Delta Lake is an open source project to build a transactional data storage layer for Spark on top of a data lake. Delta Lake adds support for relational semantics for both batch and streaming data operations, and enables the creation of a Lakehouse architecture in which Apache Spark can be used to process and query data in tables that are based on underlying files in the data lake.

This lab will take approximately ****40**** minutes to complete.

## Provision an Azure Databricks workspace

****Tip****: If you already have an Azure Databricks workspace, you can skip this procedure and use your existing workspace.

This exercise includes a script to provision a new Azure Databricks workspace. The script attempts to create a Premium tier Azure Databricks workspace resource in a region in which your Azure subscription has sufficient quota for the compute cores required in this exercise; and assumes your user account has sufficient permissions in the subscription to create an Azure Databricks workspace resource. If the script fails due to insufficient quota or permissions, you can try creating an Azure Databricks workspace interactively in the Azure portal.

1. In a web browser, sign into the [Azure portal](https://portal.azure.com/) at https://portal.azure.com.
2. In the PowerShell pane, enter the following commands to clone this repo:

rm -r mslearn-databricks -f

git clone https://github.com/MicrosoftLearning/mslearn-databricks

1. After the repo has been cloned, enter the following command to run the ****setup.ps1**** script, which provisions an Azure Databricks workspace in an available region:

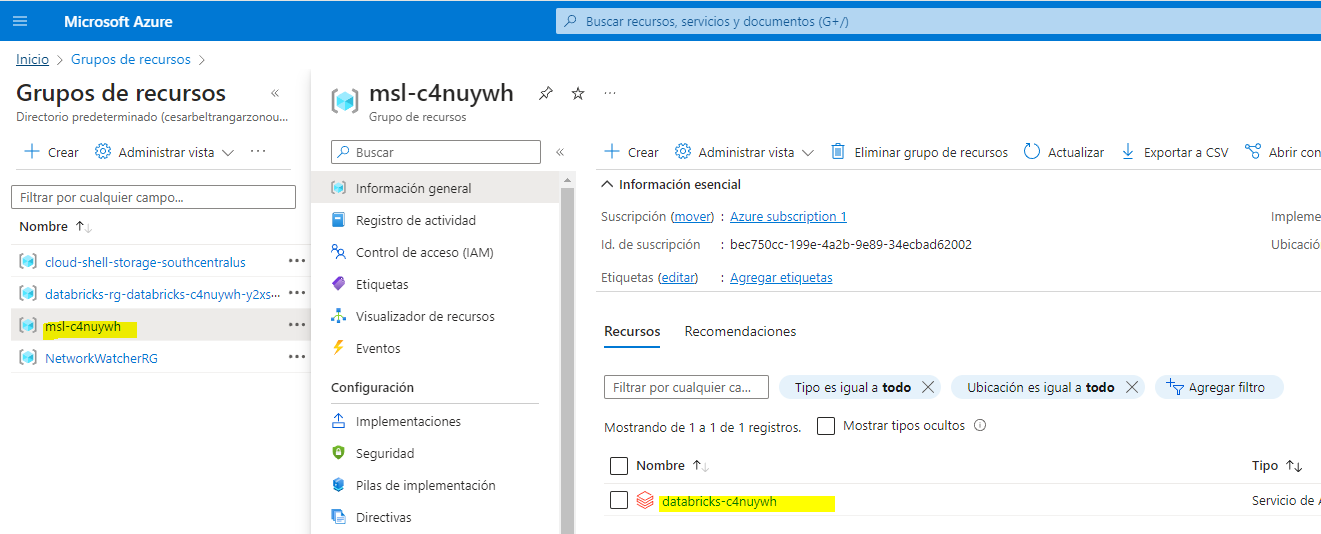
./mslearn-databricks/setup.ps1

## Create a cluster

Azure Databricks is a distributed processing platform that uses Apache Spark clusters to process data in parallel on multiple nodes. Each cluster consists of a driver node to coordinate the work, and worker nodes to perform processing tasks. In this exercise, you’ll create a single-node cluster to minimize the compute resources used in the lab environment (in which resources may be constrained). In a production environment, you’d typically create a cluster with multiple worker nodes.

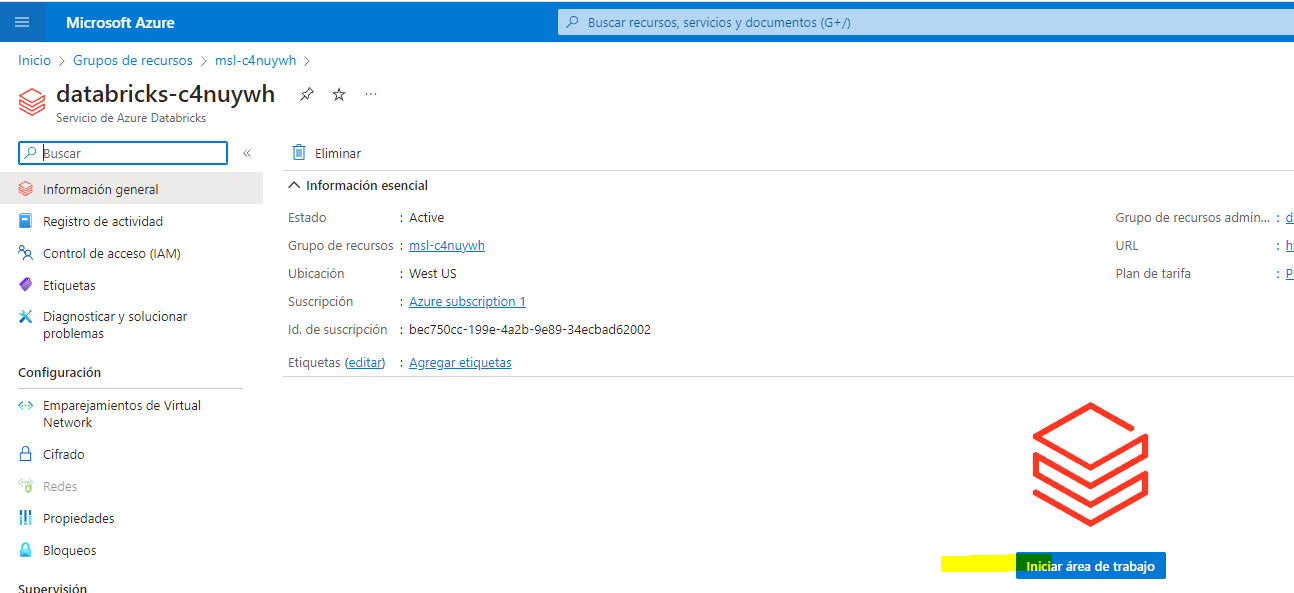
****Tip****: If you already have a cluster with a 13.3 LTS or higher runtime version in your Azure Databricks workspace, you can use it to complete this exercise and skip this procedure.

1. In the Azure portal, browse to the ****msl-**xxxxxxx** resource group that was created by the script (or the resource group containing your existing Azure Databricks workspace)
2. Select your Azure Databricks Service resource (named ****databricks-**xxxxxxx** if you used the setup script to create it).

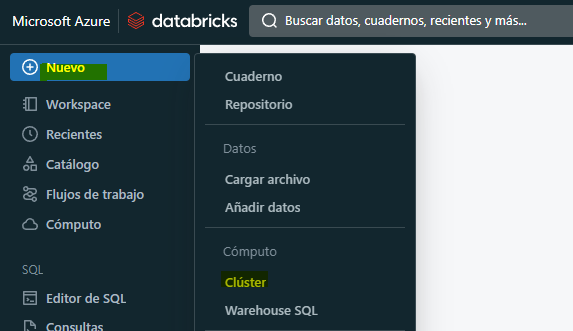


1. In the ****Overview**** page for your workspace, use the ****Launch Workspace**** button to open your Azure Databricks workspace in a new browser tab; signing in if prompted.

****Tip****: As you use the Databricks Workspace portal, various tips and notifications may be displayed. Dismiss these and follow the instructions provided to complete the tasks in this exercise.

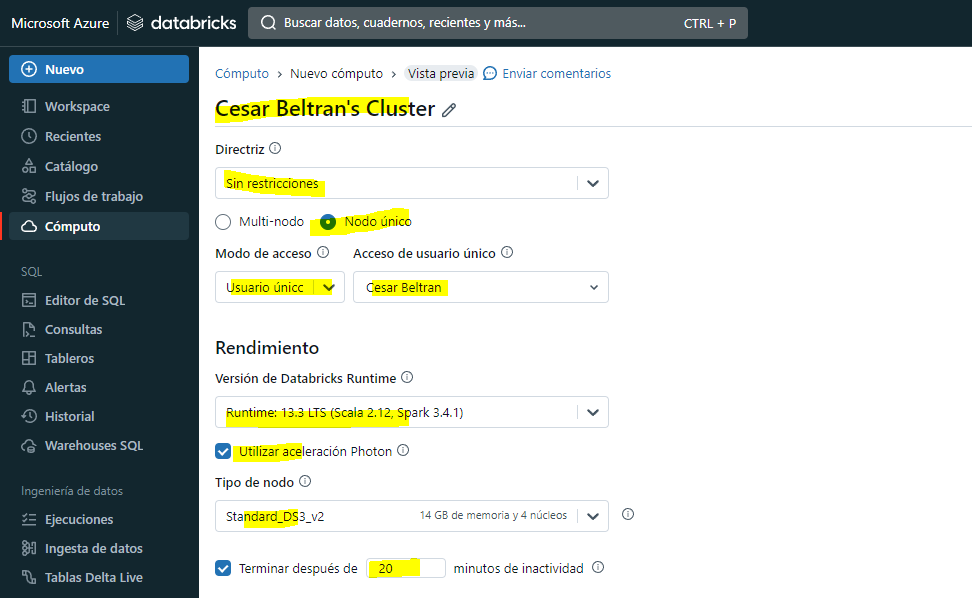


1. In the sidebar on the left, select the ****(+) New**** task, and then select ****Cluster****.



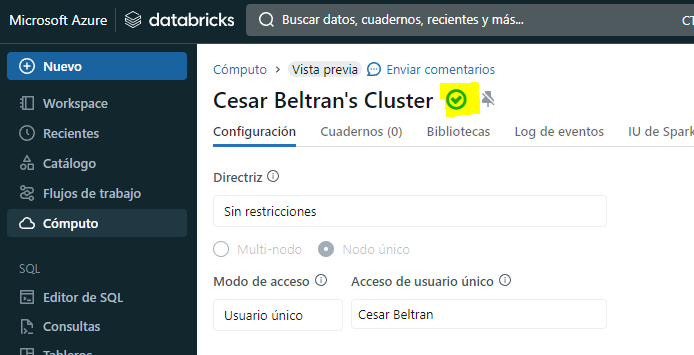
1. In the ****New Cluster**** page, create a new cluster with the following settings:

* ****Cluster name****: User Name’s cluster (the default cluster name)
* ****Policy****: Unrestricted
* ****Cluster mode****: Single Node
* ****Access mode****: Single user (with your user account selected)
* ****Databricks runtime version****: 13.3 LTS (Spark 3.4.1, Scala 2.12) or later
* ****Use Photon Acceleration****: Selected
* ****Node type****: Standard\_DS3\_v2
* ****Terminate after**** 20 ****minutes of inactivity****



1. Wait for the cluster to be created. It may take a minute or two.

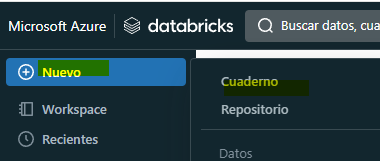
****Note****: If your cluster fails to start, your subscription may have insufficient quota in the region where your Azure Databricks workspace is provisioned. See [CPU core limit prevents cluster creation](https://docs.microsoft.com/azure/databricks/kb/clusters/azure-core-limit) for details. If this happens, you can try deleting your workspace and creating a new one in a different region. You can specify a region as a parameter for the setup script like this: ./mslearn-databricks/setup.ps1 eastus



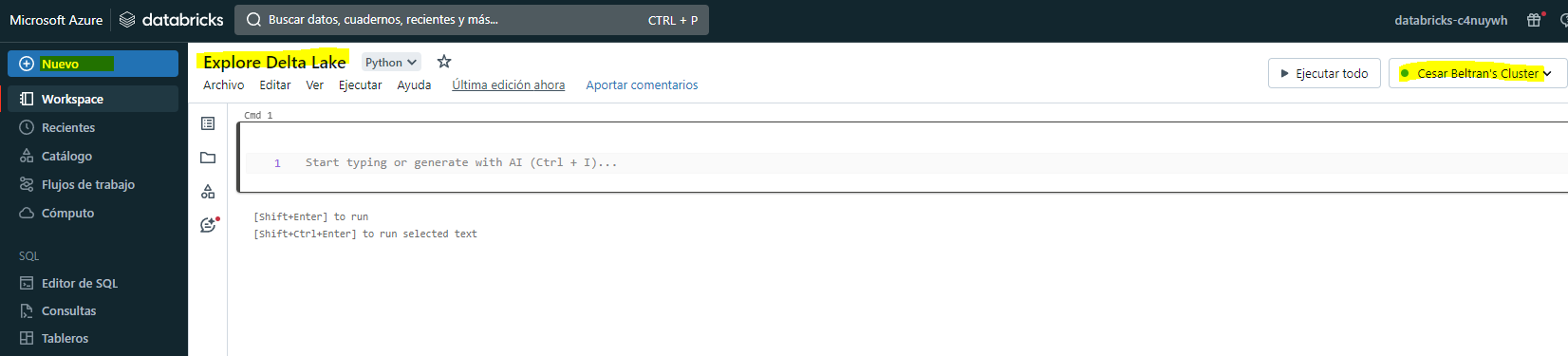
## Create a notebook and ingest data

Now let’s create a Spark notebook and import the data that we’ll work with in this exercise.

1. In the sidebar, use the ****(+) New**** link to create a ****Notebook****.



1. Change the default notebook name (****Untitled Notebook**[date]**) to ****Explore Delta Lake**** and in the ****Connect**** drop-down list, select your cluster if it is not already selected. If the cluster is not running, it may take a minute or so to start.



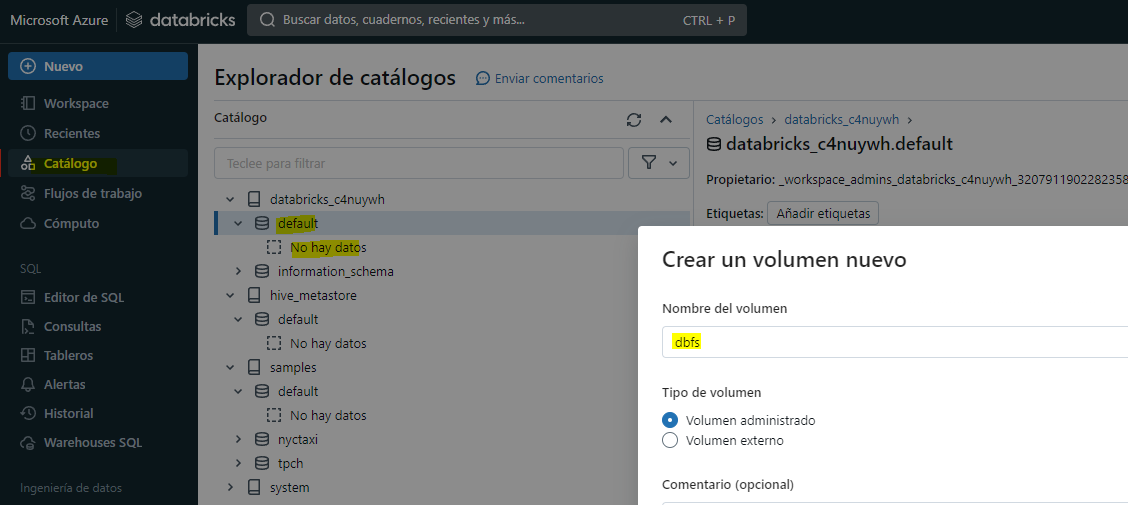
1. In the first cell of the notebook, enter the following code, which uses shell commands to download data files from GitHub into the Databricks file system (DBFS) used by your cluster.

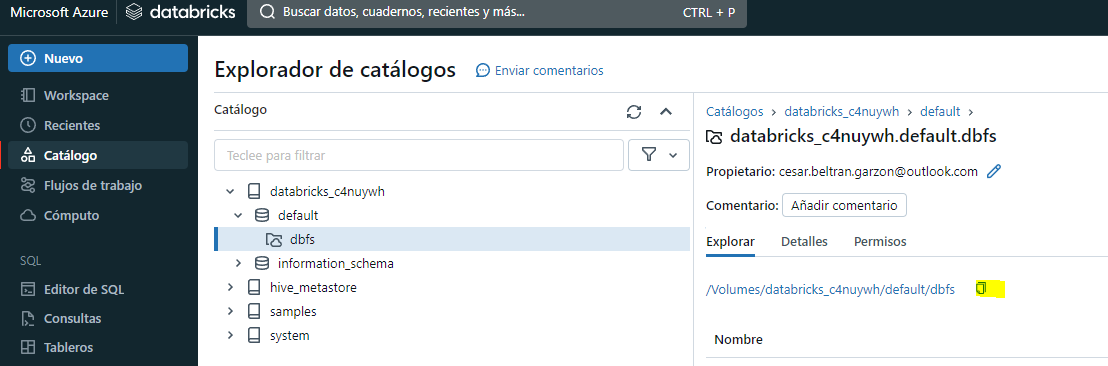
%sh

rm -r /dbfs/delta\_lab

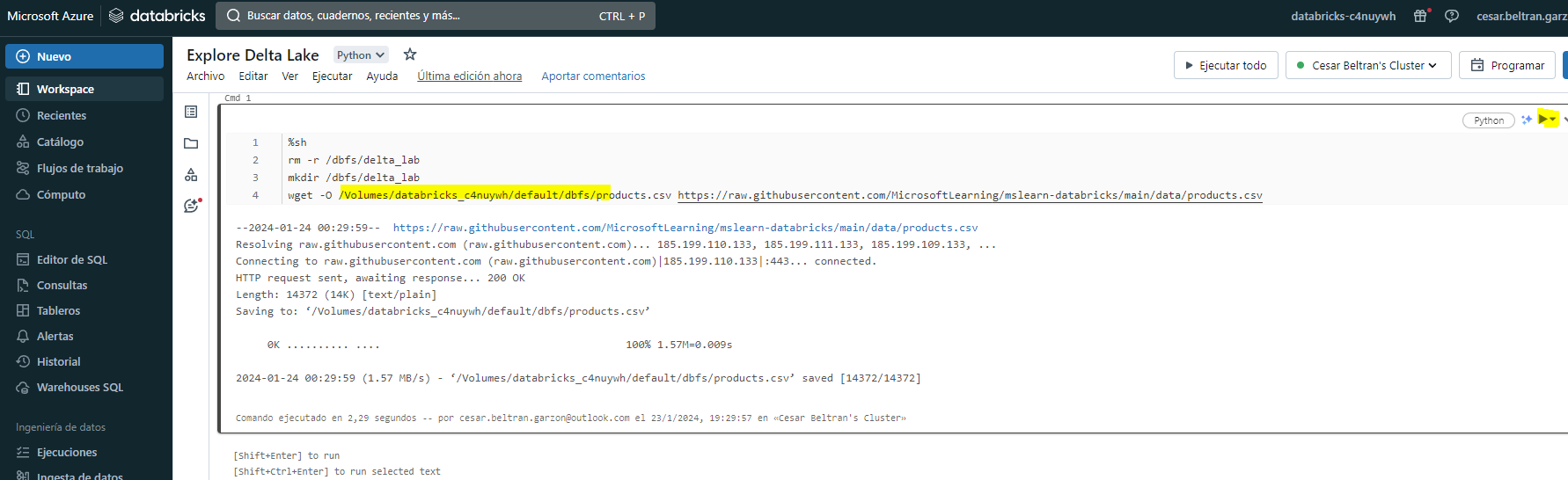
mkdir /dbfs/delta\_lab

wget -O /dbfs/delta\_lab/products.csv https://raw.githubusercontent.com/MicrosoftLearning/mslearn-databricks/main/data/products.csv





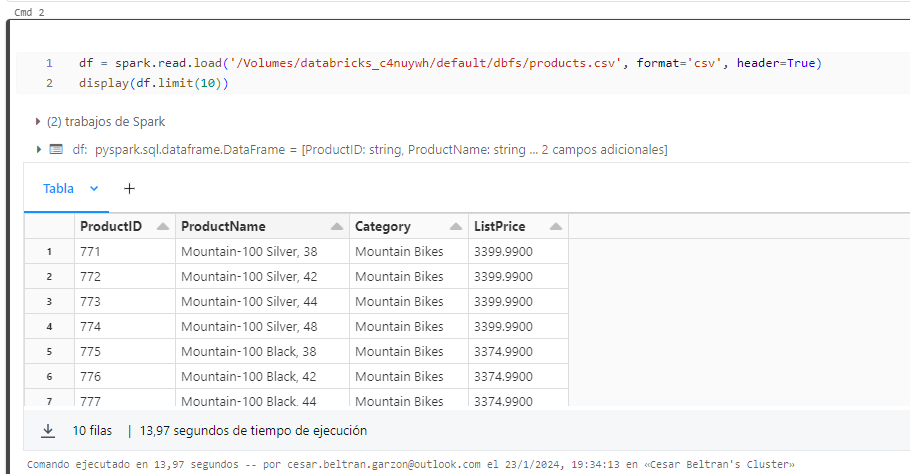
1. Use the ****▸ Run Cell**** menu option at the top-right of the following cell to run it. Then wait for the Spark job run by the code to complete.



1. Under the existing code cell, use the ****+**** icon to add a new code cell. Then in the new cell, enter and run the following code to load the data from the file and view the first 10 rows.

df = spark.read.load('/delta\_lab/products.csv', format='csv', header=True)

display(df.limit(10))



## Load the file data into a delta table

The data has been loaded into a dataframe. Let’s persist it into a delta table.

1. Add a new code cell and use it to run the following code:

delta\_table\_path = "/delta/products-delta"

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

The data for a delta lake table is stored in Parquet format. A log file is also created to track modifications made to the data.



1. Add a new code cell and use it to run the following shell commands to view the contents of the folder where the delta data has been saved.

%sh

ls /dbfs/delta/products-delta



1. The file data in Delta format can be loaded into a ****DeltaTable**** object, which you can use to view and update the data in the table. Run the following code in a new cell to update the data; reducing the price of product 771 by 10%.

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

# Create a deltaTable object

deltaTable = DeltaTable.forPath(spark, delta\_table\_path)# Update the table (reduce price of product 771 by 10%)

deltaTable.update(

condition = "ProductID == 771",

set = { "ListPrice": "ListPrice \* 0.9" })# View the updated data as a dataframe

deltaTable.toDF().show(10)

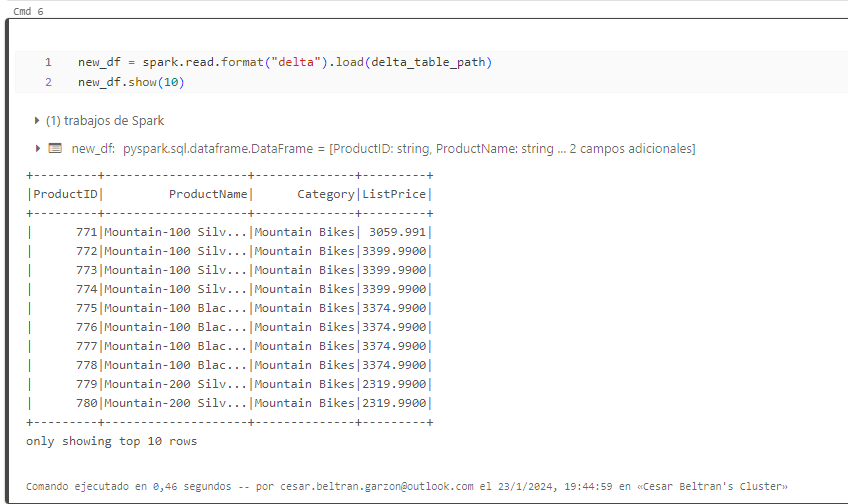
The update is persisted to the data in the delta folder, and will be reflected in any new dataframe loaded from that location.



1. Run the following code to create a new dataframe from the delta table data:

new\_df = spark.read.format("delta").load(delta\_table\_path)

new\_df.show(10)



## Explore logging and time-travel

Data modifications are logged, enabling you to use the time-travel capabilities of Delta Lake to view previous versions of the data.

1. In anew code cell, use the following code to view the original version of the product data:

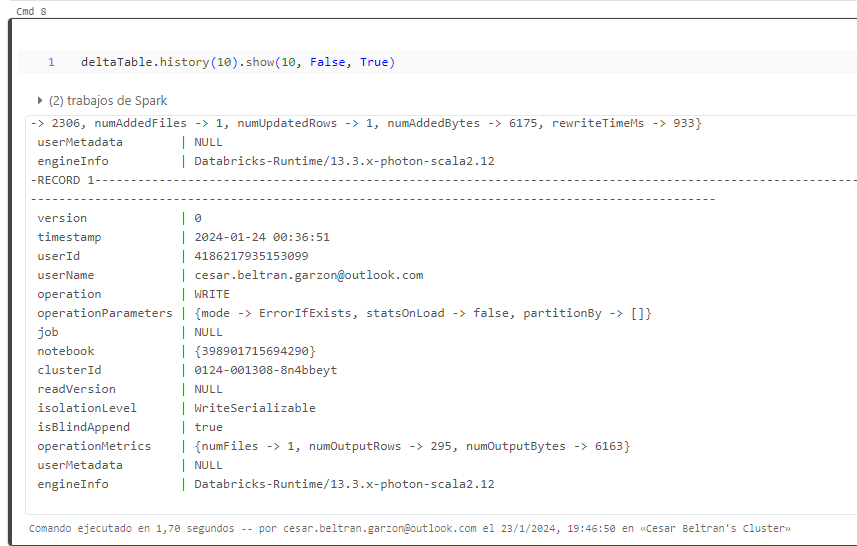
new\_df = spark.read.format("delta").option("versionAsOf", 0).load(delta\_table\_path)

new\_df.show(10)



1. The log contains a full history of modifications to the data. Use the following code to see a record of the last 10 changes:

deltaTable.history(10).show(10, False, True)



## Create catalog tables

So far you’ve worked with delta tables by loading data from the folder containing the parquet files on which the table is based. You can define catalog tables that encapsulate the data and provide a named table entity that you can reference in SQL code. Spark supports two kinds of catalog tables for delta lake:

* External tables that are defined by the path to the parquet files containing the table data.
* Managed tables, that are defined in the Hive metastore for the Spark cluster.

### Create an external table

1. Use the following code to create a new database named ****AdventureWorks**** and then creates an external tabled named ****ProductsExternal**** in that database based on the path to the Delta files you defined previously:

spark.sql("CREATE DATABASE AdventureWorks")

spark.sql("CREATE TABLE AdventureWorks.ProductsExternal USING DELTA LOCATION '{0}'".format(delta\_table\_path))

spark.sql("DESCRIBE EXTENDED AdventureWorks.ProductsExternal").show(truncate=False)

Note that the ****Location**** property of the new table is the path you specified.

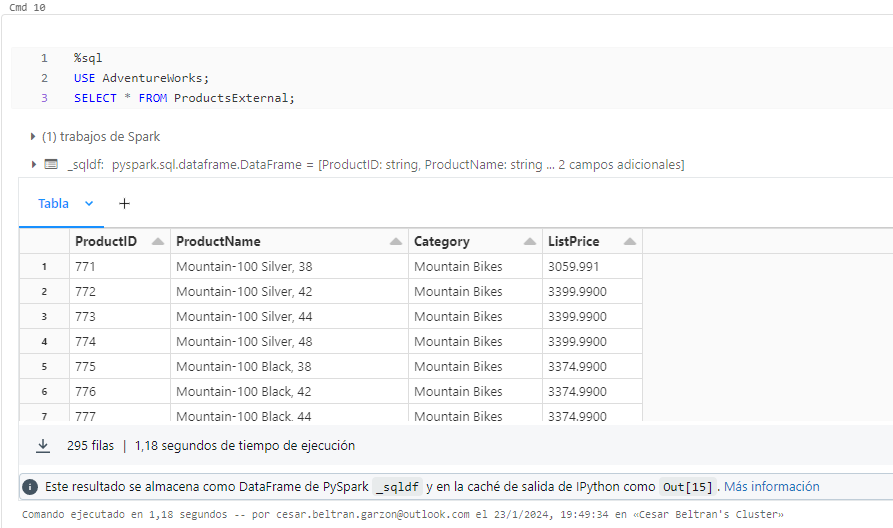


1. Use the following code to query the table:

%sql

USE AdventureWorks;

SELECT \* FROM ProductsExternal;



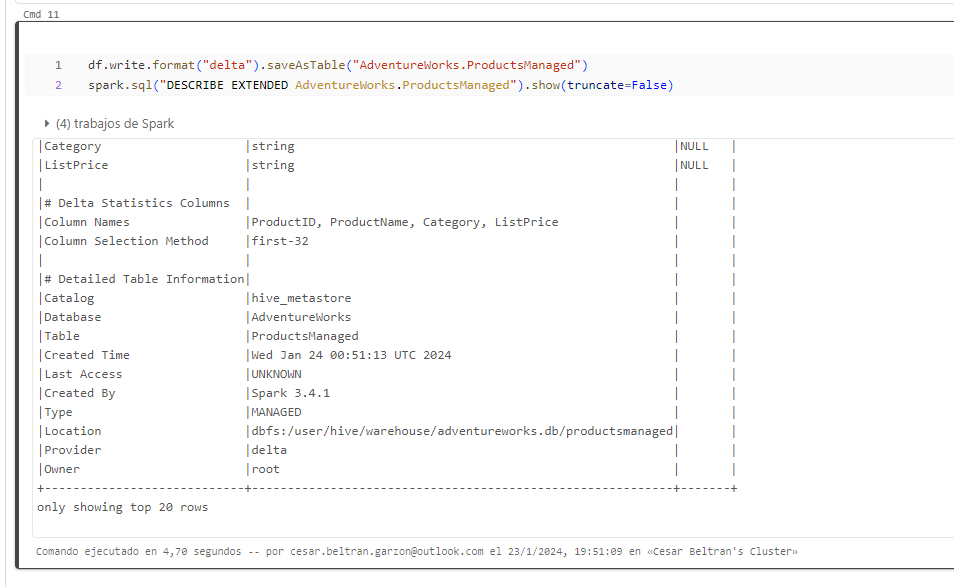
### Create a managed table

1. Run the following code to create (and then describe) a managed tabled named ****ProductsManaged**** based on the dataframe you originally loaded from the ****products.csv**** file (before you updated the price of product 771).

df.write.format("delta").saveAsTable("AdventureWorks.ProductsManaged")

spark.sql("DESCRIBE EXTENDED AdventureWorks.ProductsManaged").show(truncate=False)

You did not specify a path for the parquet files used by the table - this is managed for you in the Hive metastore, and shown in the ****Location**** property in the table description (in the ****dbfs:/user/hive/warehouse/**** path).

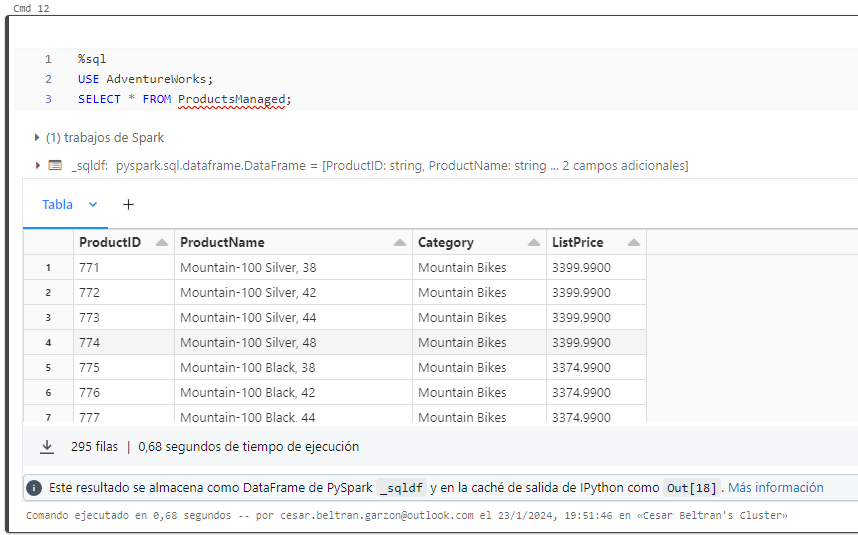


1. Use the following code to query the managed table, noting that the syntax is just the same as for a managed table:

%sql

USE AdventureWorks;

SELECT \* FROM ProductsManaged;



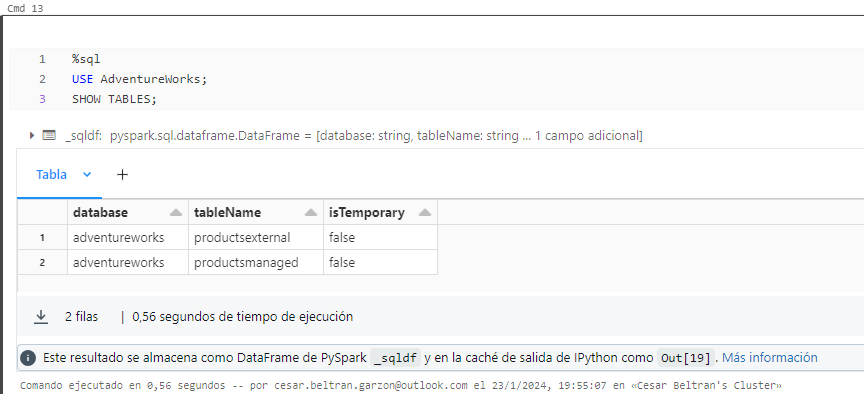
### Compare external and managed tables

1. Use the following code to list the tables in the ****AdventureWorks**** database:

%sql

USE AdventureWorks;

SHOW TABLES;



1. Now use the following code to see the folders on which these tables are based:

%sh

echo "External table:"

ls /dbfs/delta/products-delta

echo

echo "Managed table:"

ls /dbfs/user/hive/warehouse/adventureworks.db/productsmanaged



1. Use the following code to delete both tables from the database:

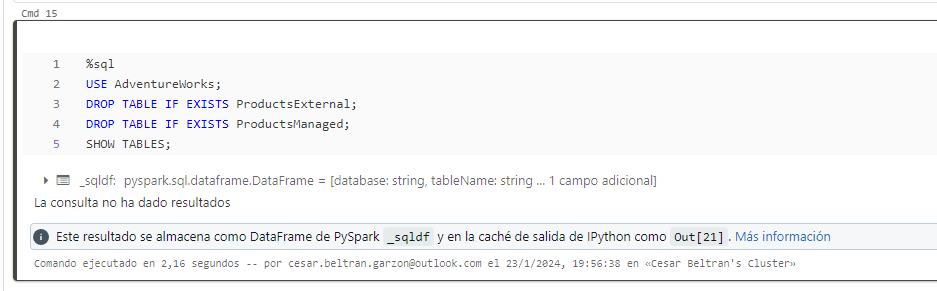
%sql

USE AdventureWorks;

DROP TABLE IF EXISTS ProductsExternal;

DROP TABLE IF EXISTS ProductsManaged;

SHOW TABLES;



1. Now rerun the cell containing the following code to view the contents of the delta folders:

%sh

echo "External table:"

ls /dbfs/delta/products-delta

echo

echo "Managed table:"

ls /dbfs/user/hive/warehouse/adventureworks.db/productsmanaged

The files for the managed table are deleted automatically when the table is dropped. However, the files for the external table remain. Dropping an external table only removes the table metadata from the database; it does not delete the data files.



1. Use the following code to create a new table in the database that is based on the delta files in the ****products-delta**** folder:

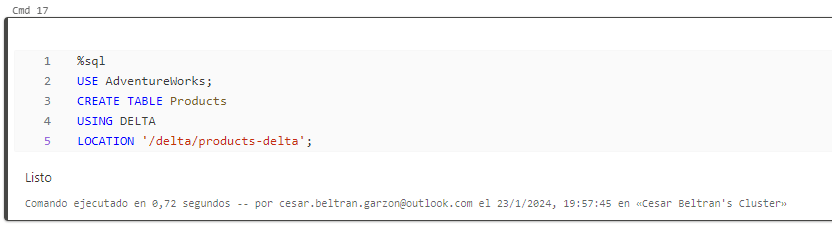
%sql

USE AdventureWorks;

CREATE TABLE Products

USING DELTA

LOCATION '/delta/products-delta';



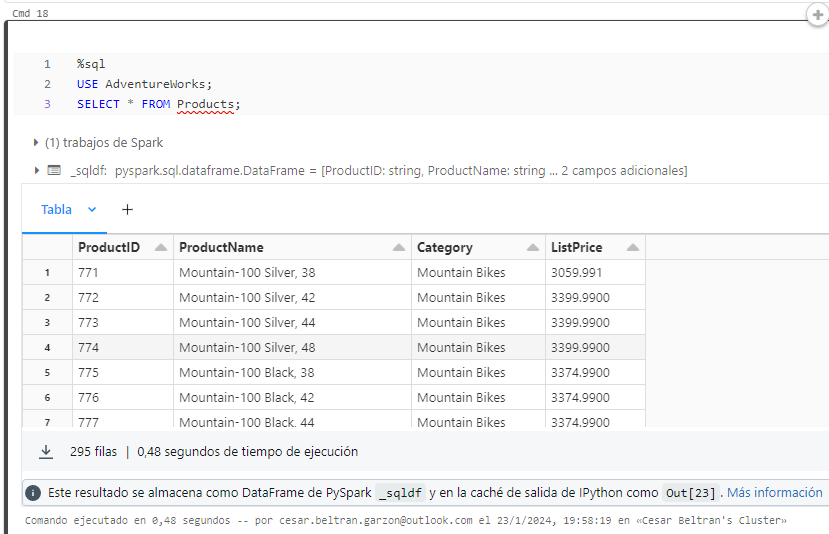
1. Use the following code to query the new table:

%sql

USE AdventureWorks;

SELECT \* FROM Products;

Because the table is based on the existing delta files, which include the logged history of changes, it reflects the modifications you previously made to the products data.



## Use delta tables for streaming data

Delta lake supports streaming data. Delta tables can be a sink or a source for data streams created using the Spark Structured Streaming API. In this example, you’ll use a delta table as a sink for some streaming data in a simulated internet of things (IoT) scenario. The simulated device data is in JSON format, like this:

CodeCopy

{"device":"Dev1","status":"ok"}

{"device":"Dev1","status":"ok"}

{"device":"Dev1","status":"ok"}

{"device":"Dev2","status":"error"}

{"device":"Dev1","status":"ok"}

{"device":"Dev1","status":"error"}

{"device":"Dev2","status":"ok"}

{"device":"Dev2","status":"error"}

{"device":"Dev1","status":"ok"}

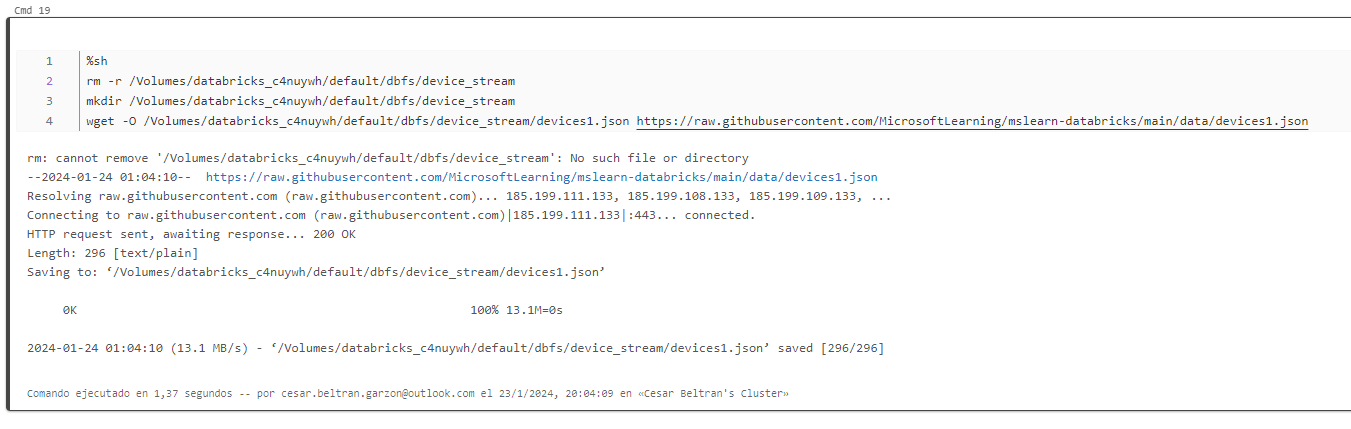
1. In a new cell, run the following code to download the JSON file:

%sh

rm -r /dbfs/device\_stream

mkdir /dbfs/device\_stream

wget -O /dbfs/device\_stream/devices1.json https://raw.githubusercontent.com/MicrosoftLearning/mslearn-databricks/main/data/devices1.json



1. In a new cell, run the following code to create a stream based on the folder containing the JSON device data:

from pyspark.sql.types import \*

from pyspark.sql.functions import \*

# Create a stream that reads data from the folder, using a JSON schema

inputPath = '/device\_stream/'

jsonSchema = StructType([

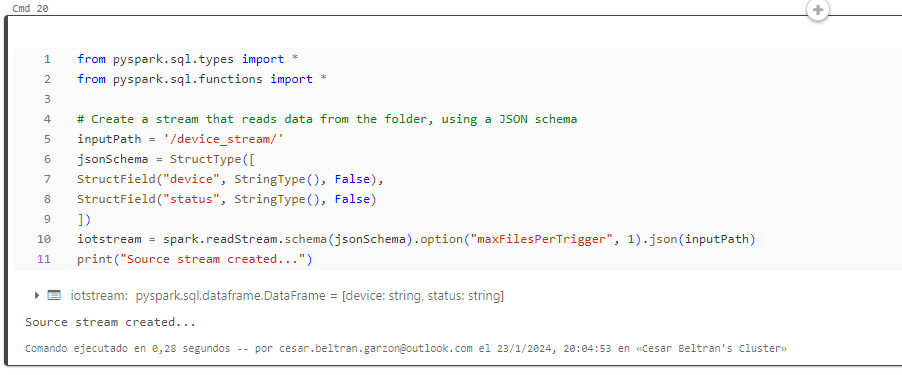
StructField("device", StringType(), False),

StructField("status", StringType(), False)

])

iotstream = spark.readStream.schema(jsonSchema).option("maxFilesPerTrigger", 1).json(inputPath)

print("Source stream created...")



1. Add a new code cell and use it to perpetually write the stream of data to a delta folder:

# Write the stream to a delta table

delta\_stream\_table\_path = '/delta/iotdevicedata'

checkpointpath = '/delta/checkpoint'

deltastream = iotstream.writeStream.format("delta").option("checkpointLocation", checkpointpath).start(delta\_stream\_table\_path)

print("Streaming to delta sink...")



1. Add code to read the data, just like any other delta folder:

# Read the data in delta format into a dataframe

df = spark.read.format("delta").load(delta\_stream\_table\_path)

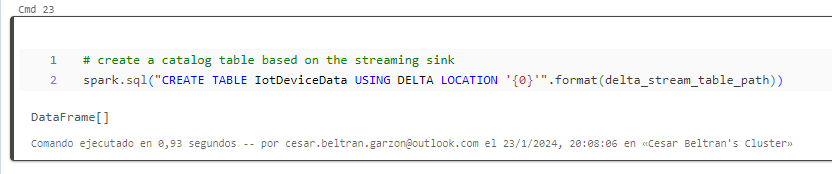
display(df)



1. Add the following code to create a table based on the delta folder to which the streaming data is being written:

# create a catalog table based on the streaming sink

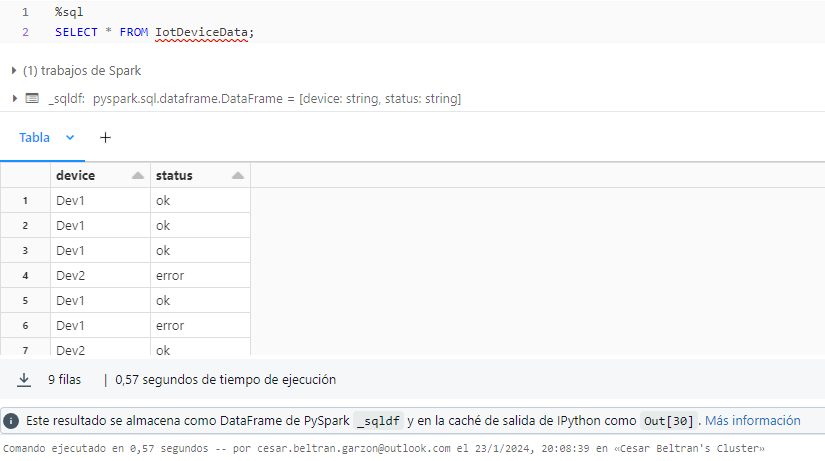
spark.sql("CREATE TABLE IotDeviceData USING DELTA LOCATION '{0}'".format(delta\_stream\_table\_path))



1. Use the following code to query the table:

%sql

SELECT \* FROM IotDeviceData;



1. Run the following code to add some fresh device data to the stream:

%sh

wget -O /dbfs/device\_stream/devices2.json https://raw.githubusercontent.com/MicrosoftLearning/mslearn-databricks/main/data/devices2.json



1. Re-run the following SQL query code to verify that the new data has been added to the stream and written to the delta folder:

%sql

SELECT \* FROM IotDeviceData;



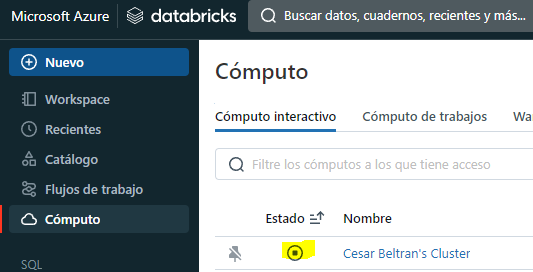
1. Run the following code to stop the stream:

deltastream.stop()



## Clean up

In Azure Databricks portal, on the ****Compute**** page, select your cluster and select ****■ Terminate**** to shut it down.



If you’ve finished exploring Azure Databricks, you can delete the resources you’ve created to avoid unnecessary Azure costs and free up capacity in your subscription.