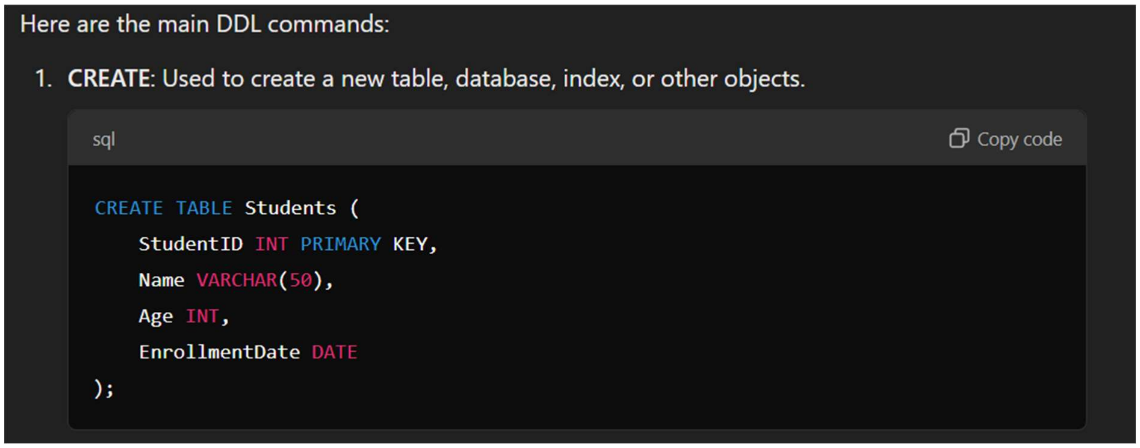
SQL Data Definition Language (DDL)

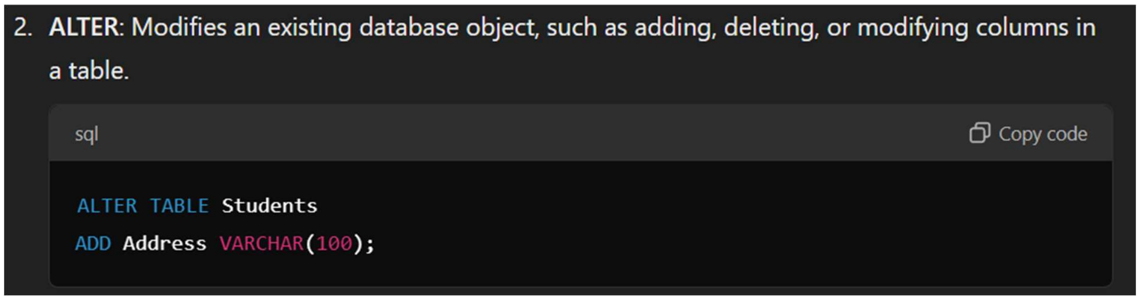
SQL's Data Definition Language (DDL) is *used **to define and manage database structures like tables, schemas, and relationships***. It consists of commands that enable you to create, alter, delete, and manage the structure of database objects.



Here are the main DDL commands:

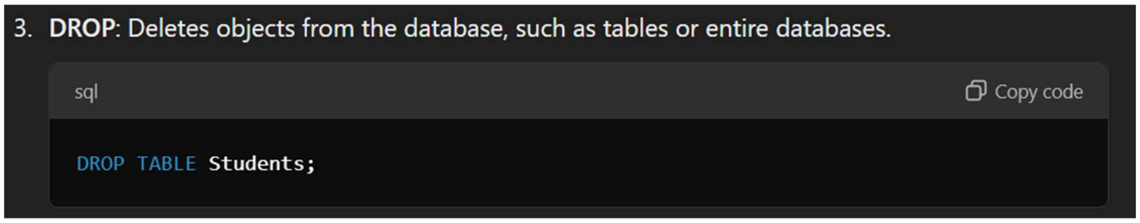
1. **CREATE**: Used to create a new table, database, index, or other objects.

```sql
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    Name VARCHAR(50),
    Age INT,
    EnrollmentDate DATE
);
```



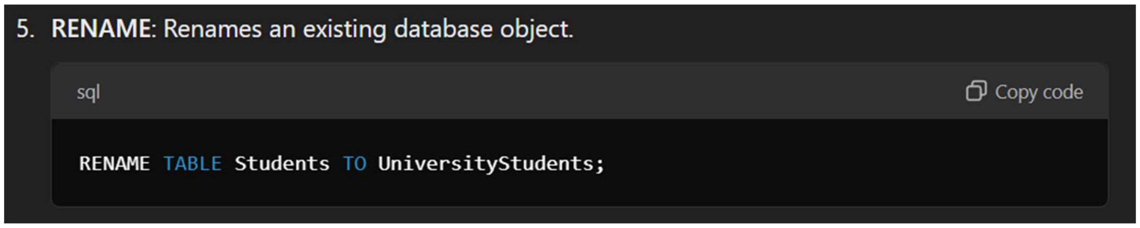
2. **ALTER**: Modifies an existing database object, such as adding, deleting, or modifying columns in a table.

```sql
ALTER TABLE Students
ADD Address VARCHAR(100);
```



3. **DROP**: Deletes objects from the database, such as tables or entire databases.

```sql
DROP TABLE Students;
```



5. **RENAME**: Renames an existing database object.

```sql
RENAME TABLE Students TO UniversityStudents;
```

SQL Data Types

Data types specify the kind of data a column can hold. Here are the main categories of data types in SQL:

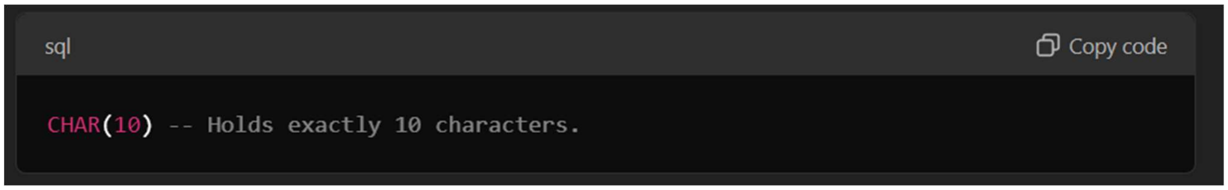
1. Numeric Data Types

- **INT**: Integer values, e.g., 5, 10, -20.
- **DECIMAL(p, s)**: Fixed-point decimal, where p is the precision, and s is the scale.

- **FLOAT**: Approximate floating-point values; can be less precise than DECIMAL.
- **DOUBLE**: Double-precision floating-point.

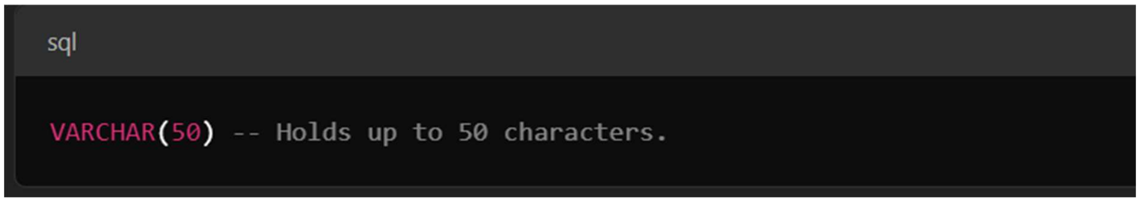
2. Character/String Data Types

- **CHAR(n)**: Fixed-length character string of length n



```sql
CHAR(10) -- Holds exactly 10 characters.
```

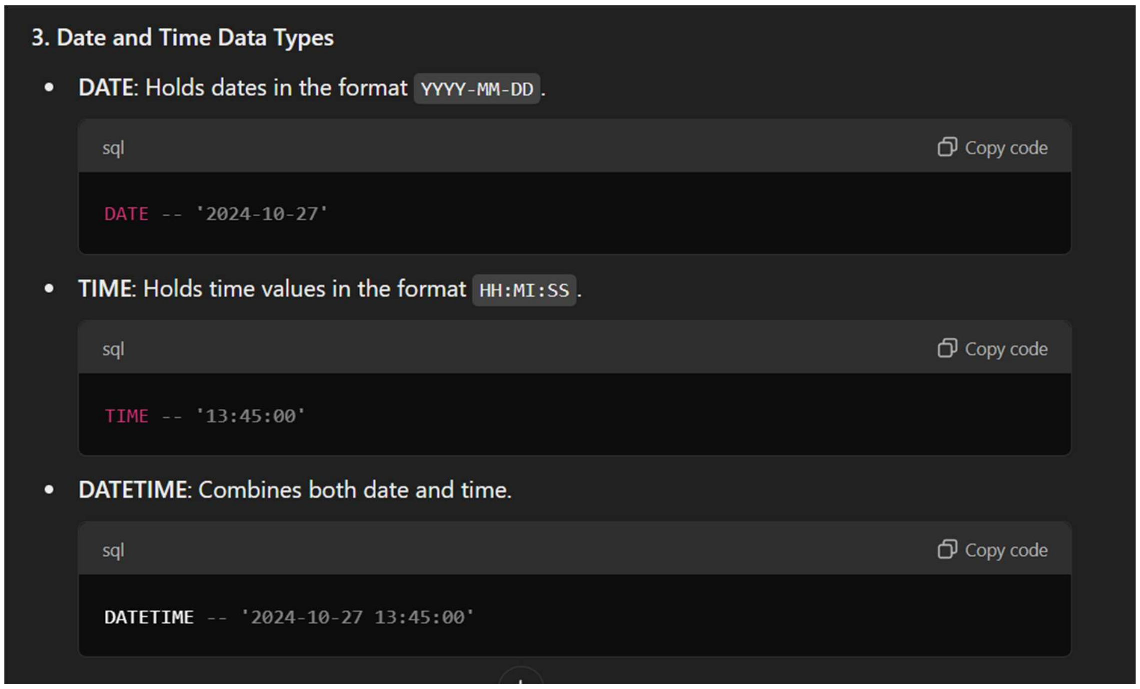
VARCHAR(n): Variable-length character string up to n characters



```sql
VARCHAR(50) -- Holds up to 50 characters.
```

Use **CHAR** when the data has a consistent length (like codes or abbreviations).

Use **VARCHAR** for data with varying lengths (like names, descriptions, or comments).



3. Date and Time Data Types

- DATE: Holds dates in the format `YYYY-MM-DD`.

```sql
DATE -- '2024-10-27'
```

- TIME: Holds time values in the format `HH:MI:SS`.

```sql
TIME -- '13:45:00'
```

- DATETIME: Combines both date and time.

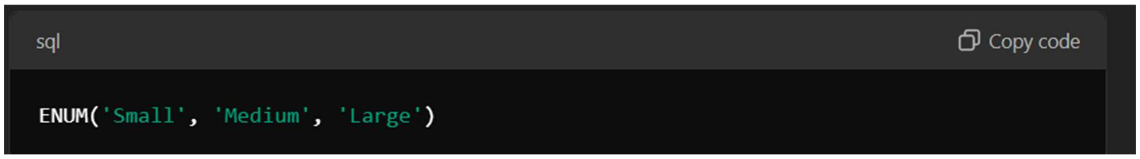
```sql
DATETIME -- '2024-10-27 13:45:00'
```

4. Binary Data Types

- **BINARY(n)**: Fixed-length binary data.
- **VARBINARY(n)**: Variable-length binary data.

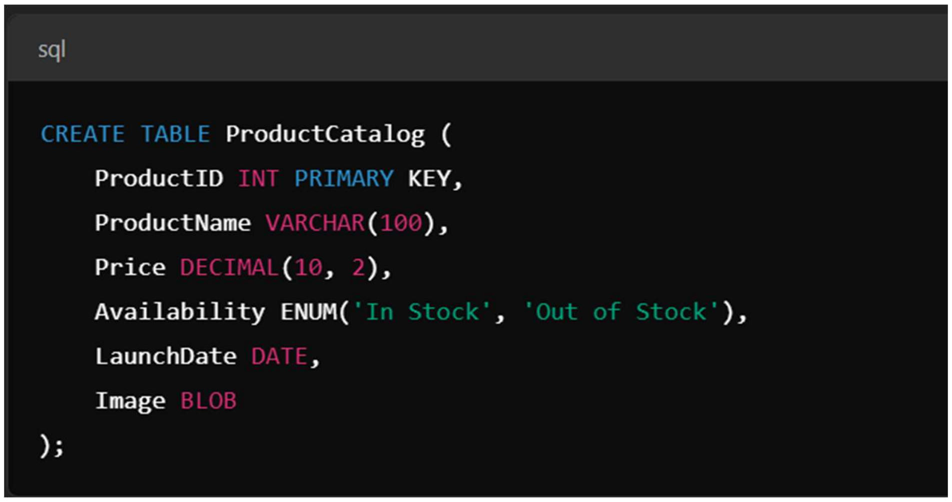
5. Miscellaneous Data Types

- **BOOLEAN**: Holds TRUE or FALSE values.
- **ENUM**: Lists a set of predefined values, with a single choice allowed.



```sql
ENUM('Small', 'Medium', 'Large')
```

BLOB: Used for large binary data, like images or audio files.

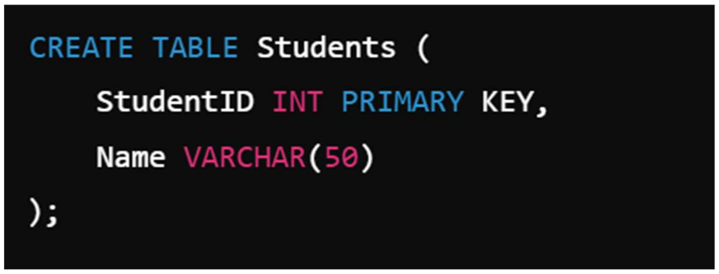


```sql
CREATE TABLE ProductCatalog (
    ProductID INT PRIMARY KEY,
    ProductName VARCHAR(100),
    Price DECIMAL(10, 2),
    Availability ENUM('In Stock', 'Out of Stock'),
    LaunchDate DATE,
    Image BLOB
);
```

Specifying Constraints in SQL

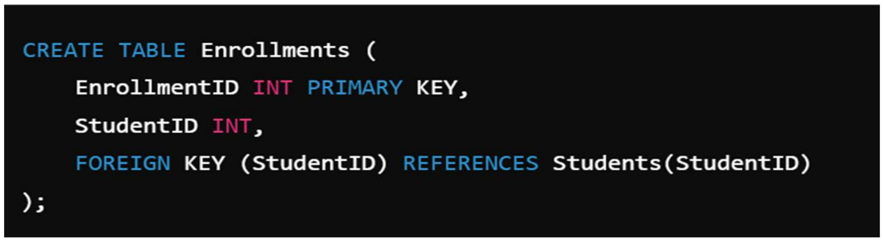
Constraints in SQL ensure data integrity and apply rules to columns within a table. Here are the main types of constraints:

1. **PRIMARY KEY**: Uniquely identifies each row in a table. A table can have only one primary key.



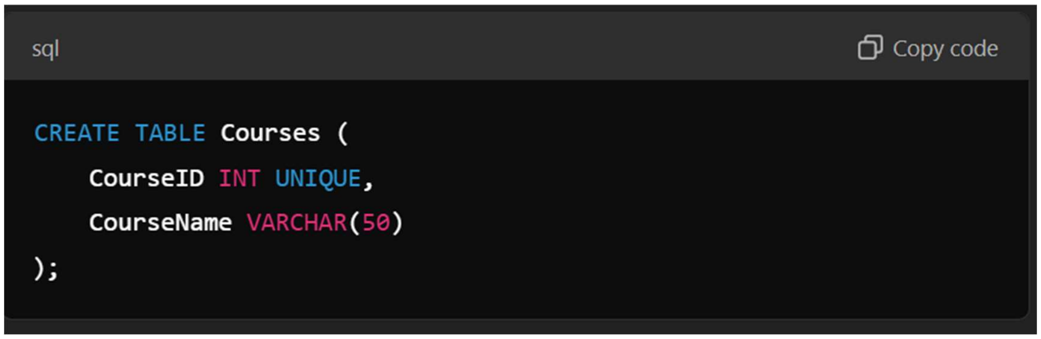
```sql
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    Name VARCHAR(50)
);
```

FOREIGN KEY: Links two tables, enforcing referential integrity between them.



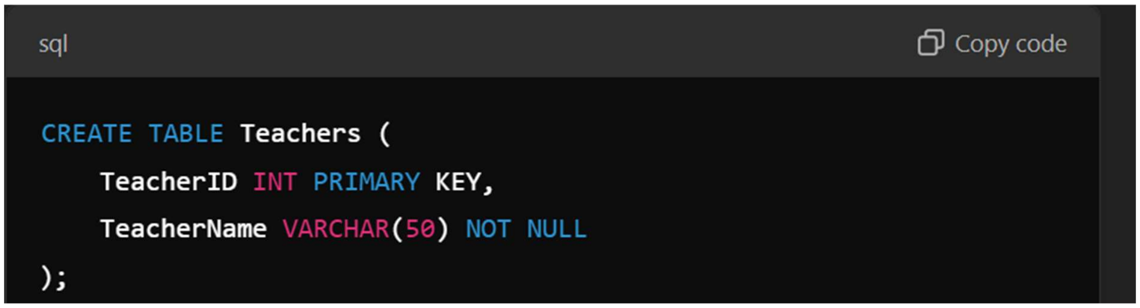
```sql
CREATE TABLE Enrollments (
    EnrollmentID INT PRIMARY KEY,
    StudentID INT,
    FOREIGN KEY (StudentID) REFERENCES Students(StudentID)
);
```

UNIQUE: Ensures all values in a column are unique.



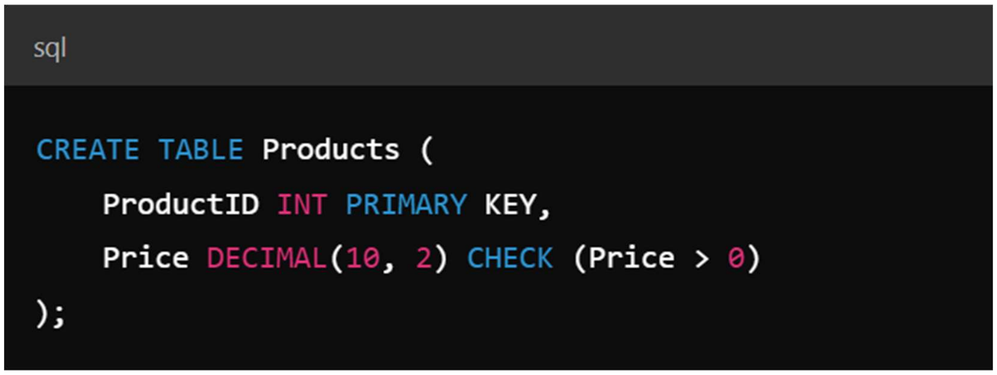
```sql
CREATE TABLE Courses (
    CourseID INT UNIQUE,
    CourseName VARCHAR(50)
);
```

NOT NULL: Prevents NULL values from being entered into a column.



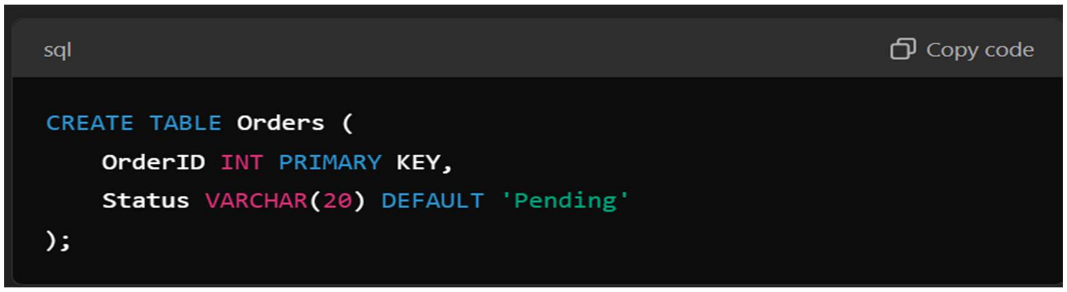
```sql
CREATE TABLE Teachers (
    TeacherID INT PRIMARY KEY,
    TeacherName VARCHAR(50) NOT NULL
);
```

CHECK: Enforces a condition that each row must satisfy.



```sql
CREATE TABLE Products (
    ProductID INT PRIMARY KEY,
    Price DECIMAL(10, 2) CHECK (Price > 0)
);
```

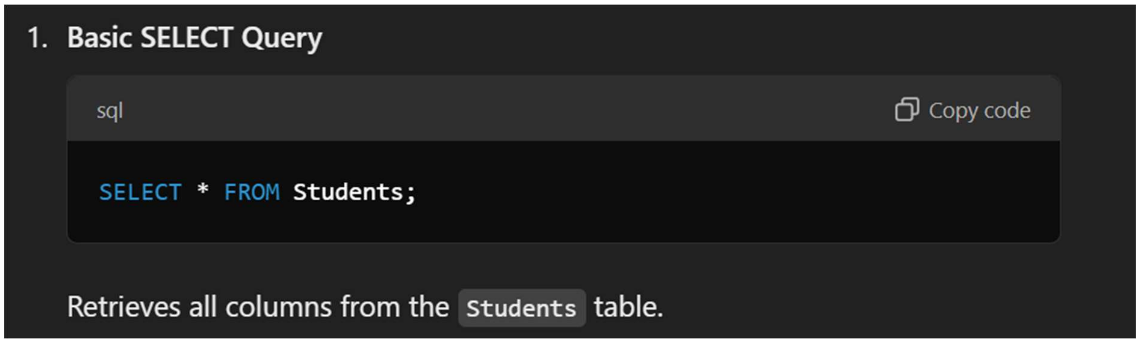
DEFAULT: Sets a default value if no value is specified during insertion.



```sql
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    Status VARCHAR(20) DEFAULT 'Pending'
);
```

Retrieval Queries in SQL

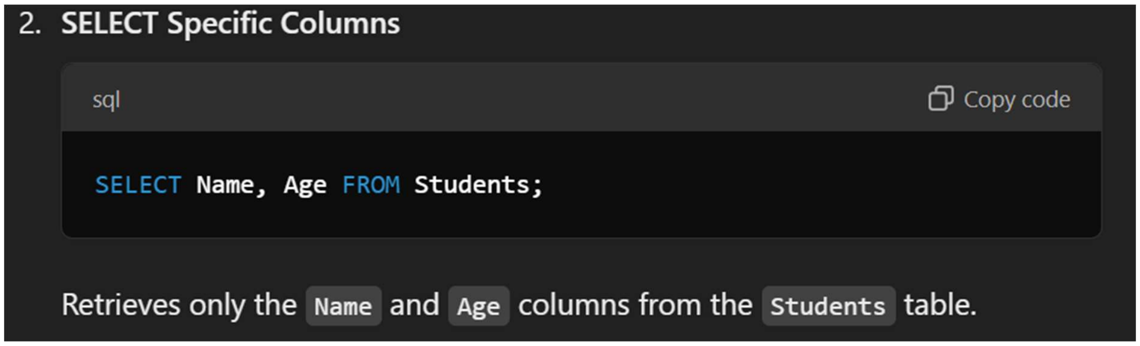
Retrieval queries, or SELECT statements, are used to retrieve data from a database. Here are examples of common retrieval queries:



1. **Basic SELECT Query**

```sql
SELECT * FROM Students;
```

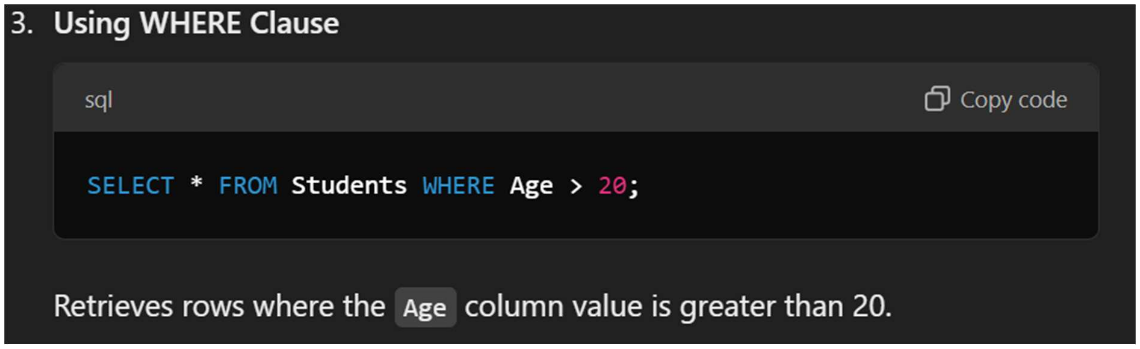
Retrieves all columns from the `Students` table.



2. **SELECT Specific Columns**

```sql
SELECT Name, Age FROM Students;
```

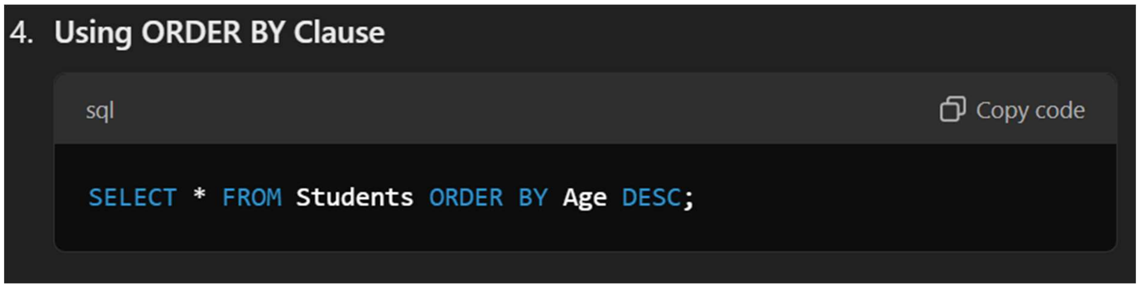
Retrieves only the `Name` and `Age` columns from the `Students` table.



3. **Using WHERE Clause**

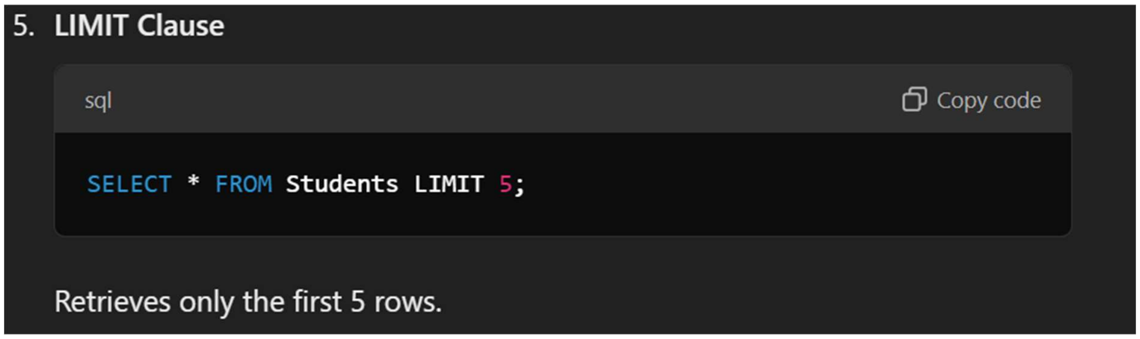
```sql
SELECT * FROM Students WHERE Age > 20;
```

Retrieves rows where the `Age` column value is greater than 20.



4. **Using ORDER BY Clause**

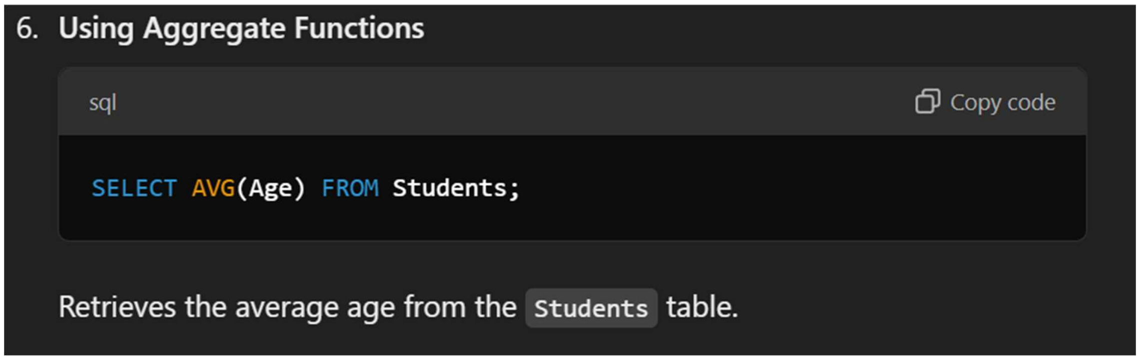
```sql
SELECT * FROM Students ORDER BY Age DESC;
```



5. **LIMIT Clause**

```sql
SELECT * FROM Students LIMIT 5;
```

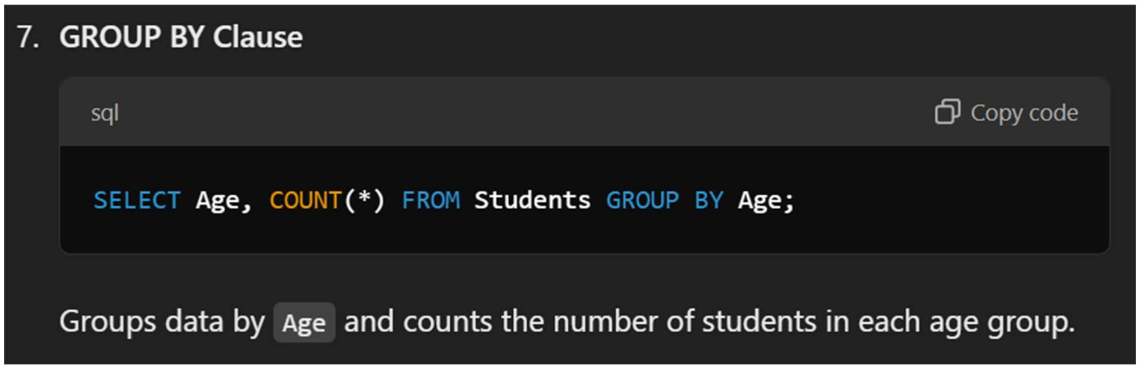
Retrieves only the first 5 rows.



6. **Using Aggregate Functions**

```sql
SELECT AVG(Age) FROM Students;
```

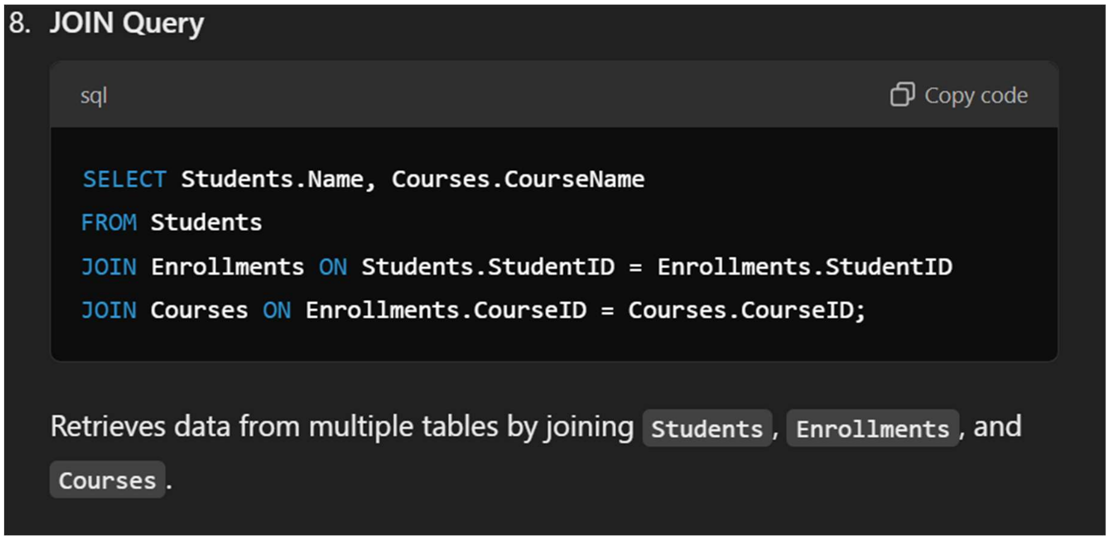
Retrieves the average age from the `Students` table.



7. **GROUP BY Clause**

```sql
SELECT Age, COUNT(*) FROM Students GROUP BY Age;
```

Groups data by `Age` and counts the number of students in each age group.



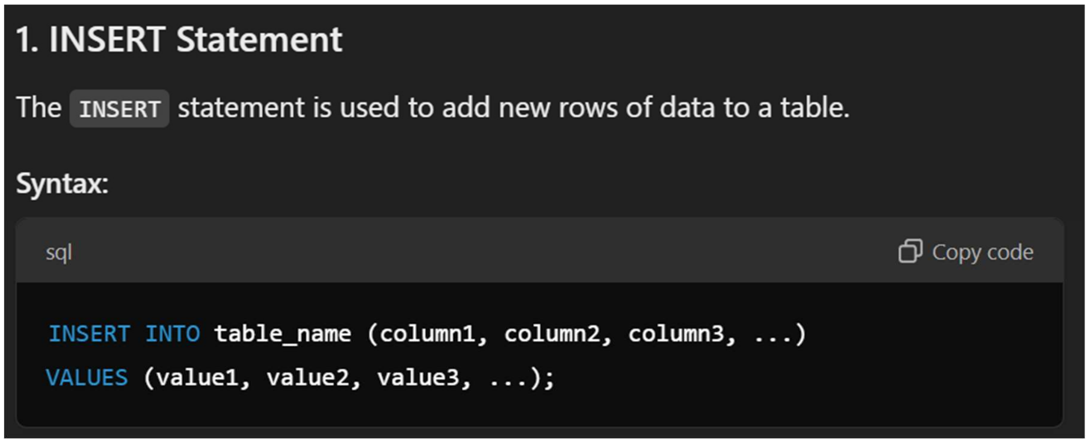
8. JOIN Query

sql

```sql
SELECT Students.Name, Courses.CourseName
FROM Students
JOIN Enrollments ON Students.StudentID = Enrollments.StudentID
JOIN Courses ON Enrollments.CourseID = Courses.CourseID;
```

Retrieves data from multiple tables by joining `Students`, `Enrollments`, and `Courses`.

In SQL, the INSERT, DELETE, and UPDATE statements are used to modify data in database tables. Here's how each works:



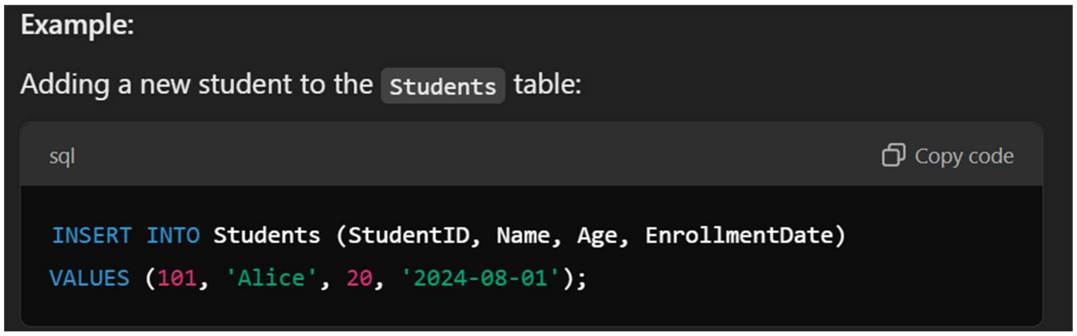
1. INSERT Statement

The `INSERT` statement is used to add new rows of data to a table.

Syntax:

sql

```sql
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

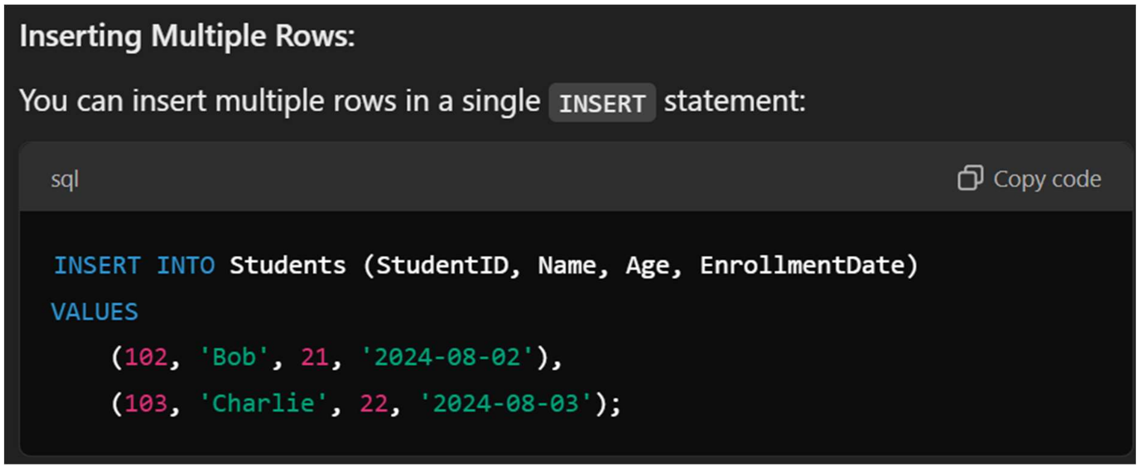


Example:

Adding a new student to the `Students` table:

sql

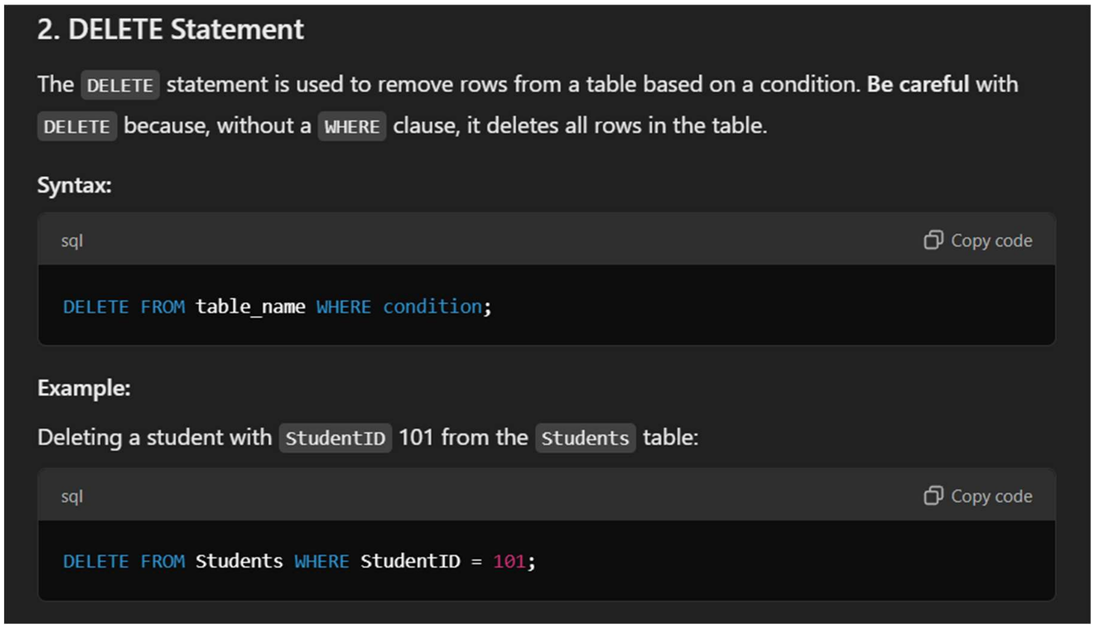
```sql
INSERT INTO Students (StudentID, Name, Age, EnrollmentDate)
VALUES (101, 'Alice', 20, '2024-08-01');
```



Inserting Multiple Rows:

You can insert multiple rows in a single `INSERT` statement:

```sql
INSERT INTO Students (StudentID, Name, Age, EnrollmentDate)
VALUES
    (102, 'Bob', 21, '2024-08-02'),
    (103, 'Charlie', 22, '2024-08-03');
```



2. DELETE Statement

The `DELETE` statement is used to remove rows from a table based on a condition. **Be careful** with `DELETE` because, without a `WHERE` clause, it deletes all rows in the table.

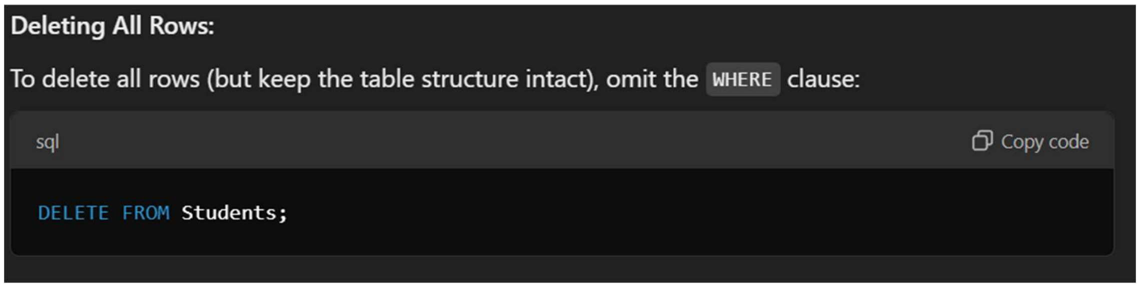
Syntax:

```sql
DELETE FROM table_name WHERE condition;
```

Example:

Deleting a student with `StudentID` 101 from the `Students` table:

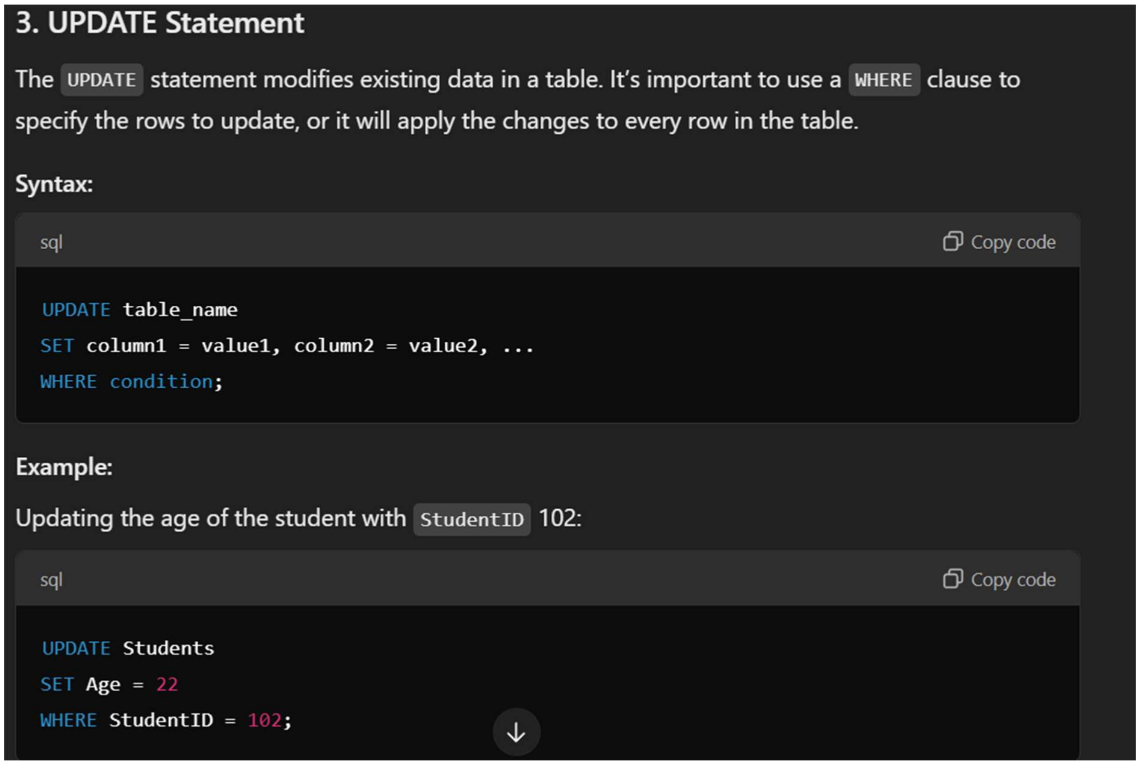
```sql
DELETE FROM Students WHERE StudentID = 101;
```



Deleting All Rows:

To delete all rows (but keep the table structure intact), omit the `WHERE` clause:

```sql
DELETE FROM Students;
```



3. UPDATE Statement

The `UPDATE` statement modifies existing data in a table. It's important to use a `WHERE` clause to specify the rows to update, or it will apply the changes to every row in the table.

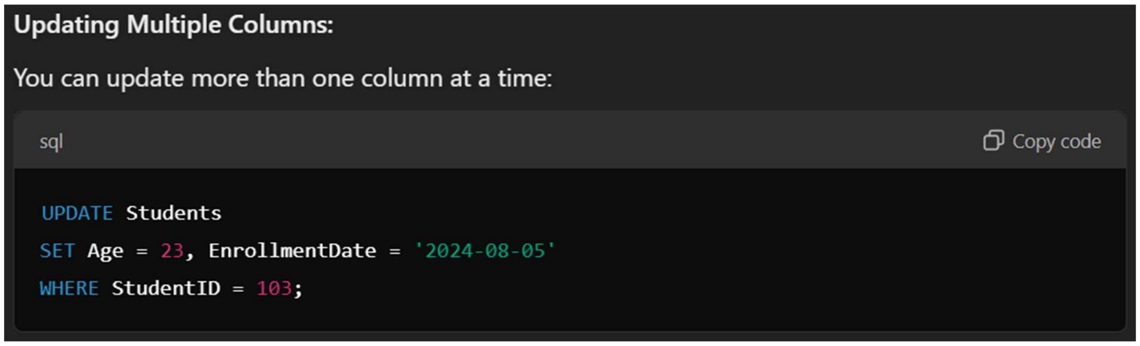
Syntax:

```sql
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

Example:

Updating the age of the student with `StudentID` 102:

```sql
UPDATE Students
SET Age = 22
WHERE StudentID = 102;
```

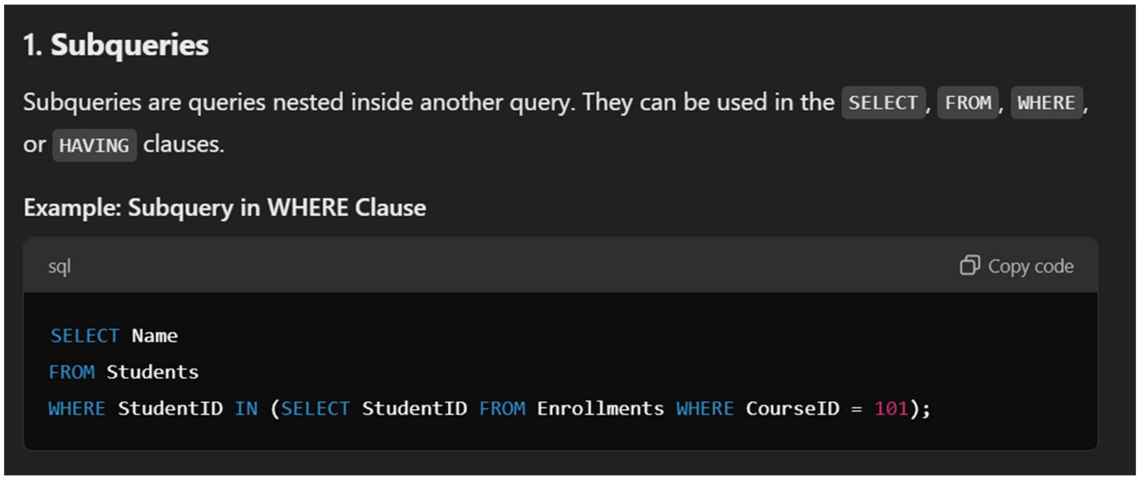


Updating Multiple Columns:

You can update more than one column at a time:

```sql
UPDATE Students
SET Age = 23, EnrollmentDate = '2024-08-05'
WHERE StudentID = 103;
```

Advances Queries: More complex SQL retrieval queries



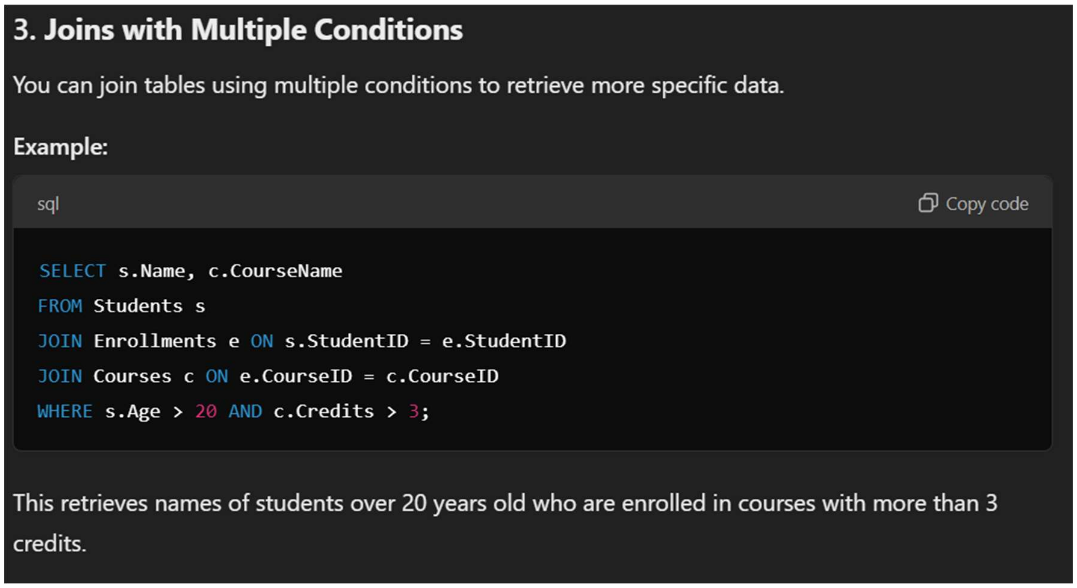
1. Subqueries

Subqueries are queries nested inside another query. They can be used in the `SELECT`, `FROM`, `WHERE`, or `HAVING` clauses.

Example: Subquery in WHERE Clause

```sql
SELECT Name
FROM Students
WHERE StudentID IN (SELECT StudentID FROM Enrollments WHERE CourseID = 101);
```

This retrieves the names of students enrolled in a specific course.



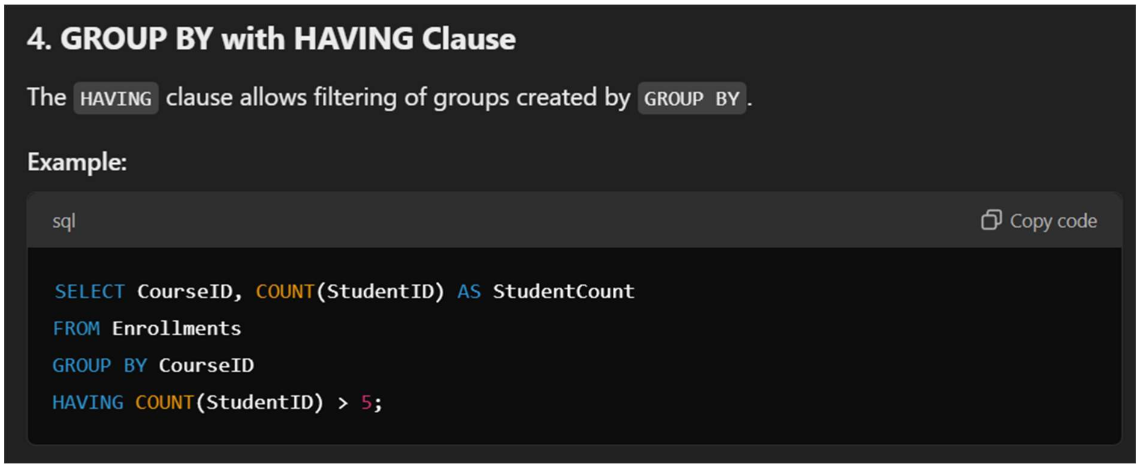
3. Joins with Multiple Conditions

You can join tables using multiple conditions to retrieve more specific data.

Example:

```sql
SELECT s.Name, c.CourseName
FROM Students s
JOIN Enrollments e ON s.StudentID = e.StudentID
JOIN Courses c ON e.CourseID = c.CourseID
WHERE s.Age > 20 AND c.Credits > 3;
```

This retrieves names of students over 20 years old who are enrolled in courses with more than 3 credits.



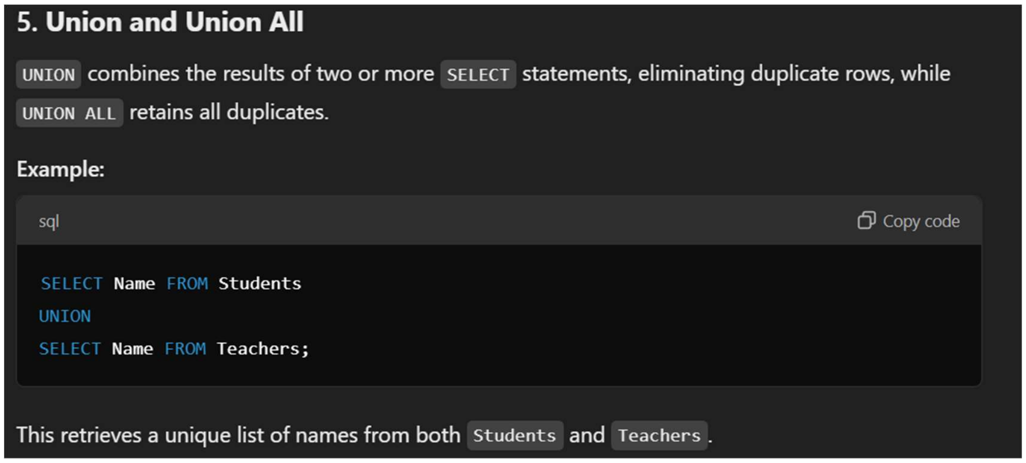
4. GROUP BY with HAVING Clause

The `HAVING` clause allows filtering of groups created by `GROUP BY`.

Example:

```sql
SELECT CourseID, COUNT(StudentID) AS StudentCount
FROM Enrollments
GROUP BY CourseID
HAVING COUNT(StudentID) > 5;
```

This retrieves the IDs of courses that have more than five enrolled students.



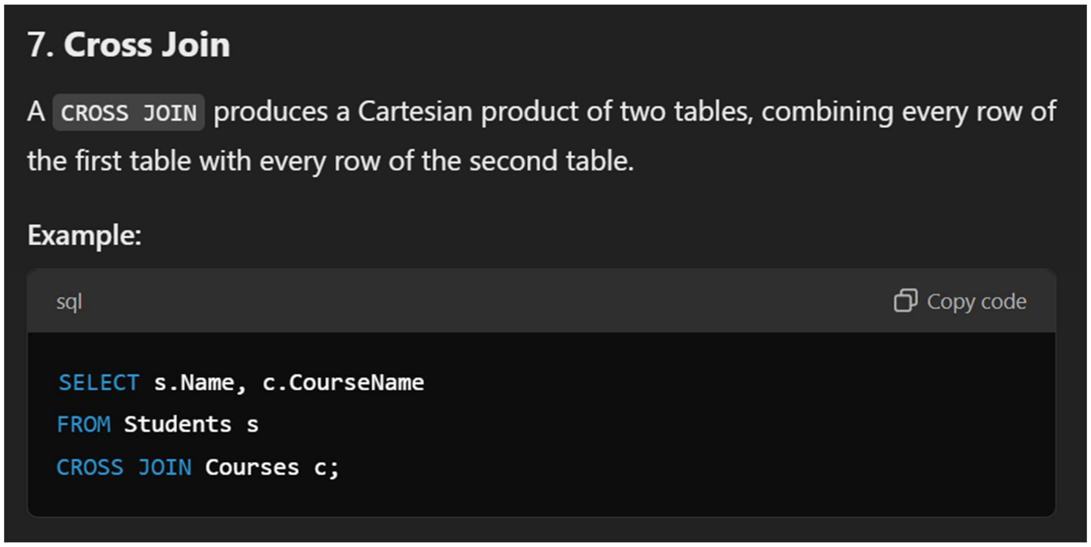
5. Union and Union All

`UNION` combines the results of two or more `SELECT` statements, eliminating duplicate rows, while `UNION ALL` retains all duplicates.

Example:

```sql
SELECT Name FROM Students
UNION
SELECT Name FROM Teachers;
```

This retrieves a unique list of names from both `Students` and `Teachers`.

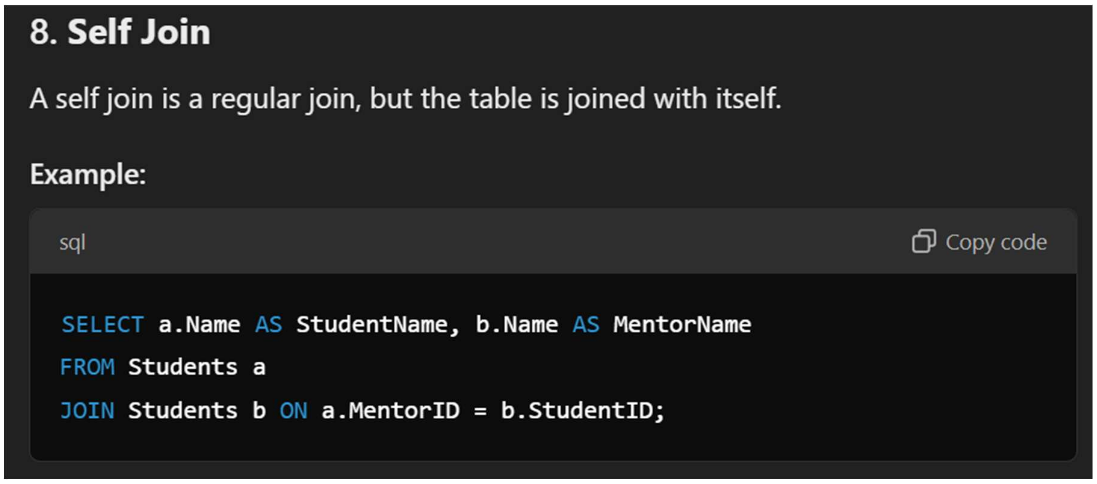


7. Cross Join

A `CROSS JOIN` produces a Cartesian product of two tables, combining every row of the first table with every row of the second table.

Example:

```sql
SELECT s.Name, c.CourseName
FROM Students s
CROSS JOIN Courses c;
```

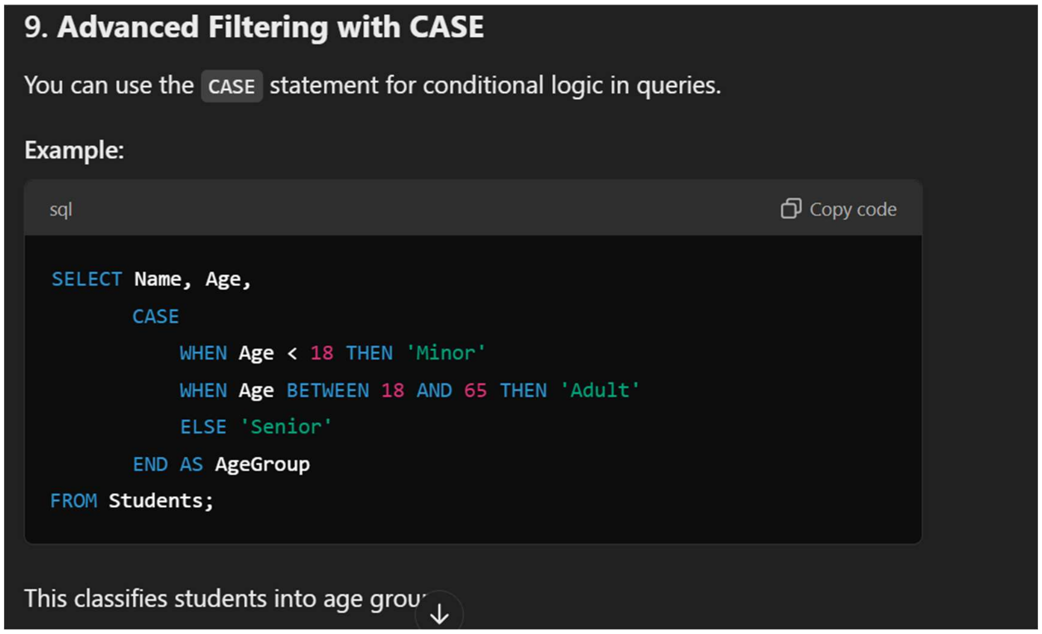


8. Self Join

A self join is a regular join, but the table is joined with itself.

Example:

```sql
SELECT a.Name AS StudentName, b.Name AS MentorName
FROM Students a
JOIN Students b ON a.MentorID = b.StudentID;
```



9. Advanced Filtering with CASE

You can use the `CASE` statement for conditional logic in queries.

Example:

```sql
SELECT Name, Age,
       CASE
           WHEN Age < 18 THEN 'Minor'
           WHEN Age BETWEEN 18 AND 65 THEN 'Adult'
           ELSE 'Senior'
       END AS AgeGroup
FROM Students;
```

This classifies students into age grou

1. Understanding EXISTS and NOT EXISTS

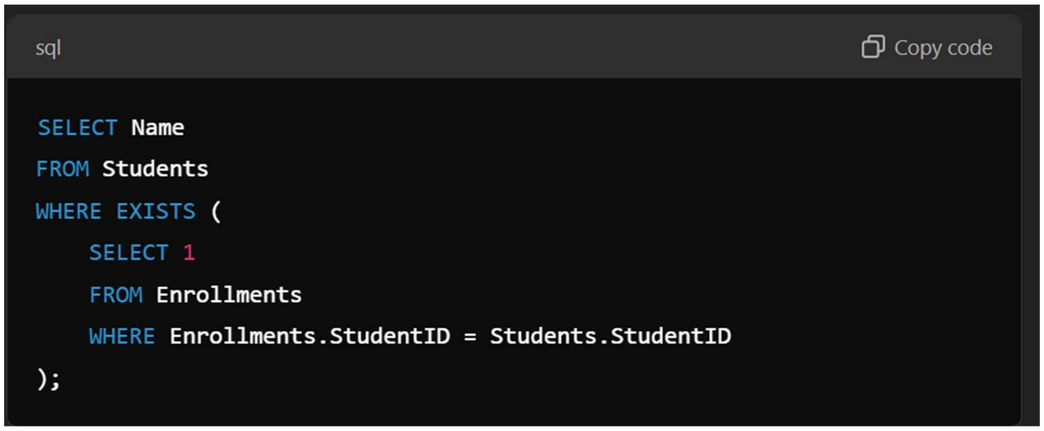
EXISTS and NOT EXISTS are used in SQL to check if certain data exists in a table based on a condition.

- **EXISTS**: This returns TRUE if there is at least one row that meets the specified condition.
- **NOT EXISTS**: This returns TRUE if **no rows** meet the specified condition.

Simple Example

Suppose we have two tables:

1. **Students**
   - Contains all student names and IDs.
   - Columns: StudentID, Name
2. **Enrollments**
   - Records which students are enrolled in which courses.
   - Columns: StudentID, CourseID

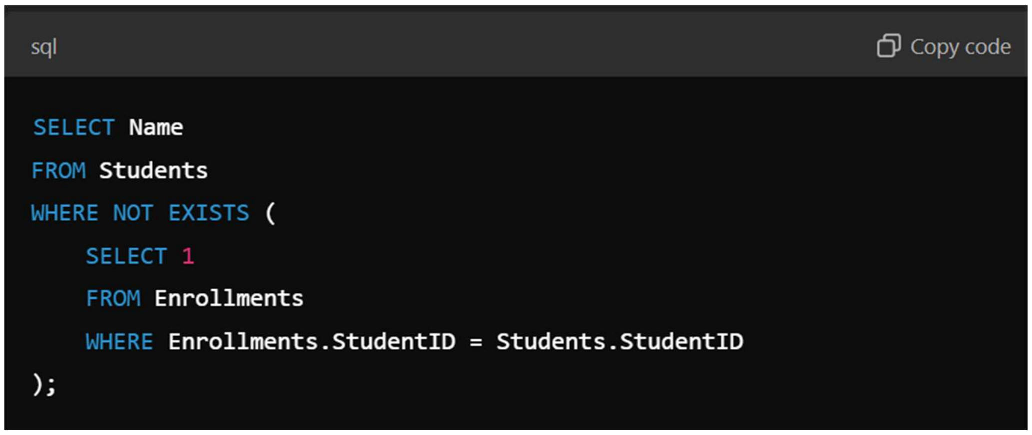


```sql
SELECT Name
FROM Students
WHERE EXISTS (
    SELECT 1
    FROM Enrollments
    WHERE Enrollments.StudentID = Students.StudentID
);
```

Explanation:

- The EXISTS checks if there's at least one entry in Enrollments for each student.
- If there is an enrollment for the StudentID, the student's name is included in the result.

Now, suppose we want to find **students who are not enrolled in any courses**.



```sql
SELECT Name
FROM Students
WHERE NOT EXISTS (
    SELECT 1
    FROM Enrollments
    WHERE Enrollments.StudentID = Students.StudentID
);
```

Explanation:

- NOT EXISTS checks if there's **no matching entry** in Enrollments for each StudentID.
- If a student's StudentID is not found in Enrollments, they are included in the result.

Explanation:

- NOT EXISTS checks if there's **no matching entry** in Enrollments for each StudentID.
- If a student's StudentID is not found in Enrollments, they are included in the result.

What is `CREATE ASSERTION`?

- The `CREATE ASSERTION` statement lets you create a **custom rule** for data integrity that must be met by the data in your tables. If any operation (like an `INSERT`, `UPDATE`, or

DELETE) would result in the assertion condition being false, the operation will fail, maintaining the integrity of your data.

Example: Enforcing an Assertion

Let's say we have the following scenario:

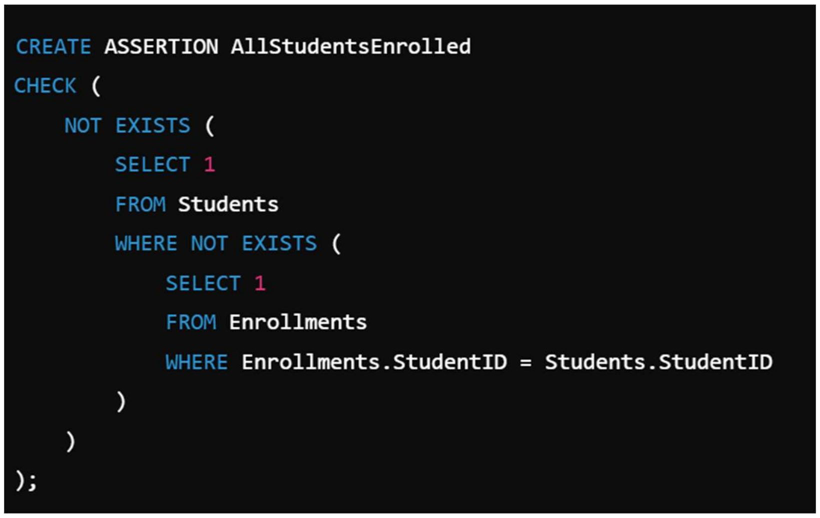
- **Every student must be enrolled in at least one course**.

Suppose we have two tables:

1. **Students**
   - Columns: StudentID, Name
2. **Enrollments**
   - Columns: StudentID, CourseID

Assertion Example

To enforce that every student in the Students table has at least one corresponding enrollment in the Enrollments table, we would write an assertion like this:



```sql
CREATE ASSERTION AllStudentsEnrolled
CHECK (
    NOT EXISTS (
        SELECT 1
        FROM Students
        WHERE NOT EXISTS (
            SELECT 1
            FROM Enrollments
            WHERE Enrollments.StudentID = Students.StudentID
        )
    )
);
```

Explanation of the Assertion

1. **Inner NOT EXISTS**:

    - SELECT 1 FROM Enrollments WHERE Enrollments.StudentID = Students.StudentID
    - This part checks if there is no entry in Enrollments for a StudentID. If a student does not have any matching entry in Enrollments, this returns TRUE.

2. **Outer NOT EXISTS**:
    - SELECT 1 FROM Students WHERE NOT EXISTS (...)
    - This ensures that there are **no students** without at least one enrollment. If there is any student without an enrollment, the assertion would fail, and any operation that violates this condition would be prevented.

What Happens When the Assertion is Violated?

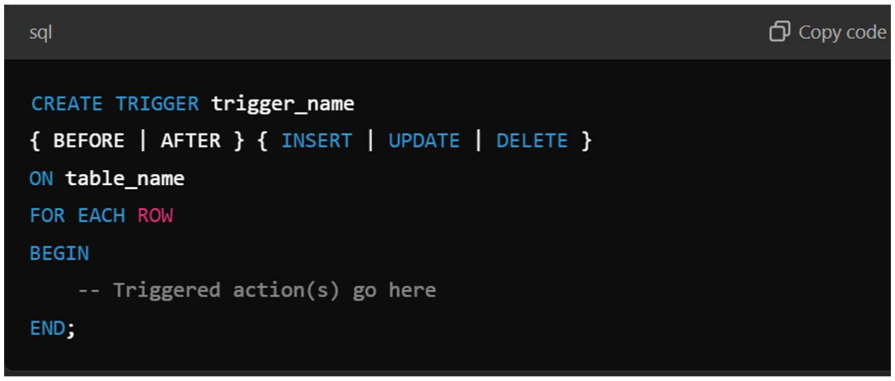
Once this assertion is in place:

- If someone tries to add a new student without an enrollment, the operation will fail.
- If someone tries to delete a student's enrollment, leaving them with none, it will also fail.

ACTION TRIGGERS

Action triggers in SQL are special routines that automatically execute specified actions in response to events like INSERT, UPDATE, or DELETE on a table. Triggers are often used to maintain **data integrity, log changes, enforce business rules, or automatically update related tables.**

Basic Trigger Structure



```sql
CREATE TRIGGER trigger_name
{ BEFORE | AFTER } { INSERT | UPDATE | DELETE }
ON table_name
FOR EACH ROW
BEGIN
    -- Triggered action(s) go here
END;
```

▯ **Trigger Timing** (BEFORE or AFTER): Specifies if the trigger should execute **before** or **after** the triggering event.

- BEFORE: Runs the trigger action before the database modification takes place.
- AFTER: Runs the trigger action after the modification is completed.

▯ **Triggering Event** (INSERT, UPDATE, DELETE): Specifies the type of data modification event that will activate the trigger.

▯ **Target Table**: The table that the trigger watches for changes.

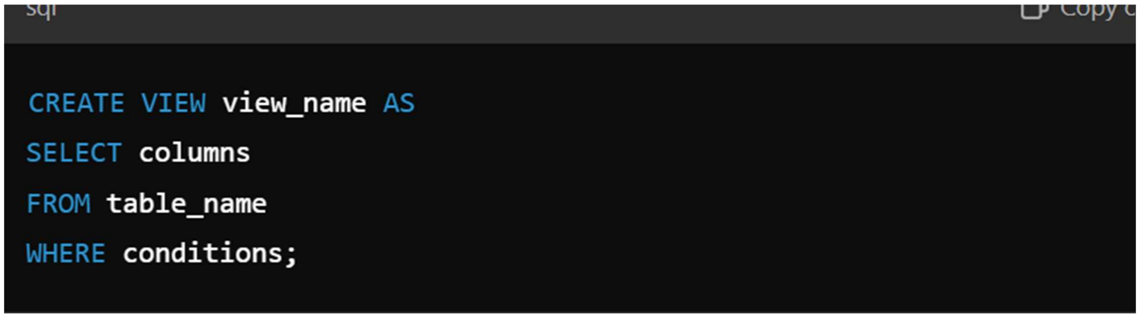
▯ **Trigger Action**: The SQL statements that define what happens when the trigger fires. This section is enclosed in BEGIN ... END.

SQL VIEWS

In SQL, **views** are virtual tables that are based on the result of a SELECT query. Views don't store data themselves; instead, they pull data from the underlying tables when accessed, providing a way to simplify complex queries, enhance security, and abstract data.

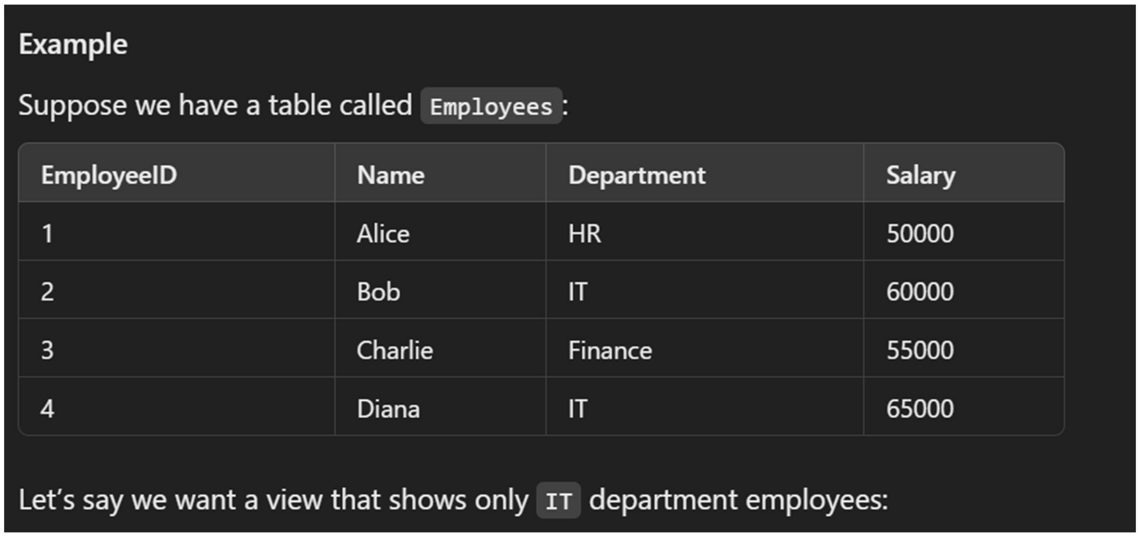
Why Use Views?

1. **Simplify Complex Queries**: Views can make it easier to manage and reuse complex SQL queries, as they encapsulate the query logic.
2. **Enhanced Security**: Views allow you to restrict access to certain columns or rows within a table, providing a level of data security.
3. **Data Abstraction**: Views help in hiding the complexity of data and presenting it in a user-friendly format.
4. **Improved Maintainability**: If the structure or logic of data changes, you can modify the view rather than updating multiple queries.



```sql
CREATE VIEW view_name AS
SELECT columns
FROM table_name
WHERE conditions;
```

- **view_name**: The name of the view.
- **SELECT columns...WHERE conditions**: The SELECT statement that defines what data the view will contain.

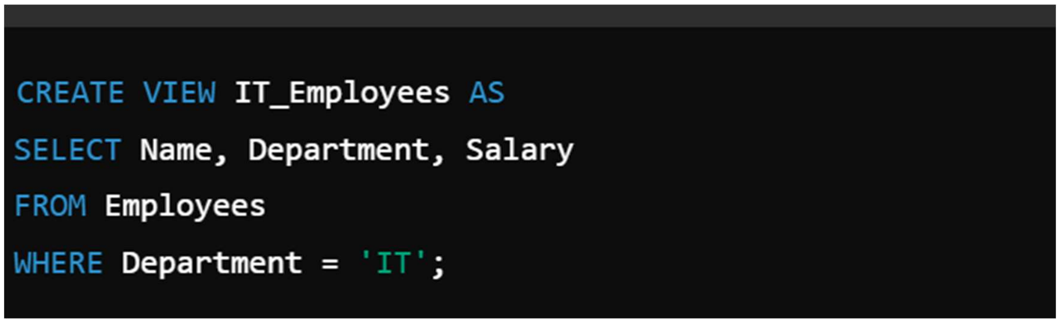


Example

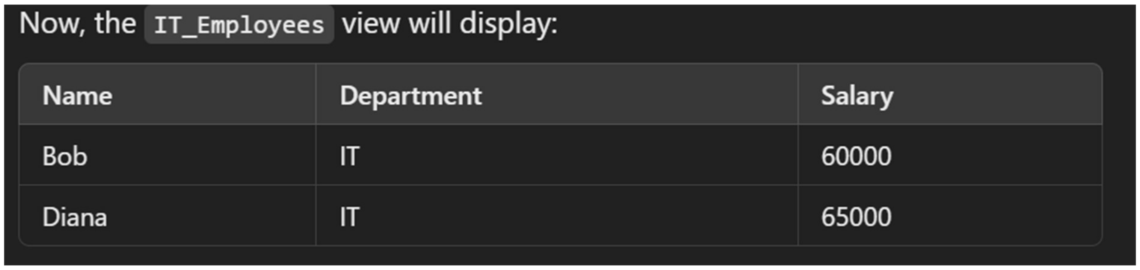
Suppose we have a table called `Employees`:

EmployeeID	Name	Department	Salary
1	Alice	HR	50000
2	Bob	IT	60000
3	Charlie	Finance	55000
4	Diana	IT	65000

Let's say we want a view that shows only `IT` department employees:



```sql
CREATE VIEW IT_Employees AS
SELECT Name, Department, Salary
FROM Employees
WHERE Department = 'IT';
```



Now, the `IT_Employees` view will display:

Name	Department	Salary
Bob	IT	60000
Diana	IT	65000

Using Views

To retrieve data from a view, you use a SELECT statement just as you would with a regular table:

```
SELECT * FROM IT_Employees;
```

Updating Data Through Views

In some cases, you can **update** data through a view if it meets certain criteria (like not containing joins, aggregations, or certain SQL functions). When updating a view, it actually updates the underlying table.

```
UPDATE IT_Employees
SET Salary = 62000
WHERE Name = 'Bob';
```

After this, Bob's salary in the Employees table would be updated to 62000.

Types of Views

1. **Simple View**: Based on a single table and doesn't contain complex SQL functions.
2. **Complex View**: Can involve multiple tables, joins, aggregations, and SQL functions. Complex views are typically read-only.

Dropping a View

If you no longer need a view, you can drop it with:

DROP VIEW view_name;

For example:

DROP VIEW IT_Employees;

Advantages and Limitations of Views

Advantages:

- Simplify complex queries by encapsulating them.
- Restrict data access for security purposes.
- Hide data complexity from end users.

Limitations:

- Views that include joins or aggregations are typically read-only.
- Views can sometimes impact performance if based on complex queries or very large tables.

Why Use a View?

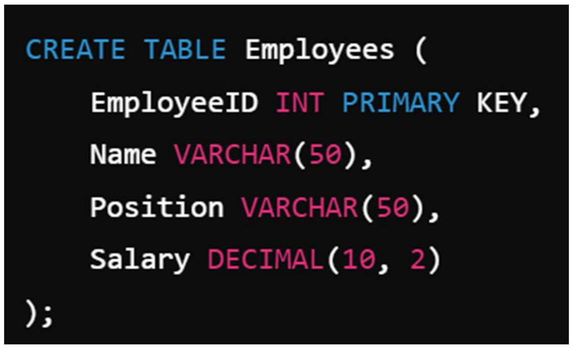
1. **Simplifies Repeated Queries**: Instead of retyping or copying the complex query every time, you can just reference the view's name. So, you aren't re-creating the complex query each time but are **calling a pre-set query**.
2. **Keeps Data Updated**: Every time you call the view, SQL fetches current data from the source tables (Customers and Orders). This means you don't need to manually refresh the view—it always shows the latest data when queried.
3. **Data Security and Control**: A view can limit access to sensitive data by only exposing selected columns or filtered rows, allowing users to see summarized or specific data without direct access to the base tables.

Schema change statements in SQL

Schema change statements in SQL allow you to modify the structure of your database tables and other objects, such as adding or removing columns, changing data types, renaming tables, or adjusting constraints. These statements are critical when adapting a database to new requirements.

1. CREATE TABLE

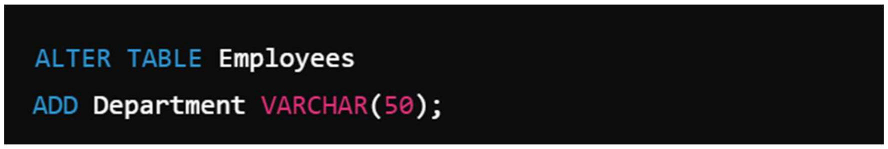
The CREATE TABLE statement is used to create a new table in the database. You define the table name, columns, and data types.



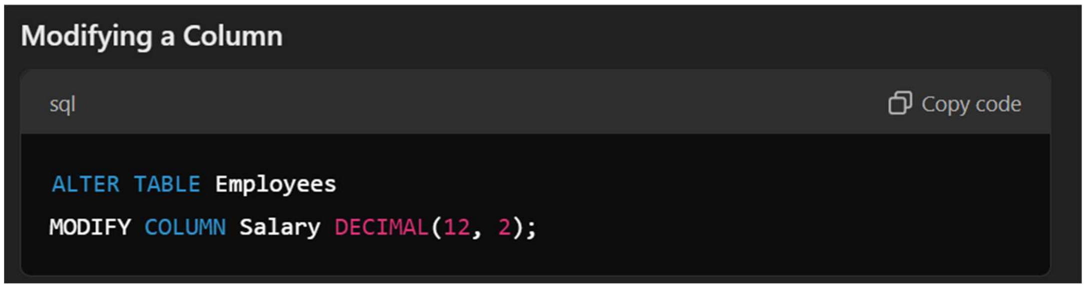
```sql
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    Name VARCHAR(50),
    Position VARCHAR(50),
    Salary DECIMAL(10, 2)
);
```

2. ALTER TABLE

The ALTER TABLE statement **modifies an existing table's structure**. You can **add, modify, or delete columns, and manage constraints**.

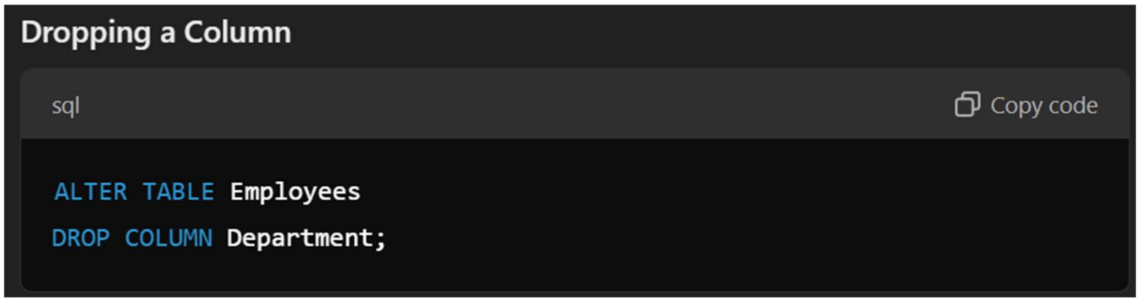


```sql
ALTER TABLE Employees
ADD Department VARCHAR(50);
```



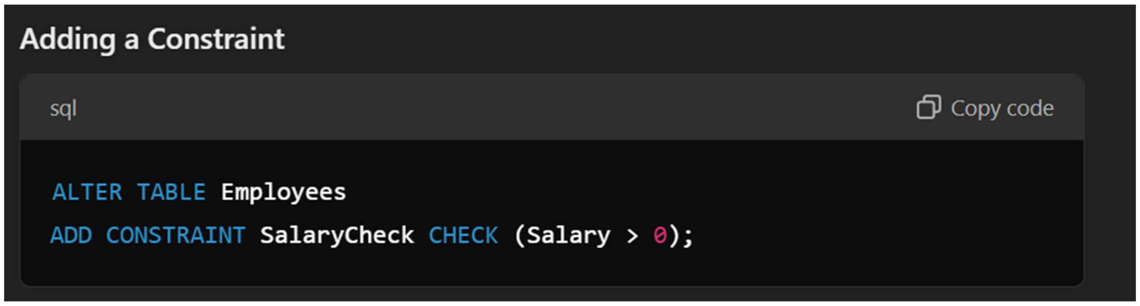
Modifying a Column

```sql
ALTER TABLE Employees
MODIFY COLUMN Salary DECIMAL(12, 2);
```



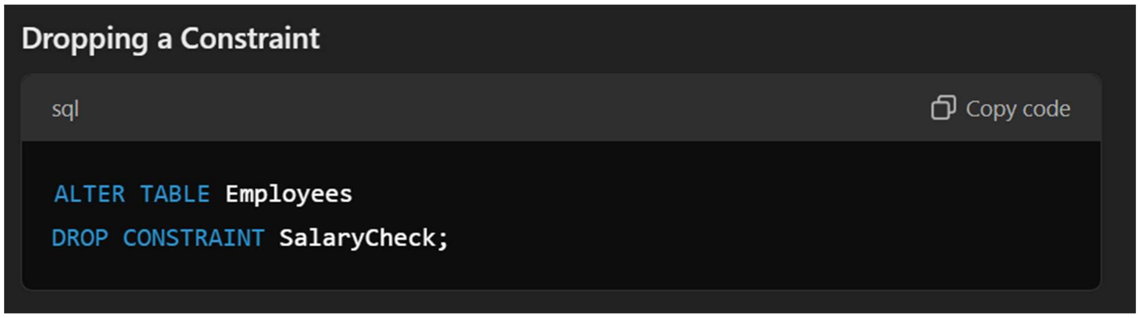
Dropping a Column

```sql
ALTER TABLE Employees
DROP COLUMN Department;
```



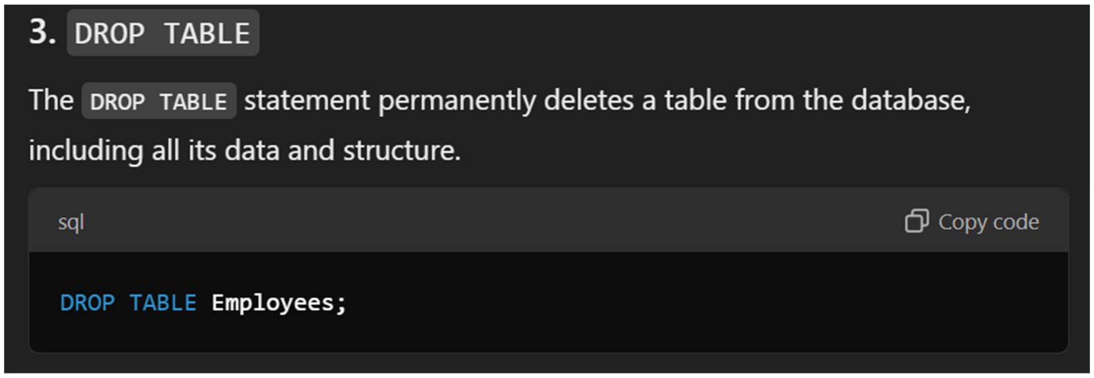
Adding a Constraint

```sql
ALTER TABLE Employees
ADD CONSTRAINT SalaryCheck CHECK (Salary > 0);
```



Dropping a Constraint

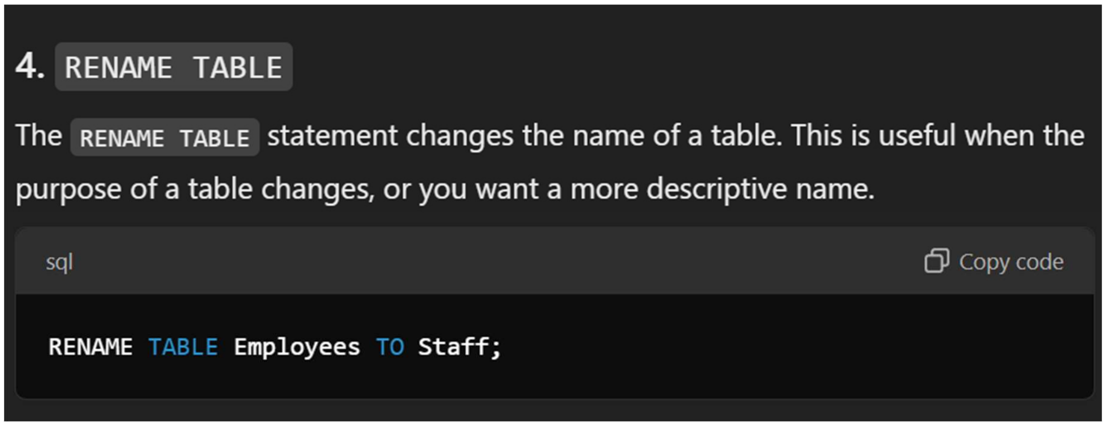
```sql
ALTER TABLE Employees
DROP CONSTRAINT SalaryCheck;
```



3. `DROP TABLE`

The `DROP TABLE` statement permanently deletes a table from the database, including all its data and structure.

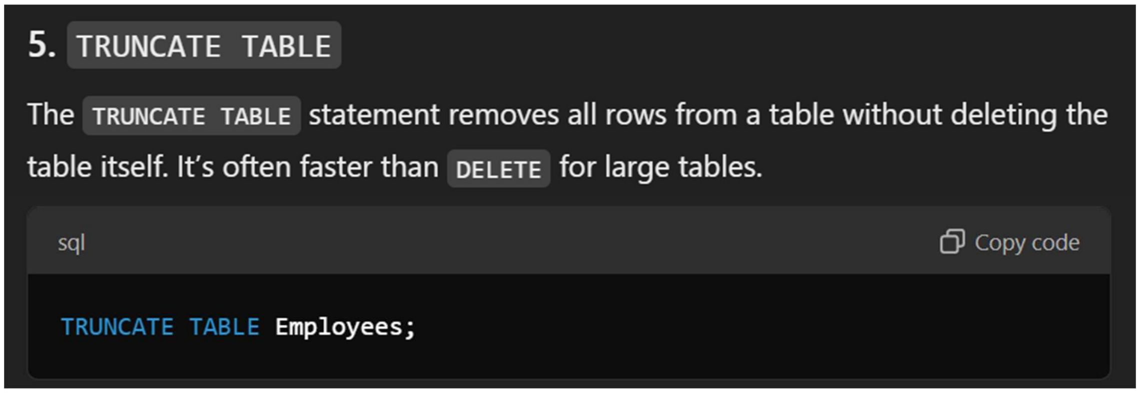
```sql
DROP TABLE Employees;
```



4. `RENAME TABLE`

The `RENAME TABLE` statement changes the name of a table. This is useful when the purpose of a table changes, or you want a more descriptive name.

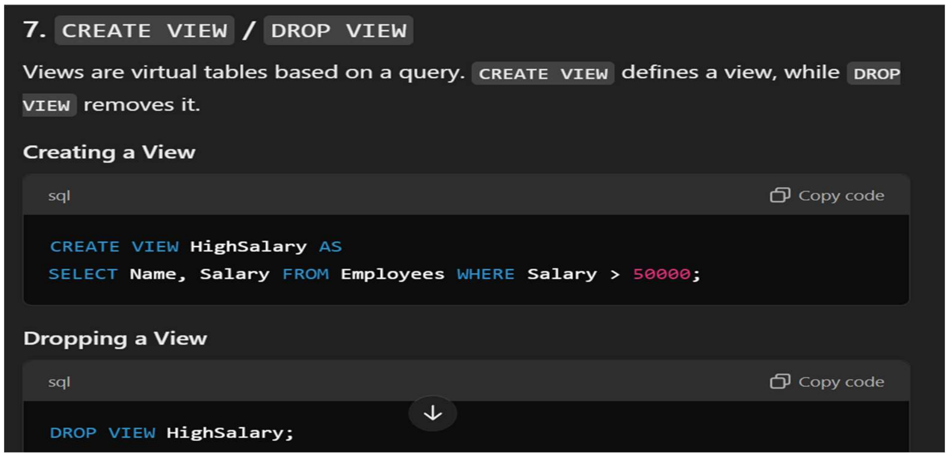
```sql
RENAME TABLE Employees TO Staff;
```



5. `TRUNCATE TABLE`

The `TRUNCATE TABLE` statement removes all rows from a table without deleting the table itself. It's often faster than `DELETE` for large tables.

```sql
TRUNCATE TABLE Employees;
```



7. `CREATE VIEW` / `DROP VIEW`

Views are virtual tables based on a query. `CREATE VIEW` defines a view, while `DROP VIEW` removes it.

Creating a View

```sql
CREATE VIEW HighSalary AS
SELECT Name, Salary FROM Employees WHERE Salary > 50000;
```

Dropping a View

```sql
DROP VIEW HighSalary;
```