SQL LIBRARY

PYTHON LIBRARY

Dbutils.widgets library

For Python graphics library use the below ones:

MATPLOTLIB

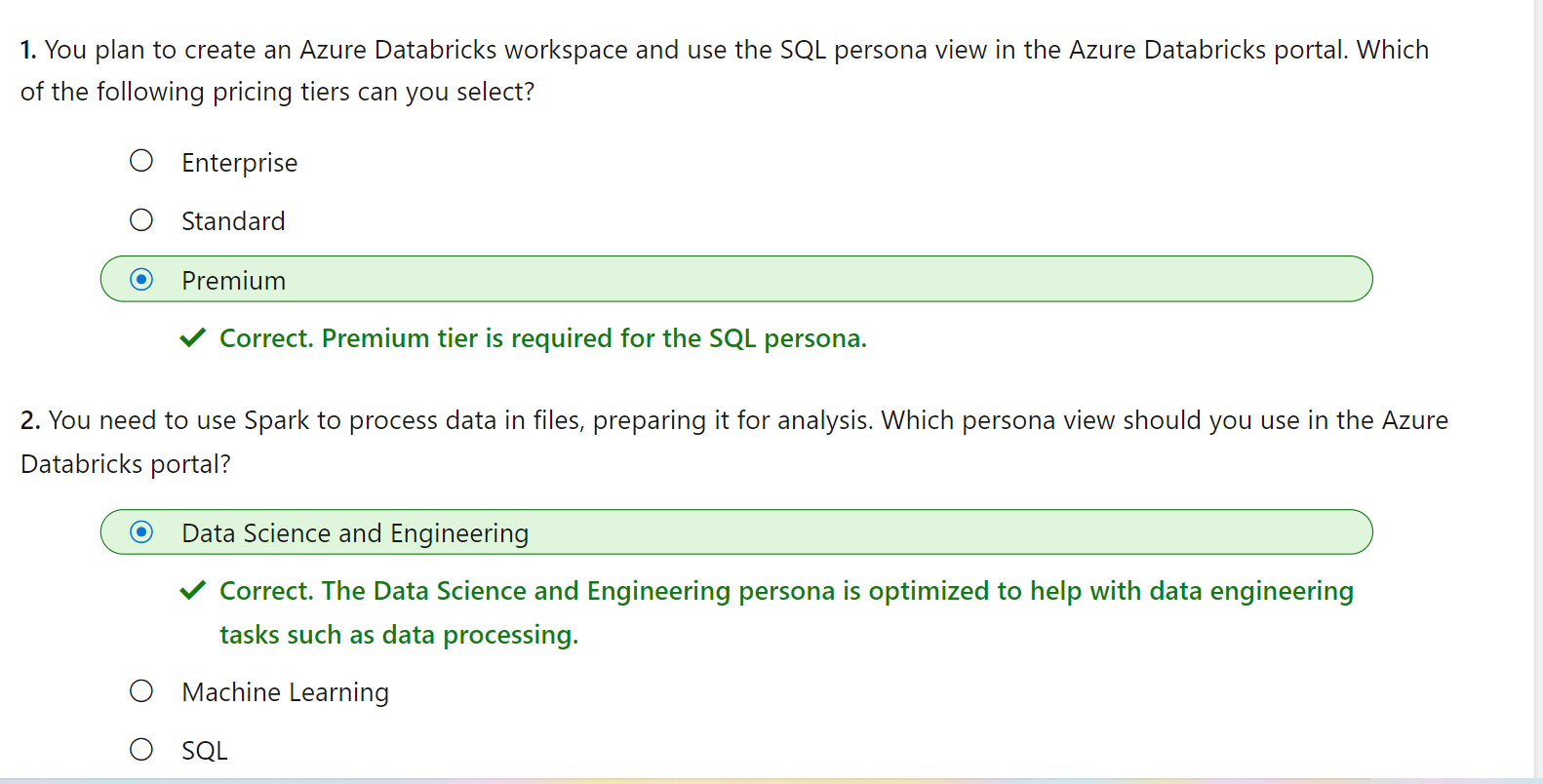
SEABORN LIBRARY

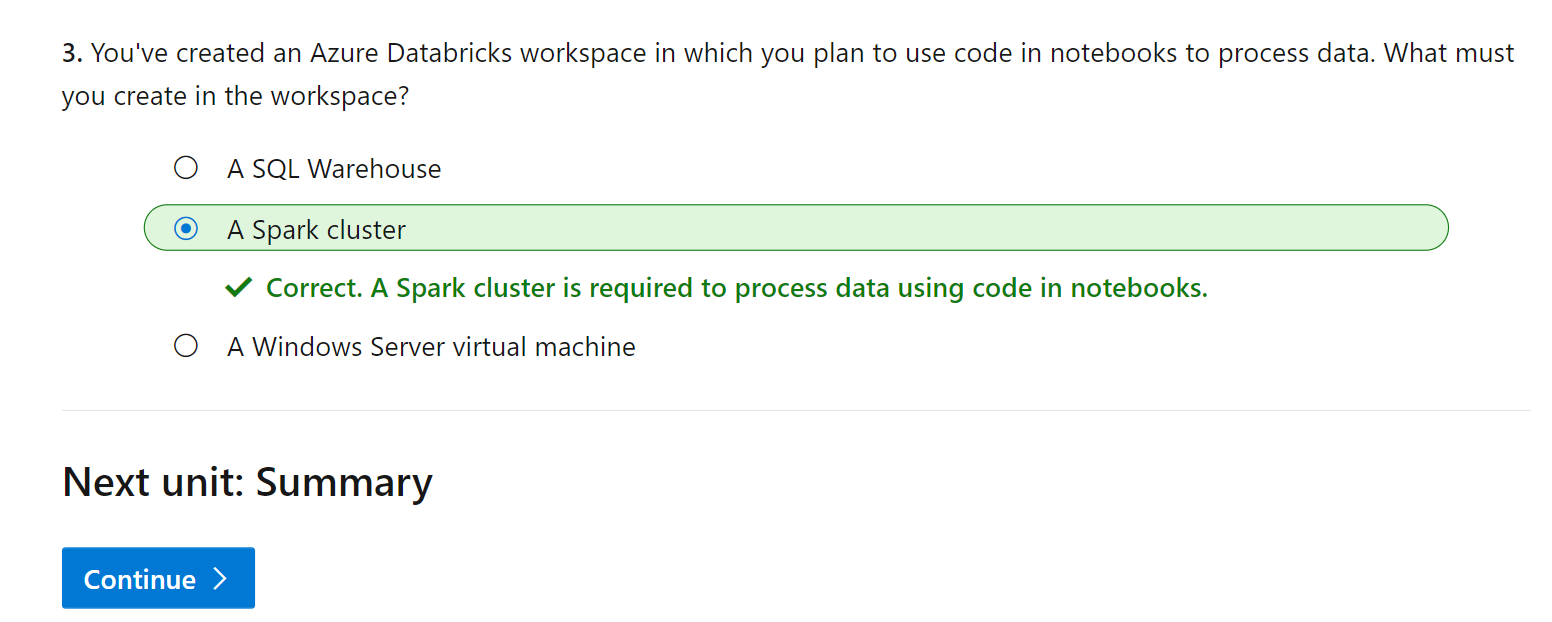
Diff bw data lake house and delta lake

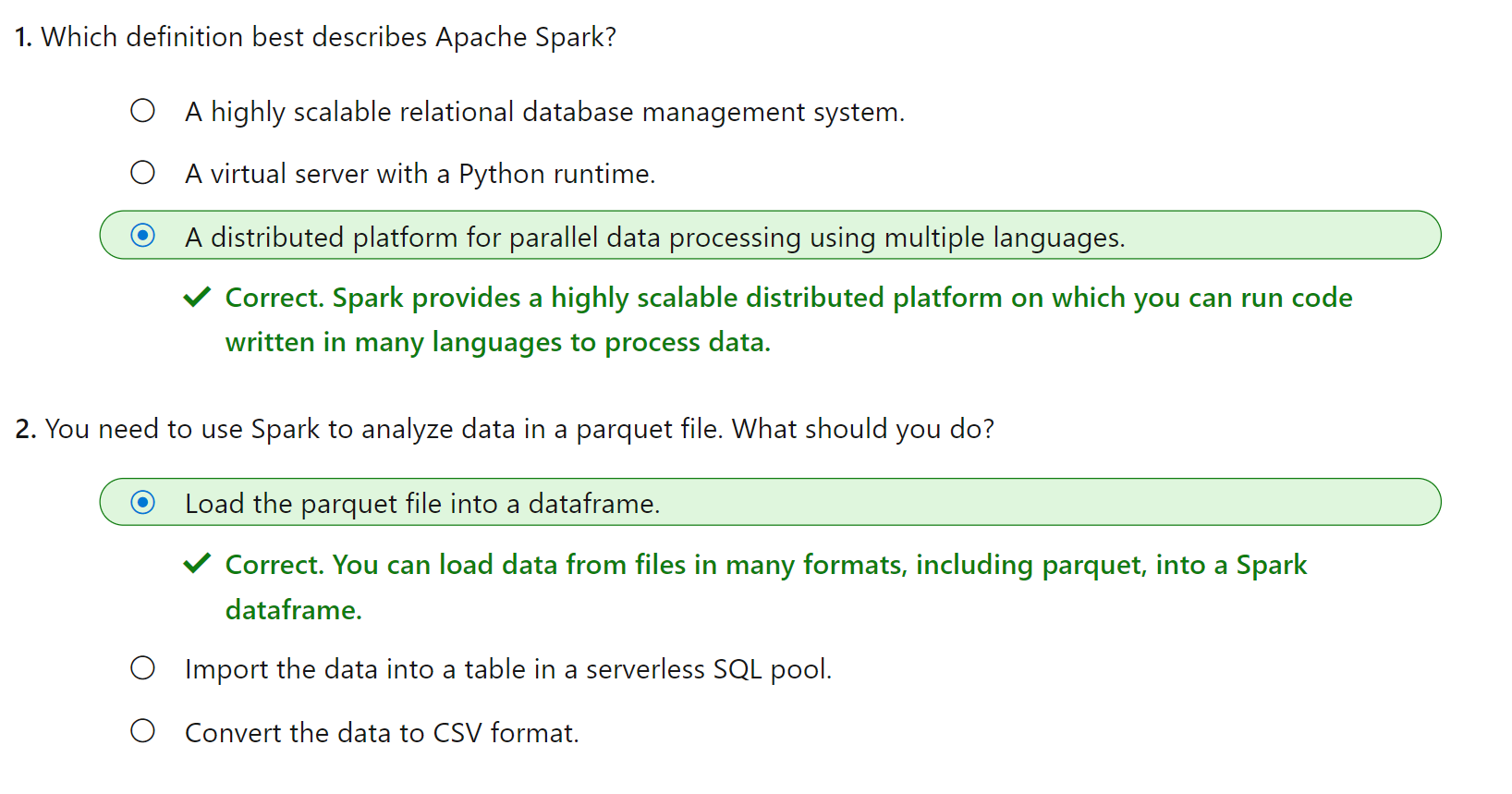
[Data Warehouse vs. Data Lake vs. Data Lakehouse: An Overview of Three Cloud Data Storage Patterns | Striim](https://www.striim.com/blog/data-warehouse-vs-data-lake-vs-data-lakehouse-an-overview/)

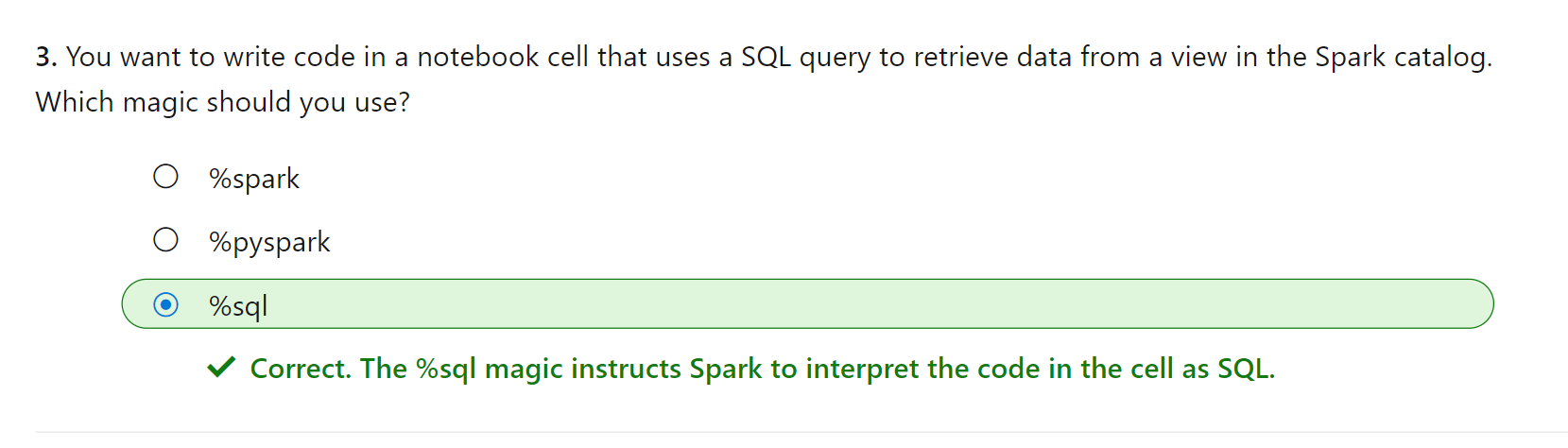
[Data Lake vs Warehouse vs Data Lakehouse | Know the Difference (xenonstack.com)](https://www.xenonstack.com/insights/data-lake-vs-warehouse-vs-data-lakehouse)

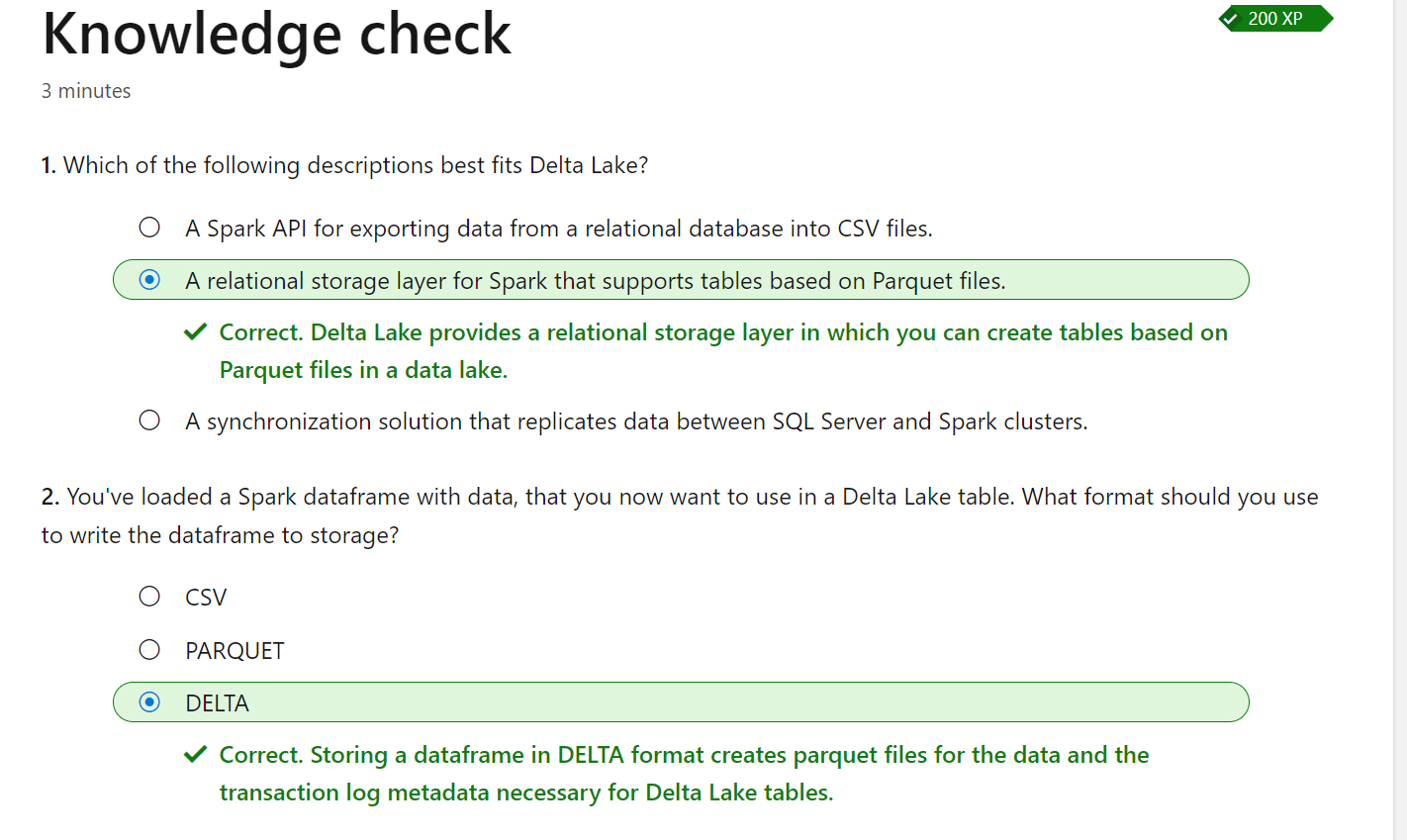
[Simplify Your Lakehouse Architecture with Azure Databricks, Delta Lake, and Azure Data Lake Storage - Microsoft Tech Community](https://techcommunity.microsoft.com/t5/analytics-on-azure-blog/simplify-your-lakehouse-architecture-with-azure-databricks-delta/ba-p/2027272)

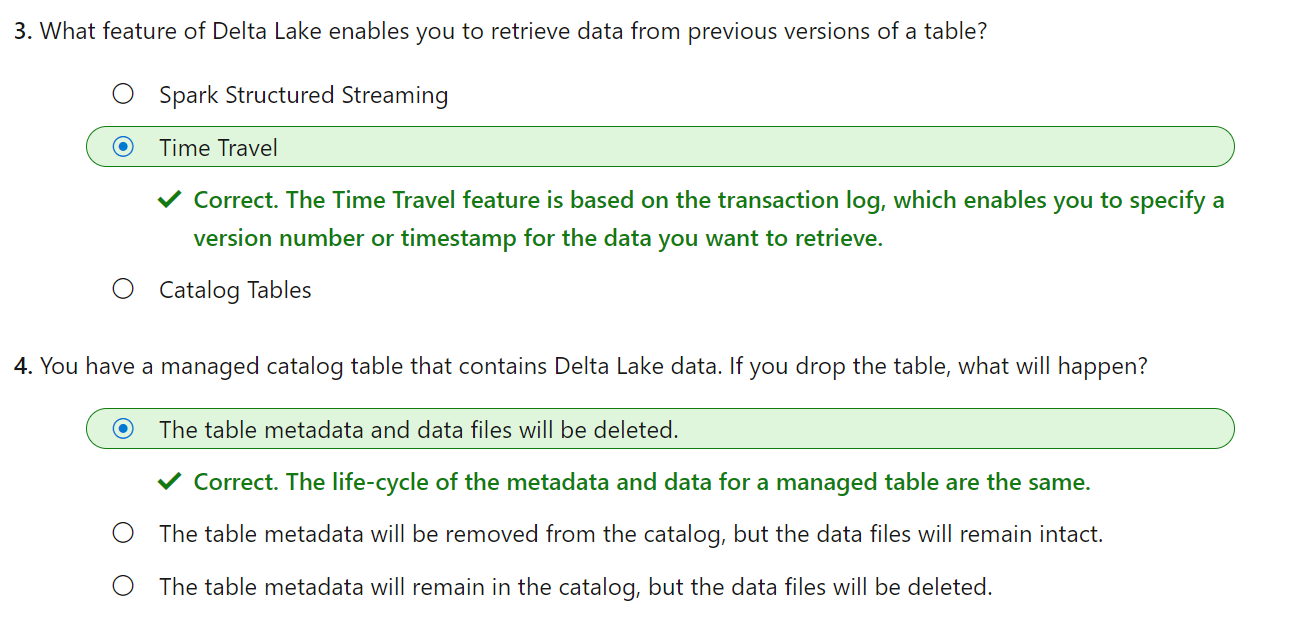


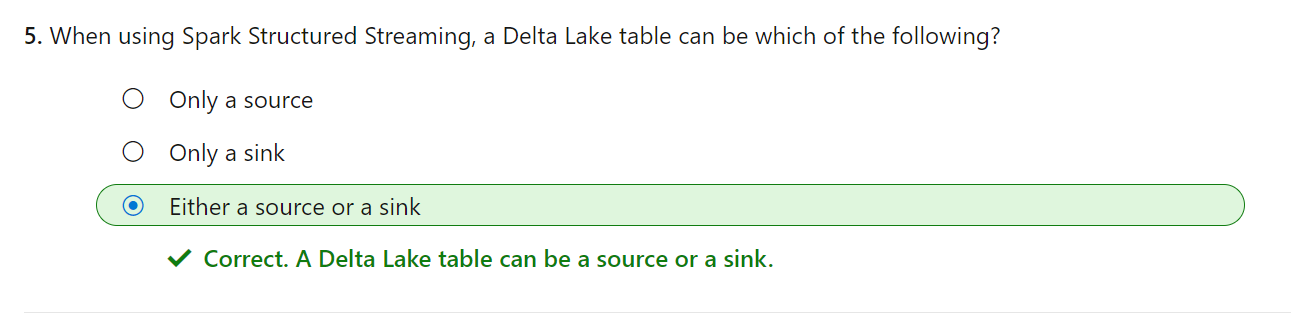


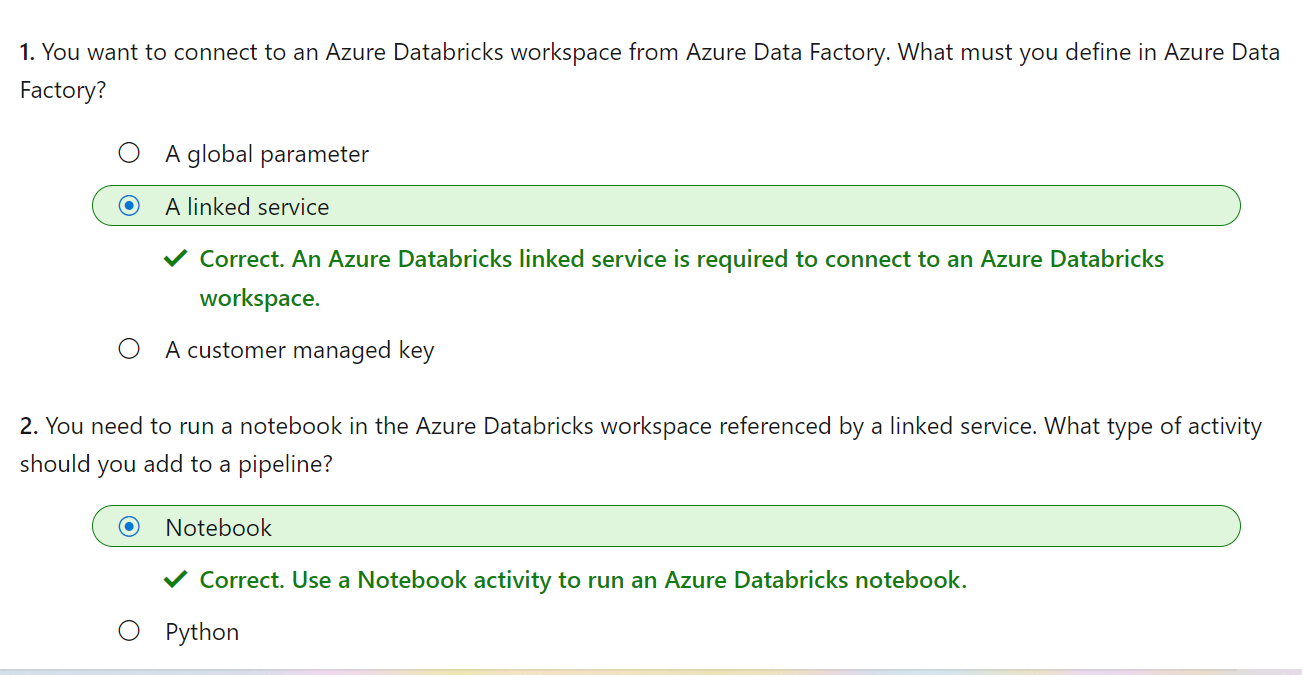


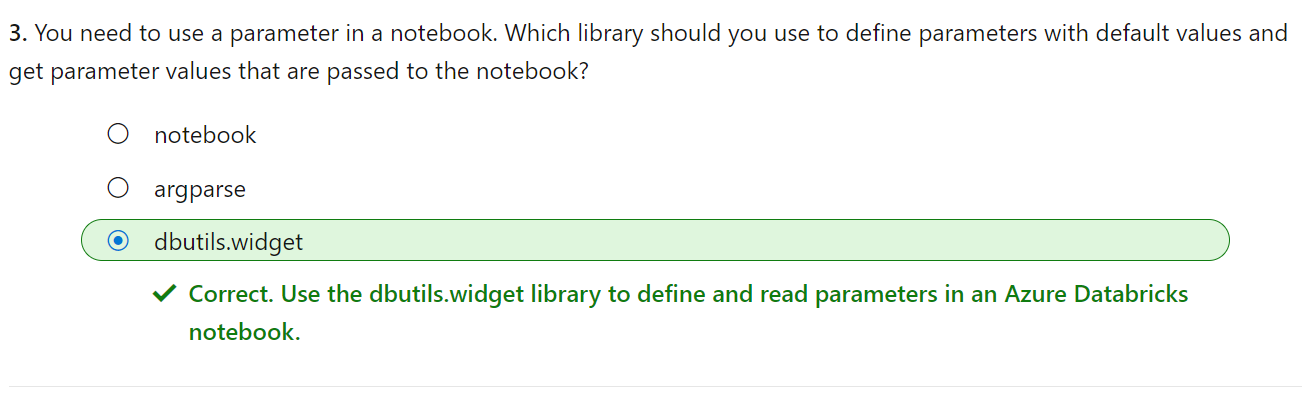


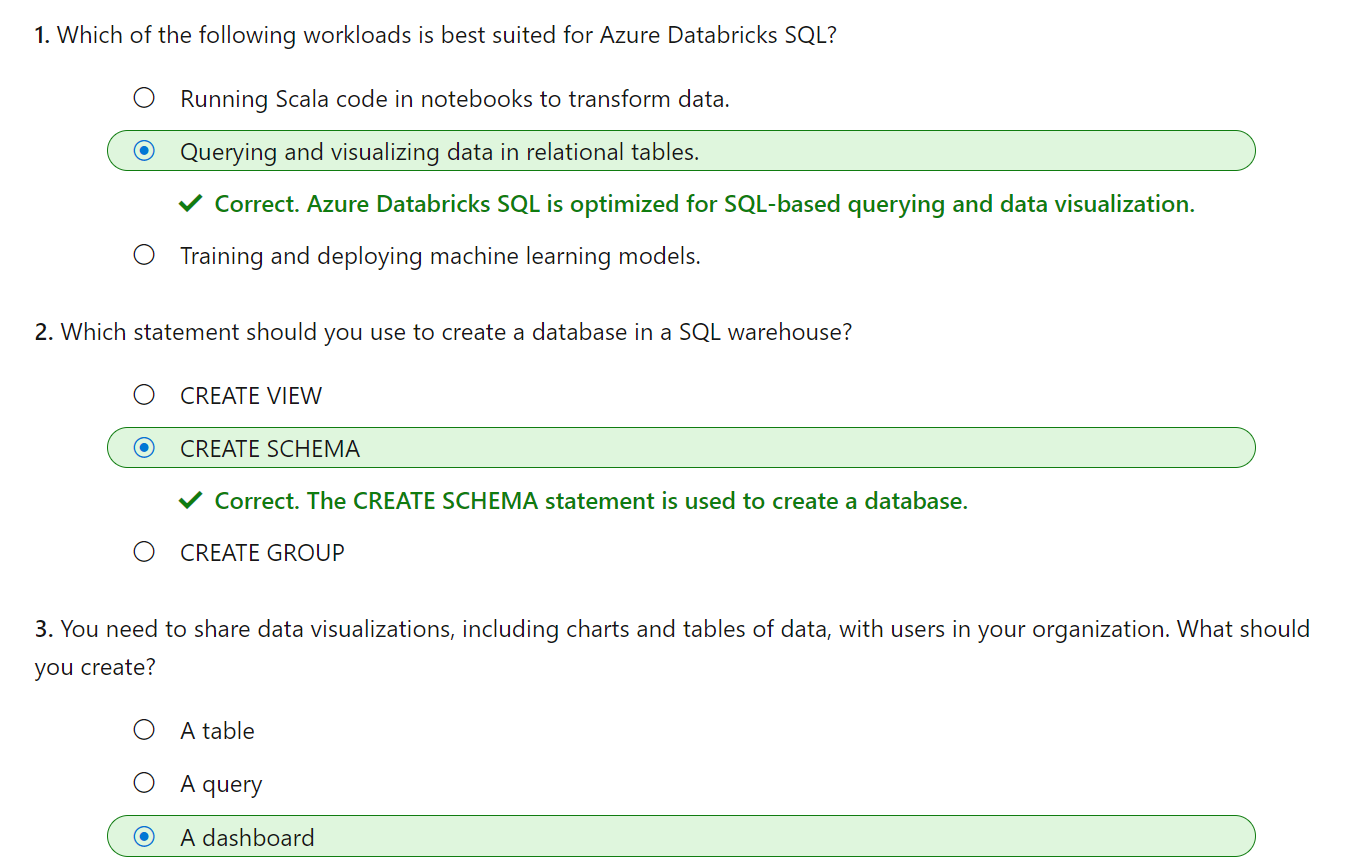












SQL Warehouses are only available in premium Azure Databricks workspaces.

Azure Databricks supports machine learning workloads that involve data exploration and preparation, training and evaluating machine learning models, and serving models to generate predictions for applications and analyses. Data scientists and ML engineers can use AutoML to quickly train predictive models, or apply their skills with common machine learning frameworks such as SparkML, Scikit-Learn, PyTorch, and Tensorflow. They can also manage the end-to-end machine learning lifecycle with MLFlow.

**Data Bricks File System (DBFS):**

DBFS is a shared, distributed file system in which they can access and operate on data files. DBFS enables you to mount storage use it to work with the persist file-based data

**HIVE METASTORE:**

Every Azure Databricks deployment has a central Hive metastore accessible by all clusters to persist table metadata. Instead of using the Azure Databricks Hive metastore, you have the option to use an existing external Hive metastore instance

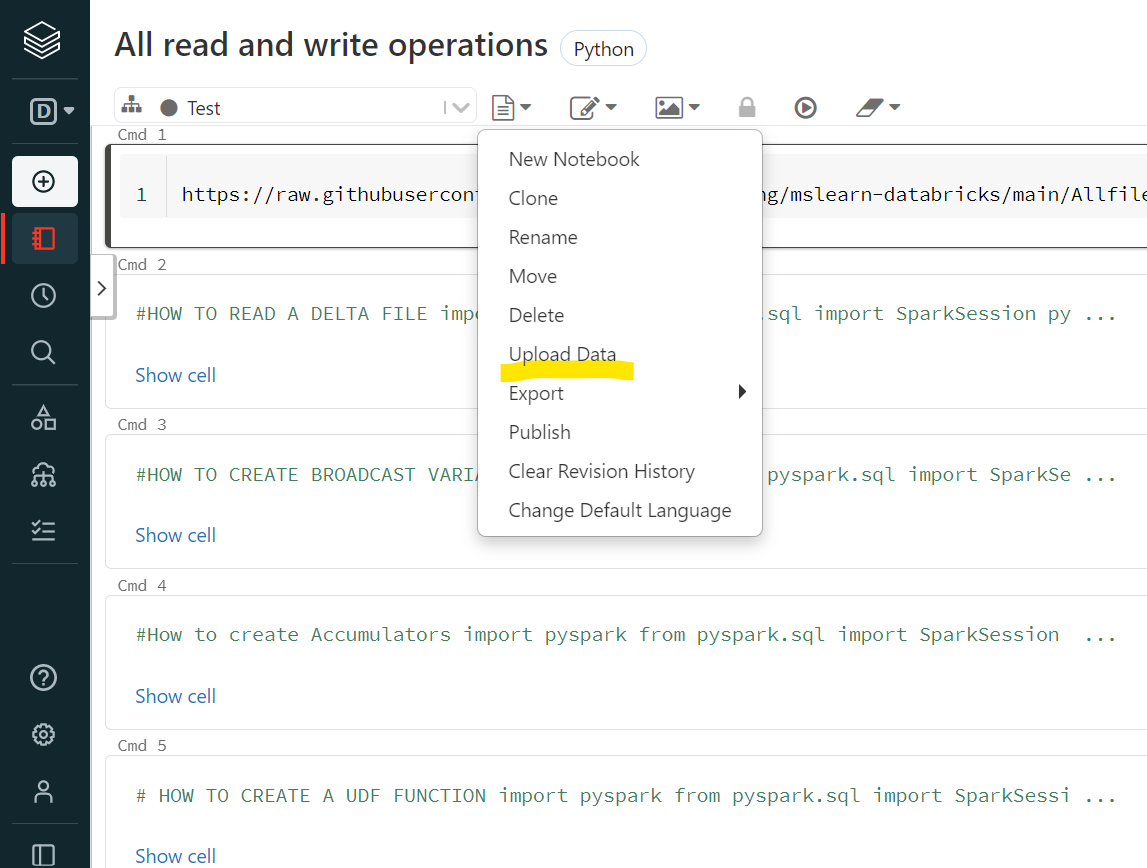
To set up external HIVE meta store in Databricks, then follow this page:

[External Apache Hive metastore - Azure Databricks | Microsoft Learn](https://learn.microsoft.com/en-us/azure/databricks/data/metastores/external-hive-metastore)

**DELTA LAKE:**

Delta Lake is an [open source storage layer](https://delta.io/) 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. Delta Lake on Azure Databricks allows you to configure Delta Lake based on your workload patterns.

[What is Delta Lake? - Azure Databricks | Microsoft Learn](https://learn.microsoft.com/en-us/azure/databricks/delta/)



To read a file directly instead of uploading it we can use the following command:

df1 = spark.read.format("csv").option("header", "true").load("dbfs:/FileStore/https:/raw.githubusercontent.com/MicrosoftLearning/mslearn-databricks/main/Allfiles/Labs/01/adventureworks/products.csv/products.csv")

<https://raw.githubusercontent.com/MicrosoftLearning/mslearn-databricks/main/Allfiles/Labs/01/adventureworks/products.csv>

Spark parallelizes jobs at two levels:

* The first level of parallelization is the *executor* - a Java virtual machine (JVM) running on a worker node, typically, one instance per node.
* The second level of parallelization is the *slot* - the number of which is determined by the number of cores and CPUs of each node.

When creating the cluster, you can specify configuration settings, including:

* A name for the cluster.
* A *cluster mode*, which can be:
  + *Standard*: Suitable for single-user workloads that require multiple worker nodes.
  + *High Concurrency*: Suitable for workloads where multiple users will be using the cluster concurrently.
  + *Single Node*: Suitable for small workloads or testing, where only a single worker node is required.
* The version of the ***Databricks Runtime***to be used in the cluster; which dictates the version of Spark and individual components such as Python, Scala, and others that get installed.
* The type of virtual machine (VM) used for the worker nodes in the cluster.
* The minimum and maximum number of worker nodes in the cluster.
* The type of VM used for the driver node in the cluster.
* Whether the cluster supports *autoscaling* to dynamically resize the cluster.
* How long the cluster can remain idle before being shut down automatically.

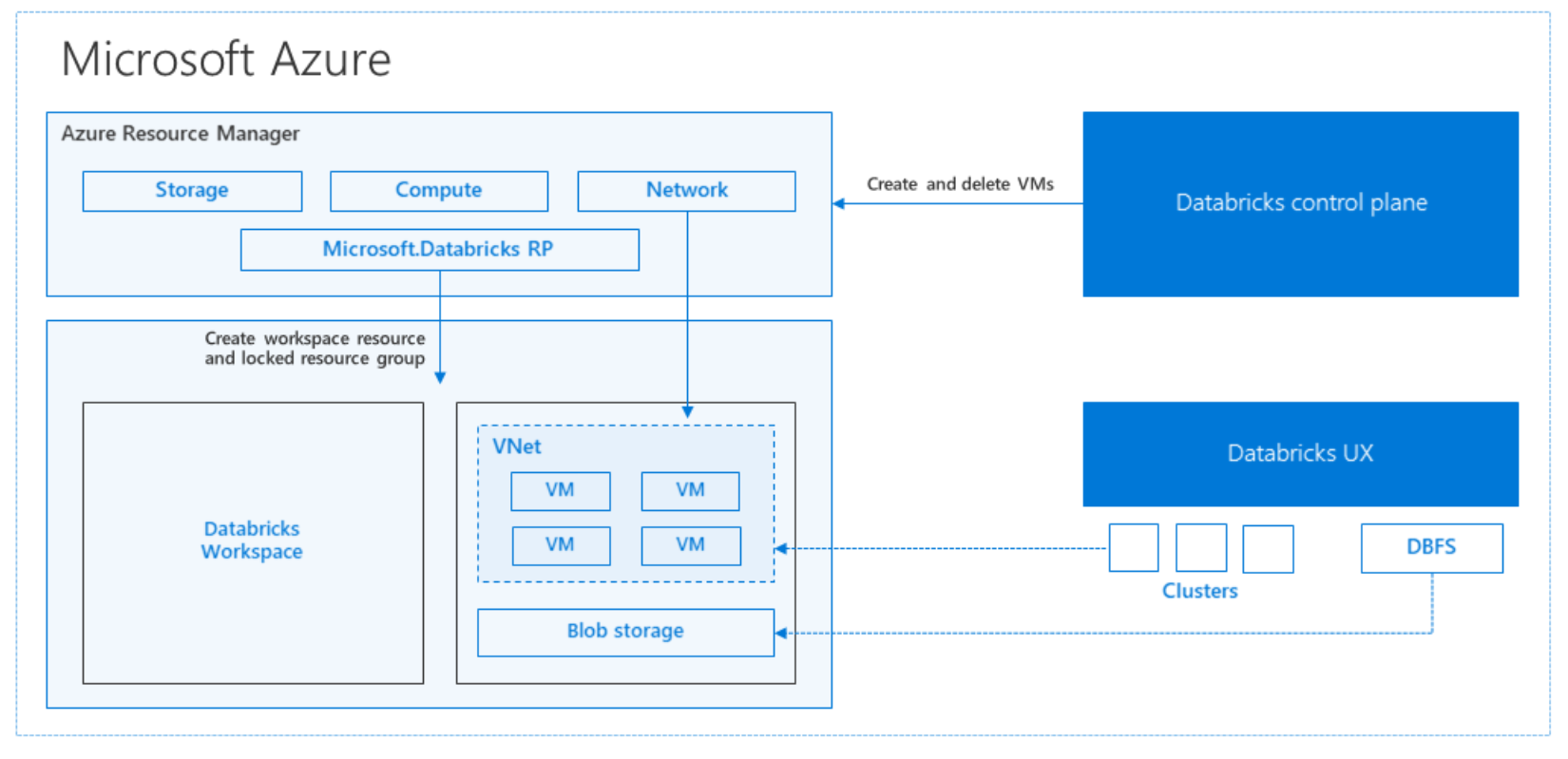
Note:

You also have the option of attaching your cluster to a pool of idle nodes to reduce cluster startup time. For more information, see [**Pools**](https://learn.microsoft.com/en-us/azure/databricks/clusters/instance-pools/) in the Azure Databricks documentation.

When creating a databricks workspace , a databricks appliance is deployed as resource group which manages worker VMs, driver, security groups and storage account. All metadata of your cluster, such as scheduled jobs are stored in a Azure database with geo-replication for fault tolerance.

Azure Kubernetes Service (AKS) is used to run Databricks control-plane and data planes via containers running on the latest generation of hardware (Dv3 VMs), with with NvMe SSDs capable of blazing 100us latency on high-performance Azure virtual machines with accelerated networking.  Azure utilizes these features to improve spark performance. You manage the Databricks clusters, autoscaling, auto-termination through UI.

[Create a Spark cluster - Training | Microsoft Learn](https://learn.microsoft.com/en-us/training/modules/use-apache-spark-azure-databricks/03-spark-cluster)



Dataframe is form SQL Library

Pandas is from Python Library

In addition to the Dataframe API, Spark SQL provides a strongly-typed Dataset API

We won't explore Spark catalog tables in depth in this module, but it's worth taking the time to highlight a few key points:

* You can create an empty table by using the spark.catalog.createTable method. Tables are metadata structures that store their underlying data in the storage location associated with the catalog. Deleting a table also deletes its underlying data.
* You can save a dataframe as a table by using its saveAsTable method.
* You can create an *external* table by using the spark.catalog.createExternalTable method. External tables define metadata in the catalog but get their underlying data from an external storage location; typically a folder in a data lake. Deleting an external table does not delete the underlying data.

Example Code:

import pyspark

from pyspark.sql import SparkSession

spark = SparkSession.builder.master('local[1]').appName('read').getOrCreate()

df = spark.read.load('dbfs:/FileStore/products.csv',format='csv', header= True)

display(df.limit(10))

from pyspark.sql.types import StructField, StructType, IntegerType, StringType, FloatType

from pyspark.sql.functions import \*

productSchema = StructType([

StructField("ProductID", IntegerType()),

StructField("ProductName", StringType()),

StructField("Category", StringType()),

StructField("ListPrice", FloatType())

])

df = spark.read.load('/FileStore/products.csv',format='csv', schema= productSchema, header = True)

display(df.limit(10))

price\_list = df.select("ProductId","ListPrice")

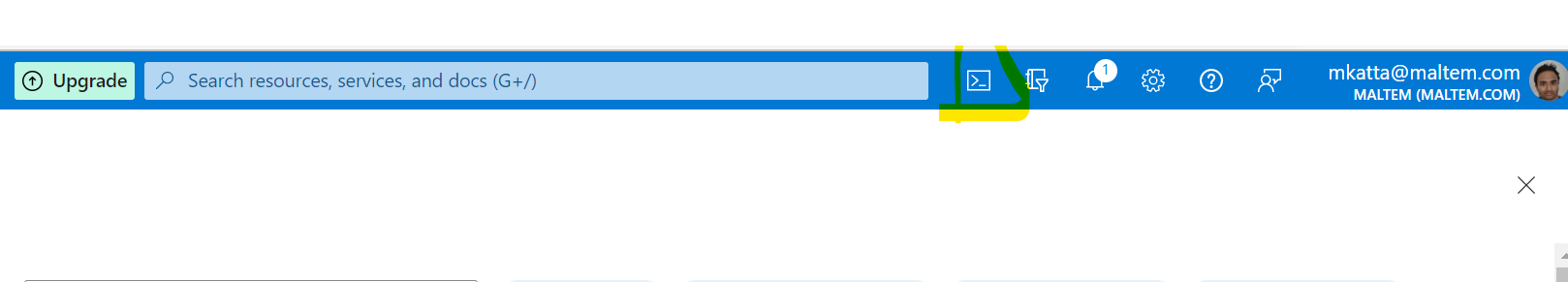
#shorcut Syntax

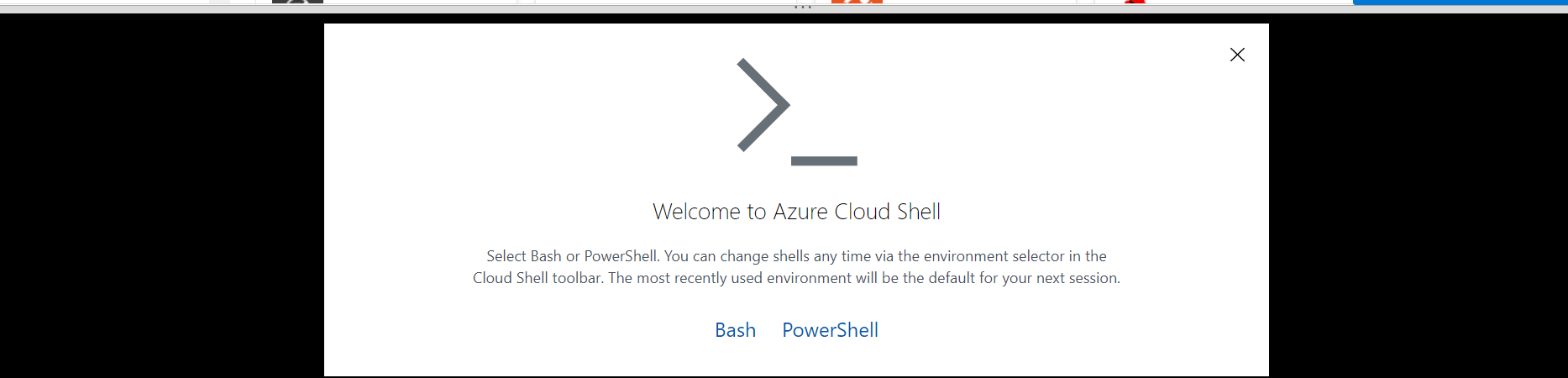
price\_list = df["ProductId","ListPrice"]

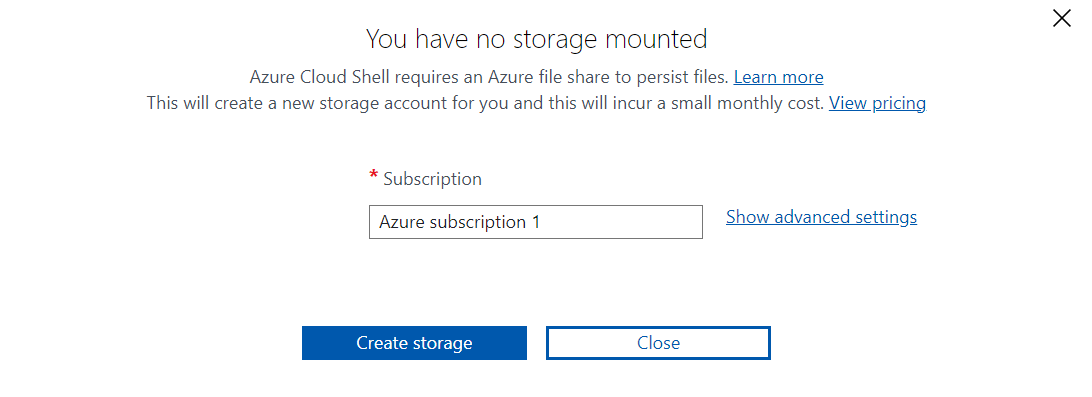
bikes\_df = df.select("ProductName","ListPrice").where((df["Category"]== "Mountain Bikes") | (df["Category"]== "Road Bikes"))

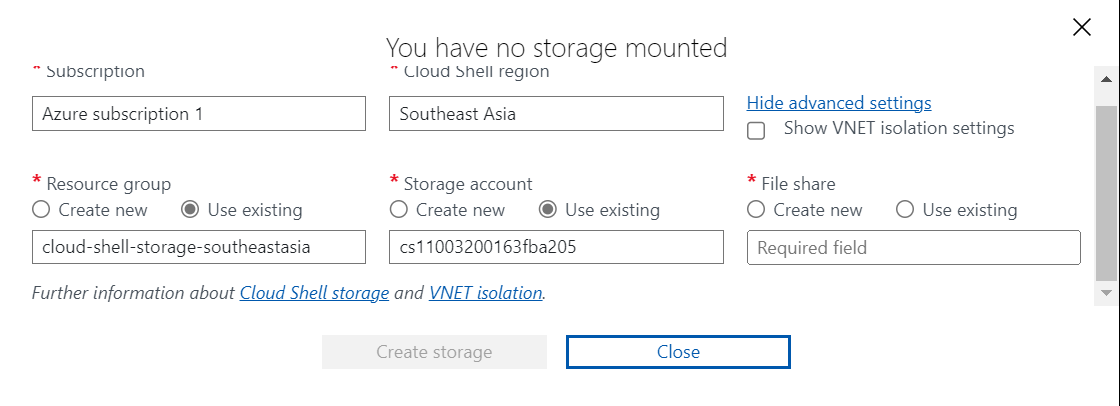
display(bikes\_df.limit(4))

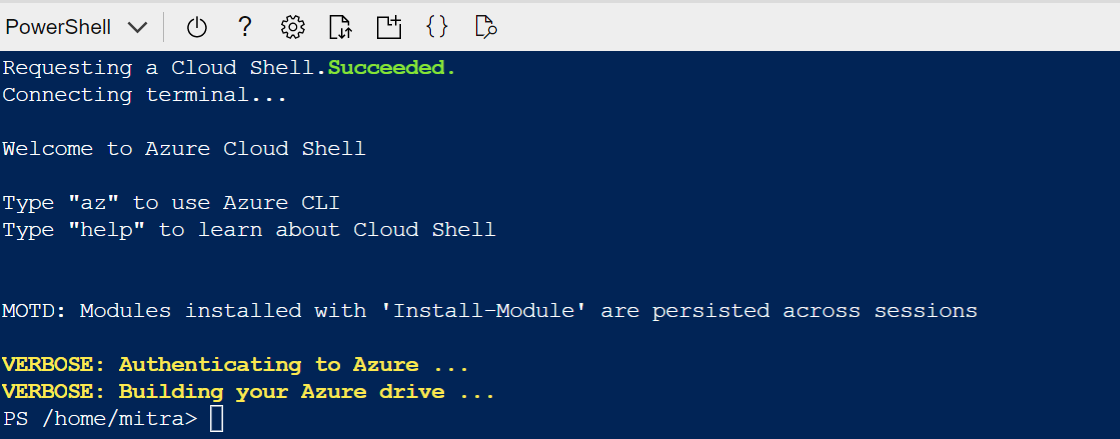
To select the cloud shell which is at the top right next to search bar











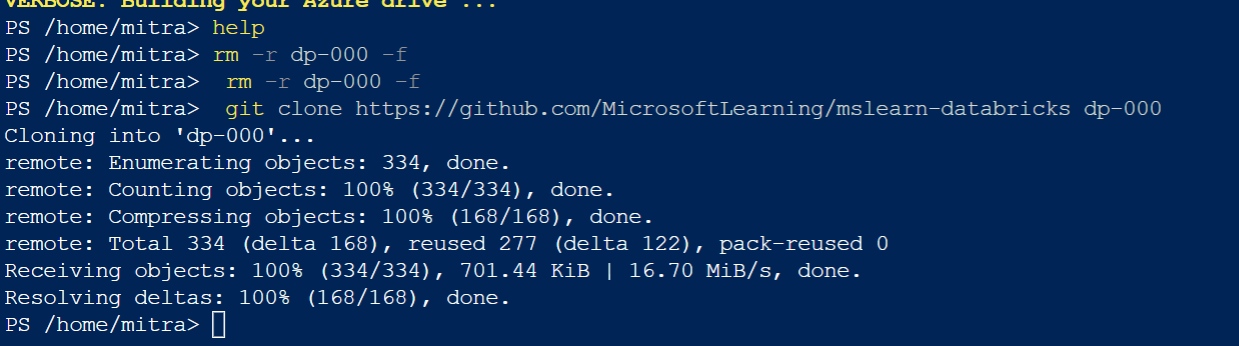
Azure storage firewall is not supported for cloud shell storage accounts.

* Cloud Shell runs on a temporary host provided on a per-session, per-user basis
* Cloud Shell times out after 20 minutes without interactive activity
* Cloud Shell requires an Azure file share to be mounted
* Cloud Shell uses the same Azure file share for both Bash and PowerShell
* Cloud Shell is assigned one machine per user account
* Cloud Shell persists $HOME using a 5-GB image held in your file share
* Permissions are set as a regular Linux user in Bash

Execute the following commands to clone the git learning repo into cloud shell:

rm -r dp-000 -f

git clone [https://github.com/MicrosoftLearning/mslearn-databricks dp-000](https://github.com/MicrosoftLearning/mslearn-databricks%20dp-000)



cd dp-000/Allfiles/Labs/01

./setup.ps1

df1.write.saveAsTable("products")

%sql

SELECT ProductName, ListPrice

FROM products

WHERE Category = 'Touring Bikes';

IF YOU WANT TO LEARN GRAPHICS USE THIS CHAPTER:

[Visualize data - Training | Microsoft Learn](https://learn.microsoft.com/en-us/training/modules/use-apache-spark-azure-databricks/06-visualize-data)

Linux foundation Delta Lake is an open-source storage layer for Spark that enables relational database capabilities for batch and streaming data. By using Delta Lake, you can implement a data lakehouse architecture in Spark to support SQL\_based data manipulation semantics with support for transactions and schema enforcement. The result is an analytical data store that offers many of the advantages of a relational database system with the flexibility of data file storage in a data lake.

Note:

The version of Delta Lake available in an Azure Databricks cluster depends on the version of the Databricks Runtime being used. The information in this module reflects Delta Lake version 1.2.1, which is installed with Databricks Runtime version 10.5 - 11.0.

Please read this Microsoft blog for the lake house architecture:

[Simplify Your Lakehouse Architecture with Azure Databricks, Delta Lake, and Azure Data Lake Storage - Microsoft Tech Community](https://techcommunity.microsoft.com/t5/analytics-on-azure-blog/simplify-your-lakehouse-architecture-with-azure-databricks-delta/ba-p/2027272)

For the differences between data lake vs data warehouse vs delta lake read this blog:

[Data Lake vs Warehouse vs Data Lakehouse | Know the Difference (xenonstack.com)](https://www.xenonstack.com/insights/data-lake-vs-warehouse-vs-data-lakehouse)

[What is Delta Lake? - Azure Databricks | Microsoft Learn](https://learn.microsoft.com/en-us/azure/databricks/delta/)

The benefits of using Delta Lake in Azure Databricks include:

* **Relational tables that support querying and data modification**. With Delta Lake, you can store data in tables that support *CRUD* (create, read, update, and delete) operations. In other words, you can *select*, *insert*, *update*, and *delete* rows of data in the same way you would in a relational database system.
* **Support for *ACID* transactions**. Relational databases are designed to support transactional data modifications that provide *atomicity* (transactions complete as a single unit of work), *consistency* (transactions leave the database in a consistent state), *isolation* (in-process transactions can't interfere with one another), and *durability* (when a transaction completes, the changes it made are persisted). Delta Lake brings this same transactional support to Spark by implementing a transaction log and enforcing serializable isolation for concurrent operations.
* **Data versioning and *time travel***. Because all transactions are logged in the transaction log, you can track multiple versions of each table row, and even use the *time travel* feature to retrieve a previous version of a row in a query.
* **Support for batch and streaming data**. While most relational databases include tables that store static data, Spark includes native support for streaming data through the Spark Structured Streaming API. Delta Lake tables can be used as both *sinks* (destinations) and *sources* for streaming data.
* **Standard formats and interoperability**. The underlying data for Delta Lake tables is stored in Parquet format, which is commonly used in data lake ingestion pipelines.

Delta lake is built on tables which provides a relational storage abstraction over files in a data lake.

Creating a Delta Lake table from a dataframe:

# Load a file into a dataframe

df = spark.read.load('/data/mydata.csv', format='csv', header=True)

# Save the dataframe as a delta table

delta\_table\_path = "/delta/mydata"

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

After saving the delta table, the path location you specified includes parquet files for the data (regardless of the format of the source file you loaded into the dataframe) and a \_delta\_log folder containing the transaction log for the table

**Note:**

The transaction log records all data modifications to the table. By logging each modification, transactional consistency can be enforced and versioning information for the table can be retained.

#overwrite

new\_df.write.format("delta").mode("overwrite").save(delta\_table\_path)

#append

new\_rows\_df.write.format("delta").mode("append").save(delta\_table\_path)

While you can make data modifications in a dataframe and then replace a Delta Lake table by overwriting it, a more common pattern in a database is to insert, update or delete rows in an existing table as discrete transactional operations. To make such modifications to a Delta Lake table, you can use the DeltaTable object in the Delta Lake API, which supports update, delete, and merge operations. For example, you could use the following code to update the price column for all rows with a category column value of "Accessories":

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 accessories by 10%)

deltaTable.update(

condition = "Category == 'Accessories'",

set = { "Price": "Price \* 0.9" })

The data modifications are recorded in the transaction log, and new parquet files are created in the table folder as required.

**Querying a previous version of a table:**

VersionAsOf

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

**OR**

timestampAsOf

df = spark.read.format("delta").option("timestampAsOf", '2022-01-01').load(delta\_table\_path)

Delta lake is built on tables which provides a relational storage abstraction over files in a data lake.

**Creating a Delta Lake table from a dataframe:**

**# Load a file into a dataframe**

df = spark.read.load('/data/mydata.csv', format='csv', header=True)

**# Save the dataframe as a delta table**

delta\_table\_path = "/delta/mydata"

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

After saving the delta table, the path location you specified includes parquet files for the data (regardless of the format of the source file you loaded into the dataframe) and a \_delta\_log folder containing the transaction log for the table.

Note:

The transaction log records all data modifications to the table. By logging each modification, transactional consistency can be enforced and versioning information for the table can be retained.

**#overwrite**

new\_df.write.format("delta").mode("overwrite").save(delta\_table\_path)

**#append**

new\_rows\_df.write.format("delta").mode("append").save(delta\_table\_path)

While you can make data modifications in a dataframe and then replace a Delta Lake table by overwriting it, a more common pattern in a database is to insert, update or delete rows in an existing table as discrete transactional operations. To make such modifications to a Delta Lake table, you can use the DeltaTable object in the Delta Lake API, which supports update, delete, and merge operations. For example, you could use the following code to update the price column for all rows with a category column value of "Accessories":

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 accessories by 10%)**

deltaTable.update(

condition = "Category == 'Accessories'",

set = { "Price": "Price \* 0.9" })

The data modifications are recorded in the transaction log, and new parquet files are created in the table folder as required.

**Querying a previous version of a table:**

**VersionAsOf**

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

**OR**

**timestampAsOf**

df = spark.read.format("delta").option("timestampAsOf", '2022-01-01').load(delta\_table\_path)

**Using the DeltaTableBuilder API**

You can use the DeltaTableBuilder API (part of the Delta Lake API) to create a catalog table, as shown in the following example:

from delta.tables import \*

DeltaTable.create(spark) \

.tableName("default.ManagedProducts") \

.addColumn("Productid", "INT") \

.addColumn("ProductName", "STRING") \

.addColumn("Category", "STRING") \

.addColumn("Price", "FLOAT") \

.execute()

**I AM NOT CONCENTRATING ON STREAMING TOPICS FOR NOW BUT NOT OBSOLETE:**

[Use Delta Lake for streaming data - Training | Microsoft Learn](https://learn.microsoft.com/en-us/training/modules/use-delta-lake-azure-databricks/05-use-delta-lake-streaming-data)

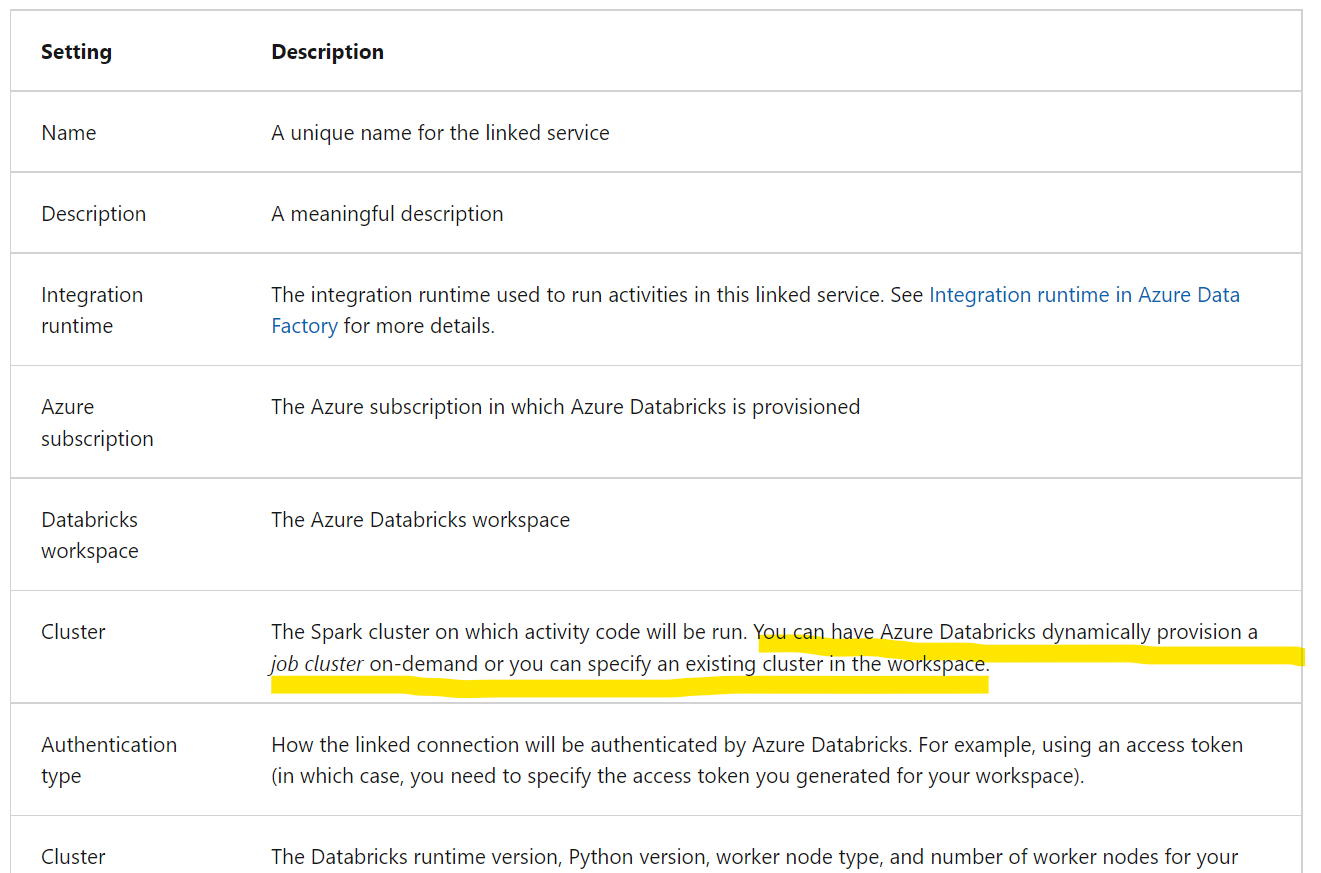
**Create linked service for Azure Databricks:**

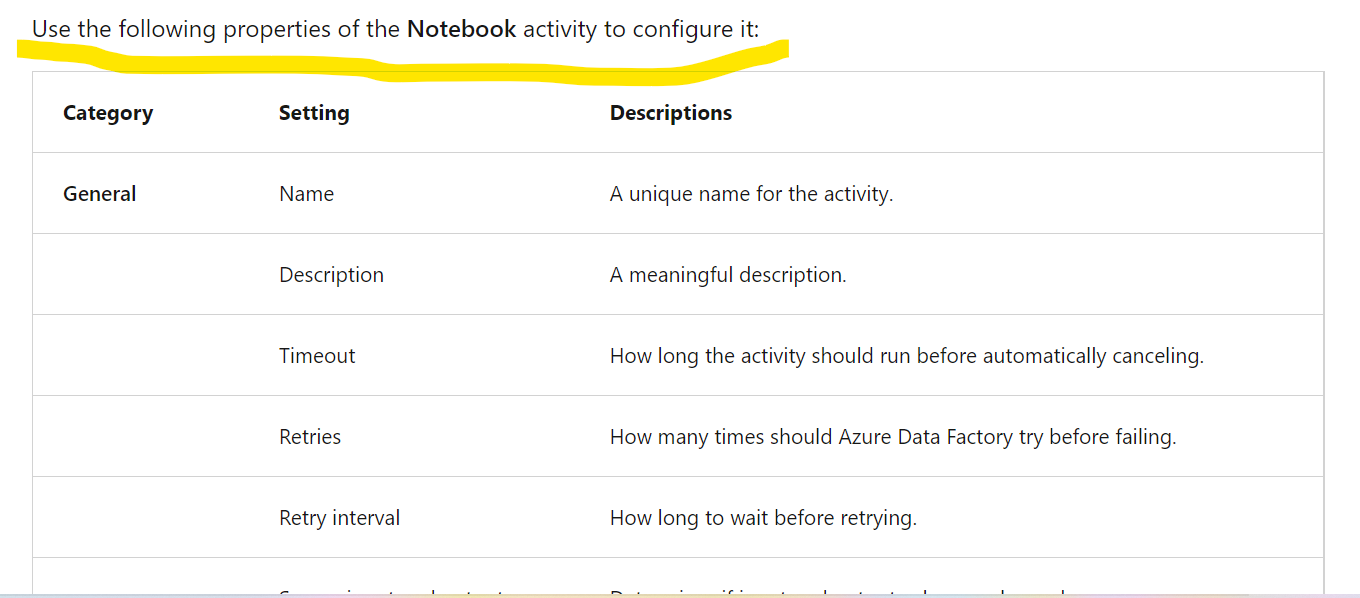
To run databricks notebooks in ADF, we need to authenticate by following options:

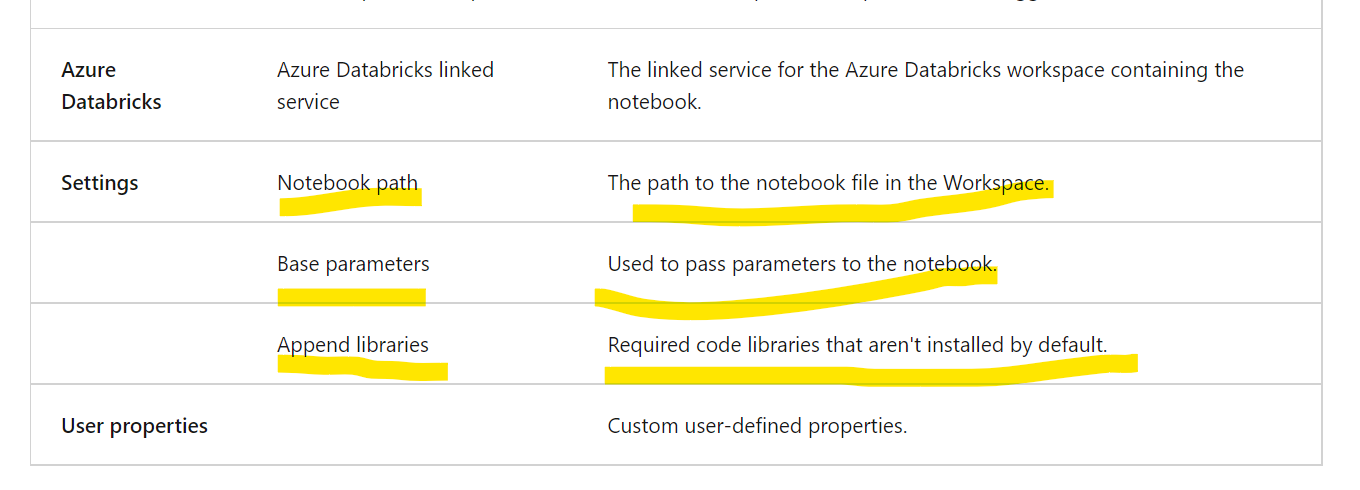
* 1. Generate access token for the Databricks workspace:

In Data bricks workspace account, user settings -> generate access token provide the life time for this token.

* 1. Create a linked service, select databricks from the compute service and select the access token generated.







# Use parameters in a notebook:

From pipeline, we can pass parameters to notebooks by doing the following:

1. We need to create widgets by using the dbutils.widgets library in your notebook code

Assgin

dbutils.widgets.text(“folder”,”data”)

dbutils.widgets.text("folder","value") #assign a default value

dbutils.widgets.get("folder") #retrieve values

# In addition to using parameters that can be passed in to a notebook, you pass values out to the calling application by using notebook.exit function

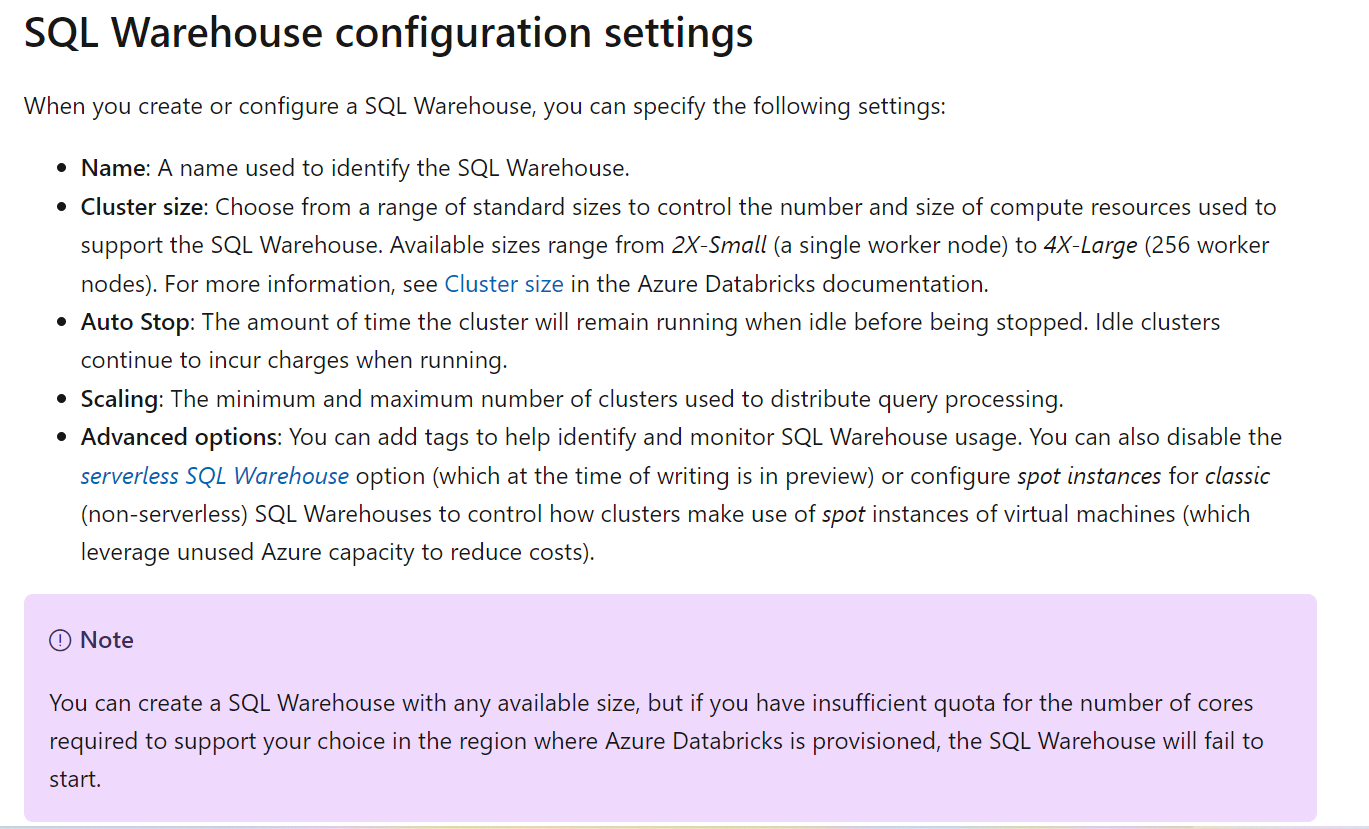
path = "dbfs:/{0}/products.csv".format(folder)

dbutils.notebook.exit(path)

**SQL WAREHOUSE CONFIGURATIONS:**

Data analysts use SQL to query relational data and create reports and dashboards.

SQL warehouses (formerly known as SQL Endpoints) provide a relational databse interface in databricks. The data stored in files are abstracted by Delta tables in a hive metastore.



The imported data is stored in Databricks File System (DBFS) storage, and a Delta table is defined for it in the Hive metastore.

CREATE TABLE salesdata.salesorders

(

orderid INT,

orderdate DATE,

)

USING DELTA

LOCATION '/data/sales/';

Azure Databricks SQL is primarily designed for data analytics and visualization workloads. To support these workloads, users can create queries to retrieve and summarize data from tables, and dashboards to share visualizations of the data.

Azure Databricks SQL is primarily designed for data analytics and visualization workloads. To support these workloads, users can create queries to retrieve and summarize data from tables, and dashboards to share visualizations of the data.

After saving the query, you can schedule it to be run automatically at regular intervals to refresh the data, or you can open it and run it interactively.

**Here starter warehouse is the databricks sql**

When you create a premium-tier Azure Databricks workspace, it includes a default SQL Warehouse named **Starter Warehouse**, which you can use to explore sample data and get started with SQL-based data analytics in Azure Databricks. You can modify the configuration of the default SQL Warehouse to suit your needs, or you can create more SQL Warehouses in your workspace.

