100 XP

Introduction

1 minute

Azure Databricks is a fully managed, cloud-based data analytics platform, which empowers developers to accelerate AI and innovation by simplifying the process of building enterprise-grade data applications. Built as a joint effort by Microsoft and the team that started Apache Spark, Azure Databricks provides data science, engineering, and analytical teams with a single platform for big data processing and machine learning.

By combining the power of Databricks, an end-to-end, managed Apache Spark platform optimized for the cloud, with the enterprise scale and security of Microsoft's Azure platform, Azure Databricks makes it simple to run large-scale Spark workloads.

In this module, you'll learn how to:

- Provision an Azure Databricks workspace.
- Identify core workloads and personas for Azure Databricks.
- Describe key concepts of an Azure Databricks solution.

Next unit: Get started with Azure Databricks

Continue

Get started with Azure Databricks

3 minutes

Azure Databricks is a cloud-based distributed platform for data processing built on Apache Spark. Databricks was designed to unify data science, data engineering, and business data analytics on Spark by creating an easy to use environment that enables users to spend more time working effectively with data, and less time focused on managing clusters and infrastructure. As the platform has evolved, it has kept up to date with the latest advances in the Spark runtime and added usability features to support common data workloads in a single, centrally managed interface.

Azure Databricks is hosted on the Microsoft Azure cloud platform, and integrated with Azure services such as Azure Active Directory, Azure Storage, Azure Synapse Analytics, and Azure Machine Learning. Organizations can apply their existing capabilities with the Databricks platform, and build fully integrated data analytics solutions that work with cloud infrastructure used by other enterprise applications.

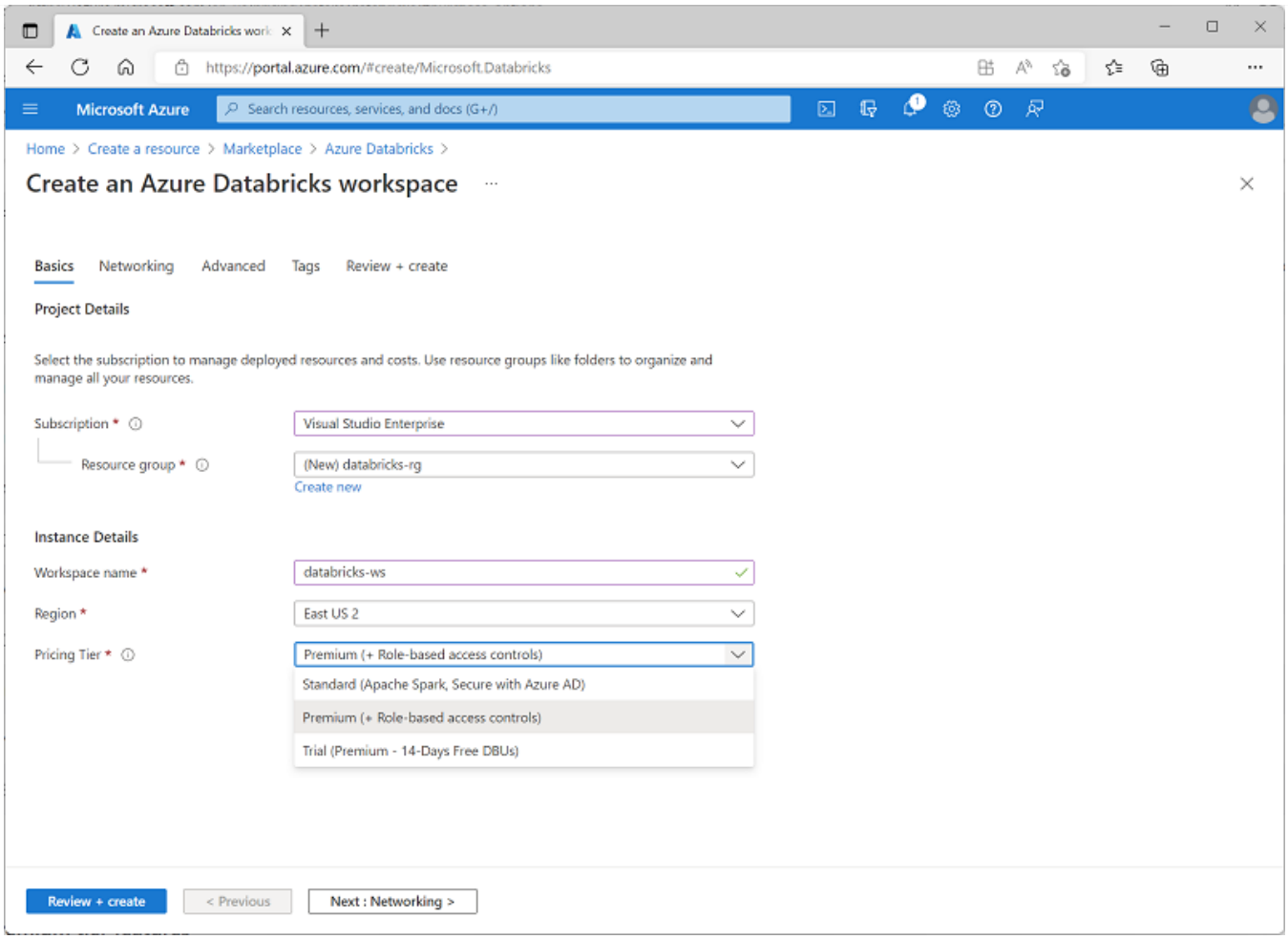
Creating an Azure Databricks workspace

To use Azure Databricks, you must create an Azure Databricks *workspace* in your Azure subscription. You can accomplish this by:

- Using the Azure portal user interface.
- Using an Azure Resource Manager (ARM) or Bicep template.
- Using the `New-AzDatabricksWorkspace` Azure PowerShell cmdlet
- Using the `az databricks workspace create` Azure command line interface (CLI) command.

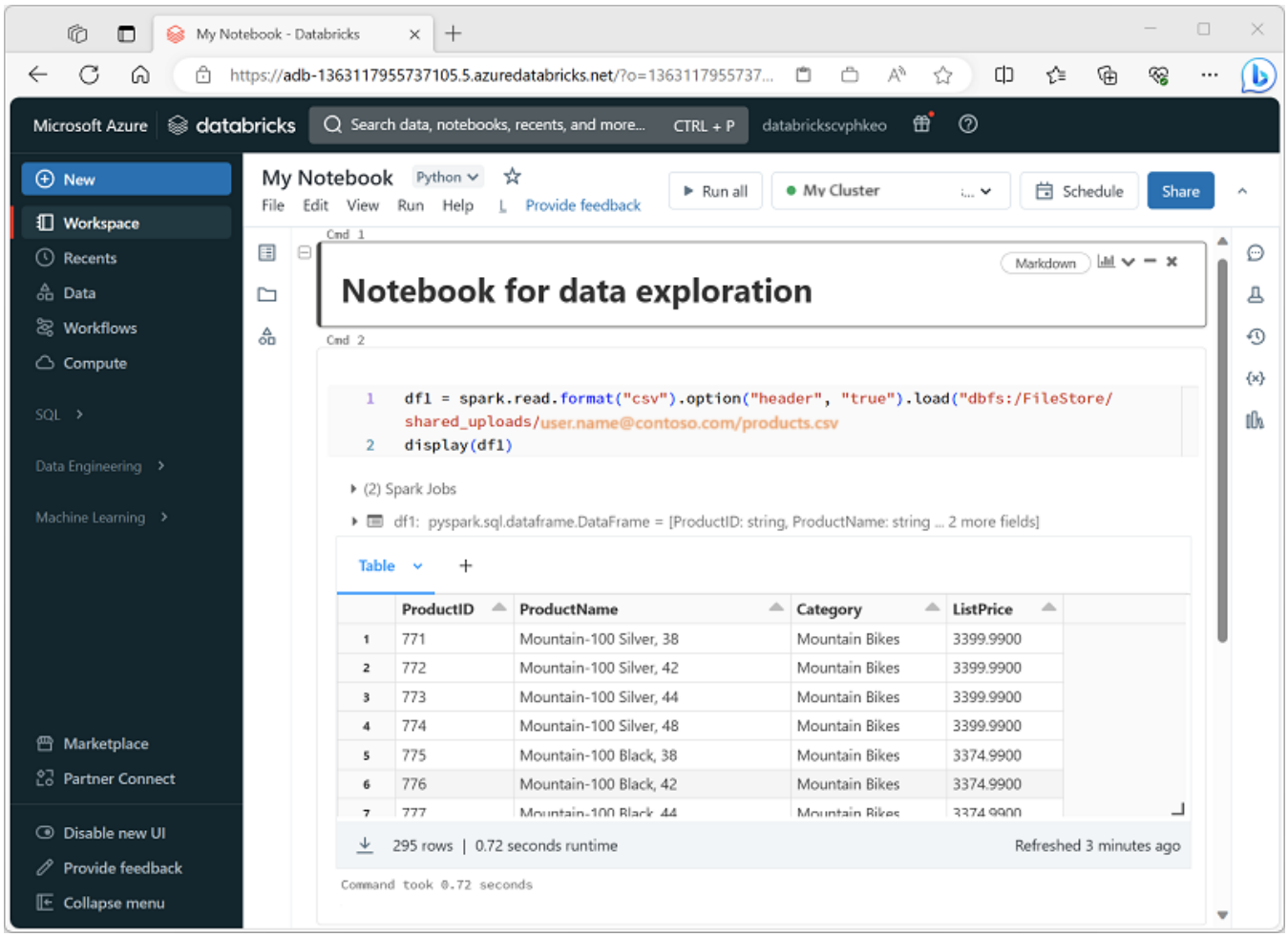
When you create a workspace, you must specify one of the following pricing tiers:

- **Standard** - Core Apache Spark capabilities with Azure AD integration.
- **Premium** - Role-based access controls and other enterprise-level features.
- **Trial** - A 14-day free trial of a premium-level workspace



Using the Azure Databricks portal

After you've provisioned an Azure Databricks workspace, you can use the Azure Databricks portal to work with data and compute resources. The Azure Databricks portal is a web-based user interface through which you can create and manage workspace resources (such as Spark clusters) and use notebooks and queries to work with data in files and tables.



Next unit: Identify Azure Databricks workloads

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Identify Azure Databricks workloads

3 minutes

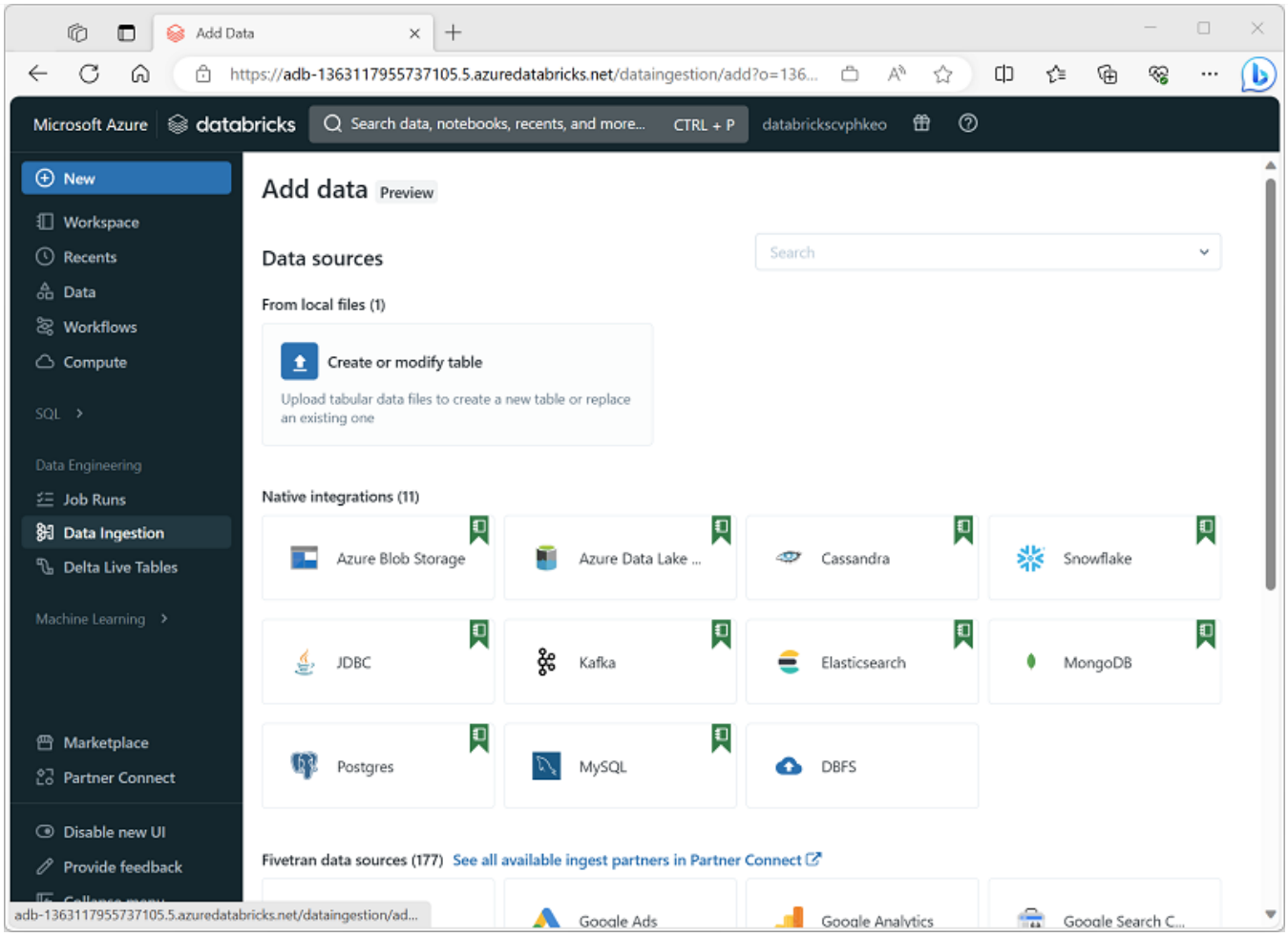
Azure Databricks is a comprehensive platform that offers many data processing capabilities. While you can use the service to support any workload that requires scalable data processing, Azure Databricks is optimized for three specific types of data workload and associated user personas:

- Data Science and Engineering
- Machine Learning
- SQL*

**SQL workloads are only available in premium tier workspaces.*

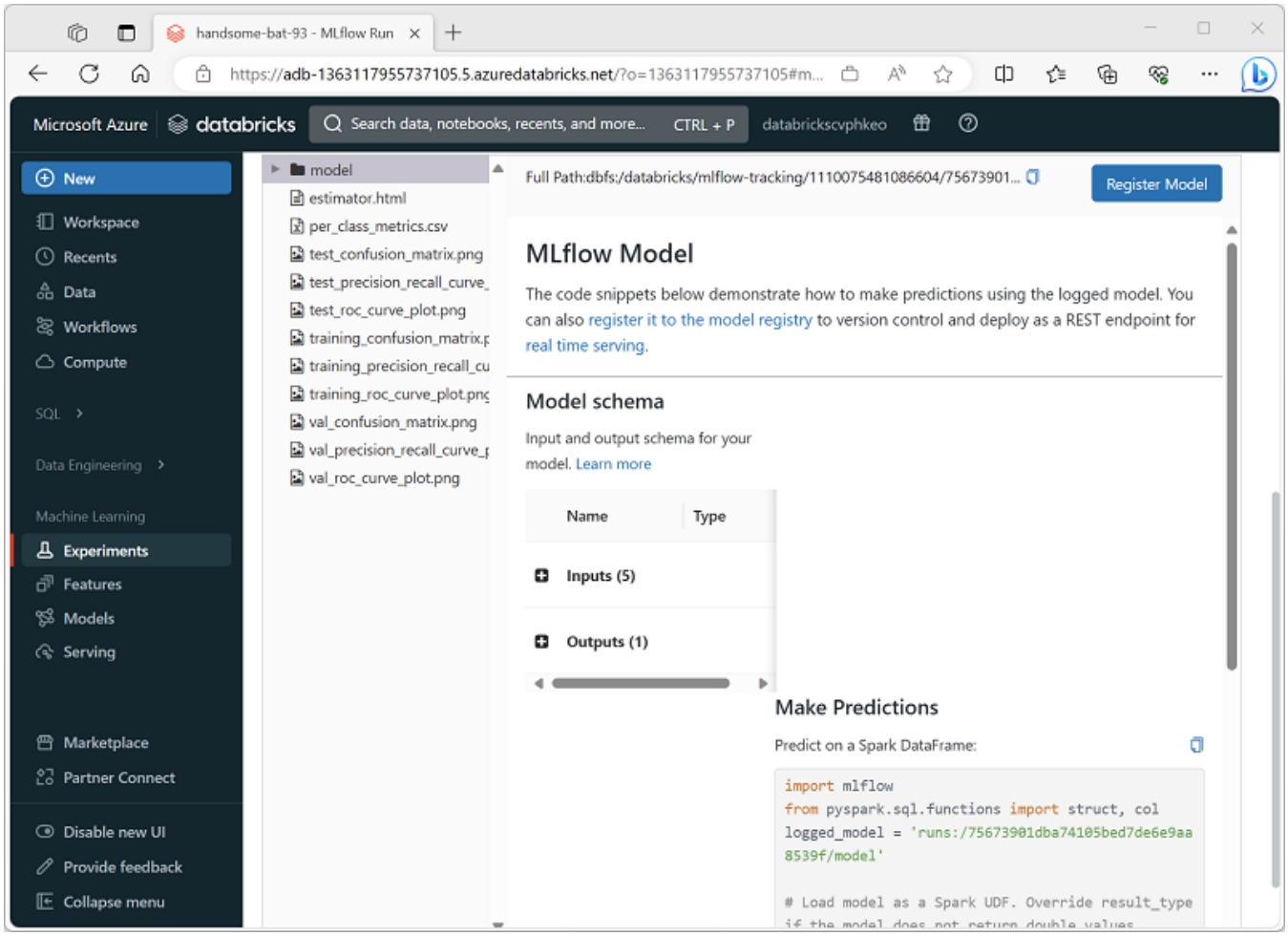
Data Science and Engineering

Azure Databricks provides Apache Spark based processing and analysis of large volumes of data in a data lake. Data engineers, data scientists, and data analysts can use interactive notebooks to run code in Python, Scala, SparkSQL, or other languages to cleanse, transform, aggregate, and analyze data.



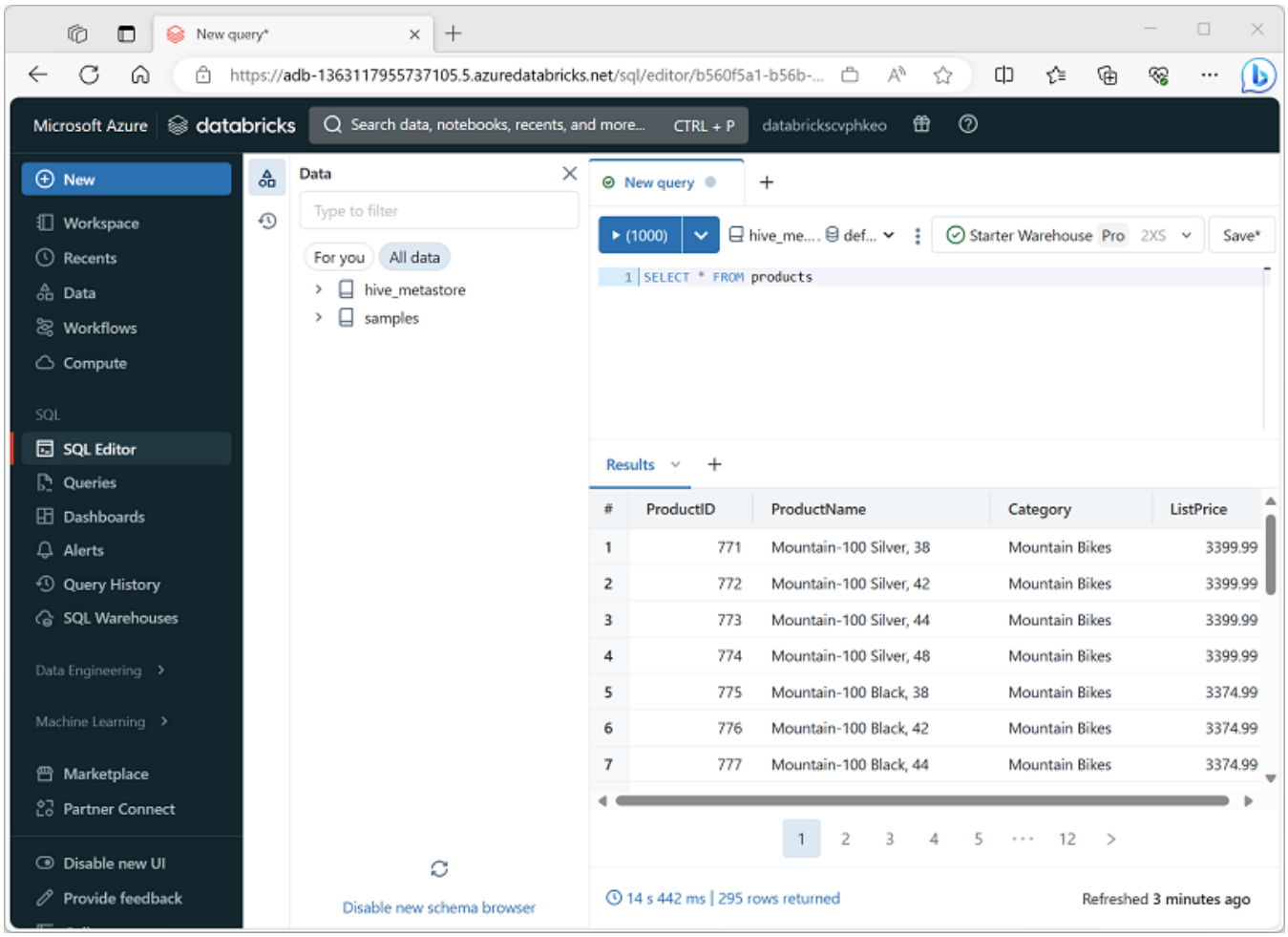
Machine Learning

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.



SQL

Azure Databricks supports SQL-based querying for data stored in tables in a *SQL Warehouse*. This capability enables data analysts to query, aggregate, summarize, and visualize data using familiar SQL syntax and a wide range of SQL-based data analytical tools.



> **Note**
>
> SQL Warehouses are only available in *premium* Azure Databricks workspaces.

Next unit: Understand key concepts

Continue >

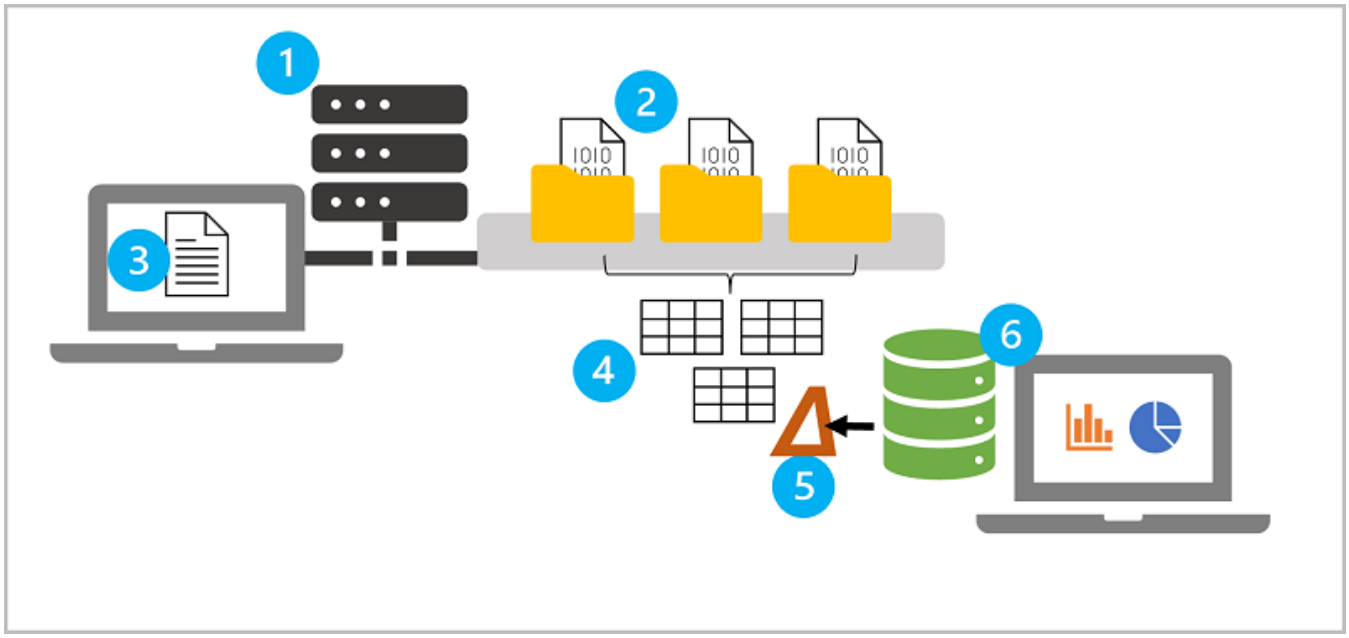
How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Understand key concepts

3 minutes

Azure Databricks is an amalgamation of multiple technologies that enable you to work with data at scale. Before using Azure Databricks, there are some key concepts that you should understand.



1. **Apache Spark clusters** - Spark is a distributed data processing solution that makes use of *clusters* to scale processing across multiple compute *nodes*. Each Spark cluster has a *driver* node to coordinate processing jobs, and one or more *worker* nodes on which the processing occurs. This distributed model enables each node to operate on a subset of the job in parallel; reducing the overall time for the job to complete. To learn more about clusters in Azure Databricks, see Clusters in the Azure Databricks documentation.
2. **Databricks File System (DBFS)** - While each cluster node has its own local file system (on which operating system and other node-specific files are stored), the nodes in a cluster have access to a shared, distributed file system in which they can access and operate on data files. The *Databricks File System* (DBFS) enables you to mount cloud storage and use it to work with and persist file-based data. To learn more about DBFS, see Databricks File System (DBFS) in the Azure Databricks documentation.
3. **Notebooks** - One of the most common ways for data analysts, data scientists, data engineers, and developers to work with Spark is to write code in *notebooks*. Notebooks provide an interactive environment in which you can combine text and graphics in *Markdown* format with cells containing code that you run interactively in the notebook

session. To learn more about notebooks, see Notebooks in the Azure Databricks documentation.

4. **Hive metastore** - *Hive* is an open source technology used to define a relational abstraction layer of tables over file-based data. The tables can then be queried using SQL syntax. The table definitions and details of the file system locations on which they're based is stored in the metastore for a Spark cluster. A *Hive metastore* is created for each cluster when it's created, but you can configure a cluster to use an existing external metastore if necessary - see Metastores in the Azure Databricks documentation for more details.
5. **Delta Lake** - *Delta Lake* builds on the relational table schema abstraction over files in the data lake to add support for SQL semantics commonly found in relational database systems. Capabilities provided by Delta Lake include transaction logging, data type constraints, and the ability to incorporate streaming data into a relational table. To learn more about Delta Lake, see Delta Lake Guide in the Azure Databricks documentation.
6. **SQL Warehouses** - *SQL Warehouses* are relational compute resources with endpoints that enable client applications to connect to an Azure Databricks workspace and use SQL to work with data in tables. The results of SQL queries can be used to create data visualizations and dashboards to support business analytics and decision making. SQL Warehouses are only available in *premium* tier Azure Databricks workspaces. To learn more about SQL Warehouses, see SQL Warehouses in the Azure Databricks documentation.

Next unit: Exercise - Explore Azure Databricks

Continue >

Knowledge check

3 minutes

1. You plan to create an Azure Databricks workspace and use it to work with a SQL Warehouse. Which of the following pricing tiers can you select? *

- ○ Enterprise

  ✗ Incorrect. There is no Enterprise tier for Azure Databricks.

- ○ Standard
- ○ Premium

  ✓ Correct. Premium tier is required for SQL Warehouses.

2. You've created an Azure Databricks workspace in which you plan to use code to process data files. What must you create in the workspace? *

- ○ A SQL Warehouse
- ○ A Spark cluster

  ✓ Correct. A Spark cluster is required to process data using code in notebooks.

- ○ A Windows Server virtual machine

3. You want to use Python code to interactively explore data in a text file that you've uploaded to your Azure Databricks workspace. What should you create? *

- ○ A SQL query
- ○ An Azure function
- ○ A Notebook

  ✓ Correct. A notebook is used to interactively explore data.

Next unit: Summary

Continue >

100 XP

Introduction

1 minute

Azure Databricks offers a highly scalable platform for data analytics and processing using Apache Spark.

Spark is a flexible platform that supports many different programming languages and APIs. Most data processing and analytics tasks can be accomplished using the **Dataframe** API, which is what we'll focus on in this module.

In this module, you'll learn how to:

- Describe key elements of the Apache Spark architecture.
- Create and configure a Spark cluster.
- Describe use cases for Spark.
- Use Spark to process and analyze data stored in files.
- Use Spark to visualize data.

Next unit: Get to know Spark

Continue

100 XP

Get to know Spark

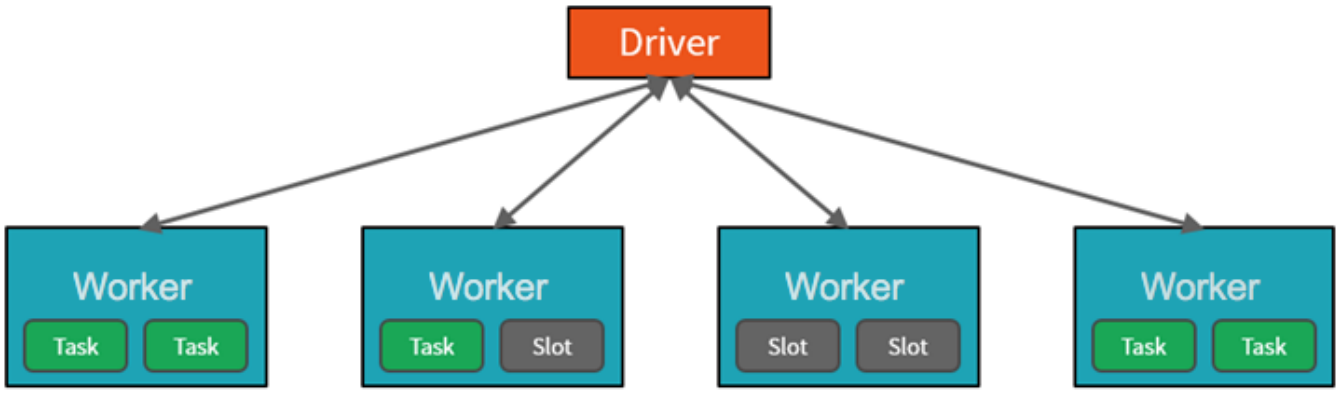
4 minutes

To gain a better understanding of how to process and analyze data with Apache Spark in Azure Databricks, it's important to understand the underlying architecture.

High-level overview

From a high level, the Azure Databricks service launches and manages Apache Spark clusters within your Azure subscription. Apache Spark clusters are groups of computers that are treated as a single computer and handle the execution of commands issued from notebooks. Clusters enable processing of data to be parallelized across many computers to improve scale and performance. They consist of a Spark *driver* and *worker* nodes. The driver node sends work to the worker nodes and instructs them to pull data from a specified data source.

In Databricks, the notebook interface is typically the driver program. This driver program contains the main loop for the program and creates distributed datasets on the cluster, then applies operations to those datasets. Driver programs access Apache Spark through a *SparkSession* object regardless of deployment location.



Microsoft Azure manages the cluster, and auto-scales it as needed based on your usage and the setting used when configuring the cluster. Auto-termination can also be enabled, which allows Azure to terminate the cluster after a specified number of minutes of inactivity.

Spark jobs in detail

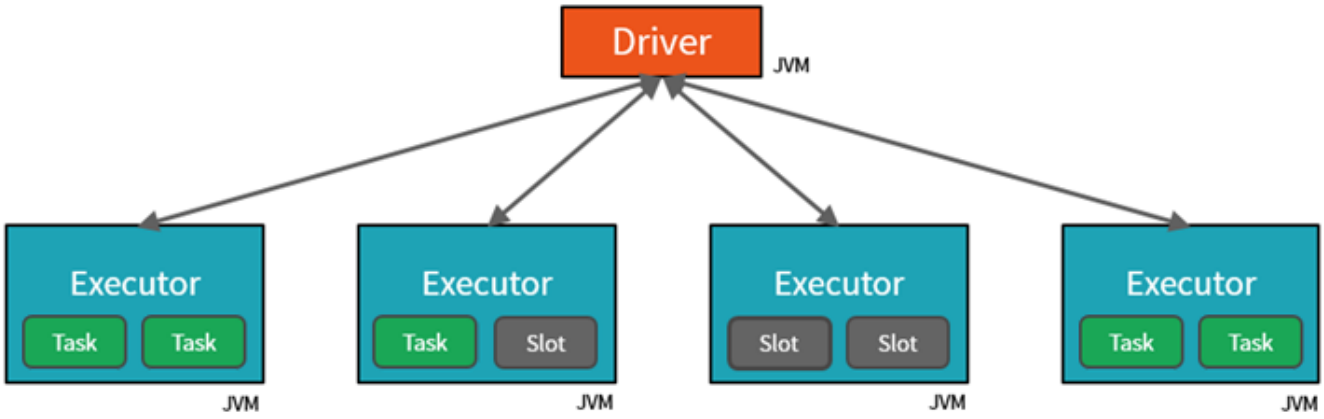
Work submitted to the cluster is split into as many independent jobs as needed. This is how work is distributed across the Cluster's nodes. Jobs are further subdivided into tasks. The input to a job is partitioned into one or more partitions. These partitions are the unit of work for

each slot. In between tasks, partitions may need to be reorganized and shared over the network.

The secret to Spark's high performance is parallelism. Scaling *vertically* (by adding resources to a single computer) is limited to a finite amount of RAM, Threads and CPU speeds; but clusters scale *horizontally*, adding new nodes to the cluster as needed.

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.
- Each executor has multiple slots to which parallelized tasks can be assigned.



The JVM is naturally multi-threaded, but a single JVM, such as the one coordinating the work on the driver, has a finite upper limit. By splitting the work into tasks, the driver can assign units of work to *slots in the executors on worker nodes for parallel execution. Additionally, the driver determines how to partition the data so that it can be distributed for parallel processing. So, the driver assigns a partition of data to each task so that each task knows which piece of data it is to process. Once started, each task will fetch the partition of data assigned to it.

Jobs and stages

Depending on the work being performed, multiple parallelized jobs may be required. Each job is broken down into *stages*. A useful analogy is to imagine that the job is to build a house:

- The first stage would be to lay the foundation.
- The second stage would be to erect the walls.
- The third stage would be to add the roof.

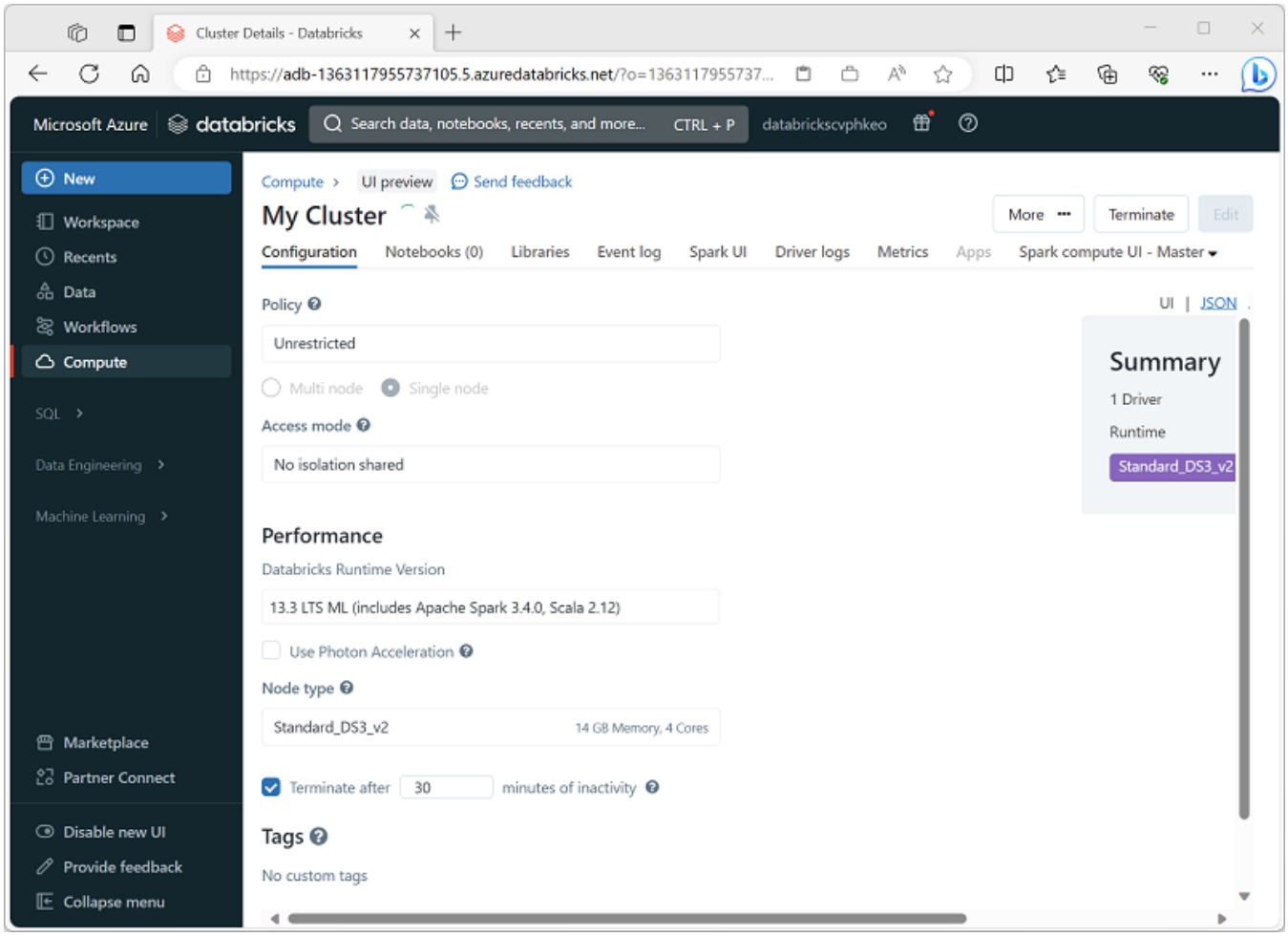
Attempting to do any of these steps out of order just doesn't make sense, and may in fact be impossible. Similarly, Spark breaks each job into stages to ensure everything is done in the right order.

100 XP

Create a Spark cluster

3 minutes

You can create one or more clusters in your Azure Databricks workspace by using the Azure Databricks portal.



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.

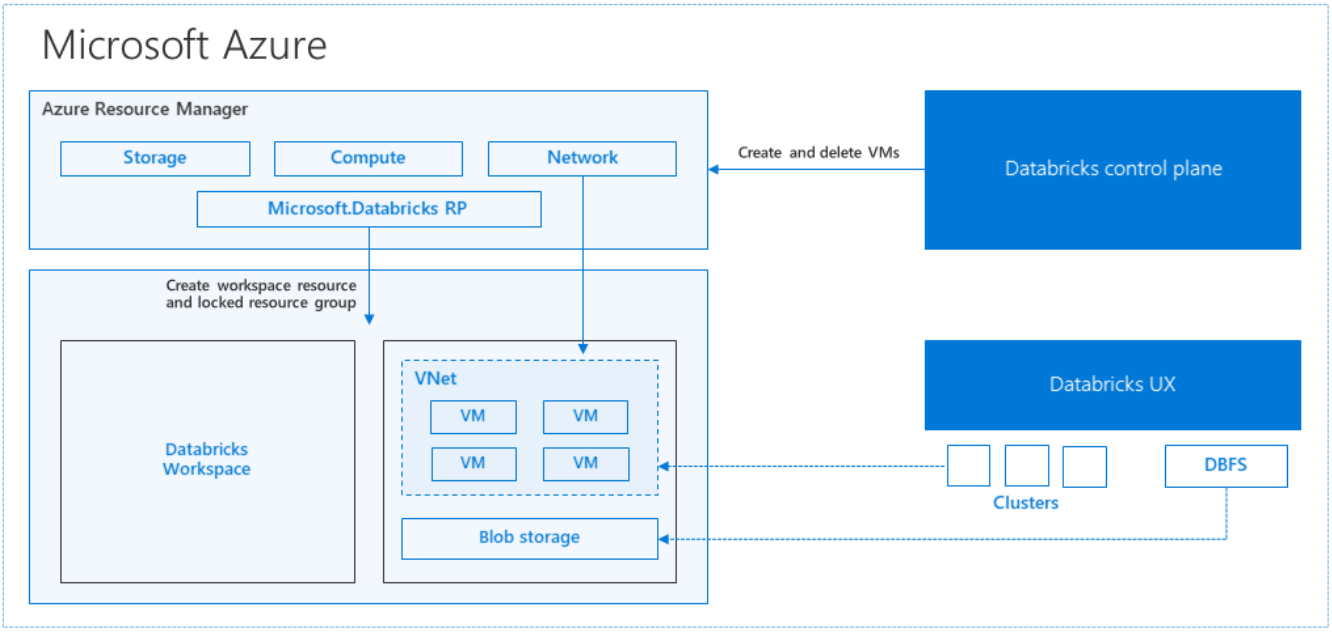
- The type of virtual machine (VM) used for the worker nodes in the cluster.
- The minimum and maximum number of worker nodes in the cluster.
- The type of VM used for the driver node in the cluster.
- Whether the cluster supports *autoscaling* to dynamically resize the cluster.
- How long the cluster can remain idle before being shut down automatically.

How Azure manages cluster resources

When you create an Azure Databricks workspace, a *Databricks appliance* is deployed as an Azure resource in your subscription. When you create a cluster in the workspace, you specify the types and sizes of the virtual machines (VMs) to use for both the driver and worker nodes, and some other configuration options, but Azure Databricks manages all other aspects of the cluster.

The Databricks appliance is deployed into Azure as a managed resource group within your subscription. This resource group contains the driver and worker VMs for your clusters, along with other required resources, including a virtual network, a security group, and a storage account. All metadata for your cluster, such as scheduled jobs, is stored in an Azure Database with geo-replication for fault tolerance.

Internally, Azure Kubernetes Service (AKS) is used to run the Azure Databricks control-plane and data-planes via containers running on the latest generation of Azure hardware (Dv3 VMs), with NvMe SSDs capable of blazing 100us latency on high-performance Azure virtual machines with accelerated networking. Azure Databricks utilizes these features of Azure to further improve Spark performance. After the services within your managed resource group are ready, you can manage the Databricks cluster through the Azure Databricks UI and through features such as auto-scaling and auto-termination.



> **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** in the Azure Databricks documentation.

Next unit: Use Spark in notebooks

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Use Spark in notebooks

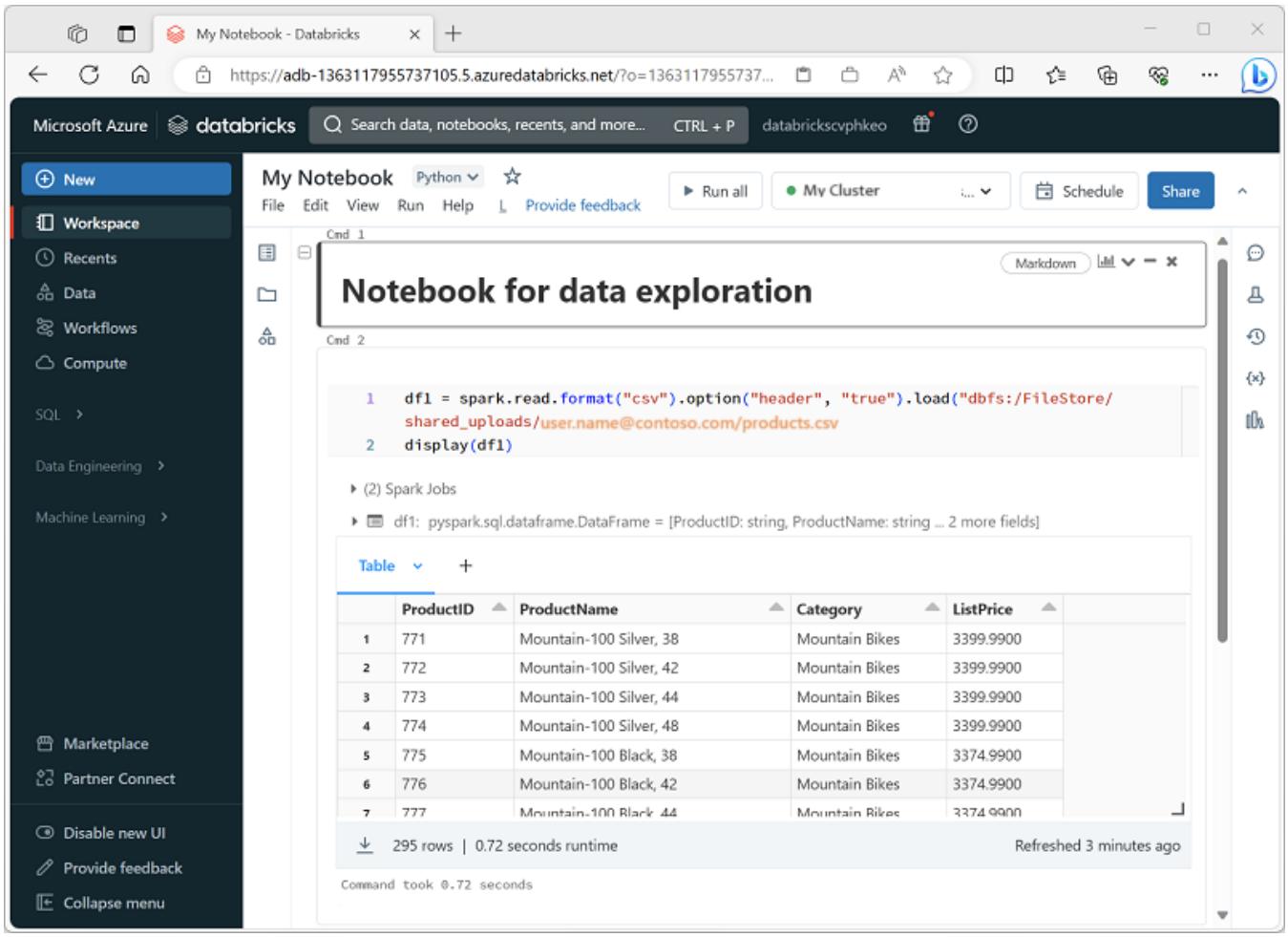
6 minutes

You can run many different kinds of application on Spark, including code in Python or Scala scripts, Java code compiled as a Java Archive (JAR), and others. Spark is commonly used in two kinds of workload:

- Batch or stream processing jobs to ingest, clean, and transform data - often running as part of an automated pipeline.
- Interactive analytics sessions to explore, analyze, and visualize data.

Running Spark code in notebooks

Azure Databricks includes an integrated notebook interface for working with Spark. Notebooks provide an intuitive way to combine code with Markdown notes, commonly used by data scientists and data analysts. The look and feel of the integrated notebook experience within Azure Databricks is similar to that of Jupyter notebooks - a popular open source notebook platform.



Notebooks consist of one or more *cells*, each containing either code or markdown. Code cells in notebooks have some features that can help you be more productive, including:

- Syntax highlighting and error support.
- Code auto-completion.
- Interactive data visualizations.
- The ability to export results.

> **Tip**
>
> To learn more about working with notebooks in Azure Databricks, see the **Notebooks** article in the Azure Databricks documentation.

Next unit: Use Spark to work with data files

Continue >

100 XP

Use Spark to work with data files

5 minutes

One of the benefits of using Spark is that you can write and run code in various programming languages, enabling you to use the programming skills you already have and to use the most appropriate language for a given task. The default language in a new Azure Databricks Spark notebook is *PySpark* - a Spark-optimized version of Python, which is commonly used by data scientists and analysts due to its strong support for data manipulation and visualization. Additionally, you can use languages such as *Scala* (a Java-derived language that can be used interactively) and *SQL* (a variant of the commonly used SQL language included in the *Spark SQL* library to work with relational data structures). Software engineers can also create compiled solutions that run on Spark using frameworks such as *Java*.

Exploring data with dataframes

Natively, Spark uses a data structure called a *resilient distributed dataset* (RDD); but while you *can* write code that works directly with RDDs, the most commonly used data structure for working with structured data in Spark is the *dataframe*, which is provided as part of the *Spark SQL* library. Dataframes in Spark are similar to those in the ubiquitous *Pandas* Python library, but optimized to work in Spark's distributed processing environment.

> **Note**
>
> In addition to the Dataframe API, Spark SQL provides a strongly-typed *Dataset* API that is supported in Java and Scala. We'll focus on the Dataframe API in this module.

Loading data into a dataframe

Let's explore a hypothetical example to see how you can use a dataframe to work with data. Suppose you have the following data in a comma-delimited text file named **products.csv** in the **data** folder in your Databricks File System (DBFS) storage:

csv

```
ProductID,ProductName,Category,ListPrice
771,"Mountain-100 Silver, 38",Mountain Bikes,3399.9900
772,"Mountain-100 Silver, 42",Mountain Bikes,3399.9900
```

```
773,"Mountain-100 Silver, 44",Mountain Bikes,3399.9900
...
```

In a Spark notebook, you could use the following PySpark code to load the data into a dataframe and display the first 10 rows:

Python

```
%pyspark
df = spark.read.load('/data/products.csv',
    format='csv',
    header=True
)
display(df.limit(10))
```

The `%pyspark` line at the beginning is called a *magic*, and tells Spark that the language used in this cell is PySpark. Here's the equivalent Scala code for the products data example:

Scala

```
%spark
val df = spark.read.format("csv").option("header",
"true").load("/data/products.csv")
display(df.limit(10))
```

The magic `%spark` is used to specify Scala.

> **Tip**
>
> You can also select the language you want to use for each cell in the Notebook interface.

Both of the examples shown previously would produce output like this:

ProductID	ProductName	Category	ListPrice
771	Mountain-100 Silver, 38	Mountain Bikes	3399.9900
772	Mountain-100 Silver, 42	Mountain Bikes	3399.9900
773	Mountain-100 Silver, 44	Mountain Bikes	3399.9900
...	...	...	...

Specifying a dataframe schema

In the previous example, the first row of the CSV file contained the column names, and Spark was able to infer the data type of each column from the data it contains. You can also specify an explicit schema for the data, which is useful when the column names aren't included in the data file, like this CSV example:

csv

```
771,"Mountain-100 Silver, 38",Mountain Bikes,3399.9900
772,"Mountain-100 Silver, 42",Mountain Bikes,3399.9900
773,"Mountain-100 Silver, 44",Mountain Bikes,3399.9900
...
```

The following PySpark example shows how to specify a schema for the dataframe to be loaded from a file named **product-data.csv** in this format:

Python

```
from pyspark.sql.types import *
from pyspark.sql.functions import *

productSchema = StructType([
    StructField("ProductID", IntegerType()),
    StructField("ProductName", StringType()),
    StructField("Category", StringType()),
    StructField("ListPrice", FloatType())
    ])

df = spark.read.load('/data/product-data.csv',
    format='csv',
    schema=productSchema,
    header=False)
display(df.limit(10))
```

The results would once again be similar to:

ProductID	ProductName	Category	ListPrice
771	Mountain-100 Silver, 38	Mountain Bikes	3399.9900
772	Mountain-100 Silver, 42	Mountain Bikes	3399.9900
773	Mountain-100 Silver, 44	Mountain Bikes	3399.9900

ProductID	ProductName	Category	ListPrice
...	...	...	...

Filtering and grouping dataframes

You can use the methods of the Dataframe class to filter, sort, group, and otherwise manipulate the data it contains. For example, the following code example uses the **select** method to retrieve the **ProductName** and **ListPrice** columns from the **df** dataframe containing product data in the previous example:

Python

```
pricelist_df = df.select("ProductID", "ListPrice")
```

The results from this code example would look something like this:

ProductID	ListPrice
771	3399.9900
772	3399.9900
773	3399.9900
...	...

In common with most data manipulation methods, **select** returns a new dataframe object.

> **Tip**
>
> Selecting a subset of columns from a dataframe is a common operation, which can also be achieved by using the following shorter syntax:
>
> ```
> pricelist_df = df["ProductID", "ListPrice"]
> ```

You can "chain" methods together to perform a series of manipulations that results in a transformed dataframe. For example, this example code chains the **select** and **where** methods

to create a new dataframe containing the **ProductName** and **ListPrice** columns for products with a category of **Mountain Bikes** or **Road Bikes**:

Python

```
bikes_df = df.select("ProductName",
"ListPrice").where((df["Category"]=="Mountain Bikes") |
(df["Category"]=="Road Bikes"))
display(bikes_df)
```

The results from this code example would look something like this:

ProductName	ListPrice
Mountain-100 Silver, 38	3399.9900
Road-750 Black, 52	539.9900
...	...

To group and aggregate data, you can use the **groupBy** method and aggregate functions. For example, the following PySpark code counts the number of products for each category:

Python

```
counts_df = df.select("ProductID", "Category").groupBy("Category").count()
display(counts_df)
```

The results from this code example would look something like this:

Category	count
Headsets	3
Wheels	14
Mountain Bikes	32
...	...

Using SQL expressions in Spark

The Dataframe API is part of a Spark library named Spark SQL, which enables data analysts to use SQL expressions to query and manipulate data.

Creating database objects in the Spark catalog

The Spark catalog is a metastore for relational data objects such as views and tables. The Spark runtime can use the catalog to seamlessly integrate code written in any Spark-supported language with SQL expressions that may be more natural to some data analysts or developers.

One of the simplest ways to make data in a dataframe available for querying in the Spark catalog is to create a temporary view, as shown in the following code example:

Python

```
df.createOrReplaceTempView("products")
```

A *view* is temporary, meaning that it's automatically deleted at the end of the current session. You can also create *tables* that are persisted in the catalog to define a database that can be queried using Spark SQL.

> **Note**
>
> 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.

Using the Spark SQL API to query data

You can use the Spark SQL API in code written in any language to query data in the catalog. For example, the following PySpark code uses a SQL query to return data from the **products** view as a dataframe.

Python

```
bikes_df = spark.sql("SELECT ProductID, ProductName, ListPrice \
                      FROM products \
                      WHERE Category IN ('Mountain Bikes', 'Road Bikes')")
display(bikes_df)
```

The results from the code example would look similar to the following table:

ProductName	ListPrice
Mountain-100 Silver, 38	3399.9900
Road-750 Black, 52	539.9900
...	...

Using SQL code

The previous example demonstrated how to use the Spark SQL API to embed SQL expressions in Spark code. In a notebook, you can also use the `%sql` magic to run SQL code that queries objects in the catalog, like this:

SQL

```
%sql

SELECT Category, COUNT(ProductID) AS ProductCount
FROM products
GROUP BY Category
ORDER BY Category
```

The SQL code example returns a resultset that is automatically displayed in the notebook as a table, like the one below:

Category	ProductCount
Bib-Shorts	3

Category	ProductCount
Bike Racks	1
Bike Stands	1
...	...

Next unit: Visualize data

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

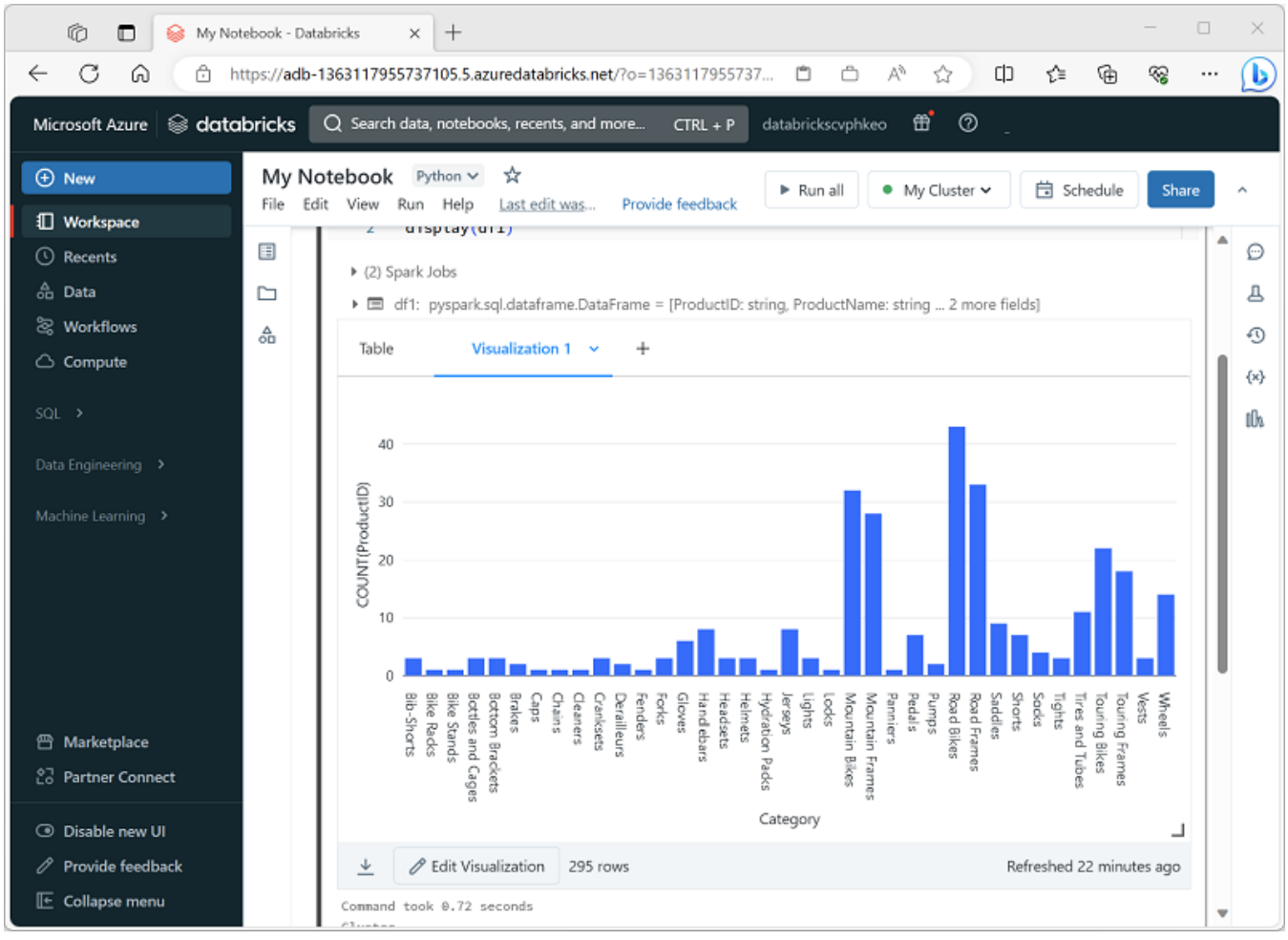
Visualize data

6 minutes

One of the most intuitive ways to analyze the results of data queries is to visualize them as charts. Notebooks in Azure Databricks provide charting capabilities in the user interface, and when that functionality doesn't provide what you need, you can use one of the many Python graphics libraries to create and display data visualizations in the notebook.

Using built-in notebook charts

When you display a dataframe or run a SQL query in a Spark notebook in Azure Databricks, the results are displayed under the code cell. By default, results are rendered as a table, but you can also view the results as a visualization and customize how the chart displays the data, as shown here:



The built-in visualization functionality in notebooks is useful when you want to quickly summarize the data visually. When you want to have more control over how the data is

formatted, or to display values that you have already aggregated in a query, you should consider using a graphics package to create your own visualizations.

Using graphics packages in code

There are many graphics packages that you can use to create data visualizations in code. In particular, Python supports a large selection of packages; most of them built on the base **Matplotlib** library. The output from a graphics library can be rendered in a notebook, making it easy to combine code to ingest and manipulate data with inline data visualizations and markdown cells to provide commentary.

For example, you could use the following PySpark code to aggregate data from the hypothetical products data explored previously in this module, and use Matplotlib to create a chart from the aggregated data.

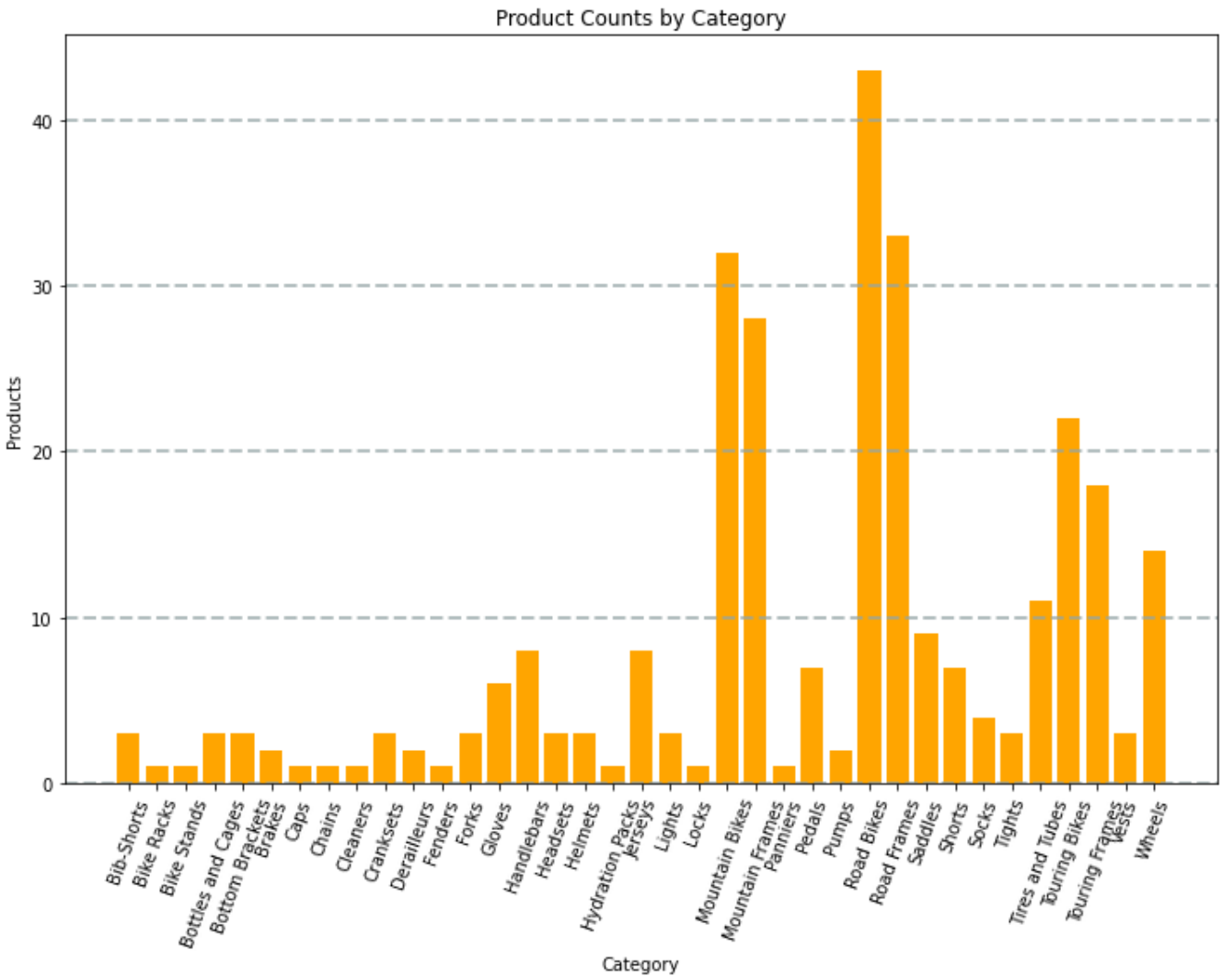
Python

```
from matplotlib import pyplot as plt

# Get the data as a Pandas dataframe
data = spark.sql("SELECT Category, COUNT(ProductID) AS ProductCount \
                  FROM products \
                  GROUP BY Category \
                  ORDER BY Category").toPandas()

# Clear the plot area
plt.clf()

# Create a Figure
fig = plt.figure(figsize=(12,8))

# Create a bar plot of product counts by category
plt.bar(x=data['Category'], height=data['ProductCount'], color='orange')

# Customize the chart
plt.title('Product Counts by Category')
plt.xlabel('Category')
plt.ylabel('Products')
plt.grid(color='#95a5a6', linestyle='--', linewidth=2, axis='y', alpha=0.7)
plt.xticks(rotation=70)

# Show the plot area
plt.show()
```

The Matplotlib library requires data to be in a Pandas dataframe rather than a Spark dataframe, so the **toPandas** method is used to convert it. The code then creates a figure with a specified

size and plots a bar chart with some custom property configuration before showing the resulting plot.

The chart produced by the code would look similar to the following image:



You can use the Matplotlib library to create many kinds of chart; or if preferred, you can use other libraries such as **Seaborn** to create highly customized charts.

> **Note**
>
> The Matplotlib and Seaborn libraries may already be installed on Databricks clusters, depending on the Databricks Runtime for the cluster. If not, or if you want to use a different library that is not already installed, you can add it to the cluster. See **Cluster Libraries** in the Azure Databricks documentation for details.

Next unit: Exercise - Use Spark in Azure Databricks

Check your knowledge

1. Which definition best describes Apache Spark? *

- ( ) A highly scalable relational database management system.

  ✗ Incorrect. Spark supports some features of relational databases, but it is not primarily a relational database engine.

- ( ) A virtual server with a Python runtime.
- ( ) A distributed platform for parallel data processing using multiple languages.

  ✓ Correct. Spark provides a highly scalable distributed platform on which you can run code written in many languages to process data.

2. You need to use Spark to analyze data in a parquet file. What should you do? *

- ( ) Load the parquet file into a dataframe.

  ✓ Correct. You can load data from files in many formats, including parquet, into a Spark dataframe.

- ( ) Import the data into a table in a serverless SQL pool.

  ✗ Incorrect. You do not need to load the data into a SQL pool in order to analyze it with Spark.

- ( ) Convert the data to CSV format.

3. You want to write code in a notebook cell that uses a SQL query to retrieve data from a view in the Spark catalog. Which magic should you use? *

- ( ) %spark
- ( ) %pyspark
- ( ) %sql

  ✓ Correct. The %sql magic instructs Spark to interpret the code in the cell as SQL.

Next unit: Summary

Continue >

100 XP

Introduction

1 minute

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.

In this module, you'll learn how to:

- Describe core features and capabilities of Delta Lake.
- Create and use Delta Lake tables in Azure Databricks.
- Create Spark catalog tables for Delta Lake data.
- Use Delta Lake tables for streaming data.

> **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.

Next unit: Get Started with Delta Lake

Continue

100 XP

Get Started with Delta Lake

3 minutes

Delta Lake is an open-source storage layer that adds relational database semantics to Spark-based data lake processing. Delta Lake is supported in Azure Synapse Analytics Spark pools for PySpark, Scala, and .NET code.

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.

> **Tip**
>
> For more information about Delta Lake in Azure Databricks, see the **Delta Lake guide** in the Azure Databricks documentation.

Next unit: Create Delta Lake tables

100 XP

Create Delta Lake tables

5 minutes

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

Creating a Delta Lake table from a dataframe

One of the easiest ways to create a Delta Lake table is to save a dataframe in the *delta* format, specifying a path where the data files and related metadata information for the table should be stored.

For example, the following PySpark code loads a dataframe with data from an existing file, and then saves that dataframe to a new folder location in *delta* format:

Python

```
# 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.

You can replace an existing Delta Lake table with the contents of a dataframe by using the **overwrite** mode, as shown here:

Python

```
new_df.write.format("delta").mode("overwrite").save(delta_table_path)
```

You can also add rows from a dataframe to an existing table by using the **append** mode:

Python

```
new_rows_df.write.format("delta").mode("append").save(delta_table_path)
```

Making conditional updates

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":

Python

```
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.

> **Tip**
>
> For more information about using the Data Lake API, see **the Delta Lake API documentation** .

Querying a previous version of a table

Delta Lake tables support versioning through the transaction log. The transaction log records modifications made to the table, noting the timestamp and version number for each transaction. You can use this logged version data to view previous versions of the table - a feature known as *time travel*.

You can retrieve data from a specific version of a Delta Lake table by reading the data from the delta table location into a dataframe, specifying the version required as a `versionAsOf` option:

Python

```
df = spark.read.format("delta").option("versionAsOf", 0).load(delta_table_-
path)
```

Alternatively, you can specify a timestamp by using the `timestampAsOf` option:

Python

```
df = spark.read.format("delta").option("timestampAsOf", '2022-01-
01').load(delta_table_path)
```

Next unit: Create and query catalog tables

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Create and query catalog tables

5 minutes

So far we've considered Delta Lake table instances created from dataframes and modified through the Delta Lake API. You can also define Delta Lake tables as catalog tables in the Hive metastore for your Spark cluster, and work with them using SQL.

External vs *managed* tables

Tables in a Spark catalog, including Delta Lake tables, can be *managed* or *external*; and it's important to understand the distinction between these kinds of table.

- A *managed* table is defined without a specified location, and the data files are stored within the storage used by the metastore. Dropping the table not only removes its metadata from the catalog, but also deletes the folder in which its data files are stored.
- An *external* table is defined for a custom file location, where the data for the table is stored. The metadata for the table is defined in the Spark catalog. Dropping the table deletes the metadata from the catalog, but doesn't affect the data files.

Creating catalog tables

There are several ways to create catalog tables.

Creating a catalog table from a dataframe

You can create managed tables by writing a dataframe using the `saveAsTable` operation as shown in the following examples:

Python

```
# Save a dataframe as a managed table
df.write.format("delta").saveAsTable("MyManagedTable")

## specify a path option to save as an external table
df.write.format("delta").option("path", "/mydata").saveAsTable("MyExternal-
Table")
```

Creating a catalog table using SQL

You can also create a catalog table by using the `CREATE TABLE` SQL statement with the `USING DELTA` clause, and an optional `LOCATION` parameter for external tables. You can run the statement using the SparkSQL API, like the following example:

Python

```
spark.sql("CREATE TABLE MyExternalTable USING DELTA LOCATION '/mydata'")
```

Alternatively you can use the native SQL support in Spark to run the statement:

SQL

```
%sql

CREATE TABLE MyExternalTable
USING DELTA
LOCATION '/mydata'
```

> **Tip**
>
> The `CREATE TABLE` statement returns an error if a table with the specified name already exists in the catalog. To mitigate this behavior, you can use a `CREATE TABLE IF NOT EXISTS` statement or the `CREATE OR REPLACE TABLE` statement.

Defining the table schema

In all of the examples so far, the table is created without an explicit schema. In the case of tables created by writing a dataframe, the table schema is inherited from the dataframe. When creating an external table, the schema is inherited from any files that are currently stored in the table location. However, when creating a new managed table, or an external table with a currently empty location, you define the table schema by specifying the column names, types, and nullability as part of the CREATE TABLE statement; as shown in the following example:

SQL

```
%sql

CREATE TABLE ManagedSalesOrders
(
    Orderid INT NOT NULL,
    OrderDate TIMESTAMP NOT NULL,
```

```
    CustomerName STRING,
    SalesTotal FLOAT NOT NULL
)
USING DELTA
```

When using Delta Lake, table schemas are enforced - all inserts and updates must comply with the specified column nullability and data types.

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:

Python

```
from delta.tables import *

DeltaTable.create(spark) \
  .tableName("default.ManagedProducts") \
  .addColumn("Productid", "INT") \
  .addColumn("ProductName", "STRING") \
  .addColumn("Category", "STRING") \
  .addColumn("Price", "FLOAT") \
  .execute()
```

Similarly to the `CREATE TABLE` SQL statement, the `create` method returns an error if a table with the specified name already exists. You can mitigate this behavior by using the `createIfNotExists` or `createOrReplace` method.

Using catalog tables

You can use catalog tables like tables in any SQL-based relational database, querying and manipulating them by using standard SQL statements. For example, the following code example uses a `SELECT` statement to query the **ManagedSalesOrders** table:

SQL

```
%sql

SELECT orderid, salestotal
FROM ManagedSalesOrders
```

Tip

For more information about working with Delta Lake, see **Table batch reads and writes** in the Delta Lake documentation.

Next unit: Use Delta Lake for streaming data

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Use Delta Lake for streaming data

5 minutes

All of the data we've explored up to this point has been static data in files. However, many data analytics scenarios involve *streaming* data that must be processed in near real time. For example, you might need to capture readings emitted by internet-of-things (IoT) devices and store them in a table as they occur.

Spark Structured Streaming

A typical stream processing solution involves constantly reading a stream of data from a *source*, optionally processing it to select specific fields, aggregate and group values, or otherwise manipulate the data, and writing the results to a *sink*.

Spark includes native support for streaming data through *Spark Structured Streaming*, an API that is based on a boundless dataframe in which streaming data is captured for processing. A Spark Structured Streaming dataframe can read data from many different kinds of streaming source, including network ports, real time message brokering services such as Azure Event Hubs or Kafka, or file system locations.

> **Tip**
>
> For more information about Spark Structured Streaming, see **Structured Streaming Programming Guide** in the Spark documentation.

Streaming with Delta Lake tables

You can use a Delta Lake table as a source or a sink for Spark Structured Streaming. For example, you could capture a stream of real time data from an IoT device and write the stream directly to a Delta Lake table as a sink - enabling you to query the table to see the latest streamed data. Or, you could read a Delta Table as a streaming source, enabling you to constantly report new data as it is added to the table.

Using a Delta Lake table as a streaming source

In the following PySpark example, a Delta Lake table is used to store details of Internet sales orders. A stream is created that reads data from the Delta Lake table folder as new data is appended.

Python

```
from pyspark.sql.types import *
from pyspark.sql.functions import *

# Load a streaming dataframe from the Delta Table
stream_df = spark.readStream.format("delta") \
    .option("ignoreChanges", "true") \
    .load("/delta/internetorders")

# Now you can process the streaming data in the dataframe
# for example, show it:
stream_df.show()
```

> **Note**
>
> When using a Delta Lake table as a streaming source, only *append* operations can be included in the stream. Data modifications will cause an error unless you specify the `ignoreChanges` or `ignoreDeletes` option.

After reading the data from the Delta Lake table into a streaming dataframe, you can use the Spark Structured Streaming API to process it. In the example above, the dataframe is simply displayed; but you could use Spark Structured Streaming to aggregate the data over temporal windows (for example to count the number of orders placed every minute) and send the aggregated results to a downstream process for near-real-time visualization.

Using a Delta Lake table as a streaming sink

In the following PySpark example, a stream of data is read from JSON files in a folder. The JSON data in each file contains the status for an IoT device in the format `{"device":"Dev1","status":"ok"}` New data is added to the stream whenever a file is added to the folder. The input stream is a boundless dataframe, which is then written in delta format to a folder location for a Delta Lake table.

Python

```
from pyspark.sql.types import *
from pyspark.sql.functions import *

# Create a stream that reads JSON data from a folder
```

```
streamFolder = '/streamingdata/'
jsonSchema = StructType([
    StructField("device", StringType(), False),
    StructField("status", StringType(), False)
])
stream_df =
spark.readStream.schema(jsonSchema).option("maxFilesPerTrigger",
1).json(inputPath)

# Write the stream to a delta table
table_path = '/delta/devicetable'
checkpoint_path = '/delta/checkpoint'
delta_stream = stream_df.writeStream.format("delta").option("checkpointLo-
cation", checkpoint_path).start(table_path)
```

> **Note**
>
> The `checkpointLocation` option is used to write a checkpoint file that tracks the state of the stream processing. This file enables you to recover from failure at the point where stream processing left off.

After the streaming process has started, you can query the Delta Lake table to which the streaming output is being written to see the latest data. For example, the following code creates a catalog table for the Delta Lake table folder and queries it:

SQL

```
%sql

CREATE TABLE DeviceTable
USING DELTA
LOCATION '/delta/devicetable';

SELECT device, status
FROM DeviceTable;
```

To stop the stream of data being written to the Delta Lake table, you can use the `stop` method of the streaming query:

Python

```
delta_stream.stop()
```

> **Tip**

For more information about using Delta Lake tables for streaming data, see **Table streaming reads and writes** in the Delta Lake documentation.

Next unit: Exercise - Use Delta Lake in Azure Databricks

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

Knowledge check

3 minutes

1. Which of the following descriptions best fits Delta Lake? *

- ○ A Spark API for exporting data from a relational database into CSV files.

  ✗ Incorrect. Delta Lake does not export data from a relational database into CSV files.

- ○ A relational storage layer for Spark that supports tables based on Parquet files.

  ✓ Correct. Delta Lake provides a relational storage layer in which you can create tables based on Parquet files in a data lake.

- ○ A synchronization solution that replicates data between SQL Server and Spark clusters.

2. You've loaded a Spark dataframe with data, that you now want to use in a Delta Lake table. What format should you use to write the dataframe to storage? *

- ○ CSV
- ○ PARQUET

  ✗ Incorrect. Although Delta Lake tables are based on Parquet files, the format must also include a transaction log.

- ○ DELTA

  ✓ Correct. Storing a dataframe in DELTA format creates parquet files for the data and the transaction log metadata necessary for Delta Lake tables.

3. What feature of Delta Lake enables you to retrieve data from previous versions of a table? *

- ○ Spark Structured Streaming
- ○ Time Travel

  ✓ Correct. The Time Travel feature is based on the transaction log, which enables you to specify a version number or timestamp for the data you want to retrieve.

- ○ Catalog Tables

Catalog Tables

4. You have a managed catalog table that contains Delta Lake data. If you drop the table, what will happen? *

- The table metadata and data files will be deleted.

  ✔ **Correct. The life-cycle of the metadata and data for a managed table are the same.**

- The table metadata will be removed from the catalog, but the data files will remain intact.

  ✕ **Incorrect. The life-cycle of the metadata and data for a managed table are the same.**

- The table metadata will remain in the catalog, but the data files will be deleted.

5. When using Spark Structured Streaming, a Delta Lake table can be which of the following? *

- Only a source
- Only a sink

  ✕ **Incorrect. A Delta Lake table can be a source or a sink.**

- Either a source or a sink

  ✔ **Correct. A Delta Lake table can be a source or a sink.**

Next unit: Summary

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Introduction

1 minute

Data analysts often use SQL to query relational data and create reports and dashboards. Azure Databricks provides support for SQL-based data analytics through SQL Warehouses and the SQL persona.

In this module, you'll learn how to:

- Create and configure SQL Warehouses in Azure Databricks.
- Create databases and tables.
- Create queries and dashboards.

> **Note**
>
> SQL Warehouses and the SQL persona are available in *premium-tier* Azure Databricks workspaces.

Next unit: Get started with SQL Warehouses

Continue

100 XP

Get started with SQL Warehouses

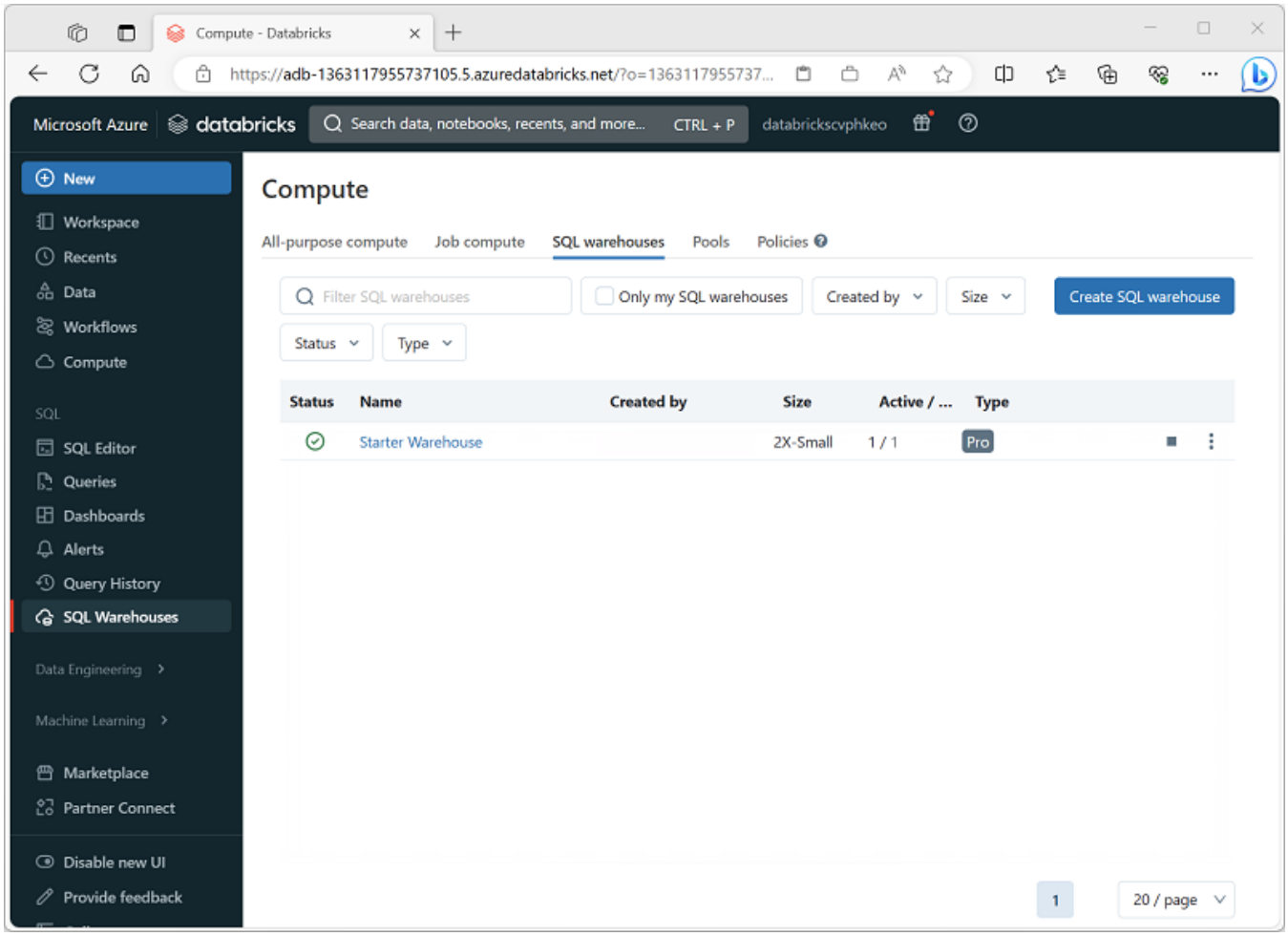
3 minutes

SQL Warehouses (formerly known as SQL Endpoints) provide a relational database interface for data in Azure Databricks. The data is stored in files that are abstracted by Delta tables in a hive metastore, but from the perspective of the user or client application, the SQL Warehouse behaves like a relational database.

Creating a SQL Warehouse

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.

You can manage the SQL Warehouses in your Azure Databricks workspace by using the Azure Databricks portal in the **SQL** persona view.



SQL Warehouse configuration settings

When you create or configure a SQL Warehouse, you can specify the following settings:

- **Name**: A name used to identify the SQL Warehouse.
- **Cluster size**: Choose from a range of standard sizes to control the number and size of compute resources used to support the SQL Warehouse. Available sizes range from *2X-Small* (a single worker node) to *4X-Large* (256 worker nodes). For more information, see Cluster size in the Azure Databricks documentation.
- **Auto Stop**: The amount of time the cluster will remain running when idle before being stopped. Idle clusters continue to incur charges when running.
- **Scaling**: The minimum and maximum number of clusters used to distribute query processing.
- **Type**: You can create a SQL Warehouse that uses *serverless* compute for fast, cost-effective on-demand provisioning. Alternatively, you can create a *Pro* or *Classic* SQL warehouse.

> **Note**
>
> You can create a SQL Warehouse with any available size, but if you have insufficient quota for the number of cores required to support your choice in the region where Azure

Databricks is provisioned, the SQL Warehouse will fail to start.

Next unit: Create databases and tables

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Create databases and tables

3 minutes

After creating and starting a SQL Warehouse, you can start to work with data in tables.

Database schema

All SQL Warehouses contain a default database schema named **default**. You can use create tables in this schema in order to analyze data. However, if you need to work with multiple tables in a relational schema, or you have multiple analytical workloads where you want to manage the data (and access to it) separately, you can create custom database schema. To create a database, use the SQL editor to run a `CREATE DATABASE` or `CREATE SCHEMA` SQL statement. These statements are equivalent, but `CREATE SCHEMA` is preferred, as shown in this example:

SQL

```
CREATE SCHEMA salesdata;
```

Tip

For more information, see **CREATE SCHEMA** in the Azure Databricks documentation.

Tables

You can use the user interface in the Azure Databricks portal to upload delimited data, or import data from a wide range of common data sources. The imported data is stored in files in Databricks File System (DBFS) storage, and a Delta table is defined for it in the Hive metastore.

If the data files already exist in storage, or you need to define an explicit schema for the table, you can use a `CREATE TABLE` SQL statement. For example, the following code creates a table named **salesorders** in the **salesdata** database, based on the */data/sales/* folder in DBFS storage.

SQL

```
CREATE TABLE salesdata.salesorders
(
    orderid INT,
    orderdate DATE,
    customerid INT,
    ordertotal DECIMAL
)
USING DELTA
LOCATION '/data/sales/';
```

Tip

For more information, see **CREATE TABLE** in the Azure Databricks documentation.

Next unit: Create queries and dashboards

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

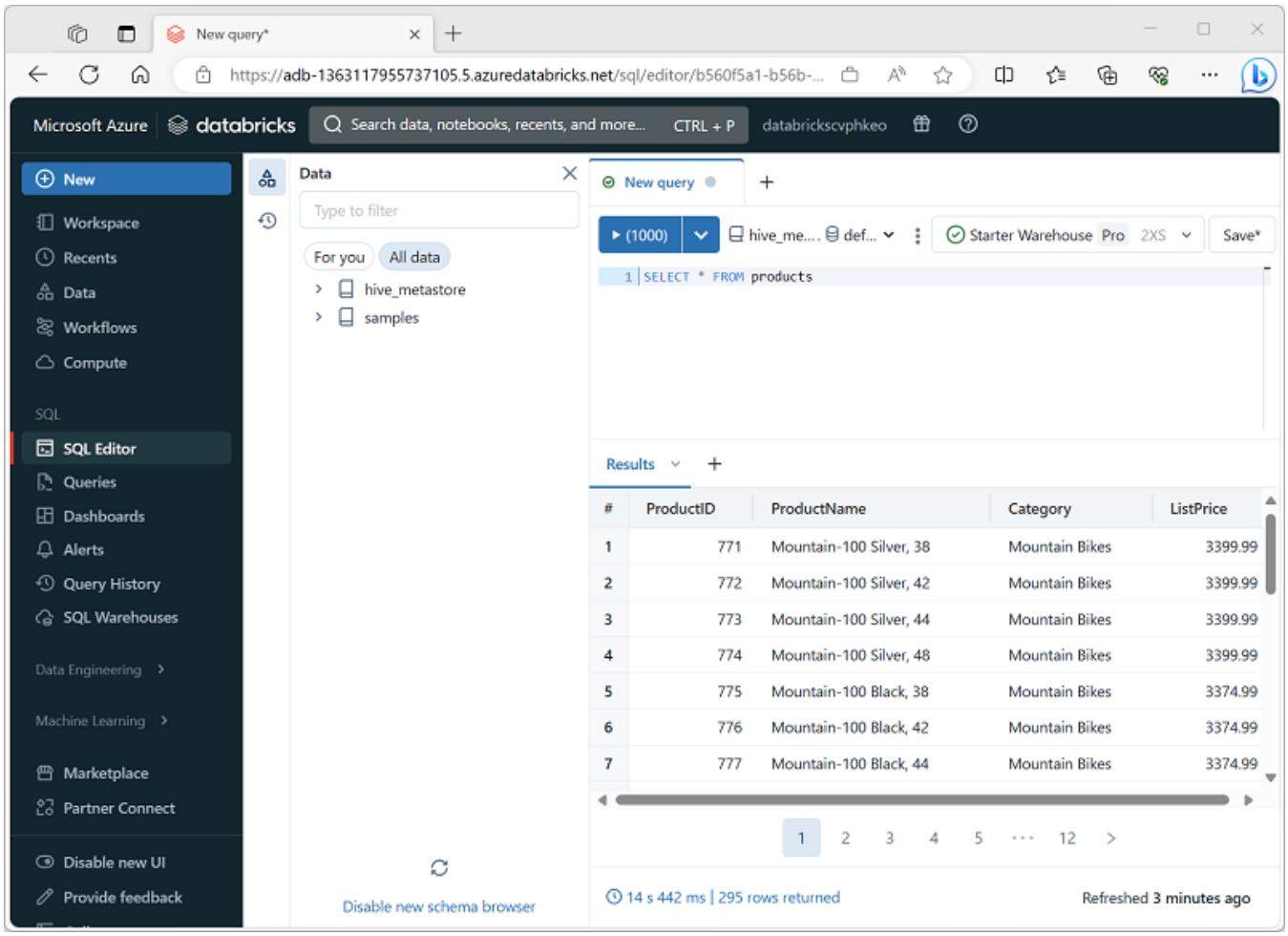
Create queries and dashboards

3 minutes

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.

Queries

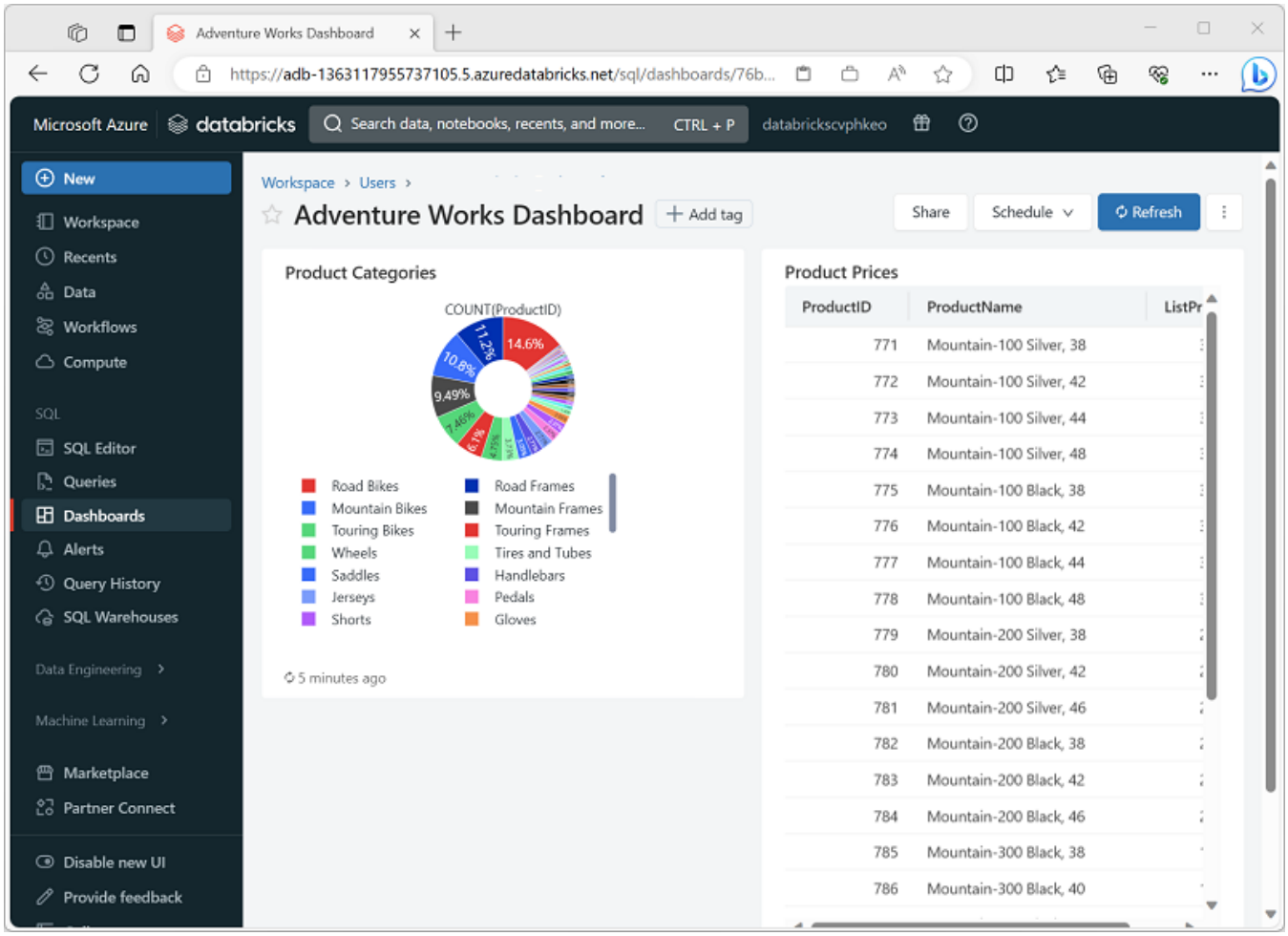
You can use the SQL Editor in the Azure Databricks portal to create a query based on any valid SQL `SELECT` statement, and then save the query with a meaningful name to be retrieved and run later.



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.

Dashboards

Dashboards enable you to display the results of queries, either as tables of data or as graphical visualizations.



You can create multiple visualizations in a dashboard and share it with users in your organization. As with individual queries, you can schedule the dashboard to refresh is data periodically, and notify subscribers by email that new data is available.

Next unit: Exercise - Use a SQL Warehouse in Azure Databricks

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Exercise - Use a SQL Warehouse in Azure Databricks

30 minutes

Now it's your chance to explore Azure Databricks SQL for yourself. In this exercise, you'll use a SQL Warehouse in Azure Databricks to query tables and create a dashboard.

> **Note**
>
> To complete this lab, you will need an **Azure subscription** in which you have administrative access.

Launch the exercise and follow the instructions.

Launch Exercise

Next unit: Knowledge check

Continue

200 XP

Knowledge check

3 minutes

1. Which of the following workloads is best suited for Azure Databricks SQL? *

- ○ Running Scala code in notebooks to transform data.
  - ✗ Incorrect. Using notebooks to transform data is more suited to the Azure Databricks Data Science and Engineering persona.
- ○ Querying and visualizing data in relational tables.
  - ✓ Correct. Azure Databricks SQL is optimized for SQL-based querying and data visualization.
- ○ Training and deploying machine learning models.

2. Which statement should you use to create a database in a SQL warehouse? *

- ○ CREATE VIEW
- ○ CREATE SCHEMA
  - ✓ Correct. The CREATE SCHEMA statement is used to create a database.
- ○ CREATE GROUP

3. You need to share data visualizations, including charts and tables of data, with users in your organization. What should you create? *

- ○ A table
  - ✗ Incorrect. Creating a table does not enable you to share data visualizations.
- ○ A query
- ○ A dashboard
  - ✓ Correct. A dashboard can be used to share data visualizations with other users.

100 XP

Introduction

1 minute

Azure Databricks enables data engineers to use code in notebooks to ingest and process data. While notebooks are designed to be used interactively, you can use them to encapsulate activities in a data ingestion or processing pipeline that is orchestrated using Azure Data Factory.

In this module, you'll learn how to:

- Describe how Azure Databricks notebooks can be run in a pipeline.
- Create an Azure Data Factory linked service for Azure Databricks.
- Use a Notebook activity in a pipeline.
- Pass parameters to a notebook.

> **Note**
>
> While this module focuses on using Azure Databricks notebooks in an Azure Data Factory pipeline, the same principles and techniques apply when using an Azure Synapse Analytics pipeline.

Next unit: Understand Azure Databricks notebooks and pipelines

Continue

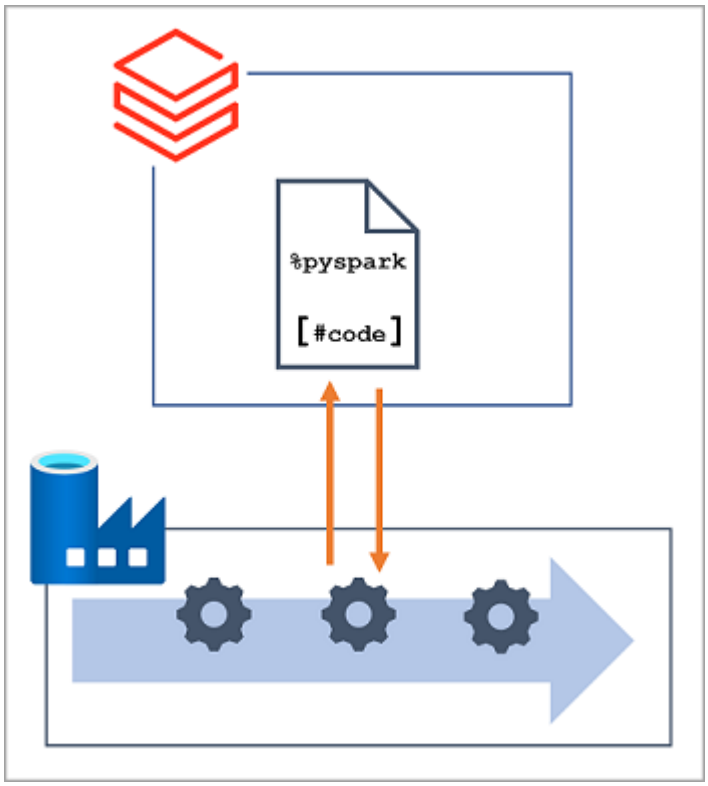
100 XP

Understand Azure Databricks notebooks and pipelines

3 minutes

In Azure Databricks, you can use notebooks to run code written in Python, Scala, SQL, and other languages to ingest and process data. Notebooks provide an interactive interface in which you can run individual code cells and use Markdown to include notes and annotations.

In many data engineering solutions, code that is written and tested interactively can later be incorporated into an automated data processing workload. On Azure, such workloads are often implemented as *pipelines* in Azure Data Factory, in which one or more *activities* are used to orchestrate a series of tasks that can be run on-demand, at scheduled intervals, or in response to an event (such as new data being loaded into a folder in a data lake). Azure Data Factory supports a **Notebook** activity that can be used to automate the unattended execution of a notebook in an Azure Databricks workspace.



Note

The same **Notebook** activity is available in pipelines built in Azure Synapse Analytics.

100 XP

Create a linked service for Azure Databricks

5 minutes

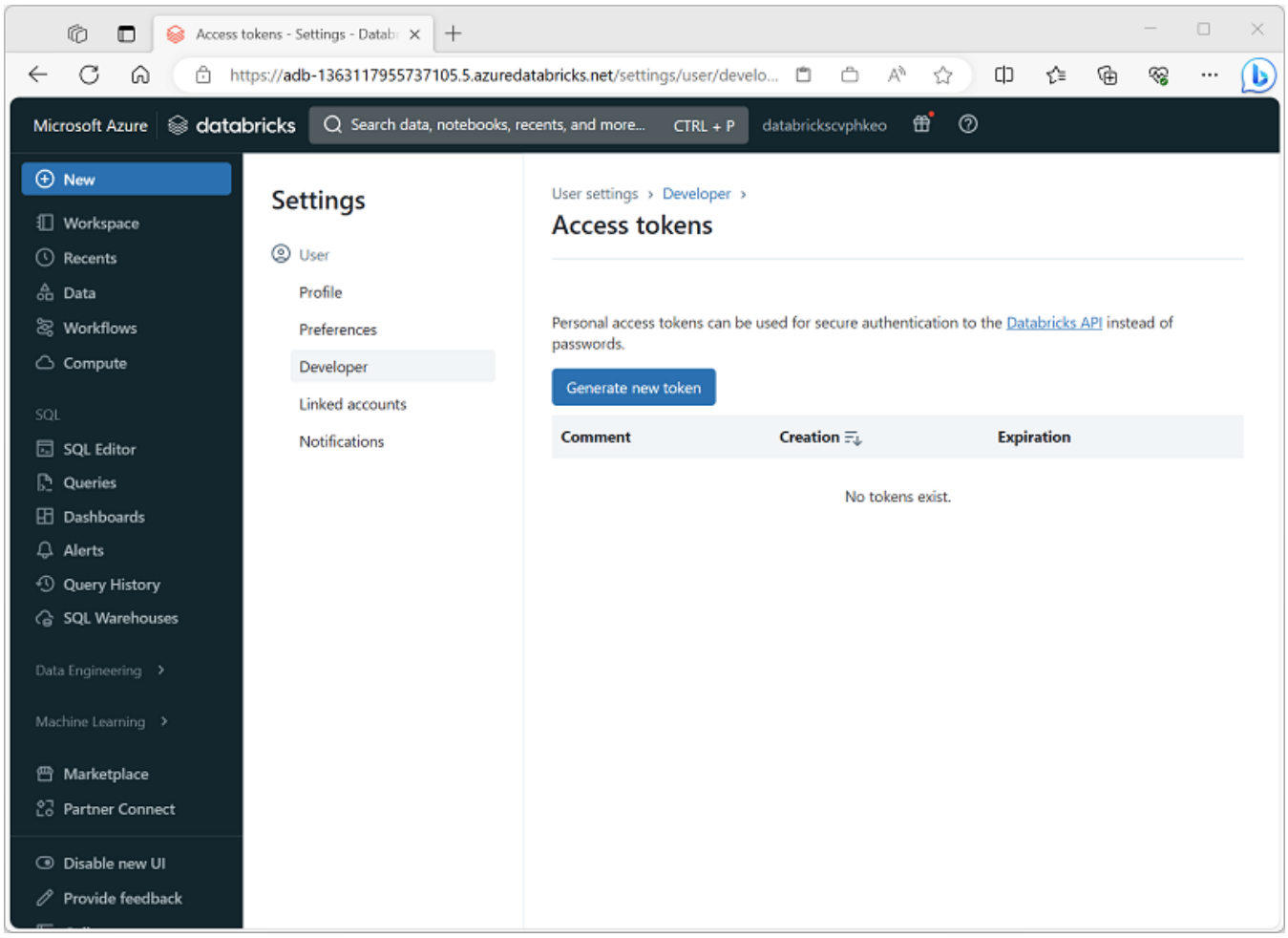
To run notebooks in an Azure Databricks workspace, the Azure Data Factory pipeline must be able to connect to the workspace; which requires authentication. To enable this authenticated connection, you must perform two configuration tasks:

1. Generate an *access token* for your Azure Databricks workspace.
2. Create a *linked service* in your Azure Data Factory resource that uses the access token to connect to Azure Databricks.

Generating an access token

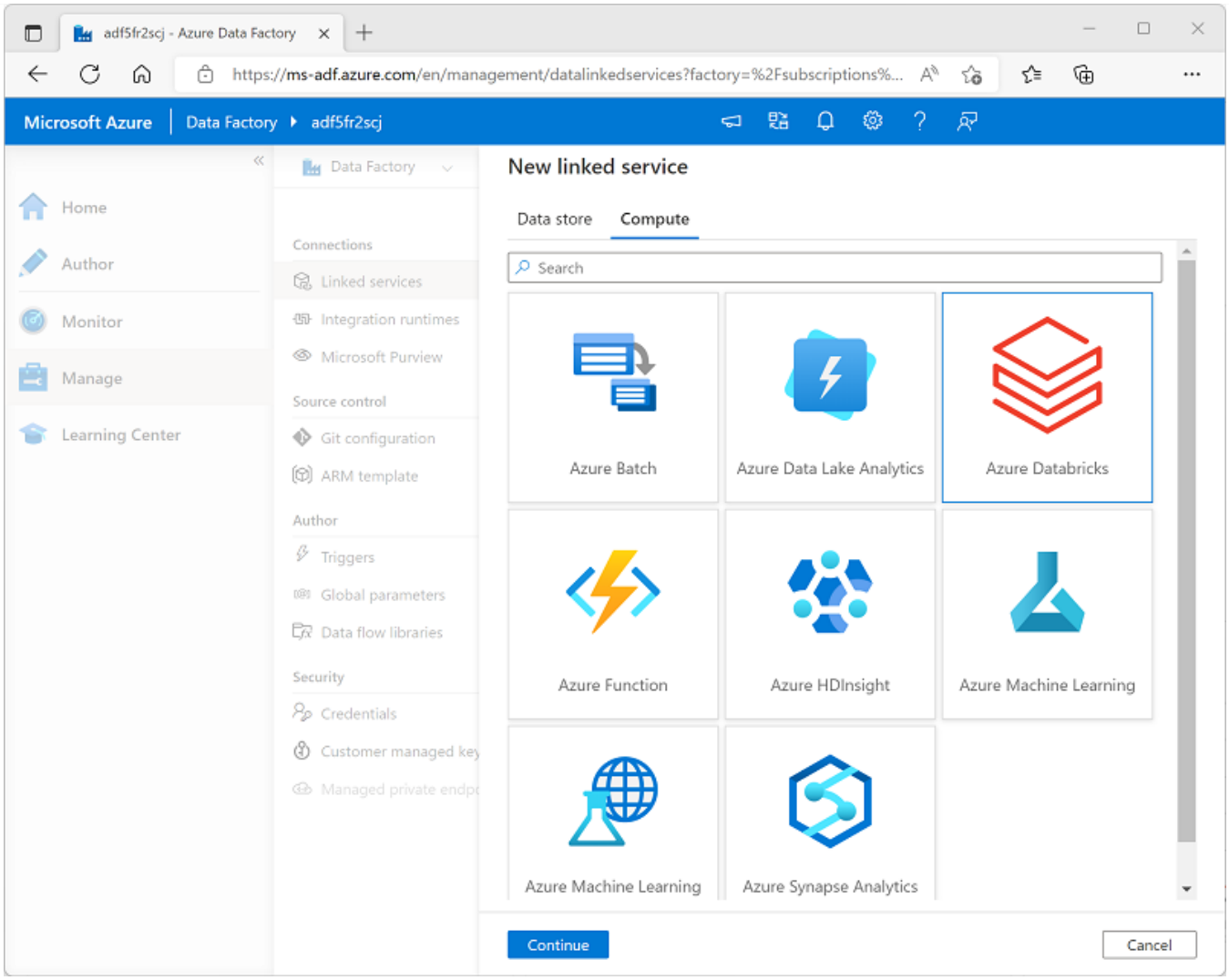
An access token provides an authentication method for Azure Databricks as an alternative to credentials on the form of a user name and password. You can generate access tokens for applications, specifying an expiration period after which the token must be regenerated and updated in the client applications.

To create an Access token, use the **Generate new token** option on the **Developer** tab of the **User Settings** page in Azure Databricks portal.



Creating a linked service

To connect to Azure Databricks from Azure Data Factory, you need to create a linked service for **Azure Databricks** compute. You can create a linked service in the **Linked services** page in the **Manage** section of Azure Data Factory Studio.



When you create an **Azure Databricks** linked service, you must specify the following configuration settings:

Setting	Description
Name	A unique name for the linked service
Description	A meaningful description
Integration runtime	The integration runtime used to run activities in this linked service. See Integration runtime in Azure Data Factory for more details.
Azure subscription	The Azure subscription in which Azure Databricks is provisioned
Databricks workspace	The Azure Databricks workspace

Setting	Description
Cluster	The Spark cluster on which activity code will be run. You can have Azure Databricks dynamically provision a *job cluster* on-demand or you can specify an existing cluster in the workspace.
Authentication type	How the linked connection will be authenticated by Azure Databricks. For example, using an access token (in which case, you need to specify the access token you generated for your workspace).
Cluster configuration	The Databricks runtime version, Python version, worker node type, and number of worker nodes for your cluster.

Next unit: Use a Notebook activity in a pipeline

Continue >

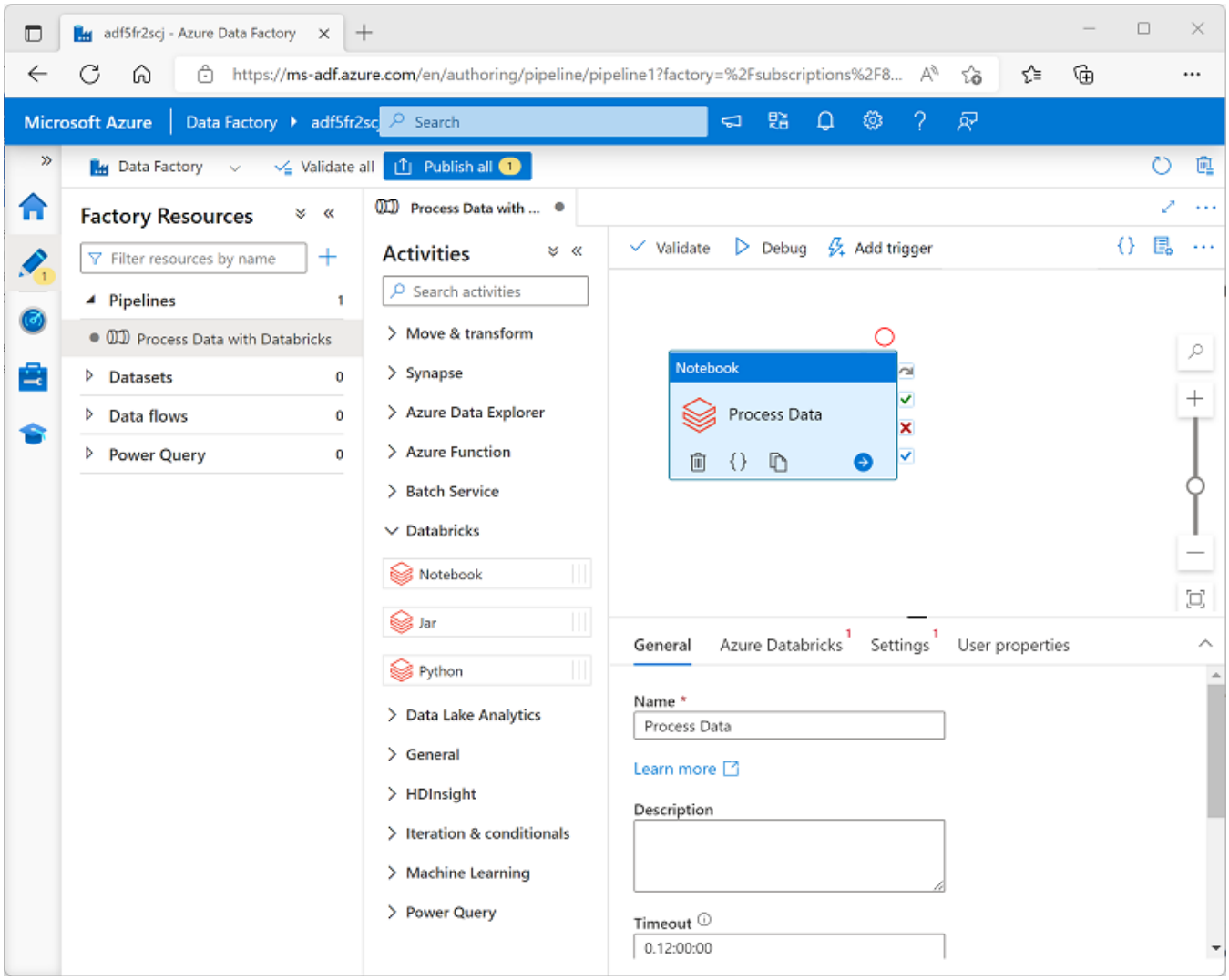
100 XP

Use a Notebook activity in a pipeline

5 minutes

After you've created a linked service in Azure Data Factory for your Azure Databricks workspace, you can use it to define the connection for a **Notebook** activity in a pipeline.

To use a **Notebook** activity, create a pipeline and from the **Databricks** category, add a **Notebook** activity to the pipeline designer surface.



Use the following properties of the **Notebook** activity to configure it:

Category	Setting	Descriptions
General	Name	A unique name for the activity.
	Description	A meaningful description.

Category	Setting	Descriptions
	Timeout	How long the activity should run before automatically canceling.
	Retries	How many times should Azure Data Factory try before failing.
	Retry interval	How long to wait before retrying.
	Secure input and output	Determines if input and output values are logged.
Azure Databricks	Azure Databricks linked service	The linked service for the Azure Databricks workspace containing the notebook.
Settings	Notebook path	The path to the notebook file in the Workspace.
	Base parameters	Used to pass parameters to the notebook.
	Append libraries	Required code libraries that aren't installed by default.
User properties		Custom user-defined properties.

Running a pipeline

When the pipeline containing the **Notebook** activity is published, you can run it by defining a trigger. You can then monitor pipeline runs in the **Monitor** section of Azure Data Factory Studio.

Next unit: Use parameters in a notebook

Continue >

100 XP

Use parameters in a notebook

3 minutes

You can use parameters to pass variable values to a notebook from the pipeline. Parameterization enables greater flexibility than using hard-coded values in the notebook code.

Using parameters in a notebook

To define and use parameters in a notebook, use the **dbutils.widgets** library in your notebook code.

For example, the following Python code defines a variable named **folder** and assigns a default value of *data*:

Python

```
dbutils.widgets.text("folder", "data")
```

To retrieve a parameter value, use the **get** function, like this:

Python

```
folder = dbutils.widgets.get("folder")
```

The **get** function will retrieve the value for the specific parameter that was passed to the notebook. If no such parameter was passed, it will get the default value of the variable you declared previously.

Passing output values

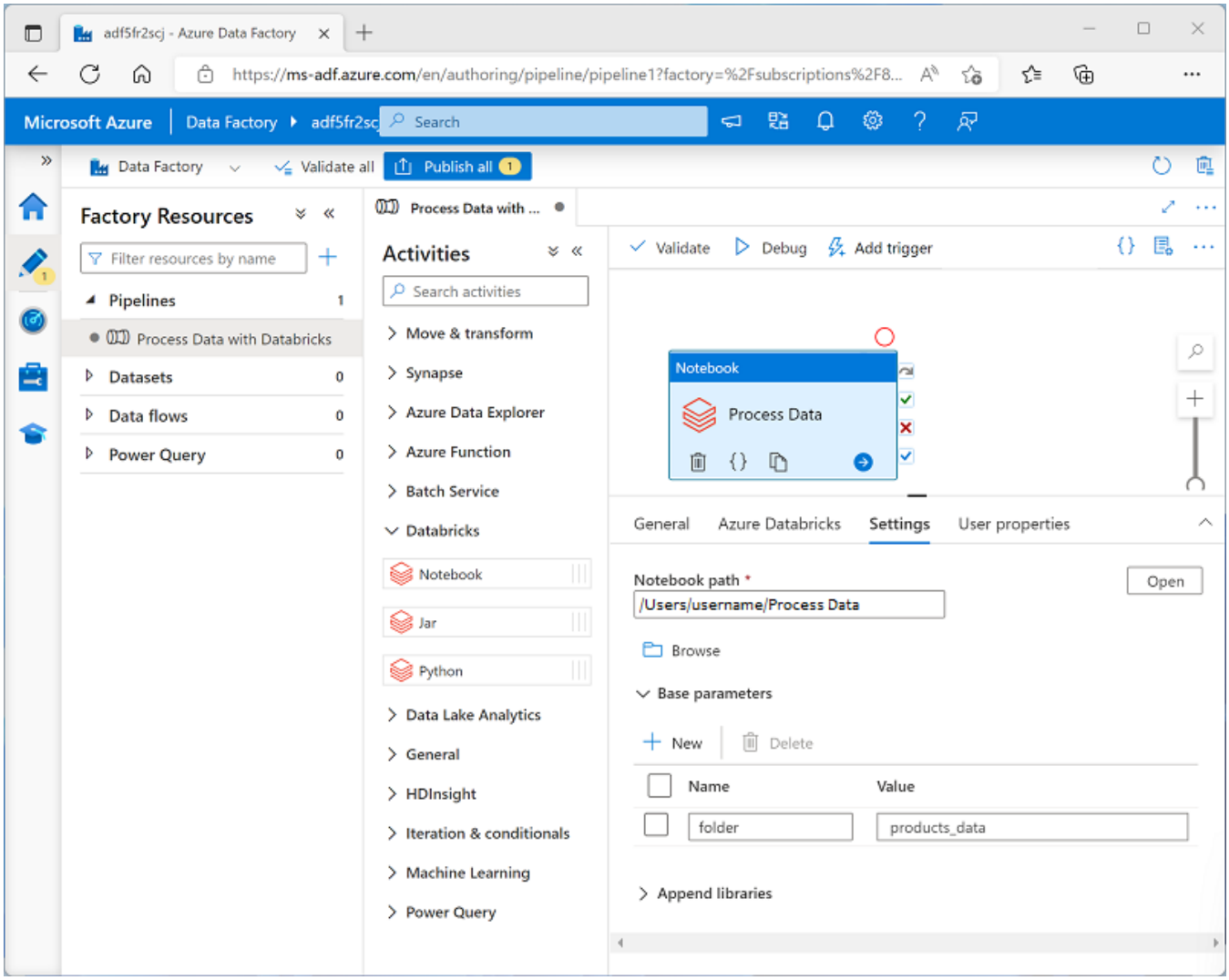
In addition to using parameters that can be passed *in* to a notebook, you can pass values *out* to the calling application by using the **notebook.exit** function, as shown here:

Python

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

Setting parameter values in a pipeline

To pass parameter values to a **Notebook** activity, add each parameter to the activity's **Base parameters**, as shown here:



In this example, the parameter value is explicitly specified as a property of the **Notebook** activity. You could also define a *pipeline* parameter and assign its value dynamically to the **Notebook** activity's base parameter; adding a further level of abstraction.

> **Tip**
>
> For more information about using parameters in Azure Data Factory, see **How to use parameters, expressions and functions in Azure Data Factory** in the Azure Data Factory documentation.

Next unit: Exercise - Run an Azure Databricks Notebook with Azure Data Factory

200 XP

Knowledge check

3 minutes

1. You want to connect to an Azure Databricks workspace from Azure Data Factory. What must you define in Azure Data Factory? *

- [ ] A global parameter
- [x] A linked service

✔ Correct. An Azure Databricks linked service is required to connect to an Azure Databricks workspace.

- [ ] A customer managed key

2. You need to run a notebook in the Azure Databricks workspace referenced by a linked service. What type of activity should you add to a pipeline? *

- [x] Notebook

✔ Correct. Use a Notebook activity to run an Azure Databricks notebook.

- [ ] Python
- [ ] Jar

3. You need to use a parameter in a notebook. Which library should you use to define parameters with default values and get parameter values that are passed to the notebook? *

- [ ] notebook

✘ Incorrect. You can use the notebook library to exit the notebook and return an output value.

- [ ] argparse
- [x] dbutils.widget

✔ Correct. Use the dbutils.widget library to define and read parameters in an Azure Databricks notebook.