**Madiha Aimon Tappal**

[madihaaimon@gmail.com](mailto:madihaaimon@gmail.com)

**Data Engineering Batch – 1**

**Day – 18 Assignment**

**Azure Databricks**

**Delta lake in azure Data bricks: -**

Delta Lake is an open-source storage layer that brings ACID transactions to Apache Spark and big data workloads. It provides a reliable and scalable solution for managing data lakes on top of cloud storage systems. Here are key theoretical aspects of Delta Lake:

1. **ACID Transactions:**
   * **Atomicity:** Delta Lake ensures that transactions are treated as single, indivisible units, guaranteeing that either all operations within a transaction are completed, or none of them are.
   * **Consistency:** The system remains in a consistent state before and after the execution of a transaction, enforcing data integrity constraints.
2. **Schema Evolution:**
   * Delta Lake supports schema evolution, allowing for the modification of table schemas without requiring a full rewrite of the data. This flexibility is crucial for adapting to changing data requirements.
3. **Time Travel:**
   * Delta Lake provides time travel capabilities, allowing users to query a table as of a specific version or timestamp. This feature is beneficial for auditing, debugging, and analyzing historical data states.
4. **Concurrency Control:**
   * Delta Lake handles concurrent read and write operations efficiently, ensuring that multiple users or processes can interact with the data lake without compromising data consistency.
5. **Scalable Metadata Handling:**
   * Delta Lake efficiently manages metadata, making it scalable for large datasets. Metadata operations are optimized to prevent bottlenecks and ensure high-performance metadata handling.
6. **Optimizations:**
   * Delta Lake incorporates various optimizations to enhance query performance. Techniques such as data skipping, Z-ordering, and caching are employed to improve the speed of data access and processing.
7. **Unified Batch and Streaming Processing:**
   * Delta Lake supports both batch and streaming processing. This allows users to seamlessly integrate data streaming into their big data workflows, maintaining consistency and reliability across both processing modes.
8. **Compatibility:**
   * Delta Lake is compatible with Apache Spark, making it easy to integrate with existing Spark-based data processing workflows. It can be used in conjunction with other Spark components and libraries.
9. **Integration with Cloud Storage:**
   * Delta Lake is designed to work seamlessly with cloud storage systems, such as Azure Data Lake Storage (ADLS) or Amazon S3. This ensures that it can be easily deployed in cloud-based environments.

In summary, Delta Lake enhances the reliability, performance, and manageability of data lakes by providing ACID transactions, schema evolution, time travel, concurrency control, scalable metadata handling, optimizations, and compatibility with cloud storage systems.

Topic: Introduction to Databricks Delta Lake

create Delta Table with Existing Data in Databricks-

Azure

Delta Lake is an open-source storage layer that brings reliability to data lakes. Delta Lake

provides ACID transactions, scalable metadata handling, and unifies streaming and batch data

processing. Delta Lake runs on top of your existing data lake and is fully compatible with Apache

Spark APIs.

This recipe teaches us how to create an external table over stored data in a specific location.

Implementation Details For ‘How To Create A Delta

Table in Databricks

1. Databricks Community Edition click here

2. Spark-scala

3. storage - Databricks File System(DBFS)

Steps For Creating Delta Table in Databricks

The following steps will show you how to create delta table in Azure Databricks using existing

data.

Step 1: Uploading data to DBFS

Follow the below steps to upload data files from local to DBFS

1. Click create in Databricks menu

2. Click Table in the drop-down menu, it will open a create new table UI

3. In UI, specify the folder name in which you want to save your files.

4. click browse to upload and upload files from local.

5. path is like /FileStore/tables/your folder name/your file

**Create a Delta table: -**

To create a Delta table, write a DataFrame out in the delta format. You can use existing Spark SQL code and change the format from parquet, csv, json, and so on, to delta.

SQL:

**CREATE** **TABLE** delta.`/tmp/delta-**table**` **USING** DELTA **AS** **SELECT** col1 **as** id **FROM** **VALUES** 0,1,2,3,4;

PYTHON:

data = spark.range(0, 5)

data.write.format("delta").save("/tmp/delta-table")

These operations create a new Delta table using the schema that was inferred from your DataFrame. For the full set of options available when you create a new Delta table, see [Create a table](https://docs.delta.io/latest/delta-batch.html#-ddlcreatetable) and [Write to a table](https://docs.delta.io/latest/delta-batch.html#-deltadataframewrites).

**Read data: -**

You read data in your Delta table by specifying the path to the files: "/tmp/delta-table":

SQL:

**SELECT** \* **FROM** delta.`/tmp/delta-**table**`;

PYTHON:

df = spark.read.format("delta").load("/tmp/delta-table")

df.show()

## [**Update table data**](https://docs.delta.io/latest/quick-start.html#id7)**: -**

Delta Lake supports several operations to modify tables using standard DataFrame APIs. This example runs a batch job to overwrite the data in the table:

[Overwrite](https://docs.delta.io/latest/quick-start.html#id8)

**INSERT** OVERWRITE delta.`/tmp/delta-**table**` **SELECT** col1 **as** id **FROM** **VALUES** 5,6,7,8,9;

If you read this table again, you should see only the values 5-9 you have added because you overwrote the previous data.

### [Conditional update without overwrite](https://docs.delta.io/latest/quick-start.html#id9): -

Delta Lake provides programmatic APIs to conditional update, delete, and merge (upsert) data into tables. Here are a few examples

PYTHON:

**from** delta.tables **import** \*

**from** pyspark.sql.functions **import** \*

deltaTable = DeltaTable.forPath(spark, "/tmp/delta-table")

# Update every even value by adding 100 to it

deltaTable.update(

condition = expr("id % 2 == 0"),

set = { "id": expr("id + 100") })

# Delete every even value

deltaTable.delete(condition = expr("id % 2 == 0"))

# Upsert (merge) new data

newData = spark.range(0, 20)

deltaTable.alias("oldData") \

.merge(

newData.alias("newData"),

"oldData.id = newData.id") \

.whenMatchedUpdate(set = { "id": col("newData.id") }) \

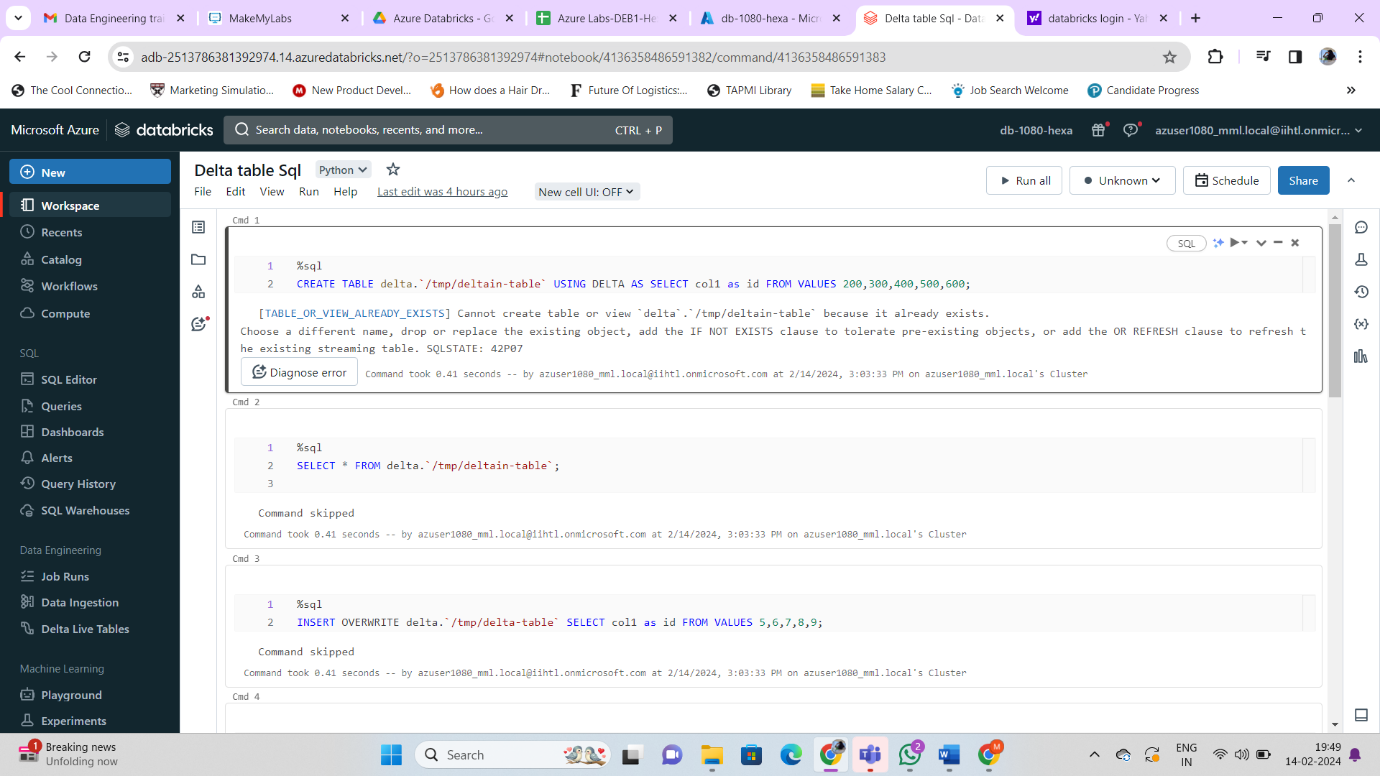
.whenNotMatchedInsert(values = { "id": col("newData.id") }) \

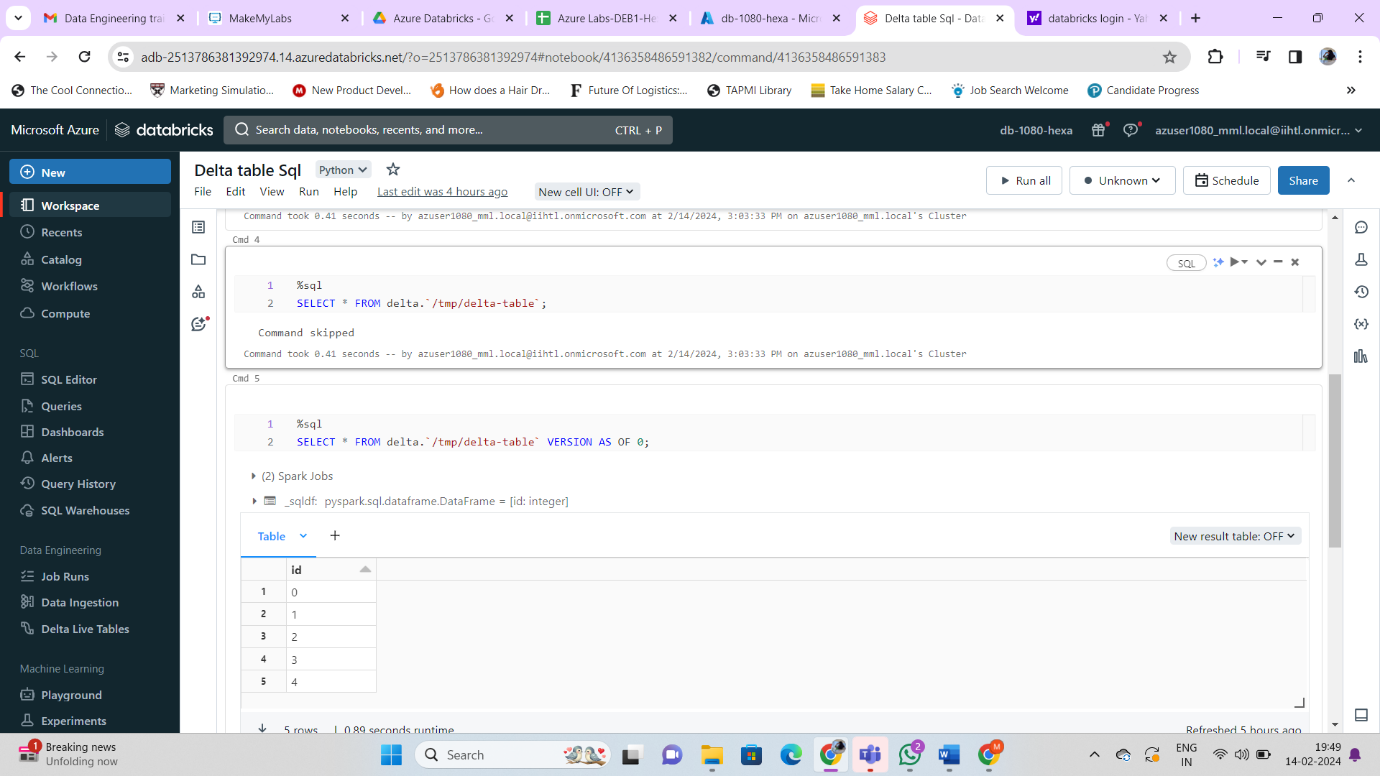
.execute()

deltaTable.toDF().show()

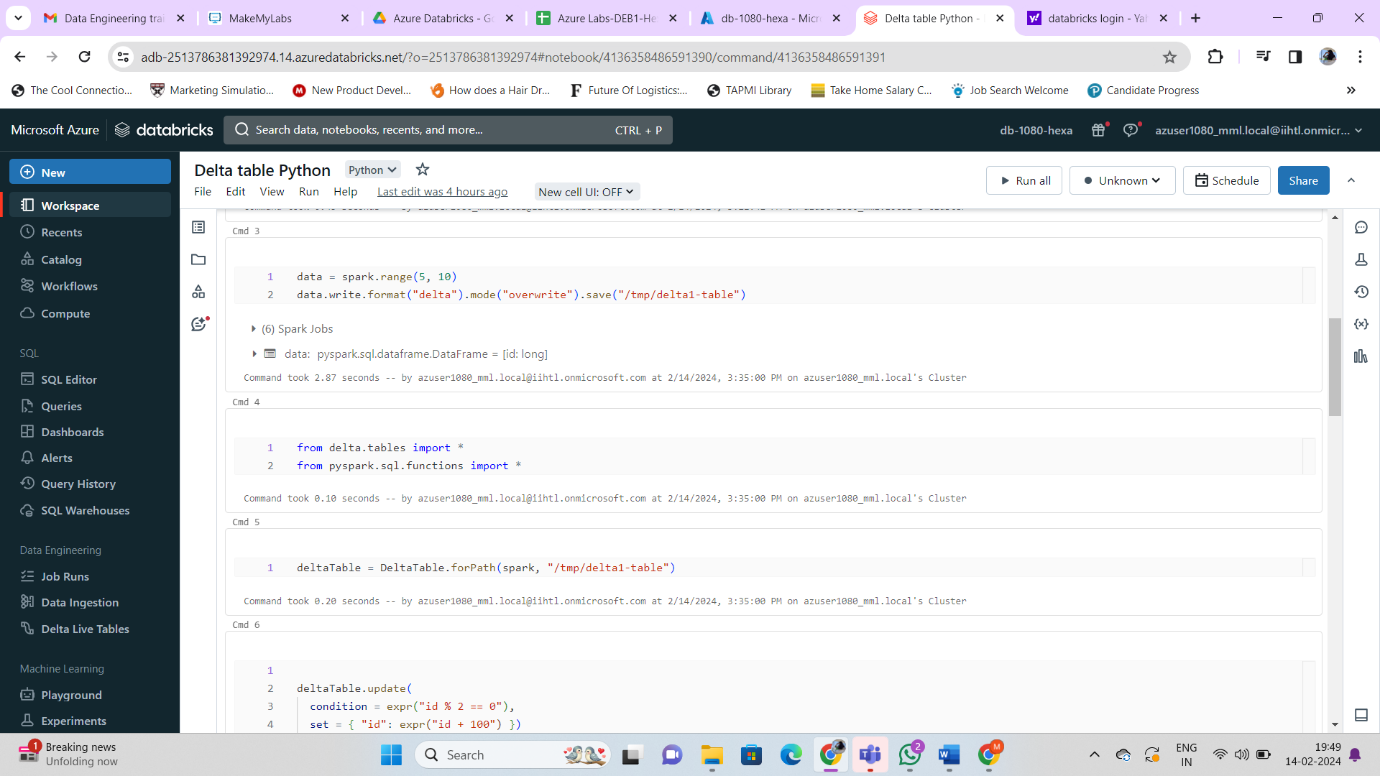
You should see that some of the existing rows have been updated and new rows have been inserted.

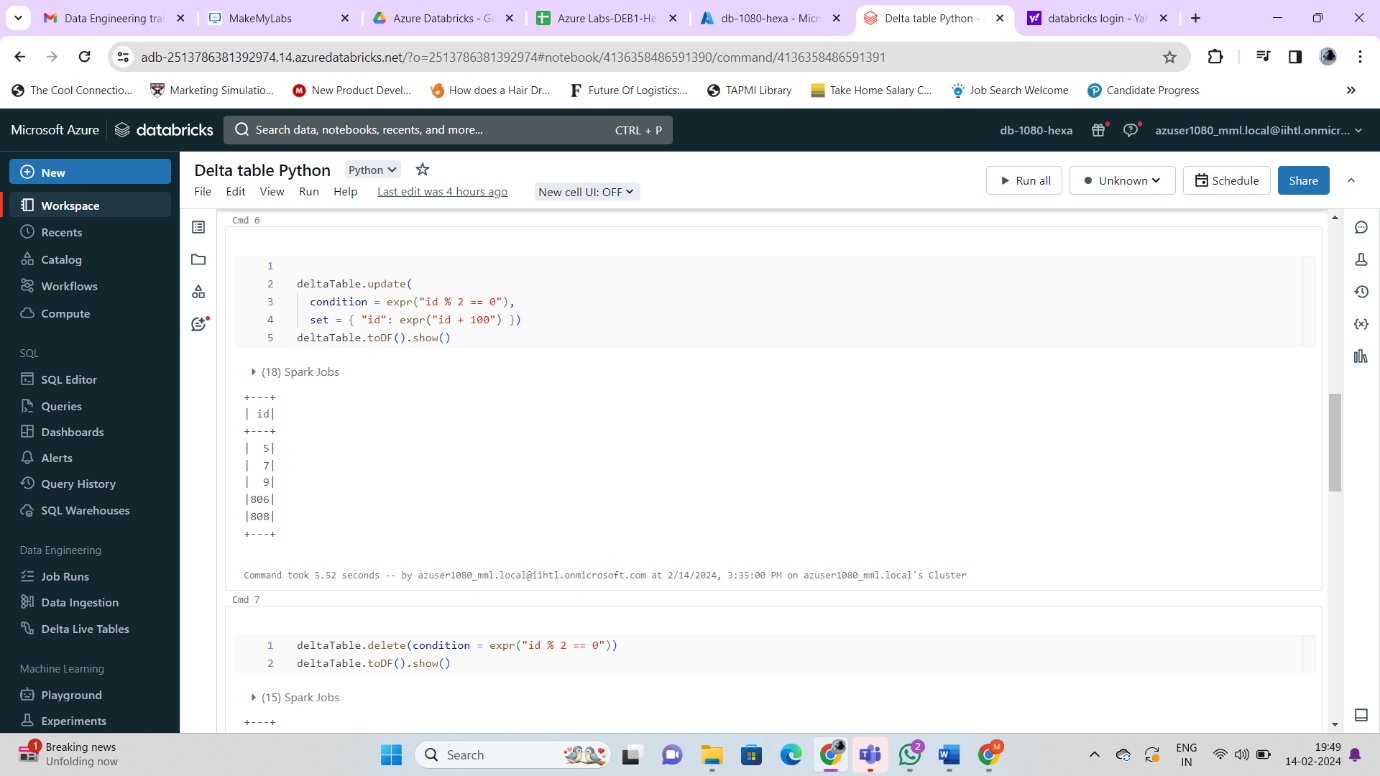
**SQL :**

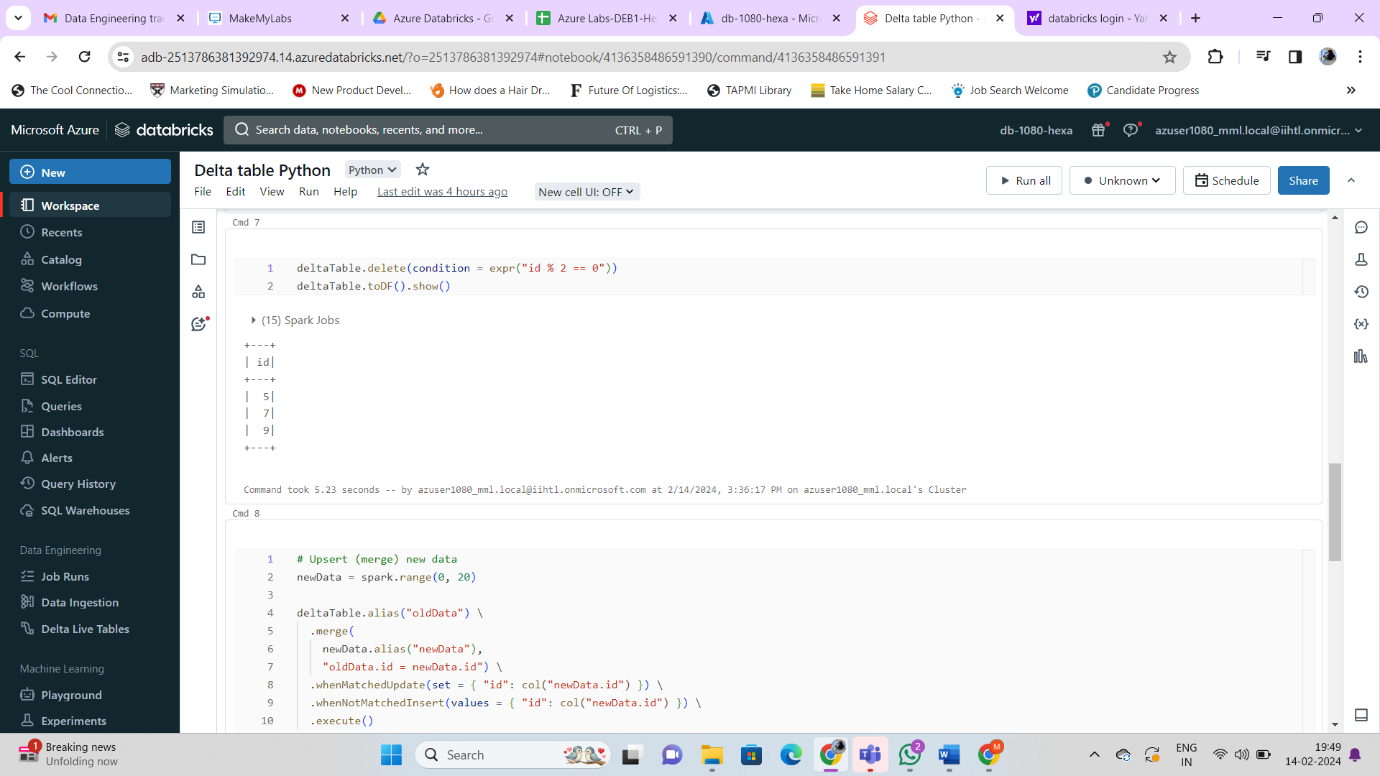
****

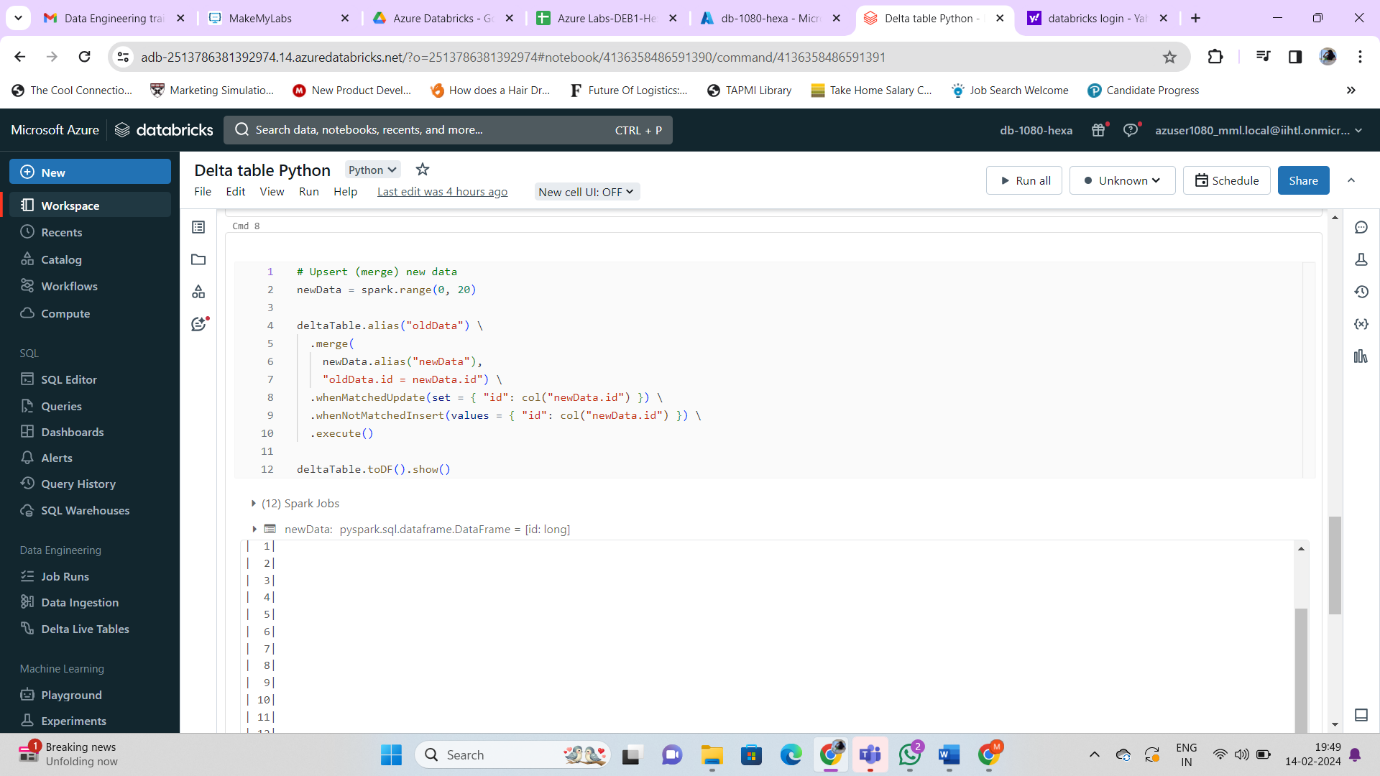
****

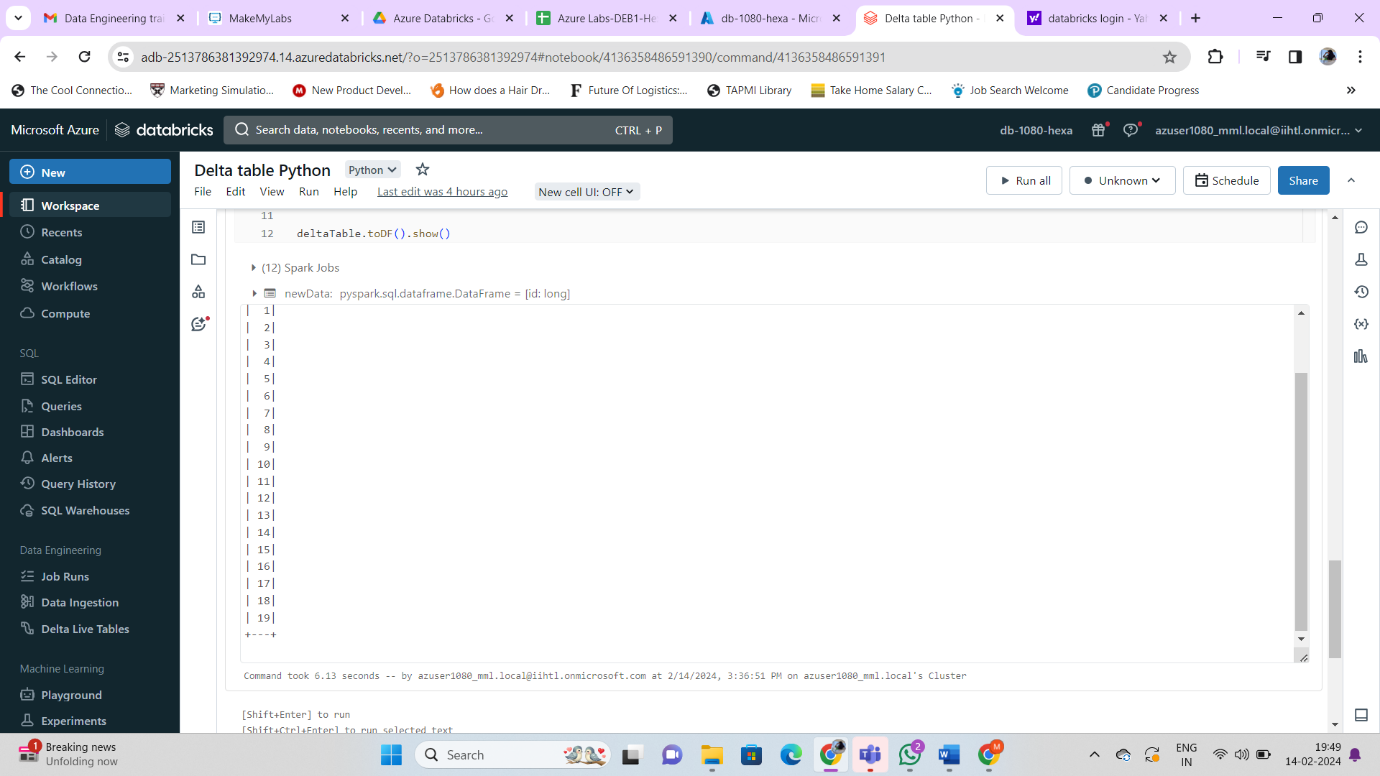
**Python:**

****

****

****

****

****