Table of Contents

[Create a workspace 2](#_Toc146028195)

[Create a lakehouse 4](#_Toc146028196)

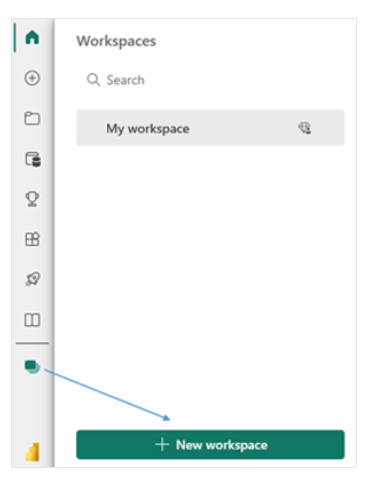
[Create Notebook 6](#_Toc146028197)

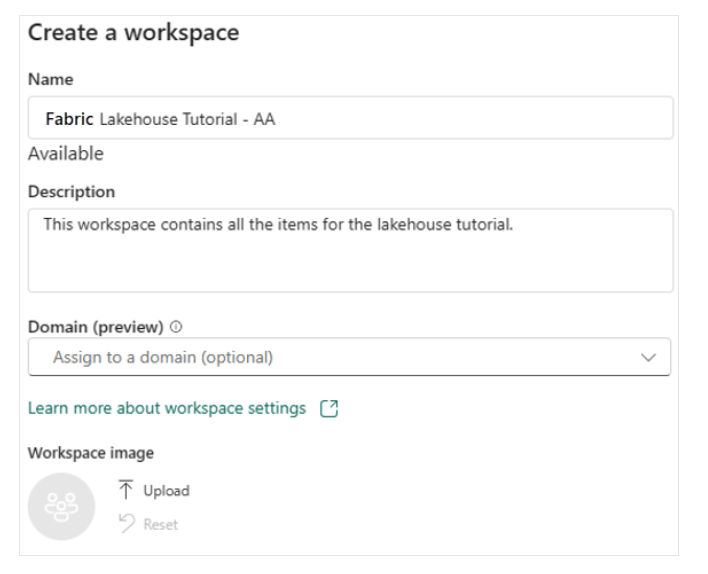
[Db\_Utils notebook 6](#_Toc146028198)

[MYSQL PIPELINE 11](#_Toc146028199)

[POSTGRES PIPELINE 14](#_Toc146028200)

# Create a workspace

1. Sign in to [Power BI](https://powerbi.com/).
2. Select **Workspaces** and **New workspace**.
3. Fill out the **Create a workspace** form with the following details:
   * **Name:** Enter *Fabric Lakehouse Tutorial*, and any extra characters to make the name unique.
   * **Description**: Enter an optional description for your workspace.



**This workspace contains all the items for the AWS to OneLake tutorial.**

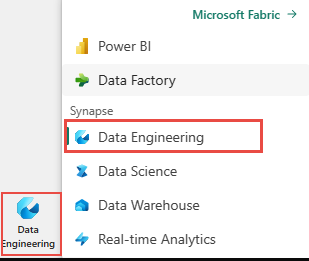
**AWS AZURE**

1. **Advanced**: Under **License mode**, select **Premium capacity** and then choose a premium capacity that you have access to.A screenshot of a computer

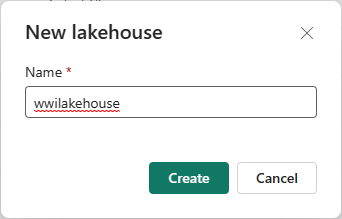
   Description automatically generated
2. Select **Apply** to create and open the workspace.

# Create a lakehouse

1. In the [Power BI service](https://powerbi.com/), select **Workspaces** from the left-hand menu.
2. To open your workspace, enter its name in the search textbox located at the top and select it from the search results.
3. From the experience switcher located at the bottom left, select **Data Engineering**.



1. In the **Data Engineering** tab, select **Lakehouse** to create a lakehouse.
2. In the **New lakehouse** dialog box, enter [AWS\_LAKEHOUSE](https://app.powerbi.com/groups/5095ec24-9758-45ce-9df1-dd12084db85f/lakehouses/5c7c74ed-6d83-4039-9806-a0ac93a7bce3?experience=data-engineering) in the **Name** field.

[](https://learn.microsoft.com/en-us/fabric/data-engineering/media/tutorial-build-lakehouse/new-lakehouse-name.png#lightbox)

**AWS\_LAKEHOUSE**

1. Select **Create** to create and open the new lakehouse.

# Create Notebook

A screenshot of a computer

Description automatically generated

# Db\_Utils notebook

* 1. import required library

from functools import partial

from pyspark.sql.dataframe import DataFrame

from delta.tables import \*

from pyspark.sql.functions import \*

* 1. This code is meant to help manage data in an Azure Data Lake Gen2 storage system by reading, writing, and updating data in Delta tables.

def write\_delta\_v2(type: str, df: DataFrame, mode: str, targetTable: str, dbname: str, sourceTable: str = None, keyCol: str = None, delta\_path: str = None, partition: str = None, partitionRange: list = None, upsert\_partition\_value: str = None) -> str:

  """

    Write a DataFrame to a Delta table in Azure Data Lake Gen2.

    :param type: Type of operation ('data\_copy' or 'transformation').

    :param df: DataFrame to write to the Delta table.

    :param mode: Write mode ('append', 'overwrite', 'upsert').

    :param targetTable: Name of the target Delta table.

    :param dbname: Database name.

    :param sourceTable: Name of the source table for upsert mode.

    :param keyCol: Column(s) to use as the key for upsert mode.

    :param delta\_path: Path to the Delta table in Data Lake Gen2.

    :param partition: Column to partition the Delta table on.

    :param partitionRange: List of partition values to write in overwrite mode.

    :param upsert\_partition\_value: Partition value to use for upsert mode.

    :return: Delta table path.

  """

  # If delta\_path is not provided, set default paths based on 'type'.

  if delta\_path == None:

    if type == 'data\_copy':

        path = 'abfss://AWS\_AZURE@onelake.dfs.fabric.microsoft.com/AWS\_LAKEHOUSE.Lakehouse'

        delta\_path = path+'/Tables/'+targetTable

    if type == 'transformation':

        path = 'abfss://AWS\_AZURE@onelake.dfs.fabric.microsoft.com/AWS\_LAKEHOUSE.Lakehouse'

        # delta\_path = path+'/TABLES/'+targetTable

  # Modify the targetTable to include the database name.

  targetTable = 'AWS\_LAKEHOUSE'+'.'+targetTable

  # Check if the specified 'mode' is valid.

  assert mode in ['append', 'overwrite', 'upsert'], "Mode should be either 'append' or 'overwrite', 'upsert'"

  # Check if the target table already exists.

  tableTest = spark.catalog.\_jcatalog.tableExists(targetTable)

  if not tableTest:

    # If the table does not exist, create it.

    print("write table for the first time: ")

    if partition is None:

      # If no partition is specified, write the DataFrame to the Delta table.

      df.write.format("delta").mode(mode).save(delta\_path)

    else:

      # If a partition is specified, write the DataFrame with partitioning.

      df.write.format("delta").mode(mode).option("mergeSchema", "true").partitionBy(partition).save(delta\_path)

    # Print table name and path for reference.

    print(targetTable, delta\_path)

  else:

    # If the target table already exists.

    print("table already exists!")

    if mode == 'upsert':

      # If the mode is 'upsert', perform upsert operation.

      print("upsert mode")

      df.createOrReplaceTempView(sourceTable)

      res = keyCol.strip('][').split(',')

      keyList = [f"tar.{c} = src.{c}" for c in res]

      keyString = " and ".join(keyList)

      columns = df.schema.names

      columns = list(map(lambda x: x.lower(), columns))

      updateCols = columns[:]

      for c in res:

        updateCols.remove(c.lower())

      updateList = [f"tar.{c} = src.{c}" for c in updateCols]

      updateString = "UPDATE SET " + ", ".join(updateList)

      insertString = f"""INSERT ({", ".join(columns)}) VALUES ({", ".join(columns)})"""

      upsertSql = f"""MERGE INTO {targetTable} AS tar \

          USING {sourceTable} AS src \

          ON {keyString} \

          WHEN MATCHED THEN \

            {updateString} \

          WHEN NOT MATCHED THEN \

            {insertString} \

          """

      if partition is not None and upsert\_partition\_value is not None:

        upsertSql = f"""MERGE INTO {targetTable} AS tar

          USING {sourceTable} AS src

          ON tar.{keyCol} = src.{keyCol} AND tar.{partition} = '{upsert\_partition\_value}'

          WHEN MATCHED THEN

            {updateString}

          WHEN NOT MATCHED THEN

            {insertString}

          """

      spark.sql(upsertSql)

    elif mode == 'overwrite':

      # If the mode is 'overwrite', perform overwrite operation.

      print("overwrite mode")

      if partitionRange: # Test if partitionRange list is not empty.

        partitionRange = [str(x) for x in partitionRange]

        writeString = ", ".join(partitionRange)

        writeCon = f"""{partition} in ({writeString})"""

if partition is None:

          df.filter(writeCon).write.format("delta").option("mergeSchema", "true").option("overwriteSchema", "true").mode(mode).option("replaceWhere", writeCon).save(delta\_path)

        else:

          df.filter(writeCon).write.format("delta").option("mergeSchema", "true").option("overwriteSchema", "true").partitionBy(partition).mode(mode).option("replaceWhere", writeCon).save(delta\_path)

      else:

        if partition is None:

          df.write.format("delta").option("mergeSchema", "true").option("overwriteSchema", "true").mode(mode).save(delta\_path)

        else:

          df.write.format("delta").option("mergeSchema", "true").option("overwriteSchema", "true").partitionBy(partition).mode(mode).save(delta\_path)

    else:

      # If the mode is 'append', perform append operation.

      print("append mode")

      if partition is None:

        df.write.format("delta").mode(mode).save(delta\_path)

      else:

        df.write.format("delta").partitionBy(partition).mode(mode).save(delta\_path)

  return delta\_path

**Parameters Explained:**

* **type**: Specify whether you're copying data (**data\_copy**) or transforming it (**transformation**).
* **df**: This is like a spreadsheet of data that you want to work with.
* **mode**: Choose how you want to handle the data in the target table. You can add new data (**append**), completely replace existing data (**overwrite**), or update existing data based on certain conditions (**upsert**).
* **targetTable**: Name of the table where you want to store or update data.
* **dbname**: The name of the database where your table resides.
* **sourceTable**: If you're using the 'upsert' mode, you specify the source table to compare and update with the target table.
* **keyCol**: If you're doing an 'upsert,' this is the column used to match records between the source and target tables.
* **delta\_path**: The location where the data will be stored in your Azure Data Lake Gen2 storage.
* **partition**: If you want to organize your data into separate sections, you can specify a column for partitioning.
* **partitionRange**: If you're in 'overwrite' mode, you can specify a range of partitions to replace.
* **upsert\_partition\_value**: If you're in 'upsert' mode and using partitions, you can specify the partition value to update.

**Explanation of the Code:**

1. It checks if you provided a location for your Delta table (**delta\_path**). If not, it sets a default location based on the operation type.
2. It modifies the target table name to include the database name.
3. It ensures that the specified mode ('append', 'overwrite', or 'upsert') is valid.
4. It checks if the target table already exists. If not, it creates the table and, if specified, partitions it.
5. If the table exists and you're in 'upsert' mode, it updates matching records in the target table using data from the source table.
6. If you're in 'overwrite' mode and have specified a range of partitions, it replaces data in those partitions.
7. If none of the above conditions apply, it appends the data to the Delta table.
8. Finally, it returns the path to the Delta table where your data is stored.

This code helps manage data by providing options for adding, updating, or replacing data in your Azure Data Lake Gen2 storage system, making it easier to work with your data in a flexible way.

# MYSQL PIPELINE

%run /db\_utils

from pyspark.sql import SparkSession

# Define database connection properties

database\_host = "intechmysql.ced69anvo4ui.ap-south-1.rds.amazonaws.com"

database\_port = '3306'

database\_name = 'init\_db'

user = 'ISPL123'

password = 'ISPL1234'

driver = "com.mysql.jdbc.Driver"

url = f"jdbc:mysql://{database\_host}:{database\_port}/{database\_name}"

# Install pymysql if not already installed (used for MySQL database connection)

!pip install pymysql

import pymysql

# Establish a connection to the MySQL server

conn = pymysql.connect(host=database\_host, user=user, password=password, database="init\_db")

# Create a cursor object to execute queries

cur = conn.cursor()

# Execute the SHOW TABLES query to fetch table names

cur.execute('SHOW TABLES')

table\_names\_temp = cur.fetchall()

table\_names = [row[0] for row in table\_names\_temp]

table\_names\_updated = []

# Check for tables that have a 'empid' column and add them to the list

for table in table\_names:

    cur.execute(f"SHOW COLUMNS FROM {database\_name}.{table} LIKE 'empid'")

    column\_exists = cur.fetchone() is not None

    if column\_exists:

        table\_names\_updated.append(table)

# Print the updated table names

for table2 in table\_names\_updated:

    print(table2)

# Close the cursor and connection to the MySQL server

cur.close()

conn.close()

# Close the cursor and connection to the MySQL server

cur.close()

conn.close()

# Set the action to either 'upsert' or 'overwrite'

# action = 'upsert'

action = 'overwrite'

# Iterate through the updated table names

for table\_name1 in table\_names\_updated:

    tgt\_table\_name = table\_name1

    # Construct the query to retrieve data from the MySQL table

    query = f"(select \* from {table\_name1})"

    # Read data from MySQL into a Spark DataFrame

    df = spark.read.format('jdbc').\

             option('url',url).\

             option('driver',driver).\

             option('user',user).\

             option('password',password).\

             option('query',query).\

             load()# .\

            #  limit(10)

    # Define the target database name

    tgt\_db\_name = 'tgt\_db\_name'

    tgt\_table\_name = table\_name1

    # Depending on the action chosen ('upsert' or 'overwrite'), call the write\_delta\_v2 function

    if action == 'upsert':

        write\_delta\_v2(type='data\_copy', df=df, mode=action, dbname=tgt\_db\_name, targetTable=tgt\_table\_name, sourceTable=tgt\_table\_name, keyCol='empid')

    else:

        write\_delta\_v2(type='data\_copy', df=df, mode=action, dbname=tgt\_db\_name, targetTable=tgt\_table\_name)

**Explanation:**

1. The code starts by importing the **write\_delta\_v2** function from the **db\_utils** module. This function is presumably used for writing data to Delta tables.
2. Database connection properties like host, port, database name, username, password, and driver are defined. These are used to connect to a MySQL database.
3. The code installs the **pymysql** library, which is a Python MySQL connector, if it's not already installed.
4. It establishes a connection to the MySQL server using the provided credentials.
5. A cursor object **cur** is created to execute SQL queries against the MySQL database.
6. The code executes a SQL query to fetch the names of all tables in the database and stores them in **table\_names**.
7. It then iterates through **table\_names** and checks if each table has a column named 'empid'. If a table has this column, its name is added to **table\_names\_updated**.
8. The code prints the names of tables that have the 'empid' column.
9. The cursor and database connection are closed after the necessary information is retrieved from the MySQL database.
10. The variable **action** is set to either 'upsert' or 'overwrite' to determine the type of operation to perform on Delta tables.
11. The code iterates through the tables in **table\_names\_updated**. For each table:
    * It sets **tgt\_table\_name** to the current table name.
    * Constructs a SQL query to select all data from the MySQL table.
    * Reads the data from the MySQL table into a Spark DataFrame using the specified connection properties.
    * Defines the target database name as **tgt\_db\_name**.
    * Depending on the **action** chosen, it calls the **write\_delta\_v2** function to write the data to a Delta table. If 'upsert' is chosen, it performs an upsert operation using the 'empid' column as the key; otherwise, it overwrites the Delta table.

# POSTGRES PIPELINE

%run /db\_utils

# Define database connection properties for a PostgreSQL database

database\_host = "intechpostgresql.ced69anvo4ui.ap-south-1.rds.amazonaws.com"

database\_port = '5432'

database\_name = 'init\_db'

user = "ISPL123"

password = "ISPL1234"

driver = "org.postgresql.Driver"

# Define the JDBC URL to connect to the PostgreSQL database

url = f"jdbc:postgresql://{database\_host}:{database\_port}/{database\_name}"

# Query to get all the table names inside the 'public' schema of the database

query = f"(SELECT tablename FROM pg\_catalog.pg\_tables where schemaname = 'public' )  as tempTable"

# Read table names from the PostgreSQL database

table\_names = (spark.read

    .format("jdbc")

    .option("driver", driver)

    .option("url", url)

    .option("dbtable", query)

    .option("user", user)

    .option("password", password)

    .load()

    )

# Print the number of table names retrieved

print(table\_names.count())

# Show the retrieved table names

table\_names.show()

# Collect the table names into a Python list

table\_names\_list = table\_names.rdd.map(lambda row: row[0]).collect()

# Define the action to be performed on Delta tables ('upsert' or 'overwrite')

# action = 'overwrite'

action = 'upsert'

# Iterate through the retrieved table names

for table\_name in table\_names\_list:

    tgt\_table\_name = table\_name

    # Construct a SQL query to select all data from the current table

    query = f"(SELECT \* FROM {table\_name}) as tempTable"

    # Read data from the PostgreSQL table into a Spark DataFrame

    df = (spark.read

    .format("jdbc")

    .option("driver", driver)

    .option("url", url)

    .option("dbtable", query)

    .option("user", user)

    .option("password", password)

    .load()

    )

    # Replace spaces with underscores in column names

    df = df.select([replace\_space\_with\_underscore(col\_name) for col\_name in df.columns])

    # Replace dashes with underscores in column names

    df = df.select([replace\_dash\_with\_underscore(col\_name) for col\_name in df.columns])

    # Define the target database name

    tgt\_db\_name = 'tgt\_db\_name'

    # Convert the table name to uppercase (Delta table naming convention)

    table\_name = table\_name.upper()

    tgt\_table\_name = table\_name

    # Write the DataFrame into a Delta Lake table

    if action == 'upsert':

        write\_delta\_v2(type='data\_copy', df=df, mode=action, dbname=tgt\_db\_name, targetTable=tgt\_table\_name, sourceTable=tgt\_table\_name, keyCol='Order\_ID')

    else:

        write\_delta\_v2(type='data\_copy', df=df, mode=action, dbname=tgt\_db\_name, targetTable=tgt\_table\_name)

# Reports

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated