# Feature Store

28 July 2023 21:27

A feature store is a centralized repository that enables data scientists to find and share features and also ensures that the same code used to compute the feature values is used for model training and inference.

Machine learning uses existing data to build a model to predict future outcomes. In almost all cases, the raw data requires preprocessing and transformation before it can be used to build a model. This process is called feature engineering, and the outputs of this process are called features - the building blocks of the model.

Developing features is complex and time-consuming. An additional complication is that for machine learning, feature calculations need to be done for model training, and then again when the model is used to make predictions. These implementations may not be done by the same team or using the same code environment, which can lead to delays and errors. Also, different teams in an organization will often have similar feature needs but may not be aware of work that other teams have done. A feature store is designed to address these problems.

Source: From <https://docs.databricks.com/machine-learning/feature-store/index.html>

The typical machine learning workflow using Feature Store follows this path:

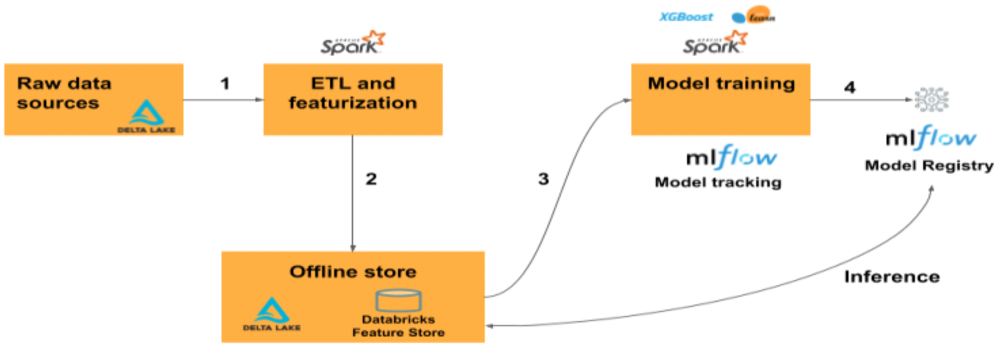
1. Write code to convert raw data into features and create a Spark DataFrame containing the desired features.

1. Write the DataFrame as a feature table in Feature Store. (Refer Here)

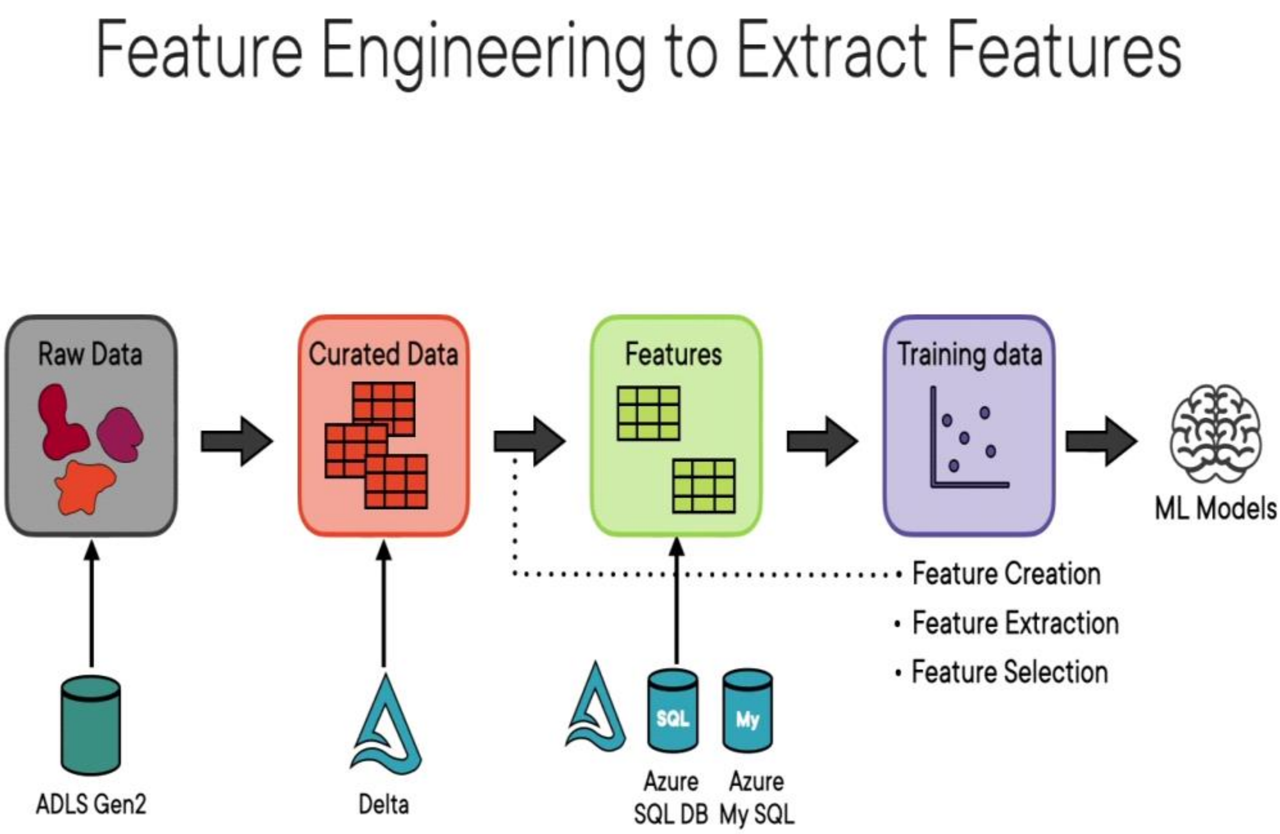
1. Train a model using features from the feature store. When you do this, the model stores the specifications of features used for training. When the model is used for inference, it automatically joins features from the appropriate feature tables.

1. Register model in Model Registry.

**For batch use cases, the model automatically retrieves the features it needs from Feature Store.**




**Below you can see a conventional view of Feature Extraction and their use:**




**Overall the Databricks feature store provide us with below mentioned functionalities:**

# Databricks Feature Store

**Discoverability:** Allows users to browse and search for features

**Lineage:** Data sources used to create feature tables are accessible and recreatable

**Integration:** Feature store integrated with model scoring and serving - model is packaged with feature metadata

**Point-in-time lookups:** Supports time-series and event-based use cases that require point-in-time correctness

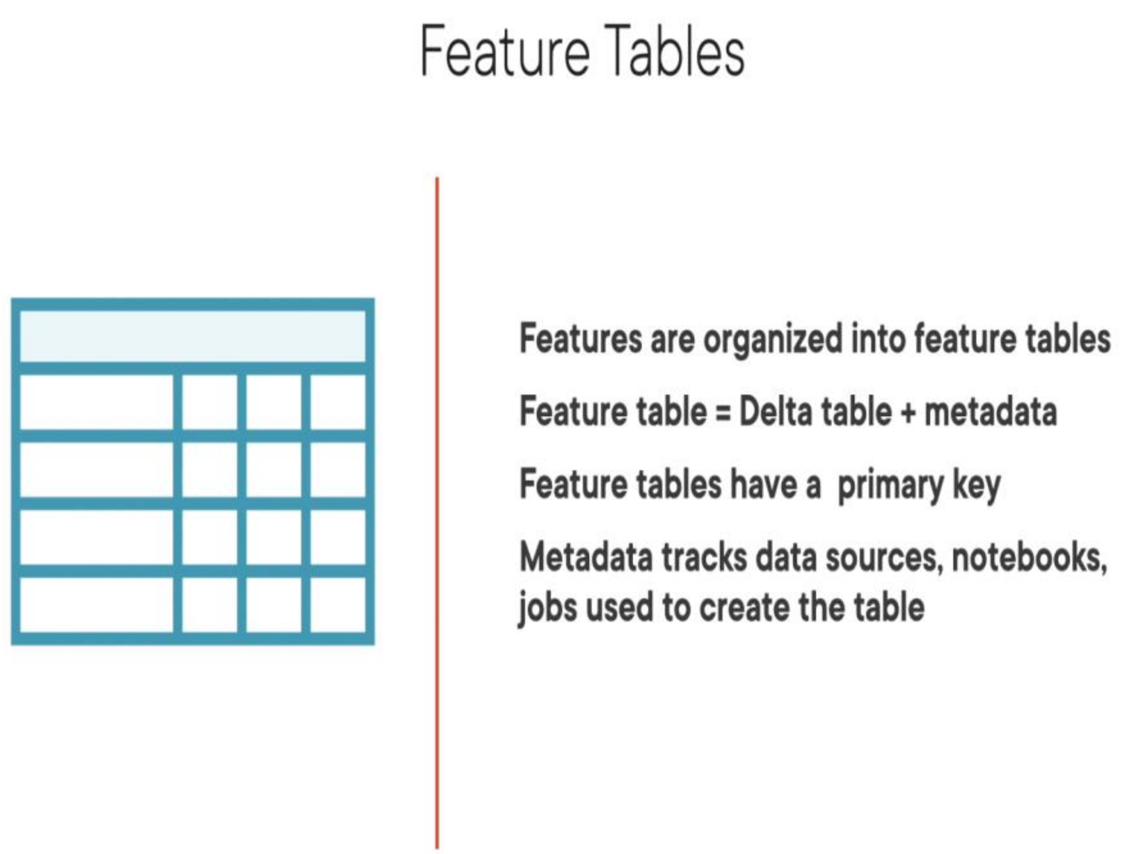
# Feature Store Concept

28 July 2023 22:18

Imagine you're building a house, and you need various materials like bricks, cement, wood, etc. You go to a store where you can find all these materials conveniently organized and ready for use. That store is like a one-stop-shop for construction materials.

In the world of data and machine learning, you often need to build complex models that require different data attributes, called features. These features are like the construction materials for your models. A feature store is like that one-stop-shop for all your machine learning features.

So, in a Databricks feature store, you have a centralized and organized repository where you can store, manage, and share all the features your machine learning projects need. This makes it easy to access and reuse these features across different parts of your data pipelines and machine learning workflows.
Just like the construction material store, a feature store helps you save time and effort by providing a single place to find and use all the necessary features, making it more efficient to build and maintain machine learning models.




A feature store is a centralized repository for storing and managing features. Features are the variables/attributes that are used to train machine learning models.

The image shows the following components of a Databricks Feature Store:

- Features: The features are stored in a table in a Delta Lake.
- Feature definitions: The feature definitions are stored in a JSON file. The feature definitions describe the features, including their name, type, and description.
- Feature groups: Feature groups are a way to organize features. Feature groups can be used to group features that are related to each other.
- Feature lineage: The feature lineage tracks the history of a feature. The feature lineage shows how the feature was created and how it has been used.

The Databricks Feature Store provides a number of benefits, including:

- Centralized storage: The Databricks Feature Store provides a centralized location for storing features. This makes it easy to manage and share features.
- Version control: The Databricks Feature Store provides version control for features. This means that you can track the history of a feature and

revert to a previous version if necessary.

- ** lineage:** The Databricks Feature Store provides feature lineage. This means that you can track the history of a feature and see how it has been used.
- Metadata: The Databricks Feature Store stores metadata about features. This metadata includes the name, type, and description of the feature.

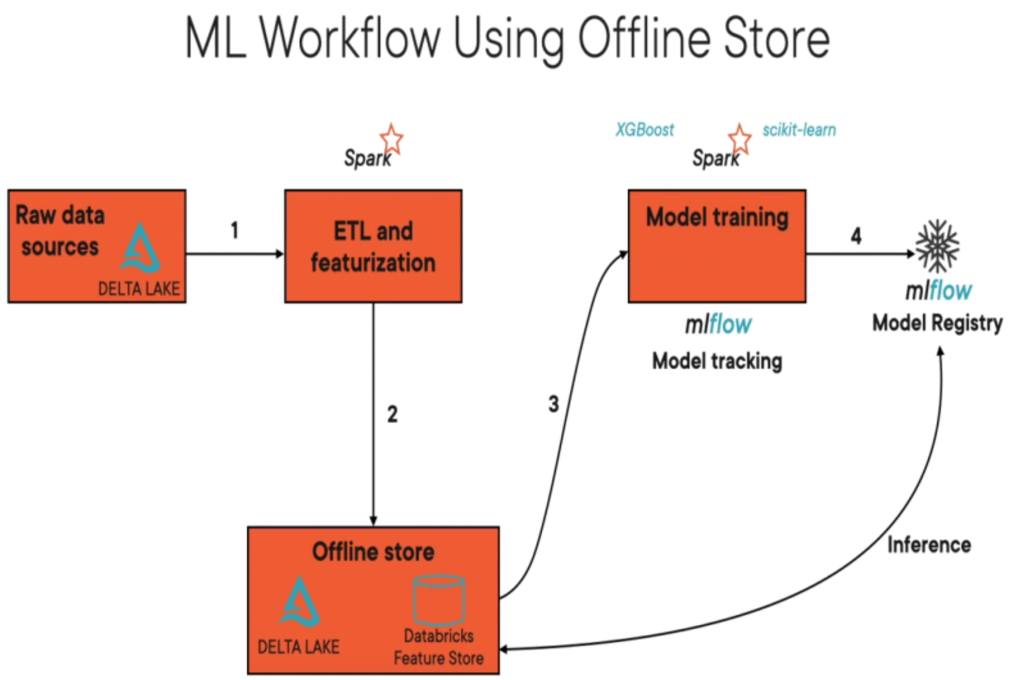
The Databricks Feature Store is a powerful tool for managing and using features. It can help you to improve the performance and accuracy of your machine learning models.

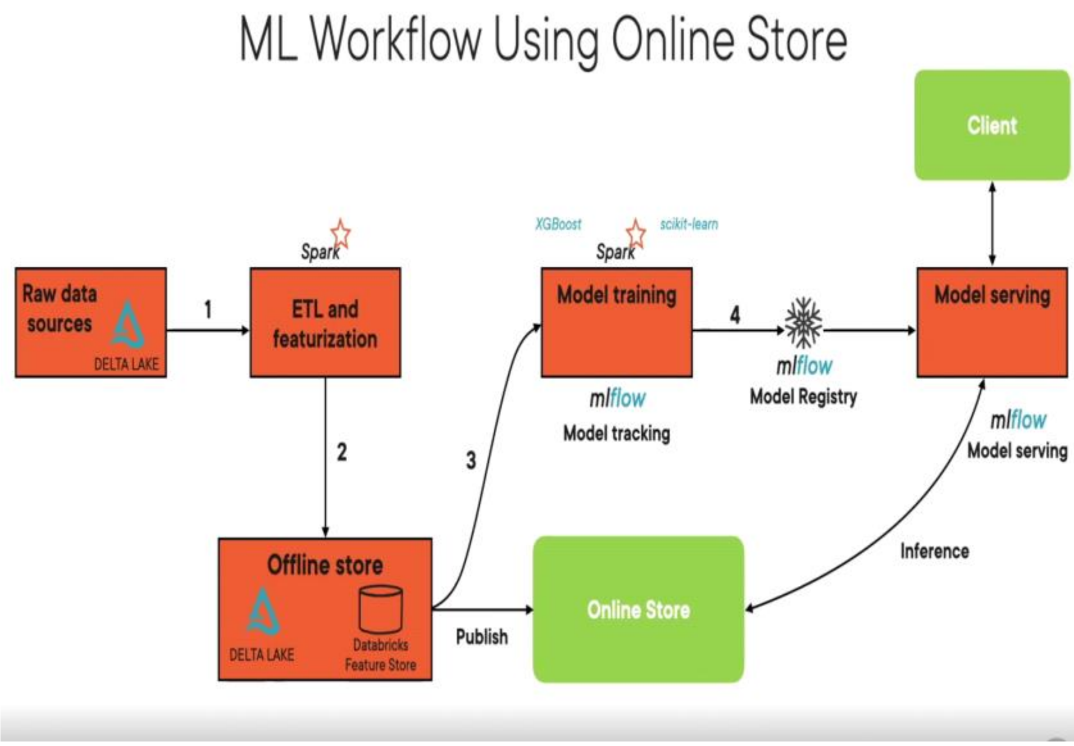
Here are some additional details about the image:

- The features are stored in a table in a Delta Lake. Delta Lake is a version-controlled data lake that provides ACID transactions, schema evolution, and built-in data quality checks.
- The feature definitions are stored in a JSON file. The feature definitions describe the features, including their name, type, and description. The feature definitions are used by the Databricks Feature Store to interpret the features.
- The feature groups are a way to organize features. Feature groups can be used to group features that are related to each other. For example, you could create a feature group for all of the features that are used to predict customer churn.
- The feature lineage tracks the history of a feature. The feature lineage shows how the feature was created and how it has been used. The feature lineage can be used to troubleshoot problems with your machine learning models.

Types of Feature Store:

- Offline Store : Used for feature discovery, model training, and batch inference - materialized as delta tables
- Online Store: Low latency database used for real-time model inference




ML Workflow Using Online Store
Client
Spark
XGBoost
Spark
scikit-learn
Raw data sources
DELTA LAKE
1
ETL and featurization
Model training
4
Model serving
mlflow
Model tracking
mlflow
Model Registry
mlflow
Model serving
2
3
Inference
Offline store
DELTA LAKE
Databricks Feature Store
Publish
Online Store

# Creating and Working with Feature Tables

29 July 2023 17:51

Below I am attaching snippets of different code segment, for feature store creation and working on it using a ML problem on Bike sharing Problem.

General Steps:

- Read in the raw data
- Create processed features using the raw data
- Save them in feature tables
- Access and use the features available in feature table according to need

Below is the detailed view of same, using the ML problem we mentioned above.

**Step 1: Import necessary libraries and load the dataset**

FeatureTables Python

File Edit View Run Help Last edit was 4 minutes ago Give feedback

Run all

Free trial ends in 14 days. Upgrade to Premium in Azure Portal

Cmd 1

```python
import datetime
import numpy as np
import pandas as pd

from databricks import feature_store

from pyspark.sql import *
from pyspark.sql.types import *
from pyspark.sql.functions import *
```

Command took 3.01 seconds -- by cloud.user@loonycorn.com at 12/12/2022, 11:33:50 on cloud user's Cluster

Cmd 2

```python
bike_sharing_data = spark.read.csv('dbfs:/FileStore/datasets/hour.csv',
                                   header = 'true', inferSchema = 'true')

display(bike_sharing_data)
```

Cmd 3

```python
bike_sharing_data = bike_sharing_data.drop(*('casual', 'registered'))

display(bike_sharing_data)
```

**Step 2: General Pre-Processing Step, will vary as per your project need (below is just a simple example)**

Cmd 4

```python
bike_sharing_data = bike_sharing_data.withColumn('instant', bike_sharing_data['instant'].cast(LongType()))
bike_sharing_data = bike_sharing_data.withColumn('hr', bike_sharing_data['hr'].cast(LongType()))
bike_sharing_data = bike_sharing_data.withColumn('cnt', bike_sharing_data['cnt'].cast(LongType()))
```

bike_sharing_data: pyspark.sql.dataframe.DataFrame

```
instant: long
dteday: string
season: string
hr: long
holiday: string
workingday: string
weathersit: string
temp: double
atemp: double
hum: double
windspeed: double
cnt: long
```

Command took 0.17 seconds -- by cloud.user@loonycorn.com at 12/12/2022, 11:36:11 on cloud user's Cluster

Cmd 5

```python
bike_sharing_data.schema
```

Cmd 6

**Step 3: Create Feature Group by selecting features that can fall in one categogy**

**Note: Remember that the Primary Key column should be part of each feature group**

Cmd 6

```
bike_sharing_day_data = bike_sharing_data.select('instant', 'dteday', 'season', 'hr', 'holiday', 'workingday', 'cnt')

bike_sharing_day_data.display()
```

▸ (1) Spark Jobs

▸ bike_sharing_day_data: pyspark.sql.dataframe.DataFrame = [instant: long, dteday: string ... 5 more fields]

Table +

| | instant | dteday | season | hr | holiday | workingday | cnt |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 01/01/11 | winter | 0 | No | No | 16 |
| 2 | 2 | 01/01/11 | winter | 1 | No | No | 40 |
| 3 | 3 | 01/01/11 | winter | 2 | No | No | 32 |
| 4 | 4 | 01/01/11 | winter | 3 | No | No | 13 |
| 5 | 5 | 01/01/11 | winter | 4 | No | No | 1 |
| 6 | 6 | 01/01/11 | winter | 5 | No | No | 1 |

Truncated results, showing first 1,000 rows. | 0.48 seconds runtime

Command took 0.48 seconds -- by cloud.user@loonycorn.com at 12/12/2022, 11:36:47 on cloud user's Cluster

Cmd 7

- **Instant is the Primary key in above example**

Cmd 7

```
bike_sharing_weather_data = bike_sharing_data.select('instant', 'weathersit', 'temp', 'atemp', 'hum', 'windspeed', 'cnt')

bike_sharing_weather_data.display()
```

▸ (1) Spark Jobs

▸ bike_sharing_weather_data: pyspark.sql.dataframe.DataFrame = [instant: long, weathersit: string ... 5 more fields]

Table +

| | instant | weathersit | temp | atemp | hum | windspeed | cnt |
|---|---|---|---|---|---|---|---|
| 1 | 1 | Clear | 0.24 | 0.2879 | 0.81 | 0 | 16 |
| 2 | 2 | Clear | 0.22 | 0.2727 | 0.8 | 0 | 40 |
| 3 | 3 | Clear | 0.22 | 0.2727 | 0.8 | 0 | 32 |
| 4 | 4 | Clear | 0.24 | 0.2879 | 0.75 | 0 | 13 |
| 5 | 5 | Clear | 0.24 | 0.2879 | 0.75 | 0 | 1 |
| 6 | 6 | Cloudy | 0.24 | 0.2576 | 0.75 | 0.0896 | 1 |

Truncated results, showing first 1,000 rows. | 0.42 seconds runtime

Command took 0.42 seconds -- by cloud.user@loonycorn.com at 12/12/2022, 11:37:19 on cloud user's Cluster

**Step 4: Create Feature table**

- **Compute desired feature**
- **The table should have the Primary key (can be single or more columns)**
- **Output need to be a spark dataframe**
- **Save the output in SQL database**

Cmd 8

```
spark.conf.set('spark.sql.execution.arrow.enabled','false')

def create_time_col_features(df, datecol):

    df = df.toPandas()
    df = df.dropna()

    df[datecol] = pd.to_datetime(df[datecol], format = '%d/%m/%y')

    df['year'] = df[datecol].dt.year
    df['month'] = df[datecol].dt.month
    df['dayofweek'] = df[datecol].dt.dayofweek

    df = pd.get_dummies(df, columns = ['dayofweek', 'month', 'year'], drop_first = True)
    df = df.drop(['dteday'], axis = 1)

    df = spark.createDataFrame(df)

    return df
```

**Output of above function**

Cmd 9

```
bike_sharing_time_features = create_time_col_features(df = bike_sharing_day_data, datecol = 'dteday')

display(bike_sharing_time_features)
```

▸ (2) Spark Jobs

▸ bike_sharing_time_features: pyspark.sql.dataframe.DataFrame = [instant: long, season: string ... 22 more fields]

Table

| | instant | season | hr | holiday | workingday | cnt | dayofweek_1 | dayofweek_2 | dayofweek_3 | dayofweek_4 | dayofweek_5 | dayofweek_6 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | winter | 0 | No | No | 16 | 0 | 0 | 0 | 0 | 1 | 0 |
| 2 | 2 | winter | 1 | No | No | 40 | 0 | 0 | 0 | 0 | 1 | 0 |
| 3 | 3 | winter | 2 | No | No | 32 | 0 | 0 | 0 | 0 | 1 | 0 |
| 4 | 4 | winter | 3 | No | No | 13 | 0 | 0 | 0 | 0 | 1 | 0 |
| 5 | 5 | winter | 4 | No | No | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 6 | 6 | winter | 5 | No | No | 1 | 0 | 0 | 0 | 0 | 1 | 0 |

Truncated results, showing first 1,000 rows. | 3.35 seconds runtime

Refreshed now

Cmd 10

```
%sql

CREATE DATABASE IF NOT EXISTS bike_share_features_db;
```

Cmd 11

```
%sql

SHOW DATABASES;
```

**Instantiate Feature table:**

Cmd 12

```
fs = feature_store.FeatureStoreClient()
```

Command took 0.12 seconds -- by cloud.user@loonycorn.com at 12/12/20

Cmd 13

**Store Feature in feature store as per your requirement:**

Cmd 13

```
bike_sharing_time_features_2011 = bike_sharing_time_features.filter(col('year_2012') == 0)

display(bike_sharing_time_features_2011)
```

**CREATE the table:**

Cmd 14

```
1  spark.conf.set('spark.sql.shuffle.partitions', '5')
2
3  fs.create_table(
4      name = 'bike_share_features_db.bike_sharing_time_features',
5      primary_keys = ['instant'],
6      df = bike_sharing_time_features_2011,
7      description = 'Bike sharing count prediction time and day based features',
8  )
9
```

**In above code:**

**Line 3: Fs is our Featurestoreclient which will create the table for us**
**Line 4: The name property sets the name of the table**
**Line 5: Mentions the Primary key our table will have (Can be one or more column)**
**Line 6: Name of dataframe or object which needs to be converted to Feature Store/table**
**Line 7: Description about the table we are creating**

**Table will be created in Feature Store segment and can be navigated to, below is the output how it will look:**

Feature Store ›

## bike_share_features_db.bike_sharing_time_features

Created: 2022-12-12 11:59:01 | Last written: 2022-12-12 11:59:10 | Last modified: 2022-12-12 11:59:10

Primary Keys: instant (LONG) | Created by: cloud.user@loonycorn.com | Last written by: cloud.user@loonycorn.com

Last modified by: cloud.user@loonycorn.com | Partition Keys:

Data Sources:

Details

Description Edit

Bike sharing count prediction time and day based features

Tags (0)

| Name | Value | Actions |
|---|---|---|

No tags found.

Name Value Add

**Step 5: Access the Feature table**

- **If you want access the data and look at its meta data or load the data use below syntax**

Cmd 15

```
1  bike_sharing_time_ft = fs.get_table('bike_share_features_db.bike_sharing_time_features')
2
3  display(bike_sharing_time_ft)
```

(3) Spark Jobs

```
<FeatureTable: keys=['instant'], tags={}>
```

Command took 1.17 seconds -- by cloud.user@loonycorn.com at 12/12/2022, 12:00:35 on cloud user's Cluster

Cmd 17

```
1  bike_sharing_time_features_read = fs.read_table('bike_share_features_db.bike_sharing_time_features')
2
3  display(bike_sharing_time_features_read)
```

(4) Spark Jobs

bike_sharing_time_features_read: pyspark.sql.dataframe.DataFrame = [instant: long, season: string ... 22 more fields]

Table +

| | instant | season | hr | holiday | workingday | cnt | dayofweek_1 | dayofweek_2 | dayofweek_3 | dayofweek_4 | dayofweek_5 | dayofweek_6 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | winter | 0 | No | No | 16 | 0 | 0 | 0 | 0 | 1 | 0 |
| 2 | 2 | winter | 1 | No | No | 40 | 0 | 0 | 0 | 0 | 1 | 0 |
| 3 | 3 | winter | 2 | No | No | 32 | 0 | 0 | 0 | 0 | 1 | 0 |
| 4 | 4 | winter | 3 | No | No | 13 | 0 | 0 | 0 | 0 | 1 | 0 |
| 5 | 5 | winter | 4 | No | No | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 6 | 6 | winter | 5 | No | No | 1 | 0 | 0 | 0 | 0 | 1 | 0 |

Truncated results, showing first 1,000 rows. | 5.51 seconds runtime Refreshed now

Command took 5.51 seconds -- by cloud.user@loonycorn.com at 12/12/2022, 12:01:56 on cloud user's Cluster

**As the table created is a delta lake table you can also access and play with the data use SQL commands.**

# Feature table as Delta Table

29 July 2023 20:40

- The Feature table are stored in databricks as a delta table in parquet format and can be operated using SQL commands.

In Databricks, Feature tables are typically stored as Delta tables in Parquet format, which allows for efficient storage, versioning, and querying of the data. Delta Lake is an open-source storage layer that provides ACID transactions on top of Apache Spark, making it well-suited for managing feature data.

- Delta tables are a type of table in Databricks that are based on the Delta Lake storage format. Delta Lake is a version-controlled data lake that provides ACID transactions, schema evolution, and built-in data quality checks.
- Parquet is a columnar storage format that is optimized for efficient data access. Parquet files are typically much smaller than CSV or JSON files, and they can be read much faster.
- SQL commands are used to interact with data in Databricks. SQL is a standard language for querying and manipulating data, and it is supported by most data warehouses and data lakes.

  Here are some examples of SQL commands that can be used to operate on feature tables in Databricks:

- CREATE TABLE: This command is used to create a new feature table.
- INSERT INTO: This command is used to insert data into a feature table.
- SELECT: This command is used to select data from a feature table.
- UPDATE: This command is used to update data in a feature table.
- DELETE: This command is used to delete data from a feature table.

Python

```
%fs

ls "dbfs:/user/hive/warehouse/bike_share_features_db.db/bike_sharing_time_features"
```

Table +

| | path | name | size |
|---|---|---|---|
| 1 | dbfs:/user/hive/warehouse/bike_share_features_db.db/bike_sharing_time_features/_delta_log/ | _delta_log/ | 0 |
| 2 | dbfs:/user/hive/warehouse/bike_share_features_db.db/bike_sharing_time_features/part-00000-b38c70d3-e931-47a7-a41b-0229795d8d48-c000.snappy.parquet | part-00000-b38c70d3-e931-47a7-a41b-0229795d8d48-c000.snappy.parquet | 3257[illegible] |
| 3 | dbfs:/user/hive/warehouse/bike_share_features_db.db/bike_sharing_time_features/part-00001-c02709fb-b51b-4e2b-9e4c-55597215985f-c000.snappy.parquet | part-00001-c02709fb-b51b-4e2b-9e4c-55597215985f-c000.snappy.parquet | 3225[illegible] |
| 4 | dbfs:/user/hive/warehouse/bike_share_features_db.db/bike_sharing_time_features/part-00002-15f66294-da8d-42d7-b0bf-b619cb8406d4-c000.snappy.parquet | part-00002-15f66294-da8d-42d7-b0bf-b619cb8406d4-c000.snappy.parquet | 1095[illegible] |

Showing all 4 rows. | 6.26 seconds runtime

Refreshed now

Command took 6.26 seconds -- by cloud.user@loonycorn.com at 12/12/2022, 12:24:36 on cloud user's Cluster

Cmd 23

```
%sql

DESCRIBE HISTORY bike_share_features_db.bike_sharing_time_features
```

# Updating a Feature Table

29 July 2023 20:48

There are two ways to update a Feature table in Databricks Feature Store:

- Append: This method adds new data to the end of the feature table.
- Merge: This method merges new data into the feature table, overwriting any existing data that has the same keys.

To append data to a Feature table, you can use the following SQL command:

```
INSERT INTO feature_table
SELECT *
FROM new_data_table;
```

To merge data into a Feature table, you can use the following SQL command:

```
MERGE INTO feature_table
USING new_data_table
ON feature_table.key = new_data_table.key
WHEN MATCHED THEN
UPDATE SET * = new_data_table.*
WHEN NOT MATCHED THEN
INSERT *;
```

The MERGE statement is a powerful tool that can be used to update Feature tables in a variety of ways. For example, you can use the MERGE statement to update data based on a condition, or to update data in a specific column.

Here are some additional details about appending and merging data to Feature tables:

- Appending data: When you append data to a Feature table, the new data is added to the end of the table. The existing data in the table is not affected.
- Merging data: When you merge data into a Feature table, the new data is merged with the existing data in the table. Any existing data that has the same keys as the new data is overwritten.
- Conditional updates: The MERGE statement can be used to update data based on a condition. For example, you could use the MERGE statement to update the price of a product if the product's price has changed.
- Column updates: The MERGE statement can be used to update data in a specific column. For example, you could use the MERGE statement to update the name of a customer if the customer's name has changed.

Below is an Example for Same:

Cmd 24

```
bike_sharing_time_features_2012 = bike_sharing_time_features.filter(col('year_2012') == 1)

bike_sharing_time_features_2012.display()
```

▸ (2) Spark Jobs

▸ bike_sharing_time_features_2012: pyspark.sql.dataframe.DataFrame = [instant: long, season: string ... 22 more fields]

Table

| | dayofweek_6 | month_2 | month_3 | month_4 | month_5 | month_6 | month_7 | month_8 | month_9 | month_10 | month_11 | month_12 | year_2012 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 3 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 4 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 5 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 6 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

Truncated results, showing first 1,000 rows. | 0.26 seconds runtime

Refreshed now

Command took 0.26 seconds -- by cloud.user@loonycorn.com at 12/12/2022, 12:25:33 on cloud user's Cluster

Cmd 25

```
fs.write_table(
  name = 'bike_share_features_db.bike_sharing_time_features',
  df = bike_sharing_time_features_2012,
  mode = 'overwrite',
)
```

***Write mode is overwrite.

When you give the mode as 'overwrite' or 'append' while writing data to a Delta table in Databricks, it determines how the operation will handle any existing data in the target table.

1. 'overwrite' mode: If you use 'overwrite' mode, it will completely replace the data in the target Delta table with the new data. This means that any existing data in the table will be deleted, and the table will be populated only with the new data.
2. 'append' mode: If you use 'append' mode, it will add the new data to the existing data in the target Delta table. This means that the new data will be appended to the end of the table, keeping the existing data intact.

Cmd 28

```
fs.write_table(
  name = 'bike_share_features_db.bike_sharing_time_features',
  df = bike_sharing_time_features_2011,
  mode = 'merge',
)
```

Cmd 29

```
%sql

SELECT year_2012, count(*) FROM bike_share_features_db.bike_sharing_time_features GROUP BY year_2012
```

***Write mode is merge.

# Partition Delta Table in Feature Store

29 July 2023 21:21

In Databricks Feature Store, the partition_column is an important property that helps improve the performance and efficiency of data storage and retrieval. It is used to partition the data in the underlying storage layer (e.g., Delta Lake) based on the values of the specified column. This concept is often referred to as partitioning.

When you create a Feature table in the Feature Store and specify a partition column, Databricks will organize the data into separate physical directories or files based on the unique values of that column. Each unique value of the partition column will form a separate data partition. This partitioning process is done transparently and automatically by Databricks in the background.

Let's look at an example to understand how the partition_column property works in the context of a Feature table:

```
from pyspark.sql import SparkSession

# Create a SparkSession
spark = SparkSession.builder.appName("PartitioningExample").getOrCreate()

# Sample DataFrame with feature data
feature_data = spark.createDataFrame([
    (1, "Male", 30),
    (2, "Female", 28),
    (3, "Male", 35),
    (4, "Female", 33)
], ["id", "gender", "age"])

# Write data to the Delta table with 'partition_column' specified
feature_data.write.format("delta").partitionBy("gender").save("/delta-table-path")
```

In this example, we create a Feature table with a partition column named "gender." The data will be organized into separate partitions based on the unique values of the "gender" column (i.e., "Male" and "Female"). Each unique value forms a separate physical directory in the Delta Lake storage.

Partitioning has several advantages:

Performance Optimization: When you query the Feature table and filter data based on the partition column, Databricks will only read the relevant partitions, leading to faster query performance. This reduces the amount of data scanned and improves query efficiency.

Data Organization: Partitioning allows for efficient organization and storage of data. It makes it easier to manage and work with large datasets as it provides a more structured layout.

Data Pruning: Partitioning helps with data pruning during query execution. If you have a WHERE clause that filters on the partition column, Databricks can skip reading unnecessary partitions, further improving query performance.

Scalability: With partitioning, data can be distributed across multiple nodes in a distributed computing environment. This enables better parallel processing, which is crucial for handling large-scale data.

Keep in mind that choosing an appropriate partition column is essential. It should be a column that is commonly used in your query predicates and exhibits high cardinality (a large number of distinct values). A column with low cardinality may not provide significant performance benefits from partitioning. It's also essential to avoid having too many partitions, as this can lead to excessive storage overhead and negatively impact performance. Finding the right balance is crucial for efficient data storage and query performance in the Feature Store.

Below is an example for same:

Cmd 31

```
fs.create_table(
    name = 'bike_share_features_db.bike_sharing_features',
    primary_keys = ['instant'],
    partition_columns = ['year_2012'],
    df = bike_sharing_time_features,
    tags = {"time": "true", "day": "true"},
    description = 'Bike sharing count prediction for all years partitioned on year',
)
```

Cmd 33

```
%fs

ls "dbfs:/user/hive/warehouse/bike_share_features_db.db/bike_sharing_features"
```

Table

| | path | name | size | modificationTime |
|---|---|---|---|---|
| 1 | dbfs:/user/hive/warehouse/bike_share_features_db.db/bike_sharing_features/_delta_log/ | _delta_log/ | 0 | 1670828369000 |
| 2 | dbfs:/user/hive/warehouse/bike_share_features_db.db/bike_sharing_features/year_2012=0/ | year_2012=0/ | 0 | 1670828367000 |
| 3 | dbfs:/user/hive/warehouse/bike_share_features_db.db/bike_sharing_features/year_2012=1/ | year_2012=1/ | 0 | 1670828367000 |

Showing all 3 rows. | 0.42 seconds runtime

Cmd 34

Python

```
%fs

ls "dbfs:/user/hive/warehouse/bike_share_features_db.db/bike_sharing_features/year_2012=0"
```

Table

| | path | name | size |
|---|---|---|---|
| 1 | dbfs:/user/hive/warehouse/bike_share_features_db.db/bike_sharing_features/year_2012=0/part-00000-5696eb0c-0e48-42f4-82f6-812bc3d8c2e2.c000.snappy.parquet | part-00000-5696eb0c-0e48-42f4-82f6-812bc3d8c2e2.c000.snappy.parquet | 32271 |
| 2 | dbfs:/user/hive/warehouse/bike_share_features_db.db/bike_sharing_features/year_2012=0/part-00001-7e668809-a8e0-4719-8236-d1109faba933.c000.snappy.parquet | part-00001-7e668809-a8e0-4719-8236-d1109faba933.c000.snappy.parquet | 31945 |
| 3 | dbfs:/user/hive/warehouse/bike_share_features_db.db/bike_sharing_features/year_2012=0/part-00001-fecbb469-4900-4e96-b37b-9d1f1e2e066c.c000.snappy.parquet | part-00001-fecbb469-4900-4e96-b37b-9d1f1e2e066c.c000.snappy.parquet | 57700 |
| 4 | dbfs:/user/hive/warehouse/bike_share_features_db.db/bike_sharing_features/year_2012=0/part-00002-1f8089d3-8acd-4afb-a7ae-96de2e9abf15.c000.snappy.parquet | part-00002-1f8089d3-8acd-4afb-a7ae-96de2e9abf15.c000.snappy.parquet | 10655 |

Showing all 4 rows. | 0.30 seconds runtime

Refreshed now

# Adding Features to Existing Feature Table

29 July 2023 21:27

To add new features to an existing Feature Store table in Databricks, you can follow these steps:

Create a new DataFrame containing the new features: First, you need to create a new DataFrame that contains the additional features you want to add to the existing Feature Store table. This DataFrame should have the same schema as the existing Feature Store table, with the new features included.

Write the new DataFrame to the Feature Store table using the 'append' mode: Once you have the new DataFrame ready, you can write it to the existing Feature Store table using the 'append' mode. The 'append' mode ensures that the new data is added to the existing table without overwriting any existing data.

Here's a code example demonstrating how to add new features to an existing Feature Store table in Databricks:

```
from pyspark.sql import SparkSession

# Create a SparkSession
spark = SparkSession.builder.appName("AddNewFeatures").getOrCreate()

# Sample DataFrame with new features to be added
new_features_data = spark.createDataFrame([
    (1, "Male", 30, "Engineer"),
    (2, "Female", 28, "Doctor"),
    (3, "Male", 35, "Data Scientist"),
    (4, "Female", 33, "Teacher")
], ["id", "gender", "age", "occupation"])

# Assuming you already have an existing Feature Store table named 'feature_table'
# Write the new features DataFrame to the existing Feature Store table using 'append' mode
new_features_data.write.format("delta").mode("append").save("/delta-table-path")
```

In this example, we have created a DataFrame called new_features_data containing the new features we want to add. We then use the write method with the 'append' mode to add the new features to the existing Feature Store table located at "/delta-table-path".

It's important to ensure that the schema of the new_features_data DataFrame matches the schema of the existing Feature Store table. The column names, data types, and order of columns must be consistent for the 'append' mode to work correctly.

By following these steps, you can seamlessly add new features to an existing Feature Store table in Databricks, making your data more comprehensive and supporting more advanced machine learning models and analyses.

**Below is the example code for same:**

The First way is to update the existing feature using computation

Cmd 36

```
def create_time_col_features_and_categorical_encode(df, datecol, categoricalcols):

    df = df.toPandas()
    df = df.dropna()

    df[datecol] = pd.to_datetime(df[datecol], format = '%d/%m/%y')

    df['year'] = df[datecol].dt.year
    df['month'] = df[datecol].dt.month
    df['dayofweek'] = df[datecol].dt.dayofweek

    df = pd.get_dummies(df, columns = ['dayofweek', 'month', 'year'], drop_first = True)
    df = pd.get_dummies(df, columns = categoricalcols, drop_first = True)

    df = df.drop(['dteday'], axis = 1)

    df = spark.createDataFrame(df)

    return df
```

Cmd 37

```
bike_sharing_features = create_time_col_features_and_categorical_encode(df = bike_sharing_day_data,
                                                                        datecol = 'dteday',
                                                                        categoricalcols = ['season', 'hr', 'holiday',  'workingday'])

display(bike_sharing_features)
```

Cmd 38

```python
fs.write_table(
  name = 'bike_share_features_db.bike_sharing_features',
  df = bike_sharing_features,
  mode = 'merge',
)
```

Cmd 39

```sql
%sql

SELECT count(*) FROM bike_share_features_db.bike_sharing_features
```

Cmd 40

```python
bike_sharing_weather_data.display()
```

Cmd 41

```sql
%sql

SELECT count(*) from bike_share_features_db.bike_sharing_time_features
```

The second way is to create a new dataframe with only the new features and merge it with the feature store table

Cmd 40

```python
bike_sharing_weather_data.display()
```

▸ (1) Spark Jobs

Table ⌄ +

| | instant | weathersit | temp | atemp | hum | windspeed | cnt |
|---|---|---|---|---|---|---|---|
| 1 | 1 | Clear | 0.24 | 0.2879 | 0.81 | 0 | 16 |
| 2 | 2 | Clear | 0.22 | 0.2727 | 0.8 | 0 | 40 |
| 3 | 3 | Clear | 0.22 | 0.2727 | 0.8 | 0 | 32 |
| 4 | 4 | Clear | 0.24 | 0.2879 | 0.75 | 0 | 13 |
| 5 | 5 | Clear | 0.24 | 0.2879 | 0.75 | 0 | 1 |
| 6 | 6 | Cloudy | 0.24 | 0.2576 | 0.75 | 0.0896 | 1 |
| 7 | 7 | Clear | 0.22 | 0.2727 | 0.8 | 0 | [illegible] |

Truncated results, showing first 1,000 rows. ⌄ | 0.29 seconds runtime

Command took 0.29 seconds -- by cloud.user@loonycorn.com at 12/12/2022, 12:39:18 on cloud user's Cluster

Cmd 42

```python
def create_weather_features(df):

    df = df.toPandas()
    df = df.dropna()

    df = pd.get_dummies(df, columns = ['weathersit'], drop_first = True)

    df = spark.createDataFrame(df)

    return df
```

Command took 0.12 seconds -- by cloud.user@loonycorn.com at 12/12/2022, 13:06:26 on cloud user's Cluster

Cmd 43

```python
bike_sharing_weather_features = create_weather_features(bike_sharing_weather_data)

display(bike_sharing_weather_features)
```

Cmd 44

```python
fs.write_table(
    name = 'bike_share_features_db.bike_sharing_time_features',
    df = bike_sharing_weather_features,
)
```

In all this the very important point to remember is that it should have the Primary Key common in both the table.

# Feature Tag in Feature Store

29 July 2023 22:37

The set_feature_table_tag function is a feature of Databricks Feature Store that allows you to set tags on Feature tables. Tags are a way to categorize Feature tables and to track their lineage.

To set a tag on a Feature table, you can use the following SQL command:

```
set_feature_table_tag(
  feature_table_name,
  tag_key,
  tag_value
);
```

For example, the following command would set the tag "version" to the value "1.0" on the Feature table "my_feature_table":

```
set_feature_table_tag(
  "my_feature_table",
  "version",
  "1.0"
);
```

You can also use the set_feature_table_tags function to set multiple tags on a Feature table at the same time. The set_feature_table_tags function takes a list of tag keys and values as input. For example, the following command would set the tags "version" to the value "1.0" and "environment" to the value "production" on the Feature table "my_feature_table":

```
set_feature_table_tags(
  "my_feature_table",
  list("version", "environment"),
  list("1.0", "production")
);
```

The set_feature_table_tag and set_feature_table_tags functions are a powerful way to manage Feature tables in Databricks Feature Store. They allow you to categorize Feature tables and to track their lineage.

Here are some additional details about setting tags on Feature tables:

- Tag keys: Tag keys are unique identifiers for tags. Tag keys must be strings.
- Tag values: Tag values can be any type of data. However, it is recommended that tag values be strings.
- Tag limits: There is a limit of 100 tags per Feature table.
- Tag visibility: Tags are visible to all users who have access to the Feature table.

## Work with feature table tags

Tags are key-value pairs that you can create and use to [search for feature tables](). You can create, edit, and delete tags using the Feature Store UI or the [Feature Store Python API]().

### Work with feature table tags in the UI

Use the Feature Store UI to search for or browse feature tables. To access the UI, in the sidebar, select Machine Learning > Feature Store.

### Add a tag using the Feature Store UI

1. Click

   ▸ Tags

   if it is not already open. The tags table appears.

   ▾ Tags

   | Name | Value | Actions |
   |---|---|---|
   | No tags found. | | |

   Name | Value | Add

2. Click in the Name and Value fields and enter the key and value for your tag.
3. Click Add.

### Edit or delete a tag using the Feature Store UI

To edit or delete an existing tag, use the icons in the Actions column.

▾ Tags

| Name | Value | Actions |
|---|---|---|
| color | blue | |
| size | L | |

Name | Value | Add

## Work with feature table tags using the Feature Store Python API

On clusters running v0.4.1 and above, you can create, edit, and delete tags using the [Feature Store Python API](#).

### Requirements

Feature Store client v0.4.1 and above

### Create feature table with tag using the Feature Store Python API

PythonCopy

```python
from databricks.feature_store import FeatureStoreClient
fs = FeatureStoreClient()
customer_feature_table = fs.create_table(
  ...
  tags={"tag_key_1": "tag_value_1", "tag_key_2": "tag_value_2", ...},
  ...
)
```

### Add, update, and delete tags using the Feature Store Python API

PythonCopy

```python
from databricks.feature_store import FeatureStoreClient
fs = FeatureStoreClient()
# Upsert a tag
fs.set_feature_table_tag(table_name="my_table", key="quality", value="gold")
# Delete a tag
fs.delete_feature_table_tag(table_name="my_table", key="quality")
```

Cmd 46

```
fs.set_feature_table_tag(table_name="bike_share_features_db.bike_sharing_time_features", key="env", value="production")

fs.delete_feature_table_tag(table_name="bike_share_features_db.bike_sharing_time_features", key="weather")
```

Cmd 47

# Streaming Features in Feature Table

29 July 2023 22:43

Step 1: Create a Feature Table to be populated with Streaming features (first create a schema)

Cmd 47

```
table_schema = StructType([
                    StructField('instant', LongType(), True),
                    StructField('season', StringType(), True),
                    StructField('hr', LongType(), True),
                    StructField('holiday', StringType(), True),
                    StructField('weathersit', StringType(), True),
                    StructField('temp', DoubleType(), True),
                    StructField('windspeed',DoubleType(), True),
                    StructField('cnt', LongType(), True)
                  ])
```

Command took 0.11 seconds -- by cloud.user@loonycorn.com at 12/12/2022, 13:23:37 on cloud user's Cluster

Cmd 48

```
fs.create_table(
    name = 'bike_share_features_db.bike_sharing_stream_features',
    primary_keys = ['instant'],
    schema = table_schema,
    description = 'Bike sharing stream features',
)
```

(9) Spark Jobs

2022/12/12 07:53:57 INFO databricks.feature_store._compute_client._compute_client: Created feature table 'bike_share_features_db.bike_sharing_stream_features'.

Out[47]: <FeatureTable: keys=['instant'], tags={}>

Command took 4.21 seconds -- by cloud.user@loonycorn.com at 12/12/2022, 13:23:54 on cloud user's Cluster

Cmd 49

Step 2: Create a streaming ingest pipeline to load data into the Delta Lake table. (Use tool like Kafka)

Cmd 49

```
dbutils.fs.mkdirs('dbfs:/FileStore/datasets/bike_share_data_streaming')
```

Cmd 50

Step 3: Create a schema for raw data | To be available from a streaming app/site

```
Cmd 50

schema = StructType([
                     StructField('instant', LongType(), True),
                     StructField('dteday', StringType(), True),
                     StructField('season', StringType(), True),
                     StructField('hr', LongType(), True),
                     StructField('holiday', StringType(), True),
                     StructField('workingday', StringType(), True),
                     StructField('weathersit', StringType(), True),
                     StructField('temp', DoubleType(), True),
                     StructField('atemp', DoubleType(), True),
                     StructField('hum', DoubleType(), True),
                     StructField('windspeed',DoubleType(), True),
                     StructField('casual', LongType(), True),
                     StructField('registered', LongType(), True),
                     StructField('cnt', LongType(), True)
                    ])

Command took 0.10 seconds -- by cloud.user@loonycorn.com at 12/12/2022, 13:24:32 on cloud user's Cluster
```

Step 4: Use Spark.stream service to constantly read the online streaming data

```
Cmd 51

bike_share_data_stream = spark.readStream \
                         .format('csv') \
                         .option('header', True) \
                         .schema(schema) \
                         .load('dbfs:/FileStore/datasets/bike_share_data_streaming')

bike_share_data_stream.display()

Cancel    Running command...
display_query_1 (id: e0ce96c7-1cd8-40cf-8ae4-4099a2c414e2)    Last updated: 10 seconds ago

Query returned no results
```

```
Cmd 52

bike_share_data_stream.isStreaming

Out[51]: True
Command took 0.09 seconds -- by cloud.user@loonycorn.com at 12/12/2022, 13:25:38 on cloud user's Cluster
```

Step 5: Compute Features from the Streaming Dataframe

Cmd 53

```python
def compute_features(df):
    return df.select('instant', 'season', 'hr', 'holiday', 'weathersit', 'temp', 'windspeed')
```

Cmd 54

```python
bike_sharing_stream_features = compute_features(bike_share_data_stream)

bike_sharing_stream_features.display()
```

Cmd 55

```python
fs.write_table(
  df = bike_sharing_stream_features,
  name = 'bike_share_features_db.bike_sharing_stream_features',
  mode = 'merge'
)
```

Cmd 56

```sql
%sql

SELECT count(*) from bike_share_features_db.bike_sharing_stream_features
```

# Training Models and Performing Inference with Feature Tables

29 July 2023 23:02

## MODEL TRAIN

Training Datasets:

- Trainings sets allow us to use the Features from Feature Store to train models
- Define the features needed for the model
- Also specify how those features should be joined with external data
- Model trained using training sets retain a reference to the original features

Feature store training sets are a type of training set that is stored in Databricks Feature Store. Feature store training sets retain a reference to the original features, which allows you to track the lineage of the features and to ensure that the features are consistent across different machine learning models.

To create a feature store training set:

1. Create a Databricks Feature Store.
2. Create a Delta Lake table to store the features.
3. Create a feature view to expose the features to machine learning models.
4. Create a training set by using the CREATE TRAINING SET command.

To use a feature store training set:

1. Create a machine learning model.
2. Specify the training set when you train the model.
3. Use the model to make predictions.

Here are some additional details about training sets:

- Training sets are a way to organize features: Training sets can be used to organize features by their purpose, such as training, validation, or test. Training sets can also be used to organize features by their type, such as categorical or numerical.
- Feature store training sets retain a reference to the original features: This allows you to track the lineage of the features and to ensure that the features are consistent across different machine learning models.
- To create a feature store training set: You can use the CREATE TRAINING SET command to create a feature store training set. The CREATE TRAINING SET command takes a number of arguments, including the name of the training set, the name of the feature view, and the version of the feature view.
- To use a feature store training set: You can use a feature store training set

when you train a machine learning model. When you train a machine learning model, you specify the training set using the SET TRAINING SET command.

## MODEL INFERENCE

- Model inference is the process of using a machine learning model to make predictions on new data.

- Feature store inference is a type of model inference that uses Databricks Feature Store to access the features that are used to train the model.

- To perform model inference using Databricks Feature Store:

  1. Create a Databricks Feature Store.
  2. Create a Delta Lake table to store the features.
  3. Create a feature view to expose the features to machine learning models.
  4. Create a machine learning model.
  5. Specify the feature view when you deploy the model.
  6. Use the model to make predictions.

Here are some additional details about model inference:

- Model inference is the process of using a machine learning model to make predictions on new data: This can be done by passing new data to the model and using the model to generate predictions.
- Feature store inference uses Databricks Feature Store to access the features that are used to train the model: This allows you to keep your features up-to-date with the latest data, and it also makes it easier to manage your features.
- To perform model inference using Databricks Feature Store: You can use the PREDICT command to perform model inference. The PREDICT command takes a number of arguments, including the name of the model, the name of the feature view, and the data to be predicted.

# Training ML Model Using Feature From Feature Store

29 July 2023 23:17

Step 1: Get the feature Engineered data

Cmd 8

```python
bike_sharing_day_features = create_time_col_features_and_categorical_encode(df = bike_sharing_day_data,
                                                                            datecol = 'dteday',
                                                                            categoricalcols = ['season', 'hr', 'holiday',  'workingday'])

display(bike_sharing_day_features)
```

▸ (2) Spark Jobs

▸ bike_sharing_day_features: pyspark.sql.dataframe.DataFrame = [instant: long, dayofweek_1: long ... 45 more fields]

Table

| | instant | dayofweek_1 | dayofweek_2 | dayofweek_3 | dayofweek_4 | dayofweek_5 | dayofweek_6 | month_2 | month_3 | month_4 | month_5 | month_6 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 2 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 3 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 4 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 5 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6 | 6 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

Truncated results, showing first 1,000 rows. | 4.79 seconds runtime   Refreshed now

Step 2: Convert it to a Feature table

Cmd 9

```python
fs = feature_store.FeatureStoreClient()

fs.create_table(
    name = 'bike_share_features_db.bike_sharing_day_features',
    primary_keys = ['instant'],
    df = bike_sharing_day_features,
    description = 'Bike sharing count prediction time and day based features',
)
```

▸ (18) Spark Jobs

```
2022/12/13 07:58:20 INFO databricks.feature_store._compute_client._compute_client: Created feature table 'bike_share_features_db.bike_sharing_day_features'.
Out[9]: <FeatureTable: keys=['instant'], tags={}>
```

Command took 11.42 seconds -- by cloud.user@loonycorn.com at 13/12/2022, 13:28:09 on cloud user's Cluster

Step 3: Create a training set

- For creating a training set we need a Feature Lookup
- It doesn't have the ID column and the Target label
- One can create multiple Feature Lookup to use features from different Feature table for Model training

Cmd 10

```python
from databricks.feature_store import FeatureLookup

feature_lookups = [
    FeatureLookup(
      table_name = 'bike_share_features_db.bike_sharing_day_features',
      feature_names = ['dayofweek_1', 'dayofweek_2', 'dayofweek_3',
                       'dayofweek_4', 'dayofweek_5', 'dayofweek_6',
                       'holiday_Yes', 'workingday_Yes',
                       'hr_1', 'hr_10', 'hr_11', 'hr_12',
                       'hr_13', 'hr_14', 'hr_15', 'hr_16', 'hr_17', 'hr_18',
                       'hr_19', 'hr_2', 'hr_20', 'hr_21', 'hr_22', 'hr_23',
                       'hr_3', 'hr_4', 'hr_5', 'hr_6', 'hr_7', 'hr_8', 'hr_9',
                       'month_10', 'month_11', 'month_12', 'month_2', 'month_3',
                       'month_4', 'month_5', 'month_6', 'month_7', 'month_8', 'month_9',
                       'season_spring', 'season_summer', 'season_winter',
                       'year_2012'],
      lookup_key = ['instant'],
    )
]

feature_lookups
```

```
Out[10]: [<FeatureLookup: feature_name=None, lookup_key=['instant'], output_name=None, table_name='bike_share_features_db.bike_sharing_day_features', timestamp_lookup_key=None>]
```

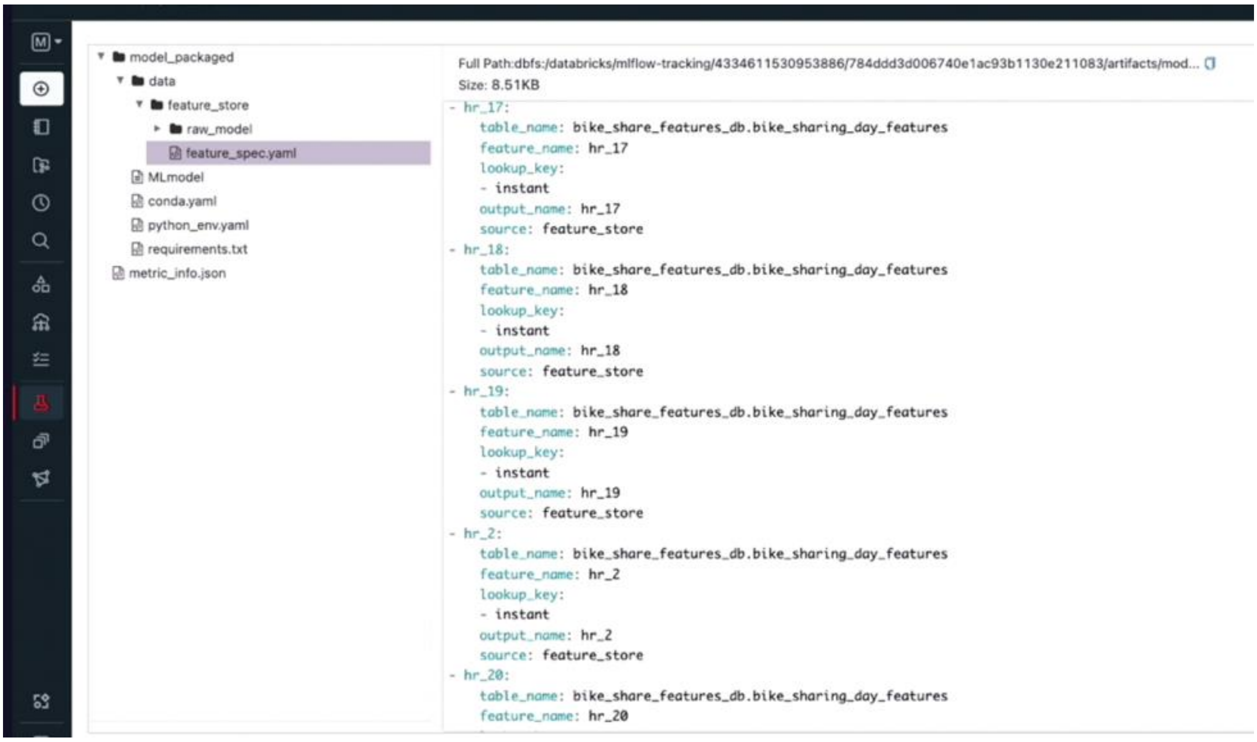
Command took 0.13 seconds -- by cloud.user@loonycorn.com at 13/12/2022, 13:28:33 on cloud user's Cluster

**Step 4: Train the model with help of training set**

```
mlflow.sklearn.autolog(log_models=False)
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error
with mlflow.start_run(run_name = 'day_features_RF'):
    training_set = fs.create_training_set(
                    df = bike_sharing_data.select('instant', 'cnt'),
                    feature_lookups = feature_lookups,
                    label = 'cnt',
                    exclude_columns = ['instant']
    )

    training_df = training_set.load_df()

    train_data = training_df.toPandas()[training_df.columns]

    train, test = train_test_split(train_data, test_size = 0.2, random_state = 123)

    X_train = train.drop(['cnt'], axis = 1)
    X_test = test.drop(['cnt'], axis = 1)

    y_train = train.cnt
    y_test = test.cnt

    model = RandomForestRegressor().fit(X_train, y = y_train.values)
    y_pred =  model.predict(X_test)

    testing_score = r2_score(y_test, y_pred)
    mean_absolute_score = mean_absolute_error(y_test, y_pred)
    mean_sq_error = mean_squared_error(y_test, y_pred)


    fs.log_model(model, artifact_path = 'model_packaged', flavor = mlflow.sklearn,
                 training_set = training_set, registered_model_name = 'bike_share_count_prediction')
```

Step 5: Review the output in Experiment part

- For our purpose, just notice the data folder, which has all the features details




We can also create a training set using Feature store table along with Features from a separate table

- **Assume we have another df with multiple features (can be live features or features from a specific file)**

```
bike_sharing_weather_data = bike_sharing_data.select('instant', 'temp', 'weathersit', 'atemp', 'hum', 'windspeed', 'cnt')

bike_sharing_weather_data.display()
```

- **We will now create a training set, with all the features from feature table along with Features from above dataframe**
    - **Make sure that your df should have the Primary key similar to Primary key of feature store**

TrainingSets Python

File Edit View Run Help Last edit was 22 hours ago Give feedback

Run all

Cmd 14

```python
mlflow.sklearn.autolog(log_models=False)

with mlflow.start_run(run_name = 'day_and_weather_features_RF'):
    training_set = fs.create_training_set(
                    df = bike_sharing_weather_data,
                    feature_lookups = feature_lookups,
                    label = 'cnt',
                    exclude_columns = ['instant', 'weathersit']
    )

    training_df = training_set.load_df()

    train_data = training_df.toPandas()[training_df.columns]

    train, test = train_test_split(train_data, test_size = 0.2, random_state = 123)

    X_train = train.drop(['cnt'], axis = 1)
    X_test = test.drop(['cnt'], axis = 1)

    y_train = train.cnt
    y_test = test.cnt

    model = RandomForestRegressor().fit(X_train, y = y_train.values)
    y_pred =  model.predict(X_test)

    testing_score = r2_score(y_test, y_pred)
    mean_absolute_score = mean_absolute_error(y_test, y_pred)
    mean_sq_error = mean_squared_error(y_test, y_pred)


    fs.log_model(model, artifact_path = 'model_packaged', flavor = mlflow.sklearn,
                 training_set = training_set, registered_model_name = 'bike_share_count_prediction')
```

**We can also create a training set by using 2 or more Feature table**

- **Create a new feature table**

TrainingSets Python

File Edit View Run Help Last edit was 22 hours ago Give feedback

Cmd 19

```python
def create_weather_features(df):

    df = df.toPandas()
    df = df.dropna()

    df = pd.get_dummies(df, columns = ['weathersit'], drop_first = True)

    df = spark.createDataFrame(df)

    return df
```

Cmd 20

```python
bike_sharing_weather_features = create_weather_features(bike_sharing_weather_data)

fs.create_table(
    name = 'bike_share_features_db.bike_sharing_weather_features',
    primary_keys = ['instant'],
    df = bike_sharing_weather_features,
    description = 'Bike sharing count prediction weather based features',
)
```

Cmd 21

```python
bike_sharing_weather_features.columns
```

- **Create Feature lookup with both the tables**

Python

```python
feature_lookups = [
    FeatureLookup(
      table_name = 'bike_share_features_db.bike_sharing_day_features',
      feature_names = None,
      lookup_key = ['instant'],
    ),
    FeatureLookup(
      table_name = 'bike_share_features_db.bike_sharing_weather_features',
      feature_names = ['temp', 'hum', 'windspeed',
                       'weathersit_Cloudy', 'weathersit_Heavy_Rain', 'weathersit_Light_Snow_and_rain'],
      lookup_key = ['instant'],
    )
]

feature_lookups
```

```
Out[22]: [<FeatureLookup: feature_name=None, lookup_key=['instant'], output_name=None, table_name='bike_share_features_db.bike_sharing_day_features', timestamp_lookup_key=None>,
 <FeatureLookup: feature_name=None, lookup_key=['instant'], output_name=None, table_name='bike_share_features_db.bike_sharing_weather_features', timestamp_lookup_key=None>]
```

# Important Links

29 July 2023 17:53

## Work with feature tables

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

# Feature store example notebooks

From <https://docs.databricks.com/machine-learning/feature-store/example-notebooks.html>

# **Databricks Feature Store**

From <https://mlops.community/learn/feature-store/databricks/>

Tags

From <https://docs.hopsworks.ai/feature-store-api/2.5.9/generated/tags/>

# Train models using the Databricks Feature Store

From <https://docs.databricks.com/machine-learning/feature-store/train-models-with-feature-store.html>

**Feature Store taxi example with Point-in-Time Lookup**

From <https://learn.microsoft.com/en-us/azure/databricks/_extras/notebooks/source/machine-learning/feature-store-taxi-example.html#feature-store>