Microsoft Fabric: Data Engineering

We now create notebooks in Fabric:

☑ Load Data from Lakehouse in Fabric using Notebook (Spark)

🧿 Prerequisites:

- You have a **Lakehouse** with data (in either *Files* or *Tables*)
- You have **Microsoft Fabric enabled workspace**
- You are using the **default Spark runtime**

◇ STEP 1: Create a New Notebook

1. Go to your **Fabric workspace**
2. Click **New → Notebook**
3. It will open a new notebook interface with **PySpark** as default language (you can confirm this in the top-left dropdown)

◇ STEP 2: Attach Notebook to Your Lakehouse

1. In the notebook toolbar, click **"Lakehouse" (⚙)**
2. Choose **"Add Lakehouse"**
3. Select your existing Lakehouse where data is stored
4. Once added, you will see:
    a. `/lakehouse/default/Files/`
    b. `/lakehouse/default/Tables/`
    This gives you Spark access to the Lakehouse paths.

◇ STEP 3: Load Data Using PySpark

🧪 Option 1: Load from Tables in Lakehouse

If you've loaded or transformed data into tables:

```
python
CopyEdit
df = spark.read.table("your_table_name")
df.show()
```

✅ Replace `your_table_name` with the actual table name from the `Tables` section.

🧪 Option 2: Load from Files (e.g., CSV or Parquet)

a) Load a CSV file:

```
python
CopyEdit
df = spark.read.option("header",
True).csv("/lakehouse/default/Files/raw/yourfile.csv")
df.show()
```

b) Load a Parquet file:

```
python
CopyEdit
df = spark.read.parquet("/lakehouse/default/Files/raw/yourfile.parquet")
df.show()
```

✅ Adjust the path if your folder is different (e.g., `/Files/data/`, etc.)

◇ STEP 4: Explore or Transform Data

You can now apply Spark functions like:

```
python
CopyEdit
df.printSchema()
df.select("column1", "column2").filter(df["column1"] > 100).show()
```

You can also use:

- `.groupBy()`
- `.agg()`
- `.withColumn()`
- `.drop()`, `.fillna()`, etc.

◇ STEP 5: Save Transformed Data (Optional)

a) Save as Table in Lakehouse:

```
python
CopyEdit
df.write.mode("overwrite").saveAsTable("new_transformed_table")
```

b) Save as CSV in Files folder:

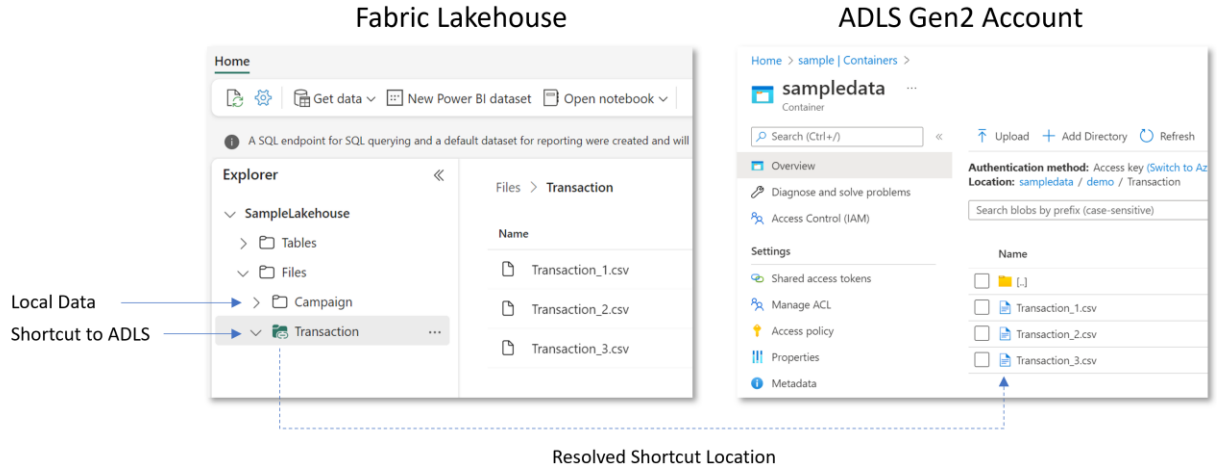
```
python
CopyEdit
df.write.mode("overwrite").option("header",
True).csv("/lakehouse/default/Files/processed/output_data")
```

Shortcuts

https://learn.microsoft.com/en-us/fabric/onelake/onelake-shortcuts

What are shortcuts

Shortcuts are objects in OneLake that point to other storage locations. The location can be internal or external to OneLake. The location that a shortcut points to is known as the target path of the shortcut. The location where the shortcut appears is known as the shortcut path. Shortcuts appear as folders in OneLake and any workload or service that has access to OneLake can use them. Shortcuts behave like symbolic links. They're an independent object from the target. If you delete a shortcut, the target remains unaffected. If you move, rename, or delete a target path, the shortcut can break.



Where can I create shortcuts?

You can create shortcuts in lakehouses and Kusto Query Language (KQL) databases. Furthermore, the shortcuts you create within these items can point to other OneLake locations, Azure Data Lake Storage (ADLS) Gen2, Amazon S3 storage accounts, or Dataverse. You can even create shortcuts to on-premises or network-restricted locations with the use of the Fabric on-premises data gateway (OPDG).

You can use the Fabric UI to create shortcuts interactively, and you can use the REST API to create shortcuts programmatically.

Managed Tables

Managed tables are those where Fabric manages both the data and metadata. This means:

- Fabric automatically controls the storage, schema, and security of the data.
- Data is stored in a default location, usually within the Tables folder of your lakehouse.
- When you delete a managed table, both the data and metadata are removed.

Advantages:

- Simplicity of management
- Seamless integration with other Fabric tools
- Automatic performance optimization
- Ideal for scenarios where the data lifecycle is closely tied to the table definition

Disadvantages:

- Less flexibility in controlling the data storage location
- Risk of accidental data loss when deleting the table

External Tables

External tables are those where Fabric manages only the metadata, while the data is stored in a location specified by the user. Important characteristics:

- You have full control over the data storage location.
- When deleting an external table, only the metadata is removed, and the data remains intact.
- They are ideal for integration with external data sources or data migration.

Advantages:

- Greater flexibility in managing data storage
- Data persistence independent of a table definition
- Facilitates integration with external systems and migration scenarios

Disadvantages:

- Requires more manual management

- It can be more complex to set up and maintain

Query Performance in Fabric

In **Microsoft Fabric**, two powerful Delta Lake operations — **OPTIMIZE** and **VACUUM** — help improve **query performance** and **storage efficiency** in Lakehouse environments.

Here's a detailed explanation with syntax, use cases, and Fabric-specific notes:

☑ 1. `OPTIMIZE` in Fabric

🔧 What it does:

- Combines **small files into larger ones** to reduce file fragmentation.
- Helps **improve read performance** for large Delta tables.
- Can be enhanced with **Z-ORDER** to cluster related data for faster filtering.

🧠 Why use it?

- Small files (created during frequent appends or streaming) lead to **slow queries**.
- `OPTIMIZE` reduces **metadata overhead** and **file open/read time**.

📓 Syntax:

```
sql
CopyEdit
OPTIMIZE [table_name]
```

With Z-Ordering:

```
sql
CopyEdit
OPTIMIZE sales ZORDER BY (order_date)
```

ZORDER organizes data in a multi-dimensional way for efficient filtering and range scans.

📌 Example:

```
sql
CopyEdit
OPTIMIZE sales_data ZORDER BY (customer_id, region)
```

📝 Notes for Fabric:

- Run this in a **Lakehouse SQL endpoint** (use SQL Analytics endpoint).
- Can also be triggered in a **notebook** using `%sql` magic command:

```
python
CopyEdit
%sql
OPTIMIZE mytable ZORDER BY (event_time)
```

☑ 2. VACUUM in Fabric

🧹 What it does:

- Cleans up **obsolete data files** from a Delta table (from deletes, updates, schema changes, OPTIMIZE, etc.).
- Reclaims **storage space**.

🧠 Why use it?

- After updates or deletions, **old versions of files** remain for time travel or rollback.
- VACUUM deletes files **no longer needed** after a retention period.

📜 Syntax:

```
sql
CopyEdit
VACUUM [table_name] RETAIN [hours] HOURS
```

Example:

```
sql
CopyEdit
VACUUM sales RETAIN 168 HOURS
```

This keeps the last 7 days (168 hours) of historical data and removes the rest.

⚠️ Default Retention:

- Minimum retention is **168 hours (7 days)** by default for safety.
- **Lowering it (e.g., to 1 hour)** is **not allowed by default** unless you explicitly override it in non-Fabric Delta environments (not recommended in Fabric).

📝 Notes for Fabric:

- Make sure no jobs are depending on time-travel before running VACUUM.
- This is especially useful after running **`DELETE`**, **`UPDATE`**, or **`OPTIMIZE`** operations.

SPARK UTILITIES

https://learn.microsoft.com/en-us/fabric/data-engineering/microsoft-spark-utilities

In **Microsoft Fabric**, **Microsoft Spark Utilities**—commonly accessed via the `mssparkutils` module—are a set of helper functions used in **Spark Notebooks** to interact with files, secrets, environment variables, and other resources in the Fabric Lakehouse environment.

These utilities make it easier to perform operations **without writing full Spark code**.

🧰 Main Categories of Microsoft Spark Utilities (`mssparkutils`)

Category	Description	Common Functions
📁 **File System**	Manage files in Lakehouse / OneLake	`fs.ls()`, `fs.cp()`, `fs.rm()`, `fs.mkdirs()`
🔐 **Secrets**	Retrieve secrets from Key Vault or linked services	`credentials.getSecret()`
🌐 **Environment**	Get job metadata and runtime info	`env.getJobId()`, `env.getWorkspaceId()`
📓 **Notebook**	Run and chain other notebooks	`notebook.run()`, `notebook.exit()`

🔍 1. File System Utilities

Used to interact with the **Files section of a Lakehouse**.

✨ Examples:

```
python
CopyEdit
```

```
from notebookutils import mssparkutils

# List files in a directory
mssparkutils.fs.ls("Files/raw/")

# Copy a file
mssparkutils.fs.cp("Files/raw/file.csv", "Files/processed/file.csv")

# Remove a file
mssparkutils.fs.rm("Files/raw/old.csv", recurse=True)
```

🔐 2. Secrets Management

Fetch credentials securely from Azure Key Vault or Linked Services.

✨ Example:

```
python
CopyEdit
# Get a secret value (like a DB password or API key)
api_key = mssparkutils.credentials.getSecret("my_keyvault", "api_key")
print(api_key)
```

🌍 3. Environment Utilities

Useful for logging or tracking job context during notebook execution.

✨ Examples:

```
python
CopyEdit
# Get job ID
job_id = mssparkutils.env.getJobId()
print(f"Running job ID: {job_id}")
```

```
# Get workspace ID
workspace_id = mssparkutils.env.getWorkspaceId()
```

🔁 4. Notebook Utilities

Trigger or chain **notebook executions** from another notebook.

✨ Example:

```
python
CopyEdit
# Run another notebook and capture output
output = mssparkutils.notebook.run("/Notebooks/CleanData", 60,
{"inputPath": "Files/raw/"})

# Exit the current notebook
mssparkutils.notebook.exit("Finished successfully")
```

📌 Real-World Use Case Example

```
python
CopyEdit
from notebookutils import mssparkutils
from pyspark.sql import SparkSession

# Load raw data if exists
if len(mssparkutils.fs.ls("Files/raw/")) > 0:
    df = spark.read.csv("Files/raw/data.csv", header=True,
inferSchema=True)

df.write.format("delta").mode("overwrite").save("Tables/cleaned_data")
```

⚠ Notes:

- Only available in **Spark Notebooks** in Fabric.
- You must use `from notebookutils import mssparkutils` (not standard Python modules).
- These are Fabric-specific and **not part of open-source PySpark**.