

Use Apache Spark in Microsoft Fabric

1. Introduction

<https://learn.microsoft.com/en-us/training/modules/use-apache-spark-work-files-lakehouse/1-introduction>

Introduction

Completed

Apache Spark is an open source parallel processing framework for large-scale data processing and analytics. Spark has become popular in "big data" processing scenarios, and is available in multiple platform implementations; including Azure HDInsight, Azure Synapse Analytics, and Microsoft Fabric.

This module explores how you can use Spark in Microsoft Fabric to ingest, process, and analyze data in a lakehouse. While the core techniques and code described in this module are common to all Spark implementations, the integrated tools and ability to work with Spark in the same environment as other data services in Microsoft Fabric makes it easier to incorporate Spark-based data processing into your overall data analytics solution.

2. Prepare to use Apache Spark

<https://learn.microsoft.com/en-us/training/modules/use-apache-spark-work-files-lakehouse/2-spark>

Prepare to use Apache Spark

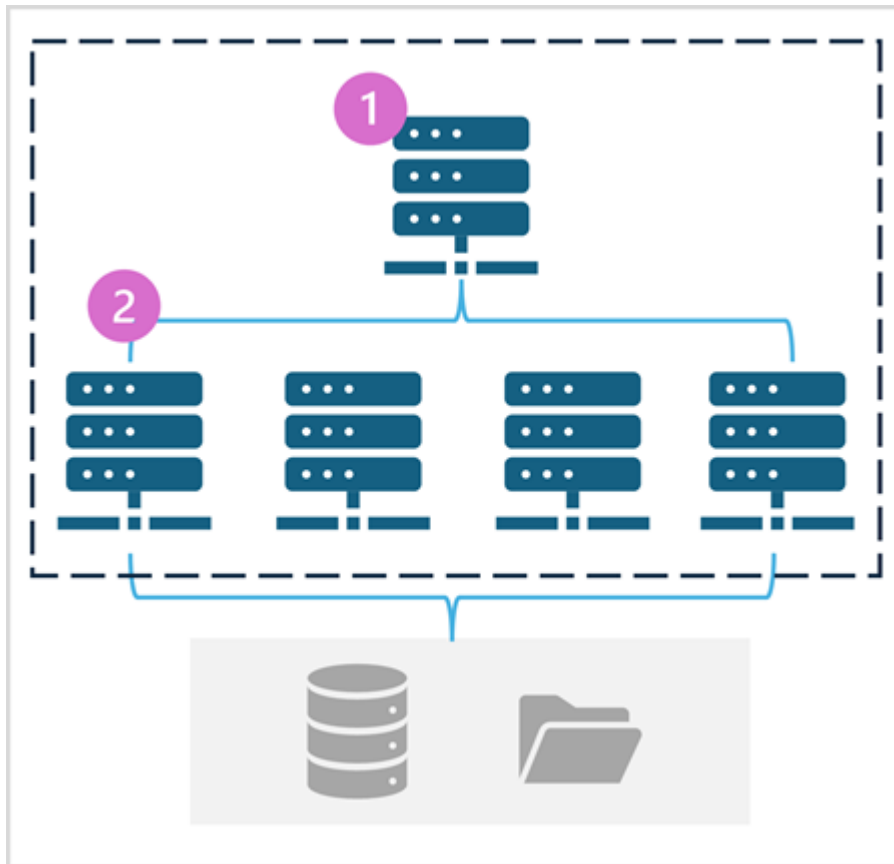
Completed

Apache Spark is a distributed data processing framework that enables large-scale data analytics by coordinating work across multiple processing nodes in a cluster, known in Microsoft Fabric as a *Spark pool*. Put more simply, Spark uses a "divide and conquer" approach to processing large volumes of data quickly by distributing the work across multiple computers. The process of distributing tasks and collating results is handled for you by Spark.

Spark can run code written in a wide range of languages, including Java, Scala (a Java-based scripting language), Spark R, Spark SQL, and PySpark (a Spark-specific variant of Python). In practice, most data engineering and analytics workloads are accomplished using a combination of PySpark and Spark SQL.

Spark pools

A Spark pool consists of compute *nodes* that distribute data processing tasks. The general architecture is shown in the following diagram.



As shown in the diagram, a Spark pool contains two kinds of node:

1. A *head* node in a Spark pool coordinates distributed processes through a *driver* program.
2. The pool includes multiple *worker* nodes on which *executor* processes perform the actual data processing tasks.

The Spark pool uses this distributed compute architecture to access and process data in a compatible data store - such as a data lakehouse based in OneLake.

Spark pools in Microsoft Fabric

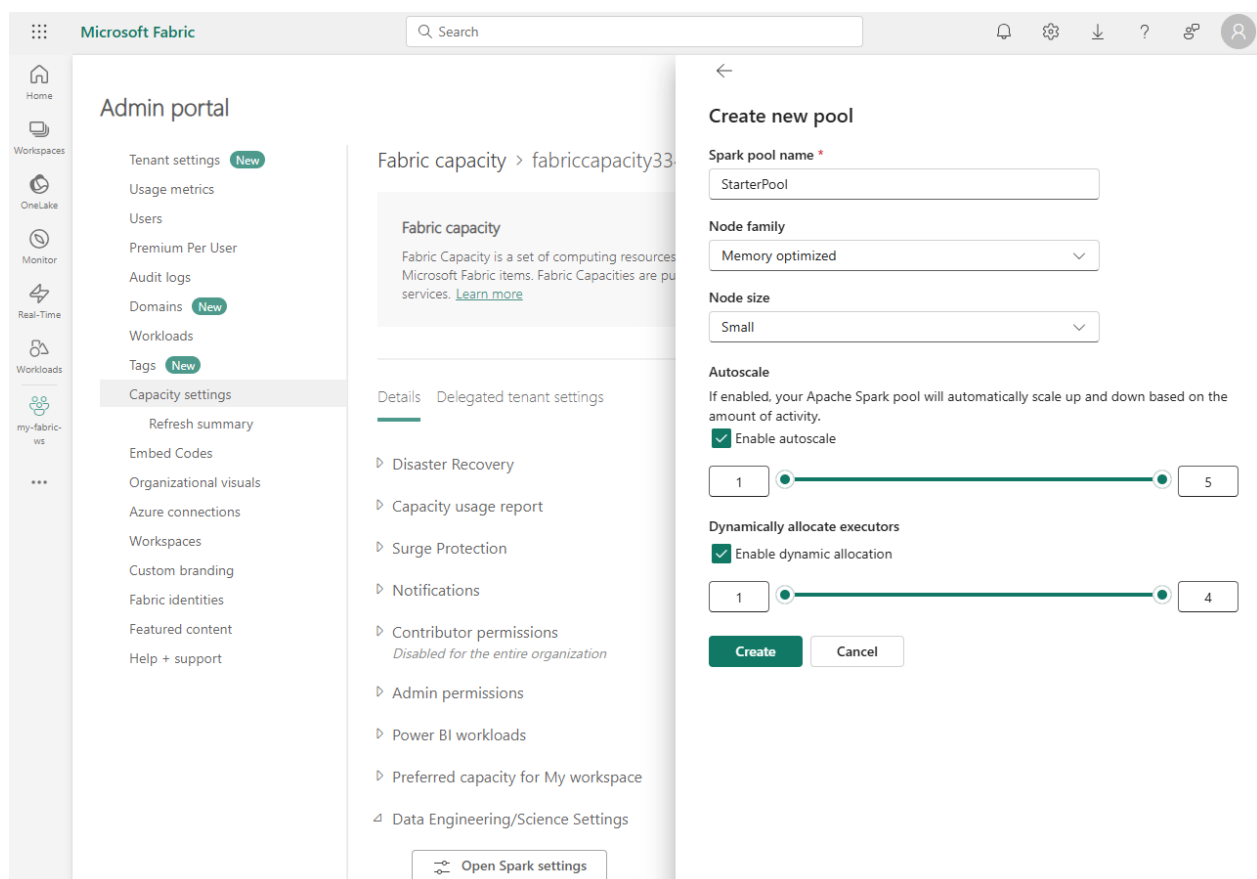
Microsoft Fabric provides a *starter pool* in each workspace, enabling Spark jobs to be started and run quickly with minimal setup and configuration. You can configure the starter pool to optimize the nodes it contains in accordance with your specific workload needs or cost constraints.

Additionally, you can create custom Spark pools with specific node configurations that support your particular data processing needs.

Note

The ability to customize Spark pool settings can be disabled by Fabric administrators at the Fabric Capacity level. For more information, see [Capacity administration settings for Data Engineering and Data Science](#) in the Fabric documentation.

You can manage settings for the starter pool and create new Spark pools in the **Admin portal** section of the workspace settings, under **Capacity settings**, then **Data Engineering/Science Settings**.



Specific configuration settings for Spark pools include:

- **Node Family:** The type of virtual machines used for the Spark cluster nodes. In most cases, *memory optimized* nodes provide optimal performance.
- **Autoscale:** Whether or not to automatically provision nodes as needed, and if so, the initial and maximum number of nodes to be allocated to the pool.
- **Dynamic allocation:** Whether or not to dynamically allocate *executor* processes on the worker nodes based on data volumes.

If you create one or more custom Spark pools in a workspace, you can set one of them (or the starter pool) as the default pool to be used if a specific pool is not specified for a given Spark job.

Tip

For more information about managing Spark pools in Microsoft Fabric, see [Configuring starter pools in Microsoft Fabric](#) and [How to create custom Spark pools in Microsoft Fabric](#) in the Microsoft Fabric documentation.

Runtimes and environments

The Spark open source ecosystem includes multiple versions of the Spark *runtime*, which determines the version of Apache Spark, Delta Lake, Python, and other core software components that are installed. Additionally, within a runtime you can install and use a wide selection of code libraries for common (and sometimes very specialized) tasks. Since a great deal of Spark processing is performed using PySpark, the huge range of Python libraries ensures that whatever the task you need to perform, there's probably a library to help.

In some cases, organizations may need to define multiple *environments* to support a diverse range of data processing tasks. Each environment defines a specific runtime version as well as the libraries that must be installed to perform specific operations. Data engineers and scientists can then select which environment they want to use with a Spark pool for a particular task.

Spark runtimes in Microsoft Fabric

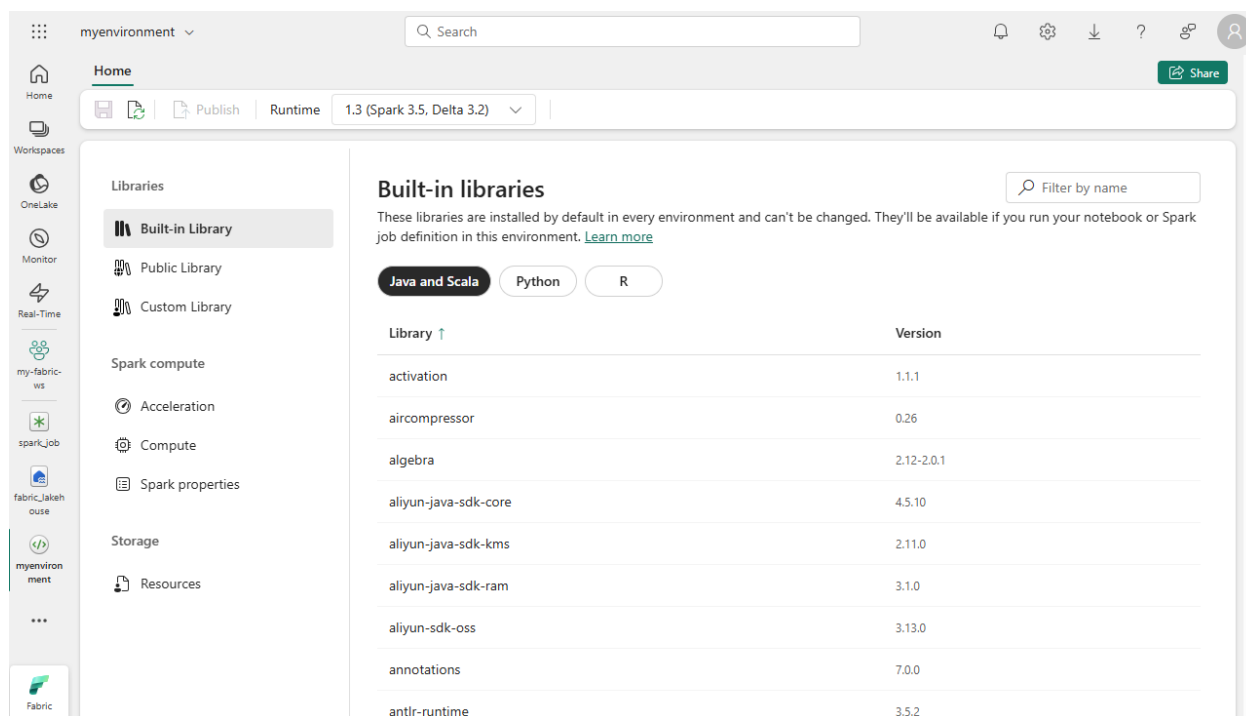
Microsoft Fabric supports multiple Spark runtimes, and will continue to add support for new runtimes as they are released. You can use the workspace settings interface to specify the Spark runtime that is used by default environment when a Spark pool is started.

Tip

For more information about Spark runtimes in Microsoft Fabric, see [Apache Spark Runtimes in Fabric](#) in the Microsoft Fabric documentation.

Environments in Microsoft Fabric

You can create custom environments in a Fabric workspace, enabling you to use specific Spark runtimes, libraries, and configuration settings for different data processing operations.



When creating an environment, you can:

- Specify the Spark runtime it should use.
- View the built-in libraries that are installed in every environment.
- Install specific public libraries from the Python Package Index (PyPI).
- Install custom libraries by uploading a package file.
- Specify the Spark pool that the environment should use.
- Specify Spark configuration properties to override default behavior.
- Upload resource files that need to be available in the environment.

After creating at least one custom environment, you can specify it as the default environment in the workspace settings.

Tip

For more information about using custom environments in Microsoft Fabric, see [Create, configure, and use an environment in Microsoft Fabric](#) in the Microsoft Fabric documentation.

Additional Spark configuration options

Managing Spark pools and environments are the primary ways in which you can manage Spark processing in a Fabric workspace. However, there are some additional options that you can use to make further optimizations.

Native execution engine

The *native execution engine* in Microsoft Fabric is a vectorized processing engine that runs Spark operations directly on lakehouse infrastructure. Using the native execution engine can significantly improve the performance of queries when working with large data sets in Parquet or Delta file formats.

To use the native execution engine, you can enable it at the environment level or within an individual notebook. To enable the native execution engine at the environment level, set the following Spark properties in the environment configuration:

- **spark.native.enabled**: true
- **spark.shuffle.manager**: org.apache.spark.shuffle.sort.ColumnarShuffleManager

To enable the native execution engine for a specific script or notebook, you can set these configuration properties at the beginning of your code, like this:

```
%%configure
{
  "conf": {
    "spark.native.enabled": "true",
    "spark.shuffle.manager": "org.apache.spark.shuffle.sort.ColumnarShuffleManager"
  }
}
```

Tip

For more information about the native execution engine, see [Native execution engine for Fabric Spark](#) in the Microsoft Fabric documentation.

High concurrency mode

When you run Spark code in Microsoft Fabric, a Spark session is initiated. You can optimize the efficiency of Spark resource usage by using *high concurrency mode* to share Spark sessions across multiple concurrent users or processes. A notebook uses a Spark session for its execution. When high concurrency mode is enabled, multiple users can, for example, run code in notebooks that use the same Spark session, while ensuring isolation of code to avoid variables in one notebook being affected by code in another notebook. You can also enable high concurrency mode for Spark jobs, enabling similar efficiencies for concurrent non-interactive Spark script execution.

To enable high concurrency mode, use the **Data Engineering/Science** section of the workspace settings interface.

Tip

For more information about high concurrency mode, see [High concurrency mode in Apache Spark for Fabric](#) in the Microsoft Fabric documentation.

Automatic MLFlow logging

MLFlow is an open source library that is used in data science workloads to manage machine learning training and model deployment. A key capability of MLFlow is the ability to log model training and management operations. By default, Microsoft Fabric uses MLFlow to implicitly log machine learning experiment activity without requiring the data scientist to include explicit code to do so. You can disable this functionality in the workspace settings.

Spark administration for a Fabric capacity

Administrators can manage Spark settings at a Fabric capacity level, enabling them to restrict and override Spark settings in workspaces within an organization.

Tip

For more information about managing Spark configuration at the Fabric capacity level, see [Configure and manage data engineering and data science settings for Fabric capacities](#) in the Microsoft Fabric documentation.

3. Run Spark code

<https://learn.microsoft.com/en-us/training/modules/use-apache-spark-work-files-lakehouse/3-spark-code>

Run Spark code

Completed

To edit and run Spark code in Microsoft Fabric, you can use *notebooks*, or you can define a *Spark job*.

Notebooks

When you want to use Spark to explore and analyze data interactively, use a notebook. Notebooks enable you to combine text, images, and code written in multiple languages to create an interactive item that you can share with others and collaborate on.

Sales Analytics Notebook

Use this notebook to explore sales order data

```

1 from pyspark.sql.types import *
2
3 orderSchema = StructType([
4     StructField("SalesOrderNumber", StringType()),
5     StructField("SalesOrderLineNumber", IntegerType()),
6     StructField("OrderDate", DateType()),
7     StructField("CustomerName", StringType()),
8     StructField("Email", StringType()),
9     StructField("Item", StringType()),
10    StructField("Quantity", IntegerType()),
11    StructField("UnitPrice", FloatType()),
12    StructField("Tax", FloatType())
13 ])
14
15 df = spark.read.format("csv").schema(orderSchema).load("Files/orders/2019.csv")
16
17 display(df)

```

[2] ✓ 3 sec - Command executed in 3 sec 488 ms by System Administrator on 3:37:44 PM, 5/19/25

Spark jobs (1 of 1 succeeded) Resources Log

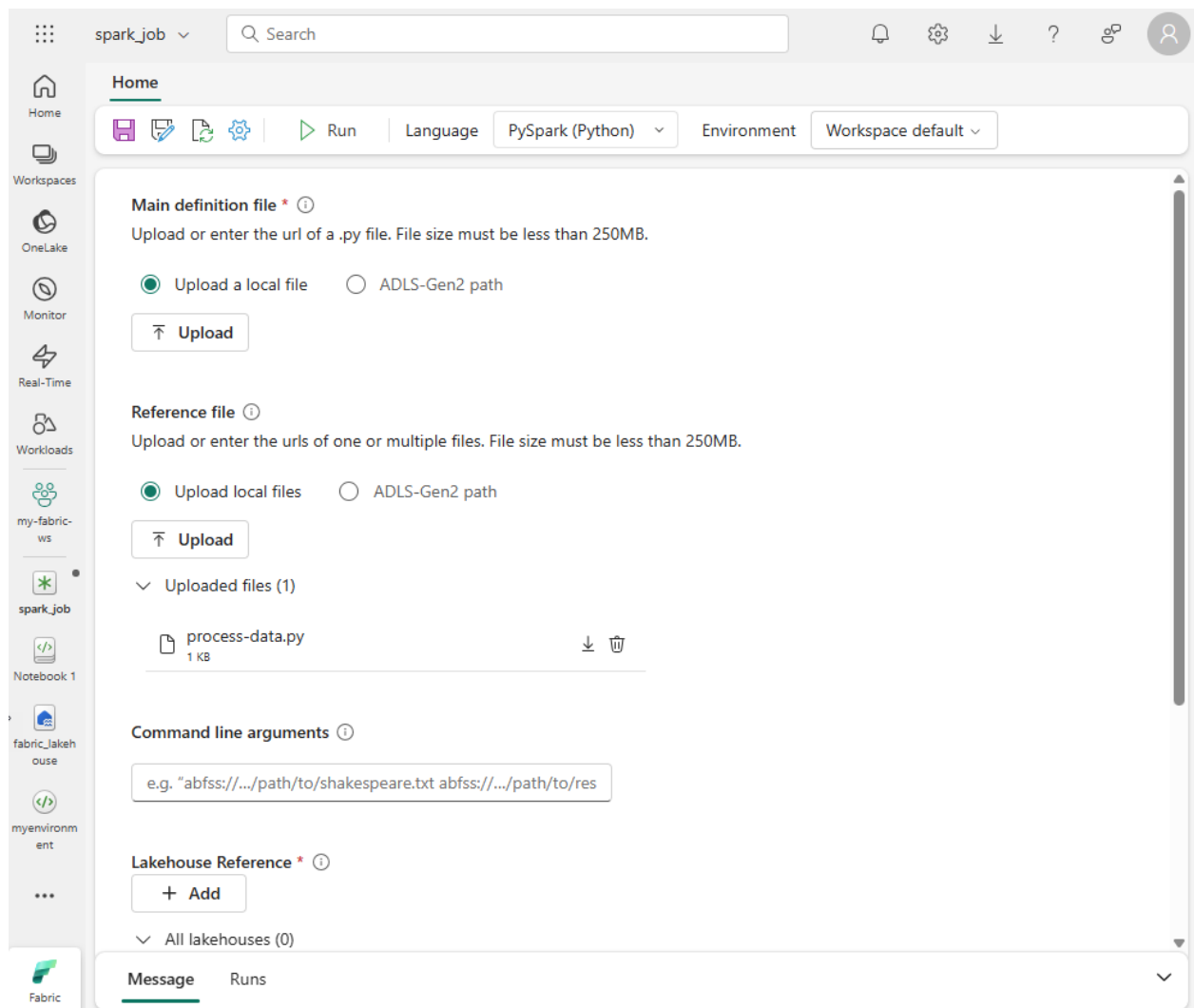
Table view

	ABC SalesOrderNumber	123 SalesOrderLineNumber	OrderDate	ABC CustomerName	ABC Email	ABC Item	123 Quanti
1	SO43701	1	2019-07-01	Christy Zhu	christy12@...	Mountain...	1
2	SO43704	1	2019-07-01	Julio Ruiz	julio1@adv...	Mountain...	1
3	SO43705	1	2019-07-01	Curtis Lu	curtis9@ad...	Mountain...	1
4	SO43700	1	2019-07-01	Ruben Prasad	ruben10@...	Road-650 ...	1

Notebooks consist of one or more *cells*, each of which can contain markdown-formatted content or executable code. You can run the code interactively in the notebook and see the results immediately.

Spark job definition

If you want to use Spark to ingest and transform data as part of an automated process, you can define a Spark job to run a script on-demand or based on a schedule.



To configure a Spark job, create a Spark Job Definition in your workspace and specify the script it should run. You can also specify a reference file (for example, a Python code file containing definitions of functions that are used in your script) and a reference to a specific lakehouse containing data that the script processes.

4. Work with data in a Spark dataframe

<https://learn.microsoft.com/en-us/training/modules/use-apache-spark-work-files-lakehouse/4-dataframe>

Work with data in a Spark dataframe

Completed

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 **Files/data** folder in your lakehouse:

```
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
...
```

Inferring a schema

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

```
%%pyspark
df = spark.read.load('Files/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. You can select the language you want to use as a default in the toolbar of the Notebook interface, and then use a magic to override that choice for a specific cell. For example, here's the equivalent Scala code for the products data example:

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

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

Both of these code samples 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 an explicit 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:

```
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:

```
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('Files/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
...	...	...	...

Tip

Specifying an explicit schema also improves performance!

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 **`ProductID`** and **`ListPrice`** columns from the **`df`** dataframe containing product data in the previous example:

```
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`**:

```
bikes_df = df.select("ProductName", "Category", "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	Category	ListPrice
Mountain-100 Silver, 38	Mountain Bikes	3399.9900
Road-750 Black, 52	Road Bikes	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:

```
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
...	...

Saving a dataframe

You'll often want to use Spark to transform raw data and save the results for further analysis or downstream processing. The following code example saves the DataFrame into a *parquet* file in the data lake, replacing any existing file of the same name.

```
bikes_df.write.mode("overwrite").parquet('Files/product_data/bikes.parquet')
```

Note

The Parquet format is typically preferred for data files that you will use for further analysis or ingestion into an analytical store. Parquet is a very efficient format that is supported by most large scale data analytics systems. In fact, sometimes your data transformation requirement may simply be to convert data from another format (such as CSV) to Parquet!

Partitioning the output file

Partitioning is an optimization technique that enables Spark to maximize performance across the worker nodes. More performance gains can be achieved when filtering data in queries by eliminating unnecessary disk IO.

To save a dataframe as a partitioned set of files, use the **partitionBy** method when writing the data. The following example saves the **bikes_df** dataframe (which contains the product data for the *mountain bikes* and *road bikes* categories), and partitions the data by category:

```
bikes_df.write.partitionBy("Category").mode("overwrite").parquet("Files/bike_data")
```

The folder names generated when partitioning a dataframe include the partitioning column name and value in a **column=value** format, so the code example creates a folder named **bike_data** that contains the following subfolders:

- **Category=Mountain Bikes**
- **Category=Road Bikes**

Each subfolder contains one or more parquet files with the product data for the appropriate category.

Note

You can partition the data by multiple columns, which results in a hierarchy of folders for each partitioning key. For example, you might partition sales order data by year and month, so that the folder hierarchy includes a folder for each year value, which in turn contains a subfolder for each month value.

Load partitioned data

When reading partitioned data into a dataframe, you can load data from any folder within the hierarchy by specifying explicit values or wildcards for the partitioned fields. The following example loads data for products in the **Road Bikes** category:

```
road_bikes_df = spark.read.parquet('Files/bike_data/Category=Road Bikes')  
display(road_bikes_df.limit(5))
```

Note

The partitioning columns specified in the file path are omitted in the resulting dataframe. The results produced by the example query would not include a **Category** column - the category for all rows would be *Road Bikes*.

5. Work with data using Spark SQL

Work with data using Spark SQL

Completed

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:

```
df.createOrReplaceTempView("products_view")
```

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.

Tables are metadata structures that store their underlying data in the storage location associated with the catalog. In Microsoft Fabric, data for *managed* tables is stored in the **Tables** storage location shown in your data lake, and any tables created using Spark are listed there.

You can create an empty table by using the `spark.catalog.createTable` method, or you can save a dataframe as a table by using its `saveAsTable` method. Deleting a managed table also deletes its underlying data.

For example, the following code saves a dataframe as a new table named **products**:

```
df.write.format("delta").saveAsTable("products")
```

Note

The Spark catalog supports tables based on files in various formats. The preferred format in Microsoft Fabric is **delta**, which is the format for a relational data technology on Spark named *Delta Lake*. Delta tables support features commonly found in relational database systems, including transactions, versioning, and support for streaming data.

Additionally, you can create *external* tables 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 the **Files** storage area of a lakehouse. Deleting an external table doesn't delete the underlying data.

Tip

You can apply the same partitioning technique to delta lake tables as discussed for parquet files in the previous unit. Partitioning tables can result in better performance when querying them.

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** table as a dataframe.

```
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:

ProductID	ProductName	ListPrice
771	Mountain-100 Silver, 38	3399.9900
839	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

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:

Category	ProductCount
Bib-Shorts	3
Bike Racks	1
Bike Stands	1
...	...

6. Visualize data in a Spark notebook

<https://learn.microsoft.com/en-us/training/modules/use-apache-spark-work-files-lakehouse/6-visualize-data>

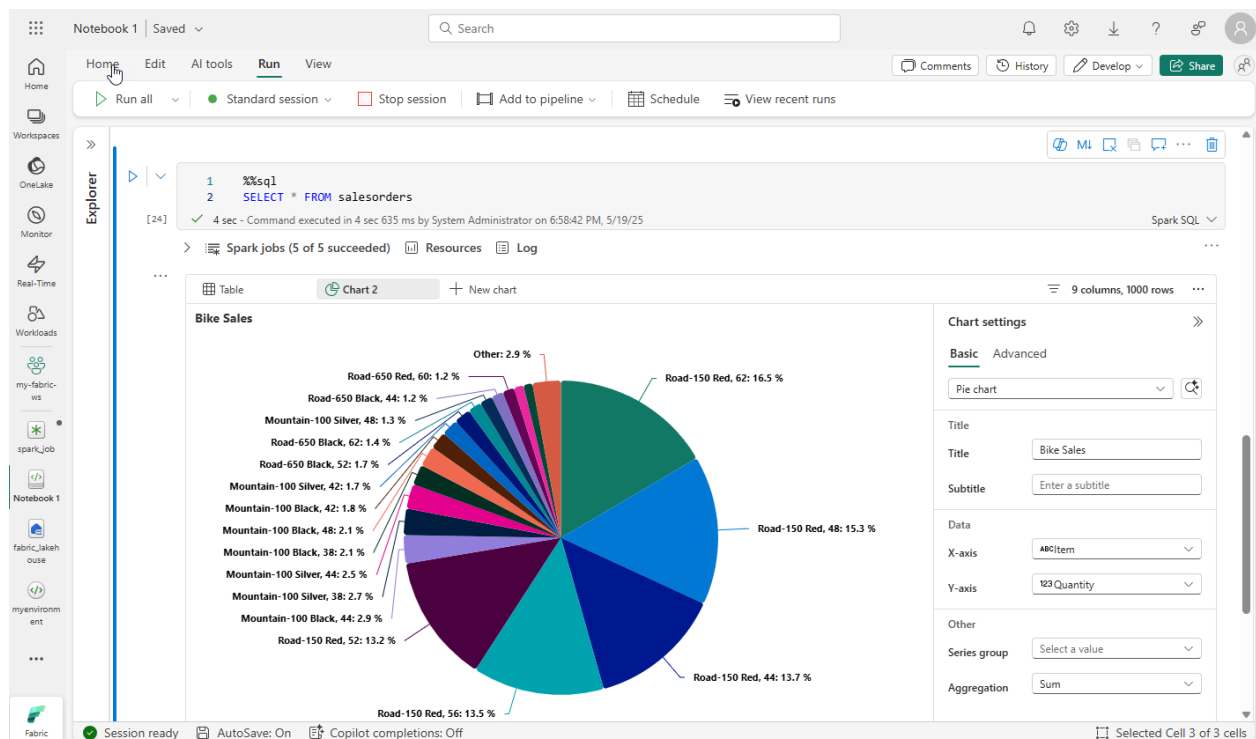
Visualize data in a Spark notebook

Completed

One of the most intuitive ways to analyze the results of data queries is to visualize them as charts. Notebooks in Microsoft Fabric provide some basic 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, the results are displayed under the code cell. By default, results are rendered as a table, but you can also change the results view to a chart and use the chart properties to customize how the chart visualizes the data, as shown here:



The built-in charting 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, 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.

```
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))

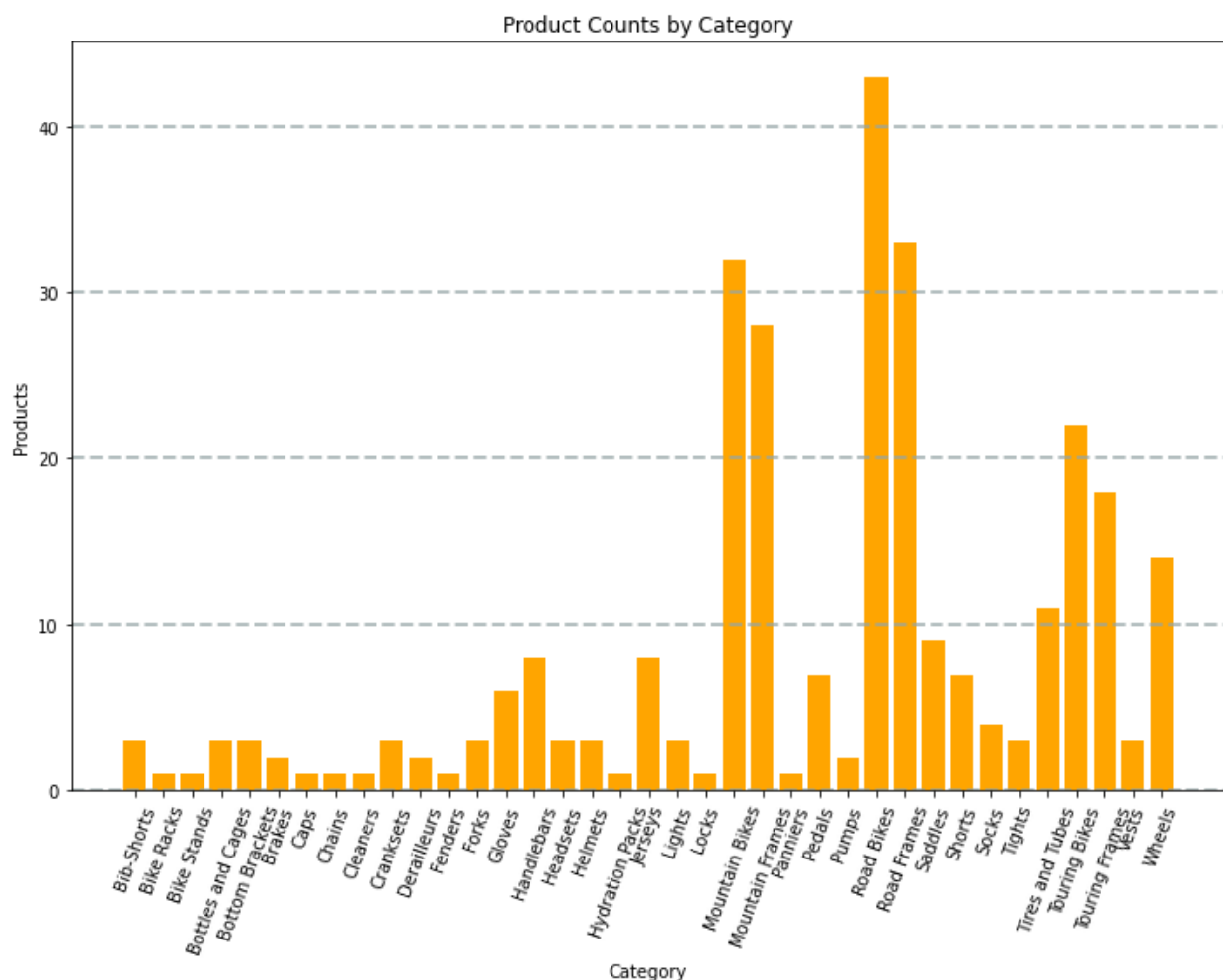
# Create a bar plot of product counts by category
plt.bar(x=data['Category'], height=data['ProductCount'], color='orange')

# 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.

7. Exercise - Analyze data with Apache Spark

<https://learn.microsoft.com/en-us/training/modules/use-apache-spark-work-files-lakehouse/7-exercise-spark>

Exercise - Analyze data with Apache Spark

Completed

Now it's your opportunity to work with Apache Spark in Microsoft Fabric. In this exercise, you'll use a Spark notebook to analyze and visualize data from files in lakehouse.

Note

You need a Microsoft Fabric trial license with the Fabric preview enabled in your tenant. See [Getting started with Fabric](#) to enable your Fabric trial license.

Launch the exercise and follow the instructions.

[Launch Exercise](#)

8. Module assessment

<https://learn.microsoft.com/en-us/training/modules/use-apache-spark-work-files-lakehouse/8-knowledge-check>

Module assessment

Completed

9. Summary

Summary

Completed

Apache Spark is a key technology used in big data analytics. Spark support in Microsoft Fabric enables you to integrate big data processing in Spark with the other data analytics and visualization capabilities of the platform.

Tip

For more information about working with data in Spark, see the [Spark SQL, DataFrames and Datasets Guide](#) in the Apache Spark documentation.