100 XP

Introduction

1 minute

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

This module explores how you can use Spark in Azure Synapse Analytics to ingest, process, and analyze data from a data lake. 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 Synapse analytical runtimes are specific to Azure Synapse Analytics.

After completing this module, you'll be able to:

- Identify core features and capabilities of Apache Spark.
- Configure a Spark pool in Azure Synapse Analytics.
- Run code to load, analyze, and visualize data in a Spark notebook.

Next unit: Get to know Apache Spark

Continue

100 XP

Get to know Apache Spark

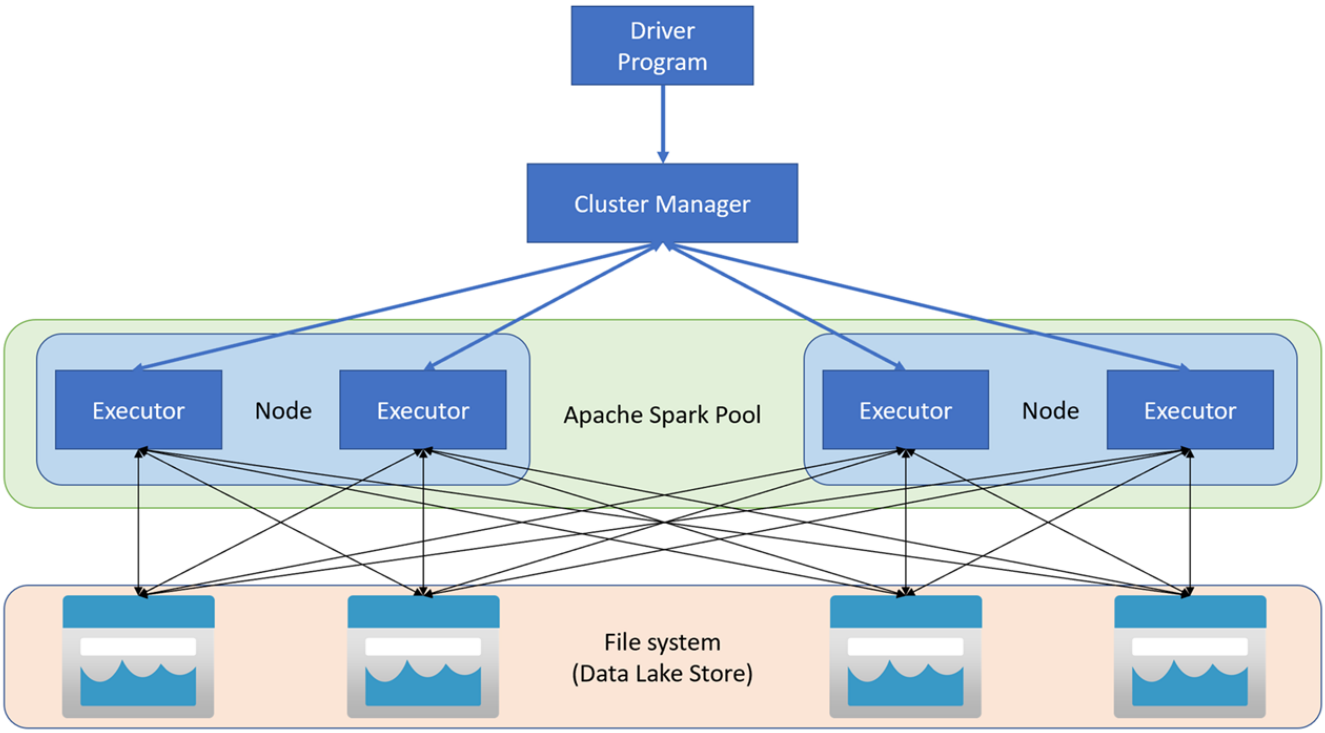
3 minutes

Apache Spark is distributed data processing framework that enables large-scale data analytics by coordinating work across multiple processing nodes in a cluster.

How Spark works

Apache Spark applications run as independent sets of processes on a cluster, coordinated by the *SparkContext* object in your main program (called the driver program). The SparkContext connects to the cluster manager, which allocates resources across applications using an implementation of Apache Hadoop YARN. Once connected, Spark acquires executors on nodes in the cluster to run your application code.

The SparkContext runs the main function and parallel operations on the cluster nodes, and then collects the results of the operations. The nodes read and write data from and to the file system and cache transformed data in-memory as *Resilient Distributed Datasets* (RDDs).



The SparkContext is responsible for converting an application to a *directed acyclic graph* (DAG). The graph consists of individual tasks that get executed within an executor process on the nodes. Each application gets its own executor processes, which stay up for the duration of the whole application and run tasks in multiple threads.

Spark pools in Azure Synapse Analytics

In Azure Synapse Analytics, a cluster is implemented as a *Spark pool*, which provides a runtime for Spark operations. You can create one or more Spark pools in an Azure Synapse Analytics workspace by using the Azure portal, or in Azure Synapse Studio. When defining a Spark pool, you can specify configuration options for the pool, including:

- A name for the spark pool.
- The size of virtual machine (VM) used for the nodes in the pool, including the option to use hardware accelerated GPU-enabled nodes.
- The number of nodes in the pool, and whether the pool size is fixed or individual nodes can be brought online dynamically to *auto-scale* the cluster; in which case, you can specify the minimum and maximum number of active nodes.
- The version of the *Spark Runtime* to be used in the pool; which dictates the versions of individual components such as Python, Java, and others that get installed.

> **Tip**
>
> For more information about Spark pool configuration options, see **Apache Spark pool configurations in Azure Synapse Analytics** in the Azure Synapse Analytics documentation.

Spark pools in an Azure Synapse Analytics Workspace are *serverless* - they start on-demand and stop when idle.

Next unit: Use Spark in Azure Synapse Analytics

Continue >

---

100 XP

Use Spark in Azure Synapse Analytics

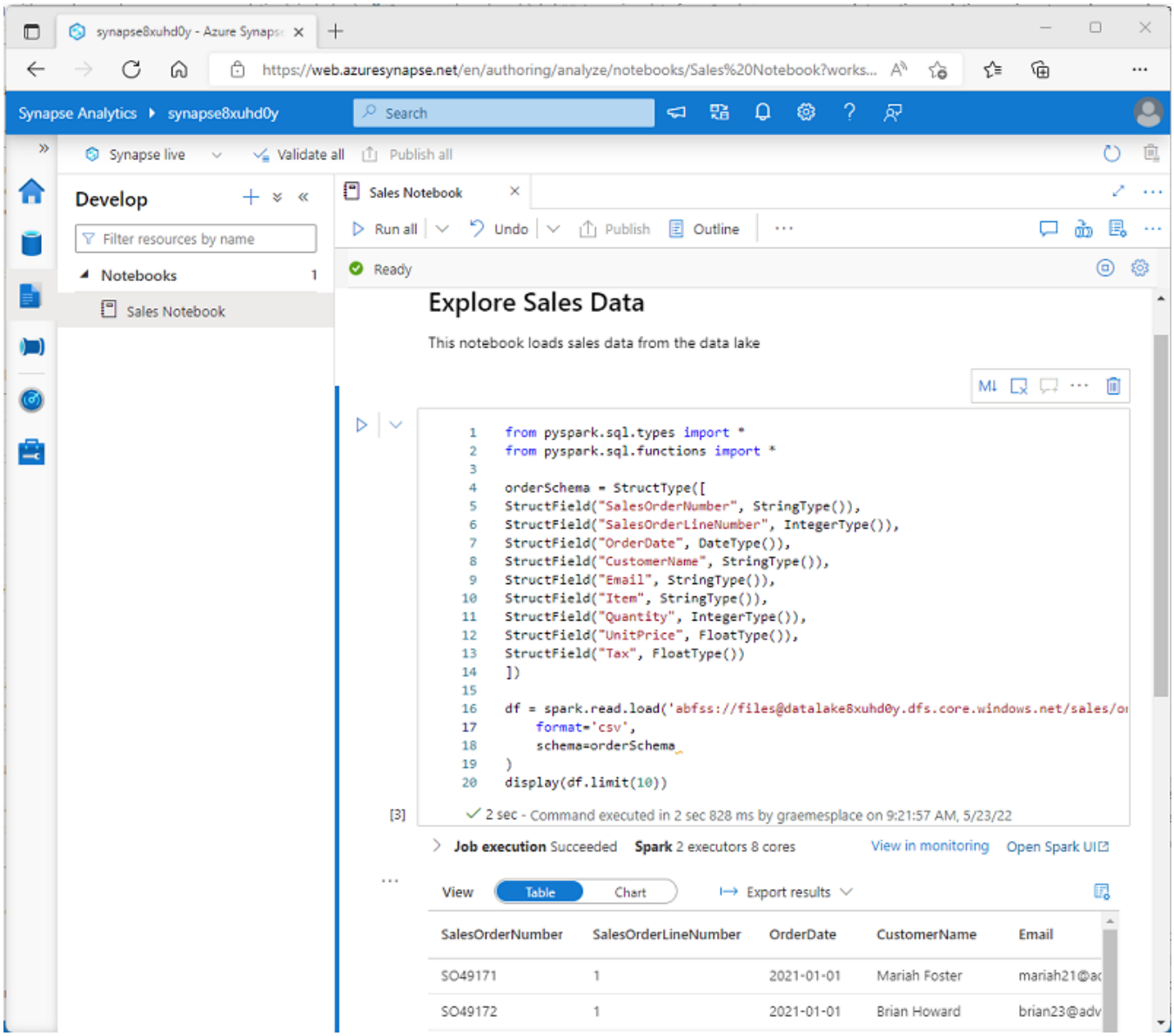
3 minutes

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 Synapse Studio 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 Synapse Studio is similar to that of Jupyter notebooks - a popular open source notebook platform.



> **Note**
>
> While usually used interactively, notebooks can be included in automated pipelines and run as an unattended script.

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 Synapse Analytics, see the **Create, develop, and maintain Synapse notebooks in Azure Synapse Analytics** article in the Azure Synapse Analytics documentation.

Accessing data from a Synapse Spark pool

You can use Spark in Azure Synapse Analytics to work with data from various sources, including:

- A data lake based on the primary storage account for the Azure Synapse Analytics workspace.
- A data lake based on storage defined as a *linked service* in the workspace.
- A dedicated or serverless SQL pool in the workspace.
- An Azure SQL or SQL Server database (using the Spark connector for SQL Server)
- An Azure Cosmos DB analytical database defined as a *linked service* and configured using *Azure Synapse Link for Cosmos DB*.
- An Azure Data Explorer Kusto database defined as a *linked service* in the workspace.
- An external Hive metastore defined as a *linked service* in the workspace.

One of the most common uses of Spark is to work with data in a data lake, where you can read and write files in multiple commonly used formats, including delimited text, Parquet, Avro, and others.

Next unit: Analyze data with Spark

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Analyze data with Spark

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 Synapse Analytics 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* and *Microsoft .NET*.

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 primary storage account for an Azure Synapse Analytics workspace:

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('abfss://container@store.dfs.core.windows.net/product-s.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:

Scala

```
%%spark
val df = spark.read.format("csv").option("header",
"true").load("abfss://container@store.dfs.core.windows.net/products.csv")
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 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('abfss://container@store.dfs.core.windows.net/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:

ProductID	ProductName	ListPrice
38	Mountain-100 Silver, 38	3399.9900
52	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 with Spark

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

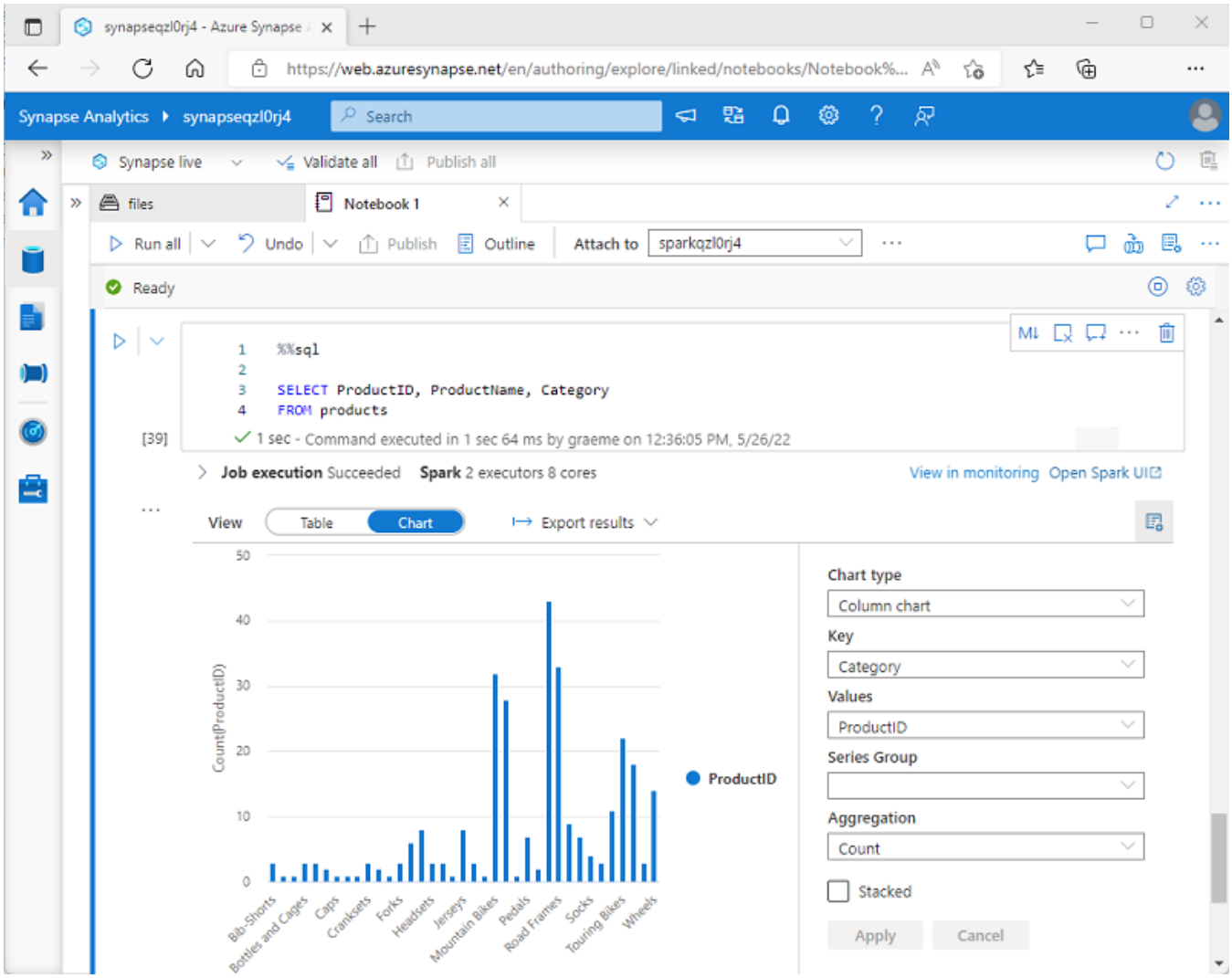
Visualize data with Spark

5 minutes

One of the most intuitive ways to analyze the results of data queries is to visualize them as charts. Notebooks in Azure Synapse Analytics 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 in Azure Synapse Analytics, 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're working with results of a query that don't include any existing groupings or aggregations, and 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.

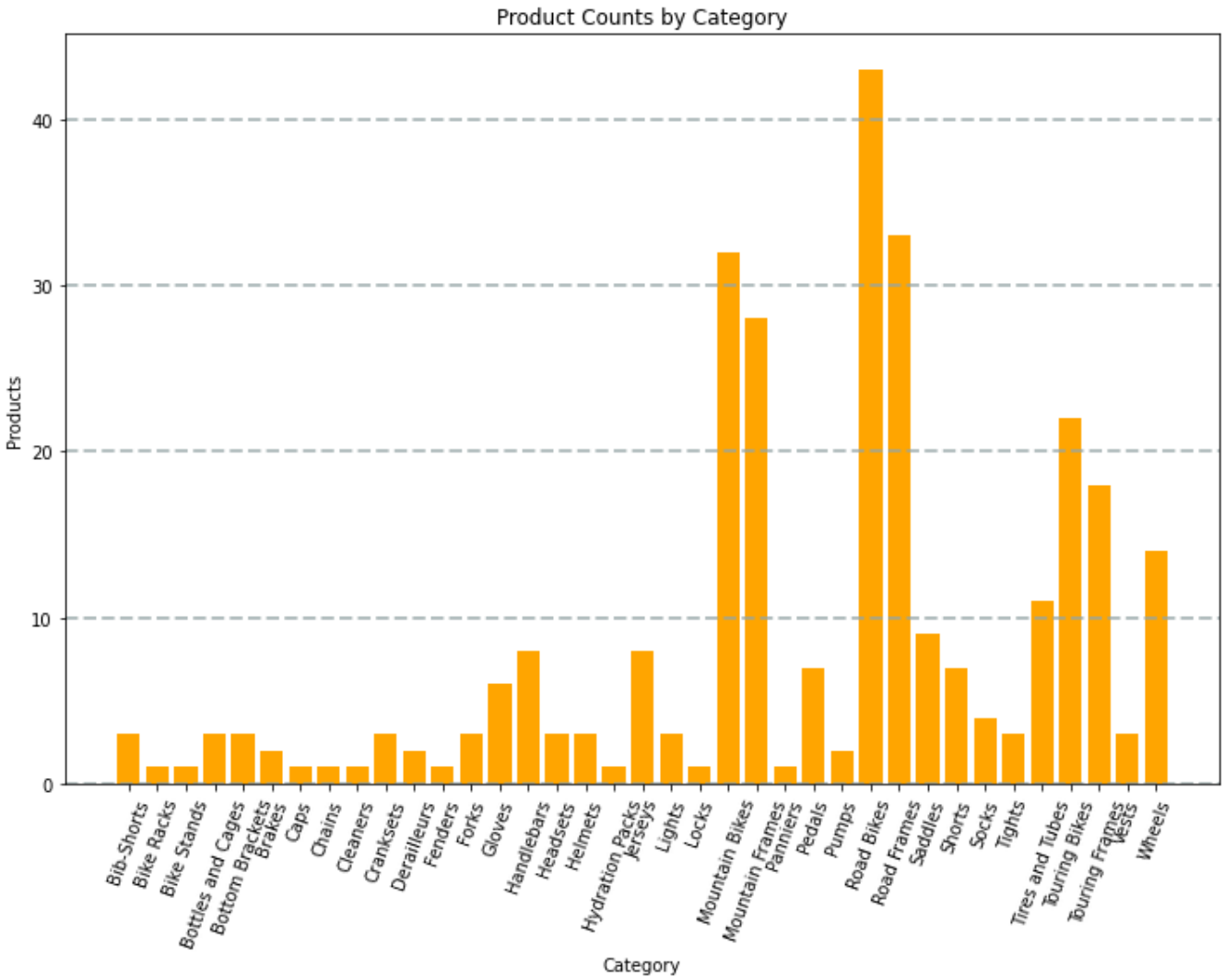
For example, you could use the following PySpark code to aggregate data from the hypothetical products data explored previously in this module, and use Matplotlib to create a chart from the aggregated data.

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.

Next unit: Exercise - Analyze data with Spark

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

3 minutes

Check your knowledge

1. Which definition best describes Apache Spark? *

- ○ A highly scalable relational database management system.
- ○ 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.
- ○ 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

✘ **Incorrect. The %%pyspark magic instructs Spark to interpret the code in the cell as Python.**

- ○ %%sql

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

Next unit: Summary

100 XP

Summary

1 minute

Apache Spark is a key technology used in big data analytics, and the Spark pool support in Azure Synapse Analytics enables you to combine big data processing in Spark with large-scale data warehousing in SQL.

In this module, you learned how to:

- Identify core features and capabilities of Apache Spark.
- Configure a Spark pool in Azure Synapse Analytics.
- Run code to load, analyze, and visualize data in a Spark notebook.

Module complete:

Unlock achievement

100 XP

Introduction

1 minute

Apache Spark provides a powerful platform for performing data cleansing and transformation tasks on large volumes of data. By using the Spark *dataframe* object, you can easily load data from files in a data lake and perform complex modifications. You can then save the transformed data back to the data lake for downstream processing or ingestion into a data warehouse.

Azure Synapse Analytics provides Apache Spark pools that you can use to run Spark workloads to transform data as part of a data ingestion and preparation workload. You can use natively supported notebooks to write and run code on a Spark pool to prepare data for analysis. You can then use other Azure Synapse Analytics capabilities such as SQL pools to work with the transformed data.

Next unit: Modify and save dataframes

Continue

100 XP

Modify and save dataframes

5 minutes

Apache Spark provides the *dataframe* object as the primary structure for working with data. You can use dataframes to query and transform data, and persist the results in a data lake. To load data into a dataframe, you use the **spark.read** function, specifying the file format, path, and optionally the schema of the data to be read. For example, the following code loads data from all .csv files in the **orders** folder into a dataframe named **order_details** and then displays the first five records.

Python

```
order_details = spark.read.csv('/orders/*.csv', header=True,
inferSchema=True)
display(order_details.limit(5))
```

Transform the data structure

After loading the source data into a dataframe, you can use the dataframe object's methods and Spark functions to transform it. Typical operations on a dataframe include:

- Filtering rows and columns
- Renaming columns
- Creating new columns, often derived from existing ones
- Replacing null or other values

In the following example, the code uses the `split` function to separate the values in the **CustomerName** column into two new columns named **FirstName** and **LastName**. Then it uses the `drop` method to delete the original **CustomerName** column.

Python

```
from pyspark.sql.functions import split, col

# Create the new FirstName and LastName fields
transformed_df = order_details.withColumn("FirstName", split(col("Customer-
Name"), " ").getItem(0)).withColumn("LastName", split(col("CustomerName"),
" ").getItem(1))

# Remove the CustomerName field
transformed_df = transformed_df.drop("CustomerName")
```

```
display(transformed_df.limit(5))
```

You can use the full power of the Spark SQL library to transform the data by filtering rows, deriving, removing, renaming columns, and any applying other required data modifications.

Save the transformed data

After your dataFrame is in the required structure, you can save the results to a supported format in your data lake.

The following code example saves the dataFrame into a *parquet* file in the data lake, replacing any existing file of the same name.

Python

```
transformed_df.write.mode("overwrite").parquet('/transformed_data/orders.-
parquet')
print ("Transformed data saved!")
```

> **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!

Next unit: Partition data files

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Partition data files

5 minutes

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.

Partition the output file

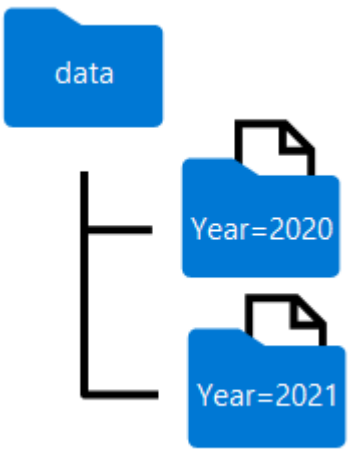
To save a dataframe as a partitioned set of files, use the **partitionBy** method when writing the data.

The following example creates a derived **Year** field. Then uses it to partition the data.

Python

```
from pyspark.sql.functions import year, col

# Load source data
df = spark.read.csv('/orders/*.csv', header=True, inferSchema=True)

# Add Year column
dated_df = df.withColumn("Year", year(col("OrderDate")))

# Partition by year
dated_df.write.partitionBy("Year").mode("overwrite").parquet("/data")
```

The folder names generated when partitioning a dataframe include the partitioning column name and value in a ***column=value*** format, as shown here:



> **Note**
>
> You can partition the data by multiple columns, which results in a hierarchy of folders for each partitioning key. For example, you could partition the order in the example 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.

Filter parquet files in a query

When reading data from parquet files into a dataframe, you have the ability to pull data from any folder within the hierarchical folders. This filtering process is done with the use of explicit values and wildcards against the partitioned fields.

In the following example, the following code will pull the sales orders, which were placed in 2020.

Python

```
orders_2020 = spark.read.parquet('/partitioned_data/Year=2020')
display(orders_2020.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 **Year** column - all rows would be from 2020.

Next unit: Transform data with SQL

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Transform data with SQL

5 minutes

The SparkSQL library, which provides the dataframe structure also enables you to use SQL as a way of working with data. With this approach, You can query and transform data in dataframes by using SQL queries, and persist the results as tables.

> **Note**
>
> Tables are metadata abstractions over files. The data is not stored in a relational table, but the table provides a relational layer over files in the data lake.

Define tables and views

Table definitions in Spark are stored in the *metastore*, a metadata layer that encapsulates relational abstractions over files. *External* tables are relational tables in the metastore that reference files in a data lake location that you specify. You can access this data by querying the table or by reading the files directly from the data lake.

> **Note**
>
> External tables are "loosely bound" to the underlying files and deleting the table *does not* delete the files. This allows you to use Spark to do the heavy lifting of transformation then persist the data in the lake. After this is done you can drop the table and downstream processes can access these optimized structures. You can also define *managed* tables, for which the underlying data files are stored in an internally managed storage location associated with the metastore. Managed tables are "tightly-bound" to the files, and dropping a managed table deletes the associated files.

The following code example saves a dataframe (loaded from CSV files) as an external table name **sales_orders**. The files are stored in the **/sales_orders_table** folder in the data lake.

Python

```
order_details.write.saveAsTable('sales_orders', format='parquet',
mode='overwrite', path='/sales_orders_table')
```

Use SQL to query and transform the data

After defining a table, you can use of SQL to query and transform its data. The following code creates two new derived columns named **Year** and **Month** and then creates a new table *transformed_orders* with the new derived columns added.

Python

```
# Create derived columns
sql_transform = spark.sql("SELECT *, YEAR(OrderDate) AS Year,
MONTH(OrderDate) AS Month FROM sales_orders")

# Save the results
sql_transform.write.partitionBy("Year","Month").saveAsTable('trans-
formed_orders', format='parquet', mode='overwrite', path='/trans-
formed_orders_table')
```

The data files for the new table are stored in a hierarchy of folders with the format of **Year=*NNNN* / Month=*N***, with each folder containing a parquet file for the corresponding orders by year and month.

Query the metastore

Because this new table was created in the metastore, you can use SQL to query it directly with the *%%sql* magic key in the first line to indicate that the SQL syntax will be used as shown in the following script:

SQL

```
%%sql

SELECT * FROM transformed_orders
WHERE Year = 2021
    AND Month = 1
```

Drop tables

When working with external tables, you can use the `DROP` command to delete the table definitions from the metastore without affecting the files in the data lake. This approach enables you to clean up the metastore after using SQL to transform the data, while making the transformed data files available to downstream data analysis and ingestion processes.

SQL

```
%%sql

DROP TABLE transformed_orders;
DROP TABLE sales_orders;
```

Next unit: Exercise: Transform data with Spark in Azure Synapse Analytics

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Exercise: Transform data with Spark in Azure Synapse Analytics

30 minutes

Now it's your chance to use Spark to transform data for yourself. In this exercise, you'll use a Spark notebook in Azure Synapse Analytics to transform data in files.

> **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

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 a Synapse Analytics Spark pool.
- Create Spark catalog tables for Delta Lake data.
- Use Delta Lake tables for streaming data.
- Query Delta Lake tables from a Synapse Analytics SQL pool.

> **Note**
>
> The version of Delta Lake available in an Azure Synapse Analytics pool depends on the version of Spark specified in the pool configuration. The information in this module reflects Delta Lake version 1.0, which is installed with Spark 3.1.

Next unit: Understand Delta Lake

Continue

100 XP

Understand Delta Lake

5 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 a Synapse Analytics Spark pool 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. Additionally, you can use the serverless SQL pool in Azure Synapse Analytics to query Delta Lake tables in SQL.

> **Tip**
>
> For more information about Delta Lake in Azure Synapse Analytics, see **What is Delta Lake** in the Azure Synapse Analytics documentation.

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 catalog tables

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Create catalog tables

6 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 pool, 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 with streaming data

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Use Delta Lake with streaming data

6 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.writeStream \
    .outputMode("append") \
    .format("console") \
    .start()
```

> **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
inputPath = '/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: Use Delta Lake in a SQL pool

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

100 XP

Use Delta Lake in a SQL pool

5 minutes

Delta Lake is designed as a transactional, relational storage layer for Apache Spark; including Spark pools in Azure Synapse Analytics. However, Azure Synapse Analytics also includes a serverless SQL pool runtime that enables data analysts and engineers to run SQL queries against data in a data lake or a relational database.

> **Note**
>
> You can only *query* data from Delta Lake tables in a serverless SQL pool; you can't *update*, *insert*, or *delete* data.

Querying delta formatted files with OPENROWSET

The serverless SQL pool in Azure Synapse Analytics includes support for reading delta format files; enabling you to use the SQL pool to query Delta Lake tables. This approach can be useful in scenarios where you want to use Spark and Delta tables to process large quantities of data, but use the SQL pool to run queries for reporting and analysis of the processed data.

In the following example, a SQL `SELECT` query reads delta format data using the `OPENROWSET` function.

SQL

```
SELECT *
FROM
    OPENROWSET(
        BULK 'https://mystore.dfs.core.windows.net/files/delta/mytable/',
        FORMAT = 'DELTA'
    ) AS deltadata
```

You could run this query in a serverless SQL pool to retrieve the latest data from the Delta Lake table stored in the specified file location.

You could also create a database and add a data source that encapsulates the location of your Delta Lake data files, as shown in this example:

SQL

```
CREATE DATABASE MyDB
      COLLATE Latin1_General_100_BIN2_UTF8;
GO;

USE MyDB;
GO

CREATE EXTERNAL DATA SOURCE DeltaLakeStore
WITH
(
    LOCATION = 'https://mystore.dfs.core.windows.net/files/delta/'
);
GO

SELECT TOP 10 *
FROM OPENROWSET(
        BULK 'mytable',
        DATA_SOURCE = 'DeltaLakeStore',
        FORMAT = 'DELTA'
    ) as deltadata;
```

> **Note**
>
> When working with Delta Lake data, which is stored in Parquet format, it's generally best to create a database with a UTF-8 based collation in order to ensure string compatibility.

Querying catalog tables

The serverless SQL pool in Azure Synapse Analytics has shared access to databases in the Spark metastore, so you can query catalog tables that were created using Spark SQL. In the following example, a SQL query in a serverless SQL pool queries a catalog table that contains Delta Lake data:

SQL

```
-- By default, Spark catalog tables are created in a database named "de-
fault"
-- If you created another database using Spark SQL, you can use it here
USE default;

SELECT * FROM MyDeltaTable;
```

> **Tip**

For more information about using Delta Tables from a serverless SQL pool, see **Query Delta Lake files using serverless SQL pool in Azure Synapse Analytics** in the Azure Synapse Analytics documentation.

Next unit: Exercise - Use Delta Lake in Azure Synapse Analytics

Continue >

How are we doing? ☆ ☆ ☆ ☆ ☆

200 XP

Knowledge check

5 minutes

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

- A Spark API for exporting 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 pools and Spark pools.

✕ Incorrect. Delta Lake does not replicate data between SQL pools and Spark pools.

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

✕ Incorrect. Delta Lake tables do support Spark Structured Streaming, but that isn't what enables querying of previous versions.

- 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

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.
- ○ 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? ☆ ☆ ☆ ☆ ☆