

# Chapter 2. Managing Data with Delta Lake

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 powers 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, table optimization, and vacuum operations.

## 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 as 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 transaction support in a data lake. ACID stands for atomicity, consistency, isolation, and durability, and represents fundamental rules that ensure operations on data are reliably executed, as in traditional databases.

This absence led to issues such as partially committed data and corrupted files, ultimately affecting the overall reliability of the data stored in the lake.

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 is. 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](#). To put it all together, we can visualize these concepts through an illustrative graph. In [Figure 2-1](#), we highlight the key elements that constitute the Delta Lake technology.

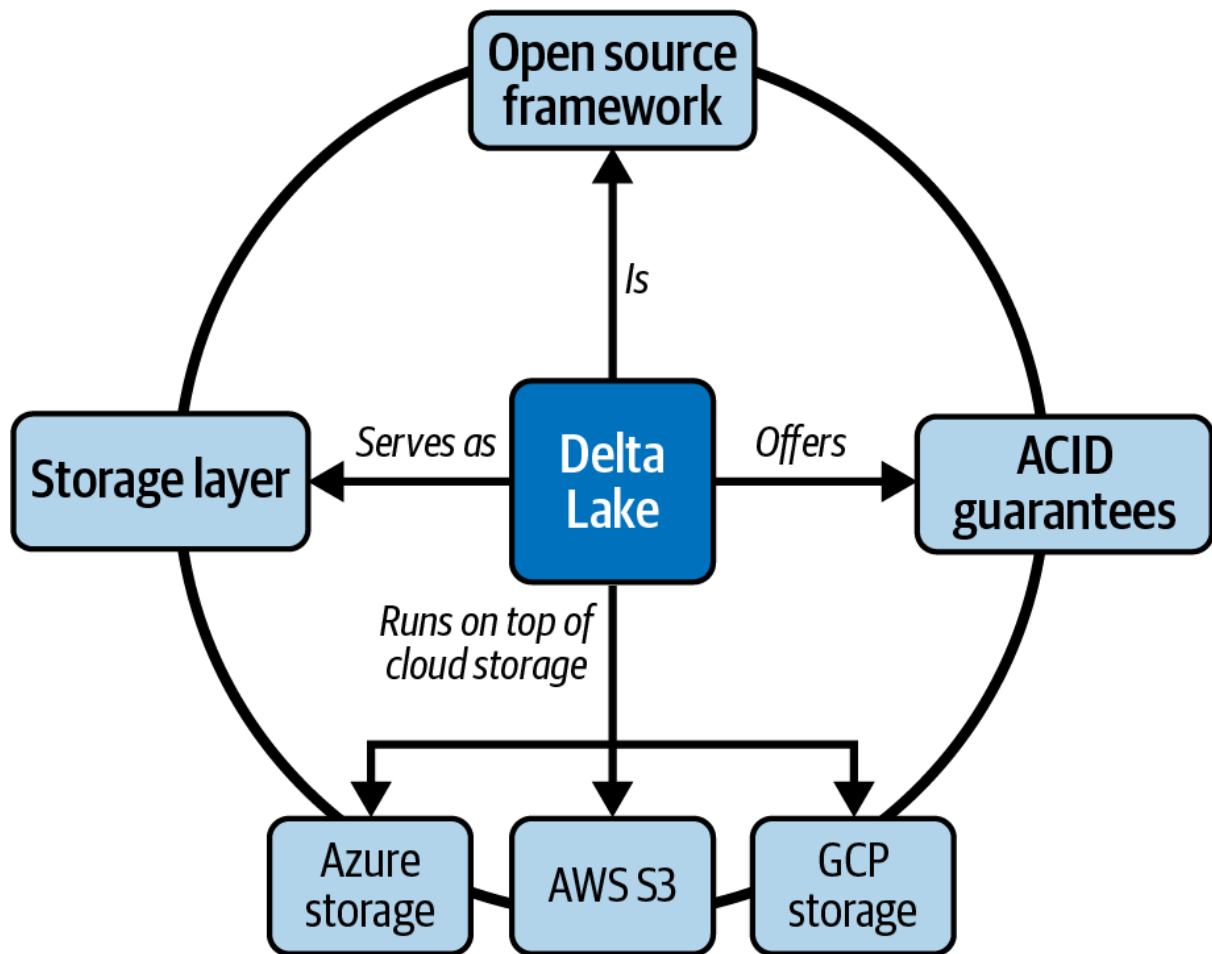


Figure 2-1. Delta Lake technology

## 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 2-2](#).

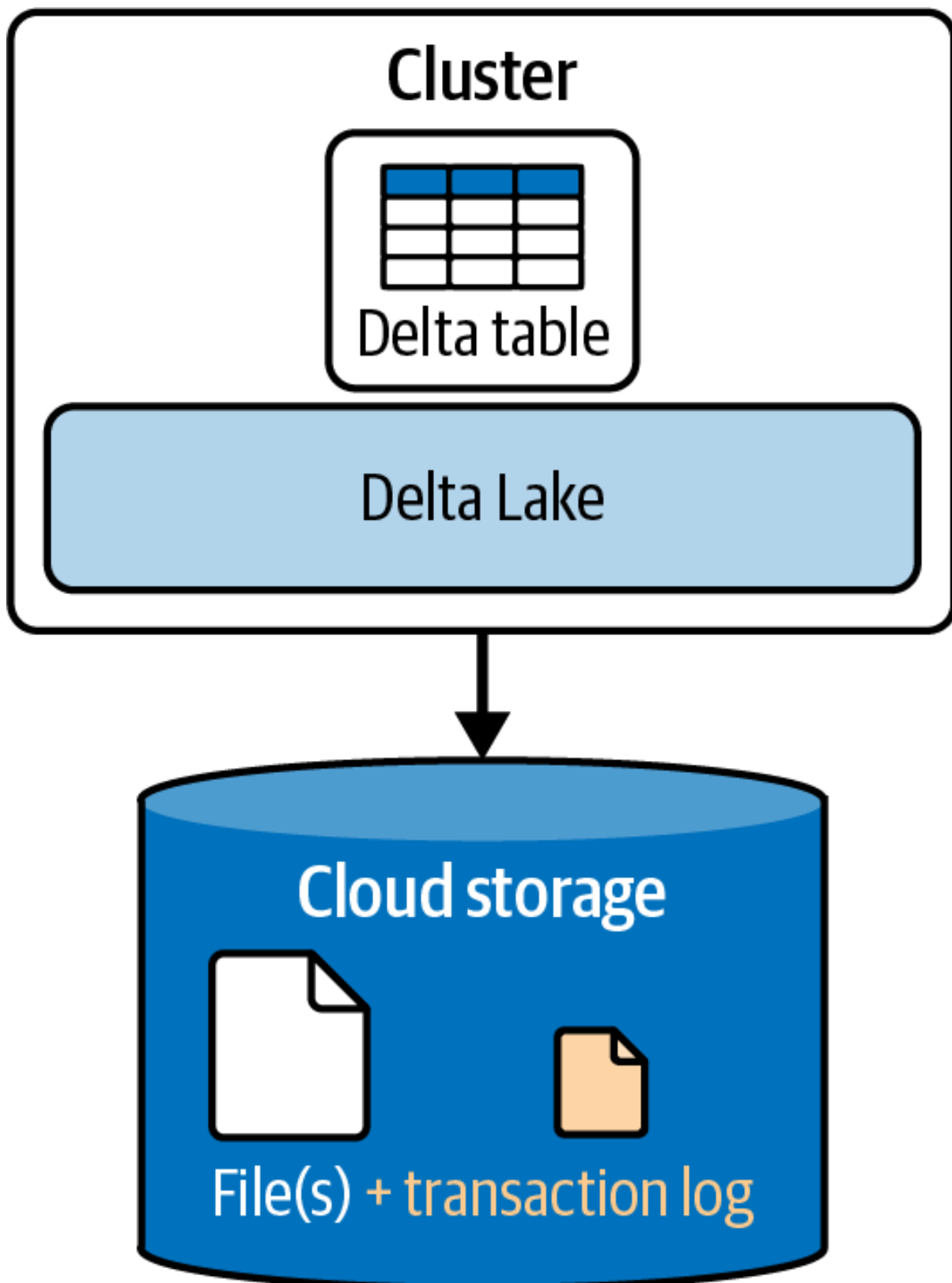


Figure 2-2. Delta Lake tables creation

The Delta Lake transaction log, often referred to as the 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 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 writing and reading, data updating, concurrent writes and reads, and, lastly, failed write attempts. Let's discuss them in detail one by one.

### Writing and reading scenario

In this first scenario, we will examine how data is written to and read from a Delta Lake table by Alice and Bob.

#### *Write operation by Alice*

Alice initiates this scenario by creating the Delta table and populating it with data, as illustrated in [Figure 2-3](#). 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` subdirectory. 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.

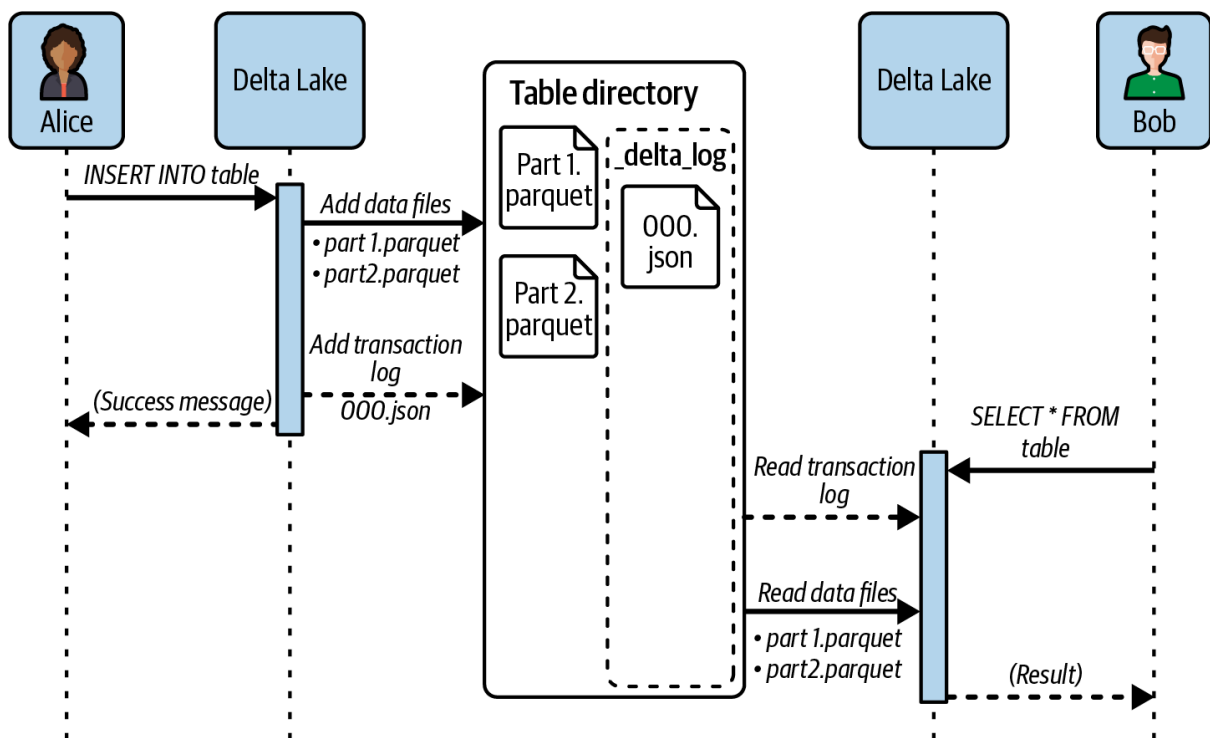


Figure 2-3. Writing and reading scenario

### Read operation by Bob

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 metadata 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 2-4](#). However, since Parquet files are immutable—meaning their contents cannot be changed after they are written—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 transaction record

(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.

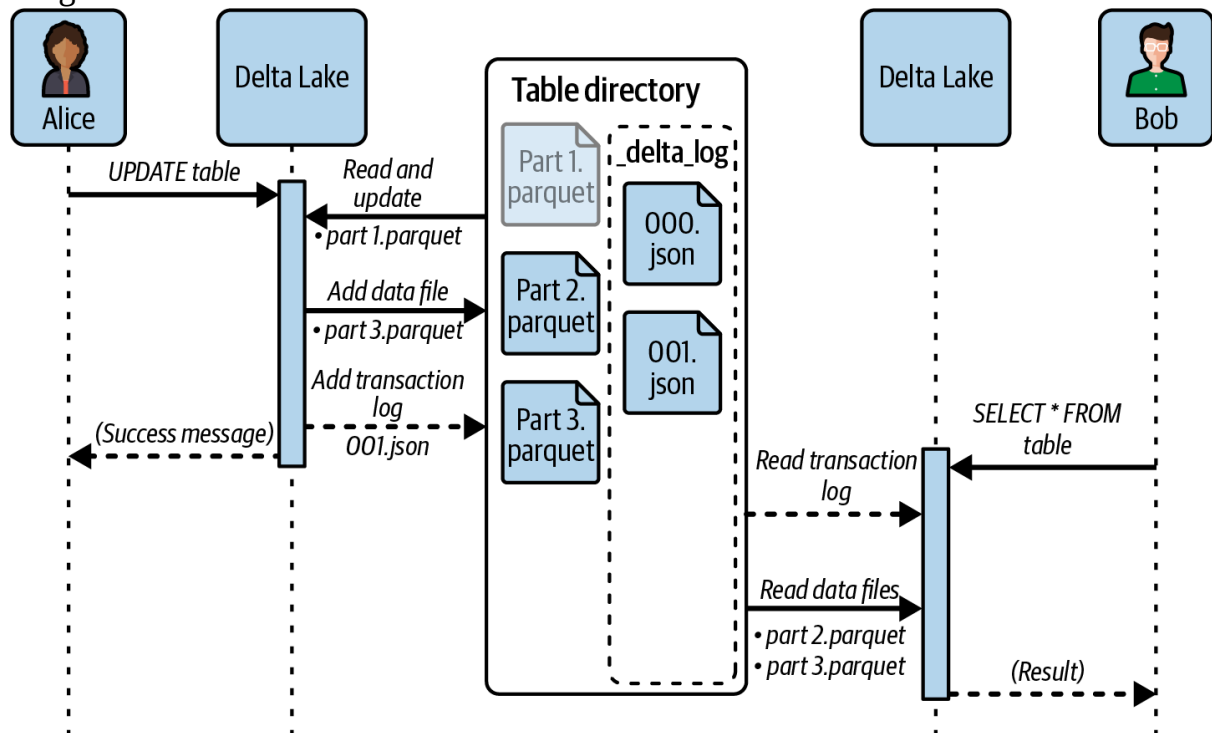


Figure 2-4. Updates scenario

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 and reads scenario

In this scenario, Alice and Bob are both interacting with the table simultaneously, as illustrated in [Figure 2-5](#). 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.

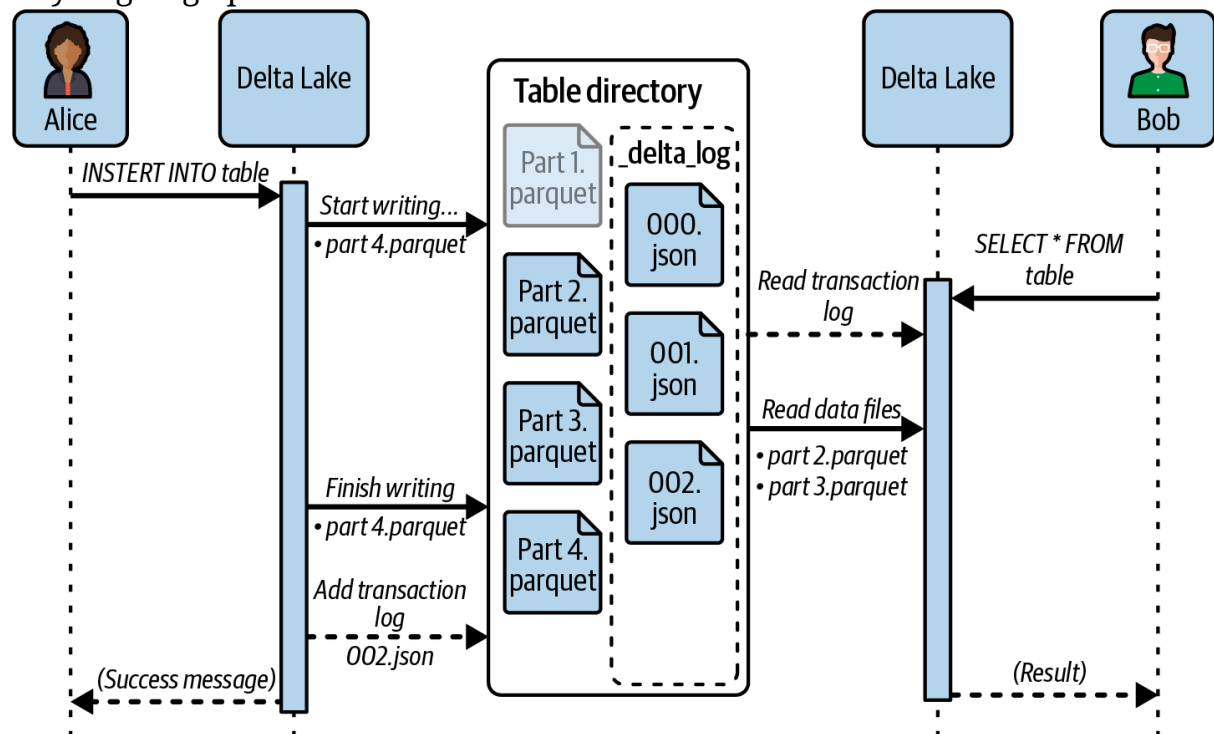


Figure 2-5. Concurrent writes and reads scenario

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

In summary, Delta Lake's transaction log helps avoid conflicts between write and read operations on the table. So, even when write and read operations are occurring simultaneously, read operations can proceed without waiting for the writes 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 2-6](#). 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 module from recording any information related to this incomplete file in the transaction log.



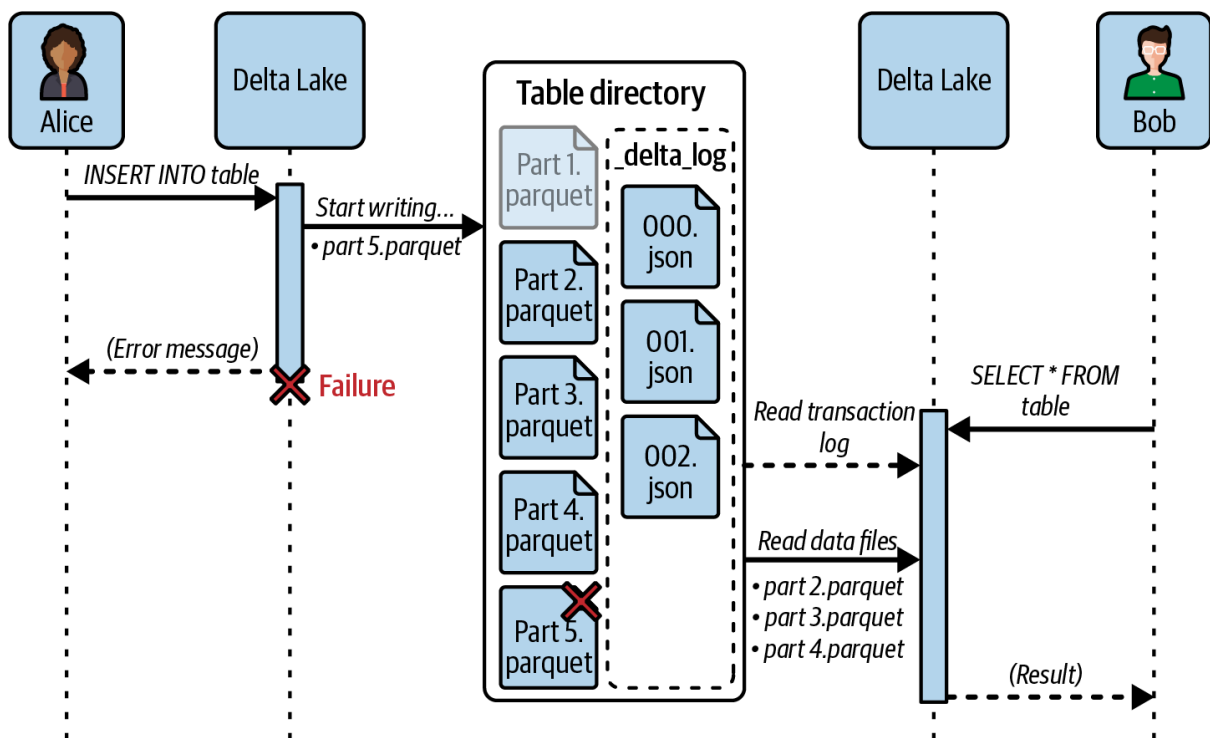


Figure 2-6. Failed writes scenario

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. Later in this chapter, we will explore how uncommitted and unused data files in the table directory can be cleaned up using vacuum operations.

## 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 instead of a centralized metastore. This strategy enhances query performance when it comes to listing large directories and reading vast amounts of data. It also includes table statistics to accelerate operations.

#### *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.

## 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 sake of simplicity and ease of storage access, we will use the `hive_metastore` catalog, which is available by default in every Databricks workspace. A detailed discussion on data catalogs will be provided in the next chapter. For the present, let us proceed by executing the following command to set the active catalog in our notebook 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, as shown in [Figure 2-7](#). To open the Catalog Explorer, click the Catalog tab in the left sidebar of your Databricks workspace.

**Catalog Explorer** [Send feedback](#) + Add ✓ Demo Cluster 14 GB, 4 Cores

Type to filter

- hive\_metastore
- default
- product\_info**
- samples

default >

**default.product\_info** [Use with BI tools](#) Create

Owner: Not set [Size: Unknown](#) Last Updated: 1 hour ago

Comment:

Columns Sample Data Details Permissions History

Column	Type	Comment
product_id	int	<a href="#">+</a>
product_name	string	<a href="#">+</a>
category	string	<a href="#">+</a>
price	double	<a href="#">+</a>
quantity	int	<a href="#">+</a>

Figure 2-7. Catalog Explorer interface

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

## Inserting Data

In Delta Lake, data insertion can be easily achieved through the use of the standard SQL `INSERT INTO` statement, as defined by ANSI SQL. 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 previous 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 wherein only the result of the last command executed within the cell is typically displayed.

#### TIP

To view the outcomes of individual SQL statements when having multiple commands in a single cell, select each specific SQL statement separately and press 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:

```
SELECT * FROM product_info
```

[Figure 2-8](#) displays the result of the `SELECT` query on the `product_info` table. It displays the four inserted records, confirming that the two transactions were successfully performed on the table.

<sup>1</sup> <sub>3</sub> product_id	<sup>A</sup> <sub>C</sub> product_name	<sup>A</sup> <sub>C</sub> category	<sup>1</sup> <sub>2</sub> price	<sup>1</sup> <sub>3</sub> 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 2-8. The result of the `SELECT` statement from the `product_info` table

Like in SQL, you can also filter data based on conditions using the `WHERE` clause and aggregate information if needed with `GROUP BY`.

## Exploring the 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
```

[Figure 2-9](#) 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.

A <sup>B</sup> <sub>C</sub> name	A <sup>B</sup> <sub>C</sub> location	1 <sup>2</sup> <sub>3</sub> numFiles	1 <sup>2</sup> <sub>3</sub> sizeInBytes
spark_catalog.default.product_info	dbfs:/user/hive/warehouse/product_info	2	338

Figure 2-9. The output of the DESCRIBE DETAIL command on the product\_info table

Additionally, the previous command 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.

#### WARNING

If you are working with Unity Catalog, be aware that tables will be stored in a different location marked by \_\_unitystorage, which is fully managed by Unity Catalog. This means that certain commands, such as listing table contents, may not function correctly due to restricted access to this managed storage. Therefore, in the context of this book, it is advisable to switch to the Hive metastore using the command USE CATALOG hive\_metastore.

To gain a deeper understanding of the table's file structure, you can use the %fs ls magic command that allows you to explore the contents of the table directory:

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

Figure 2-10 illustrates the result of executing the %fs command. It shows that the table directory indeed holds two data files, both in Parquet format.

	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 2-10. The output of the %fs command on the product\_info table directory

Furthermore, the table directory contains the \_delta\_log subdirectory, which contains the transaction 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 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 2-11).

	A <sup>B</sup> <sub>C</sub> name	1 <sup>2</sup> <sub>3</sub> size	1 <sup>2</sup> <sub>3</sub> 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 2-11. 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 to display the table metadata following this update, as illustrated in [Figure 2-12](#).

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

Figure 2-12. The output of the `DESCRIBE DETAIL` command after the update operation

The `numFiles` column 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 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](#).

## Exploring Table History

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

```
DESCRIBE HISTORY product_info
```

[Figure 2-13](#) illustrates the table history, revealing four distinct versions starting from the table creation at version 0. 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.





 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 2-13. The output of the `DESCRIBE HISTORY` command on the `product_info` table

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

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



[Figure 2-14](#) 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<sup>1</sup> files (having the `.crc` extension). Each JSON 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.

<sup>A</sup> <sub>C</sub> name	<sup>1</sup> <sub>2</sub> size	<sup>1</sup> <sub>2</sub> modificationTime
000000000000000000000000.crc	2220	1708793932000
000000000000000000000000.json	1224	1708793922000
000000000000000000000001.crc	2926	1708793940000
000000000000000000000001.json	1282	1708793937000
000000000000000000000002.crc	3619	1708793942000
000000000000000000000002.json	1274	1708793941000
000000000000000000000003.crc	3632	1708794054000
000000000000000000000003.json	1927	1708794052000

Figure 2-14. The output of the `%fs` command on the `_delta_log` folder

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 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-c000.snappy.parquet",
  "modificationTime": 1708794052000, ...}
}
{ "remove":{"path": "part-00000-485c4e80-678f-4c03-c000.snappy.parquet",
  "deletionTimestamp": 1708794052717, ...}
}
```

The output of the `%fs head` command 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 travel is a feature in Delta Lake that allows you to retrieve previous versions of data in Delta Lake tables. The key aspect of Delta Lake time travel is the automatic versioning of the table. This versioning provides an 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 result set 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` table:

```
DESCRIBE HISTORY product_info
```

[Figure 2-15](#) displays the table history, illustrating how Delta Lake's versioning system automatically assigns a unique version number and timestamp to every operation performed on a table.

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 2-15. The output of the `DESCRIBE HISTORY` command on the `product_info` table

Our table has currently four distinct versions:

### 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, which represents the latest state of the table. Additionally, note that this update operation includes the predicate used to match records in the `operationParameters` column.

## Querying Older Versions

To query older versions of a table, Delta Lake offers two distinct approaches, using either the timestamp or the 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 2-16](#).








 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 2-16. 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
```

[Figure 2-17](#) shows the result of querying version 2 of our table.






 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 2-17. The result of querying version 2 of the `product_info` 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.

## Rolling 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 it.

```
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 2-18](#)).






 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 2-18. 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
```

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


 table_size_after_restore	 num_of_files_after_restore	 num_removed_files	 num_restored_files
3384	2	0	2

Figure 2-19. The output of the `RESTORE TABLE` command on the `product_info` table  
We can also examine what really happened at our table by exploring its history:

```
DESCRIBE HISTORY product_info
```

[Figure 2-20](#) displays the table history after the restoration operation. It shows that this operation has been recorded as a new table version, labeled version 5.

version	timestamp	userName	operation	operationParameters
5	2024-02-25T00:33:33.000+00:00	Derar Alhussein	RESTORE	> {"version": "3", "timestamp": "2024-02-25T00:33:33.000+00:00"}
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 2-20. The output of the DESCRIBE HISTORY command after the restoration operation

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 specific versions 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>
```

Say 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 2-21](#), where the optimization process results in two consolidated data files instead of six small files.

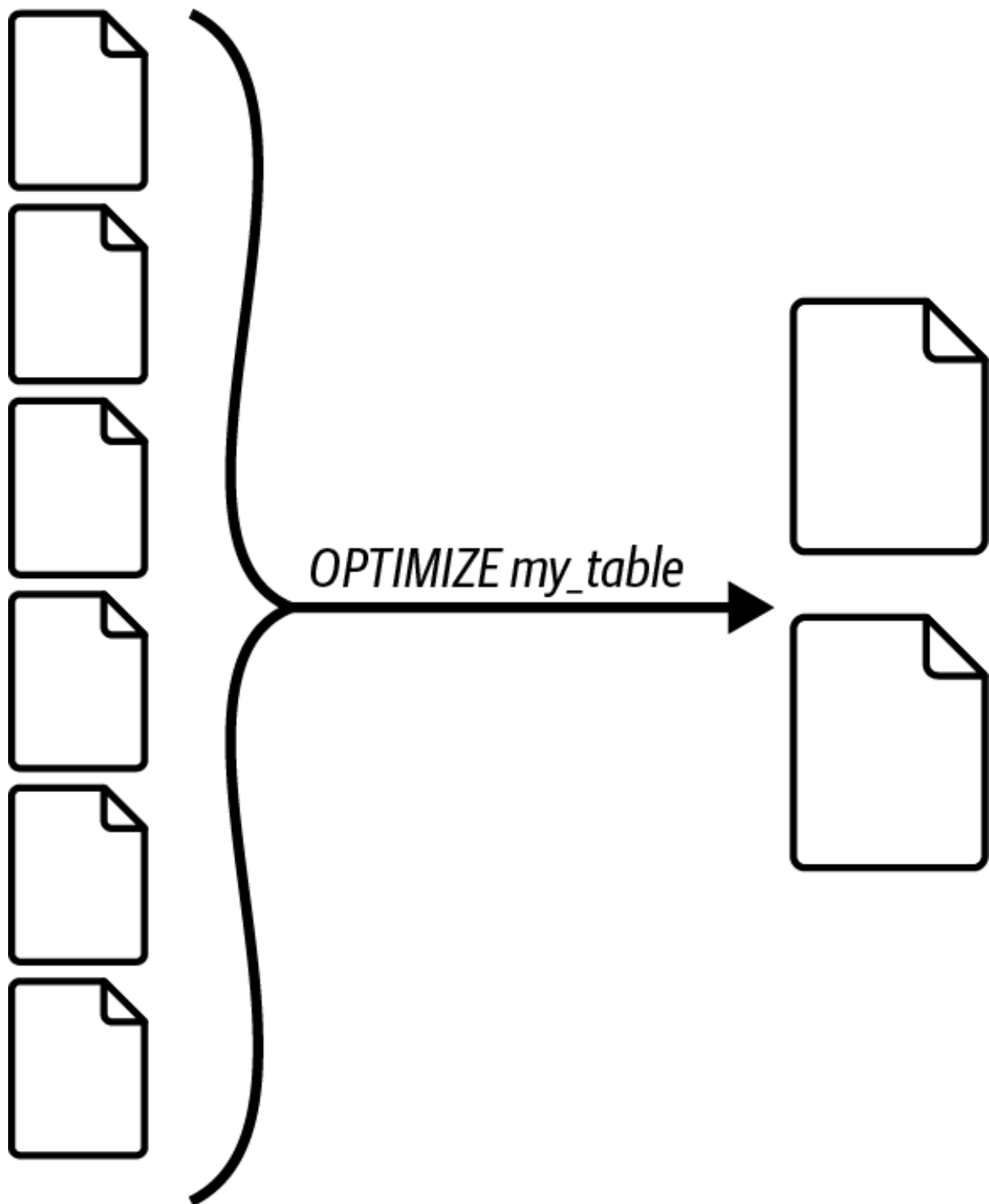


Figure 2-21. The process of optimizing Delta Lake tables using the `OPTIMIZE` command  
Table optimization improves the overall performance of the table by minimizing overhead associated with file management and enhancing the efficiency of data retrieval operations.

## 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, recalling our previous example in [Figure 2-21](#), 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 aim to have the first compacted file contain values ranging from 1 to 50, while the subsequent file contains values from 51 to 100, as illustrated in [Figure 2-22](#).

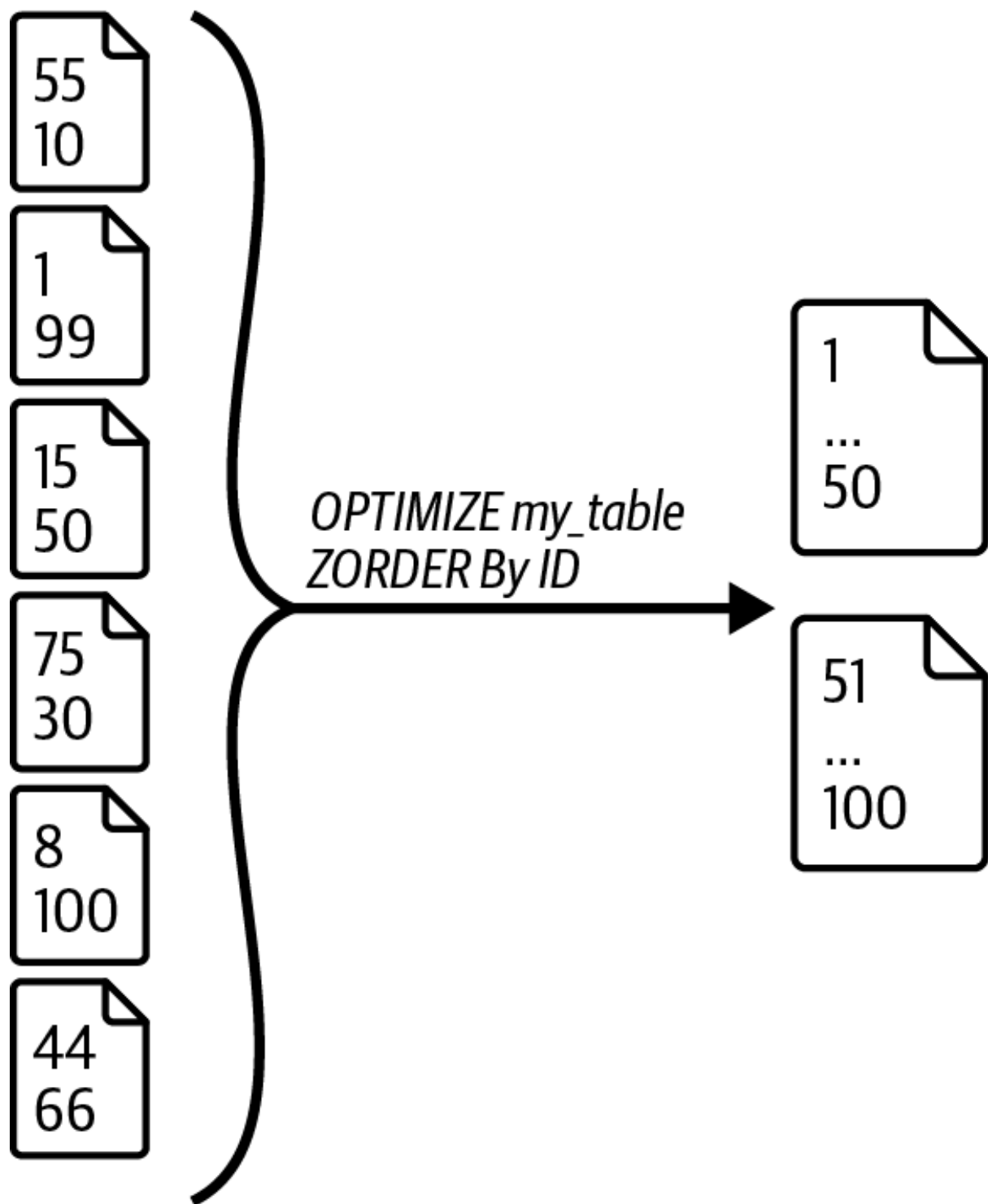


Figure 2-22. Z-Order indexing

This strategic arrangement of data enables data skipping in Delta Lake, which helps avoid unnecessary file scans during query processing. 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 two small data files, as indicated in the `numFiles` field of the table metadata (Figure 2-23):

```
DESCRIBE DETAIL product_info
```




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

Figure 2-23. The output of the `DESCRIBE DETAIL` command before optimization

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

```
OPTIMIZE product_info
ZORDER BY product_id
```

Figure 2-24 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.




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

Figure 2-24. The output of the `OPTIMIZE` command

In addition, we have added the Z-Order indexing with our `OPTIMIZE` command. As an example, 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
```

Indeed, as illustrated in Figure 2-25, the current table version consists of only one consolidated data file, indicating the success of the optimization operation.











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

Figure 2-25. The output of the `DESCRIBE DETAIL` command after optimization

In addition, we can check how the `OPTIMIZE` operation has been recorded in our table history:

```
DESCRIBE HISTORY product_info
```

As expected and illustrated in Figure 2-26, the `OPTIMIZE` command created another version of our table. This means that version 6 is the most recent version of the table.






 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":["(product_id#2123 = 3)"]}
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","par"
1	2024-02-24T16:58:57.000+00:00	Derar Alhussein	WRITE	> {"mode":"Append","statsOnLoad":"false","par"
0	2024-02-24T16:58:42.000+00:00	Derar Alhussein	CREATE TABLE	> {"partitionBy":"","description":null,"isManaged"



Figure 2-26. The output of the `DESCRIBE HISTORY` command after the optimization operation  
Lastly, let us explore the data files in our table directory:

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

In [Figure 2-27](#), we can see that there are four 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.

<b>A<sup>B</sup><sub>C</sub> name</b>	<b>i<sup>2</sup><sub>3</sub> size</b>	<b>i<sup>2</sup><sub>3</sub> modificationTime</b>
_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 2-27. The output of the `%fs` command on the `product_info` table directory after optimization  
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 seven days, meaning that the vacuum operation will prevent you from deleting files less than seven 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 our table directory:

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

As shown in [Figure 2-28](#), there are currently four data files in the table directory.

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 2-28. The output of the `%fs` command on the `product_info` table directory before vacuuming. Remember, 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 seven 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 corrupt  
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 seven 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 a `0 HOURS` retention period. To confirm its output, let's explore the table directory:

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

Indeed, as illustrated in [Figure 2-29](#), the operation this time successfully removed the old data files from the table directory.

<sup>A</sup> <sub>C</sub> name	<sup>1</sup> <sub>3</sub> size	<sup>1</sup> <sub>3</sub> modificationTime
_delta_log/	0	1708793922000
part-00000-82ca5e99-c61f-44b7-80d4-d66f3e3f72b8-c000.snappy.parquet	1745	1708829074000

Figure 2-29. The output of the `%fs` command on the `product_info` table directory after vacuuming. 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 previous `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/product_info
```

Thus, 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.

## Conclusion

Throughout this chapter, we've explored how Delta Lake works, demonstrating its essential role in transforming traditional data lakes into reliable lakehouses. By mastering Delta Lake, you can significantly enhance your data workflows and enable more robust analytics. The knowledge gained from this chapter will serve as a foundation for fully leveraging Delta Lake's potential in subsequent discussions and use cases.

## Sample Exam Questions

### Conceptual Question

1. Which of the following statements best describes the time travel feature in Delta Lake?

1. It compacts old small files into larger ones to improve query performance and optimize storage usage.
2. It partitions the table based on datetime columns, ensuring that historical data retrieval is optimized.
3. It uses Z-Order indexing to reorganize datetime column information within the same set of files, enhancing the performance of range queries.
4. It generates periodic backups of the data to ensure that all information can be easily restored in the event of system failure.
5. It allows users to query Delta Lake tables at a specific point in time, providing views of previous states of the data.

### Code-Based Question

2. A data engineer is investigating a Delta Lake table named `customer_orders`, which has experienced slow performance for the past week. The engineer has found that it contains too many small files, potentially contributing to these performance issues.

To enhance the query performance for this table, which command should the data engineer execute?

1. `ZORDER BY customer_orders`
2. `OPTIMIZE customer_orders`
3. `VACUUM customer_orders`
4. `VACUUM customer_orders RETAIN 0 HOURS`
5. `RESTORE TABLE customer_orders TO TIMESTAMP AS OF current_timestamp() - INTERVAL '7' DAYS`

The correct answers to these questions are listed in [Appendix C](#).

**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.