Databricks Feature Store

Python API

Databricks FeatureStoreClient

Module that contains the `FeatureStoreClient` class.

Defines the `FeatureStoreClient` class, which is used to interact with the Databricks Feature Store.

class **`databricks.feature_store.client.FeatureStoreClient`**(*feature_store_uri: Optional[str] = None, model_registry_uri: Optional[str] = None*)

Bases: **`object`**

Client for interacting with the Databricks Feature Store.

`create_feature_table`(*name: str, keys: Union[str, List[str]], features_df: pyspark.sql.dataframe.DataFrame = None, schema: pyspark.sql.types.StructType = None, partition_columns: Union[str, List[str]] = None, description: str = None*) → databricks.feature_store.entities.feature_table.FeatureTable

Create and return a feature table with the given name and primary keys.

The returned feature table has the given name and primary keys. Uses the provided **`schema`** or the inferred schema of the provided **`features_df`**. If **`features_df`** is provided, this data will be saved in a Delta table. Supported data types for features are: **`IntegerType`**, **`LongType`**, **`FloatType`**, **`DoubleType`**, **`StringType`**, **`BooleanType`**, **`DateType`**, and **`TimestampType`**.

Parameters:

- **name** – A feature table name of the form `<database_name>.<table_name>`, for example `dev.user_features`.
- **keys** – The primary keys. If multiple columns are required, specify a list of column names, for example `['customer_id', 'region']`.
- **features_df** – Data to insert into this feature table. The schema of `features_df` will be used as the feature table schema.
- **schema** – Feature table schema. Either `schema` or `features_df` must be provided.
- **partition_columns** – Columns used to partition the feature table. If a list is provided, column ordering in the list will be used for partitioning.

  > Note
  >
  > When choosing partition columns for your feature table, use columns that do not have a high cardinality. An ideal strategy would be such that you expect data in each partition to be at least 1 GB. The most commonly used partition column is a `date`.
  >
  > Additional info: Choosing the right partition columns for Delta tables

- **description** – Description of the feature table.

`get_feature_table`(*name: str*) → **databricks.feature_store.entities.feature_table.FeatureTable**

Get a feature table’s metadata.

Parameters: **name** – A feature table name of the form `<database_name>.<table_name>`, for example `dev.user_features`.

`read_table`(*name: str, as_of_delta_timestamp: str = None*) → **pyspark.sql.dataframe.DataFrame**

Read the contents of a feature table.

Parameters:

- **name** – A feature table name of the form `<database_name>.<table_name>`, for example `dev.user_features`.
- **as_of_delta_timestamp** – If provided, reads the feature table as of this time. Only date or timestamp strings are accepted. For example, `"2019-01-01"` and `"2019-01-01T00:00:00.000Z"`.

Returns: The feature table contents, or `None` if the feature table does not exist.

`write_table`(*name: str*, *df: pyspark.sql.dataframe.DataFrame*, *mode: str*, *checkpoint_location: Optional[str] = None*, *trigger: Dict[str, Any] = {'processingTime': '5 seconds'}*) → Optional[pyspark.sql.streaming.StreamingQuery]

Writes to a feature table. If the input DataFrame is streaming, will create a write stream.

Parameters:

- **name** – A feature table name of the form `<database_name>.<table_name>`, for example `dev.user_features`. Raises an exception if this feature table does not exist.
- **df** – Spark DataFrame with feature data. Raises an exception if the schema does not match that of the feature table.
- **mode** –
  Two supported write modes:
  - `"overwrite"` updates the whole table.
  - `"merge"` will upsert the rows in `df` into the feature table. If `df` contains columns not present in the feature table, these columns will be added as new features.
- **checkpoint_location** – Sets the Structured Streaming `checkpointLocation` option. By setting a `checkpoint_location`, Spark Structured Streaming will store progress information and intermediate state, enabling recovery after failures. This parameter is only supported when the argument `df` is a streaming DataFrame.
- **trigger** – If `df.isStreaming`, `trigger` defines the timing of stream data processing, the dictionary will be unpacked and passed to `DataStreamWriter.trigger` as arguments. For example, `trigger={'once': True}` will result in a call to `DataStreamWriter.trigger(once=True)`.

Returns: If `df.isStreaming`, returns a PySpark `StreamingQuery`. `None` otherwise.

`publish_table`(*name: str*, *online_store: databricks.feature_store.online_store_spec.online_store_spec.OnlineStoreSpec*, *filter_condition: str = None*, *mode: str = 'merge'*, *streaming: bool = False*, *checkpoint_location: Optional[str] = None*, *trigger: Dict[str, Any] = {'processingTime': '5 minutes'}*) → Optional[pyspark.sql.streaming.StreamingQuery]

Publish a feature table to an online store.

Parameters:

- **name** – Name of the feature table.
- **online_store** – Specification of the online store.
- **filter_condition** – A SQL expression using feature table columns that filters feature rows prior to publishing to the online store. For example, `"dt > '2020-09-10'"`. This is analogous to running `df.filter` or a `WHERE` condition in SQL on a feature table prior to publishing.
- **mode** –
  Specifies the behavior when data already exists in this feature table in the online store. If `"overwrite"` mode is used, existing data is replaced by the new data. If `"merge"` mode is used, the new data will be merged in, under these conditions:
  - If a key exists in the online table but not the offline table, the row in the online table is unmodified.
  - If a key exists in the offline table but not the online table, the offline table row is inserted into the online table.
  - If a key exists in both the offline and the online tables, the online table row will be updated.
- **streaming** – If `True`, streams data to the online store.
- **checkpoint_location** – Sets the Structured Streaming `checkpointLocation` option. By setting a `checkpoint_location`, Spark Structured Streaming will store progress information and intermediate state, enabling recovery after failures. This parameter is only supported when `streaming=True`.
- **trigger** – If `streaming=True`, `trigger` defines the timing of stream data processing. The dictionary will be unpacked and passed to `DataStreamWriter.trigger` as arguments. For example, `trigger={'once': True}` will result in a call to `DataStreamWriter.trigger(once=True)`.

Returns: If `streaming=True`, returns a PySpark `StreamingQuery`. `None` otherwise.

> **`create_training_set`**(*df: pyspark.sql.dataframe.DataFrame, feature_lookups: List[databricks.feature_store.entities.feature_lookup.FeatureLookup], label: Union[str, List[str]], exclude_columns: List[str] = []*) → databricks.feature_store.training_set.TrainingSet

Create a `TrainingSet`.

Parameters:

- **df** – The DataFrame used to join features into.
- **feature_lookups** – List of features to join into the DataFrame.
- **label** – Names of column in DataFrame that contain training set labels.
- **exclude_columns** – Names of the columns to drop from the `TrainingSet` DataFrame.

Returns: A `TrainingSet` object.

`log_model`(*model: Any*, *artifact_path: str*, ***, *flavor: module*, *training_set: databricks.feature_store.training_set.TrainingSet*, *registered_model_name: str = None*, *await_registration_for: int = 300*, ***kwargs*)

Log an MLflow model packaged with feature lookup information.

Note

The DataFrame returned by `TrainingSet.load_df()` **must** be used to train the model. If it has been modified (for example data normalization, add a column, and similar), these modifications will not be applied at inference time, leading to training-serving skew.

Parameters:

- **model** – Model to be saved. This model must be capable of being saved by `flavor.save_model`. See the MLflow Model API.
- **artifact_path** – Run-relative artifact path.
- **flavor** – MLflow module to use to log the model. `flavor` should have type ModuleType. The module must have a method `save_model`, and must support the `python_function` flavor. For example, `mlflow.sklearn`, `mlflow.xgboost`, and similar.
- **training_set** – The `TrainingSet` used to train this model.
- **registered_model_name** –

  Note

  Experimental: This argument may change or be removed in a future release without warning.

  If given, create a model version under `registered_model_name`, also creating a registered model if one with the given name does not exist.
- **await_registration_for** – Number of seconds to wait for the model version to finish being created and is in `READY` status. By default, the function waits for five minutes. Specify `0` or `None` to skip waiting.
- **kwargs** – Additional arguments that will be passed to `flavor.save_model`.

Returns: `None`

`score_batch`(*model_uri: str*, *df: pyspark.sql.dataframe.DataFrame*, *result_type: str = 'double'*) → pyspark.sql.dataframe.DataFrame

Evaluate the model on the provided DataFrame.

Additional features required for model evaluation will be automatically retrieved from Feature Store.

The model must have been logged with `FeatureStoreClient.log_model()`, which packages the model with feature metadata. Unless present in `df`, these features will be looked up from Feature Store and joined with `df` prior to scoring the model.

If a feature is included in `df`, the provided feature values will be used rather than those stored in Feature Store.

For example, if a model is trained on two features `account_creation_date` and `num_lifetime_purchases`, as in:

```
feature_lookups = [
    FeatureLookup(
        table_name = 'trust_and_safety.customer_features',
        feature_name = 'account_creation_date',
        lookup_key = 'customer_id',
    ),
    FeatureLookup(
        table_name = 'trust_and_safety.customer_features',
        feature_name = 'num_lifetime_purchases',
        lookup_key = 'customer_id'
    ),
]

with mlflow.start_run():
    training_set = fs.create_training_set(
        df,
        feature_lookups = feature_lookups,
        label = 'is_banned',
        exclude_columns = ['customer_id']
    )
    ...
      fs.log_model(
        model,
        "model",
        flavor=mlflow.sklearn,
        training_set=training_set,
        registered_model_name="example_model"
      )
```

Then at inference time, the caller of **`FeatureStoreClient.score_batch`** must pass a DataFrame that includes **`customer_id`**, the **`lookup_key`** specified in the **`FeatureLookups`** of the **`training_set`**. If the DataFrame contains a column **`account_creation_date`**, the values of this column will be used in lieu of those in Feature Store. As in:

```
# batch_df has columns ['customer_id', 'account_creation_date']
predictions = fs.score_batch(
    'models:/example_model/1',
    batch_df
)
```

Parameters:

- **model_uri** –
  The location, in URI format, of the MLflow model logged using `FeatureStoreClient.log_model`. One of:
  - `runs:/<mlflow_run_id>/run-relative/path/to/model`
  - `models:/<model_name>/<model_version>`
  - `models:/<model_name>/<stage>`

  For more information about URI schemes, see Referencing Artifacts.
- **df** –
  The DataFrame to score the model on. Feature Store features will be joined with `df` prior to scoring the model. `df` must:

  1. Contain columns for lookup keys required to join feature data from Feature Store, as specified in the `feature_spec.yaml` artifact.
  2. Contain columns for all source keys required to score the model, as specified in the `feature_spec.yaml` artifact.

  3. Not contain a column `prediction`, which is reserved for the model's predictions. `df` may contain additional columns.
- **result_type** – The return type of the model.

See mlflow.pyfunc.spark_udf result_type.

Returns: A DataFrame containing:

1. All columns of `df`.
2. All feature values retrieved from Feature Store.
3. A column `prediction` containing the output of the model.

Decorators

`databricks.feature_store.decorators.feature_table()`

Note

Experimental: This decorator may change or be removed in a future release without warning.

The `@feature_table` decorator specifies that a function is used to generate feature data. Functions decorated with `@feature_table` must return a single DataFrame, which will be written to Feature Store. For example:

```
from databricks.feature_store import feature_table

@feature_table
def compute_customer_features(data):
    ''' Feature computation function that takes raw data
        and returns a DataFrame of features. '''
    return (data.groupBy('cid')
      .agg(count('*').alias('num_purchases'))
    )
```

A function that is decorated with the **`@feature_table`** decorator will gain these function attributes:

`compute_and_write`(*input: Dict[str, Any], feature_table_name: str, mode: str = 'merge'*) → pyspark.sql.dataframe.DataFrame

Note

Experimental: This function may change or be removed in a future release without warning.

Calls the decorated function using the provided **`input`**, then writes the output DataFrame to the feature table specified by **`feature_table_name`**.

Example:

```
compute_customer_features.compute_and_write(
    input={
        'data': data,
    },
    feature_table_name='recommender_system.customer_features',
    mode='merge'
)
```

Parameters

- **`input`** – If **`input`** is not a dictionary, it is passed to the decorated function as the first positional argument. If **`input`** is a dictionary, the contents are unpacked and passed to the decorated function as keyword arguments.
- **`feature_table_name`** – A feature table name of the form **`<database_name>.<table_name>`**, for example **`dev.user_features`**. Raises exception if this feature table does not exist.
- **`mode`** – Two supported write modes: **`"overwrite"`** updates the whole table, while **`"merge"`** will upsert the rows in **`df`** into the feature table.

Returns

DataFrame containing feature values.

`compute_and_write_streaming`(*input: Dict[str, Any], feature_table_name: str, checkpoint_location: Optional[str] = None, trigger: Dict[str, Any] = {'processingTime': '5 minutes'}*) → pyspark.sql.streaming.StreamingQuery

Note

Experimental: This function may change or be removed in a future release without warning.

Calls the decorated function using the provided input, then streams the output DataFrame to the feature table specified by **`feature_table_name`**.

Example:

```
compute_customer_features.compute_and_write_streaming(
    input={
        'data': data,
    },
    feature_table_name='recommender_system.customer_features',
)
```

Parameters

- **`input`** – If **`input`** is not a dictionary, it is passed to the decorated function as the first positional argument. If **`input`** is a dictionary, the contents are unpacked and passed to the decorated function as keyword arguments.
- **`feature_table_name`** – A feature table name of the form **`<database_name>.<table_name>`**, for example **`dev.user_features`**. Raises exception if this feature table does not exist.
- **`checkpoint_location`** – Sets the Structured Streaming **`checkpointLocation`** option. By setting a **`checkpoint_location`**, Spark Structured Streaming will store progress information and intermediate state, enabling recovery after failures. This parameter is only supported when the argument **`df`** is a streaming DataFrame.
- **`trigger`** - **`trigger`** defines the timing of stream data processing, the dictionary will be unpacked and passed to **`DataStreamWriter.trigger`** as arguments. For example, **`trigger={'once': True}`** will result in a call to **`DataStreamWriter.trigger(once=True)`**.

Returns

A PySpark **`StreamingQuery`**.

Feature Lookup

class **`databricks.feature_store.entities.feature_lookup.FeatureLookup`**(*table_name: str, feature_name: str, lookup_key: Union[str, List[str]], output_name: str = None*)

Bases:
`databricks.feature_store.entities._feature_store_object._FeatureStoreObject`

Value class used to specify a feature to use in a `TrainingSet`.

> **`__init__`**(*table_name: str, feature_name: str, lookup_key: Union[str, List[str]], output_name: str = None*)

Initialize a FeatureLookup object.

Parameters:

- **table_name** – Feature table name.
- **feature_name** – Feature name.
- **lookup_key** – Key to use when joining this feature table with the DataFrame passed to `FeatureStoreClient.create_training_set()`. The `lookup_key` must be the columns in the DataFrame passed to `FeatureStoreClient.create_training_set()`. The type of `lookup_key` columns in that DataFrame must match the type of the primary key of the feature table referenced in this `FeatureLookup`.
- **output_name** – If provided, rename this feature in the output of `FeatureStoreClient.create_training_set()`.

> **`table_name`**

Define a table name.

> **`lookup_key`**

Define a key to lookup.

> **`feature_name`**

Define a feature name.

> **`output_name`**

Define an output name.

Training Set

Module contains the TrainingSet class.

class **`databricks.feature_store.training_set.TrainingSet`**(*feature_spec: databricks.feature_store.entities.feature_spec.FeatureSpec, df: pyspark.sql.dataframe.DataFrame, labels: List[str], feature_table_metadata_map: Dict[str,* *databricks.feature_store.entities.feature_table.FeatureTable], feature_table_data_map: Dict[str, pyspark.sql.dataframe.DataFrame]*)

Bases: `object`

Class that defines TrainingSet objects.

Note

The `TrainingSet` constructor should not be called directly. Instead, call `FeatureStoreClient.create_training_set()`.

`load_df`() → pyspark.sql.dataframe.DataFrame

Load a dataframe.

Returns a DataFrame for training.

The returned DataFrame has columns specified in the `feature_spec` and `labels` parameters provided in `FeatureStoreClient.create_training_set()`.

Feature Table

Module contains the FeatureTable class.

class **`databricks.feature_store.entities.feature_table.FeatureTable`**(*name, description, primary_keys, partition_columns, features, online_stores=None, notebook_producers=None, job_producers=None, table_data_sources=None, path_data_sources=None*)

Bases: `databricks.feature_store.entities._feature_store_object._FeatureStoreObject`

Value class describing one feature table.

This will typically not be instantiated directly, instead the `FeatureStoreClient` will create `FeatureTable` objects.

Online Store Spec

Initialize the online_store_spec module.

class **`databricks.feature_store.online_store_spec.AmazonRdsMySqlSpec`**(*hostname: str, port: str, user: Optional[str] = None, password: Optional[str] = None, database_name:*

Optional[str] = None, table_name: Optional[str] = None, driver_name: Optional[str] = None, read_secret_prefix: Optional[str] = None, write_secret_prefix: Optional[str] = None)

Bases: `databricks.feature_store.online_store_spec.online_store_spec.OnlineStoreSpec`

Class that defines and creates AmazonRdsMySqlSpec objects.

This `OnlineStoreSpec` implementation is intended for publishing features to Amazon RDS MySQL and Aurora (MySQL-compatible edition).

See `OnlineStoreSpec` documentation for more usage information, including parameter descriptions.

> **`__init__`**(*hostname: str, port: str, user: str, password: str, database_name: Union[str, None] = None, table_name: Union[str, None] = None, driver_name: Union[str, None] = None, read_secret_prefix: Union[str, None] = None, write_secret_prefix: Union[str, None] = None*)

> Parameters

- **`hostname`** - Hostname to access online store.
- **`port`** - Port number to access online store.
- **`user`** - Username that has access to the online store.
- **`password`** - Password to access the online store.
- **`database_name`** - Database name.
- **`table_name`** - Table name.
- **`driver_name`** - Name of custom JDBC driver to access the online store.

> **`database_user`**

Define the database user for connection.

> **`cloud`**

Define the cloud propert for the data store.

> **`store_type`**

Define the data store type property.

class **`databricks.feature_store.online_store_spec.AzureMySqlSpec`**(*hostname: str, port: str, user: Optional[str] = None, password: Optional[str] = None, database_name: Optional[str] = None, table_name: Optional[str] = None, driver_name: Optional[str] = None, read_secret_prefix: Optional[str] = None, write_secret_prefix: Optional[str] = None*)

Bases: `databricks.feature_store.online_store_spec.online_store_spec.OnlineStoreSpec`

Define the AzureMySqlSpec class.

This `OnlineStoreSpec` implementation is intended for publishing features to Azure Database for MySQL.

See `OnlineStoreSpec` documentation for more usage information, including parameter descriptions.

> **`__init__`**(*hostname: str, port: str, user: str, password: str, database_name: Union[str, None] = None, table_name: Union[str, None] = None, driver_name: Union[str, None] = None, read_secret_prefix: Union[str, None] = None, write_secret_prefix: Union[str, None] = None*)

> **`database_user`**

Define the database user for connection.

> **`cloud`**

Define the cloud the fature store runs.

> **`store_type`**

Define the data store type.

---

class **`databricks.feature_store.online_store_spec.AzureSqlServerSpec`**(*hostname: str, port: str, user: Optional[str] = None, password: Optional[str] = None, database_name: Optional[str] = None, table_name: Optional[str] = None, driver_name: Optional[str] = None, read_secret_prefix: Optional[str] = None, write_secret_prefix: Optional[str] = None*)

Bases: `databricks.feature_store.online_store_spec.online_store_spec.OnlineStoreSpec`

This `OnlineStoreSpec` implementation is intended for publishing features to Azure SQL Database (SQL Server).

The spec supports SQL Server 2019 and newer.

See `OnlineStoreSpec` documentation for more usage information, including parameter descriptions.

> **`__init__`**(*hostname: str, port: str, user: str, password: str, database_name: Union[str, None] = None, table_name: Union[str, None] = None, driver_name: Union[str, None] = None, read_secret_prefix: Union[str, None] = None, write_secret_prefix: Union[str, None] = None*)

> **`cloud`**

Cloud provider where this online store is located.

`store_type`

Store type.

class **`databricks.feature_store.online_store_spec.OnlineStoreSpec`**(*_type, hostname: str, port: str, user: Optional[str] = None, password: Optional[str] = None, database_name: Optional[str] = None, table_name: Optional[str] = None, driver_name: Optional[str] = None, read_secret_prefix: Optional[str] = None, write_secret_prefix: Optional[str] = None*)

Bases: `abc.ABC`

Parent class for all types of OnlineStoreSpec objects.

Abstract base class for classes that specify the online store to publish to.

If `database_name` and `table_name` are not provided, `FeatureStoreClient.publish_table` will use the offline store's database and table names.

To use a different database and table name in the online store, provide values for both `database_name` and `table_name` arguments.

The JDBC driver can be customized with the optional `driver_name` argument. Otherwise, a default is used.

Strings in the primary key should not exceed 100 characters.

The online database should already exist.

Note

It is strongly suggested (but not required), to provide read-only database credentials via the `read_secret_prefix` in order to grant the least amount of database access privileges to the served model. When providing a `read_secret_prefix`, the secrets must exist in the scope name using the expected format, otherwise `publish_table` will return an error.

Parameters:

- **hostname** – Hostname to access online store.
- **port** – Port number to access online store.
- **user** – Username that has write access to the online store, or `None` if using `write_secret_prefix`.
- **password** – Password to access the online store, or `None` if using `write_secret_prefix`.
- **database_name** – Database name.
- **table_name** – Table name.
- **driver_name** – Name of custom JDBC driver to access the online store.
- **read_secret_prefix** –

The secret scope name and secret key name prefix where read-only online store credentials are stored. These credentials will be used during online feature serving to connect to the online store from the served model. The format of this parameter should be `${scope-name}/${prefix}`, which is the name of the secret scope, followed by a `/`, followed by the secret key name prefix. The scope passed in must contain the following keys and corresponding values:

  - `${prefix}-user` where `${prefix}` is the value passed into this function. For example if this function is called with `datascience/staging`, the `datascience` secret scope should contain the secret named `staging-user`, which points to a secret value with the database username for the online store.
  - `${prefix}-password` where `${prefix}` is the value passed into this function. For example if this function is called with `datascience/staging`, the `datascience` secret scope should contain the secret named `staging-password`, which points to a secret value with the database password for the online store.

- write_secret_prefix –
  The secret scope name and secret key name prefix where read-write online store credentials are stored. These credentials will be used to connect to the online store to publish features. If `user` and `password` are passed, this field must be `None`, or an exception will be raised. The format of this parameter should be `${scope-name}/${prefix}`, which is the name of the secret scope, followed by a `/`, followed by the secret key name prefix. The scope passed in must contain the following keys and corresponding values:

  - `${prefix}-user` where `${prefix}` is the value passed into this function. For example if this function is called with `datascience/staging`, the `datascience` secret scope should contain the secret named `staging-user`, which points to a secret value with the database username for the online store.
  - `${prefix}-password` where `${prefix}` is the value passed into this function. For example if this function is called with `datascience/staging`, the `datascience` secret scope should contain the secret named `staging-password`, which points to a secret value with the database password for the online store.

`type`

Type of the online store.

`hostname`

Hostname to access the online store.

`port`

Port number to access the online store.

`database_name`

Database name.

`table_name`

Table name.

`user`

Username that has access to the online store.

Username provided in constructor that has write access to the online store. Property will be empty if `write_scret_prefix` argument was used.

`password`

Password to access the online store.

Password provided in constructor that has write access to the online store. Property will be empty if `write_scret_prefix` argument was used.

`driver`

Name of the custom JDBC driver to access the online store.

`database_user`

Username that connects to the database.

`read_secret_prefix`

Name of the secret scope and prefix that contains the username and password to access the online store with read-only credentials. See the `read_secret_prefix` parameter description for details.

`write_secret_prefix`

Name of the secret scope and prefix that contains the username and password to access the online store with read/write credentials. See the `write_secret_prefix` parameter description for details.

`cloud`

Cloud provider where this online store is located.

`store_type`

Store type.