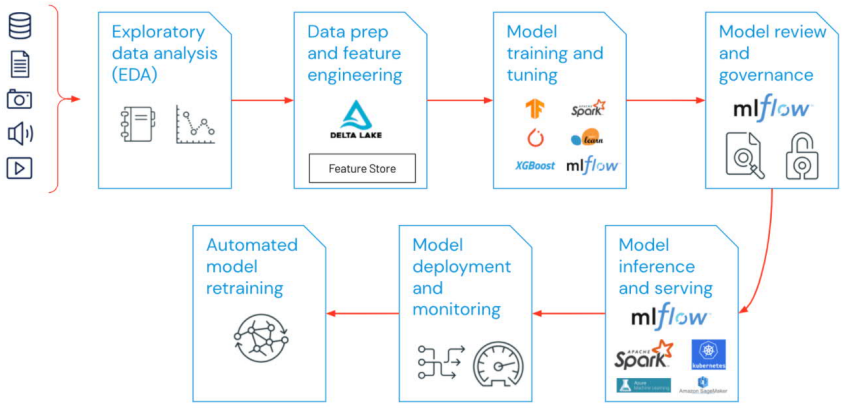
Microsoft

Azure Databricks MLOps

Learning Units covered in this Module

- Lesson 1: Feature Store
- Lesson 2: Experiment Tracking with MLflow
- Lesson 3: Model Register with MLflow

MLOps Components



Reference: https://databricks.com/blog/2020/10/13/using-mlops-with-mlflow-and-azure.html

- A majority of enterprises deploy MLOps principles across the following:Exploratory data analysis (EDA)
- Data Prep and Feature Engineering
- Model training and tuning
- Model review and governance
- Model inference and serving
- Model monitoring
- Automated model retraining

MLOps in Azure Databricks

- Create reproducible ML pipelines
- Create reusable software environments
- Register, package, and deploy models from anywhere
- Capture the governance data for the end-to-end ML lifecycle
- Notify and alert on events in the ML lifecycle
- Monitor ML applications for operational and ML-related issues
- Automate the end-to-end ML lifecycle with Azure Machine Learning and Azure Pipelines

•Create reproducible ML pipelines. Machine Learning pipelines allow you to define repeatable and reusable steps for your data preparation, training, and scoring processes.

•Create reusable software environments for training and deploying models.

•Register, package, and deploy models from anywhere. You can also track associated metadata required to use the model.

•Capture the governance data for the end-to-end ML lifecycle. The logged lineage information can include who is publishing models, why changes were made, and when models were deployed or used in production.

•Notify and alert on events in the ML lifecycle. For example, experiment completion, model registration, model deployment, and data drift detection.

•Monitor ML applications for operational and ML-related issues. Compare model inputs between training and inference, explore model-specific metrics, and provide monitoring and alerts on your ML infrastructure.

•Automate the end-to-end ML lifecycle with Azure Machine Learning and Azure Pipelines. Using pipelines allows you to frequently update models, test new models, and continuously roll out new ML models alongside your other applications and services.

Governance Data for MLOps

Azure Databricks integrates with Git

Azure Databricks Datasets help you track feature store.

Azure Databricks experiments stores a snapshot of the code, data, and computes used to train a model.

The Databricks Model Registry captures all of the metadata associated with your model (which experiment trained it, where it is being deployed, if its deployments are healthy).

https://docs.databricks.com/aws/en/mlflow/workspace-model-registry-example

This example illustrates how to use the Workspace Model Registry to build a machine learning application that forecasts the daily power output of a wind farm. The example shows how to:

- Track and log models with MLflow
- Register models with the Model Registry
- Describe models and make model version stage transitions
- Integrate registered models with production applications
- Search and discover models in the Model Registry
- Archive and delete models

The article describes how to perform these steps using the MLflow Tracking and MLflow Model Registry UIs and APIs.

Lesson 1: Feature Store in Azure Databricks

Databricks Feature Store

- It is a centralized repository of features.
- It enables feature sharing and discovery across your organization
- It ensures that the same feature computation code is used for model training and inference.
- Available only on Databricks Runtime for Machine Learning
- Deployed MLflow models can automatically retrieve features from Feature Store.

https://learn.microsoft.com/en-us/azure/databricks/machine-learning/feature-store/

Feature Store concepts

Feature table

- Features are organized as feature tables.
- Each table is backed by a Delta table and additional metadata.
- A feature table must have a primary key.
- Feature table metadata tracks the data sources and the notebooks and jobs.
- Can publish a feature table to an online store for real-time model inference.

Offline store

- Used for feature discovery, model training, and batch inference.
- It contains feature tables materialized as Delta tables.

Feature Store concepts

Online store

- a low-latency database used for real-time model inference.
- supports these online stores:
  - Azure Database for MySQL
  - Azure SQL Database

Streaming

- Can write feature values to a feature table from a streaming source
- Can utilize Structured Streaming to transform raw data streams into features.
- Can also stream feature tables from the offline store to an online store.

Feature Store concepts

Training set

- Consists of a list of features and a DataFrame containing raw training data, labels, and primary keys by which to look up features.
- Specifies features to extract from Feature Store.

Model packaging

- A machine learning model can retain references to these used features.
- At inference time, the model can optionally retrieve feature values.
- The caller only needs to provide the primary key of the features (e.g., user_id), and the model retrieves all required feature values.
- In batch inference, feature values are retrieved from the offline store and joined with new data prior to scoring. Real-time inference is not supported.
- To package a model with feature metadata, use FeatureStoreClient.log_model().

Feature Store workflow overview

1. Write code to convert raw data into features and create a Spark DataFrame containing the desired features.
2. Write the DataFrame as a feature table in Feature Store.
3. Create a training set based on features from feature tables.
4. Train a model.
5. Log the model as an MLflow model.
6. Perform batch inference on new data. The model automatically retrieves the features it needs from Feature Store.
7. (Optional) Publish the features to an online store for batch or real-time inference.

https://learn.microsoft.com/en-us/azure/databricks/machine-learning/feature-store/

Work with feature tables

- Create a database for feature tables
- Create a feature table in Databricks Feature Store
- Update a feature table
- Read from a feature table
- Create a training dataset
- Train models and perform batch inference with feature tables
- Publish features to an online feature store
- Supported data types

Work with feature tables

Create a database for feature tables

1. `%sql CREATE DATABASE IF NOT EXISTS <database_name>`
2. When you publish a feature table to an online store, can specify different names using the **`publish_table`** method.
3. Database and feature table names cannot contain hyphens (-).
4. For example, Delta table named **`customer_features`** in the database **`recommender_system name='recommender_system.customer_features'`**

Work with feature tables

Create a feature table in Databricks Feature Store

```
from databricks.feature_store import feature_table

def compute_customer_features(data):
  ''' Feature computation code returns a DataFrame with 'customer_id' as primary key'''
  pass

customer_features_df = compute_customer_features(df)
from databricks.feature_store import FeatureStoreClient
fs = FeatureStoreClient()
customer_feature_table = fs.create_table(
  name='recommender_system.customer_features',
  primary_keys='customer_id',
  schema=customer_features_df.schema,
  description='Customer features'
)
```

Work with feature tables

Create a feature table in Databricks Feature Store

```
from databricks.feature_store import feature_table

def compute_customer_features(data):
  ''' Feature computation code returns a DataFrame with 'customer_id' as PK'''
  pass

customer_features_df = compute_customer_features(df)
from databricks.feature_store import FeatureStoreClient
fs = FeatureStoreClient()
customer_feature_table = fs.create_table(
  name='recommender_system.customer_features',
  primary_keys='customer_id',
  schema=customer_features_df.schema,
  description='Customer features'
)
```

Value class describing one feature table. This will typically not be instantiated directly, instead the FeatureStoreClient.create_table will create FeatureTable objects.

take schema from DataFrame output

Work with feature tables

Create a feature table in Databricks Feature Store

```
from databricks.feature_store import feature_table

def compute_customer_features(data):
  ''' Feature computation code returns a DataFrame with 'customer_id' as primary key'''
  pass

customer_features_df = compute_customer_features(df)
from databricks.feature_store import FeatureStoreClient
fs = FeatureStoreClient()
customer_feature_table = fs.create_table(
  name='recommender_system.customer_features',
  primary_keys='customer_id',
  schema=customer_features_df.schema,
  description='Customer features'
)
```


Create feature store client

Work with feature tables

Create a feature table in Databricks Feature Store

```
from databricks.feature_store import feature_table

def compute_customer_features(data):
  ''' Feature computation code returns a DataFrame with 'customer_id' as primary
key'''
  pass

customer_features_df = compute_customer_features(df)
from databricks.feature_store import FeatureStoreClient
fs = FeatureStoreClient()
customer_feature_table = fs.create_table(
  name='recommender_system.customer_features',
  primary_keys='customer_id',
  schema=customer_features_df.schema,
  description='Customer features'
)
```

Work with feature tables

Create a feature table in Databricks Feature Store

```
# An alternative is to use `create_table` and specify the `df` argument.
# This code automatically saves the features to the underlying Delta table.
customer_feature_table = fs.create_table(
  ...
  df=customer_features_df,
  ...
 )

# To use a composite key, pass all keys in the create_table call
 customer_feature_table = fs.create_table(
   ...
   primary_keys=['customer_id', 'date'],
   ...
 )
```

Work with feature tables

Create a feature table in Databricks Feature Store

```
# An alternative is to use `create_table` and specify the `df` argument.
# This code automatically saves the features to the underlying Delta table.
customer_feature_table = fs.create_table(
  ...
  df=customer_features_df,
  ...
 )

# To use a composite key, pass all keys in the create_table call
 customer_feature_table = fs.create_table(
   ...
   primary_keys=['customer_id', 'date'],
   ...
 )
```

Work with feature tables

Create a feature table in Databricks Feature Store

```
fs.write_table(
  name='recommender_system.customer_features',
  df = customer_features_df,
  mode = 'overwrite'
)
```

Overwrite mode does a full refresh of the feature table

Work with feature tables

Update a feature table

- add new features.
- modify specific rows based on the primary key.
- The following feature table metadata cannot be updated:
  - Primary key
  - Partition key
  - Name or type of an existing feature

Work with feature tables

Update a feature table

- Add new features to an existing feature table with two methods

o Method 1

1. Update the existing feature computation function
2. run write_table with the returned DataFrame.

   This updates the feature table schema and merges new feature values based on the primary key.

o Method 2

1. Create a new feature computation function to calculate the new feature values.

   The new returned DataFrame must contain the feature tables's primary keys and partition keys (if defined).
2. Run write_table with the DataFrame to write the new features to the existing feature table, using the same primary key.

Work with feature tables

Update a feature table

- Update only specific rows in a feature table

```
fs = FeatureStoreClient()
customer_features_df = compute_customer_features(data)
fs.write_table(
  df=customer_features_df,
  name='recommender_system.customer_features',
  mode='merge'
)
```

Work with feature tables

Update a feature table

- Schedule a job to update a feature table
  - create a job that runs a notebook to update your feature table on a regular basis.
  - Or convert a non-scheduled job created to a scheduled job

- Store past values of daily features
  - Define a feature table with a composite primary key. Include the date in the primary key.
  - Create code to read from the feature table filtering date to the time period of interest.
  - To keep the feature table up to date, set up a regularly scheduled job to write features, or stream new feature values into the feature table.

Work with feature tables

Read from a feature table

- Read data from a specific timestamp

```
import datetime
yesterday = datetime.date.today() - datetime.timedelta(days=1)

customer_features_df = fs.read_table(
  name='recommender_system.customer_features',
  as_of_delta_timestamp=str(yesterday)
)
```

Use read_table to read feature values.

Set the timestamp

Work with feature tables

Create a training dataset

- Create a FeatureLookup to specify each feature you want to use in the training set.
- The argument lookup_key represents the names of the columns in the provided training_df data to join with each feature table's primary keys.
- Call create_training_set to define the training dataset.

Work with feature tables

Create a training dataset

```
from databricks.feature_store import FeatureLookup

feature_lookups = [
   FeatureLookup(
     table_name = 'recommender_system.customer_features',
     feature_name = 'total_purchases_30d',
     lookup_key = 'customer_id'
   ),
   FeatureLookup(
     table_name = 'recommender_system.product_features',
     feature_name = 'category',
     lookup_key = 'product_id'
   )
 ]
```

Create a FeatureLookup to specify each feature you want to use in the training set.

Model training flow uses these features

lookup_key represents the names of the columns in the provided training_df data to join with each feature table's primary keys.

Work with feature tables

Create a training dataset

```
fs = FeatureStoreClient()

training_set = fs.create_training_set(
  df=training_df,
  feature_lookups = feature_lookups,
  label = 'rating',
  exclude_columns = ['customer_id', 'product_id']
)

training_df = training_set.load_df()
```

Create a training set using training DataFrame and features from Feature Store

Training DataFrame training_df must have lookup keys 'customer_id' and 'product_id' and label 'rating'

Exclude those columns using exclude_columns

Work with feature tables

Train models and perform batch inference with feature tables

```
# Train model
import mlflow
from sklearn import linear_model
feature_lookups = [
    FeatureLookup(
      table_name = 'recommender_system.customer_features',
      feature_name = 'total_purchases_30d',
      lookup_key = 'customer_id',
    ),
    FeatureLookup(
      table_name = 'recommender_system.product_features',
      feature_name = 'category',
      lookup_key = 'product_id'
    )
  ]
```

Work with feature tables

Train models and perform batch inference with feature tables

```
fs = FeatureStoreClient()

with mlflow.start_run():

training_set = fs.create_training_set(
   df,
   feature_lookups = feature_lookups,
   label = 'rating',
   exclude_columns = ['customer_id', 'product_id']
 )

 training_df = training_set.load_df().toPandas()

 X_train = training_df.drop(['rating'], axis=1)
 y_train = training_df.rating
```

df has columns ['customer_id', 'product_id', 'rating']

use the DataFrame returned by TrainingSet.load_df to train the model

"training_df" columns ['total_purchases_30d', 'category', 'rating']

Work with feature tables

Publish features to an online feature store

- Publish batch-computed features to an online store

```
import datetime
from databricks.feature_store.online_store_spec import AzureMySqlSpec
def getSecret(key, scope="feature-store"):
  return dbutils.secrets.get(scope, key)

hostname = getSecret("hostname")
port = int(getSecret("port"))
user = getSecret(key="user")
password = getSecret(key="password")

online_store = AzureMySqlSpec(hostname, port, user, password)

fs.publish_table(
  name='recommender_system.customer_features',
  online_store=online_store,
  filter_condition=f"_dt = '{str(datetime.date.today())}'",
  mode='merge'
)
```

assumes that you have set up Azure Databricks secrets to store your credentials.

assumes that an online database named "recommender_system" already exists in the online store and matches the name of the offline store.

If there is no table named "customer_features" in the database, this code creates one.

It assumes that features are computed each day and stored as a partitioned column _dt.

Work with feature tables

Publish features to an online feature store

- Publish batch-computed features to an online store

```
import datetime
from databricks.feature_store.online_store_spec import AzureMySqlSpec
def getSecret(key, scope="feature-store"):
  return dbutils.secrets.get(scope, key)

hostname = getSecret("hostname")
port = int(getSecret("port"))
user = getSecret(key="user")
password = getSecret(key="password")

online_store = AzureMySqlSpec(hostname, port, user, password)

fs.publish_table(
  name='recommender_system.customer_features',
  online_store=online_store,
  filter_condition=f"_dt = '{str(datetime.date.today())}'",
  mode='merge'
)
```

assumes that you have set up Azure Databricks secrets to store your credentials.

assumes that an online database named "recommender_system" already exists in the online store and matches the name of the offline store.

If there is no table named "customer_features" in the database, this code creates one.

It assumes that features are computed each day and stored as a partitioned column _dt.

Work with feature tables

Publish features to an online feature store

- Publish batch-computed features to an online store

```
import datetime
from databricks.feature_store.online_store_spec import AzureMySqlSpec
def getSecret(key, scope="feature-store"):
  return dbutils.secrets.get(scope, key)

hostname = getSecret("hostname")
port = int(getSecret("port"))
user = getSecret(key="user")
password = getSecret(key="password")

online_store = AzureMySqlSpec(hostname, port, user, password)

fs.publish_table(
  name='recommender_system.customer_features',
  online_store=online_store,
  filter_condition=f"_dt = '{str(datetime.date.today())}'",
  mode='merge'
)
```

assumes that you have set up Azure Databricks secrets to store your credentials.

assumes that an online database named "recommender_system" already exists in the online store and matches the name of the offline store.

If there is no table named "customer_features" in the database, this code creates one.

It assumes that features are computed each day and stored as a partitioned column _dt.

Work with feature tables

Publish features to an online feature store

- Publish streaming features to an online store

```
fs.publish_table(
  name='recommender_system.customer_features',
  online_store=online_store,
  streaming=True
)
```

To continuously stream features to the online store, set streaming=True.

Work with feature tables

Publish features to an online feature store

- Publish selected features to an online store

```
fs.publish_table(
  name='recommender_system.customer_features',
  online_store=online_store,
  features=["total_purchases_30d"]
)
```

To publish only selected features to the online store, use the features argument to specify the feature name(s) to publish.

Primary keys and timestamp keys are always published.

If you do not specify the features argument or if the value is None, all features from the offine feature table are published.

Work with feature tables

Publish features to an online feature store

- Publish a feature table to a specific database

```
db = new_db
table = new_table

online_store = AzureMySqlSpec(hostname, port, user, password, db, table)
```

specify the database name (db) and the table name (table).

If you do not specify these parameters, the offline feature table name and database name are used.

Work with feature tables

Supported data types

- Feature Store supports the following PySpark data types:
  - IntegerType
  - FloatType
  - BooleanType
  - StringType
  - DoubleType
  - LongType
  - TimestampType
  - DateType
  - ShortType (Databricks Runtime 9.1 LTS ML and above)
  - BinaryType (Databricks Runtime 10.1 ML and above)
  - DecimalType (Databricks Runtime 10.1 ML and above)
  - ArrayType (Databricks Runtime 9.1 LTS ML and above)
  - MapType (Databricks Runtime 10.1 ML and above)

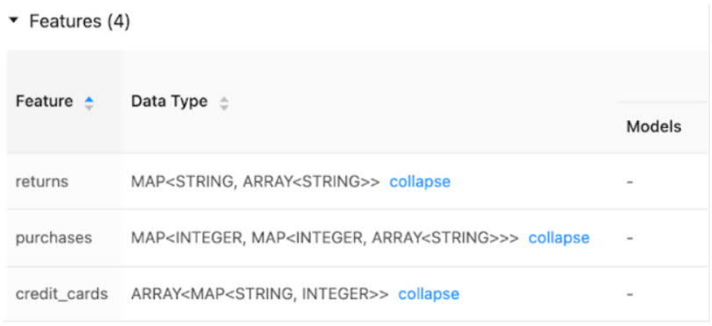
When published to online stores, ArrayType and MapType features are stored in JSON format.

Work with feature tables

Supported data types

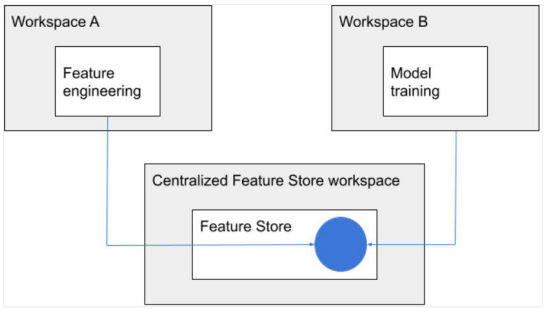
- Feature Store supports the following PySpark data types:

The Feature Store UI displays metadata on feature data types:



Share feature tables across workspaces

- From your own workspace, you can create, write to, or read from a feature table in a centralized feature store.
- For a centralized feature store, can designate a single workspace to store all feature store metadata, and create accounts for each user who needs access to the feature store.
- If your teams are also sharing models across workspaces, may dedicate the same centralized workspace for both feature tables and models, or specify different centralized workspaces for each.



Share feature tables across workspaces

- Requirements
- Set up the API token for a remote registry
- Specify a remote feature store
- Create a feature table in the remote feature store
- Use a feature table from the remote feature store
- Use a remote model registry

Share feature tables across workspaces

Requirements

- Databricks Runtime 10.2 ML or above.
- Both workspaces must have access to the raw feature data. They must share the same Hive metastore and have access to the same DBFS storage.
- If IP access lists are enabled, workspace IP addresses must be on access lists.

Share feature tables across workspaces

Set up the API token for a remote registry

- In the feature store workspace, create an access token.
- In the local workspace, create secrets to store the access token and the remote workspace information:
  - Create a secret scope
    - databricks secrets create-scope --scope <scope>
  - Create three secrets with the unique name
    - databricks secrets put --scope <scope> --key <prefix>-host
    - databricks secrets put --scope <scope> --key <prefix>-token
    - databricks secrets put --scope <scope> --key <prefix>-workspace-id

    E.g., Hostname:

    https://westus.azuredatabricks.net/

    https://adb-5555555555555555.19.azuredatabricks.net/

    Format: adb-<workspace-id>.<random-number>.azuredatabricks.net

Enter the hostname of the feature store workspace

Enter the access token from the feature store workspace.

Enter the workspace ID for the feature store workspace

You may want to share the secret scope with other users, since there is a limit on the number of secret scopes per workspace.

Share feature tables across workspaces

Specify a remote feature store

- feature_store_uri = f'databricks://<scope>:<prefix>'
- fs = FeatureStoreClient(feature_store_uri=feature_store_uri)

Construct a feature store URI

Specify the URI explicitly when you instantiate a FeatureStoreClient

Create a feature table in the remote feature store

```
fs = FeatureStoreClient(feature_store_uri=f'databricks://<scope>:<prefix>')
fs.create_table(
  name='recommender.customer_features',
  primary_keys='customer_id',
  schema=customer_features_df.schema,
  description='Customer-keyed  features'
)
```

For Databricks runtime 10.2 ML and above

For Databricks runtime 10.1 ML and below
Use FeatureStoreClient.create_feature_table
Use keys='customer_id',

Share feature tables across workspaces

Specify a remote feature store

- feature_store_uri = f'databricks://<scope>:<prefix>'
- fs = FeatureStoreClient(feature_store_uri=feature_store_uri)

Construct a feature store URI

Specify the URI explicitly when you instantiate a FeatureStoreClient

Create a feature table in the remote feature store

```
fs = FeatureStoreClient(feature_store_uri=f'databricks://<scope>:<prefix>')
fs.create_table(
  name='recommender.customer_features',
  primary_keys='customer_id',
  schema=customer_features_df.schema,
  description='Customer-keyed features'
)
```

For Databricks runtime 10.2 ML and above

For Databricks runtime 10.1 ML and below
Use FeatureStoreClient.create_feature_table
Use keys='customer_id',

Share feature tables across workspaces

Use a feature table from the remote feature store

```
fs = FeatureStoreClient(feature_store_uri=f'databricks://<scope>:<prefix>')
customer_features_df = fs.read_table(
  name='recommender.customer_features',
)
```

Set feature_store_uri for the remote feature store

Other helper methods for accessing the feature table

```
fs.read_table()
fs.get_feature_table() # in Databricks Runtime 10.1 ML and below
fs.get_table()         # in Databricks Runtime 10.2 ML and above
fs.write_table()
fs.publish_table()
fs.create_training_set()
```

Share feature tables across workspaces

Use a remote model registry

```
fs = FeatureStoreClient(model_registry_uri=f'databricks://<scope>:<prefix>')
customer_features_df = fs.log_model(
    model,
    "recommendation_model",
    flavor=mlflow.sklearn,
    training_set=training_set,
    registered_model_name="recommendation_model"
)
```

use a model registry URI to instantiate a FeatureStoreClient to specify a remote model registry for model logging or scoring

```
fs = FeatureStoreClient(
    feature_store_uri=f'databricks://<scope>:<prefix>',
    model_registry_uri=f'databricks://<scope>:<prefix>'
)
```

Using feature_store_uri and model_registry_uri, you can train a model using any local or remote feature table, and then register the model in any local or remote model registry.

Share feature tables across workspaces

Use a remote model registry

```
fs = FeatureStoreClient(model_registry_uri=f'databricks://<scope>:<prefix>')
customer_features_df = fs.log_model(
    model,
    "recommendation_model",
    flavor=mlflow.sklearn,
    training_set=training_set,
    registered_model_name="recommendation_model"
)
```

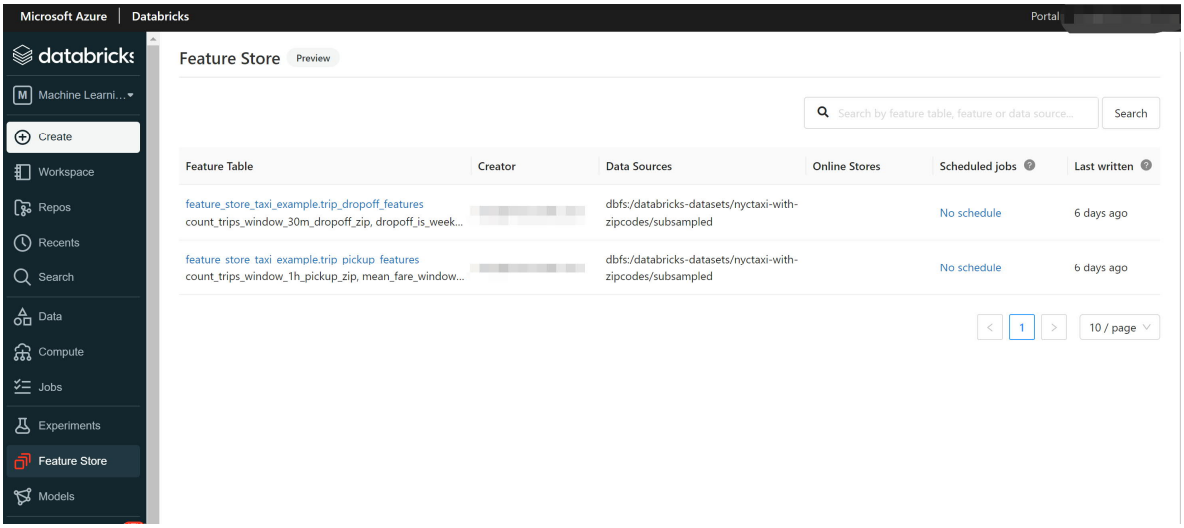
use a model registry URI to instantiate a FeatureStoreClient to specify a remote model registry for model logging or scoring

```
fs = FeatureStoreClient(
    feature_store_uri=f'databricks://<scope>:<prefix>',
    model_registry_uri=f'databricks://<scope>:<prefix>'
)
```

Using feature_store_uri and model_registry_uri, you can train a model using any local or remote feature table, and then register the model in any local or remote model registry.

Use the Feature Store UI

Search and browse for feature tables

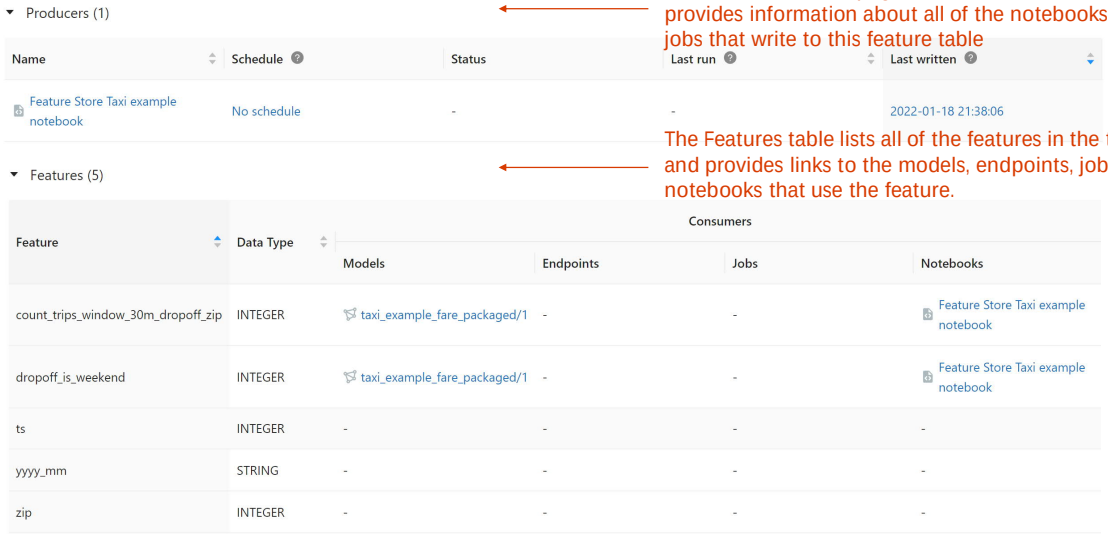


Use the Feature Store UI

Track feature lineage and freshness

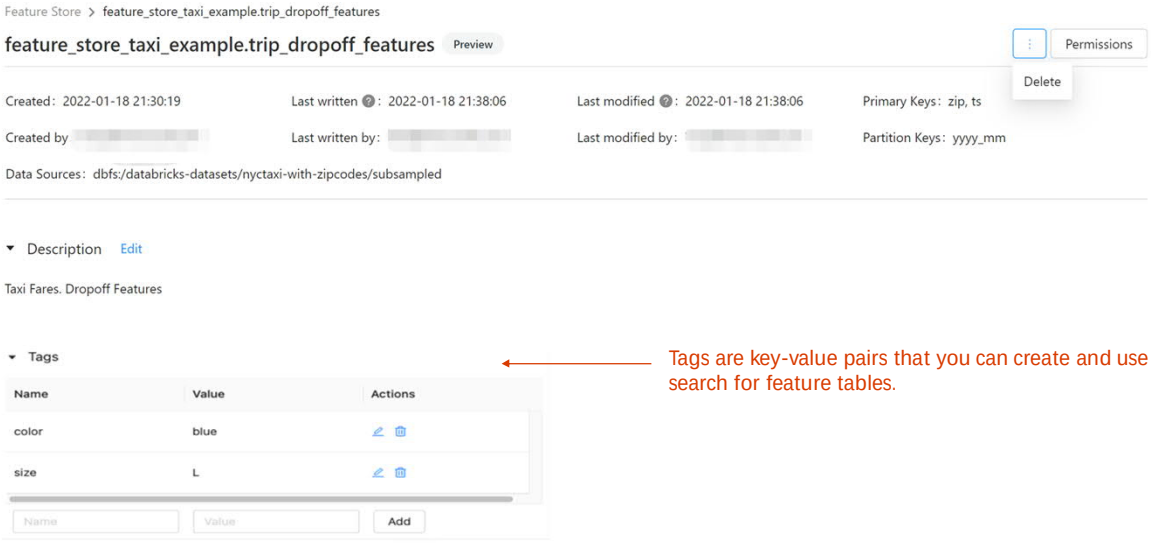
On the feature table page, the Producers table provides information about all of the notebooks and jobs that write to this feature table

The Features table lists all of the features in the table and provides links to the models, endpoints, jobs, and notebooks that use the feature.



Use the Feature Store UI

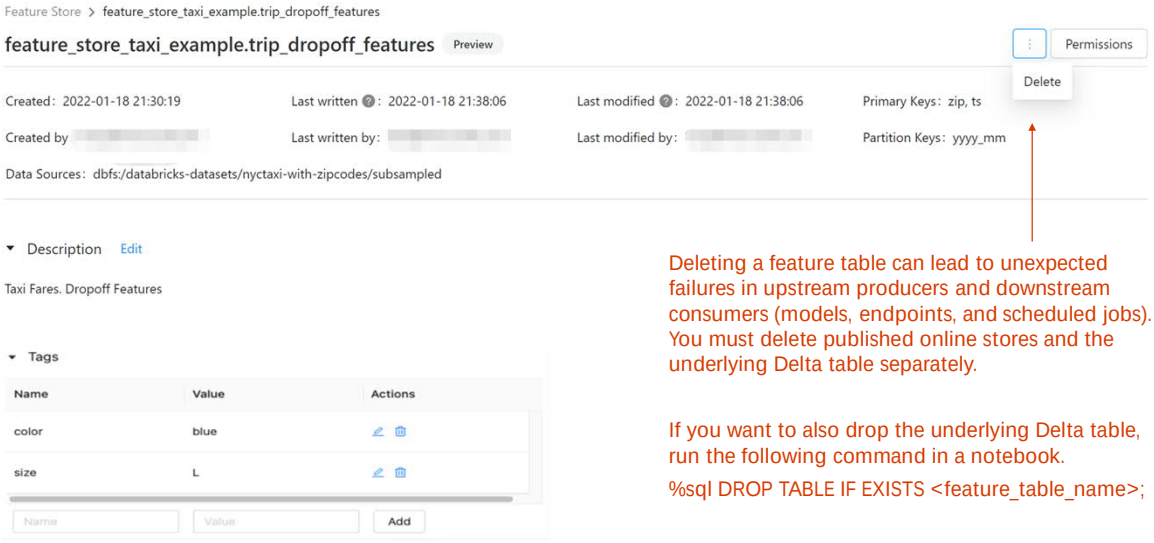
Add a tag to a feature table



Tags are key-value pairs that you can create and use to search for feature tables.

Use the Feature Store UI

Delete a feature table



Deleting a feature table can lead to unexpected failures in upstream producers and downstream consumers (models, endpoints, and scheduled jobs). You must delete published online stores and the underlying Delta table separately.

If you want to also drop the underlying Delta table, run the following command in a notebook.

%sql DROP TABLE IF EXISTS <feature_table_name>;

Control access to feature tables

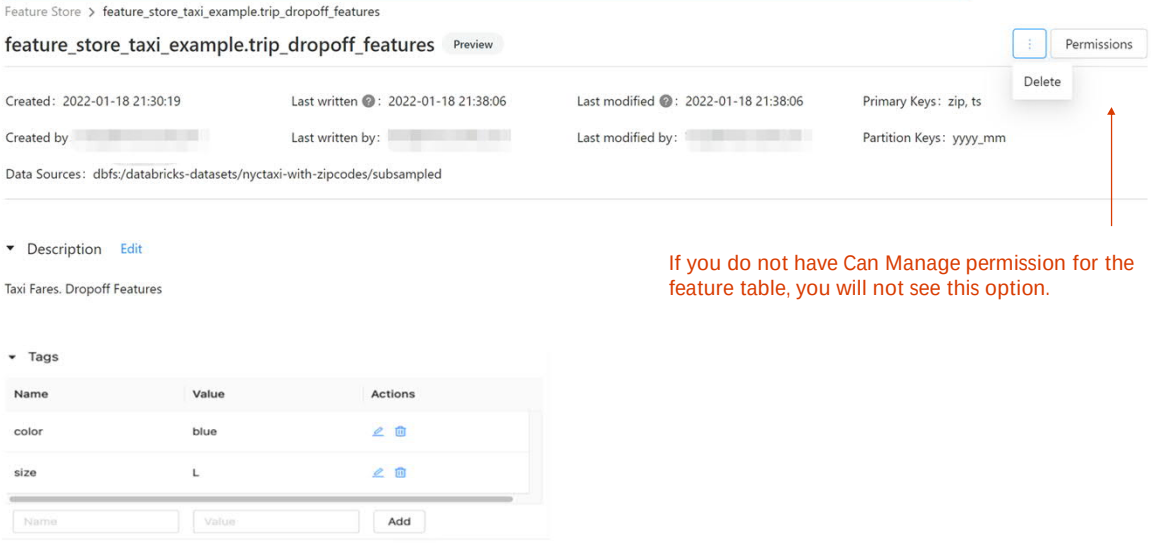
Manage Feature Store permissions

- Feature Store access control does not govern access to the underlying Delta table, which is governed by table access control.
- By default, when a feature table is created:
  - The creator has Can Manage permission
  - Workspace admins have Can Manage permission
  - Other users have No Permissions

Ability	No Permissions	Can View Metadata	Can Edit Metadata	Can Manage
Create feature table	X	X	X	X
Read feature table		X	X	X
Search feature table		X	X	X
Publish feature table to online store		X	X	X
Write features to feature table			X	X
Update description of feature table			X	X
Modify permissions on feature table				X
Delete feature table				X

Control access to feature tables

Manage Feature Store permissions



If you do not have Can Manage permission for the feature table, you will not see this option.

Lesson 2: Experiment Tracking with MLflow

Experiments and Runs

Experiment is the primary unit of organization and access control for MLflow runs belong to this experiment. It can be used for visualization, search and compare runs as well as a repository for artifacts.

Run corresponds to a single execution of model training. Run could contain following information:

- Source: Name of the notebook
- Version: Notebook revision
- Start & end time
- Parameters
- Metrics
- Tags
- Artifacts

Tracking metrics, parameters and models using MLflow

Numerical metrics

- mlflow.log_metric("accuracy",0.9)

Training parameters

- mlflow.log_param("learning_rate", 0.001)

Models

- mlflow.sklearn.log_model(model,"sk_model")

Artifacts

- mlflow.log_artifact("./confusion_matrix.png", "confusion_matrix"

Experiment Tracking

```
with mlflow.start_run(run_name="linear-model") as run:

  mlflow.log_param("trainDataSize", trainDF.count())

  # training
  featureCols = [col for col in trainDF.columns if col != 'km']
  vecAssembler = VectorAssembler(inputCols=featureCols, outputCol="features")
  lr = LinearRegression(featuresCol="features", labelCol="km")
  stages = [vecAssembler, lr]
  pipeline = Pipeline(stages=stages)
  model = pipeline.fit(trainDF)

  # Log model
  mlflow.spark.log_model(model, "linear",
input_example=trainDF.limit(5).toPandas())
```

Experiment Tracking

```
with mlflow.start_run(run_name="linear-model") as run:

  mlflow.log_param("trainDataSize", trainDF.count())

  # training
  featureCols = [col for col in trainDF.columns if col != 'km']
  vecAssembler = VectorAssembler(inputCols=featureCols, outputCol="features")
  lr = LinearRegression(featuresCol="features", labelCol="km")
  stages = [vecAssembler, lr]
  pipeline = Pipeline(stages=stages)
  model = pipeline.fit(trainDF)

  # Log model
  mlflow.spark.log_model(model, "linear",
input_example=trainDF.limit(5).toPandas())
```

Experiment Tracking

```
with mlflow.start_run(run_name="linear-model") as run:

  mlflow.log_param("trainDataSize", trainDF.count())

  # training
  featureCols = [col for col in trainDF.columns if col != 'km']
  vecAssembler = VectorAssembler(inputCols=featureCols, outputCol="features")
  lr = LinearRegression(featuresCol="features", labelCol="km")
  stages = [vecAssembler, lr]
  pipeline = Pipeline(stages=stages)
  model = pipeline.fit(trainDF)

  # Log model
  mlflow.spark.log_model(model,"linear“,input_example=trainDF.limit(5).toPandas())
```

Experiment Tracking – Train Model

```
with mlflow.start_run(run_name="linear-model") as run:

  mlflow.log_param("trainDataSize", trainDF.count())

  # training
  featureCols = [col for col in trainDF.columns if col != 'km']
  vecAssembler = VectorAssembler(inputCols=featureCols, outputCol="features")
  lr = LinearRegression(featuresCol="features", labelCol="km")
  stages = [vecAssembler, lr]
  pipeline = Pipeline(stages=stages)
  model = pipeline.fit(trainDF)

  # Log model
  mlflow.spark.log_model(model, "linear", input_example=trainDF.limit(5).toPandas())
```

Experiment Tracking

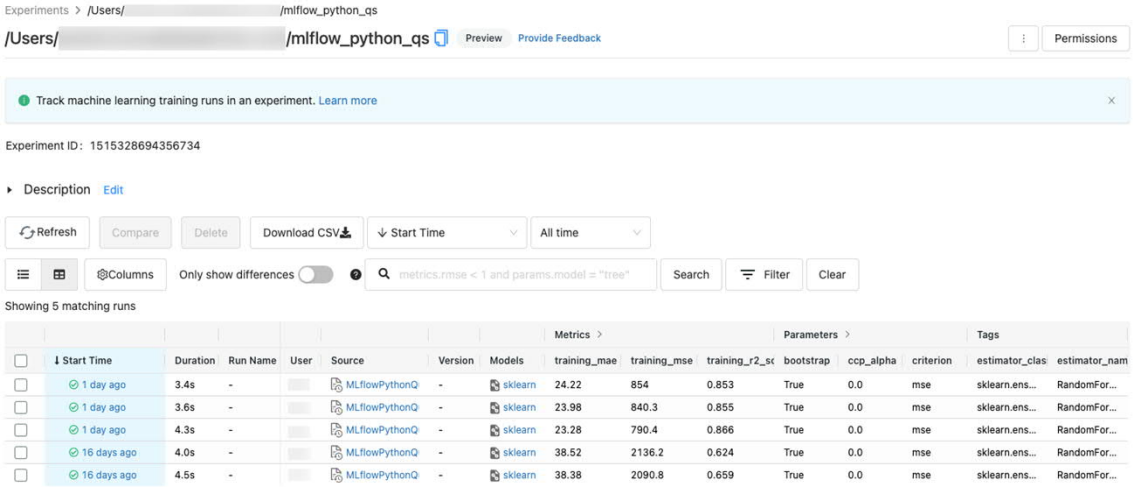
```
with mlflow.start_run(run_name="linear-model") as run:

  mlflow.log_param("trainDataSize", trainDF.count())

  # training
  featureCols = [col for col in trainDF.columns if col != 'km']
  vecAssembler = VectorAssembler(inputCols=featureCols, outputCol="features")
  lr = LinearRegression(featuresCol="features", labelCol="km")
  stages = [vecAssembler, lr]
  pipeline = Pipeline(stages=stages)
  model = pipeline.fit(trainDF)

  # Log model
  mlflow.spark.log_model(model, "linear",input_example=trainDF.limit(5).toPandas())
```

Experiments



Automatically log training runs with MLflow

- Mlflow provides **`mlflow.<framework>.autolog()`** to automatically log training code written in popular ML frameworks

```
# Also autoinstruments tf.keras
import mlflow.tensorflow
mlflow.tensorflow.autolog()
```