O'REILLY®

Databricks Certified Data Engineer Associate Study Guide

In-Depth Guidance and Practice

Early Release

RAW & UNEDITED

Derar Alhussein

Databricks Certified Data Engineer Associate Study Guide

In-Depth Guidance and Practice

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Derar Alhussein

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Databricks Certified Data Engineer Associate Study Guide
by Derar Alhussein

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Shira Evans and Aaron Black
Production Editor: Aleeya Rahman
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

February 2025: First Edition

Revision History for the Early Release
2024-04-24: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098166830* for release details.

978-1-098-16683-0

Table of Contents

1. Managing Data with Delta Lake 5
Introducing Delta Lake 5
What is Delta Lake? 6
Delta Lake Transaction Log 7
Understanding Delta Lake Functionality 8
Delta Lake Advantages 13
Working with Delta Lake Tables 13
Creating Tables 14
Catalog Explorer 14
Inserting Data 15
Exploring Table Directory 16
Exploring Table History 18
Exploring Delta Time Travel 20
Querying Older versions 21
Rollbacking Back to Previous Versions 22
Optimizing Delta Lake Tables 24
Z-Order Indexing 25
Vacuuming 29
Vacuuming in Action 29
Dropping Delta Lake Tables 31

2. Mastering Relational Entities in Databricks 33
Understanding Relational Entities 33
Databases in Databricks 33
Tables in Databricks 36
Putting Relational Entities Into Practice 39
Working in the default Schema 40
Working In a New Schema 43

Working In a Custom-Location Schema 47
Setting Up Delta Tables 49
CTAS statements 49
Comparing CREATE TABLE vs. CTAS 51
Table Constraints 52
Cloning Delta Lake Tables 52
Exploring Views 53
View Types 55
Comparison of View Types 60

CHAPTER 1

Managing Data with Delta Lake

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at sevans@gmail.com.

Data Lakehouses leverage specialized storage frameworks to enhance the functionality of traditional data lakes. Among these frameworks, Delta Lake stands out as a leading technology that empowers the Databricks Lakehouse Platform. In this chapter, we'll explore the fundamental concepts of Delta Lake by first introducing its core principles and then diving into its practical usage. Following this, we'll focus on advanced topics in Delta Lake such as Time Travel, tables optimization, and vacuum operation.

Introducing Delta Lake

Traditional data lakes often suffer from inefficiencies and encounter various challenges in processing big data. Delta Lake technology is an innovative solution designed to operate on top of data lakes to overcome these issues. To establish a clear understanding of Delta Lake, let's first study its definition as provided by its original creators at Databricks.

What is Delta Lake?

> *"Delta Lake is an open-source storage layer that brings reliability to data lakes by adding a transactional storage layer on top of data stored in cloud storage"*
>
> —Databricks

In the context of data lakehouses, a storage layer refers to the framework responsible for managing and organizing data stored within the data lake. It serves an intermediary platform through which data is ingested, queried, and processed.

In other words, Delta Lake is not a storage medium or storage format. Common storage formats like Parquet, or JSON define how data is physically stored in the lake. However, Delta Lake runs on top of such data formats to provide a robust solution that overcomes the challenges of data lakes.

While data lakes are excellent solutions for storing massive volumes of diverse data, they often encounter several challenges related to data inconsistency, and performance issues. The primary factor behind these limitations is the absence of ACID transactions support in the lake. ACID, an acronym for Atomicity, Consistency, Isolation, and Durability, represents fundamental rules that ensure operations on the data are reliably executed. This absence made it difficult to ensure data integrity, leading to issues like partially committed data or failed transactions.

What makes Delta Lake an innovative solution is its ability to overcome such challenges posed by traditional data lakes. Delta Lake provides ACID transaction guarantees for data manipulation operations in the lake. It offers transactional capabilities that enable performing data operations in an atomic and consistent manner. This ensures that there is no partially committed data; either all operations within a transaction are completed successfully, or none of them are. These capabilities allow you to build reliable data lakes that ensure data integrity, consistency, and durability.

Delta Lake is optimized for cloud object storage. It seamlessly integrates with leading cloud storage platforms such as Amazon S3, Azure Data Lake Storage, and Google Cloud Storage.

On top of all this, Delta Lake is an open-source library. Unlike proprietary solutions, Delta Lake's source code is freely available to you on Github at https://github.com/delta-io/delta

To put all together, we can visualize the concepts discussed above through an illustrative graph. In Figure 1-1, we highlight the key elements that constitute the Delta Lake technology.

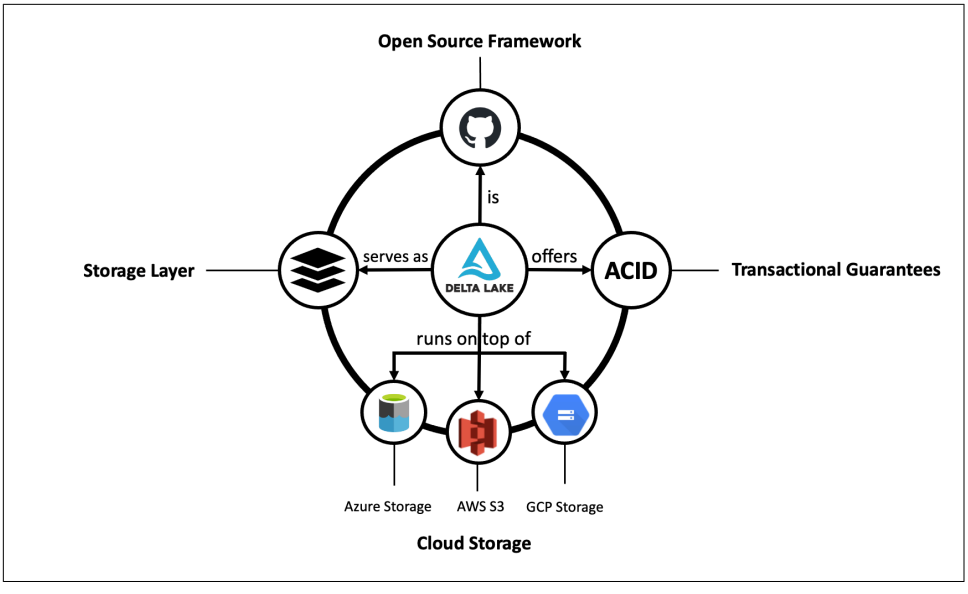


Figure 1-1. 1 Illustration of Delta Lake technology

Delta Lake Transaction Log

The Delta Lake library is deployed on the cluster as part of the Databricks runtime. When you create a Delta Lake table within this ecosystem, the data is stored on the cloud storage in one or more data files in Parquet format. However, alongside these data files, Delta Lake creates a transaction log in JSON format, as illustrated in Figure 1-2.

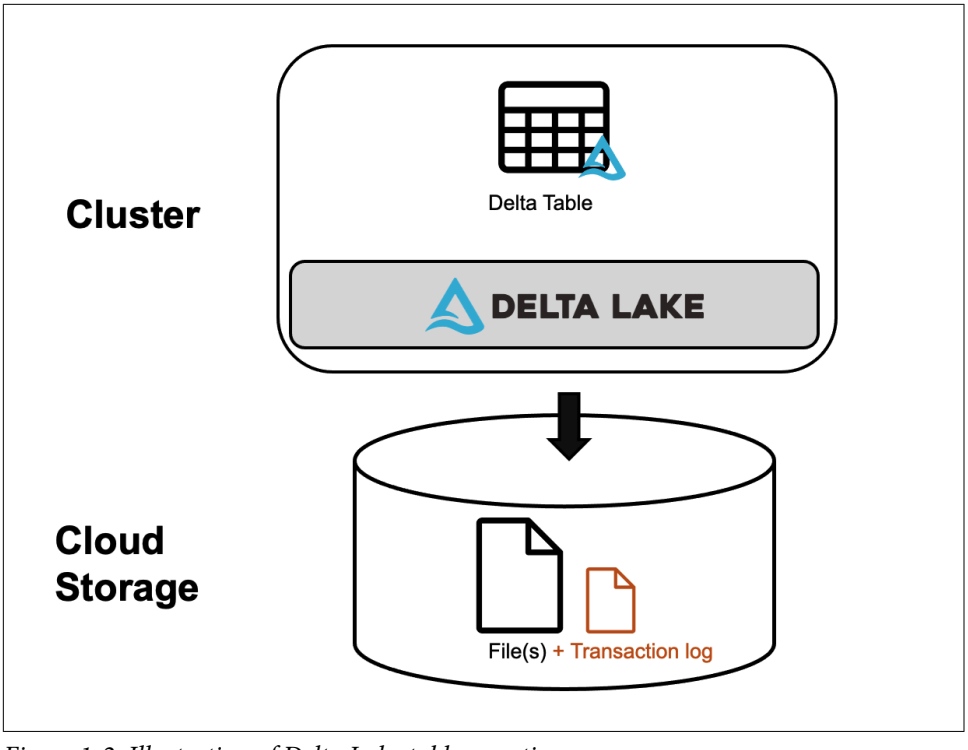


Figure 1-2. Illustration of Delta Lake tables creation

The Delta Lake transaction log, often referred to as Delta Log, is an ordered record of every transaction performed on the table since its creation. As a result, it functions as the source of truth for the table's state and history. So every time you query the table, Spark checks this transaction log to determine the most recent version of the data.

Each committed transaction is recorded in a JSON file. This file contains essential details about the operations performed, such as its type (insert, update, …, etc) and any predicate used during these operations, including conditions and filters. Beyond simply tracking the operations executed, the log also captures the names of all data files affected by these operations.

In the next section, we will see how these transactional capabilities are leveraged by Delta Lake to ensure ACID compliance during data retrieval and manipulation.

Understanding Delta Lake Functionality

Let's learn how Delta Lake functions by looking at a series of illustrative examples, each designed to provide a deeper understanding of its behavior in different scenarios. For instance, consider a situation where two users, Alice and Bob, interact with a Delta Lake Table. Alice represents a data producer, while Bob is a data consumer.

Their interaction on the table can be described in four key scenarios: data reading and writing, data updating, concurrent reads and writes, and lastly, failed write attempts. Let's discuss them in detail one by one.

Reading and Writing Scenario

1. 1- Write Operation by Alice:
   a. Alice initiates this scenario by creating the Delta Table and populating it with data, as illustrated in Figure 1-3. The Delta module stores the table, for example, in two data files ("part 1" and "part 2") and saves them in a parquet format within the table directory on the storage. Upon the completion of writing the data files, the Delta module adds a transaction log, labeled as "000.json" into the '_delta_log' sub directory. This transaction log captures metadata information about the changes made to the Delta Table. This includes the operation type, the name of the newly created data files, the transaction timestamp, and any other relevant information.

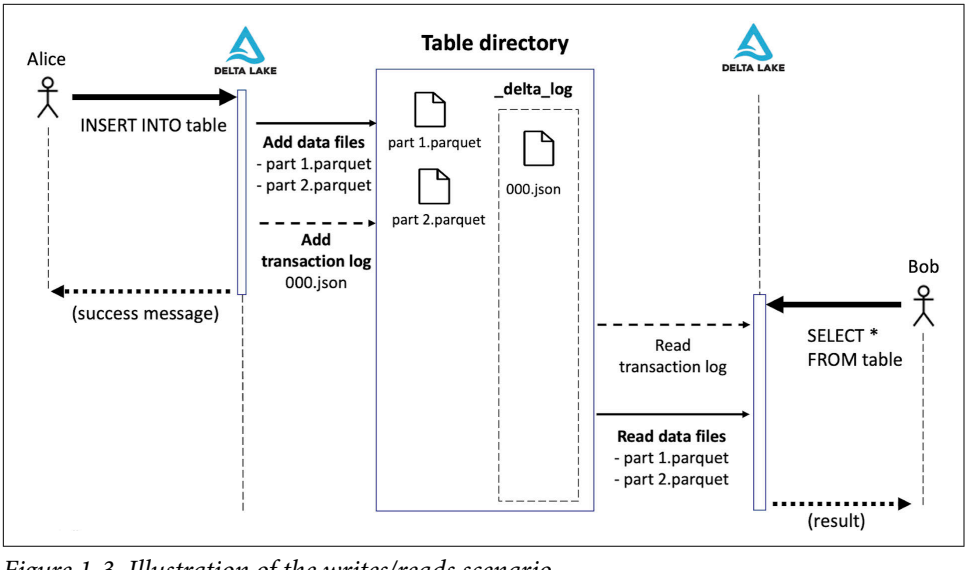


Figure 1-3. Illustration of the writes/reads scenario

1. 2- Read Operation by Bob:
   a. Subsequently, Bob queries the Delta Table through a SQL SELECT statement. However, before directly accessing the data files, the Delta module always begins by consulting the transaction log associated with the table. In this particular case, it starts by reading the "000.json" transaction log located in the '_delta_log' subfolder. This log contains information regarding the data files

"part 1.parquet" and "part 2.parquet", that capture the changes made by Alice during the write operation. The Delta module proceeds by reading these two data files and returning the results to Bob.

So, Delta Lake follows a structured approach for managing and processing the data in the lake. It always uses the transaction log as a point of reference to interact with the data files of Delta Lake tables.

Updating Scenario

In our second scenario, Alice makes an update to a record residing in file "part 1.parquet" of the Delta Table, as illustrated in Figure 1-4. However, since parquet files are immutable, Delta Lake takes a unique approach to updates. Instead of directly modifying the record within the existing file, the Delta module makes a copy of the data from the original file, and applies the necessary updates in a new data file, "part 3.parquet". It then updates the log by writing a new JSON file (001.json). The new log file is now aware that the data file "part 1.parquet" is no longer relevant to the current state of the table.

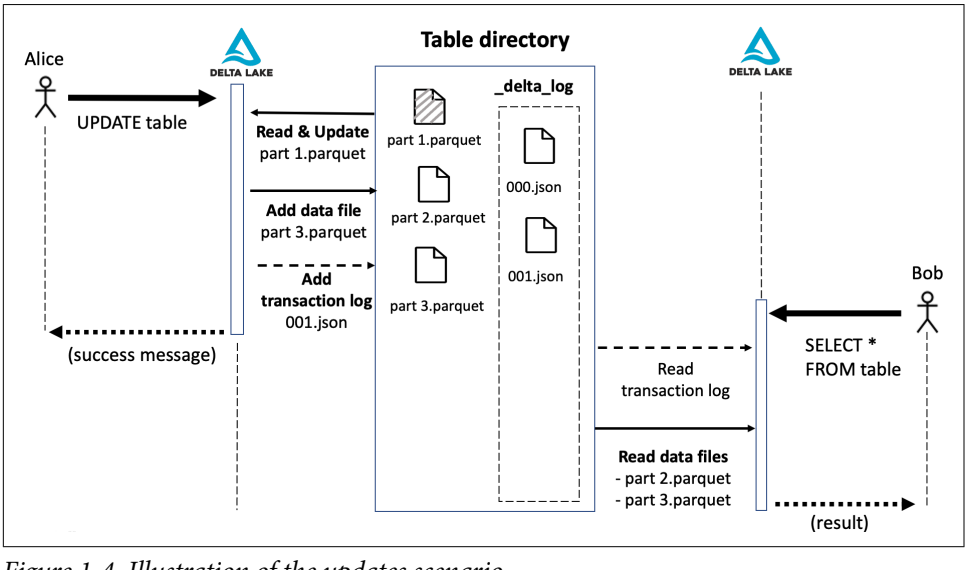


Figure 1-4. Illustration of the updates scenario

When Bob attempts to read data from the table, the Delta module first consults the transaction log to determine the valid files for the current table version. In this instance, the log indicates that only the parquet files "part 2" and "part 3" are included in the latest version of the table. As a result, the Delta module confidently reads data from these two files, and ignores the outdated file "part 1.parquet".

So, Delta Lake follows the principle of immutability; once a file is written to the storage layer, it remains unchanged. The approach of handling updates through file copying and transaction log management ensures that the historical versions of data are preserved. This offers a comprehensive record of all modifications performed on the table. We will explore in the following section how to leverage these historical versions for tasks such as auditing, rollbacks, and time travel queries.

Concurrent Writes/Reads Scenario

In this scenario, Alice and Bob are both interacting with the table simultaneously, as illustrated in Figure 1-5. Alice is inserting new data, initiating the creation of a new data file (part 4.parquet). Meanwhile, Bob is querying the table, where the Delta module starts by reading the transaction log to determine which parquet files contain the relevant data.

At the time Bob executes the query, the transaction log includes information about the parquet files "part 2" and "part 3" only, as the file "part 4.parquet" is not fully written yet. So, Bob's query reads the two latest files available that represent the current table state at that moment. Using this methodology, Delta Lake guarantees that you will always get the most recent version of the data. Your read operations will never have a deadlock state or conflicts with any ongoing operation on the table.

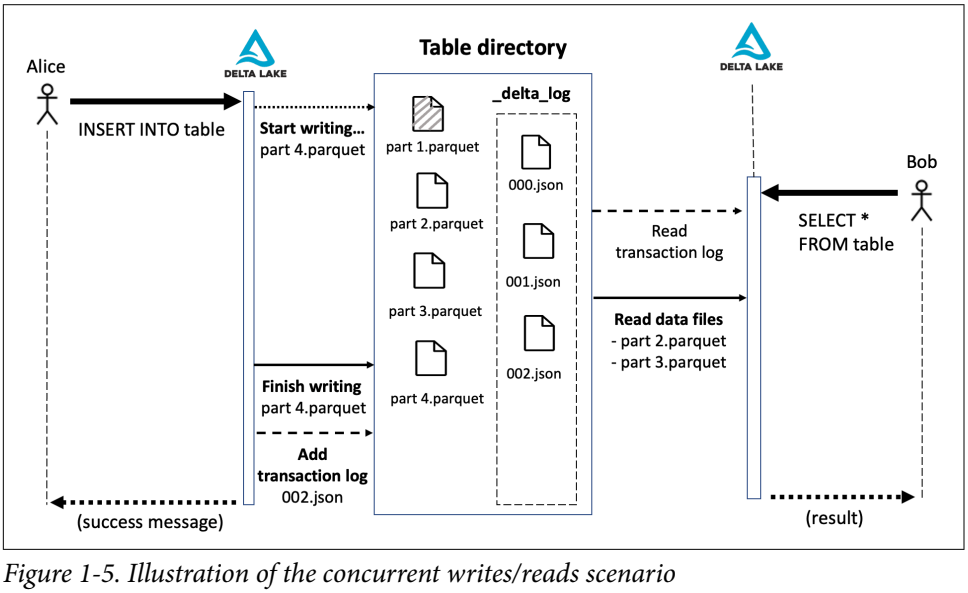


Figure 1-5. Illustration of the concurrent writes/reads scenario

Finally, once Alice's query finishes writing the new data, the Delta module adds a new json file to the transaction log, named as 002.json.

In summary, Delta Lake's transaction log helps avoid conflicts between write and read operations on the table. So, even when write operations are occurring simultaneously, read operations can proceed without waiting for them to complete. This capability helps maintain the reliability and performance of data operations on Delta Lake tables.

Failed Writes Scenario

Here is our last scenario, imagine that Alice attempts again to insert new data into the Delta Table, as illustrated in Figure 1-6. The Delta module begins writing the new data to the lake in a new file, "part 5.parquet". However, an unexpected error occurs during this operation, resulting in the creation of an incomplete file. This failure prevents the Delta Lake module from recording any information related to this incomplete file in the transaction log.

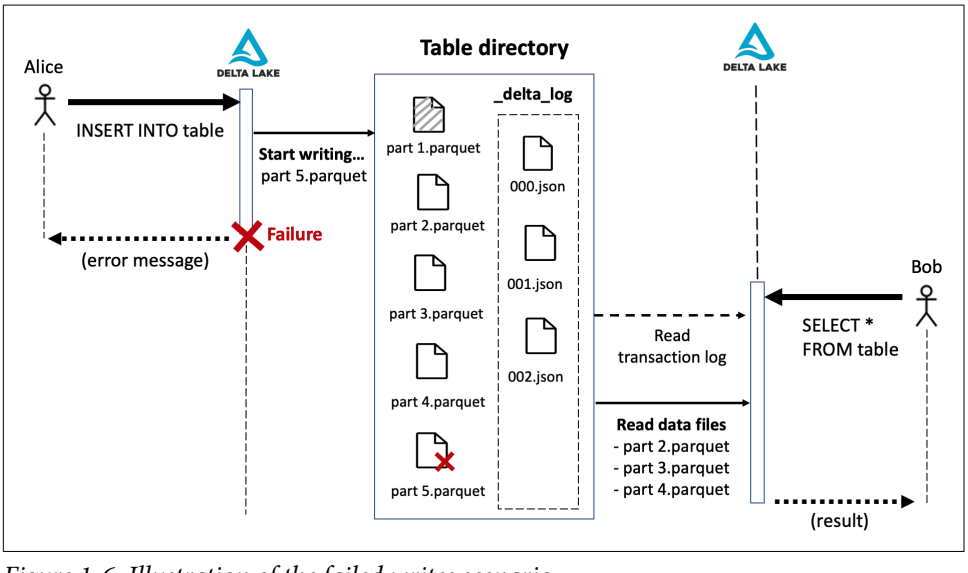


Figure 1-6. Illustration of the failed writes scenario

Now, when Bob queries the table, the Delta module starts, as usual, by reading the transaction log. Since there is no information about the incomplete file "part 5.parquet" in the log, only the parquet files "part 2", "part 3", and "part 4" will be considered for the query output. Consequently, Bob's query is protected from accessing the incomplete or dirty data created by Alice's unsuccessful write operation.

In essence, Delta Lake guarantees the prevention of reading incomplete or inconsistent data. The transaction log serves as a reliable record of committed operations on the table. And in the event of a failed write, the absence of corresponding information in the log ensures that subsequent queries won't be affected by incomplete data.

Delta Lake Advantages

Delta Lake's strength arises from its robust transaction log, which serves as the backbone of this innovative solution. This log empowers Delta Lake to deliver a range of features and advantages that can be summarized by the following key points:

Enabling ACID transactions
: The main advantage of the transaction log is that it enables Delta Lake to execute ACID transactions on traditional data lakes. This feature helps maintain data integrity and consistency when performing data operations, ensuring that they are processed reliably and efficiently.

Scalable metadata handling
: Another primary benefit of Delta Lake is the ability to handle table metadata efficiently. The table metadata, which represents information about the structure, organization, and properties of the table, is stored in the transaction log itself instead of a centralized metastore. This strategy enhances query performance when it comes to listing large directories and reading vast amounts of data.

Full audit logging
: Additionally, the transaction log serves as a comprehensive audit trail that captures every change occurring on the table. It tracks all modifications, additions, and deletions made to the data, along with the timestamps and user information associated with each operation. This allows you to trace the evolution of the data over time which facilitates troubleshooting issues, and ensures data governance.

Leveraging standard file formats
: Delta Lake uses standard file formats like Parquet and JSON. It stores the data in Parquet format, which is known for its efficiency in terms of both storage and query performance. While it records the transaction log in JSON format, which is fast to parse and generate. By leveraging these formats, Delta Lake ensures optimized data storage and retrieval.

Working with Delta Lake Tables

In this section, we dive into the practical aspects of Delta Lake. We'll walk through essential tasks such as creating Delta Lake tables, inserting data, updating tables with new information, and exploring the underlying directory structure. Through hands-on examples, you'll gain a comprehensive understanding of how Delta Lake works in your Databricks environment.

We will conduct these exercises within a new SQL notebook, named '2.1 - Delta Lake', which you can find on the book's GitHub repository.

In Databricks, tables are organized in a database within a catalog. For the purpose of these exercises, we will use the 'hive_metastore' catalog, which is available by default in every Databricks workspace. A detailed discussion on the Hive metastore will be provided in the next chapter. For the present, let us proceed by executing the following command to set the active catalog to "hive_metastore:"

```
USE CATALOG hive_metastore
```

This command configures the current notebook to use the 'hive_metastore' catalog, ensuring that all subsequent operations on Delta Lake tables are executed under this catalog.

Creating Tables

Creating Delta Lake tables closely resembles the conventional method of creating tables in standard SQL. It starts with the 'CREATE TABLE' keyword followed by the table name. Then, you provide the schema of the table by specifying the columns along with their corresponding data types. Consider the following example where we create an empty Delta Lake table named 'product_info'

```
CREATE TABLE product_info (
  product_id INT,
  product_name STRING,
  category STRING,
  price DOUBLE,
  quantity INT
)
USING DELTA;
```

In this example, 'product_info' represents a table designed to store product-related details. It includes five columns: 'product_id' of type integer, 'product_name' and 'category' of type string, 'price' of type double, and quantity (integer representing available stock of each product).

It's worth mentioning that explicitly specifying 'USING DELTA' identifies Delta Lake as the storage layer for the table, but this clause is optional. Even in its absence, the table will still be recognized as a Delta Lake table since DELTA is the default table format in Databricks.

Catalog Explorer

After creating the Delta Lake table named 'product_info' using the provided SQL script, you can explore it via the Catalog Explorer interface. To open the Catalog Explorer, click on the 'Catalog' tab in the left sidebar of your Databricks workspace.

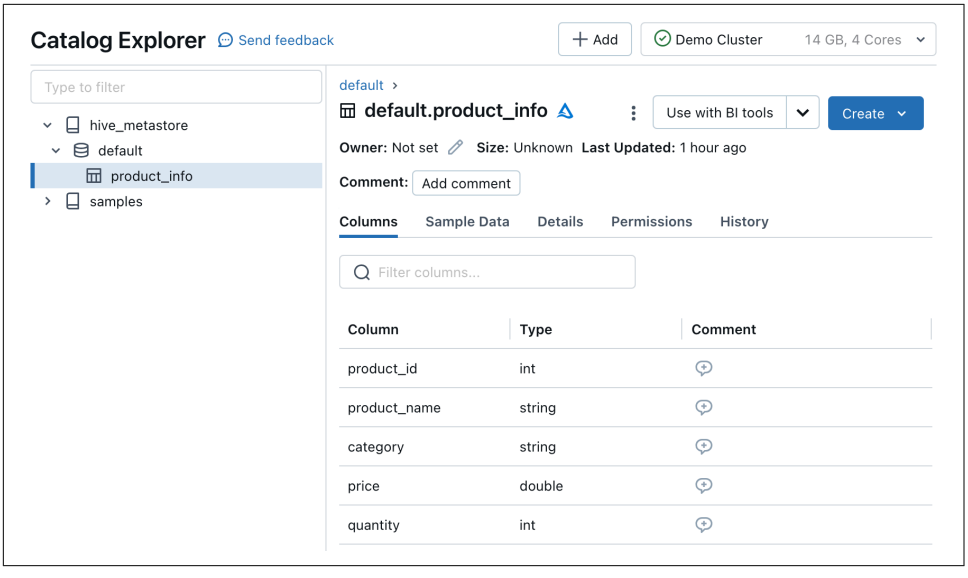


Figure 1-7. The interface of the Catalog Explorer

In the interface, navigate to the default database in the left panel to find the 'product_info' table. If you click on it, you can examine the table's columns, review sample data entries, and explore additional information displayed on the right panel, as shown in Figure 1-7.

Inserting Data

In Delta Lake, data insertion can be easily achieved through the use of the standard SQL INSERT INTO statement. Like in standard SQL, you can use this statement to add a single line or multiple lines of data:

```
INSERT INTO product_info (product_id, product_name, category, price, quantity)
VALUES (1, 'Winter Jacket', 'Clothing', 79.95, 100);
INSERT INTO product_info (product_id, product_name, category, price, quantity)
VALUES
 (2, 'Microwave', 'Kitchen', 249.75, 30),
 (3, 'Board Game', 'Toys', 29.99, 75),
 (4, 'Smartcar', 'Electronics', 599.99, 50);
```

Each operation on the table represents an individual transaction influencing the table's state. In this context, each INSERT statement generates a separate data file within the table directory. So, after executing these two INSERT commands, two distinct data files will be added to the table directory. The first file contains the initial single record, while the second data file contains the three additional records that were inserted in the subsequent INSERT statement.

This example simulates real-world scenarios where data is written to a table in several operations, such as data ingestion by multiple runs of scheduled jobs.

By executing the above two INSERT commands, four records will be inserted into the table. But if you execute them in the same cell, the displayed result will indicate the successful insertion of just three records. This outcome occurs due to the default behavior in the notebook editor where only the result of the last command executed within the cell is typically displayed.

HINT: To view the outcomes of individual SQL statements when having multiple commands in a single cell, select each specific SQL statement separately and use Shift+Ctrl+Enter to run the selected text. Alternatively, you can use separate cells for each SQL statement.

To access and verify the inserted data, simply query the table through the standard SQL SELECT statement. Like in SQL, you can also filter data based on conditions, and aggregate information if needed.

```
SELECT * FROM product_info
```

product_id	product_name	category	price	quantity
2	Microwave	Kitchen	249.75	30
3	Board Game	Toys	29.99	75
4	Smartphone	Electronics	599.99	50
1	Winter Jacket	Clothing	79.95	100

Figure 1-8. The result of the SELECT statement from the product_info table

Figure 1-8 displays the result of the SELECT query on the 'product_info' table. It displays the 4 inserted records, confirming that the two transactions were successfully performed on the table.

Exploring Table Directory

As previously discussed, the execution of the two transactional operations on the table resulted in creating two small data files in the table directory. To validate this, we can use the DESCRIBE DETAIL command on our table. This command enables you to explore the metadata of Delta Lake tables. It provides essential information about the table such as the 'numFiles' field, indicating the number of data files in the current table version.

```
DESCRIBE DETAIL product_info
```

name	location	numFiles	sizeInBytes
spark_catalog.default.product_info	dbfs:/user/hive/warehouse/product_info	2	338

Figure 1-9. Figure 1-29. The output of the DESCRIBE DETAIL command on the 'product_info' table

Figure 1-9 shows the output of the DESCRIBE DETAIL command on the 'product_info' table. The 'numFiles' column confirms that the table indeed has two data files resulting from our two INSERT operations.

Additionally, the command above shows the location of the table, indicating the directory where the table files are stored on the storage. As indicated, the 'product_info' table is stored under 'dbfs:/user/hive/warehouse/product_info'. To gain a deeper understanding of the table's file structure, we can use the %fs magic command that allows you to explore the contents of the table directory:

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

	name	size	modificationTime
1	_delta_log/	0	1708652042000
2	part-00000-aa695b3d-e814-4d72-ad60-6034c09ef2e2-c000.snappy.parquet	1595	1708652049000
3	part-00000-d406ce78-0c67-4bc6-9ec5-57b7fc13d1f7-c000.snappy.parquet	1577	1708652048000

Figure 1-10. The output of the %fs command on the 'product_info' table directory

Figure 1-10 illustrates the result of executing the above %fs command. It shows that the table directory indeed holds two data files, both in Parquet format. Furthermore, it shows the '_data_log' subdirectory, containing the translation log files of the table.

Updating Delta Lake Tables

Now, considering update operations, let's explore a scenario where the task involves adjusting the price of the product #3 (Board Game) by incrementing its price by $10.

```
UPDATE product_info
SET price = price + 10
WHERE product_id = 3
```

Examining the table directory after this update operation reveals an interesting observation: a new file addition (Figure 1-11).

	name	size	modificationTime
1	_delta_log/	0	1708793922000
2	part-00000-40ada852-ef55-4367-994a-eae08e0684d4-c000.snappy.parquet	1683	1708793936000
3	part-00000-485c4e80-678f-4c03-9330-67159e215eb8-c000.snappy.parquet	1701	1708793940000
4	part-00000-a21a2e7e-29b5-433a-a7ce-3d886f37e7dd-c000.snappy.parquet	1701	1708794052000

Figure 1-11. The output of the %fs command after the update operation

As previously mentioned, when updates occur, Delta Lake doesn't directly modify existing files, but rather creates updated copies of them. Afterward, Delta Lake leverages the transaction log to indicate which files are valid in the current version of the table. To confirm this behavior, you can run the DESCRIBE DETAIL command again.

name	location	numFiles	sizeInBytes
spark_catalog.default.product_info	dbfs:/user/hive/warehouse/product_info	2	338

Figure 1-12. The output of the DESCRIBE DETAIL command after the update operation

Figure 1-12. displays the table metadata following the update. It shows that the count of the table's files is still 2, and not 3!. These are the two files that represent the current table version, including the newly updated file resulting from the recent update operation. When querying the Delta table again, the query engine leverages the transaction logs to identify all the data files that are valid in the current version, and exclude any outdated data files. If you query the table after this update operation, you can verify that the pricing information of the product #3 has been successfully updated.

Note: Starting from Databricks Runtime Version 14, adjustments have been made to the way update and delete operations are applied, affecting the associated data files in the table directory. This change is due to the introduction of Deletion Vectors in Delta Lake (https://docs.databricks.com/delta/deletion-vectors.html)

Exploring Table History

In Delta Lake, the transaction log maintains the history of all changes made to the tables. To access the history of a table, you can use the DESCRIBE HISTORY command:

```
DESCRIBE HISTORY product_info
```

version	timestamp	userName	operation	operationParameters
3	2024-02-24T17:00:52.000+00:00	Derar Alhussein	UPDATE	{"predicate":"[\"(product_id#2123
2	2024-02-24T16:59:01.000+00:00	Derar Alhussein	WRITE	{"mode":"Append","statsOnLoad":
1	2024-02-24T16:58:57.000+00:00	Derar Alhussein	WRITE	{"mode":"Append","statsOnLoad":
0	2024-02-24T16:58:42.000+00:00	Derar Alhussein	CREATE TABLE	{"partitionBy":"[]","description":nu

Figure 1-13. 13 The output of the DESCRIBE HISTORY command on the 'product_info' table

Figure 1-13 illustrates the table history, revealing four distinct versions starting from the table creation at version zero. Moving forward, versions 1 and 2 indicate write operations on the table, representing our two insert commands, while version 3 indicates the update operation. All this information is captured within the transaction log of the table.

The transaction log is located under the '_delta_log' folder in the table directory. You can navigate to this folder using the %fs magic command:

```
%fs ls 'dbfs:/user/hive/warehouse/product_info/_delta_log'
```

name	size	modificationTime
00000000000000000000.crc	2220	1708793932000
00000000000000000000.json	1224	1708793922000
00000000000000000001.crc	2926	1708793940000
00000000000000000001.json	1282	1708793937000
00000000000000000002.crc	3619	1708793942000
00000000000000000002.json	1274	1708793941000
00000000000000000003.crc	3632	1708794054000
00000000000000000003.json	1927	1708794052000

Figure 1-14. 14 The output of the %fs command on the _delta_log folder

Figure 1-14 illustrates the contents of the '_delta_log' folder located within the 'product_info' table directory. You can observe that it contains nothing but JSON files, along with their associated checksum[1] files (having the .crc extension). Each JSON file

1 A checksum is a unique value computed from the contents of a file using an algorithm. It serves as a sort of digital fingerprint that helps determine if any changes or corruption have occurred in the associated file. In other words, a checksum ensures data integrity of the associated file.

corresponds to a distinct version of the Delta Lake table. In the context of the 'product_info' table, we observe four JSON files, corresponding precisely to the four table versions examined previously through the DESCRIBE HISTORY command.

To gain a deeper understanding of the transaction log, we can use the "%fs head" command to explore the content of one of those JSON files. In particular, we can examine the latest JSON file that represents the version #3 of the table:

```
%fs head 'dbfs:/user/hive/warehouse/product_info/_delta_log/
00000000000000000003.json'

{ "commitInfo":{"operation": "UPDATE", "timestamp": 1708794052735,
                "userName": "Derar Alhussein", ...}
}
{ "add":{"path": "part-00000-a21a2e7e-29b5-433a-a7ce-3d886f37e7dd-
c000.snappy.parquet",
         "modificationTime": 1708794052000, ...}
}
{ "remove":{"path": "part-00000-485c4e80-678f-4c03-9330-67159e215eb8-
c000.snappy.parquet",
           "deletionTimestamp": 1708794052717, ...}
}
```

The output of the '%fs head' command above shows that the JSON file contains structured JSON data about our update operation. The 'add' element specifies the new data file appended to the table, while the 'remove' element specifies the data file marked for soft deletion, in other words, It's no longer part of the latest table version.

Exploring Delta Time Travel

Time Traveling is a feature in Delta Lake for exploring the historical evolution of the data in Delta Lake tables. The key aspect of Delta Lake Time Travel is the automatic versioning of the table. This versioning provides the full audit trail of all the changes that have happened on the table. Whenever a change is made to the data, Delta Lake captures and stores this change as a new version. Each version represents the state of the table at a specific point in time.

To explore the historical versions of a Delta table, you can leverage the DESCRIBE HISTORY command in SQL. This command provides a detailed log of all the operations performed on the table, including information such as the timestamp of the operation, the type of operation (insert, update, delete, etc), and any additional metadata associated with the change.

Here's an example of how you might use the DESCRIBE HISTORY command:

```
DESCRIBE HISTORY <table_name>;
```

This command returns a table containing the operations performed on the specified table in reverse chronological order, along with relevant details for each operation.

Let's review again the history of the 'product_info' tabl

```
DESCRIBE HISTORY product_info
```

version	timestamp	userName	operation	operationParameters
3	2024-02-24T17:00:52.000+00:00	Derar Alhussein	UPDATE	{"predicate":"[\"(product_id#2123
2	2024-02-24T16:59:01.000+00:00	Derar Alhussein	WRITE	{"mode":"Append","statsOnLoad":
1	2024-02-24T16:58:57.000+00:00	Derar Alhussein	WRITE	{"mode":"Append","statsOnLoad":
0	2024-02-24T16:58:42.000+00:00	Derar Alhussein	CREATE TABLE	{"partitionBy":"[]","description":nu

Figure 1-15. 15 The output of the DESCRIBE HISTORY command on the 'product_info' table

Our table has currently four distinct versions, as illustrated in Figure 1-15:

- **Version 0:** This is the initial version of the table, representing its state at creation. Since the table was created empty, this version captures only the initial schema and metadata of the table.
- **Versions 1 and 2:** These versions indicate write operations on the table, representing our two insert commands.
- **Version 3:** This version indicates the update operation on the table. It currently represents the latest state of the table.

Querying Older versions

The versioning system in Delta Lake is automatic, ensuring that every operation performed on a table is assigned a unique version number and a timestamp. To query older versions of the table, Delta Lake offers two distinct approaches, either by timestamp or version number.

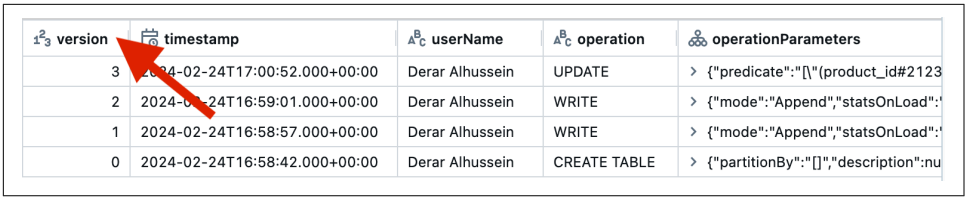
Querying by Timestamp

The first method allows you to retrieve the table's state as it existed at a specific point in time. This involves specifying the desired timestamp in the SELECT statement using the TIMESTAMP AS OF keyword:

```
SELECT * FROM <table_name> TIMESTAMP AS OF <timestamp>
```

Querying by Version Number

The second method involves using the version number associated with each operation on the table, as illustrated in the table history in Figure 1-16.



version	timestamp	userName	operation	operationParameters
3	2024-02-24T17:00:52.000+00:00	Derar Alhussein	UPDATE	{"predicate":"[\"(product_id#2123
2	2024-02-24T16:59:01.000+00:00	Derar Alhussein	WRITE	{"mode":"Append","statsOnLoad":
1	2024-02-24T16:58:57.000+00:00	Derar Alhussein	WRITE	{"mode":"Append","statsOnLoad":
0	2024-02-24T16:58:42.000+00:00	Derar Alhussein	CREATE TABLE	{"partitionBy":"[]","description":nu

Figure 1-16. 16 The output of the DESCRIBE HISTORY command on the 'product_info' table

You can use the VERSION AS OF keyword to travel back in time to a specific version of the table:

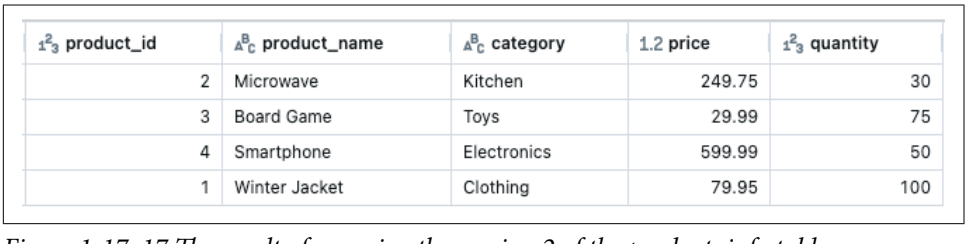
```
SELECT * FROM <table_name> VERSION AS OF <version>
```

Consider a scenario where we need to retrieve the product data exactly as it existed before the update operation, identified as version #2 in our 'product_info' table. We can simply use the following query:

```
SELECT * FROM product_info VERSION AS OF 2
```

Alternatively, you can use its short syntax represented by @v followed by the version number:

```
SELECT * FROM product_info@v2
```



product_id	product_name	category	price	quantity
2	Microwave	Kitchen	249.75	30
3	Board Game	Toys	29.99	75
4	Smartphone	Electronics	599.99	50
1	Winter Jacket	Clothing	79.95	100

Figure 1-17. 17 The result of querying the version 2 of the product_info table

Figure 1-17 shows the result of querying the version 2 of our table. So, Delta Lake's Time Travel enables you to independently investigate different versions of the data without impacting the current state of the table. This feature is possible thanks to those extra data files that had been marked as removed in our transaction log.

Rollbacking Back to Previous Versions

Delta Lake Time Travel is particularly useful in scenarios where undesired data changes need to be rolled back to a previous state. For instance, in case of bad writes or unintended data modifications, you can easily undo these changes by reverting to a previous version of the table.

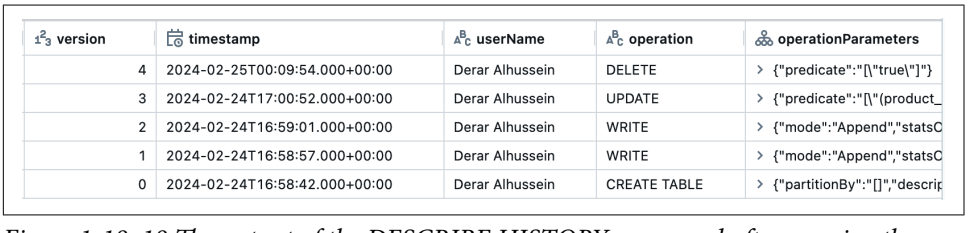
Delta Lake offers the "RESTORE TABLE" command that allows you to roll back the table to a specific timestamp or version number:

```
RESTORE TABLE <table_name> TO TIMESTAMP AS OF <timestamp>
RESTORE TABLE <table_name> TO VERSION AS OF <version>
```

Imagine a scenario where data has been accidentally deleted from our 'product_info' table, and we need to restore them.

```
DELETE FROM product_info
```

Upon executing the DELETE command, it removes all four records currently in the table. You can easily confirm this by querying the table again. In addition, we can review the table history to see that the delete operation has been recorded as a new table version, labeled as version #4 (Figure 1-18.



version	timestamp	userName	operation	operationParameters
4	2024-02-25T00:09:54.000+00:00	Derar Alhussein	DELETE	{"predicate":"[\"true\"]"}
3	2024-02-24T17:00:52.000+00:00	Derar Alhussein	UPDATE	{"predicate":"[\"(product_
2	2024-02-24T16:59:01.000+00:00	Derar Alhussein	WRITE	{"mode":"Append","statsC
1	2024-02-24T16:58:57.000+00:00	Derar Alhussein	WRITE	{"mode":"Append","statsC
0	2024-02-24T16:58:42.000+00:00	Derar Alhussein	CREATE TABLE	{"partitionBy":"[]","descrip

Figure 1-18. 18 The output of the DESCRIBE HISTORY command after running the DELETE command

To roll back the table to a previous version that existed before the deletion occurred, specifically version 3, we can use the RESTORE TABLE command:

```
RESTORE TABLE product_info TO VERSION AS OF 3
```

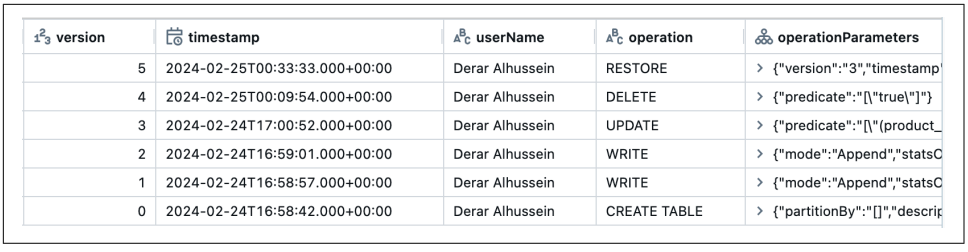
table_size_after_restore	num_of_files_after_restore	num_removed_files	num_restored_files
3384	2	0	2

Figure 1-19. 19 The output of the RESTORE TABLE command on the product_info table

Figure 1-19 displays the output of the restoration operation. It shows that 2 files have been restored, confirming the data has been successfully restored to its original state. The 'product_info' table now contains the complete dataset, as it did before the deletion took place. You can easily confirm this by querying the table again.

We can also examine what really happened at our table by exploring its history:

```
DESCRIBE HISTORY product_info
```



version	timestamp	userName	operation	operationParameters
5	2024-02-25T00:33:33.000+00:00	Derar Alhussein	RESTORE	{"version":"3","timestamp
4	2024-02-25T00:09:54.000+00:00	Derar Alhussein	DELETE	{"predicate":"[\"true\"]"}
3	2024-02-24T17:00:52.000+00:00	Derar Alhussein	UPDATE	{"predicate":"[\"(product_
2	2024-02-24T16:59:01.000+00:00	Derar Alhussein	WRITE	{"mode":"Append","statsC
1	2024-02-24T16:58:57.000+00:00	Derar Alhussein	WRITE	{"mode":"Append","statsC
0	2024-02-24T16:58:42.000+00:00	Derar Alhussein	CREATE TABLE	{"partitionBy":"[]","descrip

Figure 1-20. The output of the DESCRIBE HISTORY command after the restoration operation

Figure 1-20 displays the table history after the restoration operation. It shows that this operation has been recorded as a new table version, labeled as version #5.

In summary, Delta Lake's Time Travel brings a new level of flexibility to data management within Delta tables. It provides you with the capability to travel back in time to a specific version of your tables, and restore them to a previous state if needed.

Optimizing Delta Lake Tables

Delta Lake provides an advanced feature for optimizing table performance through compacting small data files into larger ones. This optimization is particularly significant as it enhances the speed of read queries from a Delta Lake table. You trigger compaction by executing the OPTIMIZE command:

```
OPTIMIZE <table_name>
```

If you have a table that has accumulated many small files due to frequent write operations. By running the OPTIMIZE command, these small files can be compacted into one or more larger files. This concept is illustrated in an example in Figure 1-21, where the optimization process results in two consolidated data files instead of six small files.

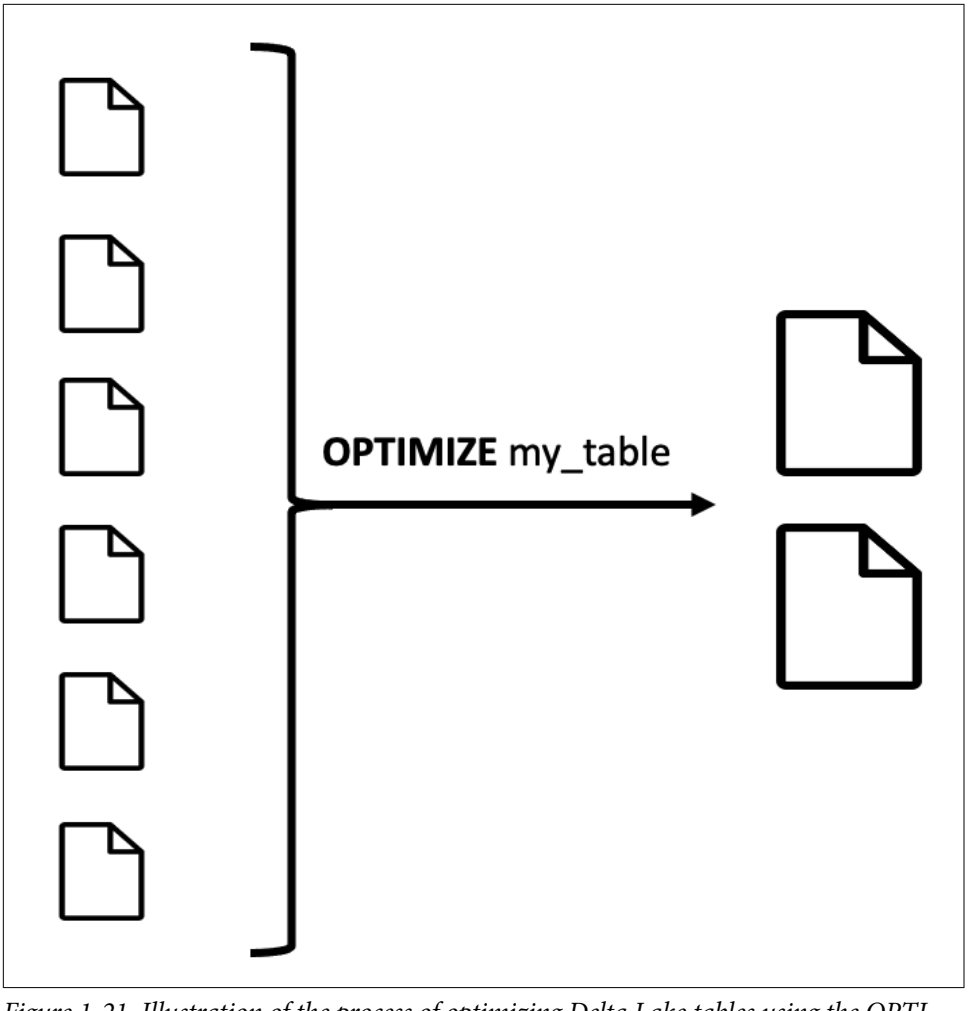


Figure 1-21. Illustration of the process of optimizing Delta Lake tables using the OPTIMIZE command

Table optimization improves the overall performance of the table by minimizing overhead associated with file management, and enhancing the efficiency of data retrieval operations.

Z-Order Indexing

A notable extension of the OPTIMIZE command is the ability to leverage Z-Order indexing. Z-Order indexing involves the reorganization and co-location of column information within the same set of files. To perform Z-Order indexing, you simply

add the ZORDER BY keyword to the OPTIMIZE command. This should be followed by specifying one or more column names on which the indexing will be applied

```
OPTIMIZE <table_name>
ZORDER BY <column_names>
```

For instance, back to our previous example in Figure 1-21, let's consider the data files containing a numerical column such as 'ID' that ranges between 1 and 100. Applying Z-Order indexing to this column during the optimization process results in different content written to the two compacted files. In this case, Z-Order indexing will ensure that the first compacted file contains values ranging from 1 to 50, while the subsequent file contains values from 51 to 100, as illustrated in Figure 1-22.

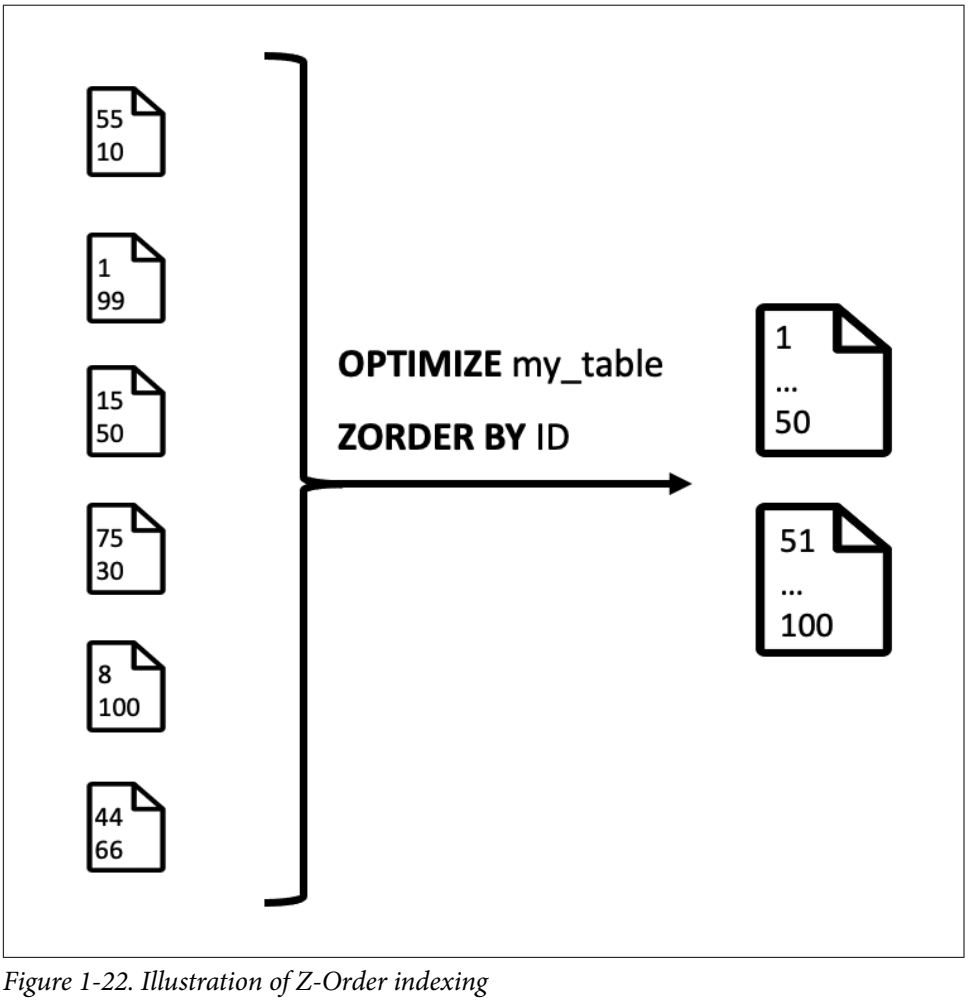


Figure 1-22. Illustration of Z-Order indexing

This strategic arrangement of data enables the application of the data skipping algorithm in Delta Lake. This algorithm leverages Z-Order indexing to skip unnecessary file scans when processing queries. In the provided example, if a query targets an ID, such as 25, Delta Lake can quickly determine that ID #25 resides in the first compacted file. Consequently, it can confidently ignore scanning the second file altogether, resulting in significant time savings.

Let's now optimize our product_info table that currently has 2 small data files, as indicated in the 'NumFiles' field of the table metadata (Figure 1-23):

```
DESCRIBE DETAIL product_info
```

name	location	numFiles	sizeInBytes
spark_catalog.default.product_info	dbfs:/user/hive/warehouse/product_info	2	3384

Figure 1-23. The output of the DESCRIBE DETAIL before optimization

We can use the OPTIMIZE command to combine these files toward an optimal size

```
OPTIMIZE product_info
ZORDER BY product_id
```

path	metrics
dbfs:/user/hive/warehouse/product_info	{"numFilesAdded":1,"numFilesRemoved":2,"filesAdded":{"min":1745,"max":174...

Figure 1-24. The output of the OPTIMIZE command

Figure 1-24 shows the output of the OPTIMIZE command. The 'numFilesRemoved' in the metrics column indicates that two small data files have been soft deleted, while the 'numFilesAdded' metric indicates that a new optimized file is added, compacting those two files. In addition, we have added the Z-order indexing with our OPTIMIZE command. In our case, we apply Z-order indexing to the product_id column. However, with such a small dataset, the benefits of Z-order indexing may not be as significant, and its impact may not be noticeable.

To confirm the result of the optimization process, let's review again the details of our table

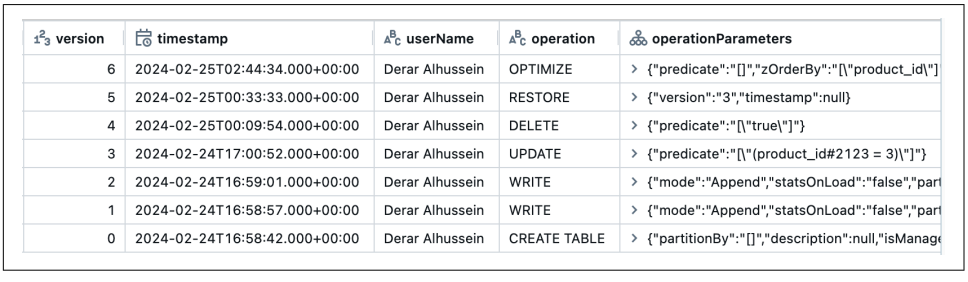
```
DESCRIBE DETAIL product_info
```

name	location	numFiles	sizeInBytes
spark_catalog.default.product_info	dbfs:/user/hive/warehouse/product_info	1	1745

Figure 1-25. The output of the DESCRIBE DETAIL after optimization

Indeed, as illustrated in Figure 1-25, the current table version consists of only one consolidated data file, indicating the success of the optimization operation. In addition, we can check how the OPTIMIZE operation has been recorded in our table history:

```
DESCRIBE HISTORY product_info
```



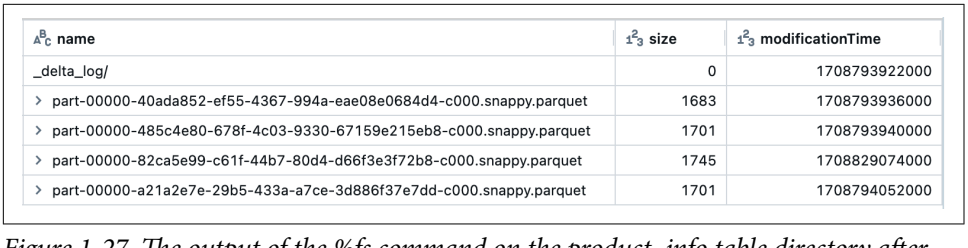
version	timestamp	userName	operation	operationParameters
6	2024-02-25T02:44:34.000+00:00	Derar Alhussein	OPTIMIZE	{"predicate":"[]","zOrderBy":"[\"product_id\"]
5	2024-02-25T00:33:33.000+00:00	Derar Alhussein	RESTORE	{"version":"3","timestamp":null}
4	2024-02-25T00:09:54.000+00:00	Derar Alhussein	DELETE	{"predicate":"[\"true\"]"}
3	2024-02-24T17:00:52.000+00:00	Derar Alhussein	UPDATE	{"predicate":"[\"(product_id#2123 = 3)\"]"}
2	2024-02-24T16:59:01.000+00:00	Derar Alhussein	WRITE	{"mode":"Append","statsOnLoad":"false","part
1	2024-02-24T16:58:57.000+00:00	Derar Alhussein	WRITE	{"mode":"Append","statsOnLoad":"false","part
0	2024-02-24T16:58:42.000+00:00	Derar Alhussein	CREATE TABLE	{"partitionBy":"[]","description":null,"isManage

Figure 1-26. The output of the DESCRIBE HISTORY command after the optimization operation

As expected and illustrated in Figure 1-26, the OPTIMIZE command created another version of our table. This means that version 6 is the most recent version of the table.

Lastly, let us explore the data files in our table directory.

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



name	size	modificationTime
_delta_log/	0	1708793922000
part-00000-40ada852-ef55-4367-994a-eae08e0684d4-c000.snappy.parquet	1683	1708793936000
part-00000-485c4e80-678f-4c03-9330-67159e215eb8-c000.snappy.parquet	1701	1708793940000
part-00000-82ca5e99-c61f-44b7-80d4-d66f3e3f72b8-c000.snappy.parquet	1745	1708829074000
part-00000-a21a2e7e-29b5-433a-a7ce-3d886f37e7dd-c000.snappy.parquet	1701	1708794052000

Figure 1-27. The output of the %fs command on the product_info table directory after optimization

In Figure 1-27, we can see that there are 4 data files in the table directory. However, it is important to remember that our current table version references only one file following the optimization operation. This means that other data files in the directory are unused files, and we can simply clean them up. In the next section, we will learn how to achieve this task with Vacuuming.

In essence, Delta Lake's OPTIMIZE command, coupled with Z-Order indexing, offers a powerful mechanism to optimize table performance. It enhances the speed of read queries by compacting small files and intelligently organizing their column information.

Vacuuming

Delta Lake's Vacuuming provides an efficient mechanism for managing unused data files within a Delta table. As data evolves over time, there might be scenarios where certain files become obsolete, either due to uncommitted changes or because they are no longer part of the latest state of the table. The VACUUM command in Delta Lake enables you to clean up these unwanted files, ensuring efficient storage management that saves storage space and cost.

Here's an example of how you might use the VACUUM command:

```
VACUUM <table_name> [RETAIN num HOURS]
```

The process involves specifying a retention period threshold for the files, so the command will automatically remove all files older than this threshold. The default retention period is set to 7 days, meaning that vacuum operation will prevent you from deleting files less than 7 days old. This is a safety measure to ensure that no active or ongoing operations are still referencing any of the files to be deleted.

It's important to note that running the VACUUM command comes with a trade-off. Once the operation is executed, and files older than the specified retention period are deleted, you lose the ability to time-travel back to a version older than that period. This is because the associated data files are no longer available. Therefore, it is crucial to carefully consider the retention period based on your data retention policies and data storage requirements.

Vacuuming in Action

Let's optimize the storage and tidy up the file structure of our product_info table. Before we start, let us first explore the data files in the table directory.

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

name	size	modificationTime
_delta_log/	0	1708793922000
part-00000-40ada852-ef55-4367-994a-eae08e0684d4-c000.snappy.parquet	1683	1708793936000
part-00000-485c4e80-678f-4c03-9330-67159e215eb8-c000.snappy.parquet	1701	1708793940000
part-00000-82ca5e99-c61f-44b7-80d4-d66f3e3f72b8-c000.snappy.parquet	1745	1708829074000
part-00000-a21a2e7e-29b5-433a-a7ce-3d886f37e7dd-c000.snappy.parquet	1701	1708794052000

Figure 1-28. The output of the %fs command on the product_info table directory before vacuuming

As shown in Figure 1-28, there are currently 4 data files in the table directory. However, it is important to remember that our current table version references only one file following the optimization operation detailed in the previous section. This means

that other data files in the directory are unused files, and we can simply clean them up using the VACUUM command.

```
VACUUM product_info
```

However, upon executing the command, you realize that the files are still present in the table directory. This is because, by default, VACUUM retains files for a period of 7 days to ensure ongoing operations can still access them if needed.

To overcome this default behavior, we attempt to specify a retention period of zero hours to retain only the current version of the data.

```
VACUUM product_info RETAIN 0 HOURS
IllegalArgumentException: requirement failed: Are you sure you would like to vacuum
files with such a low retention period? If you have
writers that are currently writing to this table, there is a risk that you may cor-
rupt the state of your Delta table.
```

However, this command throws an exception since the retention period is low, compared to the default retention period of 7 days. As a workaround solution, and for demonstration purposes only, we can temporarily disable the retention duration check in Delta Lake. It's important to note that this approach is not recommended for production environments due to potential data integrity issues.

```
SET spark.databricks.delta.retentionDurationCheck.enabled = false
```

With the retention duration check disabled, we can now proceed and rerun our VACUUM command with 0 HOURS retention period. To confirm its output, let's explore the table directory

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

name	size	modificationTime
_delta_log/	0	1708793922000
part-00000-82ca5e99-c61f-44b7-80d4-d66f3e3f72b8-c000.snappy.parquet	1745	1708829074000

Figure 1-29. The output of the %fs command on the product_info table directory before vacuuming

Indeed, as illustrated in Figure 1-29, the operation this time successfully removed the old data files from the table directory.

While the cleanup operation enhances storage efficiency, it comes at the cost of losing access to older data versions for time travel queries. Attempting to query an old table version results in a "file not found" exception, since the corresponding data files have been deleted during the above VACUUM operation.

```
SELECT * FROM product_info@v1
FileReadException: Error while reading file part-00000-40ada852-ef55-4367-994a-
```

```
eae08e0684d4-c000.snappy.parquet. File referenced in the transaction log cannot be
found.
Caused by: FileNotFoundException: Operation failed: "The specified path does not
exist.", 404, GET, PathNotFound, "The specified path does not exist.
```

Dropping Delta Lake Tables

In the final step of managing Delta Lake tables within the lakehouse architecture, we can drop the table and permanently erase its associated data. Similar to SQL syntax, we use the DROP TABLE command for this purpose

```
DROP TABLE product_info
```

Upon executing this command, the table, along with its data, will be deleted from the lakehouse environment. To confirm this action, you can attempt to query the table again, only to find that it is no longer found in the database. Furthermore, the directory containing the table's files is also completely removed.

```
%fs ls 'dbfs:/user/hive/warehouse/product_info'
FileNotFoundException: No such file or directory dbfs:/user/hive/warehouse/prod-
uct_info
```

In conclusion, the VACUUM command provides a mechanism for optimizing storage by removing unnecessary data files of Delta Lake tables. However, it's crucial to understand the impacts of file retention duration and consider the trade-offs between storage efficiency and historical data accessibility.

CHAPTER 2

Mastering Relational Entities in Databricks

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at sevans@gmail.com.

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

Databases in Databricks

In Databricks, a database essentially corresponds to a schema in the Hive metastore. 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.

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

Every Databricks workspace includes a central Hive metastore that all clusters can access to persist table metadata. By default, a database named "default" is provided in the Hive metastore. When you create tables without explicitly specifying a database name, they are created under this default database. The data for these tables is stored in the default directory for Hive, typically located at "/user/hive/warehouse" on the DBFS file system, as illustrated in Figure 2-1.

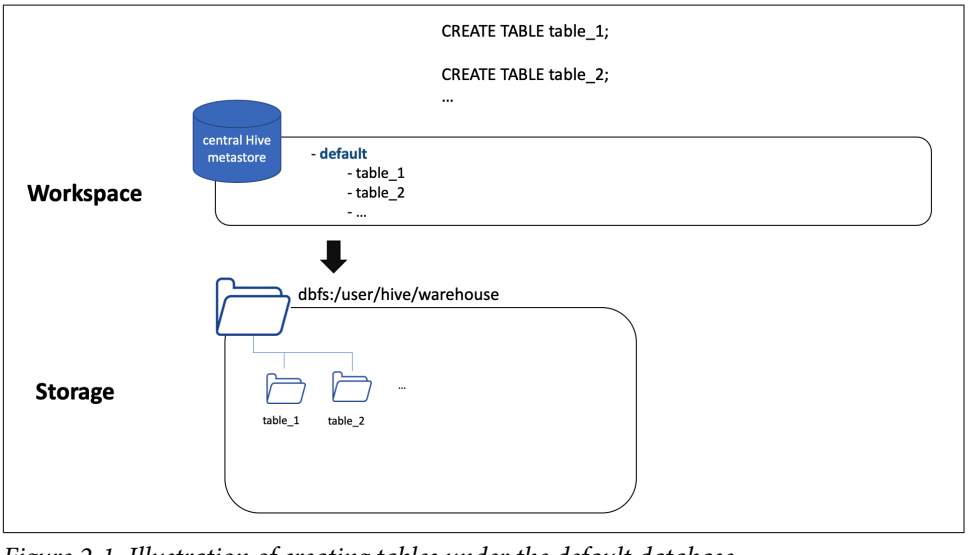


Figure 2-1. Illustration of creating tables under the default database

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 2-2.

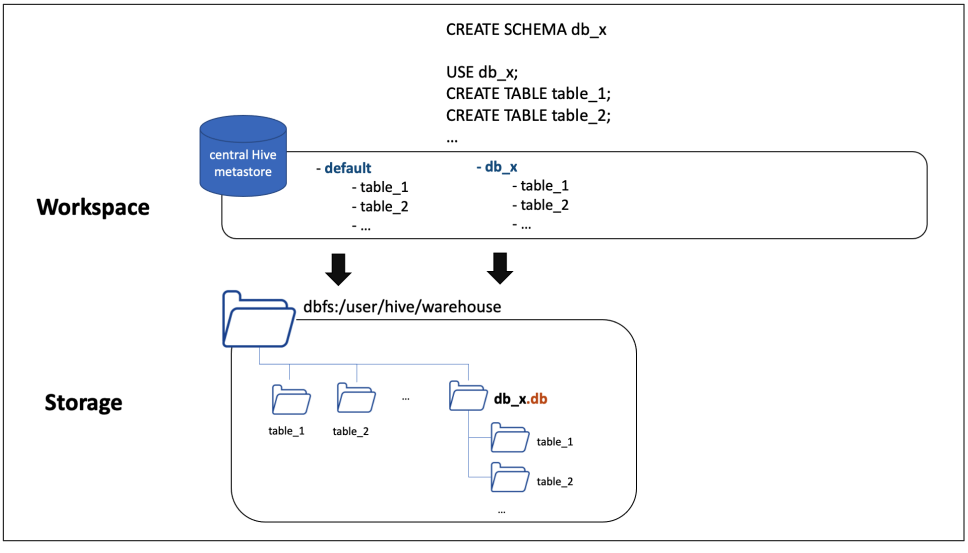


Figure 2-2. Illustration of creating an additional database and tables within this database

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 2-3.

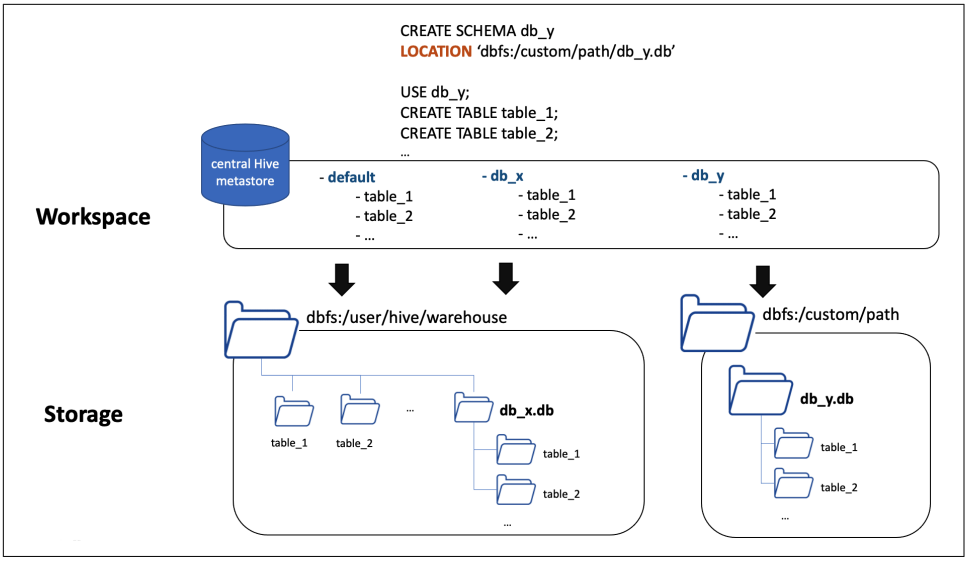


Figure 2-3. Illustration of creating a database in a custom location

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 2-1 summarizes the key differences between these two types of tables.

Table 2-1. Comparison of managed and external tables in Delta Lake

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

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 Apache Hive or Unity Catalog. When you create a managed table, the table data is stored within the database directory. 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 its metadata is 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 DBFS, like in S3 buckets or Azure storage containers.

To better understand external tables, let's revisit our diagram. Figure 2-4 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.

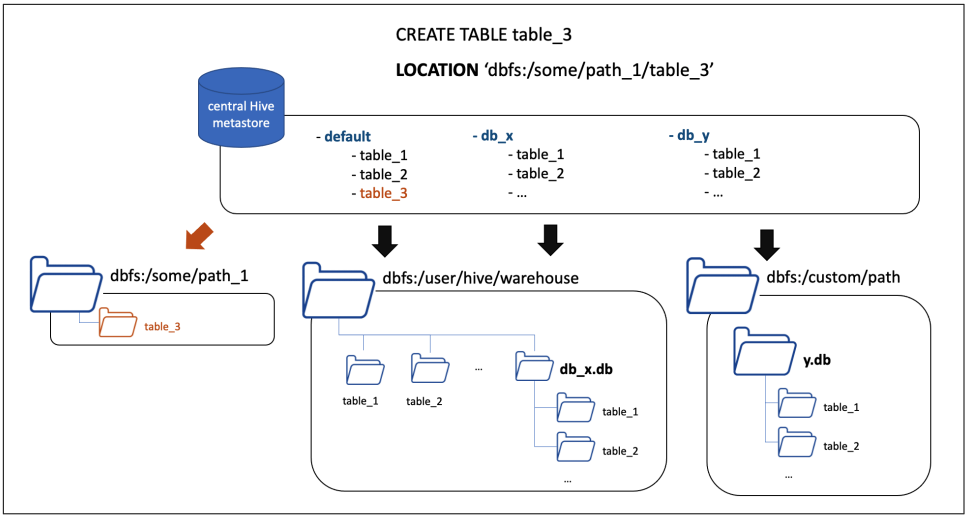


Figure 2-4. Illustration of creating an external table in the default database

Similarly, we can create an external table in any database. Figure 2-5 illustrates creating an external table in our database 'db_x'. First, we specify the database name via the USE 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.

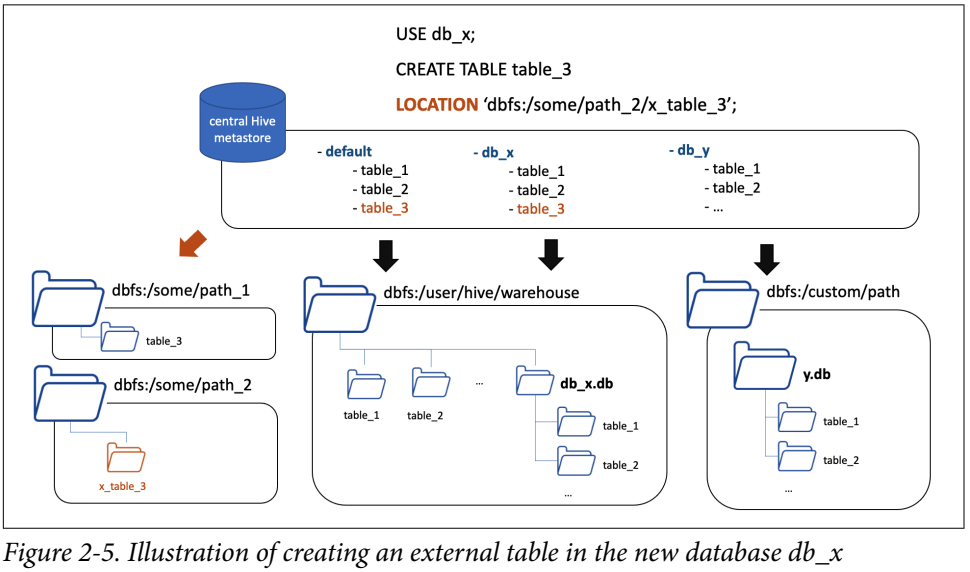


Figure 2-5. Illustration of creating an external table in the new database db_x

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 2-6 illustrates this scenario by using our custom-location database 'db_y'. Once again, we specify the database using the USE 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.

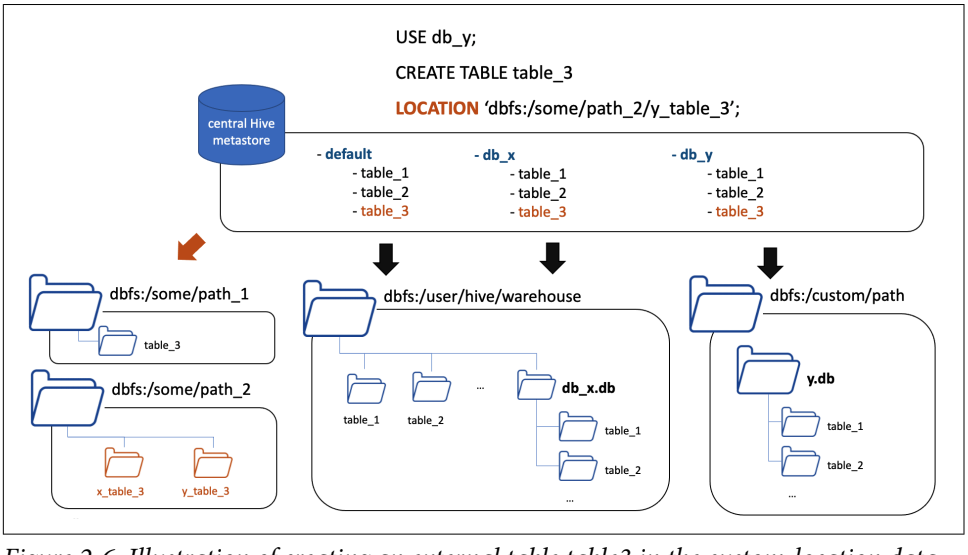


Figure 2-6. Illustration of creating an external table table3 in the custom-location database db_y

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 on 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 on the 'Catalog' tab in the left sidebar of your Databricks workspace.

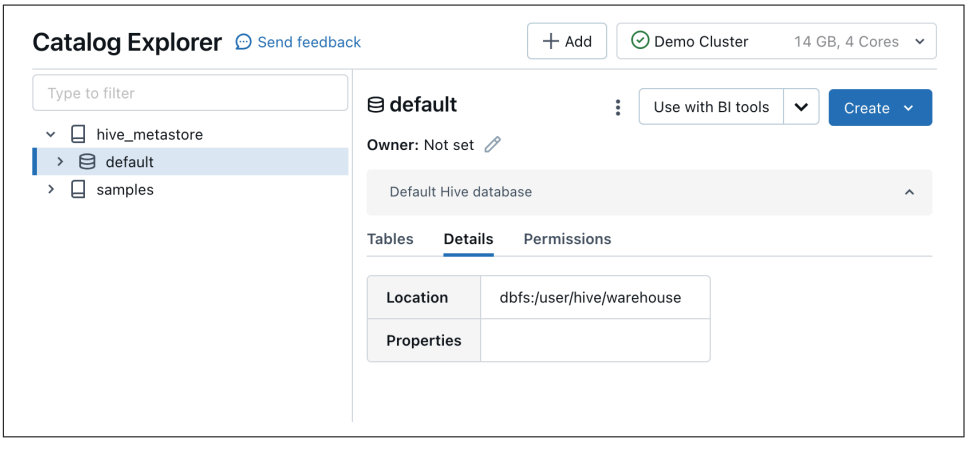


Figure 2-7. The interface of the Catalog Explorer showing the "default" database in hive_metastore

By default, under the 'hive_metastore' catalog, there's a database named "default", as illustrated in Figure 2-7. We'll begin by creating some tables within this default database.

Creating Managed Tables

Firstly, 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 on the catalog icon located in the sidebar of the notebook editor (Figure 2-8.

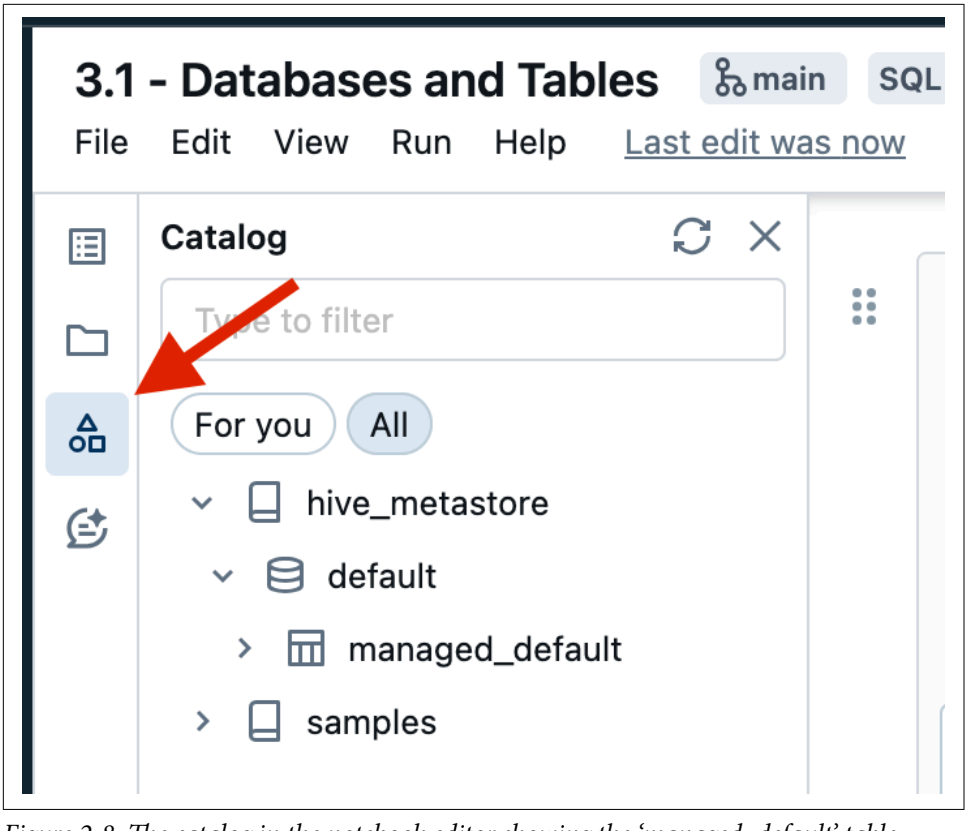


Figure 2-8. The catalog in the notebook editor showing the 'managed_default' table

Executing the "DESCRIBE EXTENDED" command on our table provides advanced metadata information, as illustrated in Figure 2-9.

```
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 2-9. 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 on the DBFS file system.

```
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 2-10.

```
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 2-10. 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, 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 in Hive, it effectively 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.

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:

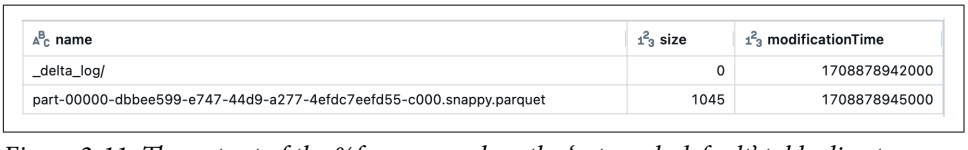
```
%fs ls 'dbfs:/user/hive/warehouse/managed_default'
FileNotFoundException: No such file or directory dbfs:/user/hive/warehouse/
managed_default
```

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

```
DROP TABLE external_default
```

Dropping the external table removes its entry from the Hive metastore, but since the underlying data is stored outside the database directory, the data files remain intact. We can easily confirm that both the table directory and its data files still persist by checking the table directory:

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



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

Figure 2-11. The output of the %fs command on the 'external_default' table directory

Figure 2-11 confirms that the data files of the external table continue to exist in the table directory even after the table has been dropped. You can manually remove the table directory and its content by the running the 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 essentially 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
```

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

Figure 2-12. The output of the DESCRIBE DATABASE EXTENDED command on the 'new_default' schema

As illustrated in Figure 2-12, the new database is stored under the default Hive directory with a '.db' extension to distinguish it from other table folders in the directory.

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 keyword

USE new_default;

```
CREATE TABLE managed_new_default
 (country STRING, code STRING, dial_code STRING);
 INSERT INTO managed_new_default
VALUES ('France', 'Fr', '+33');
----------------------------------
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 2-13:

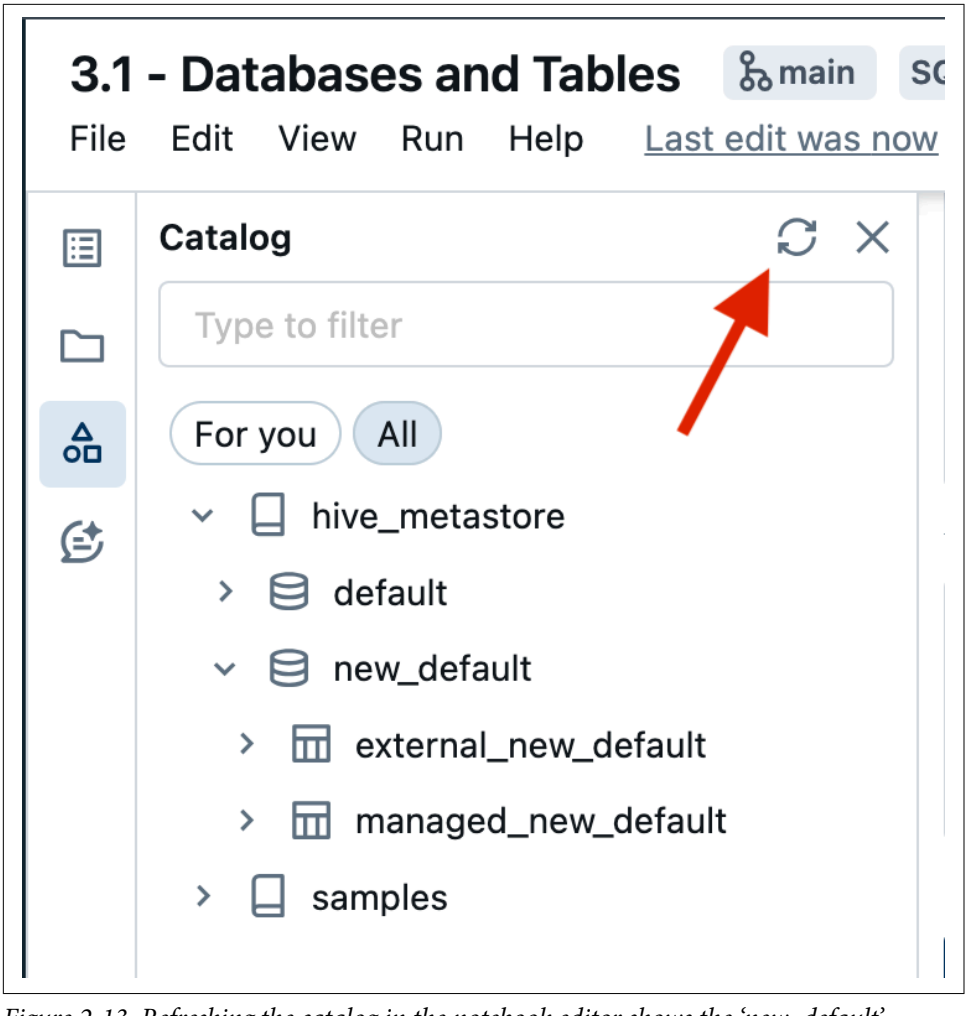


Figure 2-13. Refreshing the catalog in the notebook editor shows the 'new_default' schema and its tables

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 2-13). While the second table where we use the LOCATION keyword has been defined as an external table under '/mnt/demo' location (Figure 2-14).

```
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 2-14. 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 2-15. 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 2-16).

```
%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 2-16. The output of the %fs command on the 'external_new_default' table directory

Working In a Custom-Location Schema

Creating the Database:

In our last scenario, we will create a database in a custom location outside of the default Hive directory. To achieve this, we simply use the the CREATE SCHEMA stemement, 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 2-17).

```
DESCRIBE DATABASE EXTENDED custom
```

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

Figure 2-17. The output of the DESCRIBE DATABASE EXTENDED command on the 'custom' schema

Tables Creation:

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

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 2-18). While the 'external_custom' table is an external table since it is created outside the database directory (Figure 2-19).

```
USE custom;
CREATE TABLE managed_custom
 (country STRING, code STRING, dial_code STRING);
 INSERT INTO managed_custom
VALUES ('France', 'Fr', '+33');
---------------------------------
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');
```

```
DESCRIBE EXTENDED managed_custom
```

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

Figure 2-18. 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 2-19. 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 their entries from the Hive metastore. You can easily confirm this in the Catalog Explorer. Furthermore, this action on the managed table 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 2-20).

```
%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 2-20. 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.

In conclusion, 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.

Setting Up Delta Tables

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 statements 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 automati-

cally infer schema information from the query results, eliminating the need for manual schema declaration.

CTAS statements in Databricks offer a convenient means to perform simple 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 vs. CTAS

Table 2-2 summarizes the differences between regular CREATE TABLE statements and CTAS (Create Table As Select) statements.

Table 2-2. Comparison of CREATE TABLE and CTAS 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`
Schema Declaration	Supports manual schema declaration	Does not support manual schema declaration. It automatically infer schema
Populating Data	Creates an empty table; an INSERT INTO statement is required	The table is created with data

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, necessitating an additional step of loading data into the table using INSERT INTO statements. 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, and

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. 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 Clone

Deep clone 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 datasets.

Shallow Clone

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 clone or shallow clone, 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 2-21 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.

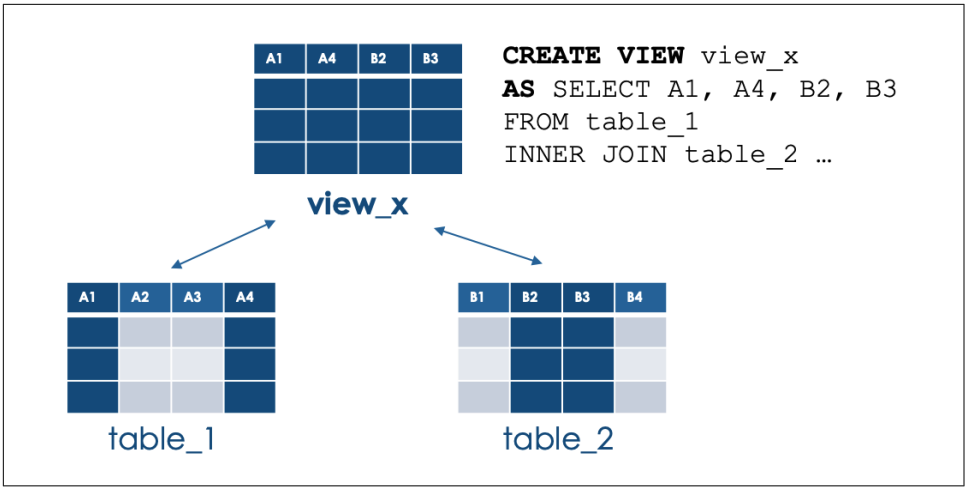


Figure 2-21. Illustration of a view object on top of two tables

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 ID, model, brand, and release year of the cars.

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.

```
          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)
SHOW TABLES
```

database	tableName	isTemporary
default	cars	false

Figure 2-22. The output of the SHOW TABLES command

Figure 2-22 displays the output of the SHOW TABLES command. As observed, we have a table named "cars" in the default database.

View Types

There are three types of views available in Databricks: Stored View, 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 using 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 indicated in the "isTemporary" column in Figure 2-23.

database	tableName	isTemporary
default	cars	false
default	view_tesla_cars	false

Figure 2-23. 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;
```

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 2-24. The result of querying the 'view_tesla_cars' stored view

Figure 2-24 displays the result of querying the 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 executed against the source table, in this case, the 'cars' table.

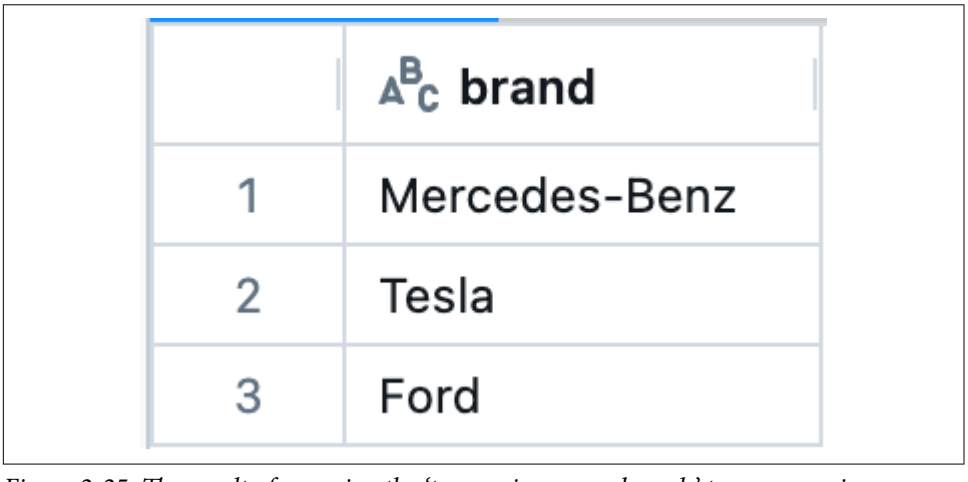
Temporary Views

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 2-25).

```
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 2-25. 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 2-26. The 'isTemporary' column indicates its temporary nature. In addition, since it's a temporary object, it is not persisted to any database, as indicated in the 'database' column.

database	tableName	isTemporary
default	cars	false
default	view_tesla_cars	false
	temp_view_cars_brands	true

Figure 2-26. 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:

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

- 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;
```

database	tableName	isTemporary
default	cars	false
default	view_tesla_cars	false

Figure 2-27. The output of the SHOW TABLES command in a new Spark session

Figure 2-27 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.

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;
```

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 2-28. The result of querying the the global temporary view

Figure 2-28 displays the result of querying the global temporary view, showing the latest entries from our 'cars' tables.

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 as 'global_temp'.

```
SHOW TABLES IN global_temp;
```

database	tableName	isTemporary
global_temp	global_temp_view_recent_cars	true
	temp_view_cars_brands	true

Figure 2-29. The output of the SHOW TABLES command in the global_temp database

In Figure 2-29, 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.

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 2-30).

database	tableName	isTemporary
global_temp	global_temp_view_recent_cars	true

Figure 2-30. 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 2-3 summarizes the differences between these three types of views:

Table 2-3. Comparison of view types

	(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

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, with their definitions stored in the database, but not the data itself. Remember, a view essentially represents a SQL query. Since stored views are saved in the database, they can be accessed across multiple sessions and clusters.

Temporary views, in contrast, are only accessible 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 also 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.

In summary, 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.