

Database Management System

IT 261

Bachelor of Information Technology

Prepared by Dr. Khaled Eskaf

Lecture 5





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.
	Relational Algebra and	Explore relational algebra and basic	
4	Basic SQL	SQL commands.	Use basic SQL queries in MySQL.
5	Advanced SQL Queries	Delve into complex SQL operations.	Execute advanced SQL commands in MySQL.
	Introduction		
6	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

3	Rules and constraints.	constraints in MySQL.	
4	Relational Algebra and Basic SQL	Explore relational algebra and basic SQL commands.	Use basic SQL queries in MySQL.
	Advanced		Execute advanced SOL commands

Every database management system must define a query language to allow users to access the data stored in the database. **Relational Algebra** is a procedural query language used to query the database tables to access data in different ways.

In relational algebra, input is a relation(table from which data has to be accessed) and output is also a relation(a temporary table holding the data asked for by the user).

We can use Relational Algebra to fetch data from this Table(relation)

Select Name students with age less than 17

Output

Name
Ckon
Dkon

ID	Name	Age
1	Akon	17
2	Bkon	19
3	Ckon	15
4	Dkon	13

The output for query is also in form of a table(relation), with results in different columns

Relational Algebra works on the whole table at once, so we do not have to use loops etc to iterate over all the rows(tuples) of data one by one. All we have to do is specify the table name from which we need the data, and in a single line of command, relational algebra will traverse the entire given table to fetch data for you.

The primary operations that we can perform using relational algebra are:

- 1.Select
- 2.Project
- 3.Union
- 4.Set Different
- 5.Cartesian product
- 6.Rename

Select Operation (σ):

This is used to fetch rows(tuples) from table(relation) which satisfies a given condition.

Syntax: $\sigma_p(r)$

Where, σ represents the Select Predicate, r is the name of relation(table name in which you want to look for data), and p is the propositional logic, where we specify the conditions that must be satisfied by the data. In propositional logic, one can use **unary** and **binary** operators like $=$, $<$, $>$ etc, to specify the conditions.

Let's take an example of the Student table we specified above in the Introduction of relational algebra, and fetch data for **students** with **age** more than 17.

$\sigma_{age > 17}$ (Student)

This will fetch the tuples(rows) from table **Student**, for which **age** will be greater than 17.

You can also use, **and**, **or** etc operators, to specify two conditions, for example,

$\sigma_{age > 17 \text{ and } gender = 'Male'}$ (Student)

This will return tuples(rows) from table **Student** with information of male students, of age more than 17.(Consider the Student table has an attribute **Gender** too.)

A	B	C
1	2	4
2	2	3
3	2	3
4	3	4

Example:

For the above relation, $\sigma(c>3)R$ will select the tuples which have c more than 3.

Note: The selection operator only selects the required tuples but does not display them.
For display, the data projection operator is used.

A	B	C
1	2	4
4	3	4

Project Operation (Π):

Project operation is used to project only a certain set of attributes of a relation. In simple words, If you want to see only the **names** all of the students in the **Student** table, then you can use Project Operation.

It will only project or show the columns or attributes asked for, and will also remove duplicate data from the columns.

Syntax: $\Pi_{A_1, A_2 \dots}(r)$

where A1, A2 etc are attribute names(column names).

For example,

$\Pi_{\text{Name, Age}}(\text{Student})$

Above statement will show us only the **Name** and **Age** columns for all the rows of data in **Student** table.

Projection(π):

It is used to project required column data from a relation.

Example: Consider Table 1. Suppose we want columns B and C from Relation R.
 $\pi(B,C)R$ will show following columns.

Note: By Default, projection removes duplicate data.

B	C
2	4
2	3
3	4

Union Operation (U):

This operation is used to fetch data from two relations (tables) or temporary relation (result of another operation). For this operation to work, the relations (tables) specified should have same number of attributes (columns) and same attribute domain. Also the duplicate tuples are automatically eliminated from the result.

Syntax: $A \cup B$

where A and B are relations.

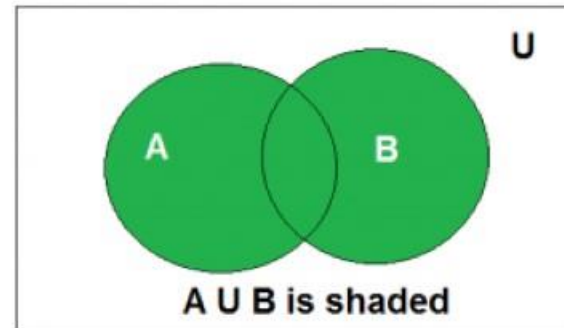
For example, if we have two tables **RegularClass** and **ExtraClass**, both have a column **student** to save name of student, then,

$\Pi_{\text{Student}}(\text{RegularClass}) \cup \Pi_{\text{Student}}(\text{ExtraClass})$

Above operation will give us name of **Students** who are attending both regular classes and extra classes, eliminating repetition.

Union

Union of the sets A and B, denoted by $A \cup B$, is the set of distinct elements that belong to set A or set B, or both.



Venn diagram of $A \cup B$

Above is the Venn Diagram of $A \cup B$.

Example: Find the union of $A = \{2, 3, 4\}$ and $B = \{3, 4, 5\}$; Solution : $A \cup B = \{2, 3, 4, 5\}$.

Union(U): Union operation in relational algebra is the same as union operation in set theory.

Example:

Student_Name	Roll_Number
Ram	01
Mohan	02
Vivek	13
Geeta	17

GERMAN	
Student_Name	Roll_Number
Vivek	13
Geeta	17
Shyam	21
Rohan	25

Consider the following table of Students having different optional subjects in their course.

$$\pi(\text{Student_Name})\text{FRENCH} \cup \pi(\text{Student_Name})\text{GERMAN}$$

Consider the following table of Students having different optional subjects in their course.

Student_Name
Ram
Mohan
Vivek
Geeta
Shyam
Rohan

Note: The only constraint in the union of two relations is that both relations must have the same set of Attributes.

Set Difference (-) :

This operation is used to find data present in one relation and not present in the second relation. This operation is also applicable on two relations, just like Union operation.

Syntax: $A - B$

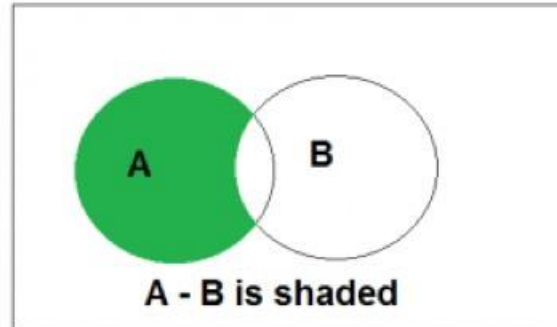
where A and B are relations.

For example, if we want to find name of students who attend the regular class but not the extra class, then, we can use the below operation:

$$\Pi_{\text{Student}}(\text{RegularClass}) - \Pi_{\text{Student}}(\text{ExtraClass})$$

Set Difference

The difference between sets is denoted by ' $A - B$ ', which is the set containing elements that are in A but not in B. i.e., all elements of A except the element of B.



Example: If $A = \{1, 2, 3, 4, 5\}$ and $B = \{2, 4, 6, 8\}$, find $A - B$ Solution: $A - B = \{1, 3, 5\}$

Set Difference(-): Set Difference in relational algebra is the same set difference operation as in set theory.

Example: From the above table of FRENCH and GERMAN, Set Difference is used as follows

$\pi(\text{Student_Name})\text{FRENCH} - \pi(\text{Student_Name})\text{GERMAN}$

Student_Name
Ram
Mohan

Note: The only constraint in the Set Difference between two relations is that both relations must have the same set of Attributes.

Set Intersection($\cap$): Set Intersection in relational algebra is the same set intersection operation in set theory.

Example: From the above table of FRENCH and GERMAN, the Set Intersection is used as follows

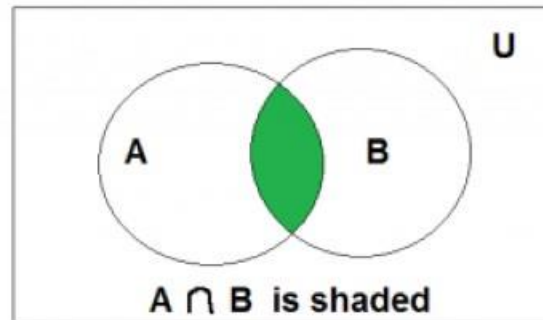
$\pi(\text{Student_Name})\text{FRENCH} \cap \pi(\text{Student_Name})\text{GERMAN}$

Student_Name
Vivek
Geeta

Note: The only constraint in the Set Difference between two relations is that both relations must have the same set of Attributes.

Intersection

The intersection of the sets A and B , denoted by $A \cap B$, is the set of elements that belong to both A and B i.e. set of the common elements in A and B .



Example: Find the intersection of $A = \{2, 3, 4\}$ and $B = \{3, 4, 5\}$ Solution : $A \cap B = \{3, 4\}$.

Rename Operation (ρ):

This operation is used to rename the output relation for any query operation which returns result like Select, Project etc. Or to simply rename a relation(table)

Syntax: $\rho(\text{RelationNew}, \text{RelationOld})$

Rename(ρ): Rename is a unary operation used for renaming attributes of a relation.

$\rho(a/b)R$ will rename the attribute 'b' of the relation by 'a'.

Cartesian Product (X) - Cross Product(X):

This is used to combine data from two different relations(tables) into one and fetch data from the combined relation.

Syntax: $A \times B$

For example, if we want to find the information for Regular Class and Extra Class which are conducted during morning, then, we can use the following operation:

$\sigma_{\text{time} = \text{'morning'}} (\text{RegularClass} \times \text{ExtraClass})$

For the above query to work, both **RegularClass** and **ExtraClass** should have the attribute **time**.

Cross Product(X): Cross-product between two relations. Let's say A and B, so the cross product between A X B will result in all the attributes of A followed by each attribute of B. Each record of A will pair with every record of B.

Example:

A

Name	Age	Sex
Ram	14	M
Sona	15	F
Kim	20	M

B

ID	Course
1	DS
2	DBMS

A X B

Name	Age	Sex	ID	Course
Ram	14	M	1	DS
Ram	14	M	2	DBMS
Sona	15	F	1	DS
Sona	15	F	2	DBMS
Kim	20	M	1	DS
Kim	20	M	2	DBMS

Note: If A has 'n' tuples and B has 'm' tuples then A X B will have 'n*m' tuples.

	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 ⋈ Boat			
CarModel	CarPrice	BoatModel	BoatPrice	CarPrice ≥ BoatPrice			
CarA	20,000	Boat1	10,000	CarModel	CarPrice	BoatModel	BoatPrice
CarB	30,000	Boat2	40,000	CarA	20,000	Boat1	10,000
CarC	50,000	Boat3	60,000	CarB	30,000	Boat1	10,000
				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

Natural Join($\bowtie$): Natural join is a binary operator. Natural join between two or more relations will result in a set of all combinations of tuples where they have an equal common attribute.

Example:

EMP

Name	ID	Dept_Name
A	120	IT
B	125	HR
C	110	Sales
D	111	IT

DEPT

Dept_Name	Manager
Sales	Y
Production	Z
IT	A

Natural join between EMP and DEPT with condition :
EMP.Dept_Name = DEPT.Dept_Name

Natural join between EMP and DEPT with condition :
EMP.Dept_Name = DEPT.Dept_Name

EMP ⋈ DEPT

Name	ID	Dept_Name	Manager
A	120	IT	A
C	110	Sales	Y
D	111	IT	A

EMP

Name	ID	Dept_Name
A	120	IT
B	125	HR
C	110	Sales
D	111	IT

DEPT

Dept_Name	Manager
Sales	Y
Production	Z
IT	A

Conditional Join: Conditional join works similarly to natural join. In natural join, by default condition is equal between common attributes while in conditional join we can specify any condition such as greater than, less than, or not equal.

Example:

R

ID	Sex	Marks
1	F	45
2	F	55
3	F	60

S

ID	Sex	Marks
10	M	20
11	M	22
12	M	59

Join between R and S with condition **R.marks >= S.marks**

Join between R and S with condition $R.marks \geq S.marks$

R.ID	R.Sex	R.Marks	S.ID	S.Sex	S.Marks
1	F	45	10	M	20
1	F	45	11	M	22
2	F	55	10	M	20
2	F	55	11	M	22
3	F	60	10	M	20
3	F	60	11	M	22
3	F	60	12	M	59

R

ID	Sex	Marks
1	F	45
2	F	55
3	F	60

S

ID	Sex	Marks
10	M	20
11	M	22
12	M	59

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 first_name, last_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, 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;
```

age < 26 age less than 26.

age > 26 age greater than 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



Thank You