SELECTING DATA- SQLITE

we will explore essential techniques for working with SQLite databases, focusing on establishing a connection, querying data, and exploring database structure. SQLite, a lightweight database engine, which allows you to run SQL queries within a file on your computer, making it ideal for local data analysis tasks.

We will cover:

1. **Connecting to SQLite Databases**: How to establish a `connection` to the database and use a `cursor` to execute SQL commands.
2. **Basic Data Retrieval**: Writing `SELECT` queries to retrieve data, using filters with `WHERE`, sorting with `ORDER BY`, and limiting results with `LIMIT`.
3. **Schema Exploration**: Using the `PRAGMA table_info(table_name)` command to inspect table structure, view column details, and ensure data is correctly typed.
4. **Modifying Database Structure**: Adding new columns with `ALTER TABLE` to support correctly-typed data for optimized analysis.

In [1]:

```
# List the files and directories in the current working directory
!ls
```

```
Database-Schema.png
SQL.ipynb
SQLite
connecting_database_using_python
data.sqlite
```

we have a file extension `.sqlite` but you will also see examples ending with `.db`

Read the data.sqlite file. Reading the file without using any library gives us a bunch of garbled nonsense

In [2]:

```
# Reading the file without using any library
with open("data.sqlite", "rb") as f:
    print(f.read(100))
```

```
b'SQLite format 3\x00\x10\x00\x01\x01\x00@  \x00\x00\x00\x12\x00\x00\x009\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x00\t\x00\x00\x00\x04\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x12\x00.G\x
a8'
```

Connection

we will use the `sqlite3` module (sqlite module). The way that this module works is that we start by opening a *connection* to the database with `sqlite3.connect`:

In [3]:

```python
#  imports the sqlite3 which provides tools to work with SQLite database
import sqlite3
# establishing connection to SQLite database
conn = sqlite3.connect('data.sqlite')
```

NB: **If the file doesn't exist, SQLite will create it in the current directory**

In [4]:

```python
# checking on connection attributes
print("data type:", type(conn))
print("uncommitted changes:", conn.in_transaction)
print("total changes:", conn.total_changes)
```

```
data type: <class 'sqlite3.Connection'>
uncommitted changes: False
total changes: 0
```

data type is sqlite3 object with no changes, meaning not performed any queries.

Cursor

A cursor in SQL allows for row-by-row processing, which is useful for tasks that require individual row handling or complex operations that aren't easy with set-based commands.

you create by calling `.cursor`

In [5]:

```python
# creating a cursor object
cur = conn.cursor()
print("data type:", type(cur))
```

```
data type: <class 'sqlite3.Cursor'>
```

Exploring Schema using Cursor

we can cursor to know what tables are contains in the database.This requires two steps:

1. Executing the query (`.execute()`)
2. Fetching the results (`.fetchone()` , `.fetchmany()` , or `.fetchall()`)

In [6]:

```python
# excute query
cur.execute("""SELECT name FROM sqlite_master WHERE type = 'table';""")
# fetch the result and store in table_names
table_names =cur.fetchall()
table_names
```

Out[6]:

```
[('orderdetails',),
 ('payments',),
 ('offices',),
 ('customers',),
 ('orders',),
 ('productlines',),
 ('products',),
 ('employees',)]
```

```
In [7]: #  getting the schema for employees
        cur.execute("""SELECT sql FROM sqlite_master WHERE type ='table' AND name = 'employ
        employees_schema = cur.fetchall()
        employees_schema
```

```
Out[7]: [('CREATE TABLE `employees` (`employeeNumber`, `lastName`, `firstName`, `extension
        `, `email`, `officeCode`, `reportsTo`, `jobTitle`)',)]
```

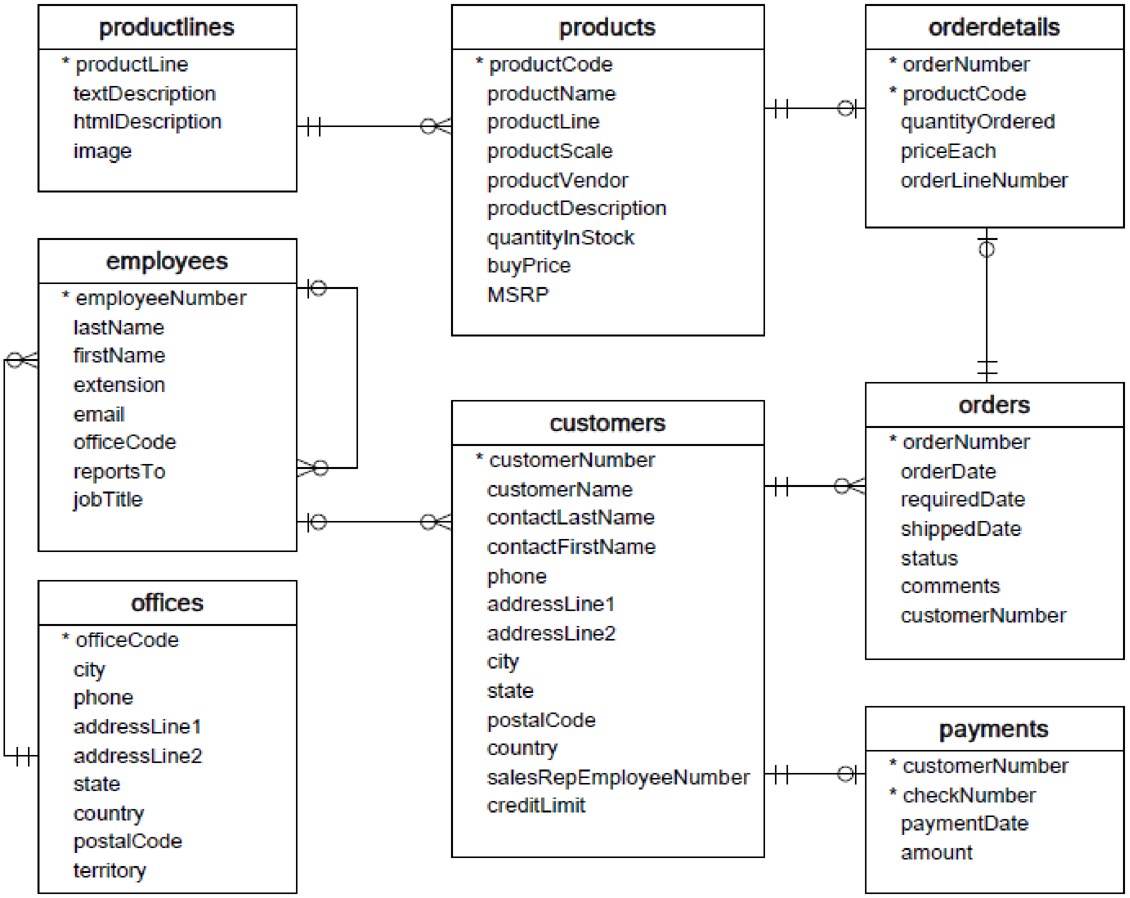
```
In [8]: # getting the schema for customers
        cur.execute("""SELECT sql FROM sqlite_master WHERE type= 'table' AND name='customer
        customers_schema = cur.fetchall()
        customers_schema
```

```
Out[8]: [('CREATE TABLE `customers` (`customerNumber`, `customerName`, `contactLastName`,
        `contactFirstName`, `phone`, `addressLine1`, `addressLine2`, `city`, `state`, `pos
        talCode`, `country`, `salesRepEmployeeNumber`, `creditLimit`, creditLimitNumeric R
        EAL)',)]
```

In summary

- *Connection and Cursor Creation*: This sets up our database interaction.
- *Schema Query*: The query "SELECT name FROM sqlite_master WHERE type='table';" retrieves the names of all tables in the database.
- *Fetch and Print*: We fetch all results and loop through them to print each table name.

ERD OVERVIEW



SELECT Clause

In [9]:

```
# getting information abt first 5 orders
"""`*` Means all columns """
cur.execute("""SELECT * FROM orders LIMIT 5;""")
cur.fetchall()
```

Out[9]:

```
[('10100', '2003-01-06', '2003-01-13', '2003-01-10', 'Shipped', '', '363'),
 ('10101',
  '2003-01-09',
  '2003-01-18',
  '2003-01-11',
  'Shipped',
  'Check on availability.',
  '128'),
 ('10102', '2003-01-10', '2003-01-18', '2003-01-14', 'Shipped', '', '181'),
 ('10103', '2003-01-29', '2003-02-07', '2003-02-02', 'Shipped', '', '121'),
 ('10104', '2003-01-31', '2003-02-09', '2003-02-01', 'Shipped', '', '141')]
```

Because `.execute()` returns the cursor object, it also possible to combine the previous two lines into one line, like so:

In [10]:

```
cur.execute("""SELECT * FROM offices LIMIT 5;""").fetchall()
```

```
Out[10]: [('1',
           'San Francisco',
           '+1 650 219 4782',
           '100 Market Street',
           'Suite 300',
           'CA',
           'USA',
           '94080',
           'NA'),
          ('2',
           'Boston',
           '+1 215 837 0825',
           '1550 Court Place',
           'Suite 102',
           'MA',
           'USA',
           '02107',
           'NA'),
          ('3',
           'NYC',
           '+1 212 555 3000',
           '523 East 53rd Street',
           'apt. 5A',
           'NY',
           'USA',
           '10022',
           'NA'),
          ('4',
           'Paris',
           '+33 14 723 4404',
           "43 Rue Jouffroy D'abbans",
           '',
           '',
           'France',
           '75017',
           'EMEA'),
          ('5',
           'Tokyo',
           '+81 33 224 5000',
           '4-1 Kioicho',
           '',
           'Chiyoda-Ku',
           'Japan',
           '102-8578',
           'Japan')]
```

For readability we can also adopt the following lines of codes

```
In [11]: First_3_employees = """
         SELECT *
         FROM employees
         LIMIT 3
         ;
         """
         cur.execute(First_3_employees).fetchall()
```

```
Out[11]: [('1002',
           'Murphy',
           'Diane',
           'x5800',
           'dmurphy@classicmodelcars.com',
           '1',
           '',
           'President'),
          ('1056',
           'Patterson',
           'Mary',
           'x4611',
           'mpatterso@classicmodelcars.com',
           '1',
           '1002',
           'VP Sales'),
          ('1076',
           'Firrelli',
           'Jeff',
           'x9273',
           'jfirrelli@classicmodelcars.com',
           '1',
           '1002',
           'VP Marketing')]
```

Structuring Results as Pandas DataFrames

In many cases, a more practical output format is to convert these results into Pandas DataFrames. An approach for doing this is to pass the `c.fetchall()` output into a Pandas DataFrame constructor.

```
In [12]: # importing pandas
         import pandas as pd
         # passing result into Pandas DataFrame
         df = pd.DataFrame(cur.execute(First_3_employees).fetchall())
         df
```

Out[12]:

	0	1	2	3	4	5	6	7
0	1002	Murphy	Diane	x5800	dmurphy@classicmodelcars.com	1		President
1	1056	Patterson	Mary	x4611	mpatterso@classicmodelcars.com	1	1002	VP Sales
2	1076	Firrelli	Jeff	x9273	jfirrelli@classicmodelcars.com	1	1002	VP Marketing

we can now access the columns names using `cur.description`

```
In [13]: cur.description
```

Out[13]:

```
(('employeeNumber', None, None, None, None, None, None),
 ('lastName', None, None, None, None, None, None),
 ('firstName', None, None, None, None, None, None),
 ('extension', None, None, None, None, None, None),
 ('email', None, None, None, None, None, None),
 ('officeCode', None, None, None, None, None, None),
 ('reportsTo', None, None, None, None, None, None),
 ('jobTitle', None, None, None, None, None, None))
```

Following the DataFrame creation, use a list comprehension to define the column names:

`df.columns = [x[0] for x in cur.description]`

In [14]:

```python
df.columns = [x[0] for x in cur.description]
df
```

Out[14]:

	employeeNumber	lastName	firstName	extension	email	offi
0	1002	Murphy	Diane	x5800	dmurphy@classicmodelcars.com	
1	1056	Patterson	Mary	x4611	mpatterso@classicmodelcars.com	
2	1076	Firrelli	Jeff	x9273	jfirrelli@classicmodelcars.com	

Pandas provides a method specifically for reading from SQL databases (reading from SQL database). Rather than using the cursor, you only need the connection object:

The code executes the SQL query to fetch data from the database and loads it directly into a Pandas DataFrame for further analysis.

`conn` connection object allows Pandas to communicate with the database and run the query.

In [15]:

```python
df = pd.read_sql(First_3_employees, conn)
df
```

Out[15]:

	employeeNumber	lastName	firstName	extension	email	offi
0	1002	Murphy	Diane	x5800	dmurphy@classicmodelcars.com	
1	1056	Patterson	Mary	x4611	mpatterso@classicmodelcars.com	
2	1076	Firrelli	Jeff	x9273	jfirrelli@classicmodelcars.com	

NB we can also use SELECT to select only certain columns, and those will be reflected in the dataframe column names:

In [16]:
```
df = pd.read_sql("""SELECT employeeNumber, lastName, firstName FROM employees LIMIT
df
```

Out[16]:

	employeeNumber	lastName	firstName
0	1002	Murphy	Diane
1	1056	Patterson	Mary
2	1076	Firrelli	Jeff
3	1088	Patterson	William
4	1102	Bondur	Gerard
5	1143	Bow	Anthony
6	1165	Jennings	Leslie
7	1166	Thompson	Leslie
8	1188	Firrelli	Julie
9	1216	Patterson	Steve

WHERE CLAUSE

`WHERE` clause filters `SELECT` query results by some **`condition`**.

Syntax

`SELECT column1, column2, ... FROM table_name WHERE condition;"""`

Key Points:

- The WHERE clause follows the FROM clause in a SQL query.
- It is used to filter the rows returned by the query based on the condition(s) provided.
- Conditions can involve comparison operators (`=, !=, <, >, <=, >=`), logical operators (`AND` , `OR` , `NOT`), and more complex expressions.

1. Selecting Customers from a Specific City `Boston`

In [17]:
```
df = pd.read_sql("""SELECT * FROM customers WHERE city = 'Boston' """, conn)
df
```

Out[17]:

	customerNumber	customerName	contactLastName	contactFirstName	phone	ac
0	362	Gifts4AllAges.com	Yoshido	Juri	6175559555	S
1	495	Diecast Collectables	Franco	Valarie	6175552555	

2. Selecting Multiple Cities `Boston` `Madrid`

In [18]:

```
df = pd.read_sql("""SELECT * FROM customers WHERE city = 'Boston' OR city = 'Madrid
df
```

Out[18]:

	customerNumber	customerName	contactLastName	contactFirstName	phone	ac
0	141	Euro+ Shopping Channel	Freyre	Diego	(91) 555 94 44	
1	237	ANG Resellers	Camino	Alejandra	(91) 745 6555	
2	344	CAF Imports	Fernandez	Jesus	+34 913 728 555	
3	362	Gifts4AllAges.com	Yoshido	Juri	6175559555	S
4	458	Corrida Auto Replicas, Ltd	Sommer	Martín	(91) 555 22 82	C,
5	465	Anton Designs, Ltd.	Anton	Carmen	+34 913 728555	
6	495	Diecast Collectables	Franco	Valarie	6175552555	

3. Customers with a creditLimit greater than 50,000

In [19]:

```
df = pd.read_sql("""SELECT * FROM customers WHERE creditLimit > 50000 """, conn)
df
```

Out[19]:

	customerNumber	customerName	contactLastName	contactFirstName	phone	a
0	103	Atelier graphique	Schmitt	Carine	40.32.2555	
1	112	Signal Gift Stores	King	Jean	7025551838	
2	114	Australian Collectors, Co.	Ferguson	Peter	03 9520 4555	
3	119	La Rochelle Gifts	Labrune	Janine	40.67.8555	
4	121	Baane Mini Imports	Bergulfsen	Jonas	07-98 9555	
...	...	...	...	...	...	
117	486	Motor Mint Distributors Inc.	Salazar	Rosa	2155559857	
118	487	Signal Collectibles Ltd.	Taylor	Sue	4155554312	
119	489	Double Decker Gift Stores, Ltd	Smith	Thomas	(171) 555-7555	
120	495	Diecast Collectables	Franco	Valarie	6175552555	
121	496	Kelly's Gift Shop	Snowden	Tony	+64 9 5555500	

122 rows × 14 columns

4. find customers who have a creditLImit greater than 50,000 and work in the 'Sales' department also who have either a salary greater than 50,000 or work in the 'Sales' department

The ORDER BY and LIMIT Clauses

- `ORDER BY` is used to sort the results.
- `LIMIT` is used to restrict the number of rows returned.
- Together, they allow you to fetch a specific subset of ordered data, which is helpful for getting top results or paginating through data.

ORDER BY Syntax

`SELECT column1, column2, ... FROM table_name ORDER BY column1 [ASC|DESC], column2 [ASC|DESC], ...;`

`ASC` : Sorts in ascending order (from lowest to highest),**which is the default.** `DESC` : Sorts in descending order (from highest to lowest)

LIMIT clause Syntax

`SELECT column1, column2, ... FROM table_name LIMIT number_of_rows;`

ORDER BY and LIMIT Clauses syntax `SELECT column1, column2, ... FROM table_name ORDER BY column1 [ASC|DESC], column2 [ASC|DESC], ... LIMIT number_of_rows;`

Quiz: find the number, name, city, and credit limit for all customers located in Boston or Madrid with a credit limit above 50,000.00, sorting by credit limit and showing only the top 15 results

In [20]:

```
quiz =  """
SELECT customerNumber, customerName, city, creditLimit
FROM customers
WHERE (city = 'Boston' OR city = 'Madrid') AND (CreditLimit > 50000)
ORDER BY CreditLimit DESC
LIMIT 15
;
"""
df = pd.read_sql(quiz, conn)
df
```

Out[20]:

	customerNumber	customerName	city	creditLimit
0	495	Diecast Collectables	Boston	85100.00
1	344	CAF Imports	Madrid	59600.00
2	362	Gifts4AllAges.com	Boston	41900.00
3	141	Euro+ Shopping Channel	Madrid	227600.00
4	458	Corrida Auto Replicas, Ltd	Madrid	104600.00
5	237	ANG Resellers	Madrid	0.00
6	465	Anton Designs, Ltd.	Madrid	0.00

The output of this query doesn't seem to respect our credit limit criterion. There are results here where the credit limit is not over 50,000.00. A little investigation shows that this is because the number is actually stored as a string!

In [21]:

```
df['creditLimit'].iloc[2]
```

```
Out[21]: '41900.00'
```

```
In [22]: print(df["creditLimit"].dtype)
```

```
object
```

PRAGMA

One additional technique we can use to understand the schema of a SQLITE table is the PRAGMA table_info command. SQLITE PRAGMA

- `PRAGMA` commands allow you to configure database-level settings and retrieve metadata.
- Unlike standard SQL, `PRAGMA` is specific to SQLite and provides useful functions for database optimization, schema exploration, and configuration.
- Most `PRAGMA` commands do not alter the database schema or data directly but control how the database engine operates.

syntax

`PRAGMA table_info(table_name);`

```
In [23]: # Import data from an SQL query and load it as a DataFrame
         df = pd.read_sql(
             """PRAGMA table_info(customers)""",  # SQL command to retrieve schema details f
             conn,                                 # Database connection object to execute th
             index_col="cid"                       # Set the 'cid' column as the index of the
         )

         # Display the DataFrame to view the schema details of 'customers'
         df
```

Out[23]:

	name	type	notnull	dflt_value	pk
cid					
0	customerNumber		0	None	0
1	customerName		0	None	0
2	contactLastName		0	None	0
3	contactFirstName		0	None	0
4	phone		0	None	0
5	addressLine1		0	None	0
6	addressLine2		0	None	0
7	city		0	None	0
8	state		0	None	0
9	postalCode		0	None	0
10	country		0	None	0
11	salesRepEmployeeNumber		0	None	0
12	creditLimit		0	None	0
13	creditLimitNumeric	REAL	0	None	0

none of the columns actually have a data type specified (the `type` column is empty) and none of the columns is marked as the primary key (`pk` column). SQLite is defaulting to treating them like strings — even creditLimit, which we clearly want to treat as a number — because the schema doesn't specify their types.

Database Administration

In this case, you control the database since it's just a file on your computer. You can perform some database administration and create a properly-typed copy of creditLimit, called creditLimitNumeric, so the complex query above will work.

Since all our queries so far have been SELECT statements, no changes have been made. It's important to keep track of conn attributes when performing any database administration.

In [24]:
```
print("uncommitted changes:", conn.in_transaction)
print("total changes:", conn.total_changes)
```

```
uncommitted changes: False
total changes: 0
```

write a query that will alter the database structure (adding a new column `creditLimitNumeric`)

In [26]:
```
# adding a new column creditLimitNumeric
add_column ="""
ALTER TABLE customers
ADD COLUMN creditLimitNum REAL;
"""
cur.execute(add_column)
```

Out[26]:
```
<sqlite3.Cursor at 0x15a634d48f0>
```

In [27]:
```
# copy all of the creditLimit values to the new creditLimitNumeric column
fill_values = """
UPDATE customers
SET creditLimitNum = creditLimit
;
"""
cur.execute(fill_values)
```

Out[27]:
```
<sqlite3.Cursor at 0x15a634d48f0>
```

In [28]:
```
# check the attribute of conn
print("Uncommitted changes:", conn.in_transaction)
print("Total changes:", conn.total_changes)
```

```
Uncommitted changes: True
Total changes: 122
```

In [29]:
```
# commit changes
conn.commit()
```

In [30]:
```
# check the attribute of conn
print("Uncommitted changes:", conn.in_transaction)
print("Total changes:", conn.total_changes)
```

```
Uncommitted changes: False
Total changes: 122
```

In [31]:
```
# looking at table info again
pd.read_sql("""PRAGMA table_info(customers)""", conn, index_col="cid")
```

Out[31]:

	name	type	notnull	dflt_value	pk
cid					
0	customerNumber		0	None	0
1	customerName		0	None	0
2	contactLastName		0	None	0
3	contactFirstName		0	None	0
4	phone		0	None	0
5	addressLine1		0	None	0
6	addressLine2		0	None	0
7	city		0	None	0
8	state		0	None	0
9	postalCode		0	None	0
10	country		0	None	0
11	salesRepEmployeeNumber		0	None	0
12	creditLimit		0	None	0
13	creditLimitNumeric	REAL	0	None	0
14	creditLimitNum	REAL	0	None	0

In [32]:

```
# trying the quiz again now using creditLimitNumeric
quiz =  """
SELECT customerNumber, customerName, city, creditLimitNum
FROM customers
WHERE (city = 'Boston' OR city = 'Madrid') AND (creditLimitNum > 50000)
ORDER BY CreditLimit DESC
LIMIT 15
;
"""
df = pd.read_sql(quiz, conn)
df
```

Out[32]:

	customerNumber	customerName	city	creditLimitNum
0	495	Diecast Collectables	Boston	85100.0
1	344	CAF Imports	Madrid	59600.0
2	141	Euro+ Shopping Channel	Madrid	227600.0
3	458	Corrida Auto Replicas, Ltd	Madrid	104600.0

In [33]:

```
# closing the file
conn.close()
```

IN SUMMARY

Here's a summary of key concepts about working with SQLite connections and performing common database tasks:

Establishing a Connection:

- Use `sqlite3.connect('database_name.sqlite')` to connect to an SQLite database file. This connection object (`conn`) allows you to execute SQL commands and manage transactions.

Using a Cursor:

- A cursor (`cur = conn.cursor()`) is required for executing SQL commands. It acts as an intermediary for sending queries and fetching results from the database.

Schema Exploration:

- To explore a database structure, use the `PRAGMA table_info(table_name)` command to get details about a specific table's columns, types, and constraints.
- Use Pandas with `pd.read_sql("PRAGMA table_info(table_name)", conn)` to load this schema information into a DataFrame for analysis.

Executing Queries and Data Retrieval

- Use SQL `SELECT` statements to retrieve data, optionally transforming the results into a DataFrame with `pd.read_sql` .

Common query features:

- WHERE filters records based on conditions.
- ORDER BY sorts results.
- LIMIT restricts the number of rows returned.

Altering Table Structure:

- Use ALTER TABLE to modify a table structure, such as adding a new column. For example:

```
ALTER TABLE customers ADD COLUMN creditLimitNumeric REAL;
```

- To populate this new column, you can convert data types with CAST.

Database Administration Note:

Typically, data scientists have read-only access to databases, focusing on SELECT queries. Altering structures is usually reserved for database administrators.