

Work with Delta Lake tables in Microsoft Fabric

1. Introduction

<https://learn.microsoft.com/en-us/training/modules/work-delta-lake-tables-fabric/1-introduction>

Introduction

Completed

Tables in a Microsoft Fabric lakehouse are based on the Linux foundation *Delta Lake* table format, commonly used in Apache Spark. 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 lakehouse architecture to support SQL-based data manipulation semantics in Spark 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.

While you don't need to work directly with Delta Lake APIs in order to use tables in a Fabric lakehouse, an understanding of the Delta Lake metastore architecture and familiarity with some of the more specialized Delta table operations can greatly expand your ability to build advanced analytics solutions on Microsoft Fabric.

2. Understand Delta Lake

<https://learn.microsoft.com/en-us/training/modules/work-delta-lake-tables-fabric/2-understand-delta-lake>

Understand Delta Lake

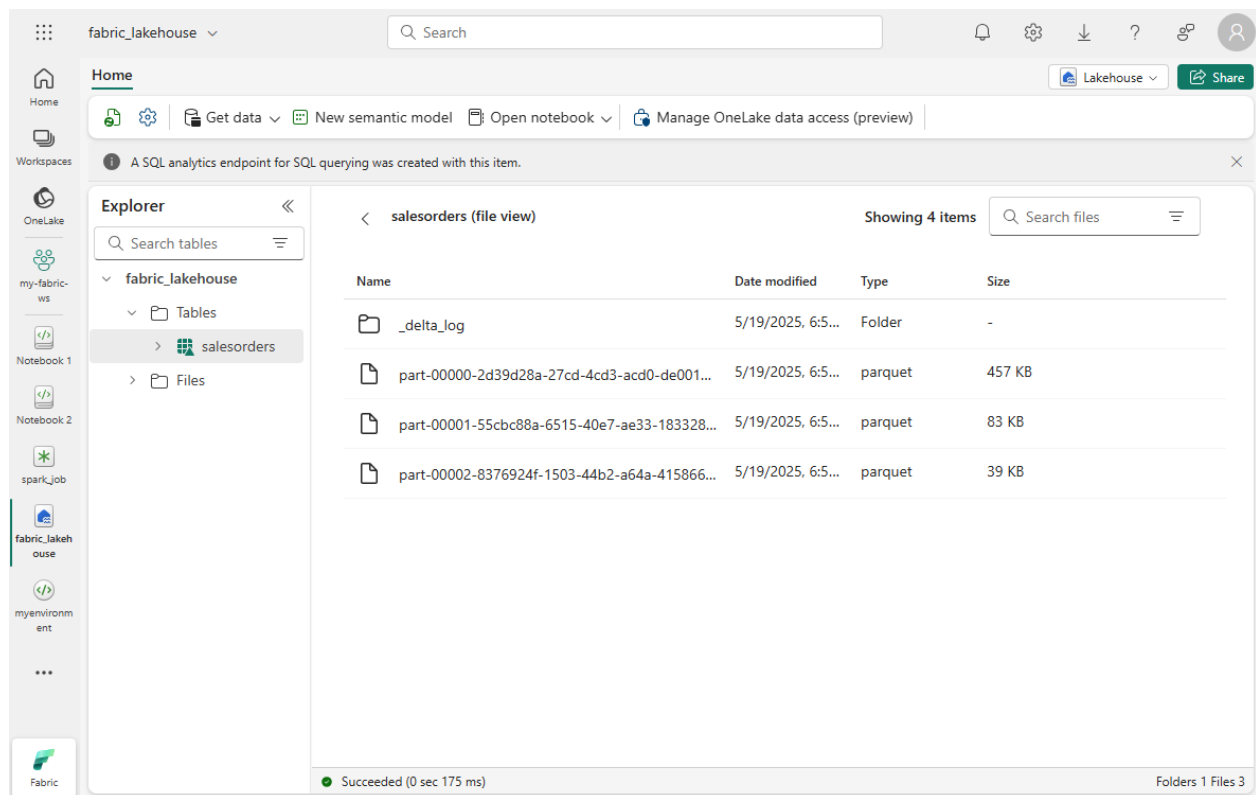
Completed

Delta Lake is an open-source storage layer that adds relational database semantics to Spark-based data lake processing. Tables in Microsoft Fabric lakehouses are Delta tables, which is signified by the triangular Delta (Δ) icon on tables in the lakehouse user interface.

The screenshot displays the Microsoft Fabric Lakehouse interface. On the left, the 'Explorer' pane shows the 'fabric_lakehouse' structure with 'Tables' and 'Files' folders. The 'salesorders' table is selected. The main area shows a table with 9 columns and 1,000 rows. The columns are: SalesOrderID, SalesOrderNumber, OrderDate, CustomerName, Email, Item, and Quantity. The table is a Delta table, indicated by a triangular icon in the first column header. A status bar at the bottom indicates 'Succeeded (3 sec 42 ms)' and 'Columns 9 Rows 1,000'.

	ABC SalesOrder...	123 SalesOrder...	OrderDate	ABC CustomerN...	ABC Email	ABC Item	123 Quantity
1	SO49171	1	7:00:00 PM	Mariah Foster	mariah21@adve...	Road-250 Black, ...	1
2	SO49172	1	7:00:00 PM	Brian Howard	brian23@advent...	Road-250 Red, 44	1
3	SO49173	1	7:00:00 PM	Linda Alvarez	linda19@advent...	Mountain-200 Si...	1
4	SO49174	1	7:00:00 PM	Gina Hernandez	gina4@adventur...	Mountain-200 Si...	1
5	SO49178	1	7:00:00 PM	Beth Ruiz	beth4@adventur...	Road-550-W Yell...	1
6	SO49179	1	7:00:00 PM	Evan Ward	evan13@advent...	Road-550-W Yell...	1
7	SO49175	1	7:00:00 PM	Margaret Guo	margaret24@ad...	Road-250 Red, 52	1
8	SO49180	1	7:00:00 PM	Mitchell Yuan	mittchell6@adve...	Road-650 Black, ...	1
9	SO49176	1	7:00:00 PM	Shawn Sharma	shawn11@adve...	Mountain-200 Si...	1
10	SO49177	1	7:00:00 PM	Barbara Chande	barbara44@adv...	Mountain-200 Si...	1
11	SO49186	1	7:00:00 PM	Cara Xu	cara8@adventur...	Road-250 Red, 52	1
12	SO49187	1	7:00:00 PM	Lacey Liu	lacey16@advent...	Road-250 Black, ...	1
13	SO49190	1	7:00:00 PM	Omar Zhu	omar13@advent...	Road-550-W Yell...	1
14	SO49185	1	7:00:00 PM	Cassandra Ferna...	cassandra17@a...	Mountain-200 BL...	1
15	SO49184	1	7:00:00 PM	Monica Martinez	monica17@adve...	Mountain-200 BL...	1
16	SO49189	1	7:00:00 PM	Marie Gonzalez	marie20@adven...	Road-650 Black, ...	1
17	SO49182	1	7:00:00 PM	Alexandra Hall	alexandra89@ad...	Road-250 Red, 48	1
18	SO49183	1	7:00:00 PM	Alejandro Raji	alejandro46@ad...	Road-250 Red, 52	1
19	SO49181	1	7:00:00 PM	Derrick Lim	derrick5@adrian...	Road-250 Black	1

Delta tables are schema abstractions over data files that are stored in Delta format. For each table, the lakehouse stores a folder containing *Parquet* data files and a **_delta_Log** folder in which transaction details are logged in JSON format.



The benefits of using Delta tables include:

- **Relational tables that support querying and data modification.** With Apache Spark, you can store data in Delta 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 tables is stored in Parquet format, which is commonly used in data lake ingestion pipelines. Additionally, you can use the SQL analytics endpoint for the Microsoft Fabric lakehouse to query Delta tables in SQL.

3. Create delta tables

<https://learn.microsoft.com/en-us/training/modules/work-delta-lake-tables-fabric/3-create-delta-tables>

Create delta tables

Completed

When you create a table in a Microsoft Fabric lakehouse, a delta table is defined in the metastore for the lakehouse and the data for the table is stored in the underlying Parquet files for the table.

With most interactive tools in the Microsoft Fabric environment, the details of mapping the table definition in the metastore to the underlying files are abstracted. However, when working with Apache Spark in a lakehouse, you have greater control of the creation and management of delta tables.

Creating a delta table from a dataframe

One of the easiest ways to create a delta table in Spark is to save a dataframe in the *delta* format. For example, the following PySpark code loads a dataframe with data from an existing file, and then saves that dataframe as a delta table:

```
# Load a file into a dataframe
df = spark.read.load('Files/mydata.csv', format='csv', header=True)

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

The code specifies that the table should be saved in delta format with a specified table name. The data for the table is saved in Parquet files (regardless of the format of the source file you loaded into the dataframe) in the **Tables** storage area in the lakehouse, along with a **_delta_log** folder containing the transaction logs for the table. The table is listed in the **Tables** folder for the lakehouse in the **Data explorer** pane.

Managed vs external tables

In the previous example, the dataframe was saved as a *managed* table; meaning that the table definition in the metastore and the underlying data files are both managed by the Spark runtime for the Fabric lakehouse. Deleting the table will also delete the underlying files from the **Tables** storage location for the lakehouse.

You can also create tables as *external* tables, in which the relational table definition in the metastore is mapped to an alternative file storage location. For example, the following code creates an external table for which the data is stored in the folder in the **Files** storage location for the lakehouse:

```
df.write.format("delta").saveAsTable("myexternaltable", path="Files/myexternaltable")
```

In this example, the table definition is created in the metastore (so the table is listed in the **Tables** user interface for the lakehouse), but the Parquet data files and JSON log files for the table are stored in the **Files** storage location (and will be shown in the **Files** node in the **Lakehouse explorer** pane).

You can also specify a fully qualified path for a storage location, like this:

```
df.write.format("delta").saveAsTable("myexternaltable", path="abfss://my_store_url")
```

Deleting an external table from the lakehouse metastore doesn't delete the associated data files.

Creating table metadata

While it's common to create a table from existing data in a dataframe, there are often scenarios where you want to create a table definition in the metastore that will be populated with data in other ways. There are multiple ways you can accomplish this goal.

Use the *DeltaTableBuilder* API

The **DeltaTableBuilder** API enables you to write Spark code to create a table based on your specifications. For example, the following code creates a table with a specified name and columns.

```
from delta.tables import *

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

Use Spark SQL

You can also create delta tables by using the Spark SQL `CREATE TABLE` statement, as shown in this example:

```
%%sql

CREATE TABLE salesorders
(
    Orderid INT NOT NULL,
```

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

The previous example creates a managed table. You can also create an external table by specifying a `LOCATION` parameter, as shown here:

```
%%sql  
  
CREATE TABLE MyExternalTable  
USING DELTA  
LOCATION 'Files/mydata'
```

When creating an external table, the schema of the table is determined by the Parquet files containing the data in the specified location. This approach can be useful when you want to create a table definition that references data that has already been saved in delta format, or based on a folder where you expect to ingest data in delta format.

Saving data in delta format

So far you've seen how to save a dataframe as a delta table (creating both the table schema definition in the metastore and the data files in delta format) and how to create the table definition (which creates the table schema in the metastore without saving any data files). A third possibility is to save data in delta format without creating a table definition in the metastore. This approach can be useful when you want to persist the results of data transformations performed in Spark in a file format over which you can later "overlay" a table definition or process directly by using the delta lake API.

For example, the following PySpark code saves a dataframe to a new folder location in *delta* format:

```
delta_path = "Files/mydatatable"  
df.write.format("delta").save(delta_path)
```

Delta files are saved in Parquet format in the specified path, and include a **_delta_log** folder containing transaction log files. Transaction logs record any changes in the data, such as updates made to external tables or through the delta lake API.

You can replace the contents of an existing folder with the data in a dataframe by using the **overwrite** mode, as shown here:

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

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

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

Tip

If you use the technique described here to save a dataframe to the **Tables** location in the lakehouse, Microsoft Fabric uses an automatic table discovery capability to create the corresponding table metadata in the metastore.

4. Optimize delta tables

<https://learn.microsoft.com/en-us/training/modules/work-delta-lake-tables-fabric/3b-optimize-delta-tables>

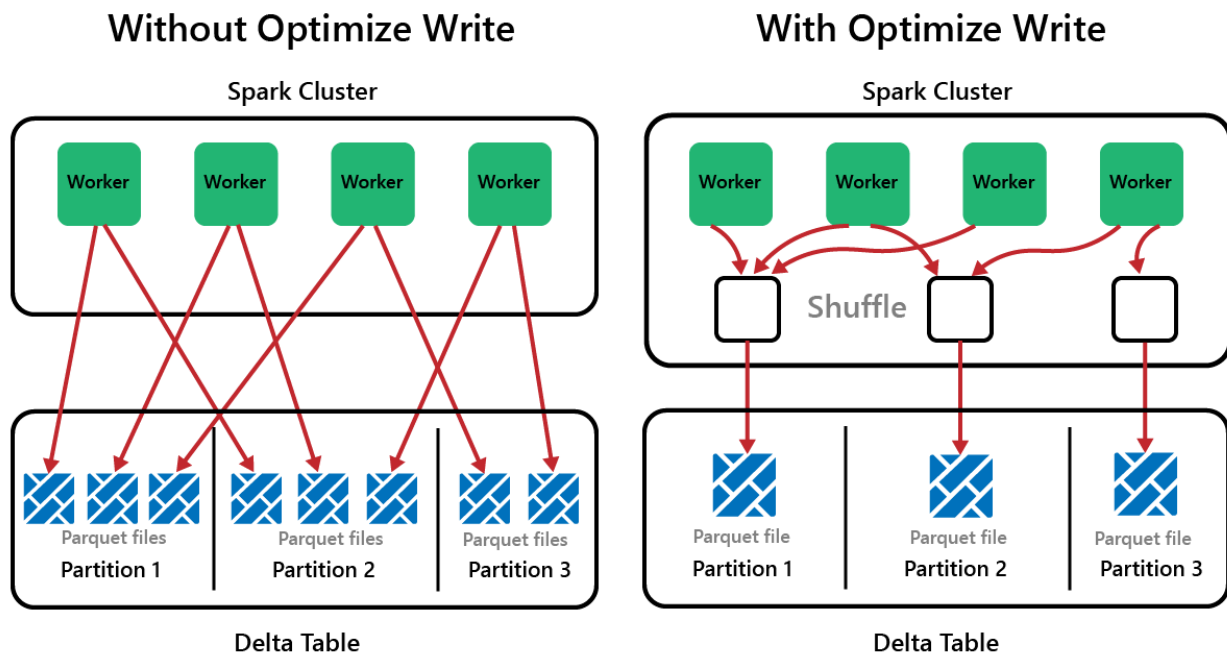
Optimize delta tables

Completed

Spark is a parallel-processing framework, with data stored on one or more worker nodes. In addition, Parquet files are immutable, with new files written for every update or delete. This process can result in Spark storing data in a large number of small files, known as the *small file problem*. It means that queries over large amounts of data can run slowly, or even fail to complete.

OptimizeWrite function

OptimizeWrite is a feature of Delta Lake which reduces the number of files as they're written. Instead of writing many small files, it writes fewer larger files. This helps to prevent the *small files problem* and ensure that performance isn't degraded.



In Microsoft Fabric, `OptimizeWrite` is enabled by default. You can enable or disable it at the Spark session level:

```
# Disable Optimize Write at the Spark session level
spark.conf.set("spark.microsoft.delta.optimizeWrite.enabled", False)

# Enable Optimize Write at the Spark session level
spark.conf.set("spark.microsoft.delta.optimizeWrite.enabled", True)

print(spark.conf.get("spark.microsoft.delta.optimizeWrite.enabled"))
```

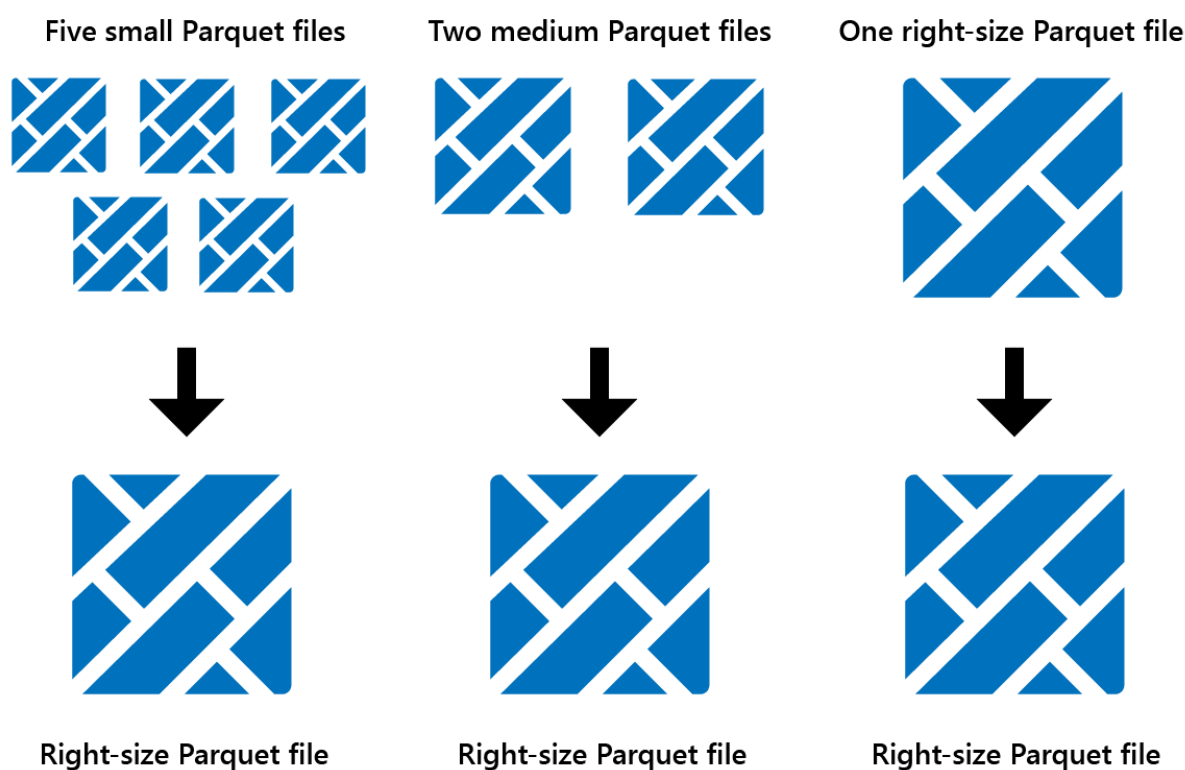
Note

`OptimizeWrite` can also be set in Table Properties and for individual write commands.

Optimize

Optimize is a table maintenance feature that consolidates small Parquet files into fewer large files. You might run Optimize after loading large tables, resulting in:

- fewer larger files
- better compression
- efficient data distribution across nodes



To run Optimize:

1. In **Lakehouse Explorer**, select the ... menu beside a table name and select **Maintenance**.
2. Select **Run OPTIMIZE command**.
3. Optionally, select **Apply V-order to maximize reading speeds in Fabric**.
4. Select **Run now**.

V-Order function

When you run Optimize, you can optionally run V-Order, which is designed for the Parquet file format in Fabric. V-Order enables lightning-fast reads, with in-memory-like data access times. It also improves cost efficiency as it reduces network, disk, and CPU resources during reads.

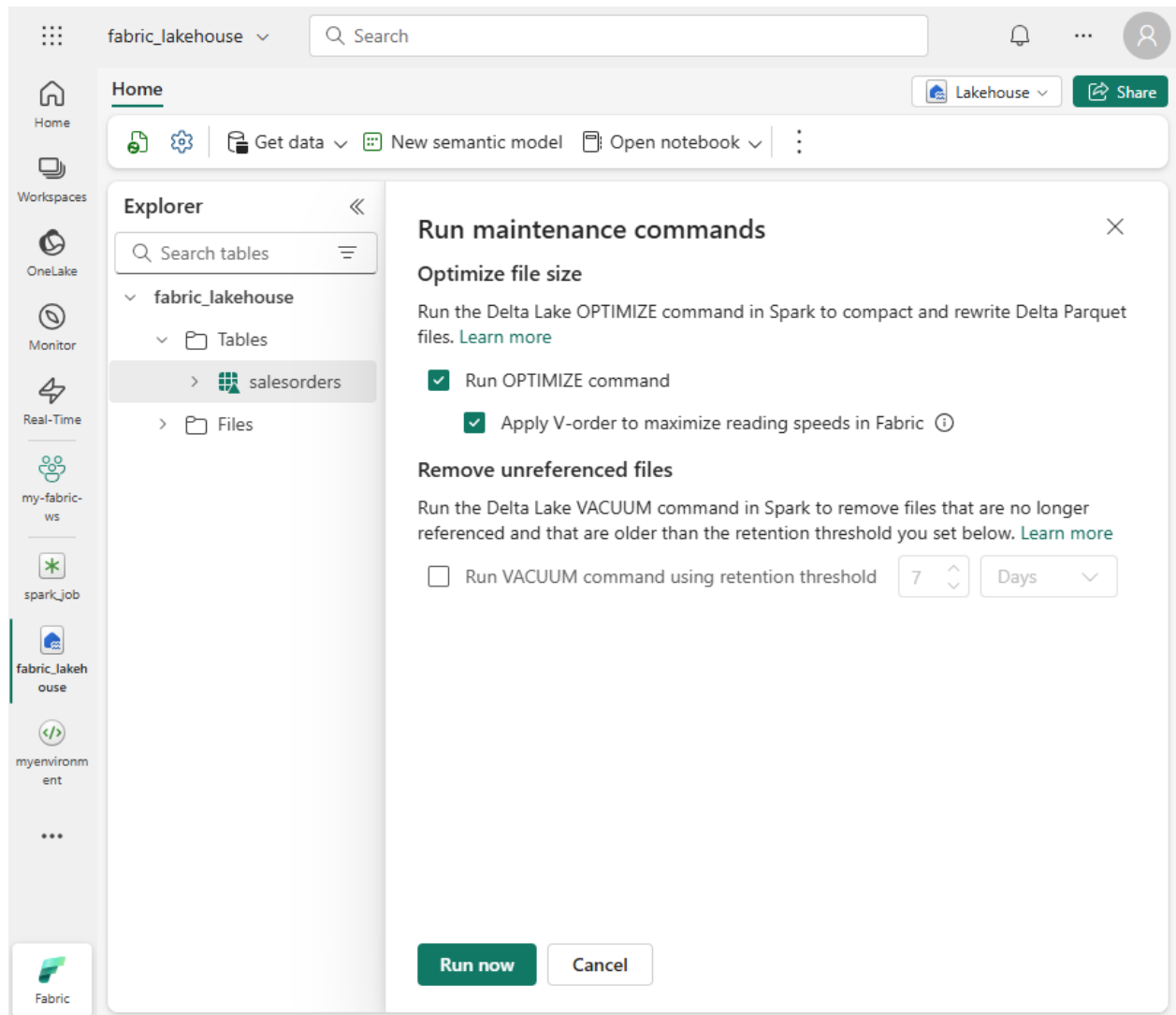
V-Order is enabled by default in Microsoft Fabric and is applied as data is being written. It incurs a small overhead of about 15% making writes a little slower. However, V-Order enables faster reads from the Microsoft Fabric compute engines, such as Power BI, SQL, Spark, and others.

In Microsoft Fabric, the Power BI and SQL engines use Microsoft Verti-Scan technology which takes full advantage of V-Order optimization to speed up reads. Spark and other engines don't use VertiScan technology but still benefit from V-Order optimization by about 10% faster reads, sometimes up to 50%.

V-Order works by applying special sorting, row group distribution, dictionary encoding, and compression on Parquet files. It's 100% compliant to the open-source Parquet format and all Parquet engines can read it.

V-Order might not be beneficial for write-intensive scenarios such as staging data stores where data is only read once or twice. In these situations, disabling V-Order might reduce the overall processing time for data ingestion.

Apply V-Order to individual tables by using the Table Maintenance feature by running the `OPTIMIZE` command.

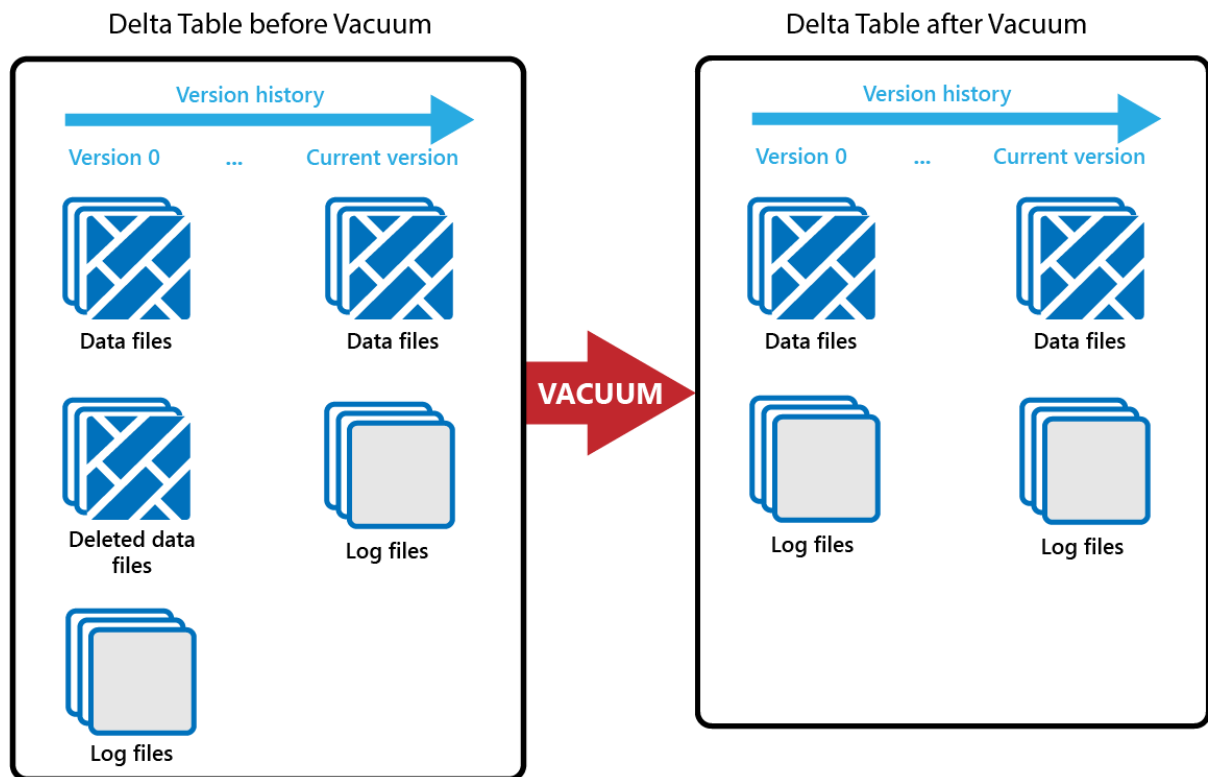


Vacuum

The VACUUM command enables you to remove old data files.

Every time an update or delete is done, a new Parquet file is created and an entry is made in the transaction log. Old Parquet files are retained to enable time travel, which means that Parquet files accumulate over time.

The VACUUM command removes old Parquet data files, but not the transaction logs. When you run VACUUM, you can't time travel back earlier than the retention period.



Data files that aren't currently referenced in a transaction log and that are older than the specified retention period are permanently deleted by running VACUUM. Choose your retention period based on factors such as:

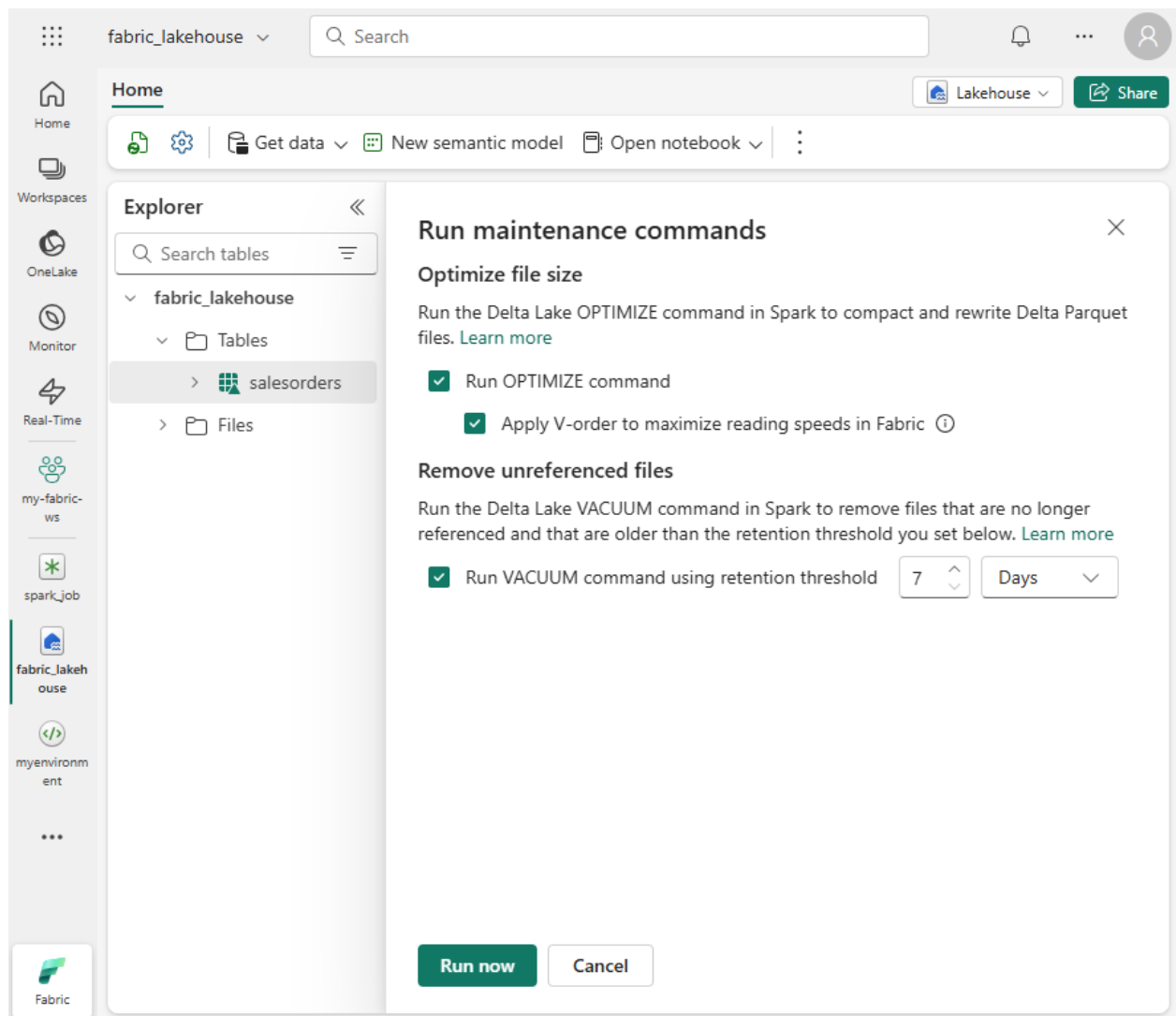
- Data retention requirements
- Data size and storage costs
- Data change frequency
- Regulatory requirements

The default retention period is 7 days (168 hours), and the system prevents you from using a shorter retention period.

You can run VACUUM on an ad-hoc basis or scheduled using Fabric notebooks.

Run VACUUM on individual tables by using the Table maintenance feature:

1. In **Lakehouse Explorer**, select the ... menu beside a table name and select **Maintenance**.
2. Select **Run VACUUM command using retention threshold** and set the retention threshold.
3. Select **Run now**.



You can also run **VACUUM** as a SQL command in a notebook:

```
%%sql
VACUUM lakehouse2.products RETAIN 168 HOURS;
```

VACUUM commits to the Delta transaction log, so you can view previous runs in DESCRIBE HISTORY.

```
%%sql
DESCRIBE HISTORY lakehouse2.products;
```

Partitioning Delta tables

Delta Lake allows you to organize data into partitions. This might improve performance by enabling *data skipping*, which boosts performance by skipping over irrelevant data objects based on an object's metadata.

Consider a situation where large amounts of sales data are being stored. You could partition sales data by year. The partitions are stored in subfolders named "year=2021", "year=2022", etc. If you

only want to report on sales data for 2024, then the partitions for other years can be skipped, which improves read performance.

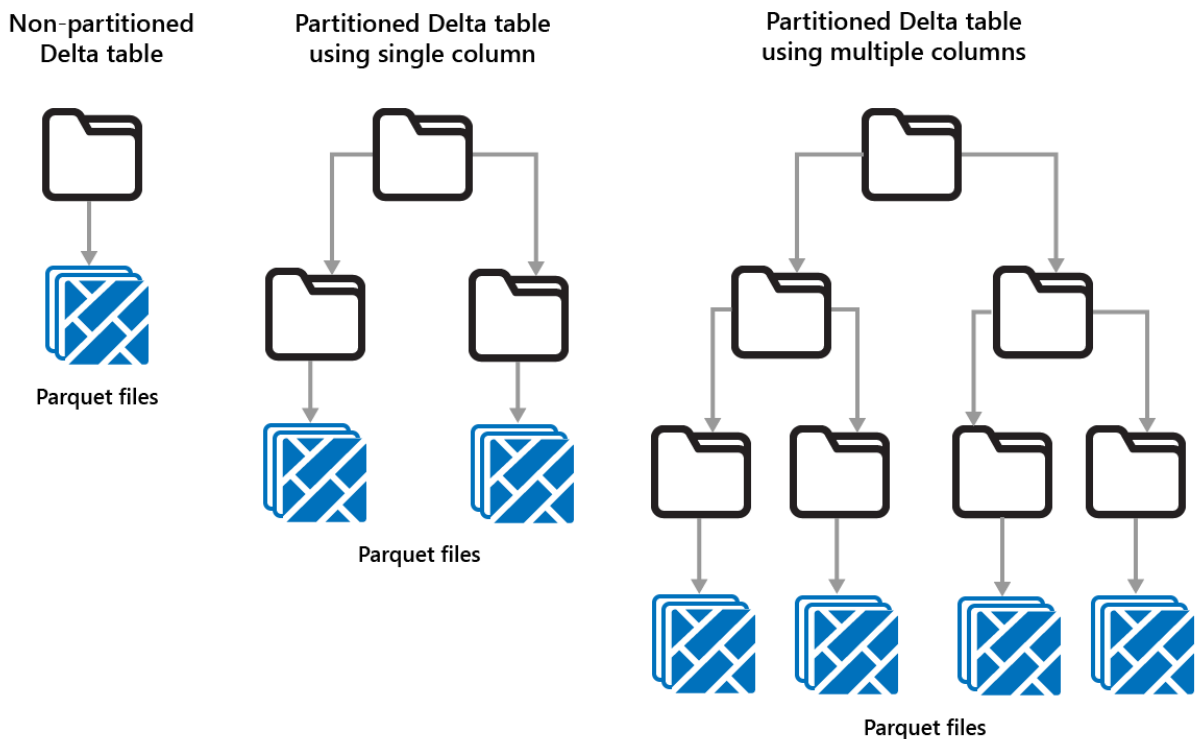
Partitioning of small amounts of data can degrade performance, however, because it increases the number of files and can exacerbate the "small files problem."

Use partitioning when:

- You have very large amounts of data.
- Tables can be split into a few large partitions.

Don't use partitioning when:

- Data volumes are small.
- A partitioning column has high cardinality, as this creates a large number of partitions.
- A partitioning column would result in multiple levels.



Partitions are a fixed data layout and don't adapt to different query patterns. When considering how to use partitioning, think about how your data is used, and its granularity.

In this example, a DataFrame containing product data is partitioned by Category:

```
df.write.format("delta").partitionBy("Category").saveAsTable("partitioned_products")
```

In the Lakehouse Explorer, you can see the data is a partitioned table.

- There's one folder for the table, called "partitioned_products."
- There are subfolders for each category, for example "Category=Bike Racks", etc.

Files > partitioned_products > Category=Bike Racks			
Name	Date modified	Type	Size
part-00022-0ae3e0aa-093e-4470-807c-c77ed1d...	8/21/2024 2:2...	parquet	1 KB

We can create a similar partitioned table using SQL:

```
%%sql
CREATE TABLE partitioned_products (
  ProductID INTEGER,
  ProductName STRING,
  Category STRING,
  ListPrice DOUBLE
)
PARTITIONED BY (Category);
```

5. Work with delta tables in Spark

<https://learn.microsoft.com/en-us/training/modules/work-delta-lake-tables-fabric/4-work-delta-data>

Work with delta tables in Spark

Completed

You can work with delta tables (or delta format files) to retrieve and modify data in multiple ways.

Using Spark SQL

The most common way to work with data in delta tables in Spark is to use Spark SQL. You can embed SQL statements in other languages (such as PySpark or Scala) by using the **spark.sql** library. For example, the following code inserts a row into the **products** table.

```
spark.sql("INSERT INTO products VALUES (1, 'Widget', 'Accessories', 2.99)")
```

Alternatively, you can use the `%%sql` magic in a notebook to run SQL statements.

```
%%sql
```

```
UPDATE products  
SET ListPrice = 2.49 WHERE ProductId = 1;
```

Use the Delta API

When you want to work with delta files rather than catalog tables, it may be simpler to use the Delta Lake API. You can create an instance of a **DeltaTable** from a folder location containing files in delta format, and then use the API to modify the data in the table.

```
from delta.tables import *  
from pyspark.sql.functions import *  
  
# Create a DeltaTable object  
delta_path = "Files/mytable"  
deltaTable = DeltaTable.forPath(spark, delta_path)  
  
# Update the table (reduce price of accessories by 10%)  
deltaTable.update(  
    condition = "Category == 'Accessories'",  
    set = { "Price": "Price * 0.9" })
```

Use *time travel* to work with table versioning

Modifications made to delta tables are logged in the transaction log for the table. You can use the logged transactions to view the history of changes made to the table and to retrieve older versions of the data (known as *time travel*)

To see the history of a table, you can use the `DESCRIBE` SQL command as shown here.

```
%%sql
```

```
DESCRIBE HISTORY products
```

The results of this statement show the transactions that have been applied to the table, as shown here (some columns have been omitted):

version	timestamp	operation	operationParameters
2	2023-04-04T21:46:43Z	UPDATE	{"predicate":"(ProductId = 1)"}
1	2023-04-04T21:42:48Z	WRITE	{"mode":"Append","partitionBy":[]}
0	2023-04-04T20:04:23Z	CREATE TABLE	{"isManaged":"true","description":null,"partitionBy":[],"properties":{}}

To see the history of an external table, you can specify the folder location instead of the table name.

```
%%sql  
  
DESCRIBE HISTORY 'Files/mytable'
```

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

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

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

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

6. Use delta tables with streaming data

<https://learn.microsoft.com/en-us/training/modules/work-delta-lake-tables-fabric/5-use-delta-lake-streaming-data>

Use delta tables with streaming data

Completed

All of the data we 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 processes batch data and streaming data in the same way, enabling streaming data to be processed in real-time using the same API.

Spark Structured Streaming

A typical stream processing solution involves:

- Constantly reading a stream of data from a *source*.
- Optionally, processing the data to select specific fields, aggregate and group values, or otherwise manipulating the data.
- 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
- File system locations.

Tip

For more information about Spark Structured Streaming, see [Structured Streaming Programming Guide](#) in the Spark documentation.

Streaming with Delta tables

You can use a Delta 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 table as a sink. You can then query the table to see the latest streamed data. Or you could read a Delta as a streaming source, enabling near real-time reporting as new data is added to the table.

Using a Delta table as a streaming source

In the following PySpark example, a Delta table is created to store details of Internet sales orders:

```
%%sql
CREATE TABLE orders_in
(
    OrderID INT,
    OrderDate DATE,
    Customer STRING,
    Product STRING,
    Quantity INT,
    Price DECIMAL
)
USING DELTA;
```

A hypothetical data stream of internet orders is inserted into the orders_in table:

```
%%sql
INSERT INTO orders_in (OrderID, OrderDate, Customer, Product, Quantity, Price)
VALUES
    (3001, '2024-09-01', 'Yang', 'Road Bike Red', 1, 1200),
    (3002, '2024-09-01', 'Carlson', 'Mountain Bike Silver', 1, 1500),
    (3003, '2024-09-02', 'Wilson', 'Road Bike Yellow', 2, 1350),
    (3004, '2024-09-02', 'Yang', 'Road Front Wheel', 1, 115),
```

```
(3005, '2024-09-02', 'Rai', 'Mountain Bike Black', 1, NULL);
```

To verify, you can read and display data from the input table:

```
# Read and display the input table
df = spark.read.format("delta").table("orders_in")

display(df)
```

The data is then loaded into a streaming DataFrame from the Delta table:

```
# Load a streaming DataFrame from the Delta table
stream_df = spark.readStream.format("delta") \
    .option("ignoreChanges", "true") \
    .table("orders_in")
```

Note

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

You can check that the stream is streaming by using the `isStreaming` property which should return True:

```
# Verify that the stream is streaming
stream_df.isStreaming
```

Transform the data stream

After reading the data from the Delta table into a streaming DataFrame, you can use the Spark Structured Streaming API to process it. For example, you could count the number of orders placed every minute and send the aggregated results to a downstream process for near-real-time visualization.

In this example, any rows with NULL in the Price column are filtered and new columns are added for `IsBike` and `Total`.

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

transformed_df = stream_df.filter(col("Price").isNotNull()) \
    .withColumn('IsBike', expr("INSTR(Product, 'Bike') > 0").cast('int')) \
    .withColumn('Total', expr("Quantity * Price").cast('decimal'))
```

Using a Delta table as a streaming sink

The data stream is then written to a Delta table:

```
# Write the stream to a delta table
output_table_path = 'Tables/orders_processed'
checkpointpath = 'Files/delta/checkpoint'
deltastream = transformed_df.writeStream.format("delta").option("checkpointLocation", checkpointpath)

print("Streaming to orders_processed...")
```

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 starts, you can query the Delta Lake table to see what is in the output table. There might be a short delay before you can query the table.

```
%%sql
SELECT *
  FROM orders_processed
 ORDER BY OrderID;
```

In the results of this query, order 3005 is excluded because it had NULL in the Price column. And the two columns that were added during the transformation are displayed - IsBike and Total.

OrderID	OrderDate	Customer	Product	Quantity	Price	IsBike	Total
3001	2023-09-01	Yang	Road Bike Red	1	1200	1	1200
3002	2023-09-01	Carlson	Mountain Bike Silver	1	1500	1	1500
3003	2023-09-02	Wilson	Road Bike Yellow	2	1350	1	2700
3004	2023-09-02	Yang	Road Front Wheel	1	115	0	115

When finished, stop the streaming data to avoid unnecessary processing costs using the `stop` method:

```
# Stop the streaming data to avoid excessive processing costs
deltastream.stop()
```

Tip

For more information about using Delta tables for streaming data, see [Table streaming reads and writes](#) in the Delta Lake documentation.

7. Exercise - Use delta tables in Apache Spark

<https://learn.microsoft.com/en-us/training/modules/work-delta-lake-tables-fabric/6-exercise-delta-tables>

Exercise - Use delta tables in Apache Spark

Completed

Now it's your turn to use Apache Spark to work with delta tables in a Microsoft Fabric lakehouse.

Note

To complete this exercise, you need a Microsoft Fabric tenant. See [Getting started with Fabric](#) to find out how to enable a Fabric trial license.

Launch the exercise and follow the instructions.

[Launch Exercise](#)

8. Module assessment

<https://learn.microsoft.com/en-us/training/modules/work-delta-lake-tables-fabric/7-knowledge-check>

Module assessment

Completed

9. Summary

Summary

Completed

Delta Lake is a technology that adds relational database semantics to Apache Spark. Tables in a Microsoft Fabric lakehouse are based on Delta Lake, enabling you to take advantage of many advanced features and techniques through the Delta Lake API.

Tip

For more information about Delta Lake, see the [Delta Lake documentation](#).