Chapter 3. Mastering Relational Entities in Databricks

Relational entities, particularly databases, tables, and views, are essential components for organizing and managing structured data in Databricks. Understanding how these entities interact with the metastore and storage locations is crucial for efficient querying and data management. In this chapter, we will cover in detail how these entities function within the Databricks environment and understand their relationship with the underlying storage.

Understanding Relational Entities

This section provides a detailed understanding of relational entities in Databricks, covering databases, tables, and views, with a focus on their interactions with both the metastore and storage systems.

Databases in Databricks

In Databricks, a database essentially corresponds to a schema in a data catalog. This means that when you create a database, you're essentially defining a logical structure where tables, views, and functions can be organized. This collection of database objects is called a *schema*. You have the flexibility to create a database using either the `CREATE DATABASE` or `CREATE SCHEMA` syntax, as they are functionally equivalent.
Every Databricks workspace includes a local data catalog, called `hive_metastore`, that all clusters can access to persist object metadata. The Hive metastore serves as a repository for metadata, storing essential information about data structures such as databases, tables, and partitions. This metadata includes details like table definitions, data formats, and storage locations.

Default database

By default, a `database` named "default" is provided in the `hive_metastore` catalog. When you create tables without explicitly specifying a database name, they are created under the default database. The data for these tables is stored in the default directory for Hive, typically located at */user/hive/warehouse* on the DBFS, as illustrated in Figure 3-1.

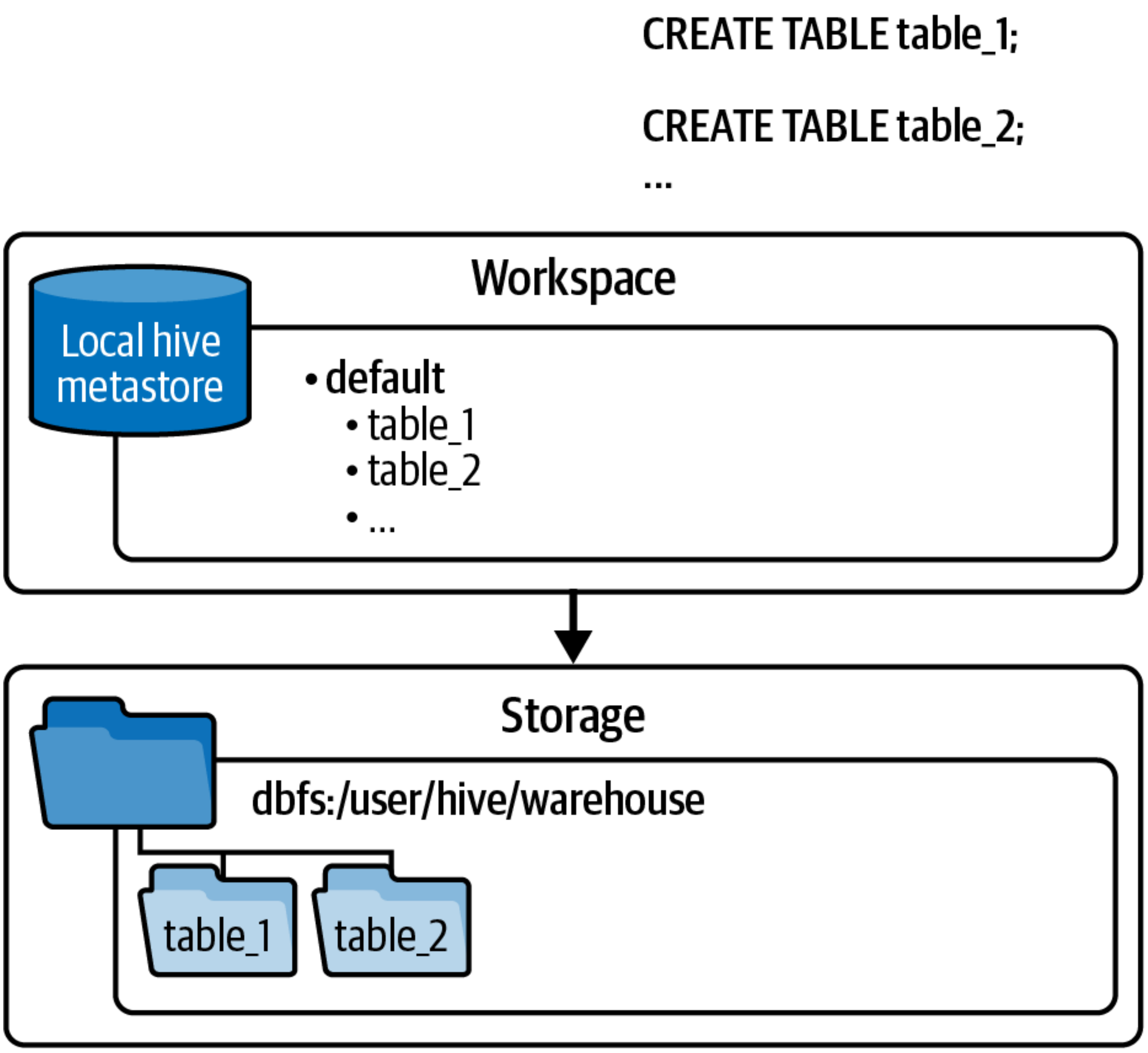


Figure 3-1. Creating tables under the default database

Creating databases

Apart from the `default` database, you can create additional databases using the `CREATE DATABASE` or `CREATE SCHEMA` syntax. These databases are also stored in the Hive metastore, with their corresponding folders under the default Hive directory in */user/hive/warehouse*. These database folders are distinguished by the *.db* extension to differentiate them from table directories, as illustrated in Figure 3-2.

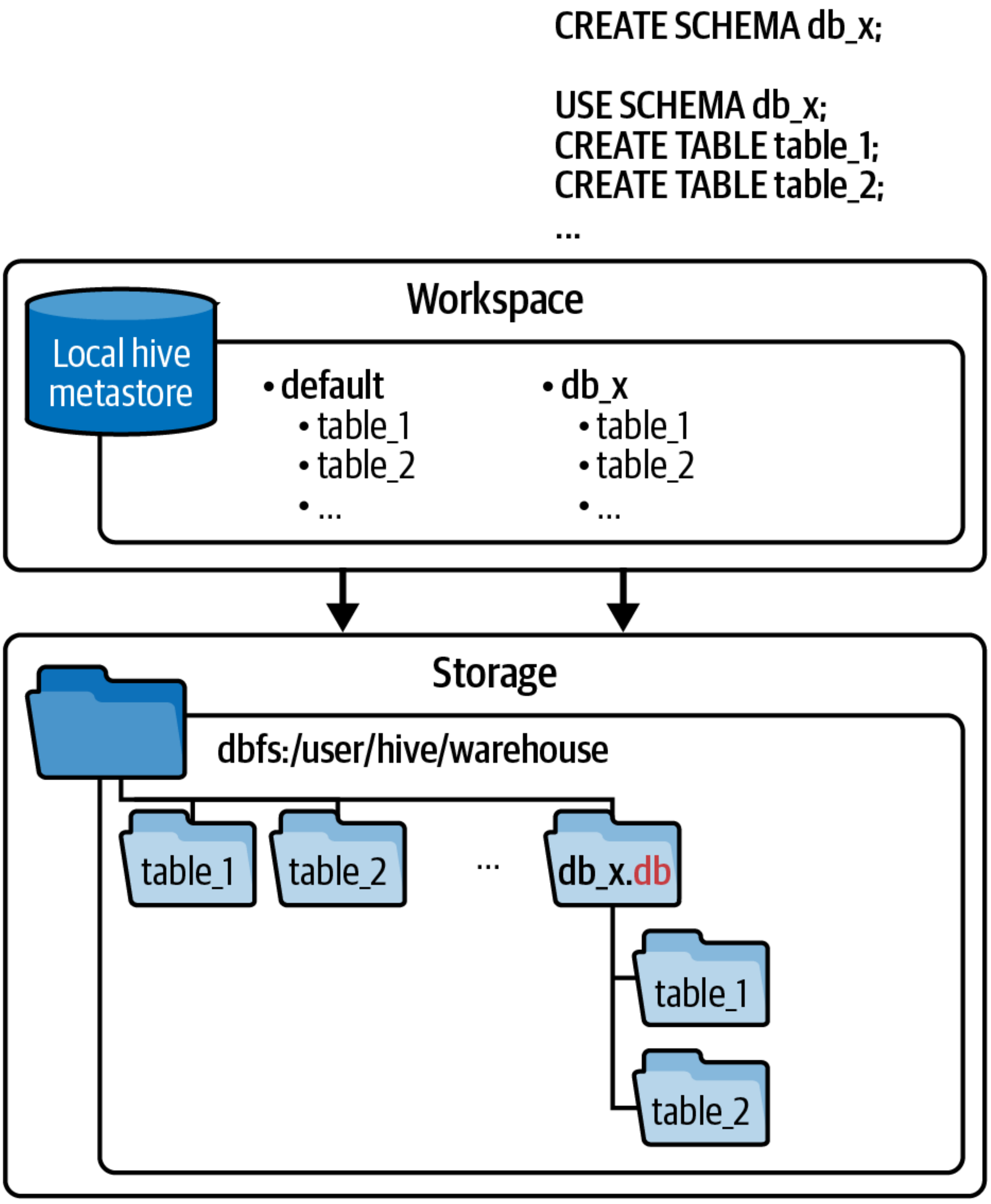


Figure 3-2. Creating an additional database and tables within this database

Custom-location databases

Moreover, you can create databases outside of the default Hive directory by specifying a custom location using the `LOCATION` keyword in the `CREATE SCHEMA` syntax. In this case, the database definition still resides in the Hive metastore, but the database folder is located in the specified custom path. Tables created within these custom databases will have their data stored in the respective database folder within the custom location, as illustrated in Figure 3-3.

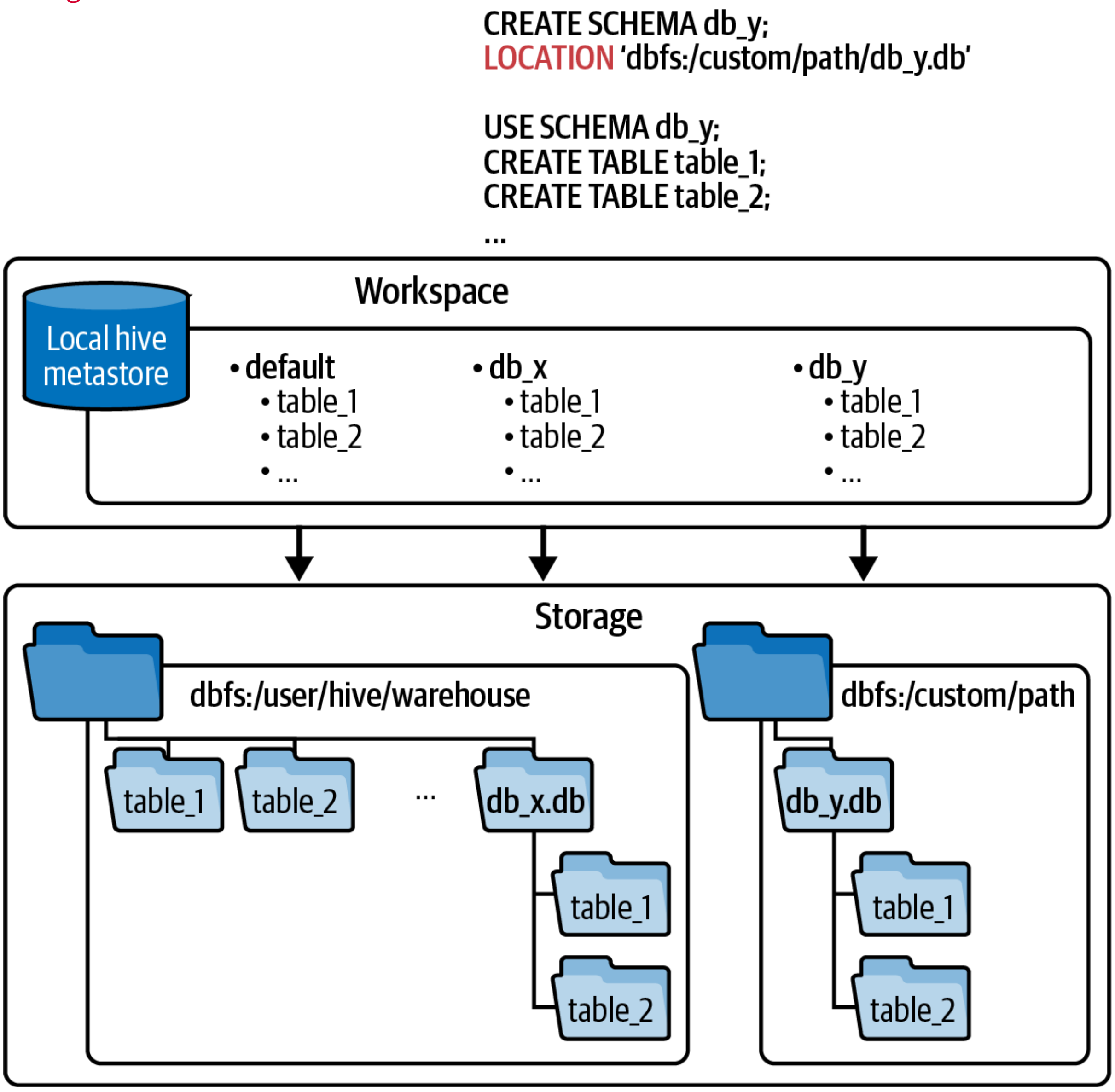


Figure 3-3. Creating a database in a custom location

Tables in Databricks

In Databricks, there are two types of tables: managed tables and external tables. Understanding the distinction between them is essential for effectively managing your data. Table 3-1 summarizes the key differences between these two types of tables.

Managed table	External table
Created within its own database directory: `CREATE TABLE table_name`	Created outside the database directory (in a path specified by the `LOCATION` keyword): `CREATE TABLE table_name LOCATION <path>`
Dropping the table deletes both the metadata and the underlying data files of the table.	Dropping the table only removes the metadata of the table. It does not delete its underlying data files.

Table 3-1. Comparison of managed and external tables in Delta Lake

Let's dive deeper to gain a comprehensive understanding of these two types of tables.

Managed tables

A managed table is the default type in Databricks, where the table and its associated data are managed by the metastore, typically the Hive metastore or Unity Catalog. When you create a managed table, the table data is stored in a location controlled by the metastore. This means that the metastore owns both the metadata and the table data, enabling it to manage the complete lifecycle of the table. This integrated management simplifies data lifecycle management tasks, such as table deletion and maintenance.

So, when you drop a managed table, not only is its metadata removed from the metastore, but the underlying data files associated with the table are also deleted from storage. This approach ensures that the data remains consistent with the table definition throughout its lifecycle. However, it's essential to exercise caution when dropping managed tables, as the associated data will be permanently removed.

External tables

In contrast to managed tables, an external table in Databricks is a table where only its metadata is managed by the metastore, while the data files themselves reside outside the database directory. When creating an external table, you specify the location of the data files using the `LOCATION` keyword:

```
CREATE TABLE table_name
LOCATION <path>
```

Since the metastore does not own the underlying data files, dropping an external table only removes the metadata associated with the table, leaving its data files intact. This distinction is crucial, as it enables you to manage the actual data files of the table separately from its metadata. This is particularly useful when working with data that is stored in external locations outside the DBFS, like in S3 buckets or Azure storage containers.

To better understand external tables, let's revisit our diagram. Figure 3-4 illustrates creating an external table in the default database. We simply use the `CREATE TABLE` statement with the `LOCATION` keyword. The definition of this external table will be in the Hive metastore under the `default` database, while the actual data files will reside in the specified external location.

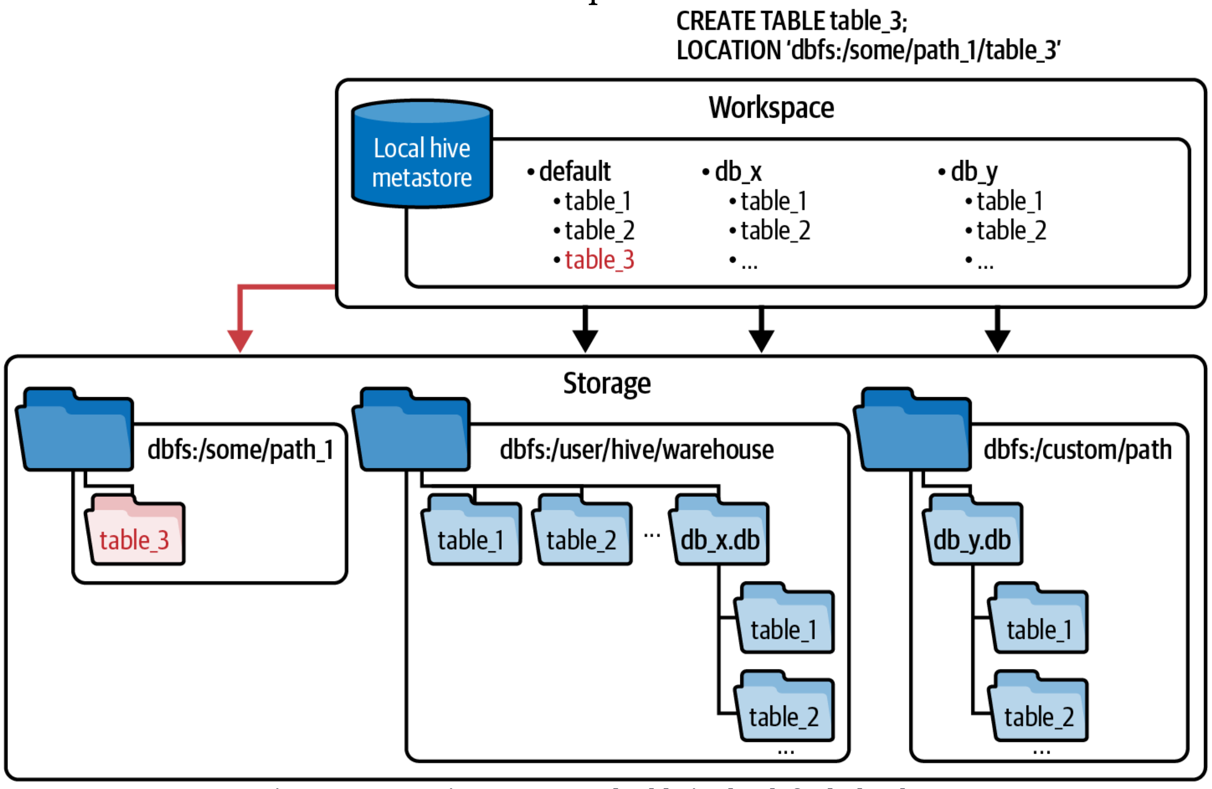


Figure 3-4. Creating an external table in the default database

Similarly, we can create an external table in any database. Figure 3-5 illustrates creating an external table in our database `db_x`. First, we specify the database name via the `USE DATABASE` or `USE SCHEMA` keyword. Then, we create the table with the `LOCATION` keyword, indicating the path where the external table data should be stored. This path could be the same as the previous one used for `default.table_3` table or a different location, depending on our requirements. And again, the table definition will be stored in the Hive metastore, while the data files will be located in the given external location.

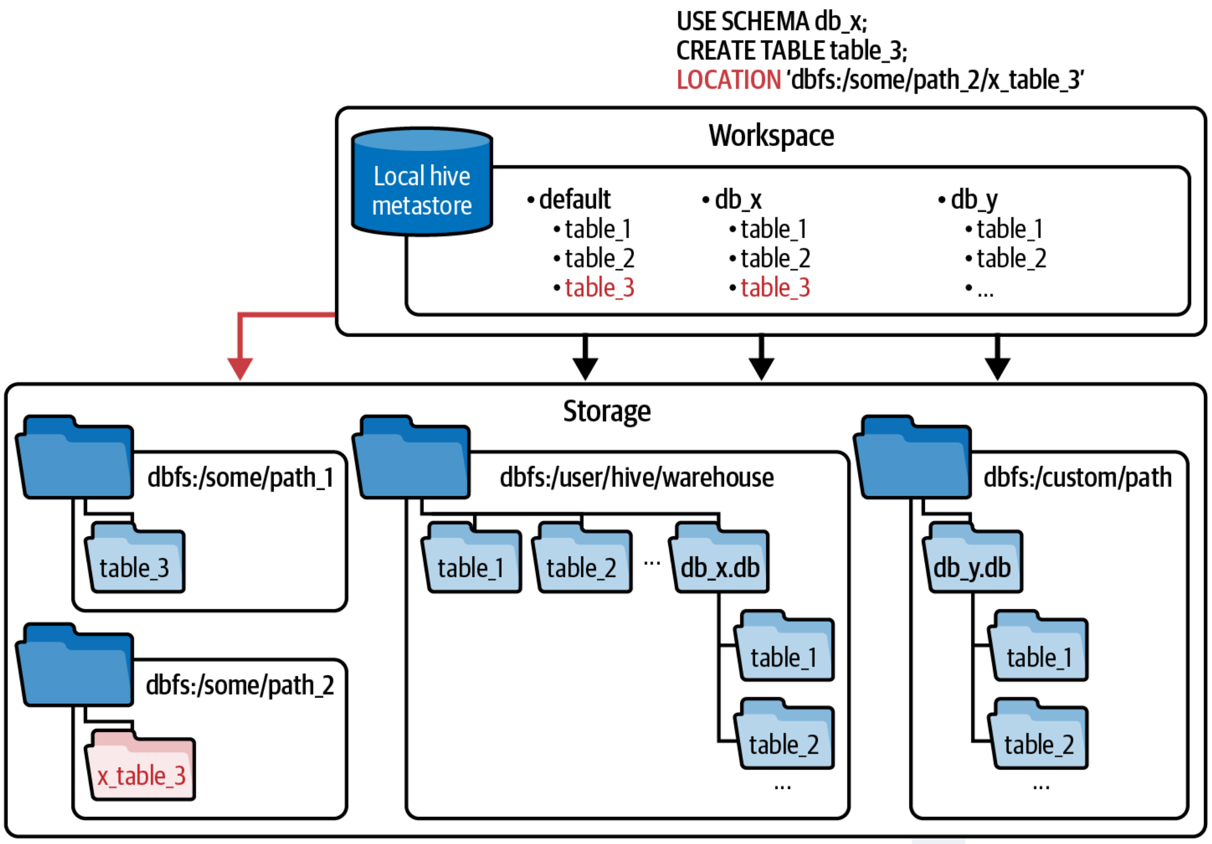


Figure 3-5. Creating an external table in the new database `db_x`

Even if the database was created in a custom location outside of the default Hive directory, we can still create external tables within it. Figure 3-6 illustrates this scenario by using our custom-location database `db_y`. Once again, we specify the database using the `USE SCHEMA` keyword and create the external table with the `LOCATION` keyword. In this scenario, let's assume we choose the same path as in the previous example. As before, the table definition will be stored in the metastore, while the data files will be located in the specified external location.

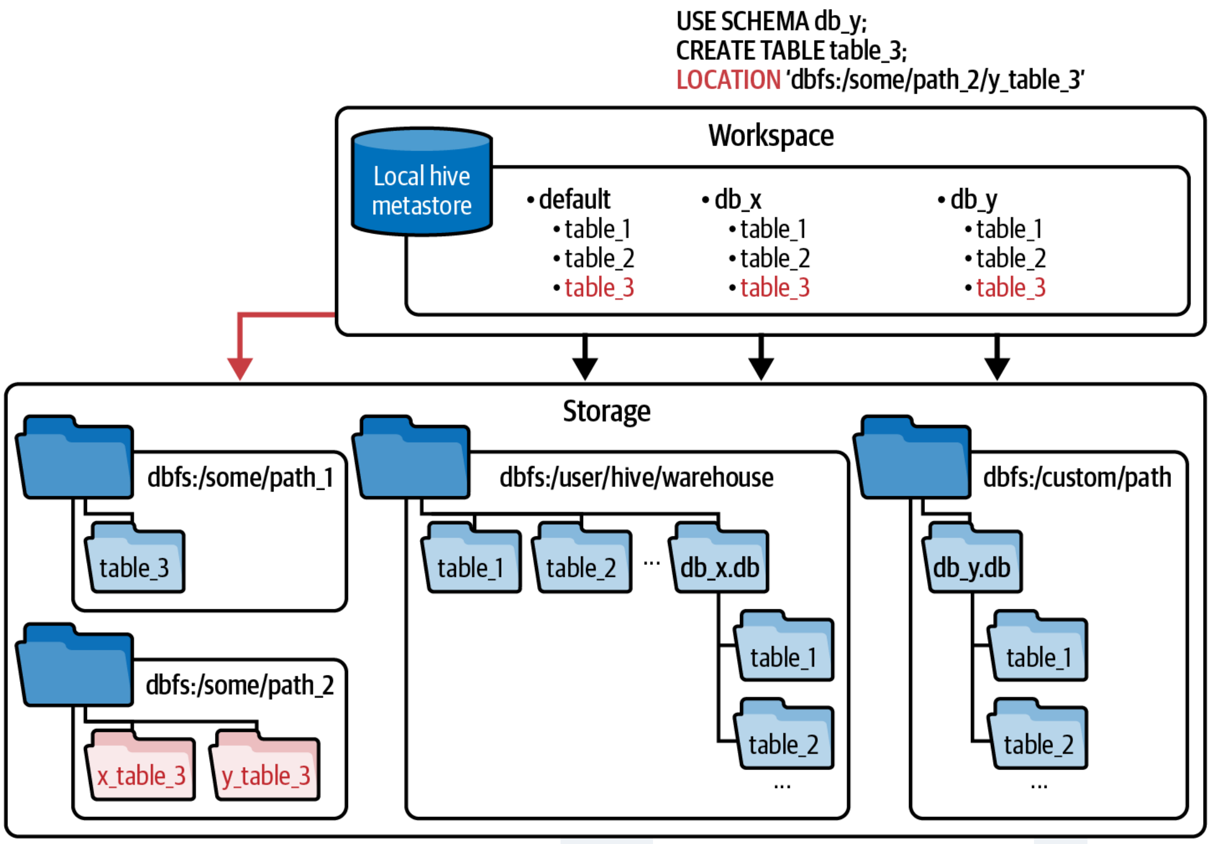


Figure 3-6. Creating an external table, `table_3`, in the custom-location database `db_y`

In summary, Databricks provides two types of tables: managed tables and external tables. Depending on the use case and data requirements, choosing the appropriate table type ensures efficient data organization, storage, and maintenance. Opting for managed tables ensures integrated management, while choosing external tables provides greater flexibility and control when managing your tables.

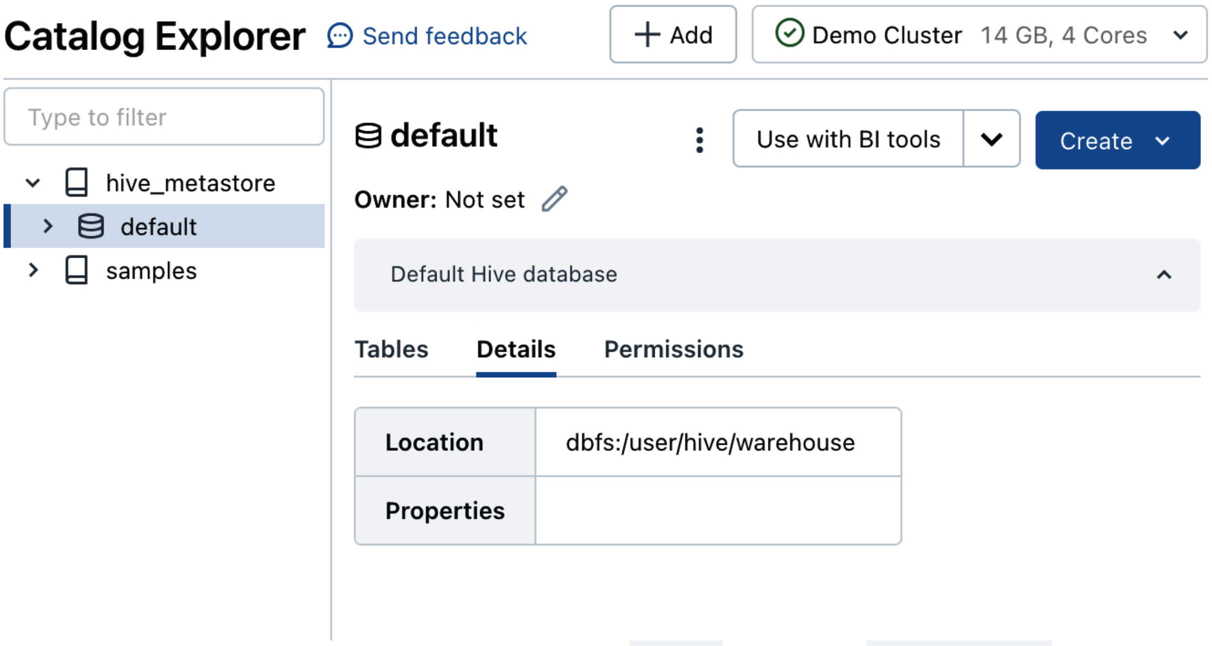
Putting Relational Entities into Practice

Let's now put theory into practice. In this section, we will use a new SQL notebook titled "3.1 - Databases and Tables" to create managed and external tables in various database types. In addition, we will explore the differences in behavior when dropping each type of table.

Working in the Default Schema

Before we start, let's explore the Catalog Explorer, where we can access the Hive metastore for our Databricks workspace. To open the Catalog Explorer, click the Catalog tab in the left sidebar of your Databricks workspace.

By default, under the `hive_metastore` catalog, there's a database named `default`, as illustrated in Figure 3-7. We'll begin by creating some tables within this default database.



Location	dbfs:/user/hive/warehouse
Properties	

Figure 3-7. Catalog Explorer showing the `default` database in `hive_metastore`

Creating managed tables

First, we create a managed table named `managed_default` and populate it with data:

```
USE CATALOG hive_metastore;

CREATE TABLE managed_default
 (country STRING, code STRING, dial_code STRING);

INSERT INTO managed_default
VALUES ('France', 'Fr', '+33')
```

Since we're not specifying the `LOCATION` keyword, this table is considered managed in this database. Checking back in the Catalog Explorer, we can confirm that the `managed_default` table has been created under the default database. Alternatively, without leaving the working notebook, you can directly access the catalog by clicking the catalog icon located in the sidebar of the notebook editor (Figure 3-8).

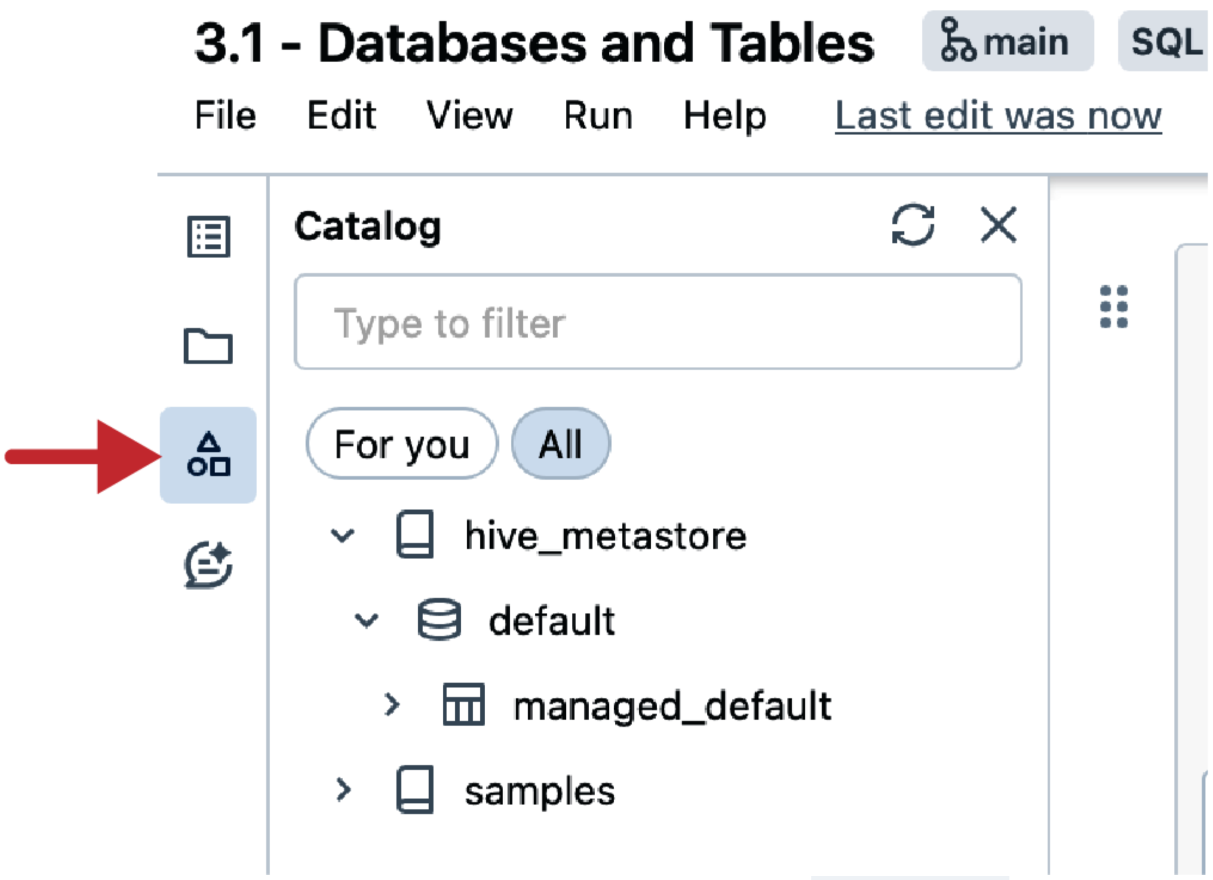
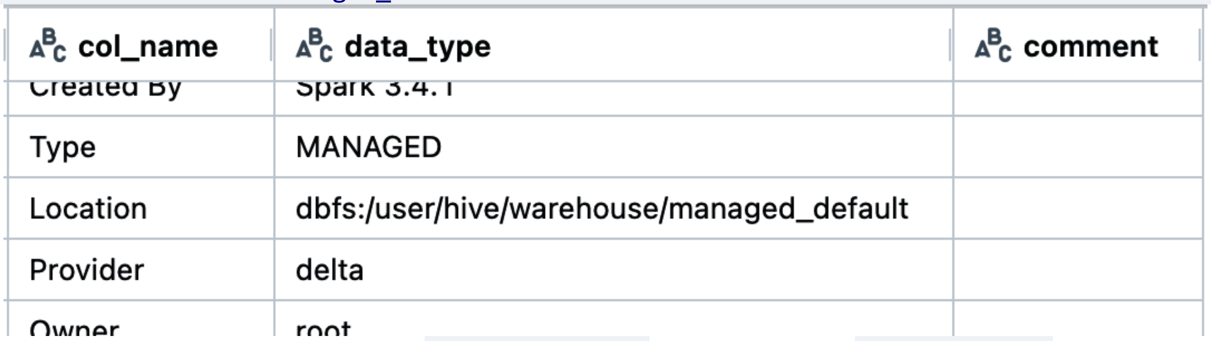


Figure 3-8. The catalog in the notebook editor showing the `managed_default` table

Executing the `DESCRIBE EXTENDED` command on our table provides advanced metadata information, as illustrated in Figure 3-9.

```
DESCRIBE EXTENDED managed_default
```



col_name	data_type	comment
Created By	Spark 3.4.1	
Type	MANAGED	
Location	dbfs:/user/hive/warehouse/managed_default	
Provider	delta	
Owner	root	

Figure 3-9. The output of the `DESCRIBE EXTENDED` command on the `managed_default` table

Among this metadata information, we focus on three key elements:

- The type of table, which is indeed `MANAGED`
- The location, which shows that our table resides in the default Hive metastore under *dbfs:/user/hive/warehouse*
- The provider, which confirms that this is a Delta Lake table

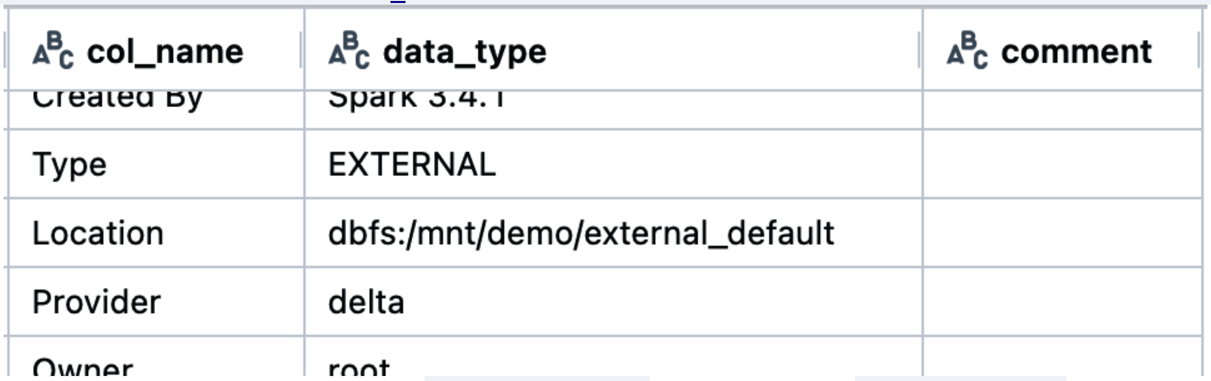
Creating external tables

Next, we create an external table within the default database. To achieve this, we simply add the `LOCATION` keyword followed by the desired storage path. In our case, we'll store this table under the */mnt/demo* directory through the DBFS:

```
CREATE TABLE external_default
 (country STRING, code STRING, dial_code STRING)
LOCATION 'dbfs:/mnt/demo/external_default';

INSERT INTO external_default
VALUES ('France', 'Fr', '+33')
```

After creating and inserting data into this external table, you can use the Catalog Explorer to verify the presence of the table in the Hive metastore. In addition, running `DESCRIBE EXTENDED` on the external table confirms its external nature and its storage location under */mnt/demo*, as illustrated in Figure 3-10.

```
DESCRIBE EXTENDED external_default
```



col_name	data_type	comment
Created By	Spark 3.4.1	
Type	EXTERNAL	
Location	dbfs:/mnt/demo/external_default	
Provider	delta	
Owner	root	

Figure 3-10. The output of the `DESCRIBE EXTENDED` command on the `external_default` table

Dropping tables

If you want to remove tables from the database, you can simply drop them using the `DROP TABLE` command. However, it is important to note that the behavior differs for managed and external tables. Let's discuss the consequences of this action on each table type. We start by running the `DROP TABLE` command on our managed table:

```
DROP TABLE managed_default
```

When you drop a table, it deletes its metadata from the metastore. This means that the table's definition, including its schema, column names, data types, and other relevant information, is no longer stored in the metastore. We can confirm this by trying to query the table, which will result in a "table not found" error:

```
SELECT * FROM managed_default
[TABLE_OR_VIEW_NOT_FOUND] The table or view `managed_default` cannot be found.
Verify the spelling and correctness of the schema and catalog.
```

Dropping the managed table not only removes its metadata from the metastore, but also deletes all associated data files from the storage. This is confirmed by "a file not found" exception received upon checking the table directory:

```
%fs ls 'dbfs:/user/hive/warehouse/managed_default'
```

```
FileNotFoundException:
No such file or directory dbfs:/user/hive/warehouse/managed_default
```

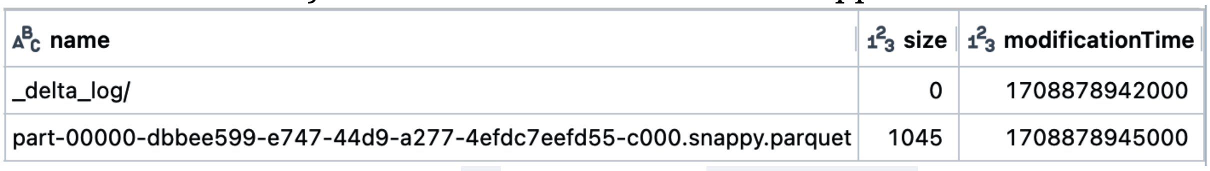
However, when the external table is dropped, we see different behavior:

```
DROP TABLE external_default
```

Dropping the external table also removes its entry from the metastore. We can confirm this by trying to query the table, which should result in a “table not found” error. However, since the underlying data is stored outside the database directory, the data files remain intact. We can easily confirm that the data files of the table still persist by checking the table directory:

```
%fs ls 'dbfs:/mnt/demo/external_default'
```

Figure 3-11 confirms that the data files of the external table continue to exist in the table directory even after the table has been dropped.



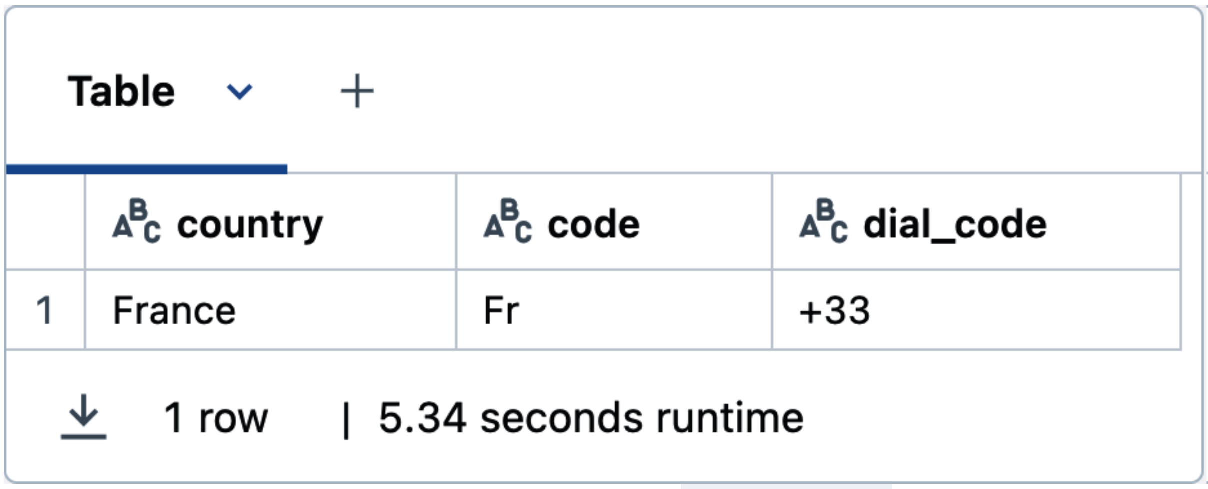
name	size	modificationTime
_delta_log/	0	1708878942000
part-00000-dbbee599-e747-44d9-a277-4efdc7eefd55-c000.snappy.parquet	1045	1708878945000

Figure 3-11. The output of the `%fs` command on the `external_default` table directory

In Databricks, you can directly access a Delta table by querying its directory using the following `SELECT` statement:

```
SELECT * FROM DELTA.`dbfs:/mnt/demo/external_default`
```

Figure 3-12 shows the result of directly querying the table directory, confirming that the data of this external table remains unaffected by dropping the table from the metastore.



	country	code	dial_code
1	France	Fr	+33

1 row | 5.34 seconds runtime

Figure 3-12. The result of directly querying the `external_default` table directory

You can manually remove the table directory and its content by the running the `dbutils.fs.rm` function in Python:

```
%python
dbutils.fs.rm('dbfs:/mnt/demo/external_default', True)
```

Working in a New Schema

In addition to the default database, we can also create additional databases and manage tables within those databases. Let’s walk through the process step-by-step.

Creating a new database

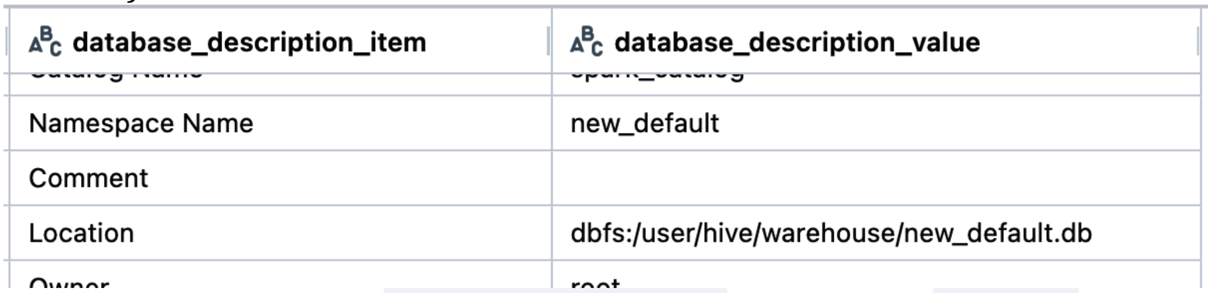
You can create a new database using either the `CREATE SCHEMA` or `CREATE DATABASE` syntax, which are interchangeable:

```
CREATE SCHEMA new_default
```

Once the database is created, you can inspect its metadata using the `DESCRIBE DATABASE EXTENDED` command. This command provides information about the database, such as its location in the underlying storage:

```
DESCRIBE DATABASE EXTENDED new_default
```

As illustrated in Figure 3-13, the new database is stored under the default Hive directory with a *.db* extension to distinguish it from other table folders in the directory.



database_description_item	database_description_value
Namespace Name	new_default
Comment	
Location	dbfs:/user/hive/warehouse/new_default.db
Owner	root

Figure 3-13. The output of the `DESCRIBE DATABASE EXTENDED` command on the `new_default` schema

Creating tables in the new database

Let’s now create managed tables and external tables within our newly created database. To create tables within a database, you need first to set it as the current schema by specifying its name through the `USE DATABASE` keyword:

```
USE DATABASE new_default;

-- create a managed table
CREATE TABLE managed_new_default
 (country STRING, code STRING, dial_code STRING);

INSERT INTO managed_new_default
VALUES ('France', 'Fr', '+33');
-----------------------------------
-- Create an external table
CREATE TABLE external_new_default
 (country STRING, code STRING, dial_code STRING)
LOCATION 'dbfs:/mnt/demo/external_new_default';

INSERT INTO external_new_default
VALUES ('France', 'Fr', '+33');
```

In the Catalog Explorer, you can locate the new schema and confirm that the two tables have been successfully created within this database. Alternatively, you can just refresh the catalog in the notebook editor to show the new objects, as shown in Figure 3-14.

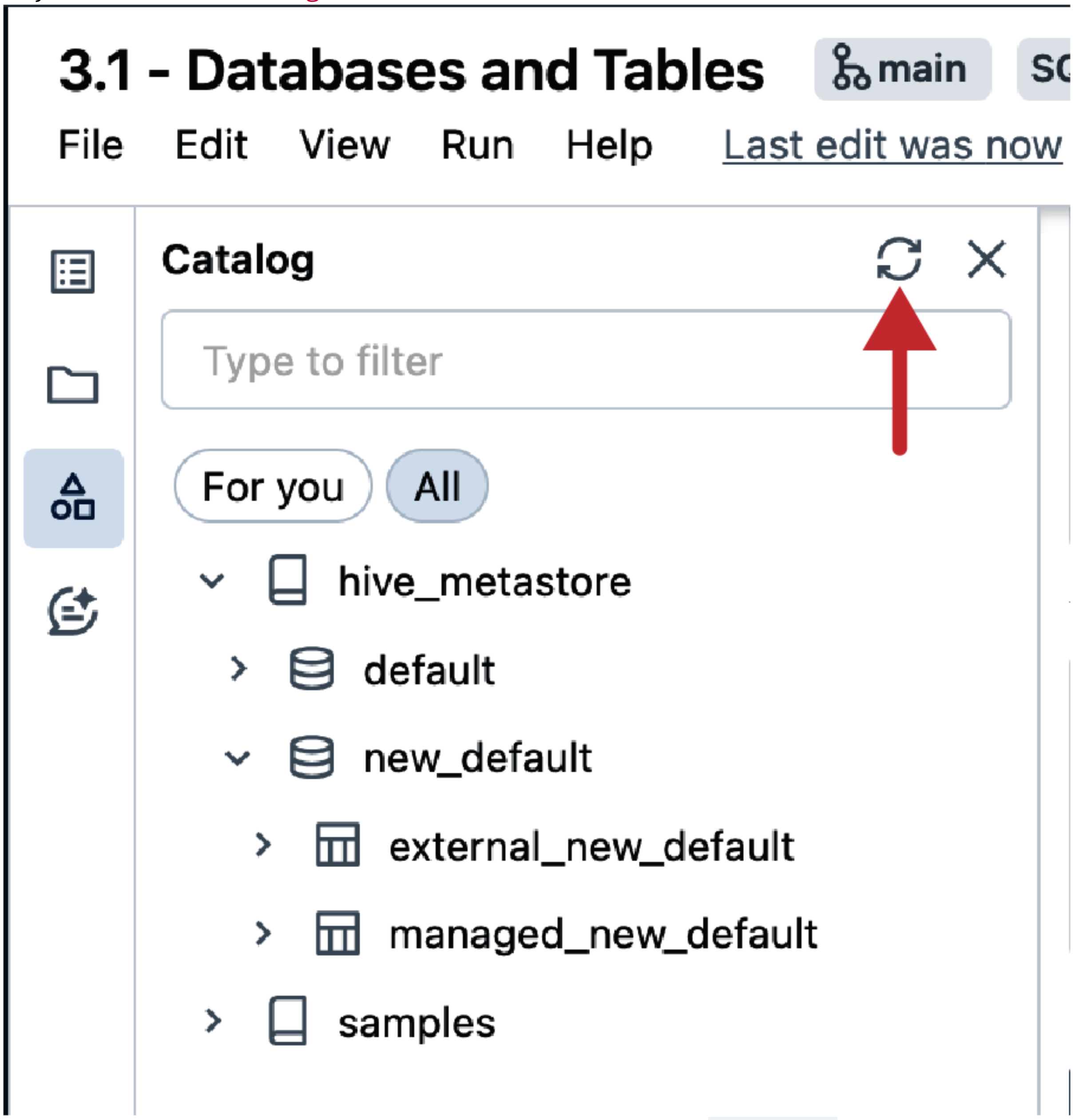
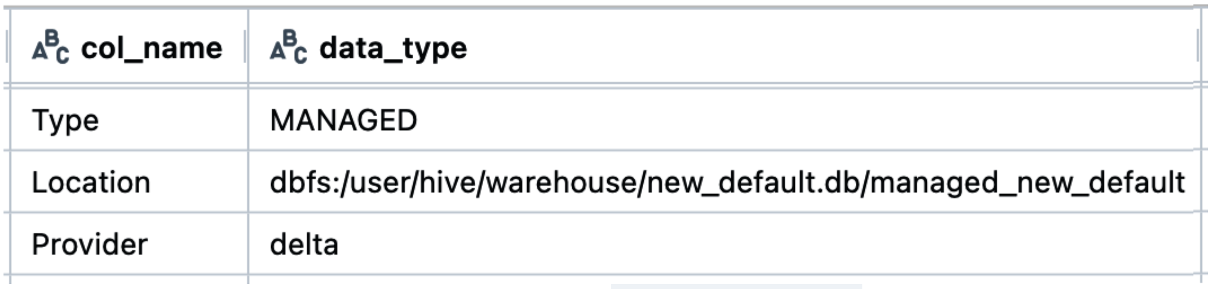


Figure 3-14. Refreshing the catalog in the notebook editor shows the `new_default` schema and its tables

By running `DESCRIBE EXTENDED` on each of these tables, we can see that the first table is indeed a managed table created in its database folder under the default Hive directory (Figure 3-15). Meanwhile, the second table, where we use the `LOCATION` keyword, has been defined as an external table under the */mnt/demo* location (Figure 3-16).

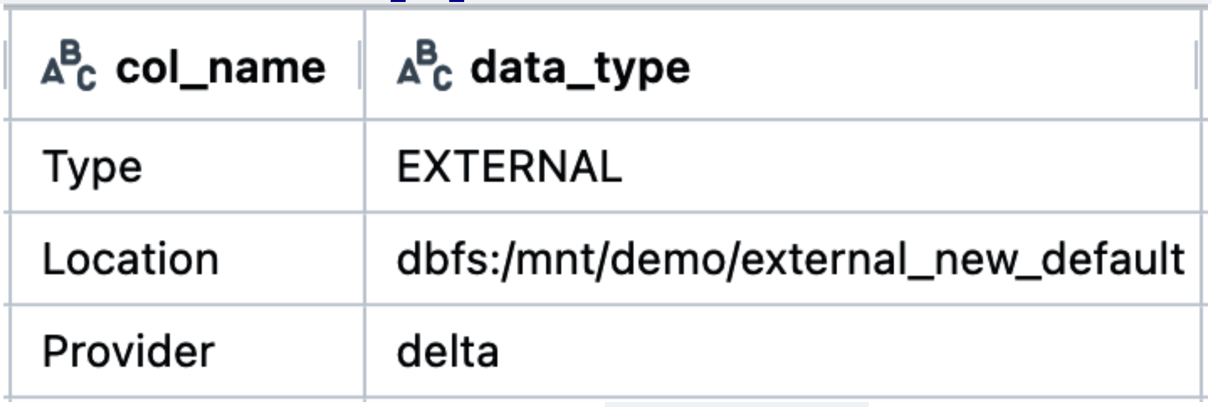
```
DESCRIBE EXTENDED managed_new_default
```



col_name	data_type
Type	MANAGED
Location	dbfs:/user/hive/warehouse/new_default.db/managed_new_default
Provider	delta

Figure 3-15. Metadata of the `managed_new_default` table

```
DESCRIBE EXTENDED external_new_default
```



col_name	data_type
Type	EXTERNAL
Location	dbfs:/mnt/demo/external_new_default
Provider	delta

Figure 3-16. Metadata of the `external_new_default` table

Dropping tables

Let’s proceed to drop the newly created tables:

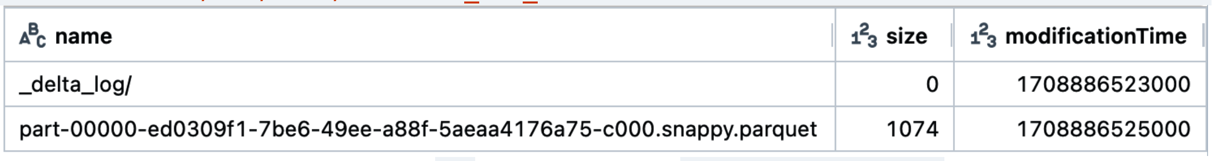
```
DROP TABLE managed_new_default;
DROP TABLE external_new_default;
```

Dropping the tables removes their entries from the Hive metastore. You can easily confirm this in the Catalog Explorer. Moreover, this action on the managed table results in the removal of its directory and associated data files from the storage:

```
%fs ls 'dbfs:/user/hive/warehouse/new_default.db/managed_new_default'
FileNotFoundException: No such file or directory
dbfs:/user/hive/warehouse/new_default.db/managed_new_default
```

However, as expected, in the case of the external table, although the table itself is dropped from the database, the directory and its data files persist in the specified external location (Figure 3-17).

```
%fs ls 'dbfs:/mnt/demo/external_new_default'
```



name	size	modificationTime
_delta_log/	0	1708886523000
part-00000-ed0309f1-7be6-49ee-a88f-5aeaa4176a75-c000.snappy.parquet	1074	1708886525000

Figure 3-17. The output of the `%fs` command on the `external_new_default` table directory

Working In a Custom-Location Schema

In our last scenario, we will create a database in a custom location outside of the default Hive directory.

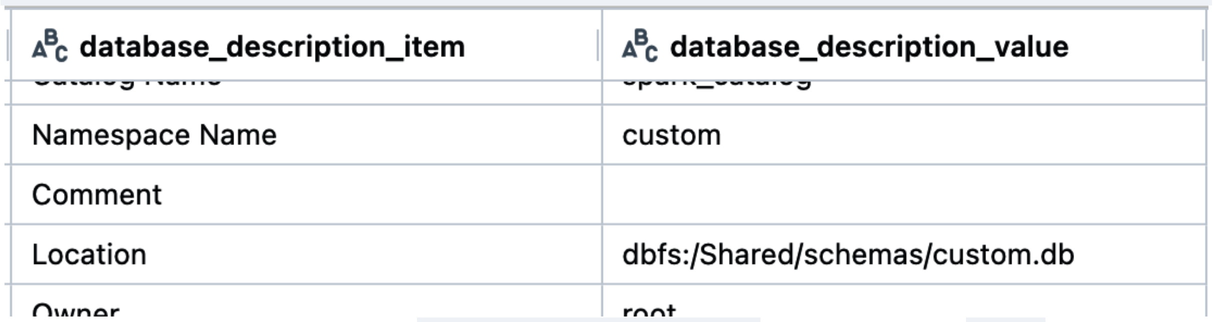
Creating the database

To achieve this, we begin by using the `CREATE SCHEMA` statement, and we add the `LOCATION` keyword followed by the desired storage path, in our case *dbfs:/Shared/schemas*:

```
CREATE SCHEMA custom
LOCATION 'dbfs:/Shared/schemas/custom.db'
```

You can inspect the Catalog Explorer to confirm that the database has been created within the Hive metastore. Upon closer examination, using the `DESCRIBE DATABASE EXTENDED` command, we confirm that the database was situated in the custom location we specified during its creation (Figure 3-18):

```
DESCRIBE DATABASE EXTENDED custom
```



database_description_item	database_description_value
Namespace Name	custom
Comment	
Location	dbfs:/Shared/schemas/custom.db
Owner	root

Figure 3-18. The output of the `DESCRIBE DATABASE EXTENDED` command on the `custom` schema

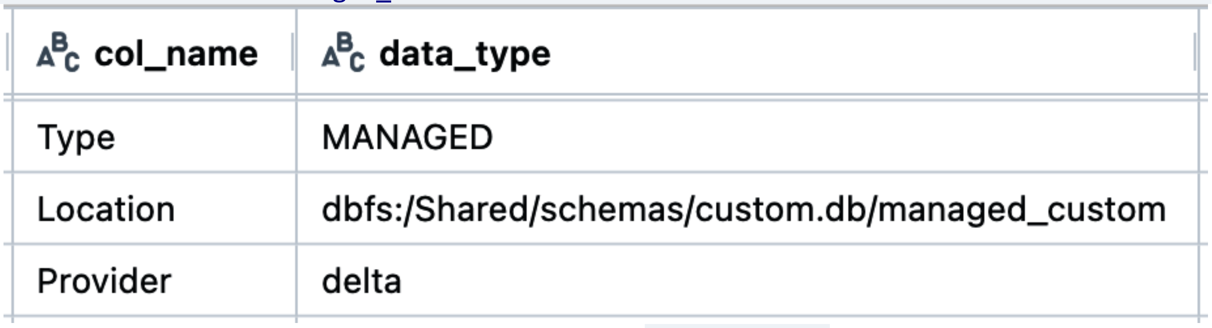
Creating tables

We proceed to use this database to create tables and populate them with data. Again, we create both managed and external tables:

```
USE DATABASE custom;

-- Create a managed table
CREATE TABLE managed_custom
 (country STRING, code STRING, dial_code STRING);

INSERT INTO managed_custom
VALUES ('France', 'Fr', '+33');
-----------------------------------
-- Create an external table
CREATE TABLE external_custom
 (country STRING, code STRING, dial_code STRING)
LOCATION 'dbfs:/mnt/demo/external_custom';

INSERT INTO external_custom
VALUES ('France', 'Fr', '+33');
```

You can inspect the Catalog Explorer to confirm that the two tables have been successfully created within our new database. In addition, by running `DESCRIBE EXTENDED` on each of these tables, we can confirm that the managed_custom table is indeed a managed table, since it is created in its database folder located in the custom location (Figure 3-19). Meanwhile, the external_custom table is an external table because its location was specified during table creation (Figure 3-20).

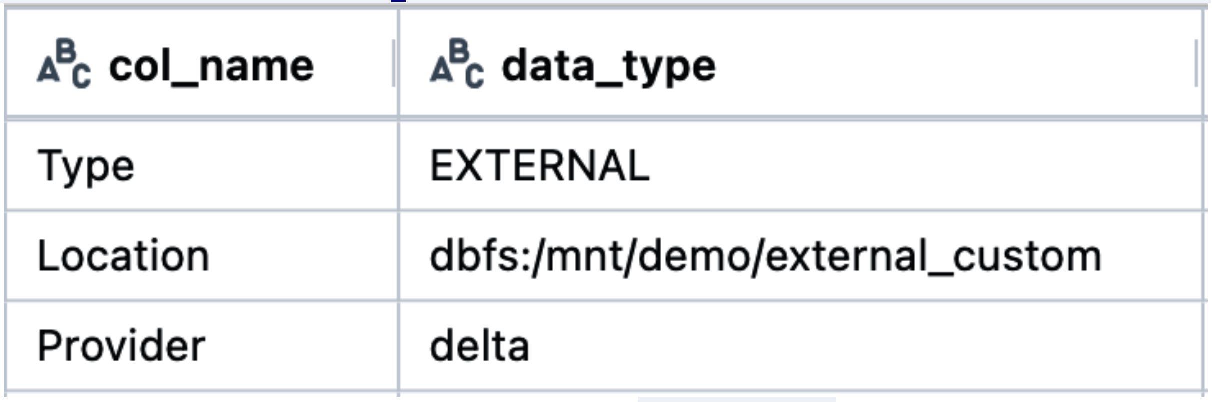
```
DESCRIBE EXTENDED managed_custom
```



col_name	data_type
Type	MANAGED
Location	dbfs:/Shared/schemas/custom.db/managed_custom
Provider	delta

Figure 3-19. Metadata of the managed_custom table

```
DESCRIBE EXTENDED external_custom
```



col_name	data_type
Type	EXTERNAL
Location	dbfs:/mnt/demo/external_custom
Provider	delta

Figure 3-20. Metadata of the external_custom table

Dropping tables

Let's proceed to drop the newly created tables:

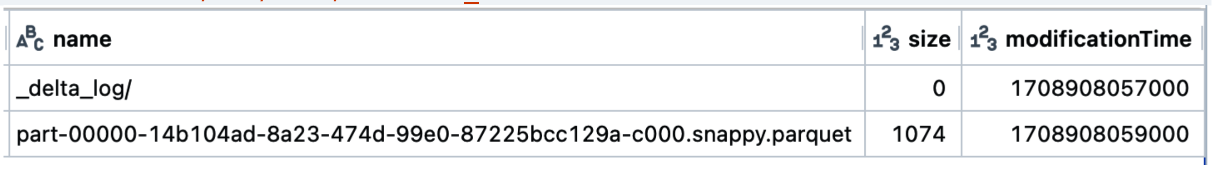
```
DROP TABLE managed_custom;
DROP TABLE external_custom;
```

Once more, dropping the tables removes both of their entries from the Hive metastore. You can easily confirm this in the Catalog Explorer. Dropping the managed table still removes its directory and associated data files from the database directory located in the custom location:

```
%fs ls 'dbfs:/Shared/schemas/custom.db/managed_custom'
FileNotFoundException:
No such file or directory dbfs:/Shared/schemas/custom.db/managed_custom
```

However, as expected, in the case of an external table, the table's directory and data files remain intact in their external location (Figure 3-21).

```
%fs ls 'dbfs:/mnt/demo/external_custom'
```



name	size	modificationTime
_delta_log/	0	1708908057000
part-00000-14b104ad-8a23-474d-99e0-87225bcc129a-c000.snappy.parquet	1074	1708908059000

Figure 3-21. The output of the %fs command on the external_custom table directory

Remember, you can manually remove the table directory and its content by running the dbutils.fs.rm function in Python.

Setting Up Delta Tables

We've explored the dynamics of managed and external tables, illustrating how they interact within the context of a different type of databases. With this understanding, we're equipped to dive into more advanced topics on Delta Lake tables in the following sections.

CTAS Statements

One of the key features of Delta Lake tables is their flexibility in creation. While traditional methods like the regular CREATE TABLE statements are available, Databricks also supports CTAS, or CREATE TABLE AS SELECT, statements. CTAS statements allow the creation and population of tables at the same time based on the results of a SELECT query. This means that with CTAS statements, you can create a new table from existing data sources:

```
CREATE TABLE table_2
AS SELECT * FROM table_1
```

This simple yet powerful syntax shows how CTAS statements work. In this example, we're creating table_2 by selecting all data from table_1. CTAS statements automatically infer schema information from the query results, eliminating the need for manual schema declaration.

CTAS statements in Databricks offer a convenient means to perform transformations on data during the creation of Delta tables. These transformations can include tasks such as renaming columns or selecting specific columns for inclusion in the target table. Let's illustrate this with an abstract example:

```
CREATE TABLE table_2
AS SELECT col_1, col_3 AS new_col_3 FROM table_1
```

In this example, the CTAS statement generates a new table named table_2, by selecting columns col_1 and col_3 from table_1. Additionally, the col_3 is renamed to new_col_3 in the resulting table.

Moreover, a range of options can be added to the CREATE TABLE clause to customize table creation, allowing for precise control over table properties and storage configurations.

```
CREATE TABLE new_users
 COMMENT "Contains PII"
 PARTITIONED BY (city, birth_date)
 LOCATION '/some/path'
 AS SELECT id, name, email, birth_date, city FROM users
```

In the provided example, we illustrate several of these options:

Comment

The `COMMENT` clause enables you to provide a descriptive comment for the table, helping in the discovery and understanding of its contents. Here, we've added a comment indicating that the table contains personally identifiable information (PII), such as the user's name and email.

Partitioning

The underlying data of the table can be partitioned into subfolders. The `PARTITIONED BY` clause allows for data partitioning based on one or more columns. In this case, we're partitioning the table by `city` and `birth_date`.

Partitioning can significantly enhance the performance of large Delta tables by facilitating efficient data retrieval. However, it's important to note that for small to medium-sized tables, the benefits of partition may be negligible or outweighed by drawbacks. One significant drawback is the potential emergence of what is known as the "small files problem." This problem arises when data partitioning results in the creation of numerous small files, each containing a relatively small amount of data.

While partitioning aims to improve query performance by reducing the amount of data scanned, the presence of many small files can prevent file compaction and efficiency in data skipping. In general, partitioning should be selectively applied based on the size and nature of the data.

External location

The location option enables the creation of external tables. Remember, the `LOCATION` keyword allows you to specify the storage location for the created table. This means that the data associated with the table will be stored in an external location specified by the provided path.

Comparing CREATE TABLE and CTAS

Table 3-2 summarizes the differences between regular `CREATE TABLE` statements and CTAS (`CREATE TABLE AS SELECT`) statements.

	`CREATE TABLE` **statement**	**CTAS statement**

	`CREATE TABLE table_2`
	`(col1 INT, col2 STRING, col3`
	`DOUBLE)`
		`CREATE TABLE table_2`
		`AS SELECT col1, col2, col3`
		`FROM table_1`

	`CREATE TABLE` **statement**	**CTAS statement**
Schema declaration	Requires manual schema declaration.	Does not allow manual schema declaration. It automatically infers the table schema.
Populating data	Creates an empty table; a data loading statement, such as `INSERT INTO`, is required to populate it.	The table is created with data as specified.

Table 3-2. Comparison of `CREATE TABLE` and CTAS statements

Let's dive deeper to gain a comprehensive understanding of these differences.

Schema declaration

Regular `CREATE TABLE` statements require manual schema declaration. For instance, you would explicitly specify the data types for each column, such as `integer` for column 1, `string` for column 2, and `double` for column 3. By contrast, CTAS statements automate schema declaration by inferring schema information directly from the results of the query.

Populating data

When using regular `CREATE TABLE` statements, an empty table is created, requiring an additional step to load data into it, such as using the `INSERT INTO` statement. By contrast, CTAS statements simplify this process by simultaneously creating the table and populating it with data from the output of the `SELECT` statement. In the upcoming module, we'll see CTAS statements in action, observing how they offer a more efficient and straightforward approach to table creation and data population compared to traditional `CREATE TABLE` statements.

Table Constraints

After creating a Delta Lake table, whether through a regular `CREATE TABLE` statement or a CTAS statement, you have the option to enhance its integrity by adding constraints. Databricks currently supports two types of table constraints:

- `NOT NULL` constraints
- `CHECK` constraints

```
ALTER TABLE table_name ADD CONSTRAINT <constraint_name> <constraint_detail>
```

When applying constraints to a Delta table, it's crucial to ensure that existing data in the table adheres to these constraints before defining them; otherwise, the statement will fail. Once a constraint is enforced, any new data that violates the constraint will result in a write failure.

For instance, let's consider the addition of a `CHECK` constraint to the `date` column of a Delta table. `CHECK` constraints resemble standard `WHERE` clauses used to filter datasets. They define conditions that incoming data must satisfy in order to be accepted into the table. For instance, suppose we want to ensure that dates in the `date` column fall within a specific range. We can add a `CHECK` constraint to enforce this condition:

```
ALTER TABLE my_table
ADD CONSTRAINT valid_date CHECK (date >= '2024-01-01' AND date <= '2024-12-31');
```

In this example, `valid_date` is the name of our constraint, and the condition ensures that the `date` column values fall within the specified range for the year 2024. Any attempt to insert or update data with dates outside this range will be rejected. This helps maintain data consistency and integrity within the Delta Lake table.

Cloning Delta Lake Tables

In Databricks, if you need to back up or duplicate your Delta Lake table, you have two efficient options: deep clone and shallow clone.

Deep cloning

Deep cloning involves copying both data and metadata from a source table to a target. Here's an example of how you might use the command:

```
CREATE TABLE table_clone
DEEP CLONE source_table
```

Simply, use the `CREATE TABLE` statement, specify the name of the new target table, and include the `DEEP CLONE` keyword followed by the name of the source table.
This copy process can occur incrementally, allowing you to synchronize changes from the source to the target location. Simply, execute `CREATE OR REPLACE TABLE` instead in order to create a new table version with the new changes:

```
CREATE OR REPLACE TABLE table_clone
DEEP CLONE source_table
```

It's important to note that because in deep cloning all the data must be copied over, this process may take quite a while, especially for large source tables.

Shallow cloning

On the other hand, the shallow clone provides a quicker way to create a copy of a table. It only copies the Delta transaction logs, meaning no data movement takes place during shallow cloning:

```
CREATE TABLE table_clone
SHALLOW CLONE source_table
```

Shallow cloning is an ideal option for scenarios where, for example, you need to test applying changes on a table without altering the current table's data. This makes it particularly useful in development environments where rapid iteration and experimentation are common.

Data integrity in cloning

Whether you choose deep cloning or shallow cloning, any modifications made to the cloned version of the table will be tracked and stored separately from the source. This ensures that changes made during testing or experimentation do not affect the integrity of the original source table.

Exploring Views

In Databricks, views serve as virtual tables without physical data. A view is nothing but a saved SQL query against actual tables, where this logical query is executed each time the view is queried.

Figure 3-22 illustrates an abstract example of creating a view on top of two tables by performing an inner join between them. Each time the view is queried, the join operation will be executed again against these tables.

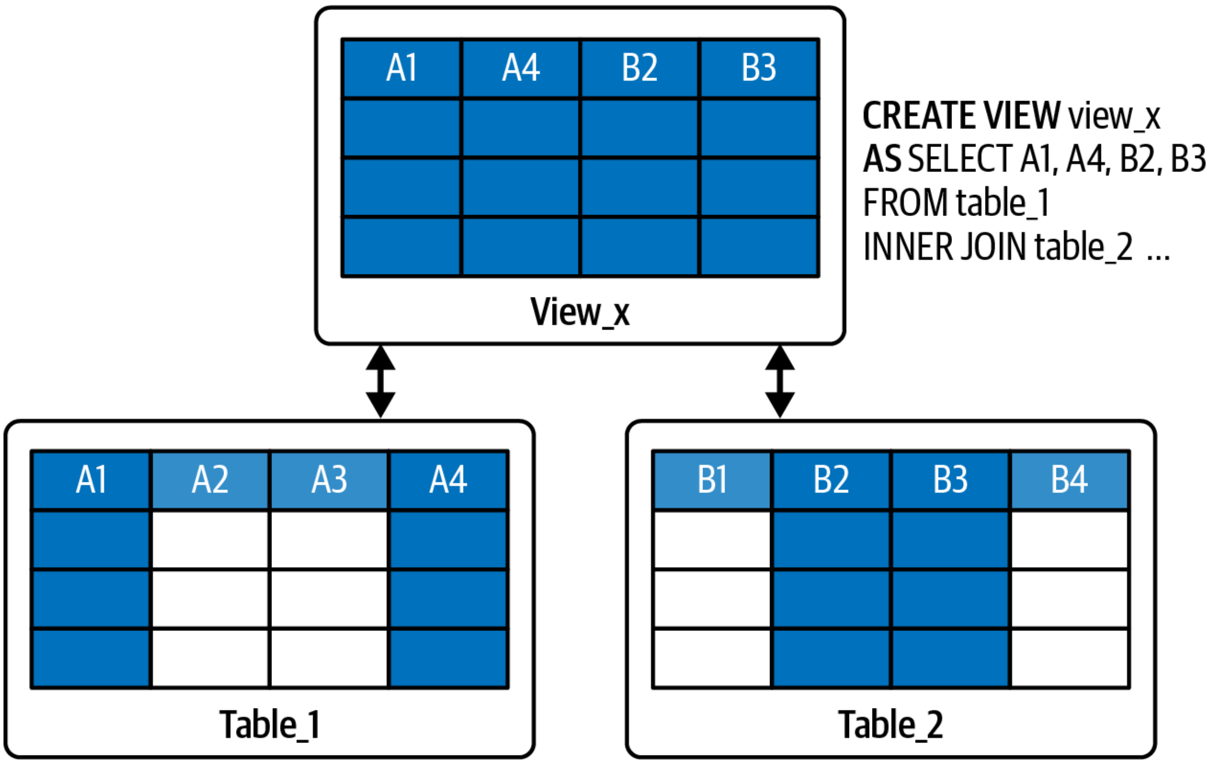


Figure 3-22. A view object on top of two tables

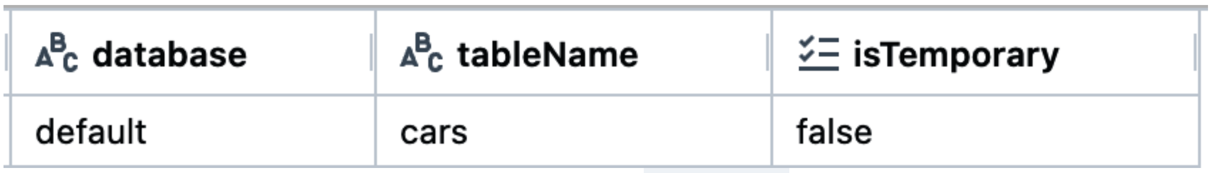
To demonstrate how views function within Databricks, we will use a new SQL notebook titled “3.2A - Views.” We start by creating a table of data to be used in this demonstration, called `cars`. This table contains columns for the ID, model, brand, and release year of the cars.

```
USE CATALOG hive_metastore;

CREATE TABLE IF NOT EXISTS cars
(id INT, model STRING, brand STRING, year INT);

INSERT INTO cars
VALUES (1, 'Cybertruck', 'Tesla', 2024),
      (2, 'Model S', 'Tesla', 2023),
      (3, 'Model Y', 'Tesla', 2022),
      (4, 'Model X 75D', 'Tesla', 2017),
      (5, 'G-Class G63', 'Mercedes-Benz', 2024),
      (6, 'E-Class E200', 'Mercedes-Benz', 2023),
      (7, 'C-Class C300', 'Mercedes-Benz', 2016),
      (8, 'Everest', 'Ford', 2023),
      (9, 'Puma', 'Ford', 2021),
      (10, 'Focus', 'Ford', 2019)
```

After creating the table and inserting some data into it, you can verify its creation in the Catalog Explorer. Additionally, we can use the `SHOW TABLES` command to list all tables and views in the default database:

```
SHOW TABLES
```

Figure 3-23 displays the output of the `SHOW TABLES` command. As observed, we have a table named `cars` in the default database.



database	tableName	isTemporary
default	cars	false

Figure 3-23. The output of the `SHOW TABLES` command

View Types

There are three types of views available in Databricks: stored views, temporary views, and global temporary views. Let's explore these different types of views and how they function within the platform.

Stored views

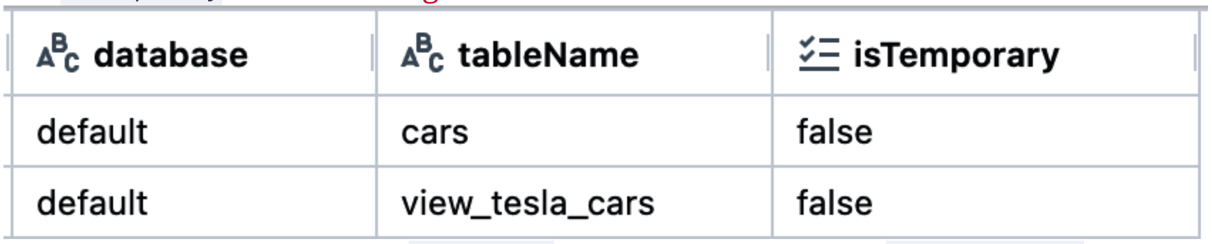
Stored views, often referred to simply as *views*, are similar to traditional database views. They are database objects where their metadata is persisted in the database. To create a stored view, we use the `CREATE VIEW` statement followed by the `AS` keyword and the logical SQL query defining the view:

```
CREATE VIEW view_name
AS <query>
```

Let's create a stored view that displays only Tesla cars from our `cars` table. We use the `CREATE VIEW` statement, naming our view `view_tesla_cars`, and specify the logical query following the `AS` keyword. This query selects all records from the `cars` table where the brand is equal to `Tesla`:

```
CREATE VIEW view_tesla_cars
AS SELECT *
    FROM cars
    WHERE brand = 'Tesla';
```

Running the `SHOW TABLES` command again confirms that the view has been persisted in the default database and it is not a temporary object, as shown in the `isTemporary` column in Figure 3-24.



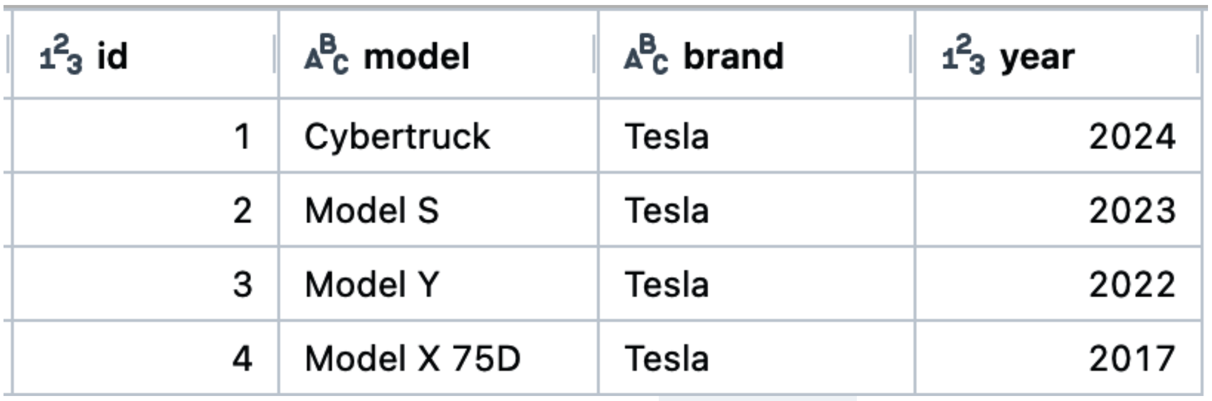
database	tableName	isTemporary
default	cars	false
default	view_tesla_cars	false

Figure 3-24. The output of the `SHOW TABLES` command after creating the `view_tesla_cars` view

Once created, you can query the stored view using a standard `SELECT` statement, treating it as if it were a table object:

```
SELECT * FROM view_tesla_cars;
```

Figure 3-25 displays the result of querying this view.



id	model	brand	year
1	Cybertruck	Tesla	2024
2	Model S	Tesla	2023
3	Model Y	Tesla	2022
4	Model X 75D	Tesla	2017

Figure 3-25. The result of querying the `view_tesla_cars` stored view

It’s worth noting that this result is retrieved directly from the `cars` table. Remember, each time the view is queried, its underlying logical query is actually executed against the source table, in this case, the `cars` table.

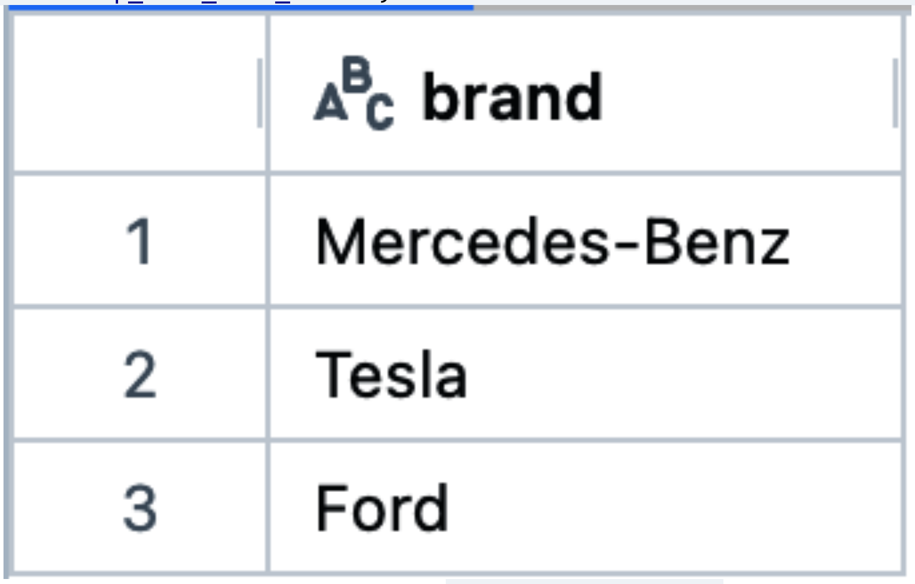
Temporary views

The second type of views in Databricks is temporary views. Temporary views are bound to the Spark session and are automatically dropped when the session ends. They are handy for temporary data manipulations or analyses. To create a temporary view, you simply add the `TEMPORARY`, or `TEMP`, keyword to the `CREATE VIEW` command:

```
CREATE TEMP VIEW view_name
AS <query>
```

Let’s create a temporary view called `temp_view_cars_brands`. This temporary view simply retrieves the unique list of brands from our `cars` table (Figure 3-26):

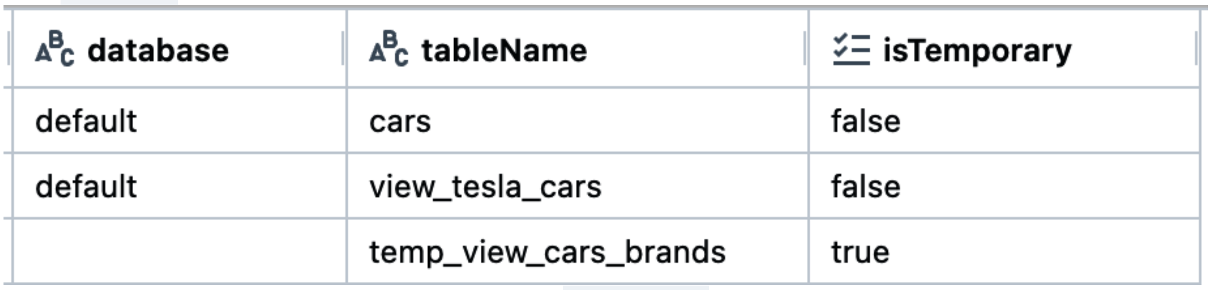
```
CREATE TEMP VIEW temp_view_cars_brands
AS  SELECT DISTINCT brand
    FROM cars;

SELECT * FROM temp_view_cars_brands;
```



	brand
1	Mercedes-Benz
2	Tesla
3	Ford

Figure 3-26. The result of querying the `temp_view_cars_brands` temporary view

Running the `SHOW TABLES` command confirms the addition of the temporary view to the list, as illustrated in Figure 3-27. The `isTemporary` column indicates its temporary nature. In addition, since it's a temporary object, it is not persisted to any database, as indicated by having no database specified in the `database` column.



database	tableName	isTemporary
default	cars	false
default	view_tesla_cars	false
	temp_view_cars_brands	true

Figure 3-27. The output of the `SHOW TABLES` command after creating the `temp_view_cars_brands` temporary view

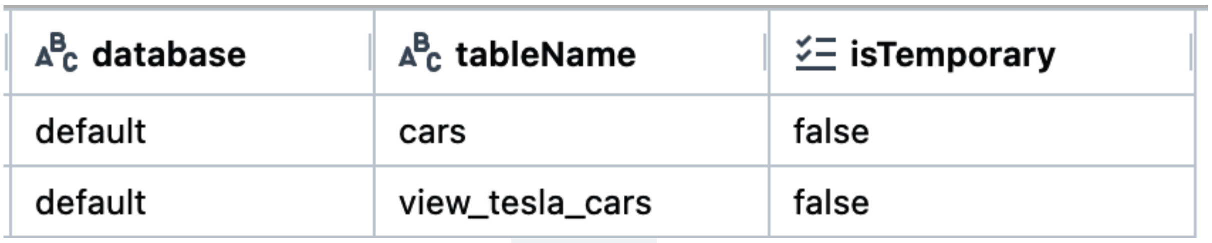
The lifespan of a temporary view is limited to the duration of the current Spark session. It's essential to note that a new Spark session is initiated in various scenarios within Databricks, such as the following:

- Opening a new notebook
- Detaching and reattaching a notebook to a cluster
- Restarting the Python interpreter due to a Python package installation
- Restarting the cluster itself

To confirm this, let's create a new notebook called "3.2B - Views (Session 2)," and observe the behavior of our created views within it. In this new Spark session, let's first run the `SHOW TABLES` command:

```
USE CATALOG hive_metastore;

SHOW TABLES;
```

Figure 3-28 displays the output of the `SHOW TABLES` command in the newly created Spark session. This result confirms the existence of the `cars` table, as expected. In addition, the stored view of Tesla cars also exists in this new notebook. However, the temporary view of the car brands does not exist in this new session.



database	tableName	isTemporary
default	cars	false
default	view_tesla_cars	false

Figure 3-28. The output of the `SHOW TABLES` command in a new Spark session

Global temporary views

Global temporary views behave similarly to other temporary views but are tied to the cluster instead of a specific session. This means that as long as the cluster is running, any notebook attached to it can access its global temporary views. To define a global temporary view, you add the `GLOBAL TEMP` keyword to the `CREATE VIEW` command:

```
CREATE GLOBAL TEMP VIEW view_name
AS <query>
```

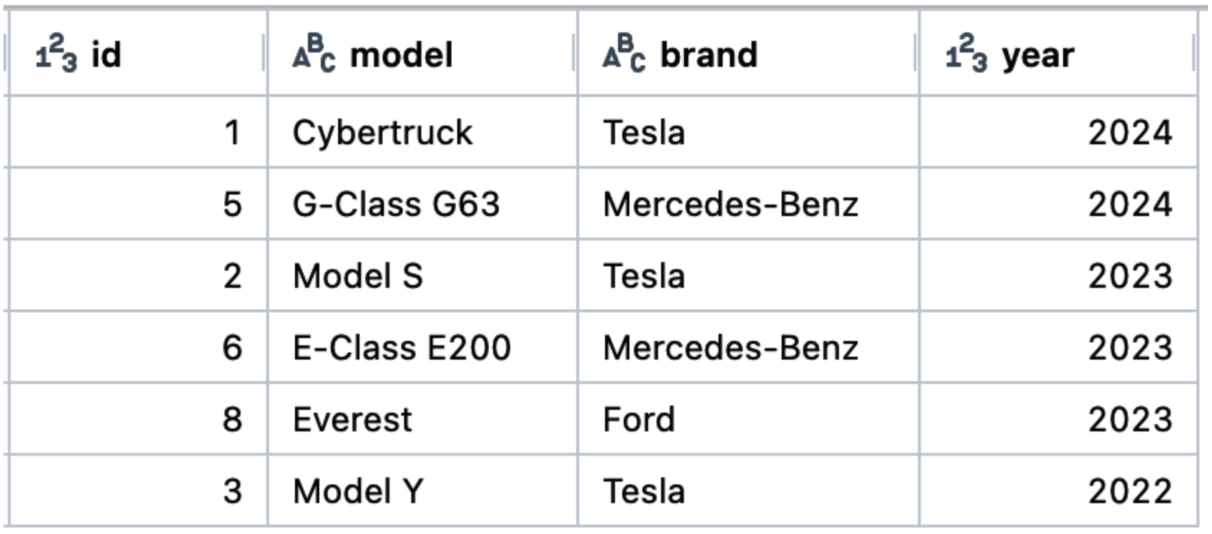
In our original “3.2A - Views” notebook, let’s create a global temporary view, called `global_temp_view_recent_cars`. This view retrieves all cars from our `cars` table released in 2022 or later, ordered in descending order:

```
CREATE GLOBAL TEMP VIEW global_temp_view_recent_cars
AS SELECT * FROM cars
    WHERE year >= 2022
    ORDER BY year DESC;
```

Global temporary views are stored in a cluster’s temporary database, named `global_temp`. When querying a global temporary view in a `SELECT` statement, you need to specify the `global_temp` database qualifier:

```
SELECT * FROM global_temp.global_temp_view_recent_cars;
```

Figure 3-29 displays the result of querying our global temporary view, showing the latest entries from the `cars` table.



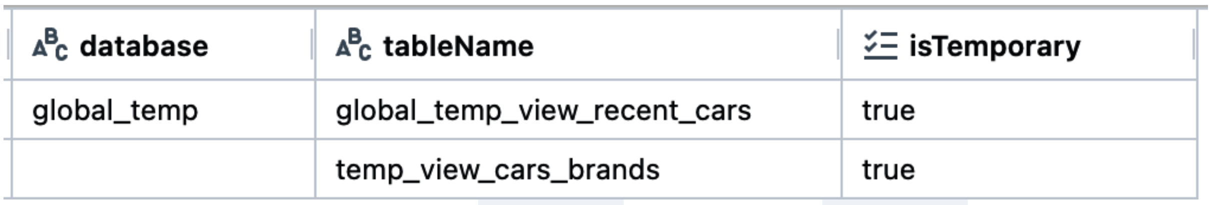
id	model	brand	year
1	Cybertruck	Tesla	2024
5	G-Class G63	Mercedes-Benz	2024
2	Model S	Tesla	2023
6	E-Class E200	Mercedes-Benz	2023
8	Everest	Ford	2023
3	Model Y	Tesla	2022

Figure 3-29. The result of querying the global temporary view

If you run the `SHOW TABLES` command, you will notice that our global temporary view is not listed among other objects. This occurs because, by default, the command only displays objects in the `default` database. Since the global temporary views are tied to the `global_temp` database, we need to use the command `SHOW TABLES IN`, explicitly specifying the database name `global_temp`:

```
SHOW TABLES IN global_temp;
```

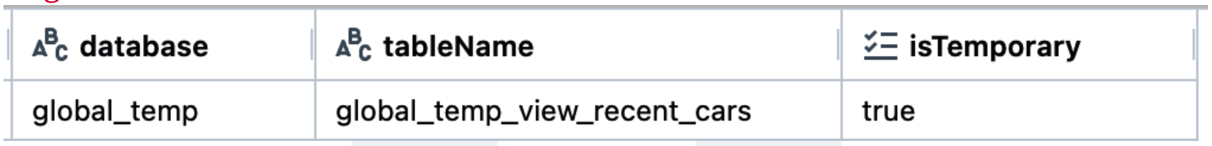
In Figure 3-30, we can see the `global_temp_view_recent_cars`, which is indeed a temporary object tied to the `global_temp` database. Since our `temp_view_cars_brands` is not tied to any database, it’s typically shown with every `SHOW TABLES` command.



database	tableName	isTemporary
global_temp	global_temp_view_recent_cars	true
	temp_view_cars_brands	true

Figure 3-30. The output of the `SHOW TABLES` command in the `global_temp` database

Now, let's switch back to the second notebook, "3.2B - Views (Session 2)." In this new Spark session, we can explore the objects in the `global_temp` database (Figure 3-31).



database	tableName	isTemporary
global_temp	global_temp_view_recent_cars	true

Figure 3-31. The output of the `SHOW TABLES` command in the `global_temp` database within the new Spark session

Since we are leveraging the same cluster, our global temporary view also exists in this new session. As long as the cluster is running, the `global_temp` database persists, and any notebook attached to the cluster can access its global temporary views. You can confirm this by querying the global temporary view to see the recent cars in this new session.

Comparison of View Types

Understanding the distinctions between the view types and their lifecycles is essential for effective data manipulation and collaboration within your Spark environment. Table 3-3 summarizes the differences between these three types of views.

	(Stored) view	**Temporary view**	**Global temporary view**
Creation syntax	`CREATE VIEW`	`CREATE TEMP VIEW`	`CREATE GLOBAL TEMP VIEW`
Accessibility	Accessed across sessions/clusters	Session-scoped	Cluster-scoped
Lifetime	Dropped only by `DROP VIEW` statement	Dropped when session ends	Dropped when cluster restarted or terminated

Table 3-3. Comparison of view types

Creation syntax

There's a slight difference in the `CREATE VIEW` statements for temporary and global temporary views. For temporary views, we include the `TEMP` keyword, whereas for global temporary views, we add the `GLOBAL TEMP` keyword.

Accessibility

Stored views are similar to tables in that their definitions are stored in the metastore, but they don't contain a physical copy of the data they reference. Remember, a view essentially represents a SQL query. Since stored views are saved in the metastore, they can be accessed across multiple sessions and clusters.

Temporary views, in contrast, are accessible only within the current session. Global temporary views bridge the gap between stored and temporary views; they can be accessed across multiple sessions but are tied to the same cluster.

Lifetime

Lastly, when it comes to removing these views, different methods apply. Stored views are dropped using the `DROP VIEW` command, while temporary views are automatically dropped when the session ends. Similarly, global temporary views are automatically dropped, but this occurs when the cluster is restarted or terminated.

Dropping Views

Let's finally drop our stored view by running the `DROP VIEW` command, like in standard SQL:

```
DROP VIEW view_tesla_cars;
```

If you want to delete temporary views without waiting for the session to end or for the cluster to terminate, you can manually achieve this by using the `DROP VIEW` command as well:

```
DROP VIEW temp_view_cars_brands;
DROP VIEW global_temp.global_temp_view_recent_cars;
```

This allows you to manually clean up such resources when they are no longer needed.

Thus, views in Databricks serve as a powerful solution for organizing and manipulating data without the need to duplicate it physically. With three types of views, Databricks offers a variety of options to suit different use cases and requirements.

Conclusion

In conclusion, mastering relational entities such as databases, tables, and views is fundamental to effectively organizing and managing structured data in Databricks. By understanding their interactions with the metastore and storage locations, you can enhance your data querying and management efficiency. This chapter has provided a comprehensive overview of these entities, setting the stage for further exploration of advanced data management techniques within the Databricks environment.

Sample Exam Questions

Conceptual Question

1. A data engineer is tasked with cleaning up unused Delta tables from a production data catalog. When they drop a Delta table, they notice that this action not only removes the table entry from the catalog but also deletes the underlying data files. Which of the following best explains this behavior?

1. The Delta table was created using a deep clone, which causes both the source table and its data files to be removed when the cloned table is dropped.
2. The Delta table was created using a shallow clone, and shallow clones automatically delete the source table’s data files when dropped.
3. The Delta table was created using an external location, so dropping it removes all associated data files.
4. The Delta table was registered as a managed table, and by default, managed tables delete both the metadata and data files when dropped.
5. The Delta table was defined as a stored view, and dropping a stored view automatically deletes the stored data files associated with that view.

Code-Based Question

2. A data engineer at a growing e-commerce company is tasked with creating an external Delta Lake table to store customer information. The table needs to be located in the directory *dbfs:/ecommerce/customers*.
Which of the following SQL statements correctly creates the external Delta Lake table?

1. `CREATE TABLE customers`

```
    (id INT, name STRING, email STRING)
    EXTERNAL 'dbfs:/ecommerce/customers';
```

2.
   ```
   CREATE TABLE customers
       USING DELTA
       (id INT, name STRING, email STRING)
       AS EXTERNAL ('dbfs:/ecommerce/customers');
   ```

3.
   ```
   CREATE TABLE customers
       (id INT, name STRING, email STRING)
       LOCATION 'dbfs:/ecommerce/customers';
   ```

4.
   ```
   CREATE TABLE customers
       USING DELTA
       (id INT, name STRING, email STRING)
       LOCATION AS OF 'dbfs:/ecommerce/customers';
   ```

5.
   ```
   CREATE EXTERNAL TABLE customers
       (id INT, name STRING, email STRING)
       PATH = 'dbfs:/ecommerce/customers';
   ```

The correct answers to these questions are listed in Appendix C.