

Database Management System

IT 261

Bachelor of Information Technology

Prepared by Dr. Khaled Eskaf

Lecture 6





Course Outlines

Week	Topic	Objective	MySQL Practical Activities
1	Introduction to Database Concepts	Understand basic database concepts.	Install MySQL, explore the interface, create a basic database.
2	Relational Data Model	Learn about the relational data model.	Create relational tables in MySQL.
3	Relational Integrity Rules	Understand relational integrity constraints.	Implement primary and foreign key constraints in MySQL.
4	Relational Algebra and Basic SQL	Explore relational algebra and basic SQL commands.	Use basic SQL queries in MySQL.
5	Advanced SQL Queries	Delve into complex SQL operations.	Execute advanced SQL commands in MySQL.
6	Introduction to Database Normalization	Begin learning database normalization principles.	Apply the first normal form (1NF) in MySQL.
7	Advanced Database Normalization	Explore advanced normalization techniques.	Normalize a database in MySQL to 2NF and 3NF.
8	Data Modeling and ER Diagrams	Understand data modeling and ER diagrams.	Create an ER diagram and implement it in MySQL.
9	Database Design for Real-World Apps	Apply database design to real-world scenarios.	Develop a database schema based on a real-world example.
10	Project and Review	Apply all learned concepts in a comprehensive project.	Complete a project in MySQL incorporating all learned skills.

Revision

	Algebra and	Explore relational algebra and basic	
4	Basic SQL	SQL commands.	Use basic SQL queries in MySQL.
	Advanced		Execute advanced SQL commands
5	SQL Queries	Delve into complex SQL operations.	in MySQL.
	Introduction		
	to Database	Begin learning database	Apply the first normal form (1NF) in

Apart from these common operations Relational Algebra is also used for **Join** operations like,

- Natural Join
- Conditional Join

Join Operation

When we have two relations that have a single column (let's just call this column *C*) in common, we can perform what is known as a **natural join**. Essentially two tables are naturally joined by taking the cross product of the two tables, then eliminating all combinations in which the column *C* from table 1 and column *C* from table 2 are not equal.

So let's take a look at a table of food, and the corresponding best way to obtain them.

FOODMETHOD

Food	ObtainFrom
Pizza	Restaurant
Curry	Restaurant
Swordfish	Fishing
Eel	Fishing

PERSON

Name	Age	FavoriteFood
John	15	Pizza
Stacy	32	Curry
Glenn	27	Swordfish

FOODMETHOD

Food	ObtainFrom
Pizza	Restaurant
Curry	Restaurant
Swordfish	Fishing
Eel	Fishing

PERSON

Name	Age	FavoriteFood
John	15	Pizza
Stacy	32	Curry
Glenn	27	Swordfish

PERSON  FOODMETHOD

Name	Age	FavoriteFood	ObtainFrom
John	15	Pizza	Restaurant
Stacy	32	Curry	Restaurant
Glenn	27	Swordfish	Fishing

There are other ways to join two tables, especially for tables which don't have a **common column**. This is known as theta join. This operation joins two tables based on **a condition** that is defined when declaring the join.

Car		Boat		$Car \bowtie_{CarPrice \geq BoatPrice} Boat$			
CarModel	CarPrice	BoatModel	BoatPrice	CarModel	CarPrice	BoatModel	BoatPrice
CarA	20,000	Boat1	10,000	CarA	20,000	Boat1	10,000
CarB	30,000	Boat2	40,000	CarB	30,000	Boat1	10,000
CarC	50,000	Boat3	60,000	CarC	50,000	Boat1	10,000
				CarC	50,000	Boat2	40,000

The condition of this join was all combinations where the Car Price is greater than or equal to the Boat Price. For whatever reason, our potential customer wants a car and boat, but doesn't want to spend more on the boat than the car.

Student

S_id	Name	Class	Age	C_id
1	Andrew	5	25	11
2	Angel	10	30	11
3	Anamika	8	35	22

Course

C_id	C_name
11	Foundation C
21	C++

Student ⋈ Course

Student ⋈ Course

S_id	Name	Class	Age	C_id	C_name
1	Andrew	5	25	11	Foundation C
2	Angel	10	30	11	Foundation C

What is SQL?

Structured query language (SQL) is a programming language for storing and processing information in a relational database.

- SQL stands for Structured Query Language
- SQL lets you access and manipulate databases
- SQL became a standard of the American National Standards Institute (ANSI)



What Can SQL do?

- SQL can execute queries against a database
- SQL can retrieve data from a database
- SQL can insert records in a database
- SQL can update records in a database
- SQL can delete records from a database
- SQL can create new databases
- SQL can create new tables in a database
- SQL can create stored procedures in a database
- SQL can create views in a database
- SQL can set permissions on tables, procedures, and views

The SQL CREATE DATABASE Statement

The **CREATE DATABASE** statement is used to create a new SQL database.

Syntax

```
CREATE DATABASE databasename;
```

The SQL CREATE TABLE Statement

The **CREATE TABLE** statement is used to create a new table in a database.

Syntax

```
CREATE TABLE table_name (  
    column1 datatype,  
    column2 datatype,  
    column3 datatype,  
    ....  
);
```

SQL CREATE TABLE Example

The following example creates a table called "Persons" that contains five columns: PersonID, LastName, FirstName, Address, and City:

```
CREATE TABLE Persons (  
    PersonID int,  
    LastName varchar(255),  
    FirstName varchar(255),  
    Address varchar(255),  
    City varchar(255)  
);
```

The empty "Persons" table will now look like this:

PersonID	LastName	FirstName	Address	City

MySQL supports a variety of data types, categorized into three main types: numeric, string (character and byte), and date/time.

Numeric Types

- **INT**: A standard integer, capable of storing both positive and negative numbers.
- **BIGINT**: A larger integer for when **INT** is not sufficient.
- **FLOAT**: A single-precision floating point number for approximate numeric data.
- **DOUBLE**: A double-precision floating point number for more precision.
- **DECIMAL**: For fixed-point numbers, offering exact precision; useful for storing monetary values.

String Types

- **VARCHAR**: A variable-length string, with a defined maximum length.
- **CHAR**: A fixed-length string; if the string is shorter than the defined length, it's right-padded with spaces.
- **TEXT**: For long text data, storing up to 65,535 bytes.
- **BLOB**: For binary data, such as images or files, with a maximum length similar to **TEXT**.

Date and Time Types

- **DATE**: Stores a date (YYYY-MM-DD).
- **TIME**: Stores time (HH:MM:SS).
- **DATETIME**: Stores both date and time (YYYY-MM-DD HH:MM:SS).
- **TIMESTAMP**: Similar to **DATETIME**, but used for tracking changes, automatically updating upon row modification.
- **YEAR**: Stores a year in a 2-digit or 4-digit format.

```
CREATE TABLE NumericExamples (  
  id INT AUTO_INCREMENT PRIMARY KEY,  
  intValue INT,  
  bigIntValue BIGINT,  
  floatValue FLOAT,  
  doubleValue DOUBLE,  
  decimalValue DECIMAL(10,2)  
);
```

In this table:

- **id** is a unique identifier for each row, automatically incrementing.
- **intValue** will store **INT** values.
- **bigIntValue** is for **BIGINT** values, useful for very large numbers.
- **floatValue** will contain **FLOAT** type numbers, which are single-precision floating point numbers.
- **doubleValue** stores **DOUBLE** type numbers, which are double-precision floating point numbers for more accuracy.
- **decimalValue** is a **DECIMAL** type, precise for fixed-point numbers like monetary values, defined to hold a maximum of 10 digits, with 2 digits after the decimal point.

```
INSERT INTO NumericExamples (intValue, bigIntValue, floatValue, doubleValue, decimalValue)
VALUES
(2147483647, 9223372036854775807, 12345.12345, 1234567890.1234567, 1234567890.12),
(-2147483648, -9223372036854775808, -12345.12345, -1234567890.1234567, -1234567890.12);
```

• **INT vs. BIGINT:** **INT** can store values from -2,147,483,648 to 2,147,483,647. For numbers outside this range, **BIGINT** is used, which can store values from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

• **FLOAT vs. DOUBLE:** Both store floating-point numbers, but **FLOAT** is less precise than **DOUBLE**. For instance, when storing a number with many digits after the decimal point, **FLOAT** might round off sooner than **DOUBLE**.

• **DECIMAL:** This type is used for numbers where precision is crucial, such as financial transactions. Unlike **FLOAT** and **DOUBLE**, **DECIMAL** does not lose precision after the decimal point.

Create the Table

We'll start by creating a table named **products** with columns for each data type:

```
CREATE TABLE products (  
  id INT AUTO_INCREMENT PRIMARY KEY,  
  name VARCHAR(100),  
  description TEXT,  
  image BLOB,  
  price DECIMAL(10, 2),  
  weight DOUBLE,  
  category CHAR(10)  
);
```

In this table:

- **id** is an **INT**, suitable for auto-incrementing unique identifiers.
- **name** uses **VARCHAR(100)**, allowing variable-length names up to 100 characters.
- **description** is a **TEXT** type, ideal for long textual descriptions.
- **image** is stored as a **BLOB**, perfect for binary data like images.
- **price** uses **DECIMAL(10, 2)** for precise monetary values, supporting 10 digits with 2 decimal places.
- **weight** is a **DOUBLE** for more precise numeric data, like weights that can be fractional.
- **category** is a **CHAR(10)**, useful for fixed-length category codes.

```
INSERT INTO products (name, description, price, weight, category) VALUES  
( 'Widget', 'A small tool useful for various purposes.', 9.99, 0.25, 'TOOL'),  
( 'Gadget', 'An electronic device with practical uses.', 15.50, 0.50, 'ELEC');
```

• **VARCHAR vs. CHAR:**

- **name** as **VARCHAR** will store the string as is, without padding.
- **category** as **CHAR(10)** will pad the value 'TOOL' with spaces until it's 10 characters long.

• **TEXT:**

- **description** can store a long text about the product. It's much larger than you'd typically need for a name, allowing for extensive descriptions.

• **BLOB:**

- While not demonstrated with a direct insert here, **image** could store binary data like an image file. This demonstrates its capacity for large binary data storage.

Create the Table

First, we create the **EventLog** table with columns for each of the date and time types:

```
CREATE TABLE EventLog (  
    ID INT AUTO_INCREMENT PRIMARY KEY,  
    EventName VARCHAR(100),  
    EventDate DATE,  
    EventTime TIME,  
    EventDateTime DATETIME,  
    EventTimestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,  
    EventYear YEAR  
);
```

```
INSERT INTO EventLog (EventName, EventDate, EventTime, EventDateTime, EventYear)  
VALUES ('Conference', '2024-03-15', '14:30:00', '2024-03-15 14:30:00', '2024');
```

ID	EventName	EventDate	EventTime	EventDateTime	EventTimestamp	EventYear
1	Conference	2024-03-15	14:30:00	2024-03-15 14:30:00	2024-02-24 10:00:00	2024



Table: Customers

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

SELECT first_name, last_name
FROM Customers;

first_name	last_name
John	Doe
Robert	Luna
David	Robinson
John	Reinhardt
Betty	Doe

Table: Customers

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

SELECT *
FROM Customers;

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

The **SELECT** statement is used to select data from a database.

Syntax:

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

```
SELECT CustomerName, City FROM Customers;
```

Table: Customers

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

SELECT first_name, last_name
FROM Customers;

first_name	last_name
John	Doe
Robert	Luna
David	Robinson
John	Reinhardt
Betty	Doe

SQL SELECT ALL

To select all columns from a database table, we use the ***** character. For example,

Table: Customers

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

SELECT *
FROM Customers;

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

SQL SELECT WHERE Clause

A SELECT statement can have an optional WHERE clause. The WHERE clause allows us to fetch records from a database table that matches **specified condition(s)**. For example,

```
-- select all columns from the customers table with last_name 'Doe'  
SELECT *  
FROM Customers  
WHERE last_name = 'Doe';
```

Here, the SQL command selects all customers from the Customers table with the last_name **Doe**.

Here, the SQL command selects all customers from the Customers table with the last_name **Doe**.

Table: Customers

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

SELECT *
FROM Customers
WHERE last_name = 'Doe';

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
5	Betty	Doe	28	UAE

Another example.

-- select age and country columns from customers table where the country is 'USA'

SELECT age, country

FROM Customers

WHERE country = 'USA';

Note: In SQL, we must enclose textual data inside either single or double quotations like 'USA'.

SQL command selects the age and country columns of all the customers whose country is **USA**.

Table: Customers

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

SELECT age, country
FROM Customers
WHERE country = 'USA';

age	country
31	USA
22	USA

SQL Operators

The WHERE clause uses operators to construct conditions. Some of the commonly used operators are:

1. Equal to Operator (=)

-- select all columns from Customers table with first name 'John'

```
SELECT *
```

```
FROM Customers
```

```
WHERE first_name = 'John';
```

selects all the customers from the Customers table having first_name **John**.

2. Greater than (>)

-- select all columns from Customers table with age greater than 25

```
SELECT *
```

```
FROM Customers
```

```
WHERE age > 25;
```

selects all the customers from the Customers table whose age is **greater than 25**.

3. AND Operator (AND)

```
-- select all columns from Customers table with last_name 'Doe' and country 'USA'  
SELECT *  
FROM Customers  
WHERE last_name = 'Doe' AND country = 'USA';
```

selects all the customers from the Customers table having last_name **Doe** and country **USA**.

Note: If none of the rows meet the WHERE clause condition, an empty result set is returned.

selects first_name and of last_name all customers whose country is **USA** and last_name is **Doe** from the Customers table.

Table: Customers

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

-- select the first_name and last_name of all customers
-- who live in 'USA' and have the last name 'Doe'

```
SELECT first_name, last_name  
FROM Customers  
WHERE country = 'USA' AND last_name = 'Doe';
```

SELECT first_name, last_name
FROM Customers
WHERE country = 'USA' AND last_name = 'Doe';

first_name	last_name
John	Doe

SQL OR Operator

The SQL OR operator selects data if any one condition is TRUE. For example,

selects first_name and last_name of all customers where the country is **USA** or if their last name is **Doe** from the Customers table.

```
-- select first and last name of customers  
-- who either live in the USA  
-- or have the last name 'Doe'
```

```
SELECT first_name, last_name  
FROM Customers  
WHERE country = 'USA' OR last_name = 'Doe';
```

Table: Customers

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

```
SELECT first_name, last_name  
FROM Customers  
WHERE country = 'USA' OR last_name = 'Doe';
```

first_name	last_name
John	Doe
Robert	Luna
Betty	Doe

SQL NOT Operator

The SQL NOT operator selects data if the given condition is FALSE. For example, selects first_name and last_name of all customers where the country is not **USA** from the Customers table.

```
-- select customers who don't live in the USA
```

```
SELECT first_name, last_name  
FROM Customers  
WHERE NOT country = 'USA';
```

Table: Customers

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

```
SELECT first_name, last_name  
FROM Customers  
WHERE NOT country = 'USA';
```

first_name	last_name
David	Robinson
John	Reinhardt
Betty	Doe

Combining Multiple Operators

It is also possible to combine multiple AND, OR and NOT operators in an SQL statement. For example, let's suppose we want to select customers where country is either **USA** or **UK**, and age is **less than 26**.

-- select customers who live in either USA or UK and whose age is less than 26

```
SELECT *  
FROM Customers  
WHERE (country = 'USA' OR country = 'UK') AND age < 26;
```

Table: Customers

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

```
SELECT *  
FROM Customers  
WHERE (country = 'USA' OR country = 'UK') AND age < 26;
```

customer_id	first_name	last_name	age	country
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK

selects all customers where the country is not **USA** and last_name is not **Doe** from the Customers table.

-- exclude customers who are from the USA and have 'Doe' as their last name

```
SELECT *  
FROM customers  
WHERE NOT country = 'USA' AND NOT last_name = 'Doe';
```

Table: Customers

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

```
SELECT *  
FROM Customers  
WHERE NOT country = 'USA' AND NOT last_name = 'Doe');
```

customer_id	first_name	last_name	age	country
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK

CONTINUE



The SELECT DISTINCT statement **retrieves distinct** values from a database table.

-- select the unique ages from the Customers table

```
SELECT DISTINCT age  
FROM Customers;
```

Here, the SQL command selects only the unique values of age from the Customers table.

Syntax of SQL SELECT DISTINCT

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

Here,

- column1, column2, ... are the table columns
- table is table name from where we retrieve the distinct columns

selects unique countries from the Customers table.

-- select the unique countries from the customers table

```
SELECT DISTINCT country  
FROM Customers;
```

Table: Customers

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

SELECT DISTINCT country
FROM Customers;

country
USA
UK
UAE

We can also use SELECT DISTINCT with multiple columns. For example,

Selects rows if combinations of country and first_name are unique. Meaning, the result will include each pair of country and first_name only once.

-- select rows if the first name and country of a customer is unique

```
SELECT DISTINCT country, first_name  
FROM Customers;
```

Customers

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

SELECT DISTINCT country, first_name
FROM Customers;

country	first_name
USA	John
USA	Robert
UK	David
UK	John
UAE	Betty

We can use SQL DISTINCT with the [COUNT\(\)](#) function to count the number of unique rows.

```
-- count the unique countries where customers are from  
SELECT COUNT(DISTINCT country)  
FROM Customers;
```

Table: Customers

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

SQL command returns the count of unique countries.

SELECT COUNT(DISTINCT country)
FROM Customers;

COUNT(DISTINCT country)

3

The AS keyword is used to give columns or tables a temporary name that can be used to identify that column or table later.

Here, the SQL command selects the first_name column from the Customers table. However, the column name is changed to name in the result set.

```
SELECT first_name AS name  
FROM Customers;
```

SQL AS Alias Syntax

The syntax of the SQL AS command is:

```
SELECT column_1 AS alias_1,  
       column_2 AS alias_2,  
... .. column_n AS alias_n  
FROM table_name;
```

Here,

- column_1, column_2,...column_n are the table columns
- alias_1, alias_2,...alias_n are the aliases of the table columns

selects the first_name column of Customers.
However, the column name will change to name in the result set.

```
SELECT first_name AS name  
FROM Customers;
```

Table: Customers

customer_id	first_name	last_name	age	country
1	John	Doe	31	USA
2	Robert	Luna	22	USA
3	David	Robinson	22	UK
4	John	Reinhardt	25	UK
5	Betty	Doe	28	UAE

```
SELECT first_name AS name  
FROM Customers;
```

name
John
Robert
David
John
Betty

We can also use aliases with more than one column.

selects **customer_id as cid** and **first_name as name**.

```
SELECT customer_id AS cid, first_name AS name  
FROM Customers;
```

customer_id	first_name
1	John
2	Jane
3	Doe

cid	name
1	John
2	Jane
3	Doe

We can combine data from multiple columns and represent it in a single column using the CONCAT() function. For example,

```
SELECT CONCAT(first_name, ' ', last_name) AS full_name FROM Customers;
```

Customers

first_name	last_name
John	Doe
Jane	Smith

full_name
John Doe
Jane Smith

SQL BETWEEN Syntax

```
SELECT column1, column2, ...  
FROM table  
WHERE column BETWEEN value1 AND value2;
```

- column1, column2, ... are the columns you want to filter
- table is the name of the table
- column is the name of the column where we want to specify a range of values
- BETWEEN is an operator used to specify a range of values for the column
- value1 and value2 are the lower and upper bounds of the range

select item and amount columns from Orders that have amounts between **300** and **500** (including **300** and **500**).

```
SELECT item, amount  
FROM Orders  
WHERE amount BETWEEN 300 AND 500;
```

Table: Orders

order_id	item	amount	customer_id
1	Keyboard	400	4
2	Mouse	300	4
3	Monitor	12000	3
4	Keyboard	400	1
5	Mousepad	250	2

```
SELECT item, amount  
FROM Orders  
WHERE amount BETWEEN 300 AND 500;
```

item	amount
Keyboard	400
Mouse	300
Keyboard	400

In SQL, the BETWEEN operator with the WHERE clause selects values within a given range.

Here, the SQL command selects all columns from the Orders table where the amount is between **200** and **600** (including **200** and **600**).

```
-- select rows where the amount is between 200 and 600
```

```
SELECT *  
FROM Orders  
WHERE amount BETWEEN 200 AND 600;
```

Orders

OrderID	CustomerID	Amount	OrderDate
1	1001	150	2023-01-01
2	1002	250	2023-01-02
3	1003	600	2023-01-03
4	1004	700	2023-01-04

OrderID	CustomerID	Amount	OrderDate
2	1002	250	2023-01-02
3	1003	600	2023-01-03

The NOT BETWEEN operator is used to exclude the rows that match the values in the range. It returns all the rows except the excluded rows.

Select all the items from Orders except the rows that have amounts between 300 and 500 (300 and 500 are also excluded).

-- exclude rows with amount between 300 and 500

```
SELECT item, amount  
FROM Orders  
WHERE amount NOT BETWEEN 300 AND 500;
```

Table: Orders

order_id	item	amount	customer_id
1	Keyboard	400	4
2	Mouse	300	4
3	Monitor	12000	3
4	Keyboard	400	1
5	Mousepad	250	2

```
SELECT item, amount  
FROM Orders  
WHERE amount NOT BETWEEN 300 AND 500;
```

item	amount
Monitor	12000
Mousepad	250

The BETWEEN operator also works with text data.

selects all orders where the item names begin with letters between I and L.

-- select rows where items begin with letters between 'I' and 'L'
-- exclude all items beginning with 'L' followed by other characters

```
SELECT item, amount  
FROM Orders  
WHERE item BETWEEN 'I' AND 'L';
```

Table: Orders

order_id	item	amount	customer_id
1	Keyboard	400	4
2	Mouse	300	4
3	Monitor	12000	3
4	Keyboard	400	1
5	Mousepad	250	2

SELECT item, amount
FROM Orders
WHERE item BETWEEN 'I' AND 'L';

item	amount
Keyboard	400
Keyboard	400

In SQL, we can also use BETWEEN to filter data between two dates.

selected the teams who registered between 2021-01-01 and 2022-11-01.

```
-- get the records of those teams  
-- who registered between given dates
```

```
SELECT *  
FROM Teams  
WHERE registered BETWEEN '2021-01-01' AND '2022-11-01';
```

In SQL, IS NULL and IS NOT NULL are used to check if a column in a table contains a NULL value or not.

IS NULL Syntax

In SQL, the IS NULL condition is used to select rows if the specified field is NULL. It has the following syntax:

```
SELECT column1, column2, ...  
FROM table  
WHERE column_name IS NULL;
```

Here,

column1, column2, ... are the table columns

table is the table name from where we select the data

column_name is the name of the column you want to check for NULL

-- select rows with **NULL email values**

```
SELECT *  
FROM Employee  
WHERE email IS NULL;
```

Table: Employee

employee_id	first_name	last_name	department	email
1	Peter	Doe	Operations	peter@gmail.com
2	Megan	Morel	Finance	NULL
3	Rose	Bailey	Operations	rose@gmail.com
4	Linda	Bailey	Finance	NULL
5	Mary	Doe	Sales	NULL

SELECT *
FROM Employee
WHERE email IS NULL;

employee_id	first_name	last_name	department	email
2	Megan	Morel	Finance	NULL
4	Linda	Bailey	Finance	NULL
5	Mary	Doe	Sales	NULL

In SQL, the IS NOT NULL condition is used to select rows if the specified field is NOT NULL. It has the following syntax:

```
SELECT column1, column2, ...  
FROM table  
WHERE column_name IS NOT NULL;
```

Here,

column1, column2, ... are the table columns

table is the table name from where we select the data

column_name is the name of the column you want to check for NOT NULL

-- select rows where email is not NULL

```
SELECT *  
FROM Employee  
WHERE email IS NOT NULL;
```

Table: Employee

employee_id	first_name	last_name	department	email
1	Peter	Doe	Operations	peter@gmail.com
2	Megan	Morel	Finance	NULL
3	Rose	Bailey	Operations	rose@gmail.com
4	Linda	Bailey	Finance	NULL
5	Mary	Doe	Sales	NULL

SELECT *
FROM Employee
WHERE email IS NOT NULL;

employee_id	first_name	last_name	department	email
1	Peter	Doe	Operations	peter@gmail.com
3	Rose	Bailey	Operations	rose@gmail.com

We can use the COUNT() function with IS NULL to count the number of rows with an empty field. For example,

```
SELECT COUNT(*)  
FROM Employee  
WHERE email IS NULL;
```

Table: Employee

employee_id	first_name	last_name	department	email
1	Peter	Doe	Operations	peter@gmail.com
2	Megan	Morel	Finance	NULL
3	Rose	Bailey	Operations	rose@gmail.com
4	Linda	Bailey	Finance	NULL
5	Mary	Doe	Sales	NULL

SELECT COUNT(*)
FROM Employee
WHERE email IS NULL;

COUNT(*)

3



مجموعة جامعة
الجميع الذكية
للتعليم القابضة

MIDOCEAN
UNIVERSITY



Thank You

