AI Developers

Leveraging SQLite for Intelligent Systems

Jonathan Zouari, Ph.D.

Jonathan.Zouari@mail.huji.ac.il

What is SQL?

The Standard Language

SQL (Structured Query Language) is the universally recognized language for interacting with and managing relational databases. It provides a standardized way to communicate with data.

Data Management

SQL allows you to retrieve, insert, update, and delete data efficiently. It's also used to define the structure of your databases, tables, and relationships.

Ubiquitous & Powerful

Used across virtually all industries, SQL is fundamental for data analysis, backend development, and powering intelligent systems that rely on structured data.

SQL Examples

```
SELECT id, first_name, salary
FROM employees
WHERE salary > 10000;

INSERT INTO employees (first_name, salary)
VALUES ('Max', 9000);

DELETE FROM employees
WHERE id = 2;
```

Database Schema

Understanding the structure of our core data tables is essential for effective SQL querying.

Employees

id	name	salary	dep_id
1	Max	10000	1
2	Julie	15000	1
3	Marc	8000	3

Departments

id	dep_name	location
1	Developers	Munich
2	Sales	Berlin
3	Accounting	Berlin

These two tables, **Employees** and **Departments**, illustrate a common relational database setup, where employees are linked to their respective departments via the `dep_id` column.

Join Tables

Employees

id	name	salary	dep_id
1	Max	10000	1
2	Julie	15000	1
3	Marc	8000	3

Departments

id	dep_name	location
1	Developers	Munich
2	Sales	Berlin
3	Accounting	Berlin

SQL Query Example: Joining Data

```
SELECT name, salary, dep_name, location
FROM employees INNER JOIN departments ON dep_id = departments.id
WHERE salary >= 10000
```

This query demonstrates how SQL can combine data from different tables (Employees and Departments) using an **INNER JOIN**, filtering results based on specific conditions.

Relational Database Management Systems (RDBMS)

The Core Software

RDBMS is the powerful software system responsible for handling the actual data management and storage within a relational database.

SQL as the Language

It uses SQL as the standard language, allowing users to interact with the database for tasks like creating tables and managing data.

Data Interaction

Through SQL, you can easily select, filter, insert, update, and delete data, providing full control over your stored information.

Popular RDBMS

MySQL

A widely used open-source relational database management system.

PostgreSQL

A powerful, open-source object-relational database system known for its advanced features.

Oracle Database

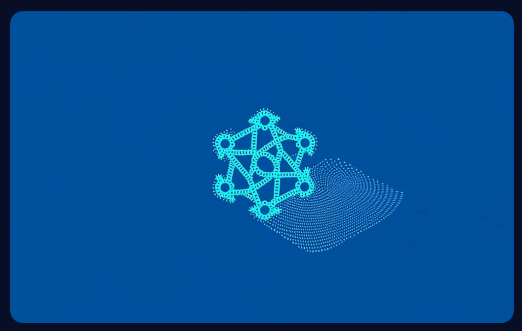
A comprehensive and robust commercial relational database system for enterprise-level applications.

Microsoft SQL Server

Microsoft's powerful RDBMS designed for large-scale data management and business intelligence.

Microsoft Access

A desktop database system, ideal for personal use and small business applications.



SQLite

A self-contained, serverless, zero-configuration, transactional SQL database engine, often embedded in applications.

...and many more database systems are available, catering to diverse needs and scales.

S.Q.L or Sequel?

Both terms are commonly used to refer to the Structured Query Language. Here's a breakdown of their origins:

S.Q.L. (Structured Query Language)

This is the original and more "official" acronym. It directly represents the full name of the language and how it's typically written.

Sequel

This pronunciation was also considered for the main language name during its development. Many find that "Sequel" flows more naturally and is easier to say.

Why Would You Use A Database (System)

Transactional Data / Application Data

This data is actively generated by users or applications and is essential for the real-time functioning of websites and mobile apps.

- Products
- Orders
- User accounts
- Blog posts
- Comments
- Likes

Almost all online platforms rely on databases to manage this kind of dynamic, frequently updated content.

Data Analytics / Report Data

This information is collected primarily for analysis, reporting, and deriving insights to inform business decisions and track performance.

- Website log data
- Sales data
- Weather data
- Energy consumption data

Such data often involves large volumes and is used for historical trends, forecasting, and business intelligence.

Databases vs Tables vs Data

Database

A collection of related tables designed to store specific sets of structured data.

Table

An organized collection of data arranged into rows and columns, such as `id`, `first_name`, and `last_name`.

Data

The actual values stored within the cells of tables, for instance, "Max", "Schwarz", "Analytics", or "Munich".

Understanding these fundamental components is key to grasping how relational databases are structured and managed.

Core SQL Syntax Rules

SQL statements follow a structured syntax with specific components:

SQL Command / Statement

The overall instruction that defines the action to be performed (e.g., `SELECT`, `INSERT`, `UPDATE`, `DELETE`).

Key Words

Reserved words like `FROM`, `WHERE`, `GROUP BY` that have special meaning and define operations within the statement.

Identifiers

Names given to database objects such as tables, columns, views, or stored procedures.

Operators

Symbols used to perform comparisons (`=`, `>`, `<`), logical operations (`AND`, `OR`), or arithmetic calculations (`+`, `-`).

Clauses

Sections of the SQL statement that serve specific purposes, often introduced by keywords (e.g., `WHERE` clause, `ORDER BY` clause).

Example SQL Statement:

```
SELECT name, salary FROM employees WHERE salary > 8000;
```

All SQL statements typically end with a semicolon (`;`) to indicate the end of the command.

SQL in Action: Retrieving Data

The `SELECT` statement is fundamental in SQL for retrieving data from one or more tables. It allows you to specify which columns you want to see and from which tables.

Example: Selecting Customer Emails

```
SELECT email FROM customers;
```

Query Results:

max@test.com
manu@test.com
juli@test.com
ken@test.com
white@test.com

This simple query fetches all email addresses from the `customers` table, demonstrating the basic functionality of data retrieval.

Core SQL Syntax Rules: Deeper Dive

Understanding these fundamental rules is crucial for writing effective and error-free SQL queries.

Rule 1: Semicolons for Multiple Statements

SQL statements are typically terminated with a semicolon (;). This is especially important when you have more than one SQL statement in a single command, allowing the database system to execute them separately.

```
SELECT email FROM customers;
SELECT first_name FROM customers;
```

Rule 2: Single Statements

While good practice, a semicolon is not strictly required for a single SQL statement in a command. The database often understands the end of the command without it.

```
SELECT name FROM employees
```

Rule 3: Case Insensitivity

SQL is generally case-insensitive regarding keywords and commands. For example, `SELECT`, `select`, and `SeLeCt` are all treated the same by the database.

```
select * FROM Products WHERE Price > 10;
```

Rule 4: Identifiers with Quotes

Database object names (like table or column names) can be enclosed in double quotes (e.g., `"TableName"`) or backticks (in MySQL) to avoid conflicts with SQL keywords, especially if your identifier name is also an SQL keyword.

```
SELECT "select" FROM customers; -- Using "select" as a column name
SELECT `order` FROM Sales;    -- Using `order` as a column name in MySQL
```

Rule 5: Clause Order Matters

SQL statements are composed of clauses, each serving a specific purpose. These clauses must appear in a predefined order within a query. For instance, `SELECT` must always come before `FROM`, and `WHERE` typically follows `FROM`.

```
SELECT column1, column2
FROM your_table
WHERE condition
ORDER BY column1;
```

SQLite: A Lightweight Relational Database

SQLite is a lightweight, serverless database system. Unlike others, it doesn't need a separate server process, making it versatile and efficient for various applications.

Key Features that Define SQLite

Serverless Design

Database stored in a single file; no server to manage.

Zero Configuration

No setup or administration needed; easy to deploy.

Python Integration

Built-in support via Python's standard `sqlite3` module.

Security & Reliability

ACID-compliant, ensuring data integrity and durability.

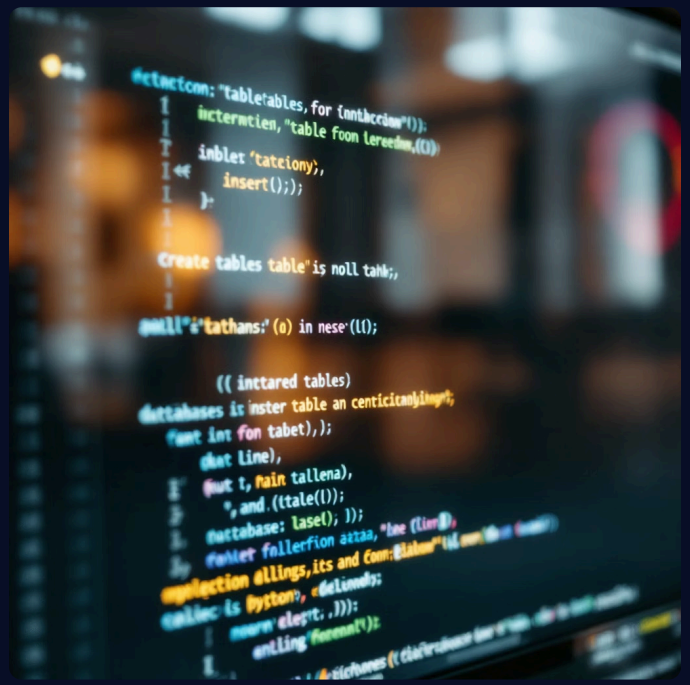
SQLite's simplicity and built-in SQL support make it an ideal entry point for learning about databases, avoiding complex server management.

Ideal Use Cases:

- Small to medium-sized applications
- Rapid prototyping and development
- Educational purposes
- Embedded databases in mobile/desktop software

SQL in Action: sqltestcode.py

The sqltestcode.py script demonstrates practical SQL operations–creating tables, inserting, and querying data–within a Python environment using the sqlite3 module. It bridges SQL syntax with its application in Python.



Step 1: How to Install and Import SQLite in Python

Getting started with SQLite in Python is straightforward, thanks to its inclusion in the standard library.

Zero Installation

`sqlite3` is built into Python, requiring no installation.

Simple Import

Import with `import sqlite3` at the top of your script.

Verify Success

Verify import success with a simple test command.

Ready to Use

Ready to create connections and execute SQL commands.

Code Example:

```
import sqlite3
print("SQLite3 module imported successfully!")
```

The `sqlite3` module integrates easily into any Python environment.

Step 2: Creating a SQLite Database Connection in Python

To perform any database operation in Python, you must first establish a connection to your SQLite database.

Create Connection

Use `sqlite3.connect('database_name.db')` to establish a link. A new file is automatically created if it doesn't exist.

Initialize Cursor

Obtain a cursor object with `connection.cursor()`. This acts as an intermediary to execute SQL commands.

Code Example:

```
import sqlite3
conn = sqlite3.connect('example.db')
cursor = conn.cursor()
# Now you can execute SQL commands like:
```

SQLite3 Data Types

Understanding how SQLite handles data types is crucial for effective database interaction, especially when working with Python. SQLite uses a flexible, dynamic typing system.

Data Type Mapping Table

SQLite Type	Python Type	Description
NULL	None	Represents missing data or unknown values
INTEGER	int	Whole numbers from -9223372036854775808 to 9223372036854775807
REAL	float	Floating point values with decimal precision
TEXT	str	Character data of any length (UTF-8, UTF-16BE or UTF-16LE)
BLOB	bytes	Binary Large Object, stores data exactly as it was input (e.g., images, files)

Step 3: How to Create a Table

Tables organize data into rows and columns, forming the structure for your SQLite database.

Define Table Structure

Determine column names and appropriate data types (e.g., INTEGER, TEXT).

Craft SQL Statement

Write your SQL `CREATE TABLE` statement, including table name, columns, and optional constraints like `PRIMARY KEY`.

Execute and Commit

Execute the statement using your Python `cursor` and call `conn.commit()` to save changes permanently.

Code Example:

```
# Creating a users table
cursor.execute('''
CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    age INTEGER
)
''')
conn.commit()
print("Table 'users' created successfully")
```

Key Point:

`PRIMARY KEY` ensures unique records, and `AUTOINCREMENT` automatically assigns sequential IDs, simplifying data management.

Breaking Down Table Creation

Execute Command

Run `CREATE TABLE` statements using `cursor.execute()`. Use triple quotes for multi-line.

Handle Duplicates

`IF NOT EXISTS` prevents errors if the table already exists.

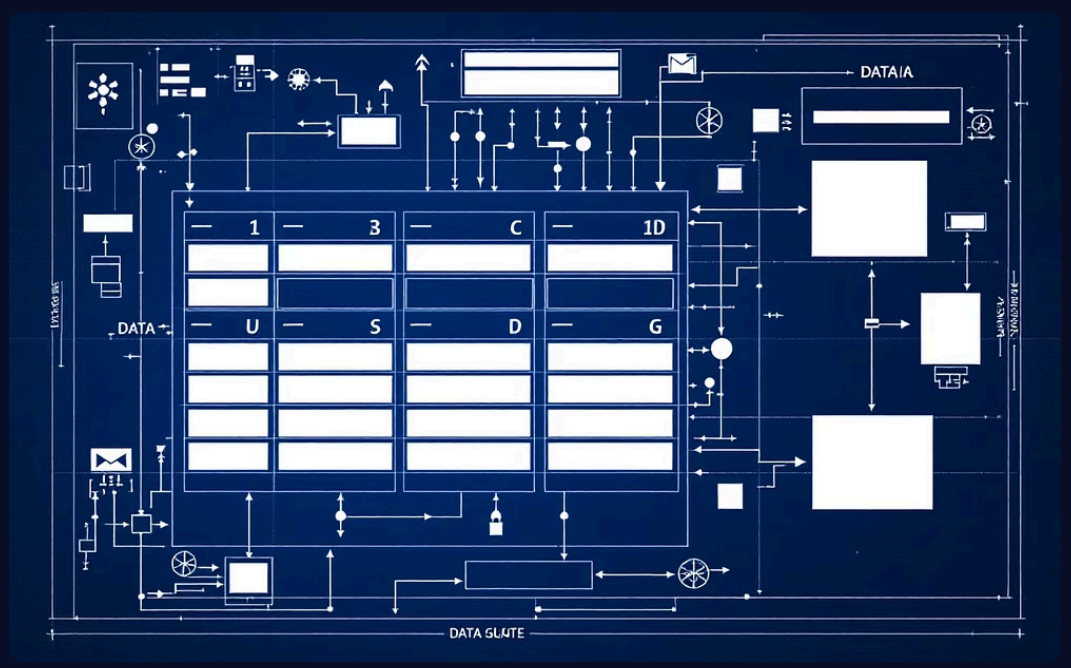
Save Permanently

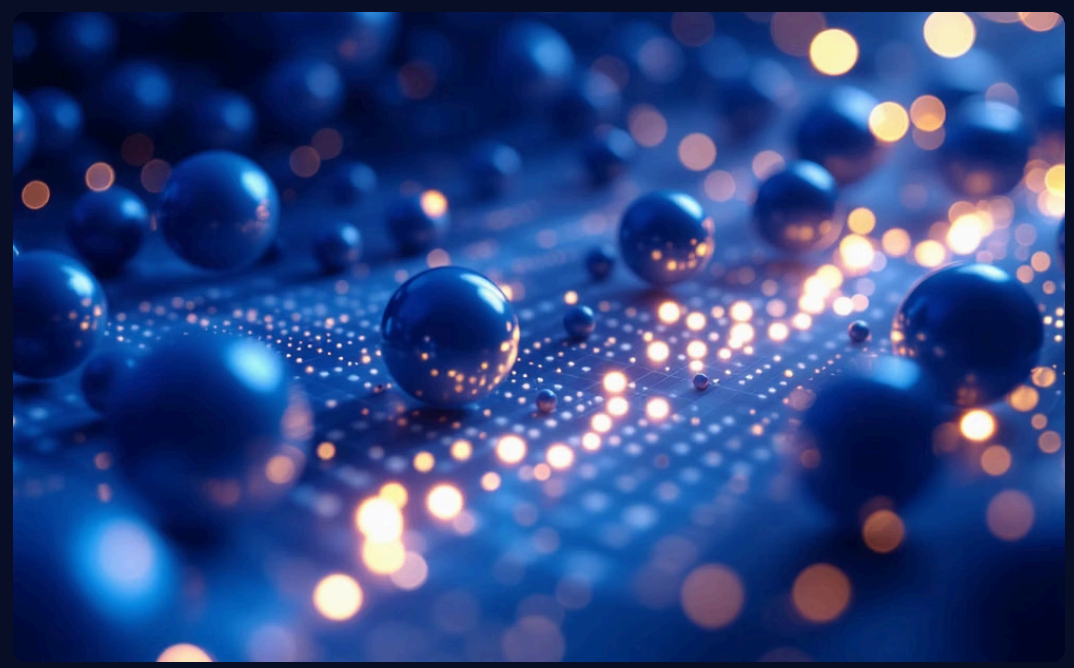
Call `conn.commit()` to save the new table structure to your database file.

Important Note: Until you call `conn.commit()`, any changes, including table creations, exist only in memory. The `commit()` action writes them permanently to your SQLite database file.

Step 4: How to Insert Data into the Table

After creating your table structure, you need to populate it with data. The SQL `INSERT` statement adds new records to your tables.





Prepare INSERT Statement

Specify the table name and columns you want to populate.

Provide VALUES

Supply data values in the correct order for each column.

Execute With Cursor

Run your statement through the cursor object.

Commit Changes

Save the inserted records permanently to the database.

Code Example:

```
# Insert data into the users table
cursor.execute("INSERT INTO users (name, age) VALUES ('Alice', 25)")
cursor.execute("INSERT INTO users (name, age) VALUES ('Bob', 30)")

# Save (commit) the changes
conn.commit()
print("Inserted 2 rows into the 'users' table")
```

Step 5: How to Retrieve (SELECT) Data from the Table

Once data is in your SQLite table, retrieving it is done using the SQL `SELECT` statement, executed through your Python cursor.

Construct Your SELECT Query

Use `SELECT * FROM tablename` to retrieve all columns, or specify individual columns for targeted data retrieval.

Execute the Query

Run `cursor.execute("SELECT * FROM users")` to send your SQL query to the database.

Fetch Results

Use `cursor.fetchall()` to get all resulting rows as a list of tuples, or `fetchone()` for a single row.

Process the Retrieved Data

Iterate through the returned rows to display, filter, or further process each record in your Python application.

Code Example:

```
import sqlite3

# Connect to the database
conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Select all data from the users table
cursor.execute("SELECT * FROM users")

# Fetch all the results
rows = cursor.fetchall()

# Process and print the results
print("Data in 'users' table:")
for row in rows:
    print(row)

# Close the connection
conn.close()
```

The `SELECT` statement is fundamental for extracting information, allowing you to access and display your stored data effectively.

Step 6: How to Update Data

Modifying existing records in your SQLite database is done using the SQL `UPDATE` statement.

Define UPDATE Statement

Use `UPDATE tablename SET column=value WHERE condition` to modify existing records.

Specify WHERE Clause

The `WHERE` clause is crucial for filtering which rows to update. Without it, `all` records in the table will be changed!

Execute with Cursor

Run your `UPDATE` query using `cursor.execute()` to perform the changes within the database.

Commit Changes

Call `connection.commit()` to permanently save your modifications to the SQLite database file.

Code Example:

```
# Update Alice's age to 26
cursor.execute("UPDATE users SET age = 26 WHERE name = 'Alice'")
conn.commit()
print("Updated Alice's age to 26")

# Verify the update by selecting Alice's record again
cursor.execute("SELECT * FROM users WHERE name = 'Alice'")
print(cursor.fetchall())
```

Output: `[(1, 'Alice', 26)]`

Step 7: How to Delete Data

Deleting records permanently removes them from your database. Always use a `WHERE` clause to target specific records.

Define DELETE Statement

Use `DELETE FROM tablename WHERE condition` syntax to precisely target and remove specific records.

Specify WHERE Clause

The `WHERE` clause is crucial. Without it, all records in the table will be permanently removed!

Execute & Commit

Run the `DELETE` statement with `cursor.execute()` and save changes permanently with `conn.commit()`.

Critical Warning: Always double-check your `WHERE` clause before executing a `DELETE` statement. Data deleted without a specific condition cannot be easily recovered.

Code Example:

```
# Delete the record for Bob
cursor.execute("DELETE FROM users WHERE name = 'Bob'")
conn.commit()
print("Deleted Bob's record")

# Verify deletion by selecting all remaining records
cursor.execute("SELECT * FROM users")
print(cursor.fetchall())
```

Output: `[(1, 'Alice', 26)]`

Step 8: Data Input with Placeholders

Using placeholders in SQL queries prevents dangerous SQL injection attacks and ensures data integrity.

Use ? Placeholders

Replace dynamic values in SQL with question marks (?).

Pass Values as Tuple

Supply data as a tuple to cursor.execute(), matching placeholder order.

Code Example:

```
# New user data, perhaps coming from user input
new_name = "Charlie"
new_age = 35

# Use placeholders (?) in the SQL, and pass the values as a tuple
cursor.execute("INSERT INTO users (name, age) VALUES (?, ?)", (new_name, new_age))
conn.commit()
print(f"Inserted new user: {new_name}, age {new_age}")

# Check that the new record was inserted
cursor.execute("SELECT * FROM users")
print(cursor.fetchall())
```

Output: [(1, 'Alice', 26), (2, 'Charlie', 35)]

How to Add Multiple Records to a Table

Efficiently adding multiple records to a database is crucial. Python's `sqlite3` module offers `executemany()` for performance.

Method 1: Individual `INSERT` Statements

Each record is inserted with a separate `INSERT` statement. Simple for small datasets, but less efficient for large ones due to repeated database operations.

```
cursor.execute("INSERT INTO users (name, age) VALUES ('Alice', 25)")
cursor.execute("INSERT INTO users (name, age) VALUES ('Bob', 30)")
cursor.execute("INSERT INTO users (name, age) VALUES ('Charlie', 35)")
conn.commit()
```

Method 2: Using `executemany()`

`executemany()` inserts multiple rows at once, using a single SQL statement with placeholders and a list of tuples. This is the most efficient method for bulk insertions.

```
users_data = [
    ('Alice', 25),
    ('Bob', 30),
    ('Charlie', 35),
    ('Diana', 28)
]
cursor.executemany("INSERT INTO users (name, age) VALUES (?, ?)", users_data)
conn.commit()
print(f"Inserted {len(users_data)} records")
```

Step 9: How to Close the Connection

Properly closing your database connection is a crucial final step to ensure data integrity and efficient resource management.

Call the close() Method

Use conn.close() to terminate the database connection when your operations are complete.

Verify Data Persistence

All committed changes remain saved in the database file after closing, ensuring your work isn't lost.

Code Example:

```
# Close the database connection when done
conn.close()
print("Database connection closed")
```

Important Note: Once you call conn.close(), the connection object and its cursor will no longer be usable. If you need to work with the database again, you must reconnect using sqlite3.connect().

Step 10: Using Python's `with` Statement for Better Resource Management

Python's `with` statement provides elegant resource management through context managers. It automatically handles setup and cleanup, regardless of how execution flows.

Enter Context

Resources are acquired and initialized when entering the `with` block.

Execute Code

Your operations run safely within the managed context.

Auto-Cleanup

Resources are properly released when exiting, even if errors occur.

Code Example:

```
# SQLite with context manager - no need to call close()
with sqlite3.connect('example.db') as conn:
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM users")
    print(cursor.fetchall())
    # Connection automatically closed after this block
```

Key Benefit:

Using `with` with SQLite ensures your database connections always close properly, preventing resource leaks even when exceptions occur.

SQLite Basic Exercise: Library Management

This exercise provides a hands-on opportunity to practice fundamental SQLite database operations within a Python environment, simulating a simple library management system.

Exercise Instructions

Setup & Connection

Import `sqlite3`, connect to `library.db`, and create a cursor.

Create Table

Define a `books` table with `id`, `title`, `author`, and `year_published` columns.

Insert Data

Add four specified book records into your new `books` table.

Select All

Retrieve and display all records from the `books` table.

Update Data

Modify the `year_published` for "1984" to 1948.

Delete Data

Remove the record for "Pride and Prejudice".

Final Select

Display all books remaining in the table after deletions and updates.

Close Connection

Properly close the database connection to ensure data integrity.

Your solution should utilize proper SQL syntax, include comments for each step, commit changes after modifications, and clearly display results.

Example

Here are the table schemas for Cities, Addresses, and Users, based on the provided design:

Cities Table

Column Name	Properties
ID	INTEGER PRIMARY KEY
Name	TEXT NOT NULL UNIQUE

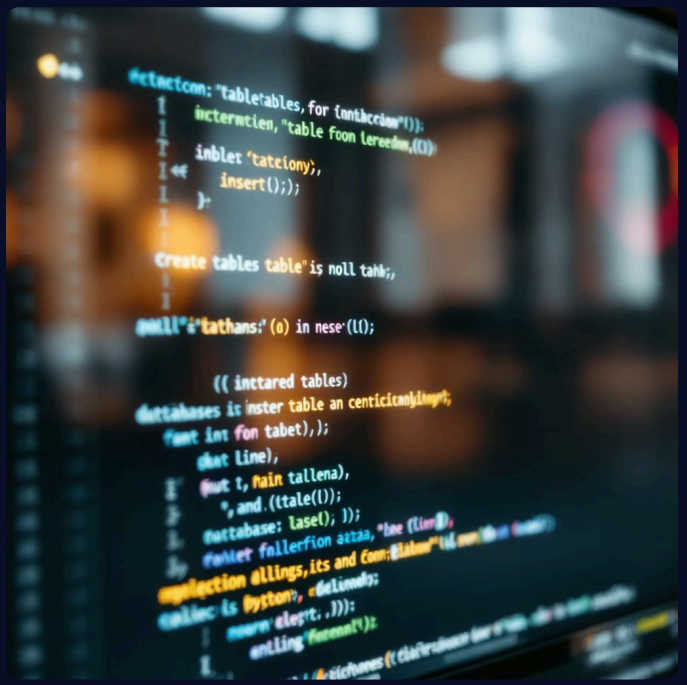
Addresses Table

Column Name	Properties
ID	INTEGER PRIMARY KEY
Street	TEXT NOT NULL
House_Number	TEXT
City_ID	INTEGER, FOREIGN KEY REFERENCES Cities(ID)

Users Table

Column Name	Properties
ID	INTEGER PRIMARY KEY
First_Name	TEXT NOT NULL
Last_Name	TEXT NOT NULL
Email	TEXT UNIQUE
Address_ID	INTEGER, FOREIGN KEY REFERENCES Addresses(ID)

SQL in Action: sqltestcode2.py



What are 'CRUD' Operations?

Basic Data Manipulation Operations

C	R	U	D
Create Data	Read Data	Update Data	Delete Data
INSERT INTO	SELECT	UPDATE	DELETE

Read Data – Basic Data Fetching (SELECT)

The `SELECT` statement retrieves data from a database, based on your specified criteria.

Basic Selection

```
SELECT <columns> FROM <table name>;
```

Fetches specified columns from a table. Use `*` for all columns, or list specific names.

Example: `SELECT name, age FROM Users;`

Conditional Selection

```
SELECT <columns> FROM <table name> WHERE <condition>;
```

Filters records, returning only those that meet the `WHERE` clause condition.

Example: `SELECT name FROM Products WHERE price > 100;`

The `WHERE` Clause

The `WHERE` clause filters records based on specified conditions.

Comparison Operators

- `=`: Equal to
- `<>` or `!=`: Not equal to
- `>`: Greater than
- `>=`: Greater than or equal to
- `<`: Less than
- `<=`: Less than or equal to

Logical Operators

- `AND`: All conditions must be true.
- `OR`: At least one condition must be true.

Special Operators

- `IS NULL`: Tests for NULL values.
- `BETWEEN`: Checks if a value is within a range (inclusive).
- `LIKE`: Searches for a specified pattern.
- `IN`: Checks for values in a list.

Important Notes:

- Always use `IS NULL` or `IS NOT NULL` for checking against NULL.
- `BETWEEN` is inclusive of both start and end values.

Ordering Data: The `ORDER BY` Clause

The `ORDER BY` clause is used to sort the result-set of a query based on one or more columns.

Ascending Order (`ASC`)

```
SELECT * FROM <table>
ORDER BY <column name>;
```

Sorts the selected data in ascending order (default behavior).

Descending Order (`DESC`)

```
SELECT * FROM <table>
ORDER BY <column name> DESC;
```

Sorts the selected data in descending order.

Ordering by Multiple Columns

```
SELECT * FROM Customers
ORDER BY Country ASC, CustomerName DESC;
```

You can specify multiple columns to sort by. The results are first sorted by the first column, then by the second within identical values of the first, and so on.

Eliminating Duplicates: The `DISTINCT` Keyword

The `DISTINCT` keyword ensures that only unique values or combinations of values are returned in your SQL query result set.

Syntax & Function

```
SELECT DISTINCT <column1>, <column2>, ...
FROM <table>;
```

Applied to selected columns, `DISTINCT` ensures each **combination** of values across those columns appears only once.

Practical Example

```
-- Imagine a 'Customers' table with columns: CustomerID, Name, Country

-- To get a list of all unique countries where customers reside:
SELECT DISTINCT Country
FROM Customers;

-- To get unique combinations of Name and Country:
SELECT DISTINCT Name, Country
FROM Customers;
```

For `SELECT DISTINCT Country`, only unique country names are returned. For `SELECT DISTINCT Name, Country`, uniqueness is determined by the specific pair of Name and Country.

> **Note:** `DISTINCT` applies to all columns selected; uniqueness is based on their combination.

Relationships: JOINing Data

Joining Data & Introducing INNER JOIN

The INNER JOIN clause combines rows from two or more tables, returning only records with matching values in specified columns of both.

addresses Table

1	Teststreet	10A
2	Some Street	5
3	My Street	18

users Table

1	Max	1
2	Manuel	3

SQL Query Example

```
SELECT u.first_name, a.street, a.house_number
FROM users AS u
INNER JOIN addresses AS a ON u.address_id = a.id;
```

This query joins the users and addresses tables on u.address_id = a.id, linking users to their addresses. The result displays each user's first name, street, and house number.

Understanding SQL Aliases

Aliases are temporary, alternative names given to tables or columns in SQL queries to make them shorter and more readable. They are defined using the `AS` keyword.

How They Work

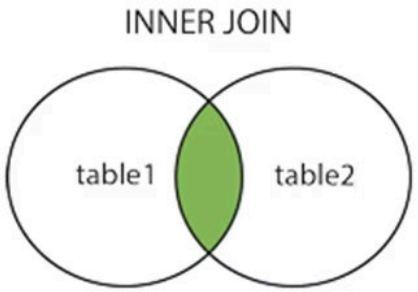
In our previous `INNER JOIN` example, `users AS u` and `addresses AS a` assign aliases "u" and "a" respectively. These aliases are then used with **dot notation** (e.g., `u.first_name`, `a.street`, `u.address_id`) to refer to specific columns from their respective tables. This explicitly clarifies which column belongs to which table.

Why Use Aliases?

Less Typing	Clarity	Readability
Reduces query length.	Identifies column origin.	Simplifies complex queries.

INNER JOIN Syntax

```
SELECT column_name(s)
FROM table1
INNER JOIN table2
ON table1.column_name = table2.column_name;
```

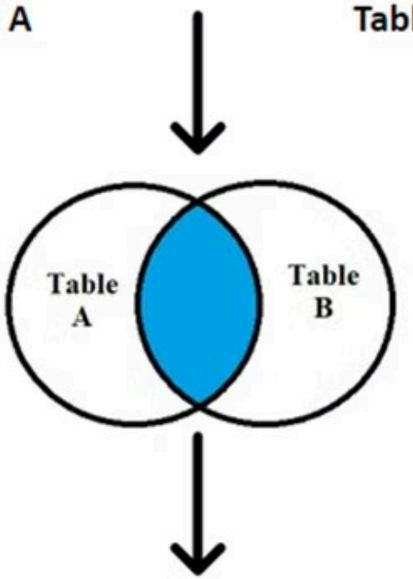


Student ID	Name
1001	A
1002	B
1003	C
1004	D

Table A

Student ID	Department
1004	Mathematics
1005	Mathematics
1006	History
1007	Physics
1008	Computer Science

Table B



Student ID	Name	Department
1004	D	Mathematics

Order of Execution

Example

```
cursor.execute("""
SELECT u.id, u.first_name, u.last_name, a.street, a.house_number, a.city_id
FROM users AS u
INNER JOIN addresses AS a ON u.address_id = a.id;
""")
```

	id	first_name	last_name	street	house_number	city_id
0	1	Max	Schwarz	Some street	10	1
1	2	Manuel	Lorenz	My street	101	2
2	3	Julie	Barnes	Teststreet	1	3

Understanding LEFT JOIN

The LEFT JOIN (or LEFT OUTER JOIN) returns all rows from the left table, and the matching rows from the right table. If there is no match, the columns from the right table will have NULL values.

students Table

1	Alice	20
2	Bob	22
3	Charlie	19
4	Diana	21
5	Eve	20

enrollments Table

101	1	Mathematics	A
102	1	Physics	B+
103	2	Chemistry	A-
104	4	Biology	B

Query: Retrieve all students and their enrolled courses, if any.

```
SELECT s.student_id, s.name, s.age, e.course_name, e.grade
FROM students AS s
LEFT JOIN enrollments AS e ON s.student_id = e.student_id
ORDER BY s.student_id;
```

Result of LEFT JOIN

1	Alice	20	Mathematics	A
1	Alice	20	Physics	B+
2	Bob	22	Chemistry	A-
3	Charlie	19	NULL	NULL
4	Diana	21	Biology	B
5	Eve	20	NULL	NULL

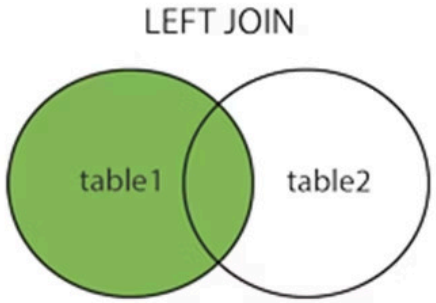
Notice that **Charlie** and **Eve** appear in the result set even though they have no matching entries in the enrollments table. Their respective course_name and grade columns are filled with NULL values.

Left Join

LEFT JOIN Syntax

```
SELECT column_name(s)
FROM table1
LEFT JOIN table2
ON table1.column_name = table2.column_name;
```

Note: In some databases LEFT JOIN is called LEFT OUTER JOIN.



Aggregate Functions

Aggregate functions perform a calculation on a set of rows and return a single summary value. They are often used with the `GROUP BY` clause to summarize data by groups.

COUNT()

Returns the number of rows that match a specified criterion or the total number of rows in a table.

SUM()

Calculates the total sum of a numeric column's values.

AVG()

Computes the average (mean) value of a numeric column.

MIN()

Finds the smallest value in a selected column.

MAX()

Finds the largest value in a selected column.

SQL Aggregate Functions in Action

Let's explore practical applications of SQL aggregate functions through specific examples, demonstrating how they summarize and analyze data effectively.

COUNT() Examples

```
-- 1. Count all customers
SELECT COUNT(*) FROM customers;
```

This query returns the total number of records (rows) in the `customers` table.

```
-- 2. Count orders from a specific date
SELECT COUNT(*) FROM orders WHERE order_date >= '2024-01-01';
```

This counts the number of orders placed on or after January 1, 2024.

SUM() Examples

```
-- 1. Sum of all sales amounts
SELECT SUM(total_price) FROM sales;
```

Calculates the grand total of the `total_price` column across all sales records.

```
-- 2. Sum of donations for a specific organization
SELECT SUM(amount) FROM donations WHERE org_id = 3;
```

Computes the total monetary amount donated to the organization with `org_id` 3.

MIN() and MAX() Examples

```
-- MIN: Find lowest salary in company
SELECT MIN(salary) FROM employees;

-- MIN: Find lowest price in specific category
SELECT MIN(price) FROM products WHERE category = 'Books';
```

`MIN()` retrieves the smallest value in a numeric column, such as the lowest salary among all employees or the cheapest book.

```
-- MAX: Find highest salary in company
SELECT MAX(salary) FROM employees;

-- MAX: Find highest score in specific subject
SELECT MAX(score) FROM exams WHERE subject = 'Math';
```

Conversely, `MAX()` finds the largest value, like the highest salary or the top score in a Math exam.

AVG() Example

```
-- Calculate average salary
SELECT AVG(salary) FROM employees;
```

This function calculates the average numerical value of the `salary` column for all employees, providing insight into the typical salary in the company.

Grouping Data: The `GROUP BY` Clause

The `GROUP BY` clause is used in SQL to arrange identical data into groups. It aggregates rows that have the same values into summary rows, often with aggregate functions, allowing you to perform calculations on each group.

Simple Example: Employee Count by Department

```
SELECT department, COUNT(*) AS employee_count
FROM employees
GROUP BY department;
```

Result:

IT	5
Sales	3
HR	2
Marketing	4

Another Example: Total Sales by Product Category

```
SELECT category, SUM(price) AS total_sales
FROM products
GROUP BY category;
```

Result:

Electronics	15000
Books	3500
Clothing	8200
Sports	6800

Key Principles of `GROUP BY`

- **Grouping Rows:** `GROUP BY` groups rows that have the same values in specified columns into a summary row.
- **Aggregate Functions:** It must be used in conjunction with aggregate functions (e.g., `COUNT`, `SUM`, `AVG`, `MIN`, `MAX`) to perform calculations on each group.
- **Column Inclusion:** All non-aggregate columns included in the `SELECT` statement must also appear in the `GROUP BY` clause.
- **Result Structure:** The query result will show one row per group, displaying the grouped column(s) and their calculated aggregate values.

Filtering Groups with `HAVING`

The `HAVING` clause is used in SQL to filter groups created by the `GROUP BY` clause, acting similarly to how `WHERE` filters individual rows.

Basic Example: Employee Count by Department

Let's find departments that have more than 2 employees.

```
SELECT department, COUNT(*) AS employee_count
FROM employees
GROUP BY department
HAVING COUNT(*) > 2;
```

Result:

IT	5
Marketing	4
Sales	3

Another Example: Departments with High Average Salary

Identify departments where the average salary is greater than 70,000.

```
SELECT department, AVG(salary) AS avg_salary
FROM employees
GROUP BY department
HAVING AVG(salary) > 70000;
```

Result:

IT	75000
Sales	72500

Key Differences: `WHERE` vs. `HAVING`

`WHERE` Clause

- Filters individual rows.
- Applied before `GROUP BY`.
- Cannot use aggregate functions.

`HAVING` Clause

- Filters groups of rows.
- Applied after `GROUP BY`.
- Can use aggregate functions.

Example Combining `WHERE` and `HAVING`

Let's filter products with a price over $100, then group them by category, and finally show only those categories with 3 or more such products.

```
SELECT category, COUNT(*) AS product_count
FROM products
WHERE price > 100
GROUP BY category
HAVING COUNT(*) >= 3;
```

This query first filters products where `price > 100` (`WHERE`), then groups them by category, and finally shows only categories with 3 or more products in that filtered set (`HAVING`).

Comprehensive Example: `GROUP BY` & `HAVING`

Let's combine our understanding of `GROUP BY` and `HAVING` with a practical example that involves joining multiple tables.

Query:

```
SELECT Employees.LastName, COUNT(Orders.OrderID) AS NumberOfOrders
FROM Orders
INNER JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID
GROUP BY LastName
HAVING COUNT(Orders.OrderID) > 10;
```

Result:

Buchanan	11
Callahan	27
Davolio	29
Fuller	20
King	14
Leverling	31
Peacock	40
Suyama	18