17 Connecting Python application to MySQL

As most software applications need to interact with data in some form, programming languages like Python provide tools for storing and accessing these data sources.

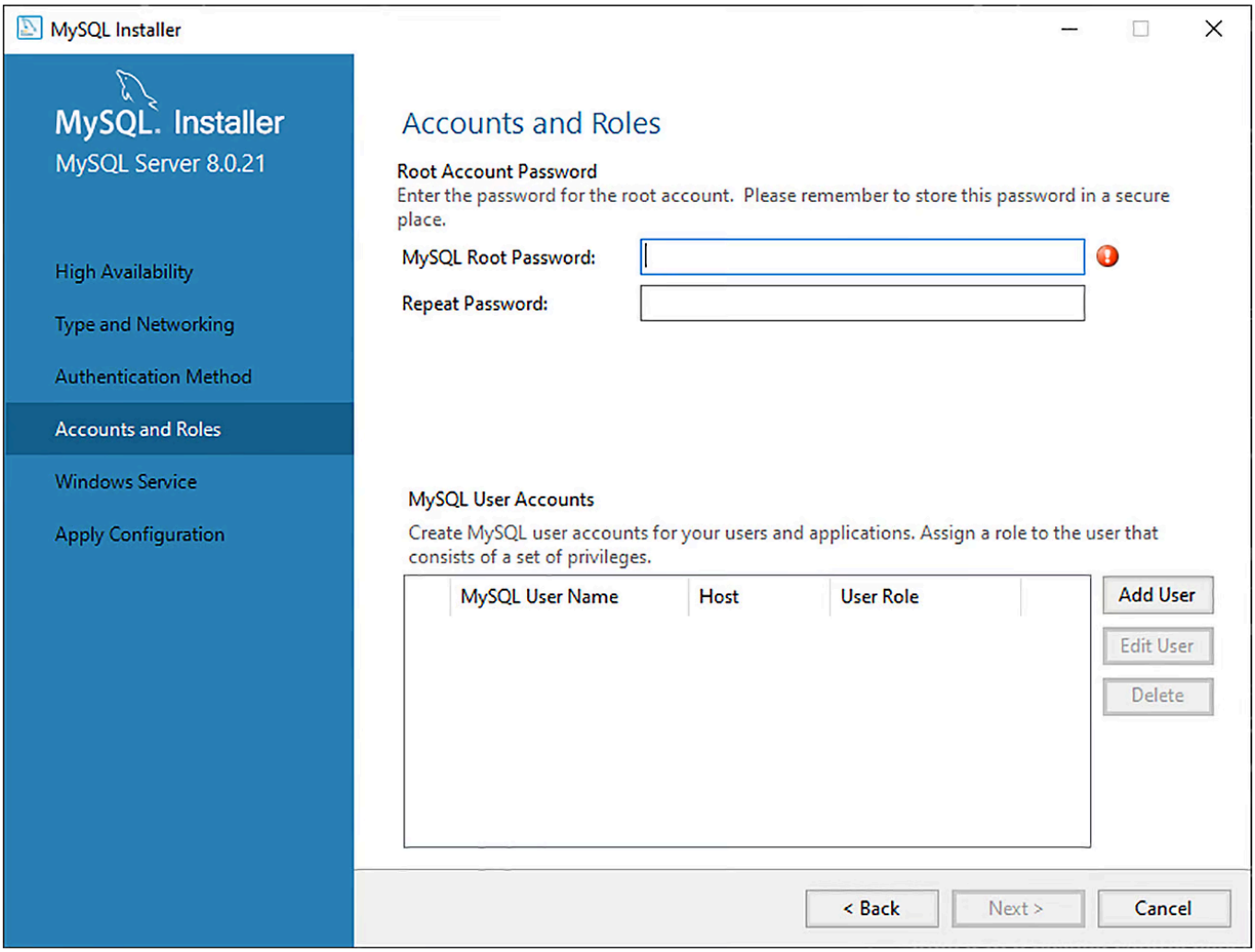
Installing MySQL Server and MySQL Connector

MySQL server will provide all the services required for handling your database. Once the server is up and running, you can connect your Python application with it using MySQL Connector/Python.

Installing MySQL Server

The official documentation details the recommended way to download and install MySQL server. You'll find instructions for all popular operating systems, including Windows, macOS, Solaris, Linux, and many more.

For Windows, the best way is to download MySQL Installer and let it take care of the entire process. The installation manager also helps you configure the security settings of the MySQL server. On the Accounts and Roles page, you need to enter a password for the **root** (admin) account and also optionally add other users with varying privileges.



Note: Remember the hostname, username, and password as these will be required to establish a connection with the MySQL server

You can also set up other helpful tools like MySQL Workbench using these installers. If you don't want to install MySQL directly in your operating system, then deploying MySQL on Linux with Docker is a convenient alternative.

Installing MySQL Connector

A database driver is a piece of software that allows an application to connect and interact with a database system. Programming languages like Python need a special driver before they can speak to a database from a specific vendor.

The **Python Database API** (DB-API) defines the standard interface with which all Python database drivers must comply.

you need to install a Python MySQL connector to interact with a MySQL database. Many packages follow the DB-API standards, but the most popular among them is MySQL Connector/Python. You can get it with `pip`

```
pip install mysql-connector-python
```

`pip` installs the connector as a third-party module in the currently active virtual environment.

To test if the installation was successful, type the following command on your Python terminal

```
>>> import mysql.connector
```

If the above code executes with no errors, then `mysql.connector` is installed and ready to use.

Establishing a Connection With MySQL Server

MySQL is a **server-based** database management system. One server might contain multiple databases. To interact with a database, you must first establish a connection with the server.

The general workflow of a Python program that interacts with a MySQL-based database is as follows:

1. Connect to the MySQL server.
2. Create a new database.
3. Connect to the newly created or an existing database.
4. Execute a SQL query and fetch results.
5. Inform the database if any changes are made to a table.
6. Close the connection to the MySQL server.

Establishing a Connection

The first step in interacting with a MySQL server is to establish a connection. To do this, you need `connect()` from the `mysql.connector` module. This function takes in parameters like `host`, `user`, and `password` and returns a `MySQLConnection` object. You can receive these credentials as input from the user and pass them to `connect()`

```
from getpass import getpass
from mysql.connector import connect, Error

try:
    with connect(
        host="localhost",
        user=input("Enter username: "),
        password=getpass("Enter password: "),
    ) as connection:
        print(connection)
```

```
except Error as e:
    print(e)
```

Creating a New Database

To create a new database, you need to execute a SQL statement

```
CREATE DATABASE your_db;
```

To execute a SQL query in Python, you'll need to use a cursor, which abstracts away the access to database records.

MySQL Connector/Python provides you with the `MySQLCursor` class, which instantiates objects that can execute MySQL queries in Python. An instance of the `MySQLCursor` class is also called a `cursor`.

To create a `cursor`, use the `.cursor()` method of your `connection` variable

```
cursor = connection.cursor()
```

A query that needs to be executed is sent to `cursor.execute()` in string format. In this, you'll send the `CREATE DATABASE` query to `cursor.execute()`

```
from getpass import getpass
from mysql.connector import connect, Error

try:
    with connect(
        host="localhost",
        user=input("Enter username: "),
        password=getpass("Enter password: "),
    ) as connection:
        create_db_query = "CREATE DATABASE online_movie_rating"
        with connection.cursor() as cursor:
            cursor.execute(create_db_query)
except Error as e:
    print(e)
```

You might receive an error here if a database with the same name already exists in your server. To confirm this, you can display the name of all databases in your server. Using the same `MySQLConnection` object from earlier, execute the `SHOW DATABASES` statement

```
>>> show_db_query = "SHOW DATABASES"
>>> with connection.cursor() as cursor:
...     cursor.execute(show_db_query)
...     for db in cursor:
...         print(db)
...
('information_schema',)
('mysql',)
('online_movie_rating',)
('performance_schema',)
('sys',)
```

The above code prints the names of all the databases currently in your MySQL server. The `SHOW DATABASES` command also outputs some databases that you didn't create in your server, like `information_schema`, `performance_schema`, and so on. These databases are generated automatically by the MySQL server and provide access to a variety of database metadata and MySQL server settings.

Connecting to an Existing Database

In many situations, you'll already have a MySQL database that you want to connect with your Python application.

You can do this using the same `connect()` function that you used earlier by sending an additional parameter called `database`

```
from getpass import getpass
from mysql.connector import connect, Error

try:
    with connect(
        host="localhost",
        user=input("Enter username: "),
        password=getpass("Enter password: "),
        database="online_movie_rating",
    ) as connection:
        print(connection)
except Error as e:
    print(e)
```

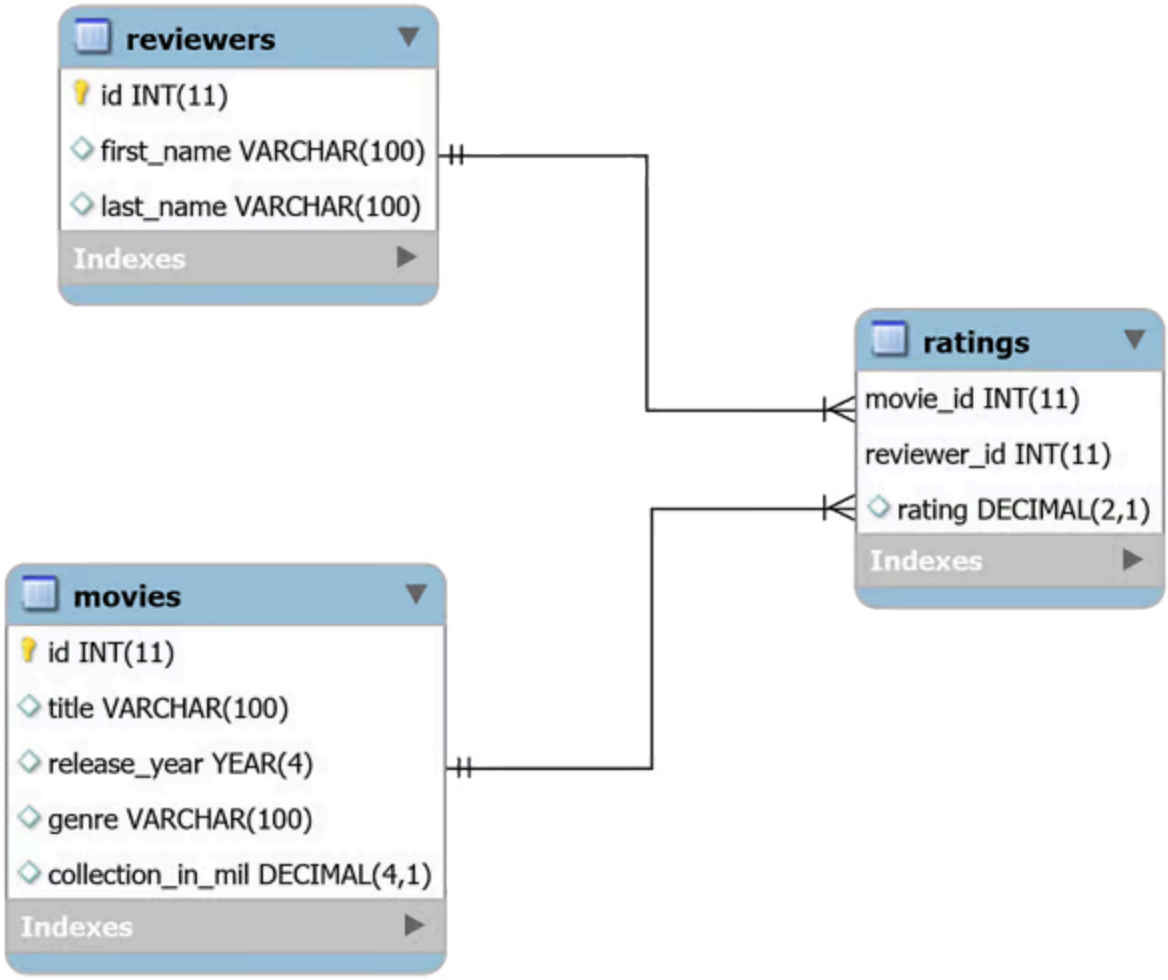
Creating, Altering, and Dropping a Table

In this section, you'll learn how to perform some basic DDL queries like `CREATE`, `DROP`, and `ALTER` with Python.

Defining the Database Schema

You can start by creating a database schema for an online movie rating system. The database will consist of three tables

1. **`movies`** contains general information about movies and has the following attributes:
    - `id`
    - `title`
    - `release_year`
    - `genre`
    - `collection_in_mil`
2. **`reviewers`** contains information about people who posted reviews or ratings and has the following attributes:
    - `id`
    - `first_name`
    - `last_name`
3. **`ratings`** contains information about ratings that have been posted and has the following attributes:
    - `movie_id` (foreign key)
    - `reviewer_id` (foreign key)
    - `rating`



The tables in this database are related to each other. `movies` and `reviewers` will have a **many-to-many** relationship since one movie can be reviewed by multiple reviewers and one reviewer can review multiple movies. The `ratings` table connects the `movies` table with the `reviewers` table.

Creating Tables Using the CREATE TABLE Statement

To create a new table, you need to pass this query to `cursor.execute()`, which accepts a MySQL query and executes the query on the connected MySQL database

```
create_movies_table_query = """
CREATE TABLE movies(
    id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(100),
    release_year YEAR(4),
    genre VARCHAR(100),
    collection_in_mil INT
)
"""
with connection.cursor() as cursor:
```

```
    cursor.execute(create_movies_table_query)
    connection.commit()
```

In MySQL, modifications mentioned in a transaction occur only when you use a `COMMIT` command in the end. Always call this method after every transaction to perform changes in the actual table.

Execute the following script to create the `reviewers` table

```
create_reviewers_table_query = """
CREATE TABLE reviewers (
    id INT AUTO_INCREMENT PRIMARY KEY,
    first_name VARCHAR(100),
    last_name VARCHAR(100)
)
"""
with connection.cursor() as cursor:
    cursor.execute(create_reviewers_table_query)
    connection.commit()
```

You can create the `ratings` table using the following script

```
create_ratings_table_query = """
CREATE TABLE ratings (
    movie_id INT,
    reviewer_id INT,
    rating DECIMAL(2,1),
    FOREIGN KEY(movie_id) REFERENCES movies(id),
    FOREIGN KEY(reviewer_id) REFERENCES reviewers(id),
    PRIMARY KEY(movie_id, reviewer_id)
)
"""
with connection.cursor() as cursor:
    cursor.execute(create_ratings_table_query)
    connection.commit()
```

The implementation of foreign key relationships in MySQL is slightly different and limited as compared to the standard SQL. In MySQL, both the parent and the child in the foreign key constraint must use the same **storage engine**.

A storage engine is the underlying software component that a database management system uses for performing SQL operations. In MySQL, storage engines come in two different flavors

1. **Transactional storage engines** are transaction safe and allow you to roll back transactions using simple commands like `rollback`. Many popular MySQL engines, including InnoDB and NDB, belong to this category.
2. **Nontransactional storage engines** depend on elaborate manual code to undo statements committed on a database. MyISAM, MEMORY, and many other MySQL engines are nontransactional.

Note that the `ratings` table uses the columns `movie_id` and `reviewer_id`, both foreign keys, jointly as the **primary key**. This step ensures that a reviewer can't rate the same movie twice.

You may choose to reuse the same cursor for multiple executions. In that case, all executions would become one atomic transaction rather than multiple separate transactions. For example, you can execute all `CREATE TABLE` statements with one cursor and then commit your transaction only once

```
with connection.cursor() as cursor:
    cursor.execute(create_movies_table_query)
    cursor.execute(create_reviewers_table_query)
    cursor.execute(create_ratings_table_query)
    connection.commit()
```

The above code will first execute all three `CREATE` statements. Then it will send a `COMMIT` command to the MySQL server that commits your transaction. You can also use `.rollback()` to send a `ROLLBACK` command to the MySQL server and remove all data changes from the transaction.

Showing a Table Schema Using the DESCRIBE statement

You can look at tables schema using the `DESCRIBE <table_name>;` SQL statement.

To get some results back from the `cursor` object, you need to use `cursor.fetchall()`. This method fetches all rows from the last executed statement.

```
>>> show_table_query = "DESCRIBE movies"
>>> with connection.cursor() as cursor:
...     cursor.execute(show_table_query)
...     # Fetch rows from last executed query
...     result = cursor.fetchall()
...     for row in result:
...         print(row)
...
('id', 'int(11)', 'NO', 'PRI', None, 'auto_increment')
```

```
('title', 'varchar(100)', 'YES', '', None, '')
('release_year', 'year(4)', 'YES', '', None, '')
('genre', 'varchar(100)', 'YES', '', None, '')
('collection_in_mil', 'int(11)', 'YES', '', None, '')
```

Modifying a Table Schema Using the ALTER Statement

You can write the following MySQL statement to modify the data type of `collection_in_mil` attribute from `INT` to `DECIMAL`

```
>>> alter_table_query = """
... ALTER TABLE movies
... MODIFY COLUMN collection_in_mil DECIMAL(4,1)
... """
>>> show_table_query = "DESCRIBE movies"
>>> with connection.cursor() as cursor:
...     cursor.execute(alter_table_query)
...     cursor.execute(show_table_query)
...     # Fetch rows from last executed query
...     result = cursor.fetchall()
...     print("Movie Table Schema after alteration:")
...     for row in result:
...         print(row)
...
Movie Table Schema after alteration
('id', 'int(11)', 'NO', 'PRI', None, 'auto_increment')
('title', 'varchar(100)', 'YES', '', None, '')
('release_year', 'year(4)', 'YES', '', None, '')
('genre', 'varchar(100)', 'YES', '', None, '')
('collection_in_mil', 'decimal(4,1)', 'YES', '', None, '')
```

`DECIMAL(4,1)` means a decimal number that can have a maximum of `4` digits, of which `1` is decimal, such as `120.1`, `3.4`, `38.0`, and so on.

Deleting Tables Using the DROP Statement

To delete a table, you need to execute the `DROP TABLE` statement in MySQL. Deleting a table is an *irreversible* process. If you execute the code below, then you'll need to call the `CREATE TABLE` query again to use the `ratings` table in the upcoming sections.

To delete the `ratings` table

```
drop_table_query = "DROP TABLE ratings"
with connection.cursor() as cursor:
```

```
    cursor.execute(drop_table_query)
```

Inserting Records in Tables

The first method, `.execute()`, works well when the number of records is small and the records can be hard-coded. The second method, `.executemany()`, is more popular and is better suited for real-world scenarios.

Using `.execute()`

You write the `INSERT INTO` query in a string and pass it to `cursor.execute()`. You can use this method to insert data into the `movies` table.

```
insert_movies_query = """
INSERT INTO movies (title, release_year, genre, collection_in_mil)
VALUES
    ("Forrest Gump", 1994, "Drama", 330.2),
    ("3 Idiots", 2009, "Drama", 2.4),
    ("Eternal Sunshine of the Spotless Mind", 2004, "Drama", 34.5),
    ("Good Will Hunting", 1997, "Drama", 138.1),
    ("Skyfall", 2012, "Action", 304.6),
    ("Gladiator", 2000, "Action", 188.7),
    ("Black", 2005, "Drama", 3.0),
    ("Titanic", 1997, "Romance", 659.2),
    ("The Shawshank Redemption", 1994, "Drama",28.4),
    ("Udaan", 2010, "Drama", 1.5),
    ("Home Alone", 1990, "Comedy", 286.9),
    ("Casablanca", 1942, "Romance", 1.0),
    ("Avengers: Endgame", 2019, "Action", 858.8),
    ("Night of the Living Dead", 1968, "Horror", 2.5),
    ("The Godfather", 1972, "Crime", 135.6),
    ("Haider", 2014, "Action", 4.2),
    ("Inception", 2010, "Adventure", 293.7),
    ("Evil", 2003, "Horror", 1.3),
    ("Toy Story 4", 2019, "Animation", 434.9),
    ("Air Force One", 1997, "Drama", 138.1),
    ("The Dark Knight", 2008, "Action",535.4),
    ("Bhaag Milkha Bhaag", 2013, "Sport", 4.1),
    ("The Lion King", 1994, "Animation", 423.6),
    ("Pulp Fiction", 1994, "Crime", 108.8),
    ("Kai Po Che", 2013, "Sport", 6.0),
    ("Beasts of No Nation", 2015, "War", 1.4),
    ("Andadhun", 2018, "Thriller", 2.9),
    ("The Silence of the Lambs", 1991, "Crime", 68.2),
    ("Deadpool", 2016, "Action", 363.6),
```

```
    ("Drishyam", 2015, "Mystery", 3.0)
"""
with connection.cursor() as cursor:
    cursor.execute(insert_movies_query)
    connection.commit()
```

Using `.executemany()`

You'll often have this data stored in some other file, or the data will be generated by a different script and will need to be added to the MySQL database.

This is where `.executemany()` comes in handy. It accepts two parameters:

1. A **query** that contains placeholders for the records that need to be inserted
2. A **list** that contains all records that you wish to insert

```
insert_reviewers_query = """
INSERT INTO reviewers
(first_name, last_name)
VALUES ( %s, %s )
"""
reviewers_records = [
    ("Chaitanya", "Baweja"),
    ("Mary", "Cooper"),
    ("John", "Wayne"),
    ("Thomas", "Stoneman"),
    ("Penny", "Hofstadter"),
    ("Mitchell", "Marsh"),
    ("Wyatt", "Skaggs"),
    ("Andre", "Veiga"),
    ("Sheldon", "Cooper"),
    ("Kimbra", "Masters"),
    ("Kat", "Dennings"),
    ("Bruce", "Wayne"),
    ("Domingo", "Cortes"),
    ("Rajesh", "Koothrappali"),
    ("Ben", "Glocker"),
    ("Mahinder", "Dhoni"),
    ("Akbar", "Khan"),
    ("Howard", "Wolowitz"),
    ("Pinkie", "Petit"),
    ("Gurkaran", "Singh"),
    ("Amy", "Farah Fowler"),
    ("Marlon", "Crafford"),
]
```

```
with connection.cursor() as cursor:
    cursor.executemany(insert_reviewers_query, reviewers_records)
    connection.commit()
```

The code uses `%s` as a placeholder for the two strings that had to be inserted in the `insert_reviewers_query`. Placeholders act as format specifiers and help reserve a spot for a variable inside a string. The specified variable is then added to this spot during execution.

You can similarly use `.executemany()` to insert records in the `ratings` table

```
insert_ratings_query = """
INSERT INTO ratings
(rating, movie_id, reviewer_id)
VALUES ( %s, %s, %s)
"""
ratings_records = [
    (6.4, 17, 5), (5.6, 19, 1), (6.3, 22, 14), (5.1, 21, 17),
    (5.0, 5, 5), (6.5, 21, 5), (8.5, 30, 13), (9.7, 6, 4),
    (8.5, 24, 12), (9.9, 14, 9), (8.7, 26, 14), (9.9, 6, 10),
    (5.1, 30, 6), (5.4, 18, 16), (6.2, 6, 20), (7.3, 21, 19),
    (8.1, 17, 18), (5.0, 7, 2), (9.8, 23, 3), (8.0, 22, 9),
    (8.5, 11, 13), (5.0, 5, 11), (5.7, 8, 2), (7.6, 25, 19),
    (5.2, 18, 15), (9.7, 13, 3), (5.8, 18, 8), (5.8, 30, 15),
    (8.4, 21, 18), (6.2, 23, 16), (7.0, 10, 18), (9.5, 30, 20),
    (8.9, 3, 19), (6.4, 12, 2), (7.8, 12, 22), (9.9, 15, 13),
    (7.5, 20, 17), (9.0, 25, 6), (8.5, 23, 2), (5.3, 30, 17),
    (6.4, 5, 10), (8.1, 5, 21), (5.7, 22, 1), (6.3, 28, 4),
    (9.8, 13, 1)
]
with connection.cursor() as cursor:
    cursor.executemany(insert_ratings_query, ratings_records)
    connection.commit()
```

Reading Records From the Database

You can read records using SQL SELECT statement

Reading Records Using the SELECT Statement

To retrieve records, you need to send a `SELECT` query to `cursor.execute()`. Then you use `cursor.fetchall()` to extract the retrieved table in the form of a list of rows or records.

```
>>> select_movies_query = "SELECT * FROM movies LIMIT 5"
>>> with connection.cursor() as cursor:
```

```
...     cursor.execute(select_movies_query)
...     result = cursor.fetchall()
...     for row in result:
...         print(row)
...
(1, 'Forrest Gump', 1994, 'Drama', Decimal('330.2'))
(2, '3 Idiots', 2009, 'Drama', Decimal('2.4'))
(3, 'Eternal Sunshine of the Spotless Mind', 2004, 'Drama', Decimal('34.5'))
(4, 'Good Will Hunting', 1997, 'Drama', Decimal('138.1'))
(5, 'Skyfall', 2012, 'Action', Decimal('304.6'))
```

In the query above, you use the `LIMIT` clause to constrain the number of rows that are received from the `SELECT` statement. Developers often use `LIMIT` to perform pagination when handling large volumes of data.

In MySQL, the `LIMIT` clause takes one or two nonnegative numeric arguments. When using one argument, you specify the maximum number of rows to return. Since your query includes `LIMIT 5`, only the first `5` records are fetched. When using both arguments, you can also specify the **offset** of the first row to return

```
SELECT * FROM movies LIMIT 2,5;
```

The first argument specifies an offset of `2`, and the second argument constrains the number of returned rows to `5`. The above query will return rows 3 to 7.

Filtering Results Using the WHERE Clause

You can filter table records by specific criteria using the `WHERE` clause. For example, to retrieve all movies with a box office collection greater than $300 million, you could run the following query along with ORDER BY clause n the last query to sort the results from the highest to the lowest earner

```
>>> select_movies_query = """
... SELECT title, collection_in_mil
... FROM movies
... WHERE collection_in_mil > 300
... ORDER BY collection_in_mil DESC
... """
>>> with connection.cursor() as cursor:
...     cursor.execute(select_movies_query)
...     for movie in cursor.fetchall():
...         print(movie)
...
```

```
('Avengers: Endgame', Decimal('858.8'))
('Titanic', Decimal('659.2'))
('The Dark Knight', Decimal('535.4'))
('Toy Story 4', Decimal('434.9'))
('The Lion King', Decimal('423.6'))
('Deadpool', Decimal('363.6'))
('Forrest Gump', Decimal('330.2'))
('Skyfall', Decimal('304.6'))
```

MySQL offers a plethora of string formatting operations like `CONCAT` for concatenating strings. Often, websites will show the movie title along with its release year to avoid confusion. To retrieve the titles of the top five grossing movies, concatenated with their release years, you can write the following query

```
>>> select_movies_query = """
... SELECT CONCAT(title, " (", release_year, ")"),
...        collection_in_mil
... FROM movies
... ORDER BY collection_in_mil DESC
... LIMIT 5
... """
>>> with connection.cursor() as cursor:
...     cursor.execute(select_movies_query)
...     for movie in cursor.fetchall():
...         print(movie)
...
('Avengers: Endgame (2019)', Decimal('858.8'))
('Titanic (1997)', Decimal('659.2'))
('The Dark Knight (2008)', Decimal('535.4'))
('Toy Story 4 (2019)', Decimal('434.9'))
('The Lion King (1994)', Decimal('423.6'))
```

If you don't want to use the `LIMIT` clause and you don't need to fetch all the records, then the `cursor` object has `.fetchone()` and `.fetchmany()` methods as well:

- **`.fetchone()`** retrieves either the next row of the result, as a tuple, or `None` if no more rows are available.
- **`.fetchmany()`** retrieves the next set of rows from the result as a list of tuples. It has a `size` argument, which defaults to `1`, that you can use to specify the number of rows you need to fetch. If no more rows are available, then the method returns an empty list. Try retrieving the titles of the five highest-grossing movies concatenated with their release years again, but this time use `.fetchmany()`

```
>>> select_movies_query = """
... SELECT CONCAT(title, " (", release_year, ")"),
...       collection_in_mil
... FROM movies
... ORDER BY collection_in_mil DESC
... """
>>> with connection.cursor() as cursor:
...     cursor.execute(select_movies_query)
...     for movie in cursor.fetchmany(size=5):
...         print(movie)
...     cursor.fetchall()
...
('Avengers: Endgame (2019)', Decimal('858.8'))
('Titanic (1997)', Decimal('659.2'))
('The Dark Knight (2008)', Decimal('535.4'))
('Toy Story 4 (2019)', Decimal('434.9'))
('The Lion King (1994)', Decimal('423.6'))
```

Handling Multiple Tables Using the JOIN Statement

If you want to find out the name of the top five highest-rated movies in your database, then you can run the following query

```
>>> select_movies_query = """
... SELECT title, AVG(rating) as average_rating
... FROM ratings
... INNER JOIN movies
...     ON movies.id = ratings.movie_id
... GROUP BY movie_id
... ORDER BY average_rating DESC
... LIMIT 5
... """
>>> with connection.cursor() as cursor:
...     cursor.execute(select_movies_query)
...     for movie in cursor.fetchall():
...         print(movie)
...
('Night of the Living Dead', Decimal('9.90000'))
('The Godfather', Decimal('9.90000'))
('Avengers: Endgame', Decimal('9.75000'))
('Eternal Sunshine of the Spotless Mind', Decimal('8.90000'))
('Beasts of No Nation', Decimal('8.70000'))
```

To find the name of the reviewer who gave the most ratings, write the following query

```
>>> select_movies_query = """
... SELECT CONCAT(first_name, " ", last_name), COUNT(*) as num
... FROM reviewers
... INNER JOIN ratings
...     ON reviewers.id = ratings.reviewer_id
... GROUP BY reviewer_id
... ORDER BY num DESC
... LIMIT 1
... """
>>> with connection.cursor() as cursor:
...     cursor.execute(select_movies_query)
...     for movie in cursor.fetchall():
...         print(movie)
...
('Mary Cooper', 4)
```

Updating and Deleting Records From the Database

Update and Delete operations can be performed on either a single record or multiple records in the table.

UPDATE Command

For updating records, MySQL uses the `UPDATE` statement

```python
update_query = """
UPDATE
reviewers
SET
last_name = "Cooper"
WHERE
first_name = "Amy"
"""
with connection.cursor() as cursor:
cursor.execute(update_query)
connection.commit()
```

```
**Note:** In the `UPDATE` query, the `WHERE` clause helps specify the
records that need to be updated. If you don't use `WHERE`, then all records
will be updated!

Suppose you need to provide an option that allows reviewers to modify
```

ratings. A reviewer will provide three values, `movie_id`, `reviewer_id`, and the new `rating`.

```python
from getpass import getpass
from mysql.connector import connect, Error

movie_id = input("Enter movie id: ")
reviewer_id = input("Enter reviewer id: ")
new_rating = input("Enter new rating: ")
update_query = """
UPDATE
    ratings
SET
    rating = %s
WHERE
    movie_id = %s AND reviewer_id = %s;

SELECT *
FROM ratings
WHERE
    movie_id = %s AND reviewer_id = %s
"""
val_tuple = (
    new_rating,
    movie_id,
    reviewer_id,
    movie_id,
    reviewer_id,
)

try:
    with connect(
        host="localhost",
        user=input("Enter username: "),
        password=getpass("Enter password: "),
        database="online_movie_rating",
    ) as connection:
        with connection.cursor() as cursor:
            for result in cursor.execute(update_query, val_tuple,
multi=True):
```

```
                if result.with_rows:
                    print(result.fetchall())
            connection.commit()
except Error as e:
    print(e)
```

To pass multiple queries to a single `cursor.execute()`, you need to set the method's `multi` argument to `True`.

If no result set is fetched on an operation, then `.fetchall()` raises an exception. To avoid this error, in the code above you use the `cursor.with_rows` property, which indicates whether the most recently executed operation produced rows.

DELETE Command

Deleting records works very similarly to updating records. You use the `DELETE` statement to remove selected records.

Note: Deleting is an *irreversible* process. If you don't use the `WHERE` clause, then all records from the specified table will be deleted. You'll need to run the `INSERT INTO` query again to get back the deleted records.

For example, to remove all ratings given by `reviewer_id = 2`, you should first run the corresponding `SELECT` query

```
>>> select_movies_query = """
... SELECT reviewer_id, movie_id FROM ratings
... WHERE reviewer_id = 2
... """
>>> with connection.cursor() as cursor:
...     cursor.execute(select_movies_query)
...     for movie in cursor.fetchall():
...         print(movie)
...
(2, 7)
(2, 8)
(2, 12)
(2, 23)

delete_query = "DELETE FROM ratings WHERE reviewer_id = 2"
with connection.cursor() as cursor:
```

```
    cursor.execute(delete_query)
    connection.commit()
```