Chapter 4. Transforming Data with Apache Spark

The Databricks platform provides numerous transformative capabilities powered by Apache Spark. In this chapter, we will navigate through various data transformations tasks such as querying data files, writing to tables with various strategies, and performing advanced ETL operations. Moreover, we will discover the potential of higher-order functions and user-defined functions (UDFs) in Spark SQL.

Querying Data Files

Querying files in Databricks is a fundamental aspect of data exploration and analysis. In this section, we will explore the process of querying file content using SQL-like syntax. The primary mechanism for this is the `SELECT` statement, which allows us to query files directly to extract the file content.

To initiate a file query, we use the `SELECT * FROM` syntax, followed by the file format and the path to the file, as illustrated in Figure 4-1. It's important to note that the filepath is specified between backticks (`` `</path/>` ``), and not single quotes (`'</path/>'`). This distinction is essential to prevent potential syntax errors and ensure the correct interpretation of the path.

A filepath in this context can refer to a single file, or it can incorporate a wildcard character to simultaneously read multiple files. Alternatively, the path can point to an entire directory, assuming that all files within that directory adhere to the same format and schema. This flexibility is particularly advantageous when dealing with large datasets spread across multiple files.

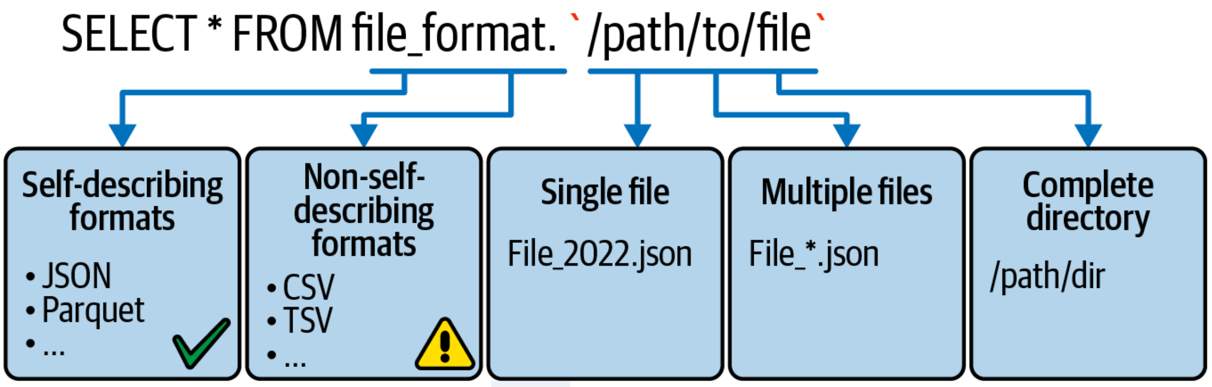


Figure 4-1. `SELECT` statement to query files

For example, when querying a JSON file located at `path/file.json`, the query would look like this:

```
SELECT * FROM json.`path/file.json`
```

We can demonstrate extracting data directly from files using a real-world dataset representing an online school environment. This dataset consists of three tables: students, enrollments, and courses, illustrated in the entity-relationship diagram shown in Figure 4-2.

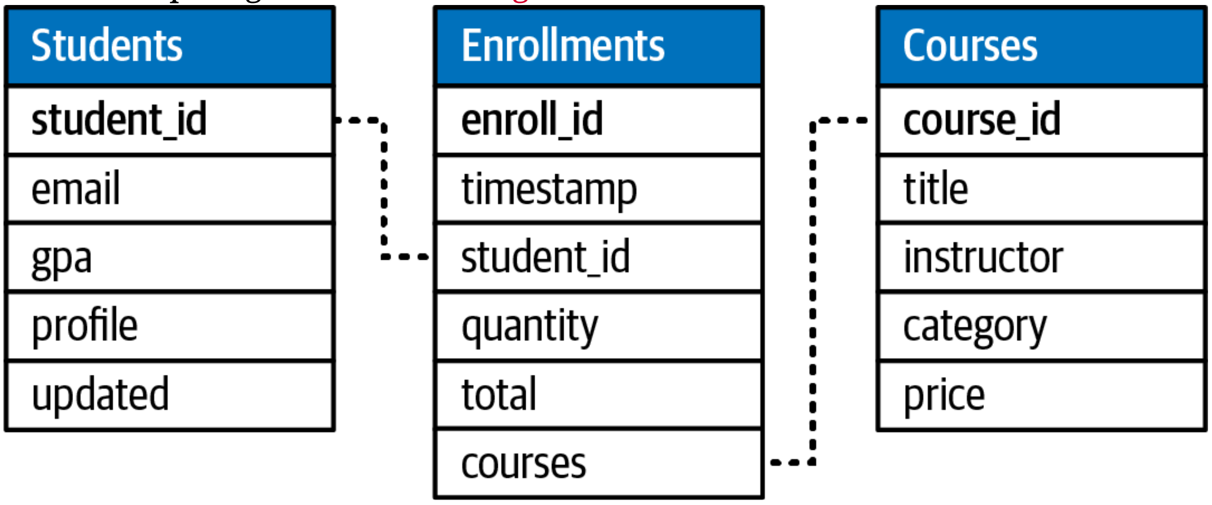


Figure 4-2. Entity-relationship diagram of the online school dataset

In this demonstration, we will use a new SQL notebook titled "4.1 - Querying Files." We begin by running a helper notebook, "School-Setup," which can be found within the *Include* subfolder in the book's GitHub repository. This helper notebook facilitates downloading the dataset to the Databricks file system and prepares the working environment accordingly:

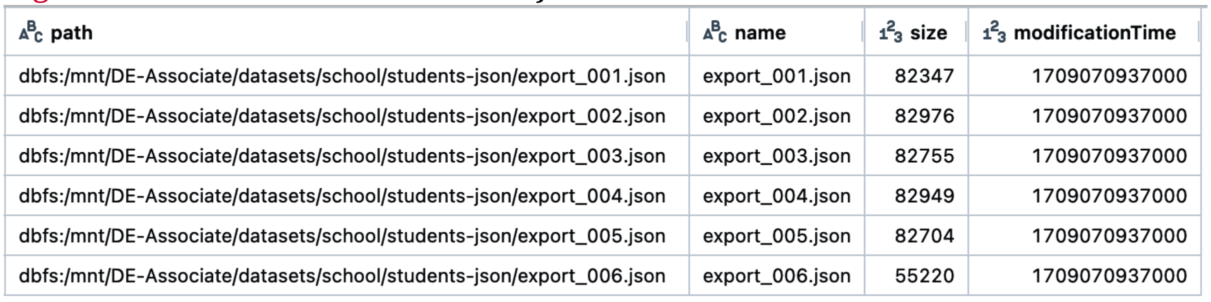
```
%run ../Includes/School-Setup
```

Querying JSON Format

The student data in this dataset is formatted in JSON. Let's review the *students* folder situated in our dataset directory. The placeholder `dataset_school` referenced in the query is a variable defined within our "School-Setup" notebook. It points to the location where the dataset files are stored on the file system:

```
%python
files = dbutils.fs.ls(f"{dataset_school}/students-json")
display(files)
```

Figure 4-3 shows that there are six JSON files in the *students* folder.



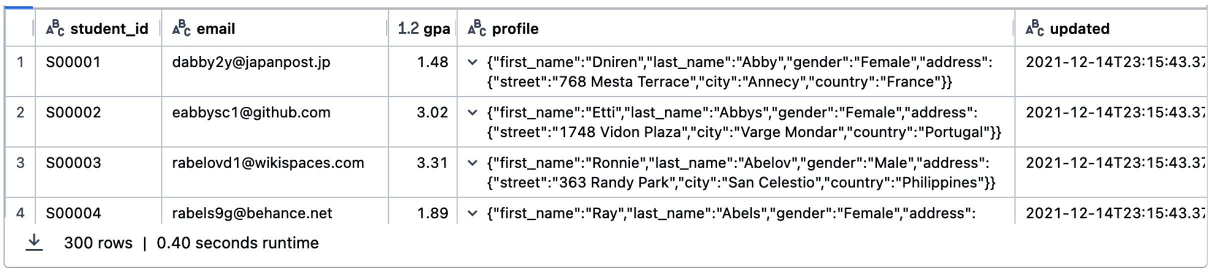
path	name	size	modificationTime
dbfs:/mnt/DE-Associate/datasets/school/students-json/export_001.json	export_001.json	82347	1709070937000
dbfs:/mnt/DE-Associate/datasets/school/students-json/export_002.json	export_002.json	82976	1709070937000
dbfs:/mnt/DE-Associate/datasets/school/students-json/export_003.json	export_003.json	82755	1709070937000
dbfs:/mnt/DE-Associate/datasets/school/students-json/export_004.json	export_004.json	82949	1709070937000
dbfs:/mnt/DE-Associate/datasets/school/students-json/export_005.json	export_005.json	82704	1709070937000
dbfs:/mnt/DE-Associate/datasets/school/students-json/export_006.json	export_006.json	55220	1709070937000

Figure 4-3. The content of the students-json directory

To read a single JSON file, the `SELECT` statement is used with the syntax `SELECT * FROM json.`, and then the full path for the JSON file is specified between backticks. In SQL, we use the `dataset.school` placeholder with the `$` character to reference the location where the dataset files are stored. This placeholder is configured in the “School-Setup” notebook:

```
SELECT * FROM json.`${dataset.school}/students-json/export_001.json`
```

The result in Figure 4-4 displays the extracted student data, including student ID, email, GPA score, profile information (in JSON format), and the last updated timestamp. As indicated, the preview display shows all 300 records from the source file.



	student_id	email	gpa	profile	updated
1	S00001	dabby2y@japanpost.jp	1.48	{"first_name":"Dniren","last_name":"Abby","gender":"Female","address":{"street":"768 Mesta Terrace","city":"Annecy","country":"France"}}	2021-12-14T23:15:43.37
2	S00002	eabbysc1@github.com	3.02	{"first_name":"Etti","last_name":"Abbys","gender":"Female","address":{"street":"1748 Vidon Plaza","city":"Varge Mondar","country":"Portugal"}}	2021-12-14T23:15:43.37
3	S00003	rabelovd1@wikispaces.com	3.31	{"first_name":"Ronnie","last_name":"Abelov","gender":"Male","address":{"street":"363 Randy Park","city":"San Celestio","country":"Philippines"}}	2021-12-14T23:15:43.37
4	S00004	rabels9g@behance.net	1.89	{"first_name":"Ray","last_name":"Abels","gender":"Female","address":	2021-12-14T23:15:43.37

300 rows | 0.40 seconds runtime

Figure 4-4. The result of querying the students data in the export_001.json file

To query multiple files simultaneously, you can use the wildcard character (`*`) in the path. For instance, you can easily query all JSON files starting with the name `export_`:

```
SELECT * FROM json.`${dataset.school}/students-json/export_*.json`
```

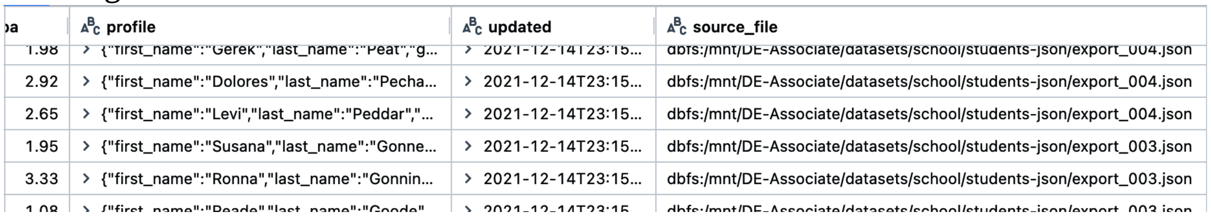
Furthermore, you can query an entire directory of files, assuming a consistent format and schema across all files in the directory. In the following query, the directory path is specified instead of an individual file:

```
SELECT * FROM json.`${dataset.school}/students-json`
```

When dealing with multiple files, adding the `input_file_name` function becomes useful. This built-in Spark SQL function records the source data file for each record. This helps in troubleshooting data-related issues by precisely pinpointing their exact source:

```
SELECT *, input_file_name() source_file
FROM json.`${dataset.school}/students-json`;
```

Figure 4-5 displays, in addition to the original columns, a new column: `source_file`. This column provides supplementary information about the origin of each record in the dataset.



pa	profile	updated	source_file
1.98	{"first_name":"Gerek","last_name":"Peat","g...	2021-12-14T23:15...	dbfs:/mnt/DE-Associate/datasets/school/students-json/export_004.json
2.92	{"first_name":"Dolores","last_name":"Pecha...	2021-12-14T23:15...	dbfs:/mnt/DE-Associate/datasets/school/students-json/export_004.json
2.65	{"first_name":"Levi","last_name":"Peddar","...	2021-12-14T23:15...	dbfs:/mnt/DE-Associate/datasets/school/students-json/export_004.json
1.95	{"first_name":"Susana","last_name":"Gonne...	2021-12-14T23:15...	dbfs:/mnt/DE-Associate/datasets/school/students-json/export_003.json
3.33	{"first_name":"Ronna","last_name":"Gonnin...	2021-12-14T23:15...	dbfs:/mnt/DE-Associate/datasets/school/students-json/export_003.json
1.08	{"first_name":"Reade","last_name":"Goode"	2021-12-14T23:15	dbfs:/mnt/DE-Associate/datasets/school/students-json/export_003.json

Figure 4-5. The result of adding the source file information to the extracted student data

Querying Using the text Format

When dealing with a variety of text-based files, including formats such as JSON, CSV, TSV, and TXT, Databricks provides the flexibility to handle them using the `text` format:

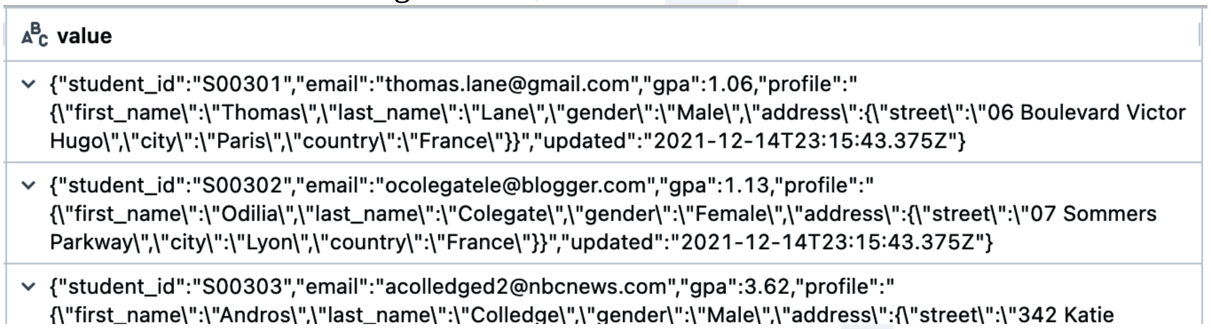
```sql
SELECT * FROM text.`path/file.txt`
```

This format allows you to extract the data as raw strings, which provide significant advantages, especially in scenarios where input data might be corrupted or contain anomalies. By extracting data as raw strings, you can leverage custom parsing logic to navigate and extract relevant values from the text-based files.

We can query our students' JSON data as raw text content using the `text` format:

```sql
SELECT * FROM text.`${dataset.school}/students-json`
```

Figure 4-6 displays the student data as raw string. Each line of the file is loaded as a record with one string column, named `value`.



```
value
{"student_id":"S00301","email":"thomas.lane@gmail.com","gpa":1.06,"profile":"
{\"first_name\":\"Thomas\",\"last_name\":\"Lane\",\"gender\":\"Male\",\"address\":{\"street\":\"06 Boulevard Victor
Hugo\",\"city\":\"Paris\",\"country\":\"France\"}}","updated":"2021-12-14T23:15:43.375Z"}
{"student_id":"S00302","email":"ocolegatele@blogger.com","gpa":1.13,"profile":"
{\"first_name\":\"Odilia\",\"last_name\":\"Colegate\",\"gender\":\"Female\",\"address\":{\"street\":\"07 Sommers
Parkway\",\"city\":\"Lyon\",\"country\":\"France\"}}","updated":"2021-12-14T23:15:43.375Z"}
{"student_id":"S00303","email":"acolledged2@nbcnews.com","gpa":3.62,"profile":"
{\"first_name\":\"Andros\",\"last_name\":\"Colledge\",\"gender\":\"Male\",\"address\":{\"street\":\"342 Katie
```

Figure 4-6. The result of querying the student data in `text` format

With this result, you can easily apply custom parsing or transformation techniques to extract specific fields, correct anomalies, or reformat the data as needed for subsequent analysis.

Querying Using binaryFile Format

Moreover, there are scenarios where the binary representation of file content is essential, such as when working with images or unstructured data. In such cases, the `binaryFile` format is suited for this task:

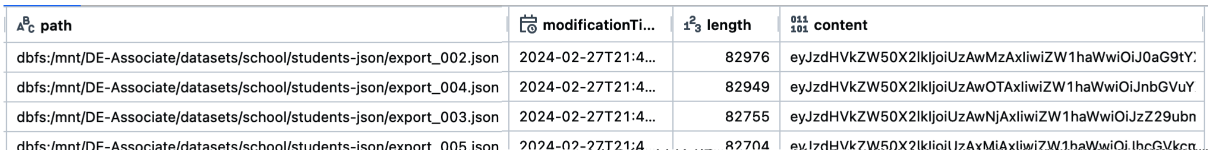
```sql
SELECT * FROM binaryFile.`path/sample_image.png`
```

In the sample query provided, the `binaryFile` format is employed to query an image file (`sample_image.png`), allowing you to work directly with the binary representation of the file's content.
We can use the `binaryFile` format to extract the raw bytes and some metadata information of the student files:

```sql
SELECT * FROM binaryFile.`${dataset.school}/students-json`
```

As shown in Figure 4-7, the output of this query provides the following details about each source file:

- `path` provides the location of the source file on the storage.
- `modificationTime` gives the last modification time of the file.
- `length` indicates the size of the file.
- `content` represents the binary representation of the file.



path	modificationTi...	length	content
dbfs:/mnt/DE-Associate/datasets/school/students-json/export_002.json	2024-02-27T21:4...	82976	eyJzdHVkZW50X2lkIjoiUzAwMzAxliwiZW1haWwiOiJ0aG9tY
dbfs:/mnt/DE-Associate/datasets/school/students-json/export_004.json	2024-02-27T21:4...	82949	eyJzdHVkZW50X2lkIjoiUzAwOTAxliwiZW1haWwiOiJnbGVuY
dbfs:/mnt/DE-Associate/datasets/school/students-json/export_003.json	2024-02-27T21:4...	82755	eyJzdHVkZW50X2lkIjoiUzAwNjAxliwiZW1haWwiOiJzZ29ubn
dbfs:/mnt/DE-Associate/datasets/school/students-json/export_005.json	2024-02-27T21:4	82704	eyJzdHVkZW50X2lkIjoiUzAxMjAxliwiZW1haWwiOiJhcGVkcm...

Figure 4-7. The result of querying the student data in binary format

So, by using the `binaryFile` format, you can access both the content and metadata of files, offering a detailed view of your dataset.

In essence, Databricks enables you to efficiently handle a wide array of data types and query them directly. Whether dealing with a single file, multiple files, or an entire directory, a simple `SELECT` statement can be used to retrieve and analyze data.

Querying Non-Self-Describing Formats

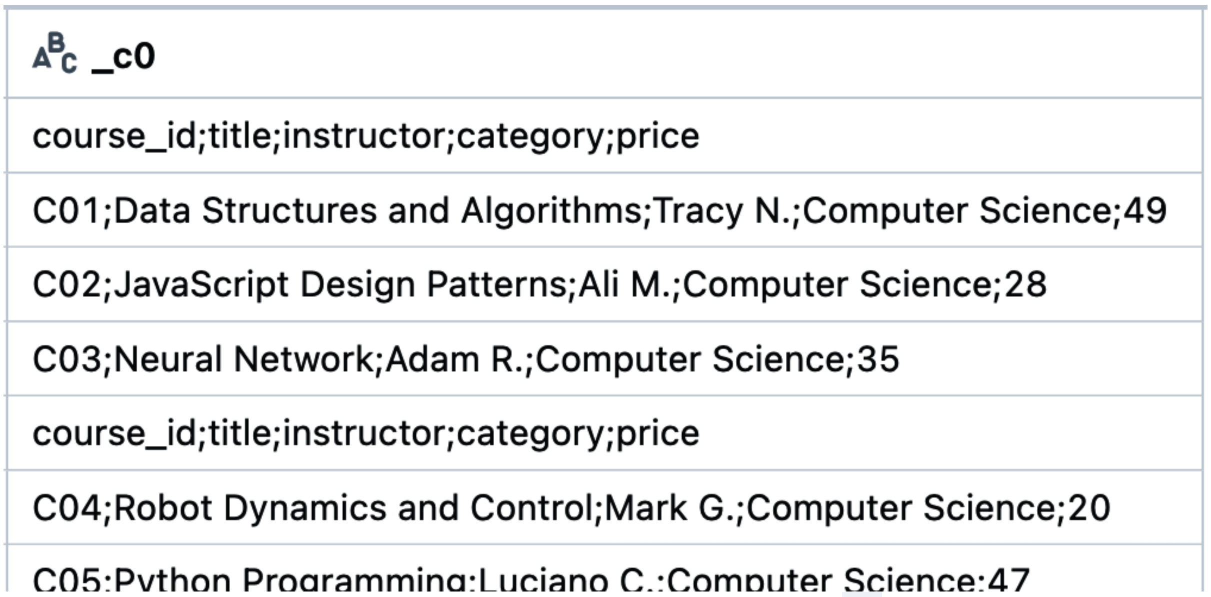
The previous querying approach is particularly effective with self-describing file formats that possess a well-defined schema, such as JSON and Parquet. By nature, these formats offer a built-in structure that makes it easy to retrieve and interpret data using `SELECT` queries.

However, when dealing with non-self-describing formats like comma-separated-value (CSV), the `SELECT` statement may not be as informative. Unlike JSON and Parquet, CSV files lack a predefined schema, making the format less suitable for direct querying. In such cases, additional steps, such as defining a schema, may be necessary for effective data extraction and analysis.

Let’s explore the result of reading the courses’ data, which is provided in CSV format. Similar to previous examples, we can try using the `SELECT` statement, but this time with the `csv` format:

```
SELECT * FROM csv.`${dataset.school}/courses-csv`
```

As shown in Figure 4-8, the output of the query is not well-parsed. The header row is extracted as a table row, and all columns are loaded into a single column, `_c0`. This behavior is explained by the delimiter—the symbol used to separate columns in the file—which, in this case, is a semicolon rather than the standard comma.



_c0
course_id;title;instructor;category;price
C01;Data Structures and Algorithms;Tracy N.;Computer Science;49
C02;JavaScript Design Patterns;Ali M.;Computer Science;28
C03;Neural Network;Adam R.;Computer Science;35
course_id;title;instructor;category;price
C04;Robot Dynamics and Control;Mark G.;Computer Science;20
C05;Python Programming;Luciano C.;Computer Science;47

Figure 4-8. The result of querying the course data in `csv` format

This issue highlights a challenge with querying files without a well-defined schema, particularly in formats like CSV. In the upcoming sections, we will learn how to address this challenge.

Registering Tables from Files with CTAS

Using CTAS (`CREATE TABLE AS SELECT`) statements allows you to register tables from files, particularly when dealing with well-defined schema sources like Parquet files. This process is crucial for loading data into a lakehouse, allowing you to take full advantage of the Databricks platform's capabilities:

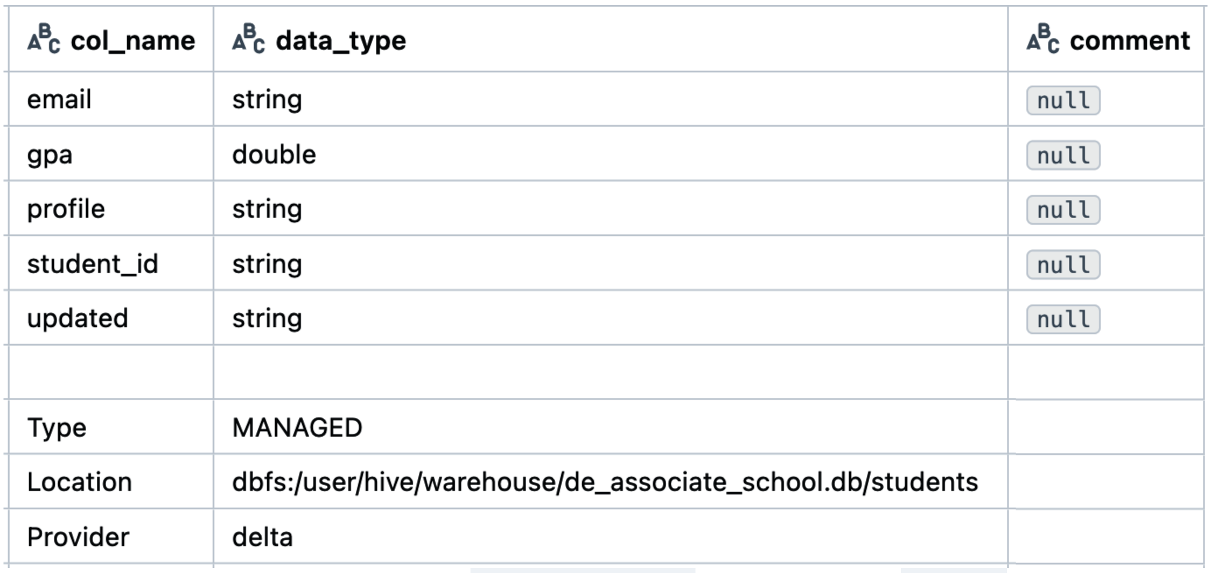
```
CREATE TABLE table_name
AS SELECT * FROM <file_format>.`/path/to/file`
```

CTAS statements simplify the process of creating Delta Lake tables by automatically inferring schema information from the query results. This eliminates the need for manual schema declaration.

In the following example, we create and populate the student data table using a CTAS statement. This ensures that the resulting table is a Delta Lake table:

```
CREATE TABLE students AS
SELECT * FROM json.`${dataset.school}/students-json`;

DESCRIBE EXTENDED students;
```

Figure 4-9 displays the metadata of our new table, `students`. The `Provider` value confirms the creation of a Delta Lake table. This means that the CTAS statement has extracted the data from the JSON files and loaded it into the `students` table in Delta format (i.e., in Parquet data files along with a Delta transaction log). Additionally, this table is identified as a managed table, as indicated by the `Type` value.



col_name	data_type	comment
email	string	null
gpa	double	null
profile	string	null
student_id	string	null
updated	string	null
Type	MANAGED	
Location	dbfs:/user/hive/warehouse/de_associate_school.db/students	
Provider	delta	

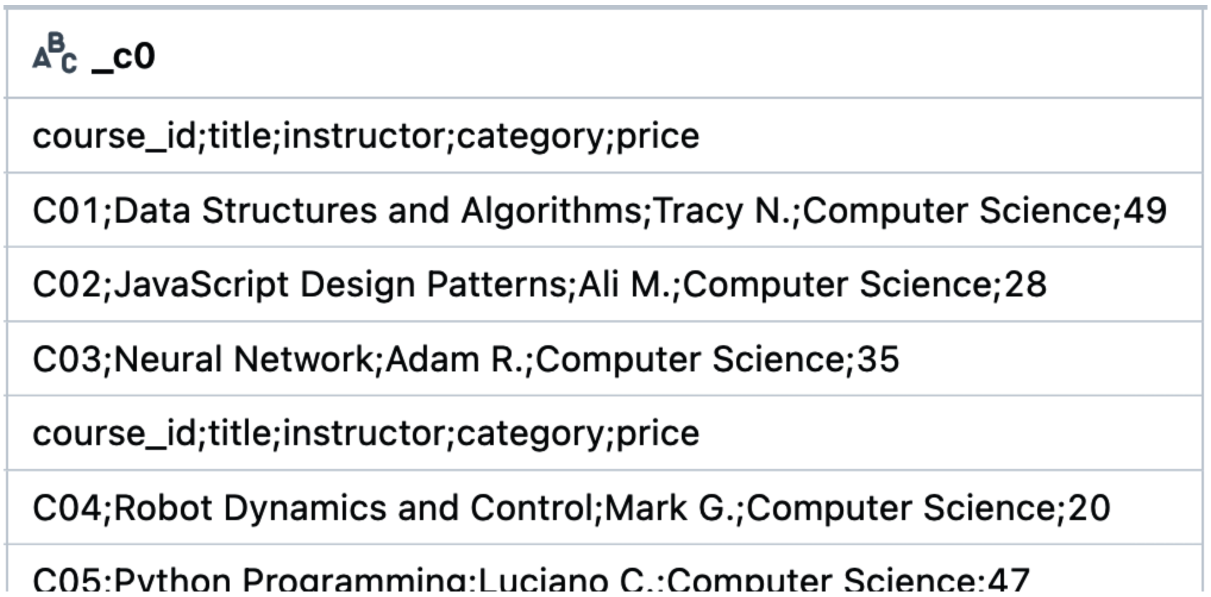
Figure 4-9. The output of the `DESCRIBE EXTENDED` command on the `students` table

Moreover, the schema has been automatically inferred from the query results, a feature common to CTAS statements. Remember, CTAS statements automatically infer schema information from the query results, making them a suitable choice for external data ingestion from sources with well-defined schemas, such as Parquet files.

However, it’s important to note that CTAS statements come with certain limitations. One significant limitation is that CTAS statements do not support specifying additional file options. This becomes a challenge when trying to ingest data from CSV files or other formats that require specific configurations:

```
CREATE TABLE courses_unparsed AS
SELECT * FROM csv.`${dataset.school}/courses-csv`;

SELECT * FROM courses_unparsed;
```

Figure 4-10 shows that we have successfully created a Delta Lake table; however, the data is not well-parsed.



_c0
course_id;title;instructor;category;price
C01;Data Structures and Algorithms;Tracy N.;Computer Science;49
C02;JavaScript Design Patterns;Ali M.;Computer Science;28
C03;Neural Network;Adam R.;Computer Science;35
course_id;title;instructor;category;price
C04;Robot Dynamics and Control;Mark G.;Computer Science;20
C05;Python Programming;Luciano C.;Computer Science;47

Figure 4-10. The output of the table created by a CTAS statement from CSV files

Typically, CSV files have delimiter or encoding options that need to be specified during the data loading process. In response to this requirement, we will now explore an alternative solution.

Registering Tables on Foreign Data Sources

In scenarios where additional file options are necessary, an alternative solution is to use the regular `CREATE TABLE` statement, but with the `USING` keyword. Unlike CTAS statements, this approach is particularly useful when dealing with formats that need specific configurations. The `USING` keyword provides increased flexibility by allowing you to specify the type of foreign data source, such as CSV format, as well as any additional files options, such as delimiter and header presence:

```
CREATE TABLE table_name
   (col_name1 col_type1, ...)
USING data_source
OPTIONS (key1 = val1, key2 = val2, ...)
LOCATION path
```

However, it's crucial to note that this method creates an external table, serving as a reference to the files without physically moving the data during table creation to Delta Lake.

Unlike CTAS statements, which automatically infer schema information, creating a table via the `USING` keyword requires you to provide the schema explicitly. So, this method offers more control over the schema definition.

Example 1: CSV

For instance, to deal with CSV files stored in an external location, the following example demonstrates the creation of a table using a CSV foreign source:

```
CREATE TABLE csv_external_table
 (col_name1 col_type1, ...)
 USING CSV
 OPTIONS (header = "true",
           delimiter = ";")
 LOCATION = '/path/to/csv/files'
```

This code sample creates an external table that points to CSV files located in the specified path. In addition, it configures the `header` option to indicate the presence of a header in the files, and the `delimiter` option is set to use a semicolon instead of the default comma separator.

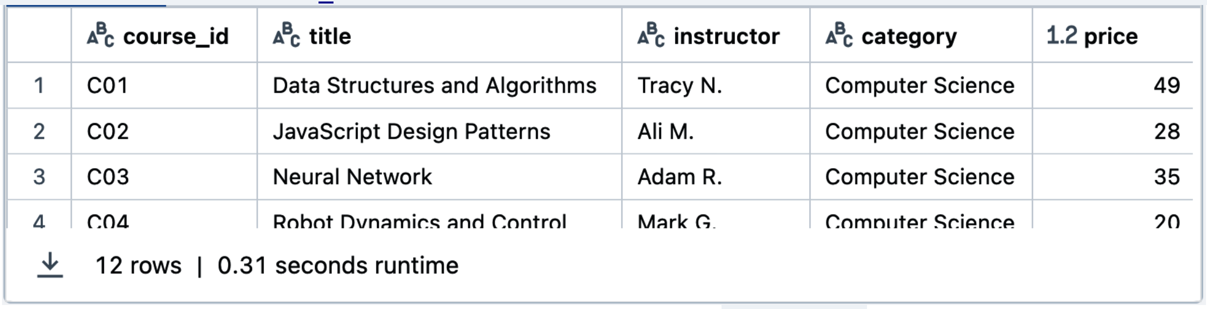
Let’s apply this method on our `courses` data:

```
1 CREATE TABLE courses_csv
2 (course_id STRING, title STRING, instructor STRING, category STRING, price
  DOUBLE)
3 USING CSV
4 OPTIONS (
5 header = "true",
6 delimiter = ";")
7 LOCATION "${dataset.school}/courses-csv"
```

In this example, the `courses_csv` table is created by specifying the CSV format as a foreign source (line 3), indicating the presence of a header in the files (line 5), defining the semicolon as the delimiter (line 6), and, lastly, specifying the location of the source files (line 7).

Once the table is created, querying it shows that we have the courses’ data in a well-structured form (Figure 4-11).

```
SELECT * FROM courses_csv
```



	course_id	title	instructor	category	price
1	C01	Data Structures and Algorithms	Tracy N.	Computer Science	49
2	C02	JavaScript Design Patterns	Ali M.	Computer Science	28
3	C03	Neural Network	Adam R.	Computer Science	35
4	C04	Robot Dynamics and Control	Mark G.	Computer Science	20

12 rows | 0.31 seconds runtime

Figure 4-11. The result of querying the `courses_csv` table

It’s essential to note that when working with CSV files as a data source, maintaining the column order becomes crucial, especially if additional data files will be added to the source directory. Spark relies on the specified order during table creation to load data and apply column names and data types correctly from CSV files. Therefore, any changes to the column order could impact the integrity of the data loading process.

Example 2: database

Another scenario where the `CREATE TABLE` statement with the `USING` keyword proves useful is when creating a table using a JDBC connection, which allows referencing data in an external SQL database. This approach enables you to establish a connection to an external database by defining necessary options

such as the connection string, username, password, and specific database table containing the data.
Here is an example of creating an external table using a JDBC connection:

```
CREATE TABLE jdbc_external_table
USING JDBC
OPTIONS (
 url = 'jdbc:mysql://your_database_server:port',
 dbtable = 'your_database.table_name',
 user = 'your_username',
 password = 'your_password'
);
```

In this example, the following apply:

- The `url` option specifies the JDBC connection string to your external database.
- The `dbtable` option indicates the specific table within the external database.
- The `user` and `password` are credentials required for authentication.

This method facilitates seamless integration of data from external SQL databases into the lakehouse environment, allowing for cross-database analysis and reporting. By creating an external table using a JDBC connection, you can access and query data from the external database without physically moving or duplicating the data.

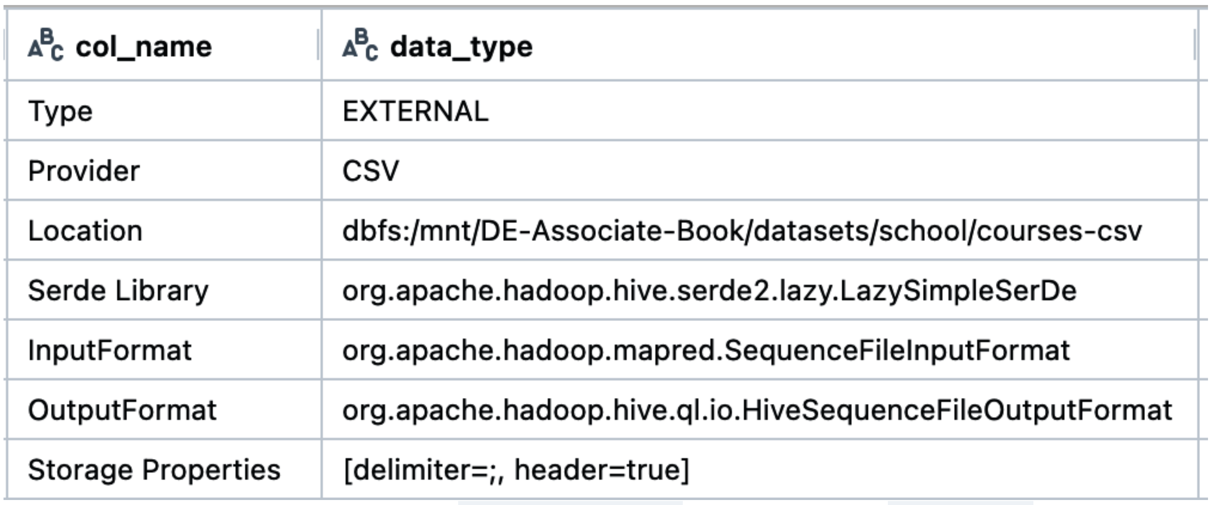
Limitation

It's crucial to be aware of the limitations associated with tables having foreign data sources—they are not Delta tables. This means that the performance benefits and features offered by Delta Lake, such as time travel and guaranteed access to the most recent version of the data, are not available for these tables. This limitation becomes especially noticeable when dealing with large database tables, potentially leading to performance issues.

Let's better understand the impact of not having a Delta table by exploring the consequences of working with an external table linked directly to CSV files. Before we start, let's review the table's type and storage details:

```
DESCRIBE EXTENDED courses_csv
```

Figure 4-12 reveals that the table is an external table, and this table is not a Delta table, as indicated in the `Provider` value. This means that no data conversion to Delta format occurred during table creation; instead, the table simply points to the CSV files stored in the external location.



col_name	data_type
Type	EXTERNAL
Provider	CSV
Location	dbfs:/mnt/DE-Associate-Book/datasets/school/courses-csv
Serde Library	org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe
InputFormat	org.apache.hadoop.mapred.SequenceFileInputFormat
OutputFormat	org.apache.hadoop.hive.ql.io.HiveSequenceFileOutputFormat
Storage Properties	[delimiter=;, header=true]

Figure 4-12. The output of the `DESCRIBE EXTENDED` command on the `courses_csv` table

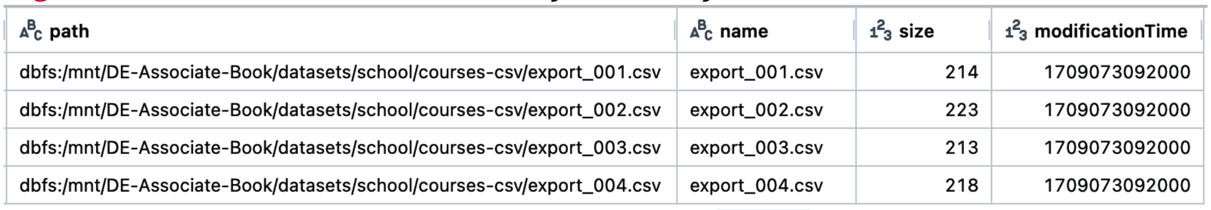
Additionally, the `Storage Properties` value captures all metadata and options specified during table creation, ensuring that data in the location is always read with these specified options.

Impact of not having a Delta table

The absence of a Delta table introduces certain limitations and impacts. Unlike Delta Lake tables, which guarantee querying the most recent version of source data, tables registered against other data sources, like CSV, may represent outdated cached data. To illustrate this, we will add new data and observe the resulting behavior of the table. First, let's check the number of files in the `courses` directory:

```
%python
files = dbutils.fs.ls(f"{dataset_school}/courses-csv")
display(files)
```

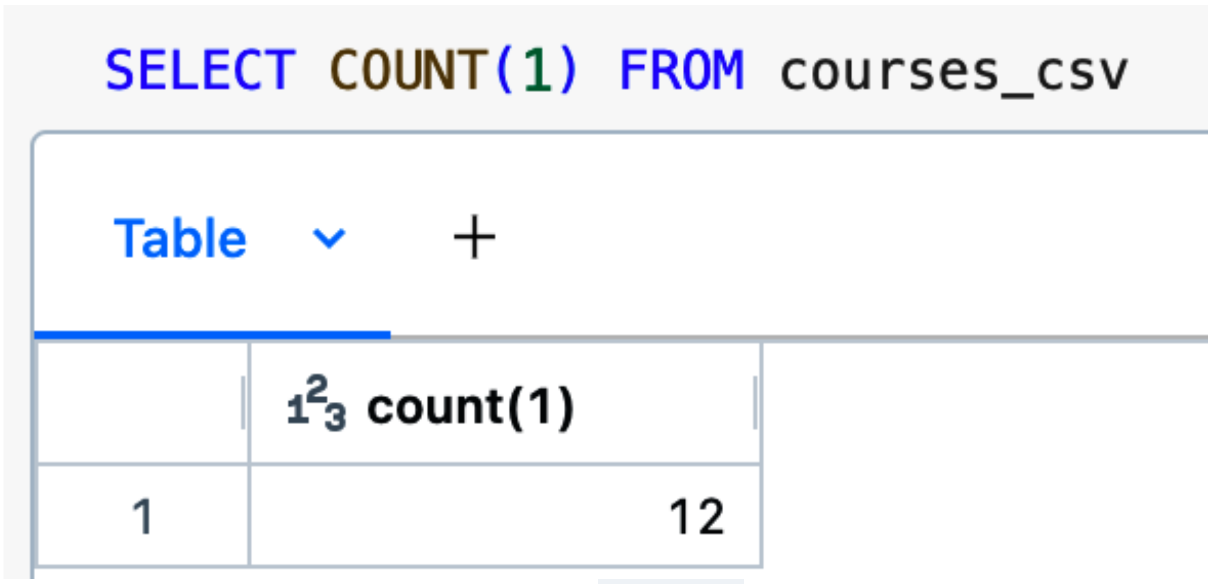
Figure 4-13 reveals that the directory currently contains four files.



path	name	size	modificationTime
dbfs:/mnt/DE-Associate-Book/datasets/school/courses-csv/export_001.csv	export_001.csv	214	1709073092000
dbfs:/mnt/DE-Associate-Book/datasets/school/courses-csv/export_002.csv	export_002.csv	223	1709073092000
dbfs:/mnt/DE-Associate-Book/datasets/school/courses-csv/export_003.csv	export_003.csv	213	1709073092000
dbfs:/mnt/DE-Associate-Book/datasets/school/courses-csv/export_004.csv	export_004.csv	218	1709073092000

Figure 4-13. The list of the files in the `courses` directory

Since each file contains three records, the table holds a total of twelve records, as shown in Figure 4-14.



```
SELECT COUNT(1) FROM courses_csv
```

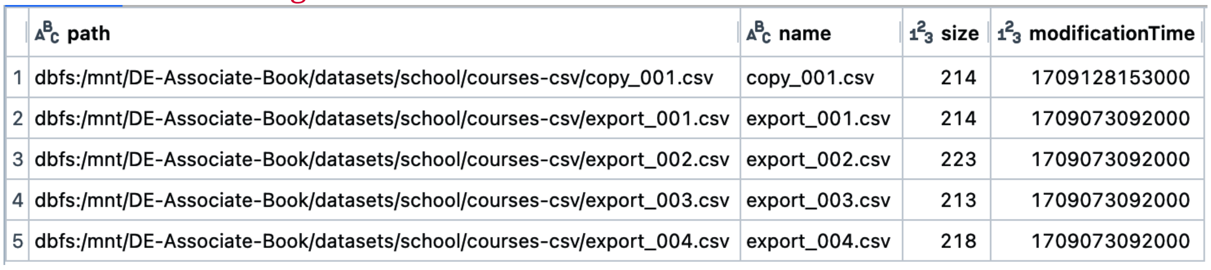
	count(1)
1	12

Figure 4-14. The number of records in the `courses_csv` table after adding the new file

Now, let's run the following Python command to duplicate and rename one of these files as `copy_001.csv`. This action simulates the ingestion of new CSV files by a source system:

```
%python
dbutils.fs.cp(f"{dataset_school}/courses-csv/export_001.csv",
              f"{dataset_school}/courses-csv/copy_001.csv")
```

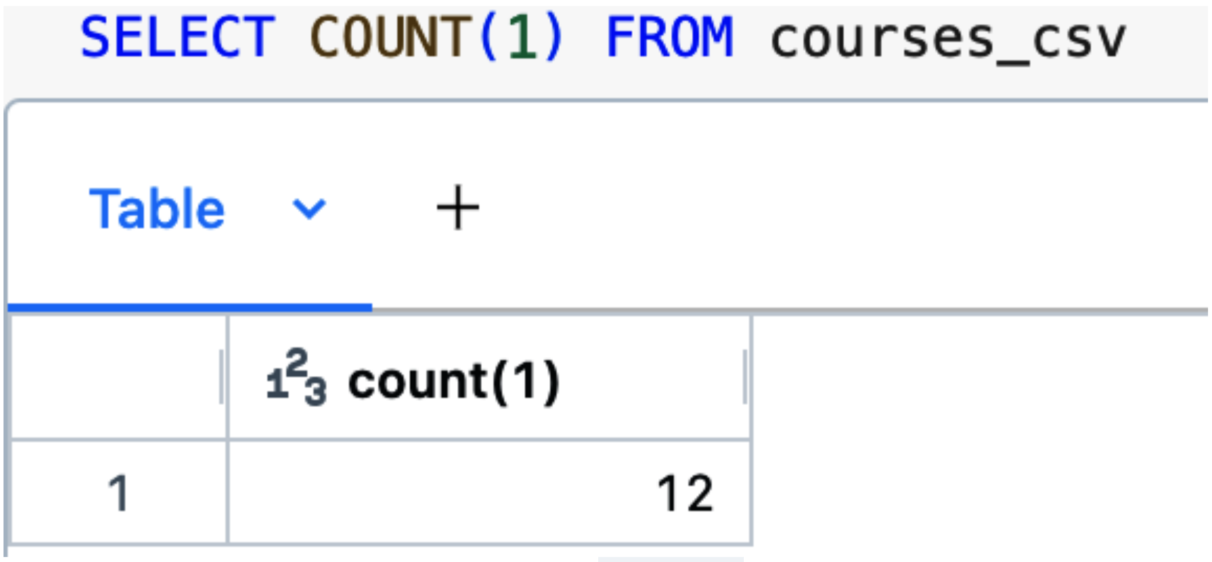
After this operation, exploring the courses directory confirms that the new file has been added (Figure 4-15).



	path	name	size	modificationTime
1	dbfs:/mnt/DE-Associate-Book/datasets/school/courses-csv/copy_001.csv	copy_001.csv	214	1709128153000
2	dbfs:/mnt/DE-Associate-Book/datasets/school/courses-csv/export_001.csv	export_001.csv	214	1709073092000
3	dbfs:/mnt/DE-Associate-Book/datasets/school/courses-csv/export_002.csv	export_002.csv	223	1709073092000
4	dbfs:/mnt/DE-Associate-Book/datasets/school/courses-csv/export_003.csv	export_003.csv	213	1709073092000
5	dbfs:/mnt/DE-Associate-Book/datasets/school/courses-csv/export_004.csv	export_004.csv	218	1709073092000

Figure 4-15. The list of the files in the courses directory after adding the new file

Despite adding new data to the directory, we notice that the table does not immediately reflect the changes from 12 to 15 records, as shown in Figure 4-16.



```
SELECT COUNT(1) FROM courses_csv
```

	count(1)
1	12

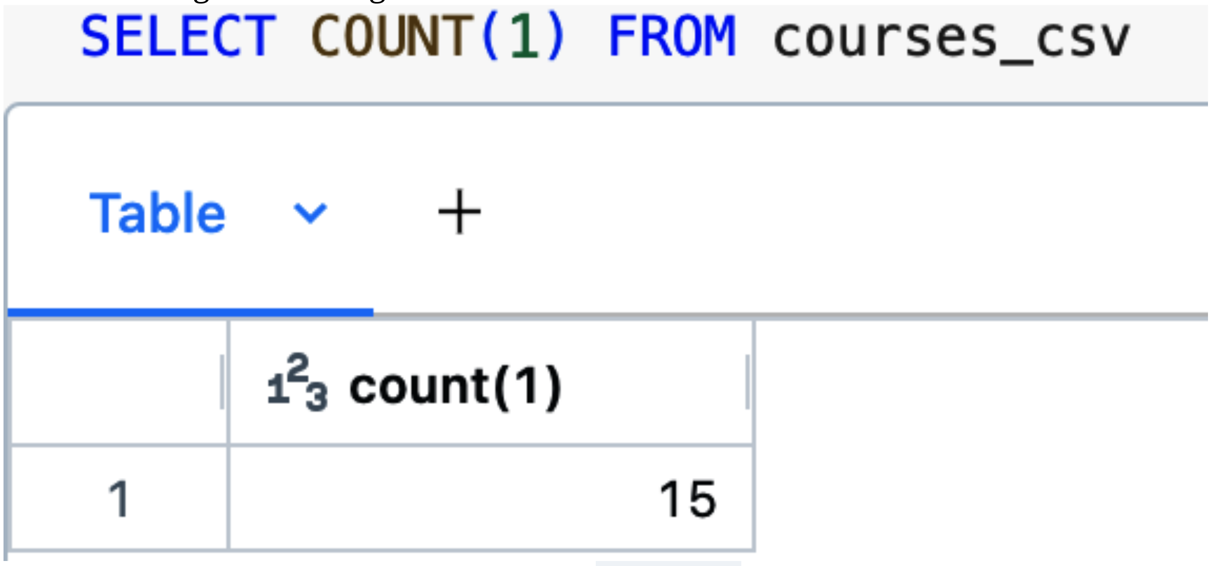
Figure 4-16. The number of records in the `courses_csv` table after adding the new file

Spark automatically caches the underlying data in local storage for better performance in subsequent queries. However, the external CSV file does not natively signal Spark to refresh this cached data. Consequently, the new data remains invisible until the cache is manually refreshed using the `REFRESH TABLE` command:

```
REFRESH TABLE courses_csv
```

However, this action invalidates the table cache, necessitating a rescan of the original data source to reload all data into memory. This process can be particularly time-consuming when dealing with large datasets.

Upon refreshing the table, querying it again retrieves the updated count, as illustrated in Figure 4-17. This confirms the need for manual cache refreshing when dealing with foreign data sources like CSV.



```
SELECT COUNT(1) FROM courses_csv
```

	count(1)
1	15

Figure 4-17. The number of records in the `courses_csv` table after refreshing the table

This observation emphasizes the trade-offs and considerations associated with choosing between Delta tables and foreign data sources when working with Databricks.

Hybrid approach

To address this limitation and leverage the advantages of Delta Lake, a workaround involves creating a temporary view that refers to the foreign data source. Then, you can execute a CTAS statement on this temporary view to extract the data from the external source and load it into a Delta table. This hybrid approach allows you to combine the benefits of external tables with the performance and features of Delta Lake.

Here’s an illustrative example of this process:

```
CREATE TEMP VIEW foreign_source_tmp_vw (col1 col1_type, ...)
 USING data_source
 OPTIONS (key1 = "val1", key2 = "val2", ..., path = "/path/to/files");

CREATE TABLE delta_table
AS SELECT * FROM foreign_source_tmp_vw
```

In this example, a temporary view is created referring to a foreign data source. Then, a Delta Lake table is created by executing a CTAS statement on the temporary view. This process moves the data into a Delta format (Parquet data files + transaction log in JSON format).

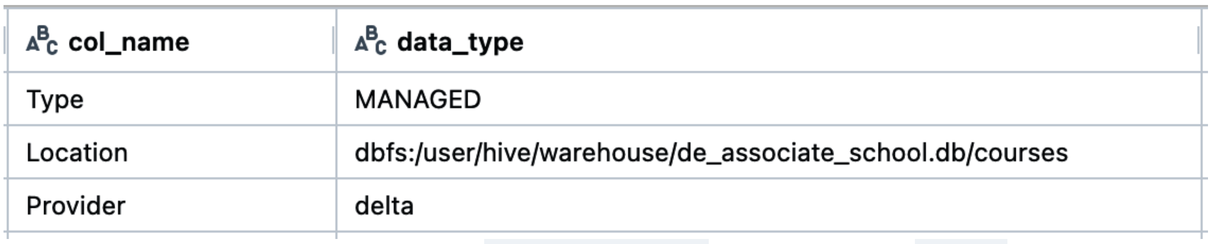
This approach highlights the flexibility of CTAS statements, as they can be employed not only to query files but also to query any object, such as a temporary view in this case.

In the same way, we can apply this approach on the course data, delivered in CSV format. We first create a temporary view and configure it to handle file options. Then, we execute a CTAS statement to make a copy of the data from the temporary view into a Delta table named `courses`:

```
CREATE TEMP VIEW courses_tmp_vw
   (course_id STRING, title STRING, instructor STRING, category STRING,
   price DOUBLE)
USING CSV
OPTIONS (
 path = "${dataset.school}/courses-csv/export_*.csv",
 header = "true",
 delimiter = ";"
);

CREATE TABLE courses AS
 SELECT * FROM courses_tmp_vw;
```

Figure 4-18 displays the metadata information of the `courses` table. It confirms that it is a Delta Lake table.

```
DESCRIBE EXTENDED courses
```

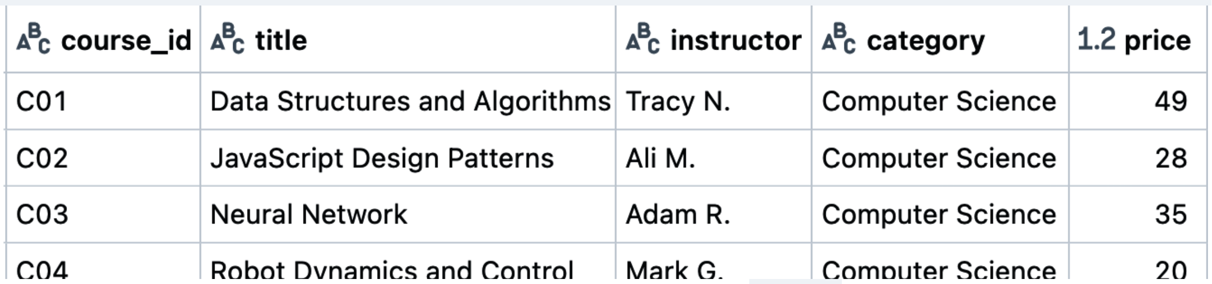


col_name	data_type
Type	MANAGED
Location	dbfs:/user/hive/warehouse/de_associate_school.db/courses
Provider	delta

Figure 4-18. The output of the `DESCRIBE EXTENDED` command on the `courses` table

Finally, querying the table confirms that it contains well-parsed data from the CSV files, as illustrated in Figure 4-19.

```
SELECT * FROM courses
```



course_id	title	instructor	category	1.2 price
C01	Data Structures and Algorithms	Tracy N.	Computer Science	49
C02	JavaScript Design Patterns	Ali M.	Computer Science	28
C03	Neural Network	Adam R.	Computer Science	35
C04	Robot Dynamics and Control	Mark G.	Computer Science	20

Figure 4-19. The result of querying the `courses` table

In the following sections of the book, we will regularly refer to this table whenever we need to access the `courses` data.

Writing to Tables

In this section, we cover the SQL syntax used for inserting and updating records in Delta Lake tables. We will continue using our online school dataset, consisting of three tables: `students`, `enrollments`, and `courses`, as illustrated in Figure 4-20.

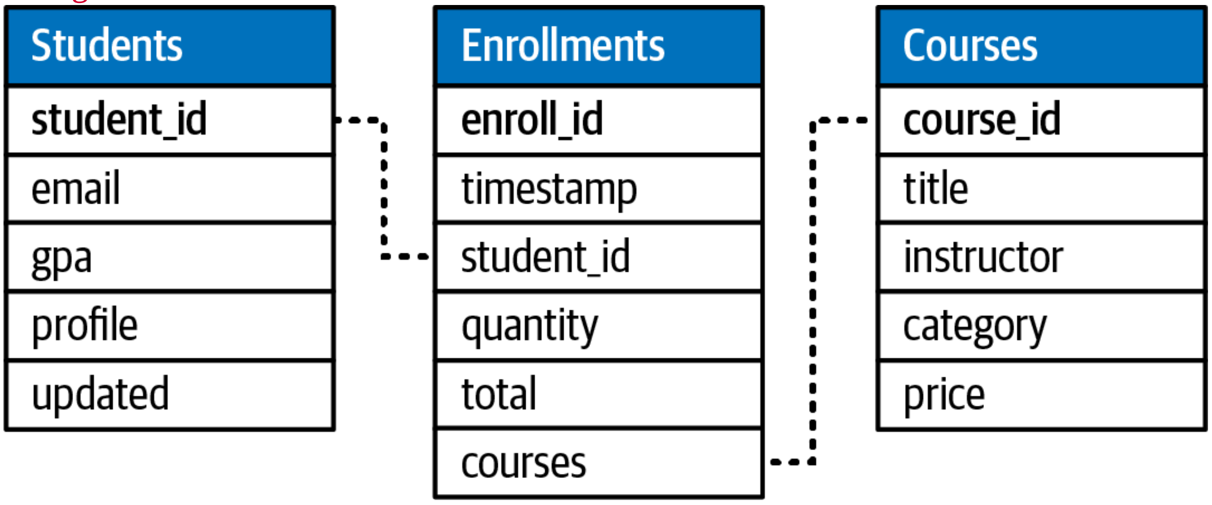


Figure 4-20. Entity-relationship diagram of the school dataset

In this demonstration, we will use a new SQL notebook titled "4.2 - Writing to Tables." We begin by running the "School-Setup" notebook to prepare our environment:

```
%run ../Includes/School-Setup
```

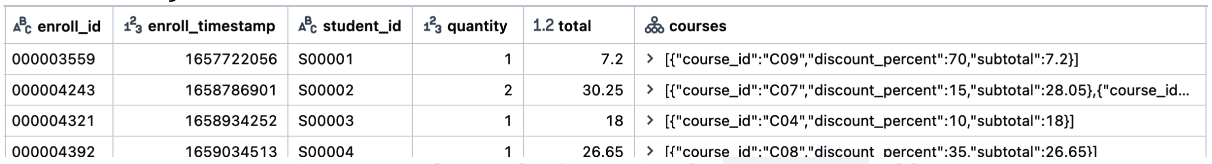
We initiate our exploration by using a CTAS statement to create the `enrollments` Delta table from Parquet files:

```
CREATE TABLE enrollments AS
SELECT * FROM parquet.`${dataset.school}/enrollments`
```

Once the table is created, we proceed to query its content:

```
SELECT * FROM enrollments
```

Figure 4-21 shows the query result. Since Parquet files have a well-defined schema, we observe that Delta Lake has accurately captured the schema and successfully extracted the data.



enroll_id	enroll_timestamp	student_id	quantity	total	courses
000003559	1657722056	S00001	1	7.2	> [{"course_id":"C09","discount_percent":70,"subtotal":7.2}]
000004243	1658786901	S00002	2	30.25	> [{"course_id":"C07","discount_percent":15,"subtotal":28.05},{"course_id...
000004321	1658934252	S00003	1	18	> [{"course_id":"C04","discount_percent":10,"subtotal":18}]
000004392	1659034513	S00004	1	26.65	> [{"course_id":"C08","discount_percent":35,"subtotal":26.65}]

Figure 4-21. The result of querying the `enrollments` table

Replacing Data

You can completely replace the content of a Delta Lake table either by overwriting the existing table or by other traditional methods like dropping and re-creating it. However, overwriting Delta tables offers several advantages over the approach of merely dropping and re-creating tables. Table 4-1 outlines these benefits.

	Drop and recreate table	**Overwrite table**
Processing time	Time-consuming as it involves recursively listing directories and deleting large files.	Fast process since the updated data is just a new table version.
Leveraging Delta's time travel	Deletes the old versions of the table, making its historical data unavailable for retrieval.	Preserves the old table versions, allowing easy retrieval of historical data.
Concurrency	Concurrent queries are unable to access the table while the operation is ongoing.	Concurrent queries can continue reading the table seamlessly while the operation is in progress.

	Drop and recreate table	**Overwrite table**
ACID guarantees	If the operation fails, the table cannot be reverted to its original state.	If the operation fails, the table will revert to its previous state.

Table 4-1. Comparison of dropping and re-creating table versus overwriting table methods

In summary, the process of overwriting tables provides efficiency, reliability, and seamless integration with Delta's features such as time travel and ACID transactions. In Databricks, there are two methods to completely replace the content of Delta Lake tables:

- `CREATE OR REPLACE TABLE` statements
- `INSERT OVERWRITE` statements

1. CREATE OR REPLACE TABLE statement

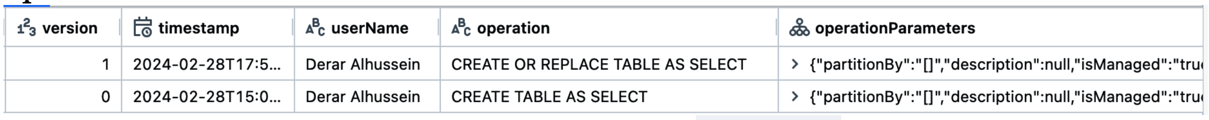
The first method to achieve a complete table overwrite in Delta Lake is by using the `CREATE OR REPLACE TABLE` statement, also known as the CRAS (`CREATE OR REPLACE AS SELECT`) statement. This statement fully replaces the content of a table each time it executes:

```
CREATE OR REPLACE TABLE enrollments AS
SELECT * FROM parquet.`${dataset.school}/enrollments`
```

Upon executing this statement, the `enrollments` table will be overwritten with the newer data. To better understand what happened in the table, let's examine the table history:

```
DESCRIBE HISTORY enrollments
```

As illustrated in Figure 4-22, version 0 is nothing but a CTAS statement. Meanwhile, the `CREATE OR REPLACE` statement has generated a new table version. This new version reflects the updated state of the table after the overwrite operation.



version	timestamp	userName	operation	operationParameters
1	2024-02-28T17:5...	Derar Alhussein	CREATE OR REPLACE TABLE AS SELECT	{"partitionBy":"[]","description":null,"isManaged":"tru
0	2024-02-28T15:0...	Derar Alhussein	CREATE TABLE AS SELECT	{"partitionBy":"[]","description":null,"isManaged":"tru

Figure 4-22. The history log of the `enrollments` table

2. INSERT OVERWRITE

The second method for overwriting data in Delta Tables involves using the `INSERT OVERWRITE` statement:

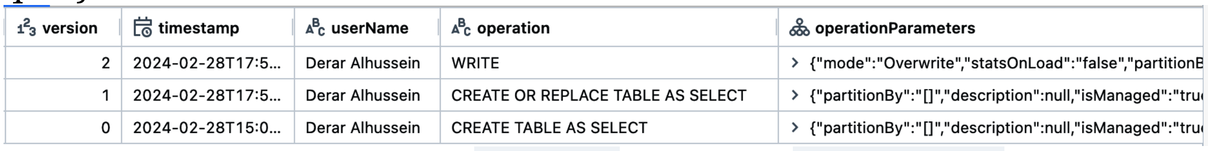
```
INSERT OVERWRITE enrollments
SELECT * FROM parquet.`${dataset.school}/enrollments`
```

While this statement achieves a similar outcome to the `CREATE OR REPLACE TABLE` approach mentioned earlier, there are some key differences and nuances to consider. Unlike the `CREATE OR REPLACE TABLE` statement, which can create a new table if it doesn't exist, `INSERT OVERWRITE` can only overwrite an existing table. This means that the target table must already exist prior to performing the operation.

After executing the `INSERT OVERWRITE` statement, the table history is updated to reflect the overwrite operation:

```
DESCRIBE HISTORY enrollments
```

As displayed in Figure 4-23, Delta Lake records this operation as a new version, categorized as a standard `WRITE` operation. However, the mode of this operation is marked as `"Overwrite"` in the `operationParameters` field. This indicates that the existing data was replaced with the new records from the query.



version	timestamp	userName	operation	operationParameters
2	2024-02-28T17:5...	Derar Alhussein	WRITE	> {"mode":"Overwrite","statsOnLoad":"false","partitionB
1	2024-02-28T17:5...	Derar Alhussein	CREATE OR REPLACE TABLE AS SELECT	> {"partitionBy":"[]","description":null,"isManaged":"tru
0	2024-02-28T15:0...	Derar Alhussein	CREATE TABLE AS SELECT	> {"partitionBy":"[]","description":null,"isManaged":"tru

Figure 4-23. The history log of the `enrollments` table after the `INSERT OVERWRITE` command

One significant advantage of using `INSERT OVERWRITE` is its ability to overwrite only the new records that match the current table schema. This prevents any risk of accidentally modifying the table structure. Thus, `INSERT OVERWRITE` is considered a more secure approach for overwriting existing tables.

When attempting to overwrite data using the `INSERT OVERWRITE` statement with a schema that differs from the existing table schema, a schema mismatch error will be generated. Let's consider an example where we attempt to add an extra column, containing the source file name, to our table:

```
INSERT OVERWRITE enrollments
SELECT *, input_file_name() FROM parquet.`${dataset.school}/enrollments`

AnalysisException: A schema mismatch detected when writing to the Delta table
```

The previous command results in an exception indicating a schema mismatch. This occurs because the schema of the new data being inserted does not match the existing schema of the `enrollments` table.

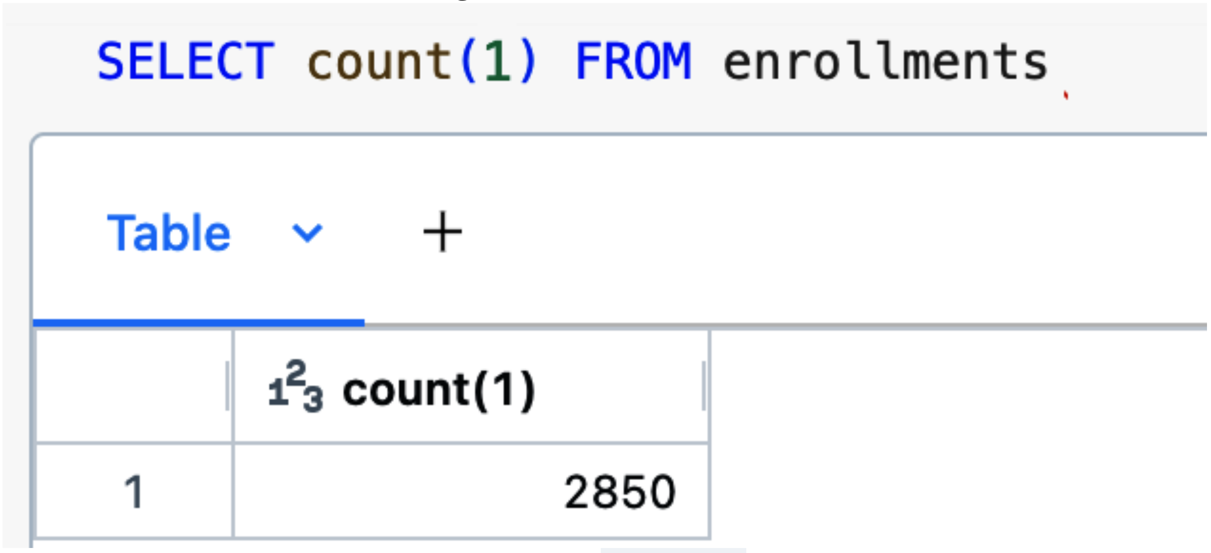
Delta Lake tables are by definition schema-on-write, which means that Delta Lake enforces schema consistency during write operations. Any attempt to write data with a schema that differs from the table's schema will be rejected to maintain data integrity. This behavior differs from the first method of the `CREATE OR REPLACE TABLE` statement, which replaces the entire table along with its schema.

Appending Data

One of the simplest methods to append records to Delta Lake tables is through the use of the `INSERT INTO` statement. This statement allows you to easily add new data to existing tables from the result of a SQL query. Let's explore how this process works with the following command:

```
INSERT INTO enrollments
SELECT * FROM parquet.`${dataset.school}/enrollments-new`
```
In our scenario, we use the `INSERT INTO` statement to add new records to the `enrollments` table. Note that we are not explicitly providing the corresponding column values to be added. Instead, we're using an input query to retrieve the new data from Parquet files located in a given directory. This query serves as the source of our new records, which we then insert into the designated table using the `INSERT INTO` clause.

By executing this `INSERT INTO` statement, we will insert 700 new records into our table. To confirm the success of our operation, we can perform a quick check to verify the updated number of records in the `enrollments` table. Figure 4-24 shows that the number of enrollments has indeed increased, now totaling 2850 records.



```
SELECT count(1) FROM enrollments
```

	count(1)
1	2850

Figure 4-24. The number of records in the `enrollments` table after inserting new data

While the `INSERT INTO` statement provides a convenient means of appending records to tables, it lacks built-in mechanisms to prevent the insertion of duplicate data. This means that if the insertion query is executed multiple times, it will write the same records to the target table repeatedly, leading to the creation of duplicate entries.

To address this issue effectively, we turn to an alternative method: the `MERGE INTO` statement.

Merging Data

The `MERGE INTO` statement enables you to perform upsert operations—meaning you can insert new data and update existing records—and even delete records, all within a single statement. Let's explore how we can use this statement to update the student data in our online school dataset.

In this specific scenario, we aim to update student data with modified email addresses and add new students into the table. To accomplish this, we first create a temporary view containing the updated student data. This view will serve as the source from which we'll merge changes into our `students` table:

```
CREATE OR REPLACE TEMP VIEW students_updates AS
SELECT * FROM json.`${dataset.school}/students-json-new`;
```

The following merge operation is executed to merge the changes from the `student_updates` temporary view into the target `students` table, using the student ID as the key for matching records. Let's first look at the query, and then go into its details:

```
MERGE INTO students c
USING students_updates u
ON c. student_id = u. student_id
WHEN MATCHED AND c.email IS NULL AND u.email IS NOT NULL THEN
 UPDATE SET email = u.email, updated = u.updated
WHEN NOT MATCHED THEN INSERT *
```

Within this `MERGE INTO` statement, we define two primary actions based on the matching status of records:

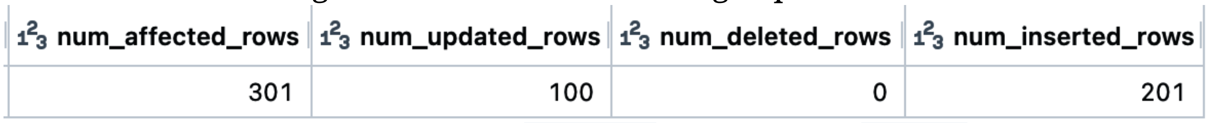
Update action (`WHEN MATCHED` clause)

When a match is found between the source and target records, an update action is performed. This action involves updating the email address and the last updated timestamp. Notice that we introduce additional conditions to this action. Specifically, we check if the email address in the current row is null while the corresponding record in the `student_updates` view contains a valid email address. For such records, we proceed by updating the email field and the last updated timestamp in the target table.

Insert action (`WHEN NOT MATCHED` clause)

For records in the `student_updates` view that do not match any existing students based on the student ID, an insert action is triggered. This ensures that all new students are added into our target table.

Let's now proceed with the execution of this query. Figure 4-25 presents the metrics summarizing the outcomes of our merge operation.



num_affected_rows	num_updated_rows	num_deleted_rows	num_inserted_rows
301	100	0	201

Figure 4-25. The output of the `MERGE INTO` command on the `students` table

We observe that 100 records have been updated, reflecting the changes in email addresses and last updated timestamps. In addition, 201 new records have been inserted into the `students` table. No records have been deleted during this process since there was no delete action included in the query (`WHEN MATCHED [condition] THEN DELETE`).

One of the key advantages of the `MERGE INTO` statement is its ability to execute updates, inserts, and deletes within a single atomic transaction. This ensures data consistency and integrity by treating all operations as a single unit, thereby minimizing the risk of inconsistencies or partial changes on the table data.

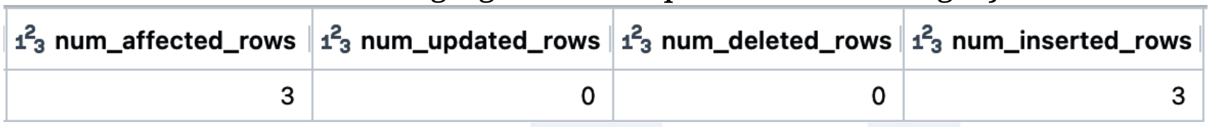
Additionally, the merge operation serves as an excellent solution for preventing duplicates during record insertion. Let's consider another scenario where we have a set of new courses to be inserted, delivered in CSV format. To facilitate this, we'll establish a temporary view based on this new data:

```
CREATE OR REPLACE TEMP VIEW courses_updates
   (course_id STRING, title STRING, instructor STRING,
    category STRING, price DOUBLE)
USING CSV
OPTIONS (
 path = "${dataset.school}/courses-csv-new",
 header = "true",
 delimiter = ";"
);
```

Now, we can use the `MERGE INTO` statement to synchronize the `courses` table with the information sourced from the temporary view `courses_updates`.
In this scenario, we exclusively focus on the condition where there is no match. This implies that we'll only insert new data if it doesn't already exist in our target table, based on the unique key comprising both the `course_id` and the `title` fields. Among the new courses, our interest lies only in inserting those categorized under computer science. For this purpose, we'll specify that only records categorized under `Computer Science` are eligible for insertion by adding an additional criterion:

```
MERGE INTO courses c
USING courses_updates u
ON c.course_id = u.course_id AND c.title = u.title
WHEN NOT MATCHED AND u.category = 'Computer Science' THEN
 INSERT *
```

As displayed in Figure 4-26, the query execution resulted in the insertion of three new records, all belonging to the computer science category.

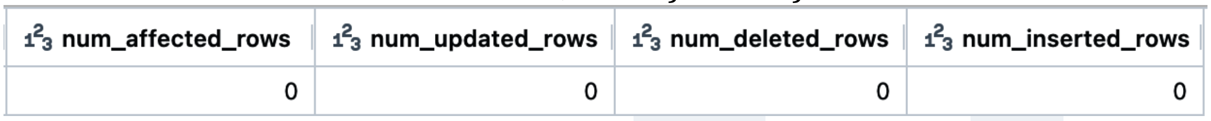


num_affected_rows	num_updated_rows	num_deleted_rows	num_inserted_rows
3	0	0	3

Figure 4-26. The output of the `MERGE INTO` command on the `courses` table

This operation is called insert-only merge, which demonstrates one of the primary advantages of the merge operation: its ability to prevent duplicate entries. To confirm this, let's rerun the previous query and see the resulting behavior.

As shown in Figure 4-27, the second execution of our merge statement didn't lead to the reinsertion of the records, as they already exist in the table.



num_affected_rows	num_updated_rows	num_deleted_rows	num_inserted_rows
0	0	0	0

Figure 4-27. The output of the second run of the `MERGE INTO` command on the `courses` table

In conclusion, while the `INSERT INTO` statement offers a straightforward method for appending records to tables, its drawback of duplicate record insertion necessitates the adoption of more robust strategies, such as the `MERGE INTO` statement. With `MERGE INTO`, you can effectively upsert data to avoid duplicates.

Performing Advanced ETL Transformations

In this section, we will explore advanced transformations available in Spark SQL, covering the capabilities it provides for handling nested and complex data structures. We will continue using our online school dataset, consisting of three tables: `students`, `enrollments`, and `courses`.

In this demonstration, we will use a new SQL notebook titled “4.3 - Advanced Transformations.” We begin by running the “School-Setup” notebook to prepare our environment:

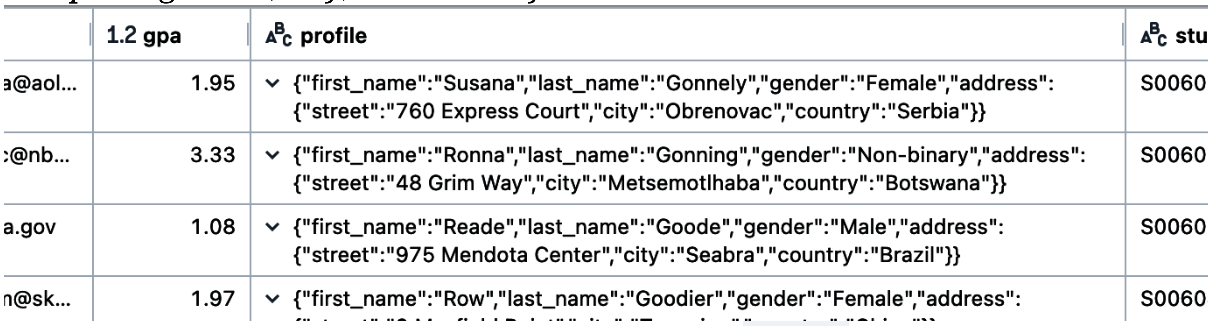
```
%run ../Includes/School-Setup
```

Dealing with Nested JSON Data

Let’s first recall our student data:

```
SELECT * FROM students
```

Figure 4-28 displays the result of querying the `students` table created in the previous section. It shows a column containing the profile information of each student, represented as a nested JSON structure. Specifically, we notice the address information of the profile is stored as a JSON object as well, comprising street, city, and country details.



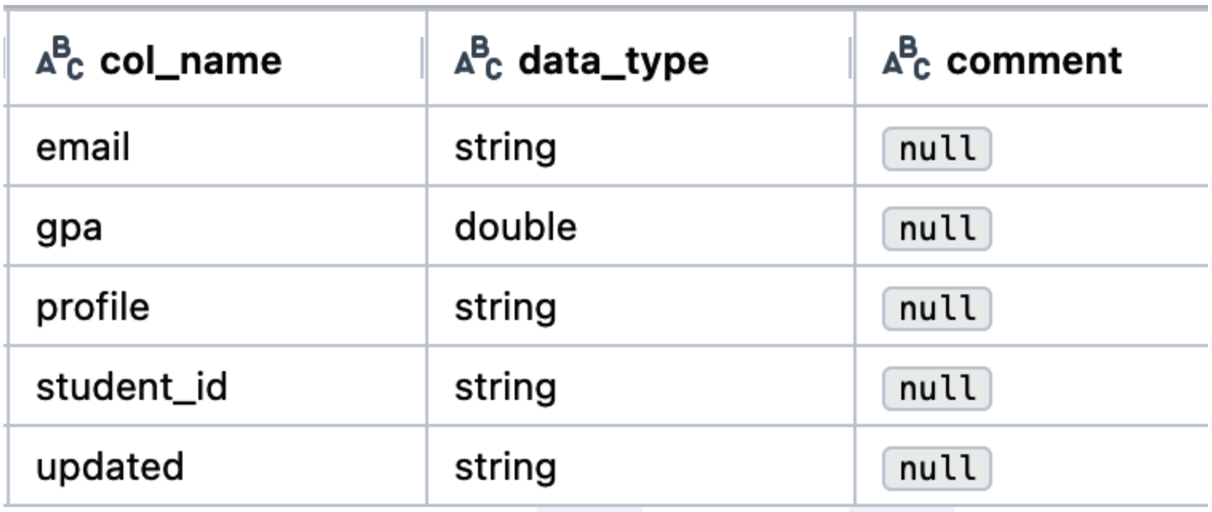
	1.2 gpa	profile	stu
a@aol...	1.95	{"first_name":"Susana","last_name":"Gonnely","gender":"Female","address":{"street":"760 Express Court","city":"Obrenovac","country":"Serbia"}}	S0060
:@nb...	3.33	{"first_name":"Ronna","last_name":"Gonning","gender":"Non-binary","address":{"street":"48 Grim Way","city":"Metsemotlhaba","country":"Botswana"}}	S0060
a.gov	1.08	{"first_name":"Reade","last_name":"Goode","gender":"Male","address":{"street":"975 Mendota Center","city":"Seabra","country":"Brazil"}}	S0060
n@sk...	1.97	{"first_name":"Row","last_name":"Goodier","gender":"Female","address":	S0060

Figure 4-28. The result of querying the `students` table

To check the data type of the `profile` column, we can use the `DESCRIBE` command, which helps in exploring the schema of the table:

```
DESCRIBE students
```

In Figure 4-29, we observe that the `profile` column is nothing but a string; it’s a JSON string.



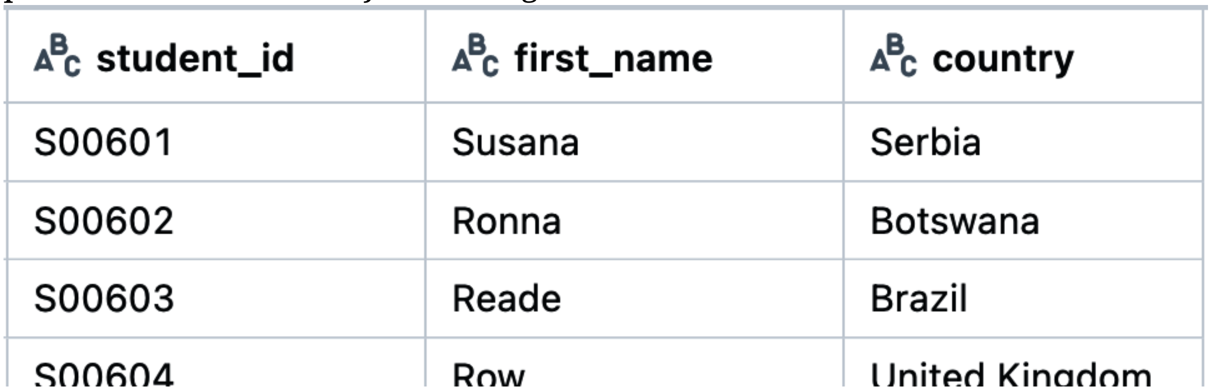
col_name	data_type	comment
email	string	null
gpa	double	null
profile	string	null
student_id	string	null
updated	string	null

Figure 4-29. The output of the `DESCRIBE` command on the `students` table

Spark SQL facilitates interaction with such JSON data by using a colon syntax (`:`) to navigate through its nested structures. In this example, we access the first name within the `profile` column using the colon syntax. Similarly, we extract the nested value of the country from the address within the profile:

```
SELECT student_id, profile:first_name, profile:address:country
FROM students
```

The output in Figure 4-30 confirms that we have successfully extracted the profile details from the JSON string.



student_id	first_name	country
S00601	Susana	Serbia
S00602	Ronna	Botswana
S00603	Reade	Brazil
S00604	Row	United Kingdom

Figure 4-30. The result of extracting the profile details using the colon syntax

Parsing JSON into Struct Type

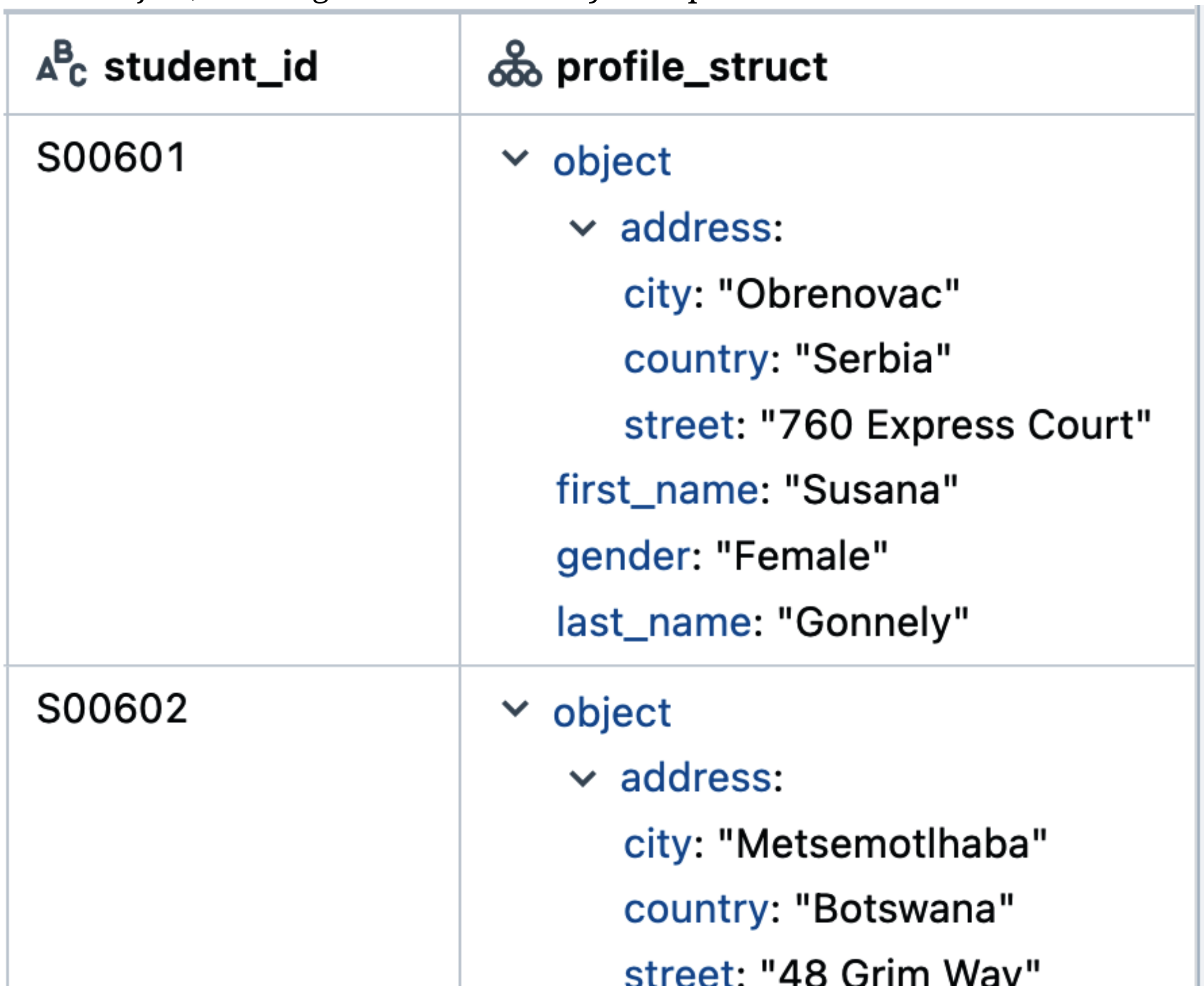
Spark SQL goes further by providing functionality to parse JSON objects into struct types—a native Spark type with nested attributes. The `from_json` function is employed for this task, but it requires knowledge of the schema of the JSON object in advance:

```
SELECT from_json(profile, <schema>) FROM students;
```

In response to this requirement, we can use the `schema_of_json` function, which derives the schema from sample data of the JSON object, provided the fields are non-null. In the following example, we provide sample data of a student's profile to obtain the corresponding schema. This schema is then used in the `from_json` function to allow successful parsing of JSON objects into struct types. Note that we could also use a SQL-style column-type declaration for the schema instead of inferring it. Additionally, we store the resulting records in a temporary view for further analysis:

```
CREATE OR REPLACE TEMP VIEW parsed_students AS
 SELECT student_id, from_json(profile, schema_of_json('{"first_name":"Sarah",
 "last_name":"Lundi", "gender":"Female", "address":{"street":"8 Greenbank Road",
 "city":"Ottawa", "country":"Canada"}}')) AS profile_struct
 FROM students;

SELECT * FROM parsed_students
```

Figure 4-31 shows the result of parsing the profile JSON objects into struct types. As illustrated, the preview display allows us to expand and collapse the struct object, offering a convenient way to explore its contents.



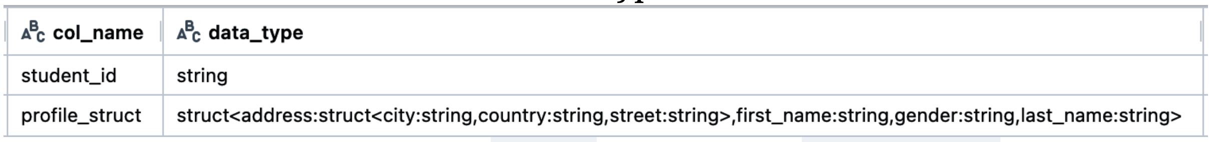
student_id	profile_struct
S00601	object address: city: "Obrenovac" country: "Serbia" street: "760 Express Court" first_name: "Susana" gender: "Female" last_name: "Gonnely"
S00602	object address: city: "Metsemotlhaba" country: "Botswana" street: "48 Grim Way"

Figure 4-31. The result of parsing the profile JSON objects into struct types

Let's check again the data type of the `profile` column by running the `DESCRIBE` command on our view:

```
DESCRIBE parsed_students
```

Figure 4-32 confirms that the column `profile_struct` is indeed of a struct type, and its inner address field is of a struct type as well.



col_name	data_type
student_id	string
profile_struct	struct<address:struct<city:string,country:string,street:string>,first_name:string,gender:string,last_name:string>

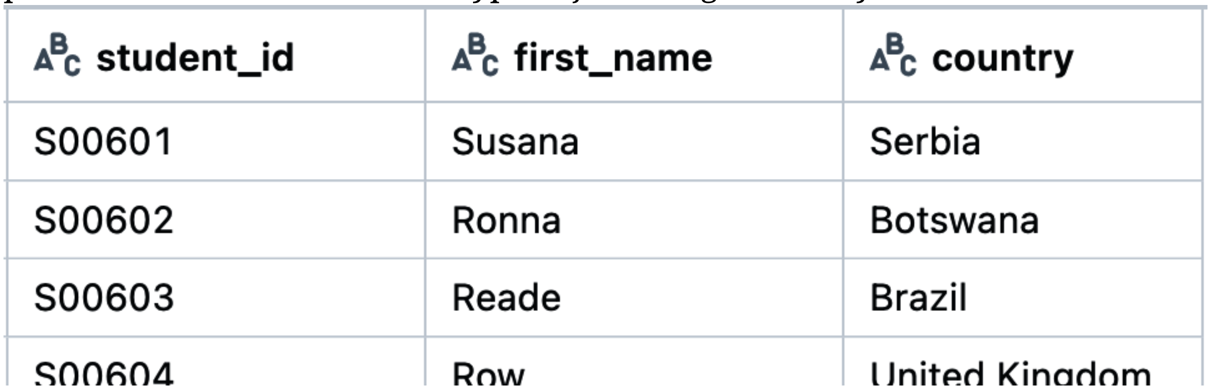
Figure 4-32. The output of the `DESCRIBE` command on the `parsed_students` table

Interacting with Struct Types

When working with struct types, a notable aspect is the ability to interact with nested objects using standard period or dot (.) syntax, compared to the colon syntax used for JSON strings. This makes the code more intuitive and aligns with Spark's native representation.

```
SELECT student_id, profile_struct.first_name, profile_struct.address.country
FROM parsed_students
```

The output in Figure 4-33 confirms that we have successfully extracted the profile details from the struct type object using the dot syntax.



student_id	first_name	country
S00601	Susana	Serbia
S00602	Ronna	Botswana
S00603	Reade	Brazil
S00604	Row	United Kingdom

Figure 4-33. The result of extracting the profile details using the dot syntax

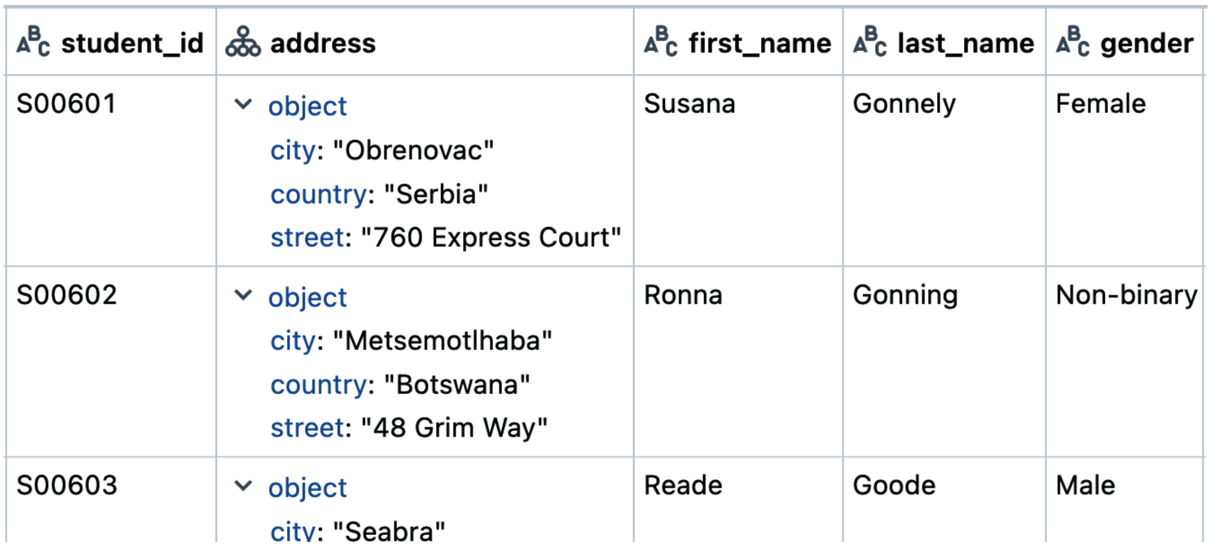
Flattening Struct Types

Once a JSON string is converted to a struct type, Spark SQL introduces a powerful feature—the ability to use the star (*) operation to flatten fields and create separate columns:

```
CREATE OR REPLACE TEMP VIEW students_final AS
 SELECT student_id, profile_struct.*
 FROM parsed_students;

SELECT * FROM students_final
```

The output in Figure 4-34 confirms that this transformation resulted in distinct columns for the first name, last name, gender, and address elements of the profile field.

In summary, Spark SQL's advanced transformations empower you to handle nested and complex data structures with ease, providing functionalities for parsing JSON objects into struct types and performing operations on them.



student_id	address	first_name	last_name	gender
S00601	object city: "Obrenovac" country: "Serbia" street: "760 Express Court"	Susana	Gonnely	Female
S00602	object city: "Metsemotlhaba" country: "Botswana" street: "48 Grim Way"	Ronna	Gonning	Non-binary
S00603	object city: "Seabra"	Reade	Goode	Male

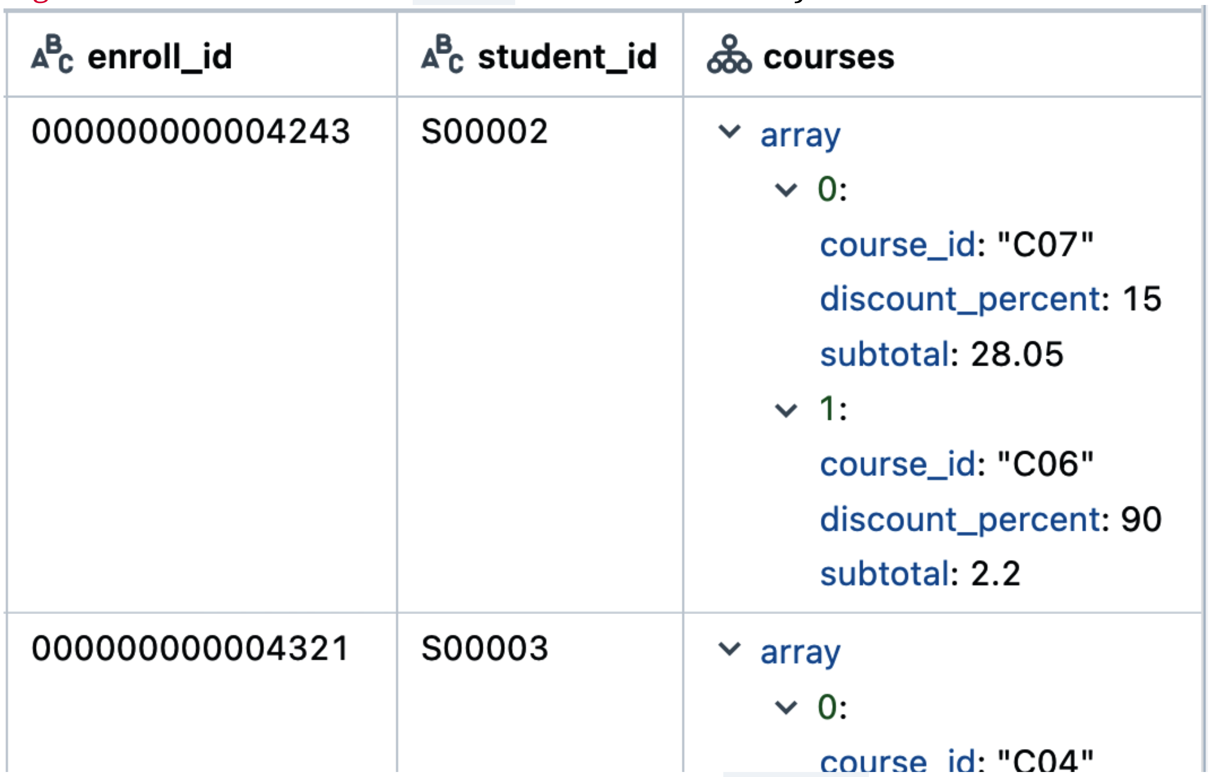
Figure 4-34. The result of the star operation flattens the profile's fields into separate columns

Leveraging the explode Function

In this section, we shift our focus to the `enrollments` table and explore an advanced feature in Spark SQL—the `explode` function. Let's begin by reviewing some fields within our table:

```
SELECT enroll_id, student_id, courses
FROM enrollments
```

Figure 4-35 shows that the `courses` column is an array of structs.



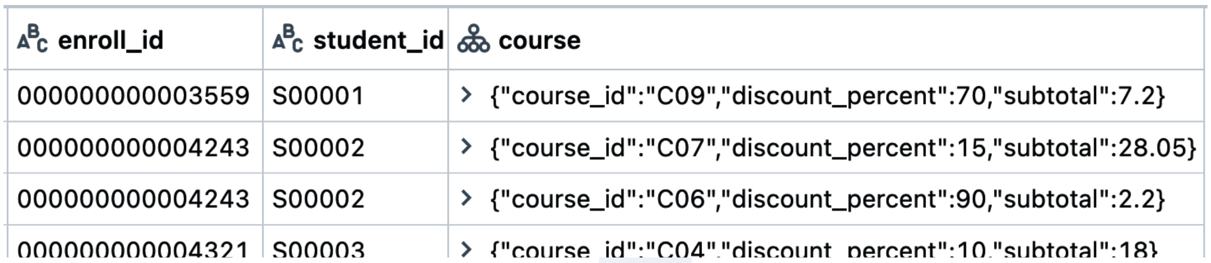
enroll_id	student_id	courses
000000000004243	S00002	array 0: course_id: "C07" discount_percent: 15 subtotal: 28.05 1: course_id: "C06" discount_percent: 90 subtotal: 2.2
000000000004321	S00003	array 0: course_id: "C04"

Figure 4-35. The result of querying the `enrollments` table

Spark SQL provides dedicated functions for efficiently handling arrays, like the `explode` function. This function allows us to transform an array into individual rows, each representing an element from the array:

```
SELECT enroll_id, student_id, explode(courses) AS course
FROM enrollments
```

Figure 4-36 displays the results of applying the `explode` function to the `courses` array. Each course element now occupies its own row, with the other information such as student ID and enrollment ID being duplicated for each course.



enroll_id	student_id	course
000000000003559	S00001	> {"course_id":"C09","discount_percent":70,"subtotal":7.2}
000000000004243	S00002	> {"course_id":"C07","discount_percent":15,"subtotal":28.05}
000000000004243	S00002	> {"course_id":"C06","discount_percent":90,"subtotal":2.2}
000000000004321	S00003	> {"course_id":"C04","discount_percent":10,"subtotal":18}

Figure 4-36. The result of applying the `explode` function to the courses column

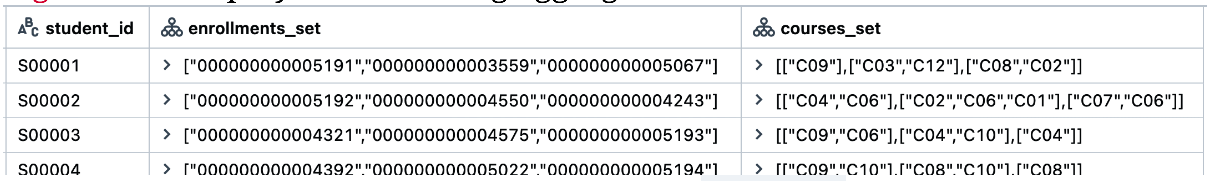
This layout is particularly useful when examining course-level patterns or performing operations such as aggregations and joins with other tables.

Aggregating Unique Values

Moving forward, we explore another interesting function—the `collect_set` function. This function is an aggregation function that returns an array of unique values for a given field. It can even deal with fields within arrays. In this example, the `courses_set` column is formed as an array of arrays:

```
SELECT student_id,
 collect_set(enroll_id) AS enrollments_set,
 collect_set(courses.course_id) AS courses_set
FROM enrollments
GROUP BY student_id
```

Figure 4-37 displays the resulting aggregations.



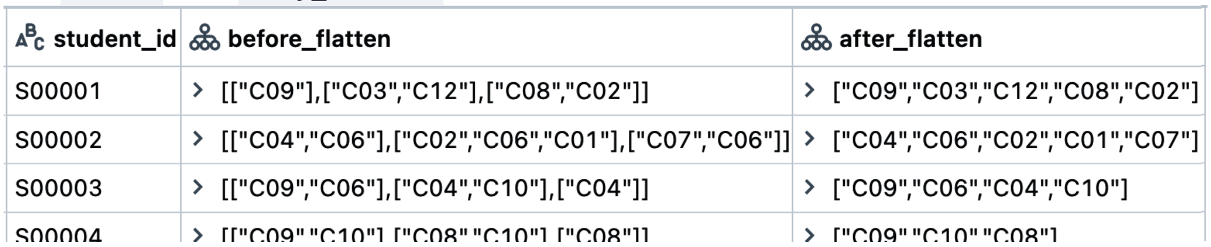
student_id	enrollments_set	courses_set
S00001	> ["000000000005191","000000000003559","000000000005067"]	> [["C09"],["C03","C12"],["C08","C02"]]
S00002	> ["000000000005192","000000000004550","000000000004243"]	> [["C04","C06"],["C02","C06","C01"],["C07","C06"]]
S00003	> ["000000000004321","000000000004575","000000000005193"]	> [["C09","C06"],["C04","C10"],["C04"]]
S00004	> ["000000000004392","000000000005022","000000000005194"]	> [["C09","C10"],["C08","C10"],["C08"]]

Figure 4-37. The result of applying the `collect_set` function

In the `courses_set` column, we notice that, for instance, the course with identifier `C06` appears in multiple elements in the array of student `S00002`. To avoid such an issue, we can flatten this nested array and retain only the distinct values. This can be achieved in a two-step process. First, we apply the `flatten` function to flatten the array, and then, we use the `array_distinct` function to retain only the unique values:

```
SELECT student_id,
 collect_set(courses.course_id) As before_flatten,
 array_distinct(flatten(collect_set(courses.course_id))) AS after_flatten
FROM enrollments
GROUP BY student_id
```

Figure 4-38 offers a before-and-after perspective, showcasing the original state of the data and the result achieved after applying the `flatten` and `array_distinct` functions.



student_id	before_flatten	after_flatten
S00001	[["C09"],["C03","C12"],["C08","C02"]]	["C09","C03","C12","C08","C02"]
S00002	[["C04","C06"],["C02","C06","C01"],["C07","C06"]]	["C04","C06","C02","C01","C07"]
S00003	[["C09","C06"],["C04","C10"],["C04"]]	["C09","C06","C04","C10"]
S00004	[["C09","C10"],["C08","C10"],["C08"]]	["C09","C10","C08"]

Figure 4-38. The result of flattening the array

In practice, the `flatten` function is employed to transform the nested array into a flat structure. Following this, the `array_distinct` function is applied to eliminate any duplicate values. This confirms, for example, that our course identifier `C06` of student `S00002` is now represented only once in the resulting array.

Mastering Join Operations in Spark SQL

Spark SQL also supports join operations to facilitate blending data from different tables. It offers a variety of standard join operations, including inner, outer, left, right, anti, cross, and semi joins. In the following example, we'll focus on an inner join operation, where we combine the result of an explode operation with the `courses` lookup table to extract the course's details, such as course titles and instructor names.

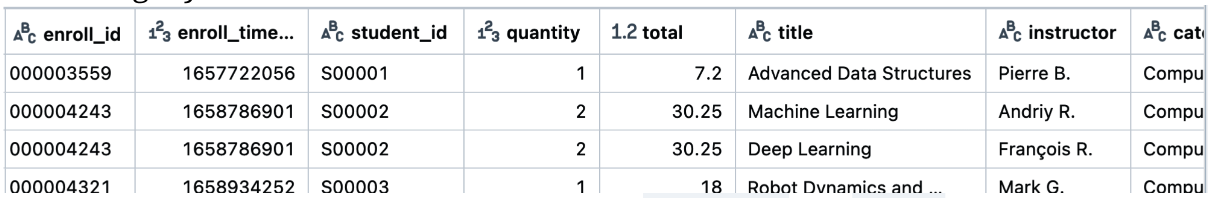
The first step in this operation involves performing an explode operation on our dataset to transform array elements into individual rows. Subsequently, we desire to enrich this exploded data with additional information from the `courses` lookup table. To achieve this, we execute an inner join based on the common key, in this case, the `course_id`.

The syntax used for joining data in Spark SQL follows the conventions of standard SQL. We specify the type of join we want (inner, outer, left, right, etc.), the tables we are joining, and the conditions for the join. In this example, an inner join is applied based on matching the `course_id` key. This ensures that only matching records from both tables are retained in the final result set.

Lastly, we store the enriched data in a temporary view named `enrollments_enriched`:

```
CREATE OR REPLACE VIEW enrollments_enriched AS
SELECT *
FROM (
   SELECT *, explode(courses) AS course
   FROM enrollments) e
INNER JOIN courses c
ON e.course.course_id = c.course_id;

SELECT * FROM enrollments_enriched
```

Figure 4-39 displays the result of this join operation, incorporating information from both the exploded data and the `courses` lookup table. For each course, we can now easily access its details like the title, instructor name, and category.



enroll_id	enroll_time...	student_id	quantity	total	title	instructor	cat
000003559	1657722056	S00001	1	7.2	Advanced Data Structures	Pierre B.	Compu
000004243	1658786901	S00002	2	30.25	Machine Learning	Andriy R.	Compu
000004243	1658786901	S00002	2	30.25	Deep Learning	François R.	Compu
000004321	1658934252	S00003	1	18	Robot Dynamics and ...	Mark G.	Compu

Figure 4-39. The result of joining `enrollments` with `courses`

Exploring Set Operations in Spark SQL

Similar to relational databases, Spark SQL supports various set operations such as union, intersect, and except/minus. Let's explore these set operations by applying them to our `enrollments` table, which currently holds 2,150 records, alongside a temporary view that will introduce 700 new records. To begin, we'll create the view under the name `enrollments_updates`:

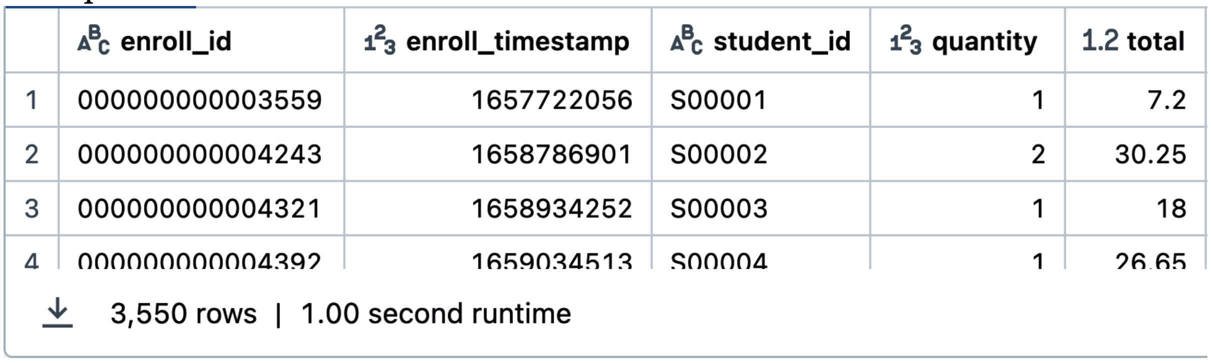
```
CREATE OR REPLACE TEMP VIEW enrollments_updates
AS SELECT * FROM parquet.`${dataset.school}/enrollments-new`;
```

Union operation

The union operation in Spark SQL enables the combination of two datasets by stacking them vertically, with two variants available: `UNION` and `UNION ALL`. While `UNION` (or `UNION DISTINCT`) returns only distinct rows, `UNION ALL` includes all rows from both datasets, preserving duplicates. In the following example, we demonstrate the `UNION ALL` operation by combining the old and new data of the `enrollments` table. This results in a unified dataset that includes all records from both sources, including duplicates:

```
SELECT * FROM enrollments
UNION ALL
SELECT * FROM enrollments_updates
```

Figure 4-40 displays 3,550 records as a result of this union operation, which includes duplicate entries. This provides a comprehensive view that incorporates both old and new records.



	enroll_id	enroll_timestamp	student_id	quantity	total
1	000000000003559	1657722056	S00001	1	7.2
2	000000000004243	1658786901	S00002	2	30.25
3	000000000004321	1658934252	S00003	1	18
4	000000000004392	1659034513	S00004	1	26.65

3,550 rows | 1.00 second runtime

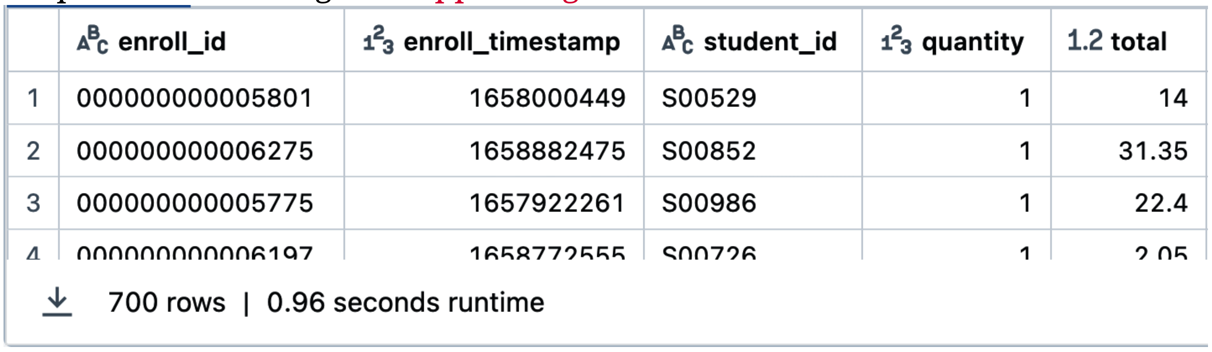
Figure 4-40. The result of the union operation

Intersect operation

The intersect operation, on the other hand, returns the common rows found in both datasets. This operation is useful when identifying overlaps between two datasets. In the following scenario, the `INTERSECT` command is applied to find rows that exist in both the `enrollments` table and the `enrollments_updates` view:

```
SELECT * FROM enrollments
INTERSECT
SELECT * FROM enrollments_updates
```

Figure 4-41 reveals that there are 700 records present in both sources. This stems from the insertion of these 700 records into the `enrollments` table, which we performed during the "Appending Data" section.



	enroll_id	enroll_timestamp	student_id	quantity	total
1	000000000005801	1658000449	S00529	1	14
2	000000000006275	1658882475	S00852	1	31.35
3	000000000005775	1657922261	S00986	1	22.4
4	000000000006197	1658772555	S00726	1	2.05

700 rows | 0.96 seconds runtime

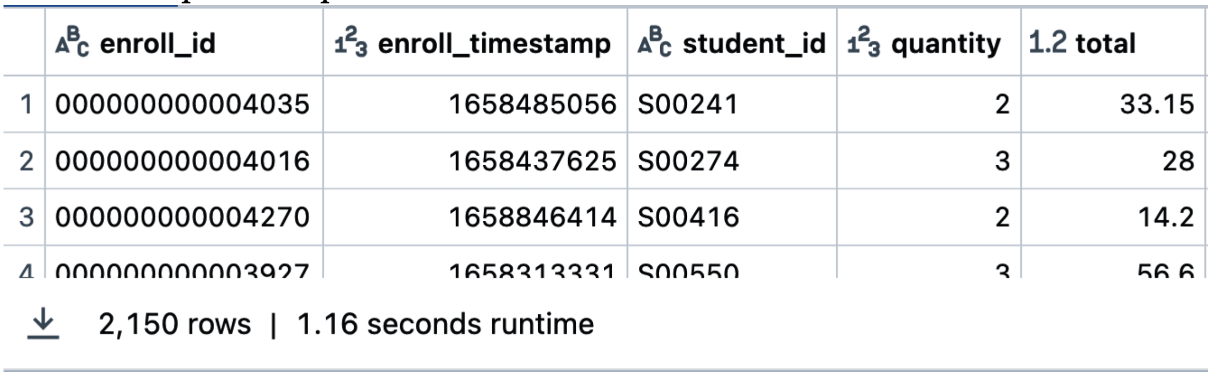
Figure 4-41. The result of the intersect operation

Minus operation

An interesting use case of set operations involves leveraging the `MINUS` operation to obtain records exclusive to one dataset. For instance, if we execute `enrollments` minus `enrollments_updates`, we effectively retrieve only the data from the original `enrollments` table that does not overlap with the 700 new records present in the `enrollments_updates` view:

```
SELECT * FROM enrollments
MINUS
SELECT * FROM enrollments_updates
```

Figure 4-42 displays the entries exclusive to the `enrollments` table after excluding the 700 shared records. The minus operation is particularly useful for isolating records of interest, allowing you to focus only on them. In the provided example, this allows you to focus on the `enrollments`' data before the last insert operation performed on the table.



	enroll_id	enroll_timestamp	student_id	quantity	total
1	000000000004035	1658485056	S00241	2	33.15
2	000000000004016	1658437625	S00274	3	28
3	000000000004270	1658846414	S00416	2	14.2
4	000000000003927	1658313331	S00550	3	56.6

2,150 rows | 1.16 seconds runtime

Figure 4-42. The result of the minus operation

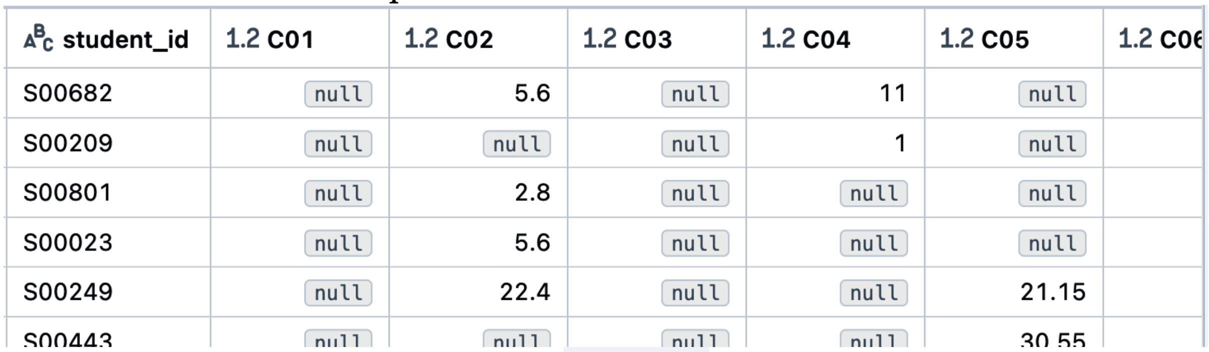
In conclusion, the set operations available in Spark SQL enable you to perform a range of tasks including combining, comparing, and isolating datasets.

Changing Data Perspectives

In addition to its support for set operations, Spark SQL supports creating pivot tables for transforming data perspectives using the `PIVOT` clause. This provides a means to generate aggregated values based on specific column values. This transformation results in a pivot table, wherein the aggregated values become multiple columns. Let's explore the `PIVOT` clause with a practical example, where we aggregate and flatten the enrollment information for each student. Before analyzing the query syntax, let's first execute the query and examine its output:

```
1 SELECT * FROM (
2 SELECT student_id, course.course_id AS course_id, course.subtotal AS subtotal
3  FROM enrollments_enriched
4 )
5 PIVOT (
6 sum(subtotal) FOR course_id IN (
7   'C01', 'C02', 'C03', 'C04', 'C05', 'C06',
8   'C07', 'C08', 'C09', 'C10', 'C11', 'C12')
9 )
```

Figure 4-43 displays the resulting pivot table that illustrates the aggregated sum of subtotal amounts per course for each student.



student_id	1.2 C01	1.2 C02	1.2 C03	1.2 C04	1.2 C05	1.2 C0
S00682	null	5.6	null	11	null	
S00209	null	null	null	1	null	
S00801	null	2.8	null	null	null	
S00023	null	5.6	null	null	null	
S00249	null	22.4	null	null	21.15	
S00443	null	null	null	null	30.55	

Figure 4-43. The `enrollments` pivot table

The query syntax for generating the pivot table involves the following steps:

1. Selecting its input data from a table or subquery (lines 1–4).
2. Calling the pivot clause (lines 5–9), which consists of three key components:
    1. Aggregation function: The `sum(subtotal)` specifies the aggregation function to be applied, along with the column to be aggregated.

2. `FOR` subclause: This subclause defines the pivot column, `course_id`, which is the basis for creating multiple columns in the output.
3. `IN` operator: The `IN` operator lists the distinct values of the pivot column. In our case, it lists the distinct course IDs (from `C01` to `C12`), each presented as separate columns in the pivot table.

In essence, the `PIVOT` clause in Spark SQL empowers you to reshape and aggregate data dynamically. This capability is essential for many analytical and machine learning tasks.

Working with Higher-Order Functions

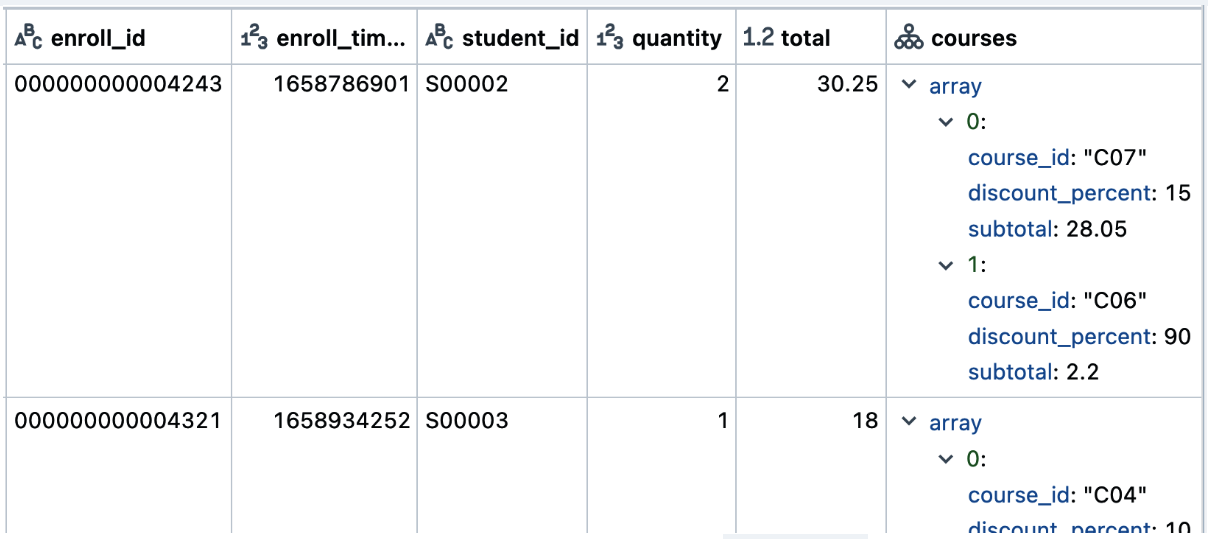
Higher-order functions in Databricks provide a powerful toolset for working with complex data types, such as arrays. In this section, we'll cover two essential functions: `FILTER` and `TRANSFORM`.

In this demonstration, we will use a new SQL notebook titled "4.4 - Higher-Order Functions." To ensure our environment is properly configured, we start by executing the "School-Setup" notebook, maintaining our focus on using the online school dataset:

```
%run ../Includes/School-Setup
```

Let's first review our student enrollment data, illustrated in Figure 4-44:

```
SELECT * FROM enrollments
```



enroll_id	enroll_tim...	student_id	quantity	total	courses
000000000004243	1658786901	S00002	2	30.25	array 0: course_id: "C07" discount_percent: 15 subtotal: 28.05 1: course_id: "C06" discount_percent: 90 subtotal: 2.2
000000000004321	1658934252	S00003	1	18	array 0: course_id: "C04" discount_percent: 10

Figure 4-44. The result of querying the `enrollments` table

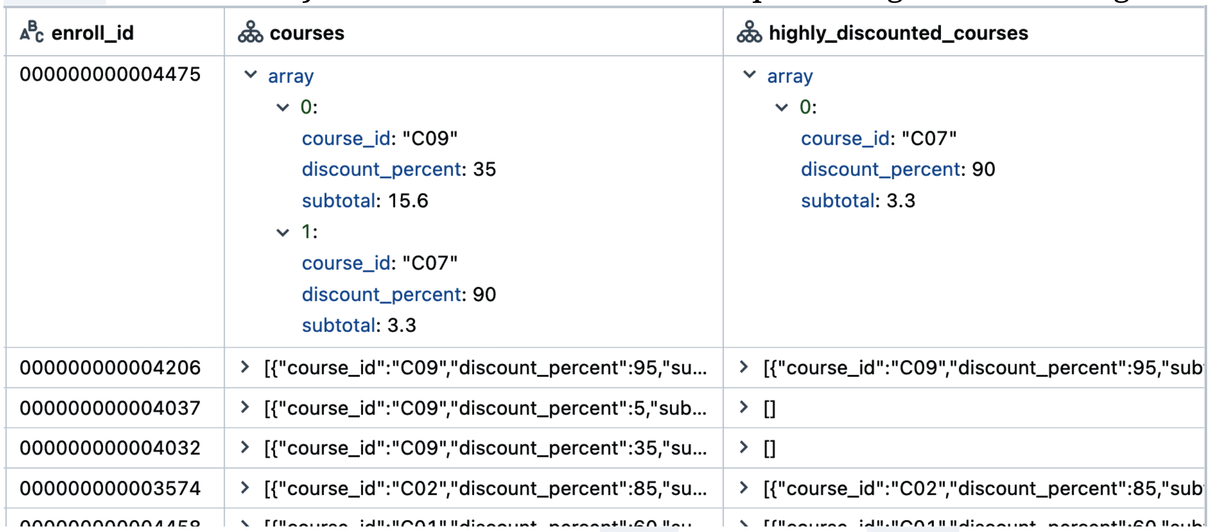
The query result demonstrates that the `courses` column is of complex data type, specifically an array of struct objects. To effectively work with such hierarchical data, it is essential to use higher-order functions.

Filter Function

The `FILTER` function is a fundamental higher-order function that enables the extraction of specific elements from an array based on a given lambda function.
In the following example, we create a new column named `highly_discounted_courses` to identify courses that were purchased with a significant discount. This column is populated by filtering the `courses` field to only include courses with a discount percentage of 60% or higher:

```
SELECT
enroll_id,
courses,
FILTER (courses,
        course -> course.discount_percent >= 60) AS highly_discounted_courses
FROM enrollments
```

Figure 4-45 displays the filtered data, where the column `highly_discounted_courses` contains only the courses with a discount percentage of 60% or higher.



enroll_id	courses	highly_discounted_courses
000000000004475	array 0: course_id: "C09" discount_percent: 35 subtotal: 15.6 1: course_id: "C07" discount_percent: 90 subtotal: 3.3	array 0: course_id: "C07" discount_percent: 90 subtotal: 3.3
000000000004206	[{"course_id":"C09","discount_percent":95,"su...	[{"course_id":"C09","discount_percent":95,"sub
000000000004037	[{"course_id":"C09","discount_percent":5,"sub...	[]
000000000004032	[{"course_id":"C09","discount_percent":35,"su...	[]
000000000003574	[{"course_id":"C02","discount_percent":85,"su...	[{"course_id":"C02","discount_percent":85,"sub

Figure 4-45. The result of applying the `FILTER` function on the `courses` column

However, we observe that the column has several empty arrays. To resolve this, we can use a `WHERE` clause to display only non-empty array values. However, because a derived column generally cannot be referenced directly within a `WHERE` clause, using a subquery is essential to achieve the desired outcome:

```
SELECT enroll_id, highly_discounted_courses
FROM (
 SELECT
   enroll_id,
   courses,
   FILTER (courses,
      course -> course.discount_percent >= 60) AS highly_discounted_courses
 FROM enrollments)
WHERE size(highly_discounted_courses) > 0;
```

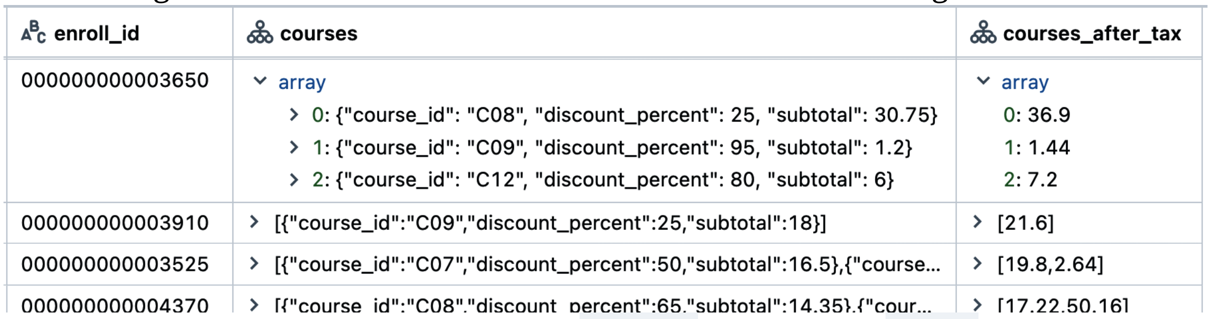
By using this subquery that applies the `WHERE` clause to the size of the returned column, you can successfully eliminate all empty arrays.

Transform Function

The TRANSFORM function is another essential higher-order function that facilitates the application of a transformation to each item in an array, extracting the transformed values. In our example, we apply a 20% tax to the subtotal value for each course in the courses array:

```
SELECT
enroll_id,
courses,
TRANSFORM (
   courses,
   course -> ROUND(course.subtotal * 1.2, 2) ) AS courses_after_tax
FROM enrollments;
```

Figure 4-46 displays the result of applying the TRANSFORM function, which adds a new column, courses_after_tax, containing an array of transformed values for each element in the courses array. The transformation, in this case, involves calculating a 20% tax on the subtotal value and then rounding the result.



enroll_id	courses	courses_after_tax

000000000003650	array
	0: {"course_id": "C08", "discount_percent": 25, "subtotal": 30.75}
	1: {"course_id": "C09", "discount_percent": 95, "subtotal": 1.2}
	2: {"course_id": "C12", "discount_percent": 80, "subtotal": 6}
		array
		0: 36.9
		1: 1.44
		2: 7.2

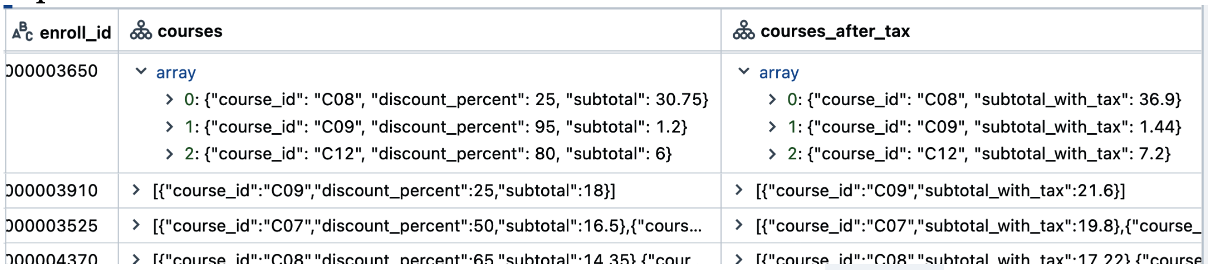
000000000003910	[{"course_id":"C09","discount_percent":25,"subtotal":18}]	[21.6]
000000000003525	[{"course_id":"C07","discount_percent":50,"subtotal":16.5},{"course...	[19.8,2.64]
000000000004370	[{"course_id":"C08","discount_percent":65,"subtotal":14.35},{"cour...	[17.22,50.16]

Figure 4-46. The result of applying the TRANSFORM function on the courses column

Clearly, the transform function extracts only the transformed values by default. Instead, we can create a struct object containing multiple elements. This struct object contains two fields: the course ID for the original course, and subtotal_with_tax reflecting the subtotal amount after applying the tax:

```
SELECT
enroll_id,
courses,
TRANSFORM (
   courses,
   course -> (course.course_id,
                ROUND(course.subtotal * 1.2, 2) AS subtotal_with_tax)
) AS courses_after_tax
FROM enrollments;
```

Figure 4-47 displays the result of generating struct objects with the TRANSFORM function. This allows for more structured and detailed representation of the transformed data.



enroll_id	courses	courses_after_tax

000003650	array
	0: {"course_id": "C08", "discount_percent": 25, "subtotal": 30.75}
	1: {"course_id": "C09", "discount_percent": 95, "subtotal": 1.2}
	2: {"course_id": "C12", "discount_percent": 80, "subtotal": 6}
		array
		0: {"course_id": "C08", "subtotal_with_tax": 36.9}
		1: {"course_id": "C09", "subtotal_with_tax": 1.44}
		2: {"course_id": "C12", "subtotal_with_tax": 7.2}

000003910	[{"course_id":"C09","discount_percent":25,"subtotal":18}]	[{"course_id":"C09","subtotal_with_tax":21.6}]
000003525	[{"course_id":"C07","discount_percent":50,"subtotal":16.5},{"cours...	[{"course_id":"C07","subtotal_with_tax":19.8},{"course_
000004370	[{"course_id":"C08","discount_percent":65,"subtotal":14.35},{"cour	[{"course_id":"C08","subtotal_with_tax":17.22},{"course

Figure 4-47. The result of generating struct types with the TRANSFORM function

In summary, higher-order functions in Databricks, like `FILTER` and `TRANSFORM`, empower you to manipulate and extract specific information from complex data structures.

Developing SQL UDFs

SQL user-defined functions (UDFs) are a powerful way to encapsulate custom logic with a SQL-like syntax, making it reusable across different SQL queries. Unlike external UDFs written in Scala, Java, Python, or R, which appear as black boxes to the Spark Optimizer, SQL UDFs leverage Spark SQL directly. This typically provides better performance when applying custom logic to large datasets.

In this section, we'll explore the creation and usage of SQL UDFs in a new SQL notebook titled "4.5 - SQL UDFs." As we will continue using our online school dataset, we begin by running the "School-Setup" notebook to prepare our environment:

```
%run ../Includes/School-Setup
```

Creating UDFs

To create a SQL UDF, you need to specify a function name, optional parameters, the return type, and the custom logic. In the following example, we create a UDF named `gpa_to_percentage` for converting students' grade point average (GPA) scores into percentage equivalents. The UDF accepts a GPA score as a parameter of type `DOUBLE` and returns the percentage score as `integer`. The conversion logic assumes a GPA scale of 4.0, which is then translated into a percentage scale by multiplying it by 25. Additionally, the calculated percentage is rounded to the nearest integer using the `round` function and then cast to an integer data type:

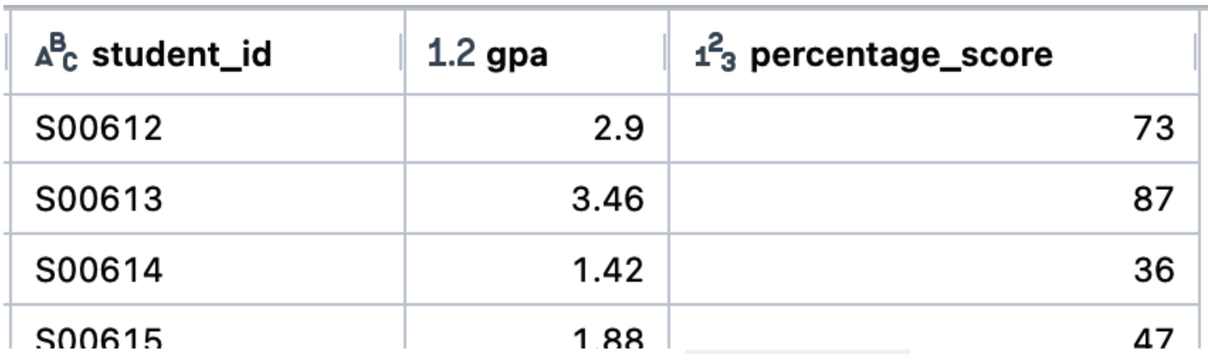
```
CREATE OR REPLACE FUNCTION gpa_to_percentage(gpa DOUBLE)
RETURNS INT

RETURN cast(round(gpa * 25) AS INT)
```

Applying UDFs

Once the UDF is created, you can use it in any SQL query like a native function. In the following example, we apply the `gpa_to_percentage` UDF on the `gpa` column within the `students` table:

```
SELECT student_id, gpa, gpa_to_percentage(gpa) AS percentage_score
FROM students
```

Figure 4-48 confirms that our function has been successfully applied. The column `percentage_score` accurately provides the equivalent percentage score for each student's GPA.



student_id	gpa	percentage_score
S00612	2.9	73
S00613	3.46	87
S00614	1.42	36
S00615	1.88	47

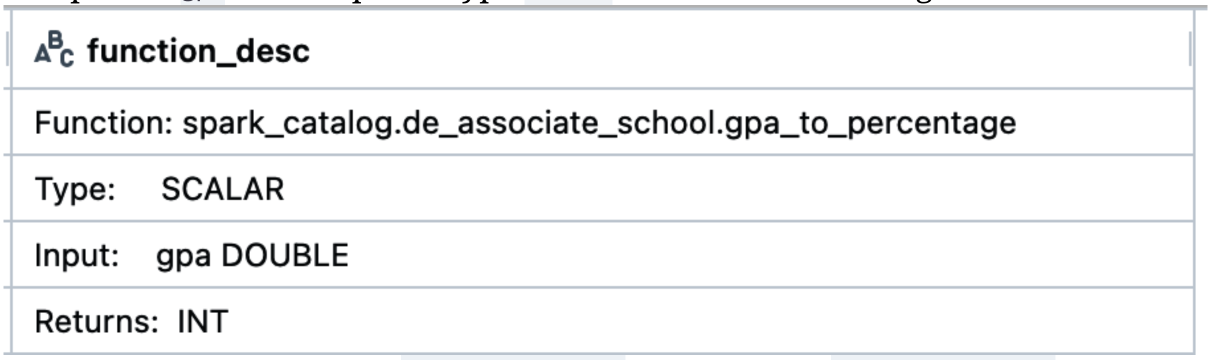
Figure 4-48. The result of applying the `gpa_to_percentage` UDF

Understanding UDFs

SQL UDFs are permanent objects stored in the database, allowing them to be used across different Spark sessions and notebooks. The `DESCRIBE FUNCTION` command provides basic information about the UDF, such as the database, input parameters, and return type:

```
DESCRIBE FUNCTION gpa_to_percentage
```

As shown in Figure 4-49, the function belongs to the `de_associate_school` database, created in the "School-Setup" notebook. It accepts the `gpa` as an input of type `DOUBLE` and returns an integer.



function_desc
Function: spark_catalog.de_associate_school.gpa_to_percentage
Type: SCALAR
Input: gpa DOUBLE
Returns: INT

Figure 4-49. The output of `DESCRIBE FUNCTION` command on the `gpa_to_percentage` UDF

Furthermore, the `DESCRIBE FUNCTION EXTENDED` command offers more details, including the SQL logic used in the function:

```
DESCRIBE FUNCTION EXTENDED gpa_to_percentage
```

Figure 4-50 displays some of the extended metadata information about our UDF. Specifically, the Body field reveals the SQL logic implemented within the function.

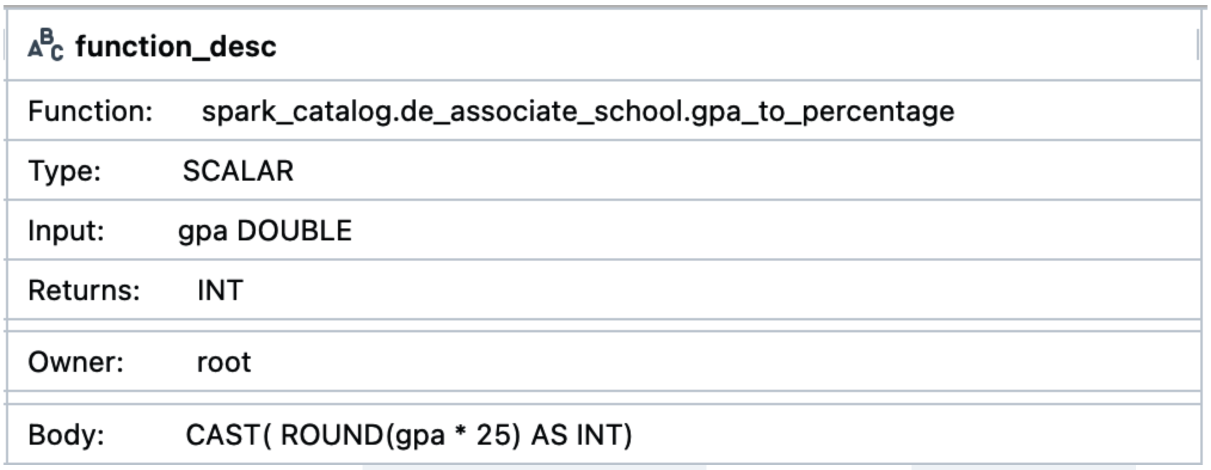

```
function_desc
Function:     spark_catalog.de_associate_school.gpa_to_percentage
Type:         SCALAR
Input:        gpa DOUBLE
Returns:      INT
Owner:        root
Body:         CAST( ROUND(gpa * 25) AS INT)
```

Figure 4-50. The output of the `DESCRIBE FUNCTION EXTENDED` command on the `gpa_to_percentage` UDF

Complex Logic UDFs

SQL UDFs can incorporate complex logic, such as using standard SQL `CASE WHEN` statements to evaluate multiple conditions. In the following example, we define a UDF that takes a student's GPA and returns its corresponding letter grade based on the grading scale in Table 4-2.

GPA (4.0 scale)	Grade letter
3.50–4.0	A
2.75–3.44	B
2.0–2.74	C
Below 2.0	F

Table 4-2. Grading scale

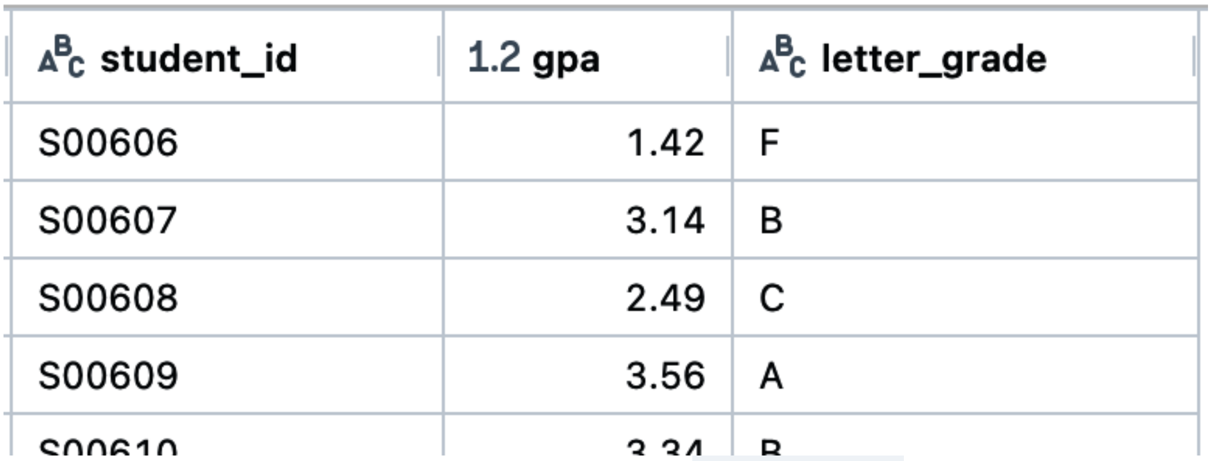
To map GPA scores to their corresponding letter grades, we use a `CASE WHEN` statement within a function named `get_letter_grade`:

```
CREATE OR REPLACE FUNCTION get_letter_grade(gpa DOUBLE)
RETURNS STRING
RETURN CASE
         WHEN gpa >= 3.5 THEN "A"
         WHEN gpa >= 2.75 AND gpa < 3.5 THEN "B"
         WHEN gpa >= 2 AND gpa < 2.75 THEN "C"
         ELSE "F"
      END
```

We can now apply this complex UDF to our dataset:

```
SELECT student_id, gpa, get_letter_grade(gpa) AS letter_grade
FROM students
```

Figure 4-51 confirms that we have successfully applied our UDF. As expected, the column `letter_grade` gives us the corresponding letter grade to each student's GPA.



student_id	gpa	letter_grade
S00606	1.42	F
S00607	3.14	B
S00608	2.49	C
S00609	3.56	A
S00610	3.34	B

Figure 4-51. The result of applying the `get_letter_grade` UDF

Thus, SQL UDFs in Databricks offer flexibility, reusability, and the ability to incorporate complex logic. All this, while benefiting from Spark's optimization for parallel execution.

Dropping UDFs

Finally, you can remove UDFs when they are no longer needed by using the `DROP FUNCTION` command:

```
DROP FUNCTION gpa_to_percentage;
DROP FUNCTION get_letter_grade;
```

After executing these commands, both UDFs will be completely removed from the database.

Conclusion

In conclusion, this chapter has highlighted key techniques for transforming data efficiently on the Databricks platform using Apache Spark. We've explored methods for querying and writing data, implemented advanced ETL operations, and leveraged the flexibility of higher-order functions and UDFs. With these tools, you are now equipped to perform robust data transformations that are both powerful and adaptable to diverse data processing needs.

Sample Exam Questions

Conceptual Question

1. A data engineer is tasked with replacing the content of a Delta table in a data pipeline. During a team meeting, they discuss the best approach for overwriting the table without disrupting ongoing analyses while ensuring optimal performance. The team considers two commands: `INSERT OVERWRITE` and `CREATE OR REPLACE TABLE`.
Given this scenario, which of the following factors should the data engineer consider when justifying the use of the `INSERT OVERWRITE` command over `CREATE OR REPLACE TABLE`?
1. The `INSERT OVERWRITE` operation is a more dynamic technique that ensures schema evolution when overwriting the table.
2. The `INSERT OVERWRITE` operation is a safer approach for overwriting the table without changing its schema.
3. The `INSERT OVERWRITE` operation can automatically optimize the table's layout for better query performance after the overwrite.
4. All of the above reasons explain why `INSERT OVERWRITE` is recommended over `CREATE OR REPLACE TABLE`.
5. None of the above! Both commands operate the same way.

Code-Based Question

2. A data engineer at a financial services company is tasked with creating a reusable SQL user-defined function (UDF). This function will calculate interest based on dynamic inputs across various datasets. The engineer needs to decide which code block would be appropriate for this task.

Which of the following code blocks should the data engineer use to create the SQL UDF?

1.
```
CREATE FUNCTION calc_interest(amount DOUBLE, rate DOUBLE)
    RETURNS cast(amount * rate AS DOUBLE);
```
2.
```
CREATE UDF calc_interest(amount DOUBLE, rate DOUBLE)
    RETURN amount * rate;
```
3.
```
CREATE UDF calc_interest(amount DOUBLE, rate DOUBLE)
    RETURNS DOUBLE
    RETURN amount * rate;
```
4.
```
CREATE FUNCTION calc_interest(amount DOUBLE, rate DOUBLE)
    RETURNS DOUBLE
    RETURN amount * rate;
```
5.
```
DEF calc_interest(amount DOUBLE, rate DOUBLE)
    RETURN cast(amount * rate AS DOUBLE);
```

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