t.TID = o.TID and
      t.EID = o.EID and
      t.CTS = o.CTS ;
```



	number of rows updated	number of multi-joined rows updated
1	500000	0

Query Details
Query duration 16s
Rows 1
Query ID 01b301b7-0708-c82a-0...
number of rows updated 100% filled
number of multi-joined rows updated 100% filled

This time because we have impacted a variant column, and sub-element (key) in the **dynamic table** where the source values have changed, we see the expected update applied in the **dynamic table**.



132 Refreshes (Mar 14, 2024, 4 PM - Mar 15, 2024, 4 PM)

Warehouse used only | Status All

SOURCE DATA TIMESTAMP	STATUS	REFRESH DURATION	REFRESH LAG	ROWS CHANGED	QUERY PROFILE
Mar 15, 2024, 3:19:53 PM	Succeeded	5.9s	1m 13s	+500K -500K	
Mar 15, 2024, 3:19:05 PM	Succeeded	361ms	1m 3s	+0 -0	

Recommendations

1. Recreating a **Source table re-created** requires re-creation of the downstream **dynamic tables** (***Feature Views***).
2. ***Feature View*** pipelines**. *Feature Views*** may be sourced directly from other ***Feature Views***. Any downstream ***Feature View*** dependencies will also need to be re-created, when upstream object is recreated.
3. **Visibility/Alerting.**
   - Snowsight can be used to check on the status of **dynamic tables.**
   - This section in the **dynamic table documentation covers monitoring.**
4. **Costly Backfill re-computation.** ***Feature Views*** may contain a considerable amount of historical feature-values that have been built up incrementally through the **dynamic table**. When the ***Feature View*** is re-created, this full historical back-fill needs to be rebuilt from scratch. This could be costly and time consuming to rebuild. An alternative is to leave the broken **dynamic table** in place, and create a new one that derives the features going forward from the point-in-time the original ‘broke’. A union-all **view** can

be used to combine the two into a single logical **Feature View**. The efficacy of this approach, and whether the effort is worthwhile, will vary on a case by case basis.

View based Feature Views

Instead of using a **dynamic table** for the ***Feature View*** we can use the other supported Feature Store option of **views**, which have a different behaviour, when underlying source objects are modified. This may be useful, if we know they underlying data-sources are likely to be volatile to change. To create a **view** based ***Feature View,*** we set `refresh_freq = None` (the default), or remove it, in the ***Feature View*** definition.

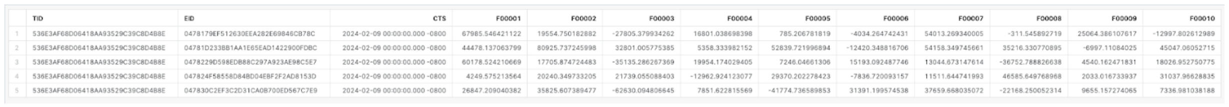
```
SQL
managed_fv = FeatureView(
    name="TEST_FEATURES_0000000001_0000000010",
    entities=[entity],
    feature_df=feature_df,
    timestamp_col="CTS",
    desc="TEST_FEATURES_0000000001_0000000010"
)

registered_fv_0: FeatureView = fs.register_feature_view(
    feature_view=managed_fv,
    version="1",
    block=False,
    override=False,
)
```

Which results in the creation of the following **view**.

```
SQL
create or replace view TEST_FEATURES_0000000001_0000000010$V1(
        TID,
        EID,
        CTS,
        F00001,
        F00002,
        F00003,
        F00004,
        F00005,
        F00006,
        F00007,
        F00008,
        F00009,
        F00010
) comment ='TEST_FEATURES_0000000001_0000000010'
 as
 select "TID", "EID", "CTS",
 "FEAT_VECT":F00001::FLOAT F00001, "FEAT_VECT":F00002::FLOAT F00002,
 "FEAT_VECT":F00003::FLOAT F00003, "FEAT_VECT":F00004::FLOAT F00004,
 "FEAT_VECT":F00005::FLOAT F00005, "FEAT_VECT":F00006::FLOAT F00006,
 "FEAT_VECT":F00007::FLOAT F00007, "FEAT_VECT":F00008::FLOAT F00008,
 "FEAT_VECT":F00009::FLOAT F00009, "FEAT_VECT":F00010::FLOAT F00010,
 from SIMON.SCRATCH.FS_SOURCE_TABLE;
```

We can select from the **view** to get the features computed on-the-fly from the source table.

```sql
select *
from TEST_FEATURES_0000000001_0000000010$V1
limit 5;
```



	TID	EID	CTS	F00001	F00002	F00003	F00004	F00005	F00006	F00007	F00008	F00009	F00010
1	536E3AF68D06418AA93529C39C8D4B8E	0478179EF512630EEA282E69846CB78C	2024-02-09 00:00:00.000 -0800	67985.546421122	19554.750182882	-27805.379934262	16801.038698398	785.206781819	-4034.264742431	54013.269340005	-311.545892719	25064.386107617	-12997.802612989
2	536E3AF68D06418AA93529C39C8D4B8E	04781D233BB1AA1E65EAD1422900FDBC	2024-02-09 00:00:00.000 -0800	44478.137063799	80925.737245998	32801.005775385	5358.333982152	52839.721996894	-12420.348816706	54158.349745661	35216.330770895	-6997.11084025	45047.06052715
3	536E3AF68D06418AA93529C39C8D4B8E	0478229D598EDB88C297A923AE98C5E7	2024-02-09 00:00:00.000 -0800	60178.524210669	17705.874724483	-35135.286267369	19954.174029405	7246.04661306	15193.092487746	13044.673147614	-36752.788826638	4540.162471831	18026.952750775
4	536E3AF68D06418AA93529C39C8D4B8E	047824F58558D84BD04EBF2F2AD8153D	2024-02-09 00:00:00.000 -0800	4249.575213564	20240.349733205	21739.055088403	-12962.924123077	29370.202278423	-7836.720093157	11511.644741993	46585.649768968	2033.016733937	31037.96628835
5	536E3AF68D06418AA93529C39C8D4B8E	047830C2EF3C2D31CA0B700ED567C7E9	2024-02-09 00:00:00.000 -0800	26847.209040382	35825.607389477	-62630.094806645	7851.622815569	-41774.736589853	31391.199574538	37659.668035072	-22168.250052314	9655.157274065	7336.981038188

We recreate the base table using the same CTAS operation as before and re-query the **view**. In the case of a **view** it works, and returns the same result as previously. The **view** is not broken by the change of underlying source table as it references by name, not its object-id.

As before we create a new version of the source table and swap it.

```sql
-- We count the number of records from the View before the SWAP
select count(*)
from TEST_FEATURES_0000000001_0000000010$V1;
-- Returns 1000000

create or replace table FS_SOURCE_TABLE_NEW as
select *
from  SIMON.FEATUREVIEW_JOINTEST_4.TEST_TABLE_1000_FIDS_X_1000000_EIDS_FVECT_CTS_W_D1 limit 500000;
 --Note the record limit, so we can identify which source table our VIEW is now referencing. Our
original with 10m records, or the new swapped table with 5m records

alter table FS_SOURCE_TABLE swap with FS_SOURCE_TABLE_NEW;

-- We count the number of records from the View after the SWAP
select count(*) from TEST_FEATURES_0000000001_0000000010$V1;
-- Returns 500000 !!
```

So the **view** continues to reference the original source object-name (`FS_SOURCE_TABLE` with 5m rows), the swapped in object. Rather than referencing the original source object (now `FS_SOURCE_TABLE_NEW` with the original 10m rows).

This behaviour allows underlying source objects to be managed/maintained without breaking the **view** referencing them. The same behaviour is not possible against **dynamic tables** due to their need to maintain a time-reference for the incremental computation to the original source objects.

Feature Views : Dynamic Table vs View based ?

Dynamic tables maintain (materialise) the result of their SQL definition as a table. Where possible, this will be done incrementally for changing source data, once the initial table has been materialised from source data. When queried the **dynamic table** already has its result pre-computed. **Views** only materialise the result of their SQL definition when they are queried, but filters and projections in the query against the **view** may be pushed down through the **view** to reduce the amount of computation required.

For example, we have a ***Feature View*** computing 100 daily varying time-windowed customer behaviour features for all of our 10M customers, across 3 years of customer event-history. For training we decide we only need to work with a 10% sample (1M) customers, for a point-in-time (day) 1 week prior, which we have in our ***Spine***. We use `generate_training_set` to retrieve the features, which constructs a SQL ASOF join between the ***Spine*** and our ***Feature View***.

For the **dynamic table,** we have already computed the features for all customers, and we can now join to filter and retrieve them. For the **view,** we need to compute the features on-the-fly, but the SQL should apply the filters to this work so logically we only need to compute them for 10% of the customers in event-history, and potentially a subset of the time. The **dynamic table** will most likely be faster to query than the **view,** but depending on the `refresh_freq` of the **dynamic table** it will regularly be running computation to derive and maintain the full feature value history for all 10M customers. If we also only need to use a subset of the features in the ***Feature View***, via a ***Feature Slice***, we will only need to compute that subset of the features on the fly, whereas the **dynamic table** is maintaining all 100 features constantly.

So is a **view** more costly to query when feature-retrieval is needed, than maintaining the features with incremental computation and storage of all Feature Values via a **Dynamic Table**? The answer will depend on a number of factors:

- the number of times the ***Feature View*** is queried to retrieve training/inference, and the scope of those queries.
- the complexity of the feature-derivations, and whether they can be computed incrementally (or not) in a **dynamic table**.
- the volatility of the feature-derivations i.e. how frequently does a feature-value change, which might depend on both its time-span, frequency of events, and how active entities are.
- how well organised the underlying source tables for the ASOF joins issued by ***Feature Store*** e.g. by entity and event-timestamp.
- whether and how well any filters applied in queries (i.e. from the ***Spine*** table) can be pushed down to reduce the total amount of I/O and computation needed.
- Whether the ***Feature View*** is derived from others in a pipeline, and whether those are pre-computed via **dynamic table,** or **view** based. i.e. how much work needs to be computed on-the-fly, versus using pre-computed.

Considerations & Recommendation.

- During the ***Feature View*** experimentation and development phase where it is likely to be more volatile to changes, use a **view** based ***Feature View*** which will minimize the amount of re-computation required. Where possible work with a sample of data via the ***Spine*** to maximise the opportunity to reduce the overall data-scope of on-the-fly computation against the **view**.
- If underlying data-sources are also known to be very volatile to schema or data-changes (insert, update, delete), it will often be better to use a **view** based ***Feature View*** to avoid the re-creation and any heavy incremental changes on a **dynamic table**.
- Tolerance to `refresh_freq`. A **dynamic table** is only as current as its last completed refresh. A longer `refresh_freq` will reduce the amount of computation required. Do you need the very latest state during model-development? It may be acceptable to work with data that is a few days/weeks old depending on application. The `refresh_freq` can be adjusted e.g. if the ***Feature View*** is being used for production inference (batch). You can use the `<feature_view>.update_feature_view(refresh_freq= '<new_frequency>')` to adjust the frequency.
- Align `refresh_freq` to the known change frequency of underlying sources, and the required frequency of retrieval against it. E.g. if you are only using it for a daily scheduled batch inference, `refresh_freq = 'day'`, or a cron schedule that aligns to the batch inference.
- Consider pausing the **dynamic table** refresh (`suspend_feature_view)` when it is not actively in use. You can either `resume_feature_view`, when you need it to be up to date, or manually `refresh_feature_view` to update it on as needed basis.
    - It may not be immediately obvious to users that the ***Feature View*** is suspended (stale). Encourage use of `list_feature_views` or `get_refresh_history` to check status before use.
    - **Dynamic table** can become stale if they are not refreshed within the change data retention period (DATA_RETENTION_TIME_IN_DAYS) of the source tables. You can also consider adjusting the MAX_DATA_EXTENSION_TIME_IN_DAYS setting to suit your needs.

Format Dataframe generated SQL.

The SQL generated from Snowpark dataframe operations is not formatted to make it easy for a human-readability. As Snowflake FeatureStore converts these DataFrames into database objects (Dynamic Tables or Views), passing the query through a formatting tool can make the objects easier to read/review directly.

The following function can be used to format the SQL nicely for readability, and if preferred can convert sub-queries to Common Table Expressions.

```python
import sqlglot
import sqlglot.optimizer.optimizer

def formatSQL (df_in, subq_to_cte = False, print_sql = True):
  """
  Prettify the given SQL from the input dataframe to nest/indent appropriately.
  Optionally replace subqueries with CTEs.
  query_in    : The raw SQL query to be prettified
  subq_to_cte : When TRUE convert nested sub-queries to CTEs

  return: a dataframe with the formatted SQL
  """

  query_in = df_in.queries['queries'][0]

  expression = sqlglot.parse_one(query_in)

  if subq_to_cte:
      query_in = (sqlglot.optimizer.
                     optimizer.eliminate_subqueries(expression).sql() )
  pretty_sql = sqlglot.transpile(query_in,
                               read='snowflake', pretty=True)[0]

  if print_sql:
      print(query_in)
      print(pretty_sql)
  return session.sql(pretty_sql)
```

The function returns a dataframe encapsulating the prettified SQL, which can then be used in the `feature_df` argument of the ***Feature View*** definition.

Time Windowed Features

It is common for features to be derived from aggregations over many varying time windows and aggregation functions. To make it easier to express these types of transformations `snowpark.DataFrameAnalyticsFunctions` have been provided.

In the `TPCH_Sample_Feature_Store Notebook` (section `ROLLING_WINDOWS`) we show an example of using these functions to build a dataframe of time-windowed features and then create a FeatureView from those.

Extending Snowflake native functions with UDFs in Feature Views

Snowflake SQL provides an extensive set of scalar, table and aggregate functions for transforming data, which are also exposed within the Snowpark dataframe API. Combining and using these will typically be the most efficient way to transform your data into features.

When native functions do not have the functionality to support a required transformation, Snowflake provides User-Defined Functions (UDFs). Snowflake UDF's can be written in Python, Java, Scala, Javascript or SQL. UDF's can be scalar, table or aggregate.

You can use UDFs in your ***Feature View*** dataframe definition to apply transformations. ***Dynamic table*** based ***Feature Views*** only support incremental computation when a UDF is defined with the **`IMMUTABLE`** property, not **`VOLATILE`**. An **`IMMUTABLE`** UDF assumes that the function, when called with the same inputs, will always return the same result, e.g. does not contain some random or temporal element.

One application for using a UDF within a ***Feature View*** is to create a view that includes the fitted ML model to perform transformation or inference on the input features.

Lets look at some examples of using UDFs within ***Feature Views.***

FEATURE STORE USAGE AND APPLICATION

Derived Features - Pre-Processing

Model-specific feature processing (transformations)

A common question (or assumption) we hear is *“Where should the feature pre-processing required for a model reside?”* or *“Can I store pre-processed features in the Feature Store?”*

Definition : Feature pre-processing
Feature pre-processing prepares raw data for machine-learning and is the process of treating and standardising features such that they can be directly used within an ML model and are better suited to the modelling problem. For example, many ML models make assumptions about the form of their input features, or may not perform well with untreated features. It is an important step in the ML process, and can have the most impact on the performance and stability of the model.

Pre-Processing Examples:

Categorical features
Features are converted to a suitable numeric representation. E.g.
- One-hot encoded.
- Hash encoded.

Numeric features
Features are scaled suitably within a common range. E.g.
- Standard, min-max, max-abs or robust scaled

Feature pre-processing is normally a two step process where the transformation is fitted (capturing some state from the data), and then applied to transform the data.

Feature pre-processing in a Feature Store context
As might be implied in our definition, the pre-processing applied to features is often performed specific to the modelling function we are using, and problem we are solving for. They may be re-usable for similar models we create in the future.

As a general best-practice, model-specific processing transformations should be done as part of the model training pipeline. Not as a generalised transform applied to all data within the ***Feature Store*** using a ***Feature View***. There are a number of reasons for this, but arguably the most critical is to avoid information leakage (from shared state) between training, validation and test datasets. Most of the feature transformations capture global state (e.g. min-max scaling gets the minimum and maximum value for each numeric feature) from its input data. Using the entire dataset for this may derive different values, than if applied only to only the training dataset, which in turn may lead to some inaccuracy (over optimistic) results in the models performance. On the other hand, it might be simpler and more compute efficient to implement them once for the entire dataset, and in some cases can provide more stable parameters due to a larger sample size of the data, particularly for rare categories.

This is likely not so much an issue for encoders (e.g categorical), assuming the categories are relatively stable/fixed, in the data, but can be an issue for the scaling type processors that capture volatile state from the sample e.g. min/max scaler.

Ideally, from a statistical perspective, the scaler fit needs to be applied to each distinct sample of data at model training / testing / validation time to avoid data leakage. Hence, it will differ for each new sample of data used in training, testing and cross-validation.   If this was done within the ***Feature Store*** via ***Feature Views***, specific to each sample of data used for a model execution, you would end up with potentially 100's of feature-views, 1 per model/version. It is generally more appropriate for those state values to be fitted and saved within the model-pipeline. The model-pipeline can be stored within the Snowflake model-registry, providing end-to-end lineage from ;

raw source data →
model-generic feature-views →
model-specific preprocessing →
model.

Tip: Snowflake Model-Registry supports transform only pipelines. You no longer need an ML model as the final step of the pipeline. If you have a common set of transformations you want to apply consistently, you can create a transformation only pipeline, fit and store it as a model in Model-Registry, and use it to transform whenever needed.

Encoding Categorical Data

It is common for categorical data to be encoded into numeric values for use within a model. There are various common approaches for this:

- **Ordinal Encoding** - Each distinct value is assigned an integer value used to represent the categorical value. Simplist feature encoding scheme.
- **Label Encoding** - As for Ordinal Encoding, but no inherent ordering in values and their index. Typically used for encoding target variables rather than features.
- **One-Hot Encoding** - Each distinct categorical value is represented as new feature of boolean or 0/1 numeric type. This technique can result in significant feature expansion where a large number of distinct values exist in the unencoded categorical feature and there are various techniques used to avoid this. e.g. only applying encoding to features

with distinct values below a set level, or bucketing distinct values with low cardinality in a feature into a new single category.

Feature storage and query processing overheads a particular consideration for categorical features. Lets assume that we have on average 100 values per categorical feature and we transform these with one-hot encoding we will have 100 new sparse features, per categorical feature, to store in the ***Feature View***. Querying and retrieval of those features will also proportionally take much longer.

There are a number of possible approaches that can be used for. Here's a couple:

1. use Snowpark (or SQL) to generate dataframe code that represents the transformation directly, which is then used as the input dataframe for a FeatureView.
2. use the preprocessing functions available in `scikit-learn` or the scalable equivalents in `snowflake.ml.modelling`. Fit a preprocessing model/pipeline with the data from ***Feature Views***. Save the model into model-registry. Reuse the model/pipeline function (udf) within a dataframe that is then used as the input to derive a ***Feature View***.

In the `TPCH_Sample_Feature_Store Notebook` (section `CATEGORICAL_PREPROCESSING`) we show an example using the first approach showing how a One-Hot Encoding scheme can be achieved using Snowpark Dataframe code alone. For this, we capture the distinct values in categorical columns, derive new columns (via case expressions) using these values to derive the one-hot-encodedWhen you use an Entity within that Feature Store. For simplicity we have assumed that all string based columns are categorical. We have also assigned a limit of 100 as the maximum number of distinct values that we will accept in a categorical column for one-hot encoding before an alternative treatment is necessary. This prevents the column (feature) explosion issue mentioned earlier.

We then show a hybrid example where we use snowflake.ml to fit an `ordinal_encoder`, and extract then encoded values from the `ordinal_encoder` to create a dataframe case expression that will convert the categorical data.

As this converts to usable SQL it is more efficient than running these transformations through a Python UDF.

Numeric Scaling

Numeric scaling is more typically applied as model-specific transformations within the Model pipeline. Once fitted these can be stored within the Snowflake Model-Registry and used on new data when needed. Numeric Scaling techniques store global-state related to the data-set used to FIT them.

That said, it may also be useful to maintain and persist a version of these features computed over the global state, rather than training sample. e.g. for exploratory analysis, and to support 'quick/approximate' experimentation. Below we show an approach to deriving and maintaining these within the ***Feature Store*** via a downstream *statistical summary* ***Feature View***. As the upstream ***Feature View*** changes the downstream *statistical summary* ***Feature View*** global-state values will change to reflect those. We will show an example of this approach below.

In the `TPCH_Sample_Feature_Store Notebook` (section `NUMERIC_PREPROCESSING`) we show an approach to deriving and maintaining these within the ***Feature Store*** as ***Feature Views***.

Partitioned Pre-processing

Rather than pre-processing over an entire dataset, we might need to partition the dataset in some way, and pre-process the data for each partition. For example if we are training many models, one per partition, or we know that there is a high-degree of variance across the partitions.

There are a number of different ways of achieving this. Firstly, we can use scikit-learn or snowflake.ml to fit a preprocessing function on each sub-sample and capture the fitted model. For example we could use a partitioned table function to fit on each subset of data.

Once we have the fitted models we can make use of them from model-registry with Snowpark/SQL to 'transform' data for each partition, and this can also be encapsulated within a ***Feature View***.

In the `TPCH_Sample_Feature_Store Notebook` (section `PARTITIONED_PREPROCESSING`) we show an approach to deriving and maintaining these within the ***Feature Store*** as ***Feature Views***.

Discovery & Metadata

Snowsight

Snowsight supports the exploration of Feature Store within the user-interface.

Python API

The ***Feature Store*** python API contains `list_` and `get_` functions to support the discovery and retrieval of ***Feature Store*** objects (***Entities*** , ***Feature Views*** etc).

Stored Procedures using Feature Store Python API

Should you want to return metadata results from Snowflake SQL, you can use the Feature Store API within Snowpark Stored Procedures. These Stored Procedures return tabular results, that can then be used with `result_scan(<query-id>),` Procedures or via querying the Stored Procedure with `table()`.

list_entities()

SQL

```
CREATE OR REPLACE PROCEDURE LIST_ENTTITIES(fs_database string, fs_schema string)
RETURNS TABLE(NAME VARCHAR, JOIN_KEYS ARRAY, DESC VARCHAR, OWNER VARCHAR )
LANGUAGE PYTHON
RUNTIME_VERSION = '3.9'
PACKAGES = ('snowflake-snowpark-python', 'snowflake-ml-python')
HANDLER = 'run'
AS
$$

from snowflake.snowpark import Session, DataFrame
import snowflake.snowpark.functions as F
import snowflake.snowpark.types as T

from snowflake.ml.feature_store import (
    FeatureStore,
    FeatureView,
    Entity,
    CreationMode
)

def run(session, fs_database, fs_schema):

    vw = session.sql('''select current_warehouse()::varchar()''').collect()[0][0]

    fs = FeatureStore(
        session=session,
        database=fs_database,
        name=fs_schema,
        default_warehouse=vw,
        creation_mode=CreationMode.FAIL_IF_NOT_EXIST,
    )

    return fs.list_entities().select(F.col("NAME"),
F.parse_json(F.col("JOIN_KEYS")).astype(T.ArrayType()).alias("JOIN_KEYS"), F.col("DESC"),
F.col("OWNER") )

$$;

CALL LIST_ENTTITIES('TPCXAI_SF0010_INC','TRAINING_FEATURE_STORE');
SELECT * FROM TABLE(RESULT_SCAN(LAST_QUERY_ID()));

SELECT * FROM TABLE(LIST_ENTTITIES('TPCXAI_SF0010_INC','TRAINING_FEATURE_STORE'));
```

list_feature_views()

SQL

```
CREATE OR REPLACE PROCEDURE LIST_FEATURE_VIEWS(fs_database string, fs_schema string, optional object)
RETURNS TABLE(NAME VARCHAR, VERSION VARCHAR, DATABASE VARCHAR, SCHEMA VARCHAR, CREATED_ON TIMESTAMP,
OWNER VARCHAR, DESC VARCHAR, ENTITIES ARRAY)
LANGUAGE PYTHON
RUNTIME_VERSION = '3.9'
PACKAGES = ('snowflake-snowpark-python', 'snowflake-ml-python')
HANDLER = 'run'
EXECUTE AS CALLER
AS
$$

from snowflake.snowpark import Session, DataFrame
import snowflake.snowpark.functions as F
import snowflake.snowpark.types as T

from snowflake.ml.feature_store import (
    FeatureStore,
    FeatureView,
    Entity,
    CreationMode
)

def run(session, fs_database, fs_schema, optional):

    en = optional.get("entity_name", None)
    fvn = optional.get("feature_view_name", None)

    vw = session.sql('''select current_warehouse()::varchar()''').collect()[0][0]

    fs = FeatureStore(
        session=session,
        database=fs_database,
        name=fs_schema,
        default_warehouse=vw,
        creation_mode=CreationMode.FAIL_IF_NOT_EXIST,
    )

    return fs.list_feature_views(entity_name = en, feature_view_name = fvn) \
             .select(F.col("NAME"),
                    F.col("VERSION"),
                    F.col("DATABASE_NAME"),
                    F.col("SCHEMA_NAME"),
                    F.col("CREATED_ON"),
                    F.col("OWNER"),
                    F.col("DESC"),
                    F.col("ENTITIES").astype(T.ArrayType()).alias("ENTITIES") )

$$;


CALL LIST_FEATURE_VIEWS('TPCXAI_SF0010_INC','TRAINING_FEATURE_STORE', {} );

SELECT * FROM TABLE(LIST_FEATURE_VIEWS('TPCXAI_SF0010_INC','TRAINING_FEATURE_STORE', {} );

CALL LIST_FEATURE_VIEWS('TPCXAI_SF0010_INC','TRAINING_FEATURE_STORE', {'entity_name':'CUSTOMER'} );

SELECT * FROM TABLE(LIST_FEATURE_VIEWS('TPCXAI_SF0010_INC','TRAINING_FEATURE_STORE',
{'entity_name':'CUSTOMER'} ));

CALL LIST_FEATURE_VIEWS('TPCXAI_SF0010_INC','TRAINING_FEATURE_STORE',
{'feature_view_name':'FV_UC01_PREPROCESS'} );

SELECT * FROM TABLE(LIST_FEATURE_VIEWS('TPCXAI_SF0010_INC','TRAINING_FEATURE_STORE',
{'feature_view_name':'FV_UC01_PREPROCESS'} ));


CALL LIST_FEATURE_VIEWS('TPCXAI_SF0010_INC','TRAINING_FEATURE_STORE',
{'feature_view_name':'FV_UC01_PREPROCESS$V1'} );

SELECT * FROM TABLE(LIST_FEATURE_VIEWS('TPCXAI_SF0010_INC','TRAINING_FEATURE_STORE',
{'feature_view_name':'FV_UC01_PREPROCESS$V1'} ));
```

We can take this one step further and execute `retrieve_feature_values()` to return data from a ***Feature View*** with the Python API.

```
SQL


CREATE OR REPLACE PROCEDURE GENERATE_DATASET (fs_database string,
                                                     fs_schema string,
                                                     spine_query string,
                                                     features array,
                                                     output_tablename string,
                                                     optional object)
RETURNS  OBJECT  --TABLE(MESSAGE VARCHAR, FQ_TABLENAME VARCHAR, QUERY_REF VARCHAR)
LANGUAGE PYTHON
RUNTIME_VERSION = '3.9'
PACKAGES = ('snowflake-snowpark-python', 'snowflake-ml-python')
HANDLER = 'run'
EXECUTE AS CALLER
AS
$$

from snowflake.snowpark import Session, DataFrame
import snowflake.snowpark.functions as F
import snowflake.snowpark.types as T
```

```
from snowflake.ml.feature_store import (
    FeatureStore,
    FeatureView,
    Entity,
    CreationMode
)

def run(session, fs_database, fs_schema, spine_query, features, output_tablename, optional):

    spine_df = session.sql(f'''select * from ({spine_query})''')

    materialized_table = output_tablename
    spine_timestamp_col = optional.get("spine_timestamp_col", None)
    save_mode = optional.get("save_mode", "errorifexists")
    include_feature_view_timestamp_col = optional.get("include_feature_view_timestamp_col", None)
    desc = optional.get("desc", None)

    vw = session.sql('''select current_warehouse()::varchar()''').collect()[0][0]

    fs = FeatureStore(
        session=session,
        database=fs_database,
        name=fs_schema,
        default_warehouse=vw,
        creation_mode=CreationMode.FAIL_IF_NOT_EXIST,
    )

    fv = fs.get_feature_view(features[0][0],features[0][1])
    #fv = fs.get_feature_view('FV_UC01_PREPROCESS','V_1')

    gen_ds = fs.generate_dataset(spine_df = spine_df,
                                features = [fv],
                                materialized_table = materialized_table,
                                spine_timestamp_col = spine_timestamp_col,
                                save_mode = save_mode,
                                include_feature_view_timestamp_col =
include_feature_view_timestamp_col,
                                desc = desc
    )

    output_tablename = gen_ds.materialized_table
    qref = session.sql(f'''select system$reference('table', '{output_tablename}', 'PERSISTENT',
'SELECT' ); ''').collect()[0][0]

    return {"MESSAGE":"COMPLETED TRAINING DATASET CREATION", "FQ_TABLENAME":
output_tablename,"QUERY_REF":qref}
$$;
```

Which we can call using

```
SQL
set spine_qref = (Select O_CUSTOMER_SK, max(DATE) from TPCXAI_SF0010_INC.TRAINING.ORDER group by 1)

CALL GENERATE_DATASET(  'TPCXAI_SF0010_INC',            -- database name
                        'TRAINING_FEATURE_STORE',       -- schema/feature-store name
```

```
                            $spine_qref,                    -- query-reference to the Spine query
                            [['FV_UC01_PREPROCESS','V_1']], -- Feature View name/version
                            'FOO3',                         -- Output dataset (table) of training data
                            {} );                           -- optional arguments
```

Feature Store ‘Metadata Views’

The current ***Feature Store*** implementation uses Object Tagging to mark and identify ***FeatureStore*** objects. For example, these are used by the ***Feature Store*** Python API when needed to retrieve ***Feature Store*** objects like ***Feature Views***, and the information returned is then used with Information_Schema and SHOW commands to gather detailed information metadata.

For non-Python or users that want to interact with ***Feature Store*** from SQL working with raw object-tagging views and information-schema metadata requires more in-depth knowledge of the underlying Tag’s and their usage.

To avoid this, and make it easier for users to interact with the ***Feature Store*** from SQL the Views provided below can be used within a ***Feature Store*** schema to retrieve metadata via SQL.

Performance running queries against these views can take multiple seconds (typically 20-30’) depending on the scale of the ***Feature Store*** being interrogated. They may take significantly longer to run against a large ***Feature Store*** schema.

Metadata Views using Feature Store Object Tags

```sql
/*
Note: You will need to change these references below to the database/schema that your FeatureStore resides within
 <database hosting Snowflake Feature Store>
 <schema hosting Snowflake Feature Store>
*/

-- Get current FeatureViews in FeatureStore
create or Replace view FeatureViews as
select
   concat(tr.TAG_DATABASE, '.', tr.TAG_SCHEMA,'.',tr.OBJECT_NAME)  FQ_OBJECT_NAME,
   tr.OBJECT_DATABASE,
   tr.OBJECT_SCHEMA,
   tr.OBJECT_NAME,
   parse_json(fvm.TAG_VALUE):entities::array ENTITY_ARRAY,
   parse_json(fvm.TAG_VALUE):timestamp_col::varchar TIMESTAMP_COL
from snowflake.account_usage.tag_references tr left outer join
   table(snowflake.account_usage.tag_references_with_lineage(
  trim('<database hosting Snowflake Feature Store>')||'.'
||trim('<schema hosting Snowflake Feature Store>')||
       '.SNOWML_FEATURE_VIEW_METADATA')) fvm
on tr.TAG_DATABASE = fvm.OBJECT_DATABASE
   and tr.TAG_SCHEMA = fvm.OBJECT_SCHEMA
   and tr.OBJECT_NAME = fvm.OBJECT_NAME
where true
   and tr.TAG_DATABASE = trim('<database hosting Snowflake Feature Store>')
   and tr.OBJECT_SCHEMA = trim('<schema hosting Snowflake Feature Store>')
   and tr.TAG_NAME = 'SNOWML_FEATURE_STORE_OBJECT'
   -- Ensure we only see the current objects, not including those DROPped
   and tr.OBJECT_DELETED is null
   and fvm.OBJECT_DELETED is null
   and fvm.DOMAIN = 'TABLE';


create or replace view FeatureView_Columns as
select
   fv.OBJECT_DATABASE,
   fv.OBJECT_SCHEMA,
   fv.OBJECT_NAME,
   fvc.COLUMN_ID,
   fvc.COLUMN_NAME,
   ENTITY_ARRAY,
   case when fvc.COLUMN_NAME = fv.TIMESTAMP_COL then true else false end TIMESTAMP_IND
from
   FeatureViews fv left outer join
   table(snowflake.account_usage.tag_references_with_lineage(
   trim('<database hosting Snowflake Feature Store>')||'.'||
   trim('<schema hosting Snowflake Feature Store>')||
        '.SNOWML_FEATURE_STORE_OBJECT')) fvc
   on   fv.OBJECT_DATABASE = fvc.OBJECT_DATABASE
   and  fv.OBJECT_SCHEMA = fvc.OBJECT_SCHEMA
   and  fv.OBJECT_NAME = fvc.OBJECT_NAME
where true
  and DOMAIN = 'COLUMN'
  -- Ensure we only see the current objects, not including those DROPped
  and OBJECT_DELETED is null
  -- Remove Stream metadata columns attached to table
  and left(COLUMN_NAME,9) <> 'METADATA$'
  -- In the current FeatureStore
  and fv.OBJECT_DATABASE = trim('<database hosting Snowflake Feature Store>')
  and fv.OBJECT_SCHEMA = trim('<schema hosting Snowflake Feature Store>')
;
```

Backfiling, Archiving & Tiling best practices

The techniques outlined below are the current best practice approach for managing history and backfill in a ***Feature View*** (**dynamic table**). These techniques are likely to be replaced by new features that are currently in Private Preview in Snowflake, when they become generally available. Namely:

- Immutability Constraints
- Storage lifecycle policies

Managing feature-view history retention dynamically

We may want to control the amount of feature-view history created and retained in a ***Feature View***. Highly volatile Features, those subject to a lot of change, could create a very large number of timestamped history values, increasing ***Feature View*** table size significantly. In many cases the most historical feature values will suffice for training. The methodology below provides an example of how the number of values stored per entity can be adjusted dynamically based on a parameter table.

First, define a parameter table to hold the max number of rows (timestamps) per ***Feature View***

```sql
create or replace table
TPCXAI_SF0010_INC.SERVING_FEATURE_STORE.FV_RETENTION_PARAMETER
(fq_feature_view varchar,
 max_rows_per_order integer);
```

Populate it for the ***Feature View*** we are creating.

```sql
insert into TPCXAI_SF0010_INC.SERVING_FEATURE_STORE.FV_RETENTION_PARAMETER
select 'TPCXAI_SF0010_INC.SERVING_FEATURE_STORE.FV_RETENTION_PARAMETER', 5;
```

Define your 'feature-view' with your required declarative logic to create the full-history of all feature-values.

But in addition, add a join to the parameter table, based on the ***Feature View*** name and add a `row_number/qualify` to your ***Feature View*** logic to set the number of rows based on the parameter.

```sql
create or replace dynamic table TPCXAI_SF0010_INC.SERVING_FEATURE_STORE.FV_UC02_ORDERS_N_LATEST$V_1(
    O_ORDER_ID,
    O_CUSTOMER_SK,
    WEEKDAY,
    STORE,
    ORDER_DATE
) lag = '1 minute' refresh_mode = AUTO initialize = ON_CREATE warehouse = TPCXAI_SF0010_INC_SERVING
 COMMENT='Latest Feature Value rows per Order'
 as
Select
 O_ORDER_ID,
 O_CUSTOMER_SK,
 WEEKDAY,
 STORE,
 "DATE" ORDER_DATE
from TPCXAI_SF0010_INC.SERVING."ORDER" ,
TPCXAI_SF0010_INC.SERVING_FEATURE_STORE.FV_RETENTION_PARAMETER
where O_CUSTOMER_SK in (14933, 11057, 34341, 3390, 20335)
  and fq_feature_view = 'TPCXAI_SF0010_INC.SERVING_FEATURE_STORE.FV_UC02_ORDERS_N_LATEST$V_1'
qualify
    row_number() over (partition by O_CUSTOMER_SK order by DATE desc) <= max_rows_per_order ;
```

In my example, initially I should have five rows for each `O_CUSTOMER_SK`, assuming that there are enough orders per customer history to generate that number. (I made sure to select `O_CUSTOMER_SK`'s with the most orders `(14933, 11057, 34341, 3390, 20335)`). So 25 rows in total in the ***Feature View***.

We then updated the parameter with

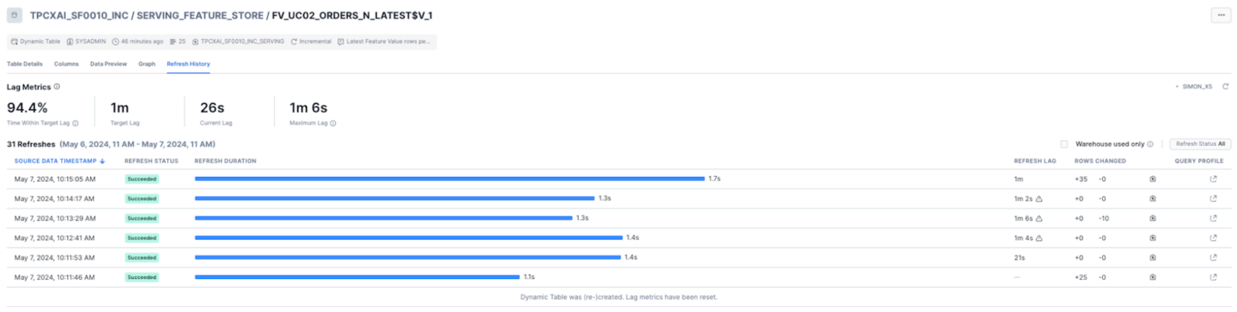
```sql
update TPCXAI_SF0010_INC.SERVING_FEATURE_STORE.FV_RETENTION_PARAMETER
set max_rows_per_order = 3
where fq_feature_view =  'TPCXAI_SF0010_INC.SERVING_FEATURE_STORE.FV_UC02_ORDERS_N_LATEST$V_1';
```

which correctly reduced the rows by -10.

And then increased the parameter

```sql
update TPCXAI_SF0010_INC.SERVING_FEATURE_STORE.FV_RETENTION_PARAMETER
set max_rows_per_order = 10;
```

which correctly increased the rows by 35.


TPCXAI_SF0010_INC / SERVING_FEATURE_STORE / FV_UC02_ORDERS_N_LATEST$V_1

Dynamic Table · SYSADMIN · 46 minutes ago · 25 · TPCXAI_SF0010_INC_SERVING · Incremental · Latest Feature Value rows pe...

Lag Metrics: 94.4% Time Within Target Lag · 1m Target Lag · 26s Current Lag · 1m 6s Maximum Lag

31 Refreshes (May 6, 2024, 11 AM - May 7, 2024, 11 AM)

SOURCE DATA TIMESTAMP	REFRESH STATUS	REFRESH DURATION	REFRESH LAG	ROWS CHANGED		QUERY PROFILE
May 7, 2024, 10:15:05 AM	Succeeded	1.7s	1m	+35	-0	
May 7, 2024, 10:14:17 AM	Succeeded	1.3s	1m 2s ⚠	+0	-0	
May 7, 2024, 10:13:29 AM	Succeeded	1.3s	1m 6s ⚠	+0	-10	
May 7, 2024, 10:12:41 AM	Succeeded	1.4s	1m 4s ⚠	+0	-0	
May 7, 2024, 10:11:53 AM	Succeeded	1.4s	21s	+0	-0	
May 7, 2024, 10:11:46 AM	Succeeded	1.1s	—	+25	-0	

Dynamic Table was (re-)created. Lag metrics have been reset.

This approach gives us the flexibility to define the amount of history in the table based on a parameter which can be increased/reduced as needed, without having to re-create the full **dynamic table**. The query plan to perform the increase/reduction should run incrementally and be computed relatively efficiently only for the difference.

Reducing history will delete rows, whereas increasing will require recomputation of the additional request history. Even if under the hood the **dynamic table** performs a full re-compute this approach provides good usability/control of backfill history. For example we can keep our ***Feature View*** definition/logic the same for all environments (Dev, Prod), and adjust the amount of history retained in each through setting the parameter in the parameter table.

At present Snowpark Python API does not support the `QUALIFY` clause. Instead you can chain two dataframes, where the first generates the `row_number()`, and 2nd downstream dataframe reads and `filter`'s that with a join to the parameter table.

Support was recently added to **dynamic tables** to allow `current_timestamp()` to be used in the definition. With this it is now possible to to perform timestamp maths and filtering and truncation of history from the **dynamic table**. E.g. remove all timestamps older than 3 years from current_timestamp, where 3 is the parameter stored in our parameter table.

Backfilling

If the dataset is huge with very deep historical scope the initial population of the ***Feature View*** (**dynamic table**) could be costly and time-consuming. The full history may not be needed in entirety, at least initially, while we are experimenting with some new features.
The same technique outlined above for archiving we can be used to control this using a parameter table containing a date or period-duration, used to bound the depth of history to be computed when the **dynamic table** is initially created.

For example, say we have 5 years of transactional/event data in our source data, but initially we think we only need to compute feature-history for the most recent year of transactional data.

We create a Feature-Store parameter-table (i.e. shared metadata for the ***Feature Store***) that contains two columns (`FeatureView_Name` (varchar), and `History_Min_Date` (timestamp). My ***Feature View*** then does a join to the parameter-table, `where FeatureView name = 'myfeatureviewname',` and uses the `History_Min_Date` is time-bound the range of transactional data in scope for the **dynamic table**. e.g. `txn_tbl.transaction_timestamp >= History_Min_Date`. You could create a metadata table, per ***Feature View*** as an alternative to having a single table for the whole FeatureStore.

This avoids the full history being computed initially, and allows the the historical scope to be adjusted by updating the `HistoryMinDate` whenever needed in the parameter-table. When `History_Min_Date` is updated in the parameter-table the DynamicTable detects the change as you'd expect and updates the **dynamic table** accordingly, incrementally. This approach should be more cost efficient than dropping the **dynamic table**, and re-creating it with a new hard-coded timestamp for the time-scope.

Avoid hard-coded Constants / Literals in FeatureViews

This same approach should be used to avoid hardcoding any 'constants' or literals within the ***Feature View*** **dynamic table**, especially if these values might need to be changed in the future. It is better to store these within the parameter-table, e.g. in a `variant` column, and join and apply them in the query from that `variant` column. This allows the **dynamic table** to be adjusted, through updating the metadata in the parameter table, rather than through re-creation of the **dynamic table**.

Archiving

Extracting historical data and storing it in alternative table, lower-cost storage. The above approach can be adapted to write the history older than `History_Min_Date` to another table, or COPY to S3 etc, before it is removed by the UPDATE adjusting the `History_Min_Date` in the control table. A Snowflake Task can be used to periodically capture the oldest time-period from the table and write it to another table, before it is truncated from **dynamic table**.

The need to retain a greater ***Feature View*** history scope, than available in the source data. If the source-data history is very granular and therefore expensive to store, I may delete source-data (events) older than a certain time-range (e.g. 1 year). To avoid the **dynamic table** also losing the older history when older source-data is deleted, you need to 'archive' those oldest ***Feature View*** rows to another standard table to retain them, prior to the deletion of those source rows from the source-data tables. A similar approach to the one outlined above is therefore needed where the older ***Feature View*** data is read and stored in archive table via a scheduled task. We would then create a new ***Feature View*** (as a View), that UNION ALL's both the Archive Table (older feature-history) with the **dynamic table** (newer feature data) to create a single logical ***Feature View*** covering the full history scope. I'd expect some customers to consider the above approach more complex than creating an 'external' process (e.g. DBT based) to manage this, rather than using **dynamic tables** and Tasks for this.

Feature View Tiling

'Tiling' in feature engineering refers to a technique where we pre-compute and store aggregated features at different time granularities to optimize the maintenance, performance and computational efficiency. Instead of calculating aggregations, up to every required time granularity on-the-fly from raw event data, we create intermediate "tiles" of pre-aggregated data that can be reused. This tiling approach is particularly useful when dealing with large-scale event data where direct aggregation from raw data to say monthly level would be computationally expensive.

Key benefits of tiling:

- Reduced computation time
- Lower resource usage
- Faster computation times for higher level time-granularities
- Reusability of intermediate results

Here's an example of the **dynamic table** SQL, that demonstrates tiling for calculating weekly and monthly customer purchase metrics from daily and weekly data:

```sql
SQL
-- Assume we have a raw events table
CREATE OR REPLACE TABLE raw_events (
    event_id STRING,
    user_id STRING,
    event_type STRING,
    amount DECIMAL(10,2),
    event_timestamp TIMESTAMP
);

-- 1. First tile: Daily aggregations
CREATE OR REPLACE DYNAMIC TABLE fv_daily_aggregates$v1
    TARGET_LAG = '1 hours'
    WAREHOUSE = 'FEATURE_STORE_WH'
AS
-- Daily Select aggregation
SELECT
    user_id,
    DATE_TRUNC('day', event_timestamp) as event_date,
    COUNT(*) as daily_event_count,
    SUM(amount) as daily_amount_sum,
    AVG(amount) as daily_amount_avg
FROM raw_events
GROUP BY
    user_id,
    DATE_TRUNC('day', event_timestamp);

-- 2. Second tile: Weekly aggregations using daily pre-computations
CREATE OR REPLACE DYNAMIC TABLE fv_weekly_aggregates$v1
    TARGET_LAG = '24 hours'
    WAREHOUSE = 'FEATURE_STORE_WH'
AS
-- Weekly Select aggregation
SELECT
    user_id,
    DATE_TRUNC('week', event_date) as event_week,
    SUM(daily_event_count) as weekly_event_count,
    SUM(daily_amount_sum) as weekly_amount_sum,
    AVG(daily_amount_avg) as weekly_amount_avg
FROM fv_daily_aggregates$v1
GROUP BY
    user_id,
    DATE_TRUNC('week', event_date);

-- 3. Third tile: Monthly aggregations using weekly pre-computations
CREATE OR REPLACE DYNAMIC TABLE fv_monthly_aggregates$v1
    TARGET_LAG = '24 hours'
    WAREHOUSE = 'FEATURE_STORE_WH'
AS
-- Monthly Select aggregation
SELECT
    user_id,
    DATE_TRUNC('month', event_week) as event_month,
    SUM(weekly_event_count) as monthly_event_count,
    SUM(weekly_amount_sum) as monthly_amount_sum,
    AVG(weekly_amount_avg) as monthly_amount_avg
FROM fv_weekly_aggregates$v1
GROUP BY
    user_id,
    DATE_TRUNC('month', event_week);

-- Example query using monthly tiles for feature generation
CREATE OR REPLACE DYNAMIC TABLE fv_monthly_windowed_features$v1
SELECT
    t1.user_id,
    t1.event_month as current_month,
    -- Compare with previous month
    LAG(t1.monthly_amount_sum) OVER (
        PARTITION BY t1.user_id
        ORDER BY t1.current_month
    ) as prev_month_amount,
    -- Calculate 3-month rolling average
    AVG(t1.monthly_amount_sum) OVER (
        PARTITION BY t1.user_id
        ORDER BY t1.current_month
        ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
    ) as rolling_3month_avg_amount
FROM fv_monthly_aggregates$v1 t1;
```

These can be defined within the Feature Store Python API, by supplying the SQL Select as the `feature_df` source (e.g. embedded within `session.sql('''<select aggregation> ''').`

Key Advantages of this implementation approach:

1. Incremental Processing:
    - **dynamic tables** automatically handle incremental updates
    - Each level only processes new or changed data from the previous level
2. Refresh Scheduling:
    - TARGET_LAG defines how fresh the data needs to be
    - Different lags at each level optimize resource usage
3. Dependencies:
    - Each level builds upon the previous level's aggregations ('tiles')
    - Reduces the amount of data that needs to be processed in the aggregation significantly as we move up through the time-granularities
4. Scalability
    - Can handle large volumes of data efficiently
    - Easy to add new aggregation levels
5. Maintenance
    - Self-maintaining through **dynamic tables**
    - Automatic handling of late-arriving data
    - Better warehouse utilization
    - Reduces compute resources needed

Snowflake FeatureStore & ML-Container-Runtime Integration

One of the key accelerations of Container runtime, is the new Data Connector that parallelises the retrieval of data from a Snowflake warehouse, into the Container Runtime. This can significantly reduce the duration to retrieve data for training and inference. Snowflake Feature Store API supports the following mechanisms for creating training/inference datasets:

- `generate_dataset()` : → Dataset
- `generate_training_set()` : → DataFrame (or optionally materialise a table)
- `read_feature_view()` : → DataFrame
- `retrieve_feature_values()` : → DataFrame

Dataloader as a way to load the data for model training. That process starts the ray cluster in the container runtime ml. If we use the FeatureStore api to access, it will bypass the Dataloader altogether. Not sure what would be the behavior when we will get to model training. We can also go the ***Feature View***'s **dynamic table** or View in the schema to load the data with Dataloader but that will kill the purpose of having the feature store/api. If I am missing something please let me know.

Data Connector can retrieve data from both Datasets and DataFrames:

- `DataConnector.from_dataframe()`
- `DataConnector.from_dataset()`

Warning: When using Dataset with DataConnector, the resulting in-memory dataframe retrieved can be larger than when using a DataFrame (table) containing the same data. This is due to a known issue, where larger numeric data-types are allocated than required to contain the data. This issue is in the process of being fixed, but If your data is large and contains a significant number of integer or numeric columns, please use the DataFrame (table) approach as an alternative. Particularly if you see memory failures with Dataset.

OPERATIONS & MONITORING

Feature View (Dynamic Table) monitoring.

The **`get_refresh_history()`** can be used to monitor refreshes for a specific **dynamic table**. To monitor all of the Feature Views in a Feature Store you can use SQL to directly query the **`information_schema`** data.

```Python
# Retrieve Refresh History for a single Feature View
fs.get_refresh_history(FV_NATION__V1)

# Retrieve current Dynamic Table information for all the Feature Views in a Feature Store
session.sql(f"""SELECT
  database_name,
  schema_name feature_store,
  name feature_view,
  scheduling_state,
  target_lag_type,
  target_lag_sec,
FROM
  TABLE ( {fs._config.database}.INFORMATION_SCHEMA.DYNAMIC_TABLES() )
WHERE True
  and database_name = '{fs._config.database}'
  and schema_name = '{fs._config.schema}'
  -- add addtional filters to see a subset of the Feature Views
  and name = 'FV_ORDERS_DMFS$V1'
ORDER BY
  name
""").show()


# Retrieve refresh history information for all the Feature Views in a Feature Store
session.sql(f"""SELECT
  name feature_view,
  state,
  state_code,
  state_message,
  query_id,
  data_timestamp,
  refresh_start_time,
  refresh_end_time,
  statistics,
  refresh_action
FROM
  TABLE (
    {fs._config.database}.INFORMATION_SCHEMA.DYNAMIC_TABLE_REFRESH_HISTORY (
      NAME_PREFIX => '{fs._config.database}.{fs._config.schema}.FV_ORDERS_DMFS$V1' --, ERROR_ONLY =>
TRUE
    )
  )
ORDER BY
  name,
  data_timestamp desc
""").show()
```

This doc page provides additional information on monitoring dynamic tables :
https://docs.snowflake.com/en/user-guide/dynamic-tables-monitor

This section of the documents also explains how you can set up monitoring via a Snowflake Event table, and then setup Alerts to trigger based on required conditions. For example, to send an alert if a ***Feature View*** (**dynamic table**) refresh has failed

Charge Back - Accounting for Consumption

Some thoughts to guide this:

- Creating a feature store in itself doesn't incur any cost. and you can have many feature stores in the same account.
- If teams are each building dedicated feature stores, attributing cost is easy, whatever team the Creator is on would incur all charges for that particular store.
- If multiple teams are going to use a single feature store instance, you'd break down the consumption based on the following costs:
    - Initializing or automatically refreshing a ***Feature View***. Presumably, you'd charge this to the team that created a particular view. Automatic refresh cost is viewable the same way that **dynamic table** refreshes are, so you can see warehouse compute associated with refreshes, and then charge back to the ***Feature View*** owner for automatic refreshes.
    - Auto-refresh require specifying a warehouse, so if you do warehouse-level separation today for chargeback (e.g. each team accesses and uses a different warehouses) you can just charge to the team that owns the WH that is handling the refresh
- Refreshes can also be triggered manually- this should show up in query history and compute could be charged for the warehouse consumption back to the User that triggered a manual refresh (again if they use warehouse-level separation, this is also easy)

Feature data-quality & statistical monitoring

The data in **Dynamic table** based ***Feature Views*** will be evolving based on the changes in the underlying source tables. We might want to monitor these for data-quality issues and any statistical variances that might have an impact on downstream dependent models.

Snowflake provides Data Metric Functions (DMF) as a mechanism for monitoring data-quality in tables and we can use these to set up statistical data monitoring on Feature Views. You can read more about DMF's in the documentation here. In particular, if you're interested in using these for monitoring your feature views, you will need to ensure that the role/user that you are using has the relevant privileges outlined in the documentation. The code below assumes the user has the relevant privileges.

In the code samples below we show how you can setup DMF's on your **dynamic table** based ***Feature Views***, and query the results. Values captured from DMFs are captured at the specified intervals, or triggered on changes to the **Dynamic table**. You can plot and visualise these values, or use other tools to monitor and alert for significant change or outliers, once a reasonable amount of historical values is available. E.g. using Snowflake ML Anomaly Detection.

We will need to monitor features in a statistically appropriate way for their data-type. We create some simple functions that return the features by type.

```python
from snowflake.snowpark.types import *

def get_numeric_columns(dataframe):
    numeric_types = (IntegerType, DoubleType, DecimalType, FloatType, LongType)
    return [field.name for field in dataframe.schema.fields
            if isinstance(field.datatype, numeric_types)]

def get_string_columns(dataframe):
    string_types = (StringType)
    return [field.name for field in dataframe.schema.fields
            if isinstance(field.datatype, string_types)]

def get_time_date_columns(dataframe):
    time_date_types = (TimestampType, TimeType, DateType)
    return [field.name for field in dataframe.schema.fields
            if isinstance(field.datatype, time_date_types)]

# Usage
numeric_cols = get_numeric_columns(orders_fv_v1.feature_df)
print(numeric_cols)
string_columns = get_string_columns(orders_fv_v1.feature_df)
print(string_columns)
time_date_columns = get_time_date_columns(orders_fv_v1.feature_df)
print(time_date_columns)
```

The function below creates all the applicable Data Metric Functions per feature, for a given Feature View.

```python
def fv_create_dmfs(fv):
    if refresh_freq = None:
        print(f'FeatureView fv.fully_qualified_name is a View, not a Dynamic Table')

    fv_df = fv.feature_df
    fv_schema = fv_df.schema
    fv_cols = fv_df.columns
    dmf_cols = fv.feature_names
    anytype_dmf = ['DUPLICATE_COUNT','NULL_COUNT', 'NULL_PERCENT', 'UNIQUE_COUNT' ]
    string_dmf = ['BLANK_COUNT','BLANK_PERCENT' ]
    numeric_dmf = ['AVG', 'MAX', 'MIN', 'STDDEV']
    fv_df_features = fv_df.select(fv.feature_names)

    try:
        session.sql(f'''ALTER TABLE {fv.fully_qualified_name()} SET DATA_METRIC_SCHEDULE =
'TRIGGER_ON_CHANGES';''').collect()
```

```
    except:
        print(f'Schedule TRIGGER_ON_CHANGES on  featureview {fv.fully_qualified_name()}')

    for nc in get_numeric_columns(fv_df_features):
        for dmf_fun in numeric_dmf:
            try:

                session.sql(f''' ALTER TABLE {fv.fully_qualified_name()} ADD DATA METRIC FUNCTION
SNOWFLAKE.CORE.{dmf_fun} ON ({nc});''').collect()
            except:
                print(f'Function {dmf_fun} failed column {nc} on featureview on
{fv.fully_qualified_name()}')

    for sc in get_string_columns(fv_df_features):
        for dmf_fun in string_dmf:
            try:
                session.sql(f''' ALTER TABLE {fv.fully_qualified_name()} ADD DATA METRIC FUNCTION
SNOWFLAKE.CORE.{dmf_fun} ON ({sc});''').collect()
            except:
                print(f'Function {dmf_fun} failed on column {sc} on featureview
{fv.fully_qualified_name()}')

    for ac in dmf_cols:
        for dmf_fun in anytype_dmf:
            try:
                session.sql(f''' ALTER TABLE {fv.fully_qualified_name()} ADD DATA METRIC FUNCTION
SNOWFLAKE.CORE.{dmf_fun} ON ({ac});''').collect()
            except:
                print(f'Function {dmf_fun} failed on column {ac} on featureview
{fv.fully_qualified_name()}')
```

Example usage:

```
Python
# Usage

## Create the data-metric functions on the Features of the Feature View
fv_create_dmfs(orders_fv_v1)

## After some time Return a dataframe of results for further downstream
## analysis.
## Note: You will need a Feature View, that is actively recieving changes
##       in order to see a history of value changes over time.
session.sql(f'''
select
  table_schema feature_store,
  table_name feature_view,
  argument_names[0]::varchar feature_name,
  metric_name,
  measurement_time,
  value measurement_value
from SNOWFLAKE.LOCAL.DATA_QUALITY_MONITORING_RESULTS
where
```

```
   table_database = '{orders_dmfs_fv_v1.database}' and
   table_schema   = '{orders_dmfs_fv_v1.schema}' and
   table_name     = '{orders_dmfs_fv_v1.name}${orders_dmfs_fv_v1.version}'
 order by table_name, feature_name, metric_name, change_commit_time desc
 ''').show()
```

The above example can be expanded as required. For example, we have used the built-in system data-metric functions for the example, but you can create [custom data-metric functions]() that enable additional specific tests.

The `TPCH_Sample_Feature_Store Notebook` (section `DATA_METRIC_FUNCTIONS`) provides the code shared above, plus additional worked example for using it with the TPCH sample data.