`MySQL`

Table of contents

Loading [MathJax]/extensions/Safe.js

Below topics are not covered due to limited time. Apologies for inconvenience.

> Soon uploading in provided link
>
> ---

> Soon uploading in provided link

---

1. INTRODUCTION

Hey !

In this file, I will guide you on an extensive journey exploring SQL databases from the very basics to advanced concepts. Brace yourself for a challenging but fulfilling adventure, as I am confident in your ability to conquer it **:)**

For assistance, I would be delighted to answer any questions you may have. Feel free to reach out to me on my Twitter account, @_TurkiSultan.

GOOD LUCK

---

2. INTRODUCTION TO DATABASE

Introductio to Database

Introduction

We all use data and databases in our daily online lives. For example, uploading photos to our social media feeds, downloading files at work, and playing games online, are all examples of database usage

Databases and Data

What is Data?

In basic terms, data is facts and figures about anything. For example, if data were collected on a person, then that data might include their name, age, email, and date of birth. Or the data could be facts and figures related to an online purchase. This could be the order number, description, order quantity, and date, and even the customer's email.

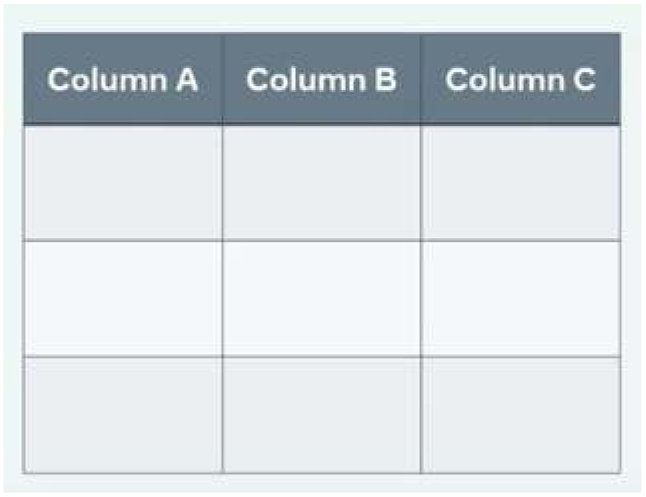
Introduction to SQL

SQL: Standard language that can interact with structured data on database

SQL subsets:

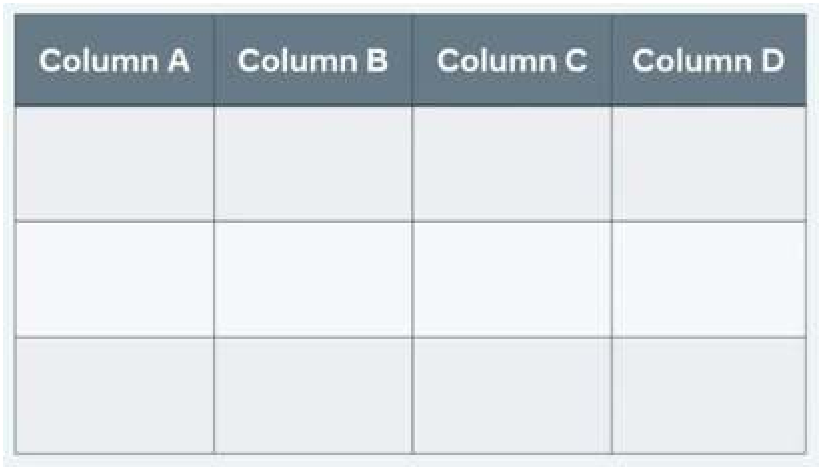
1. Data definitionlanguage (DDL)
2. Data manipulation language (DML)
3. Data query language (DQL)
4. Data control language (DCL)

DDL Create command: Used to create storage objects in a database, like tables



Column A	Column B	Column C

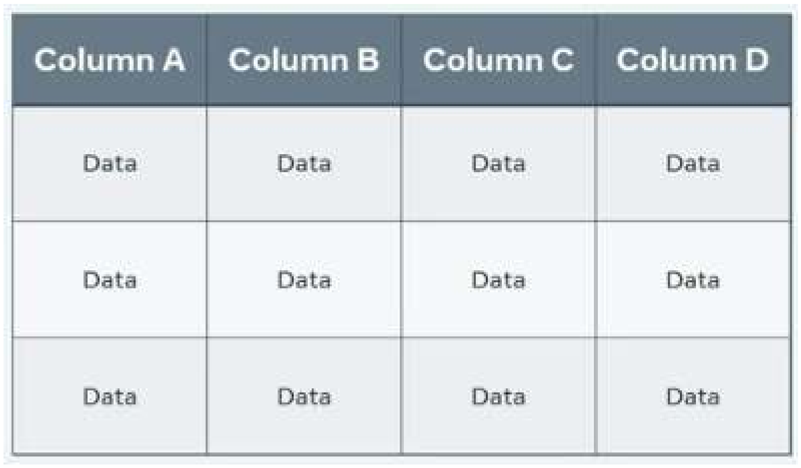
DDL Alter command: Modify the structure of a table object in a database



Column A	Column B	Column C	Column D

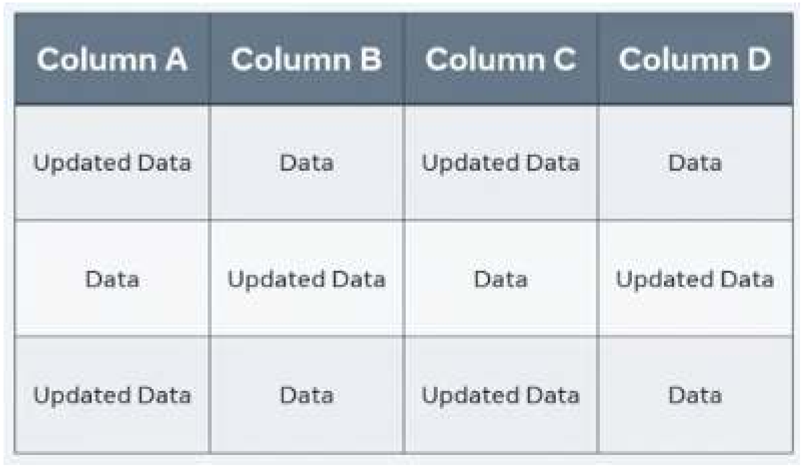
DDL Drop command: Remove an existing object from a database

DML Insert command: Insert records (Rows) of data into a database table



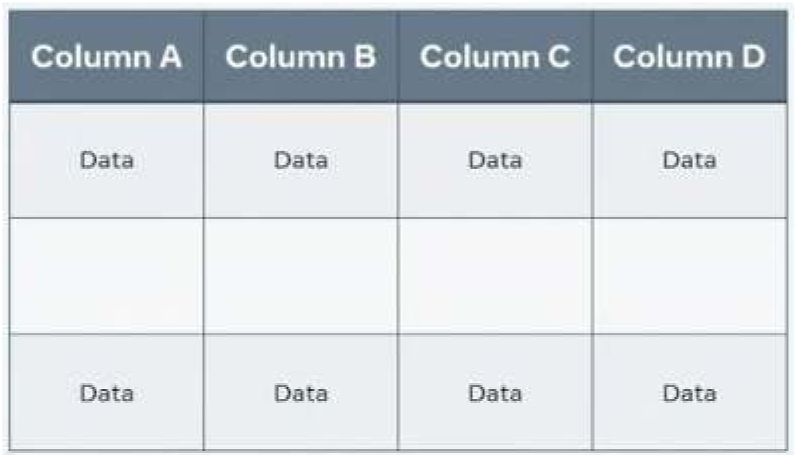
Column A	Column B	Column C	Column D
Data	Data	Data	Data
Data	Data	Data	Data
Data	Data	Data	Data

DML Update command: Edit data that already exists in a database table



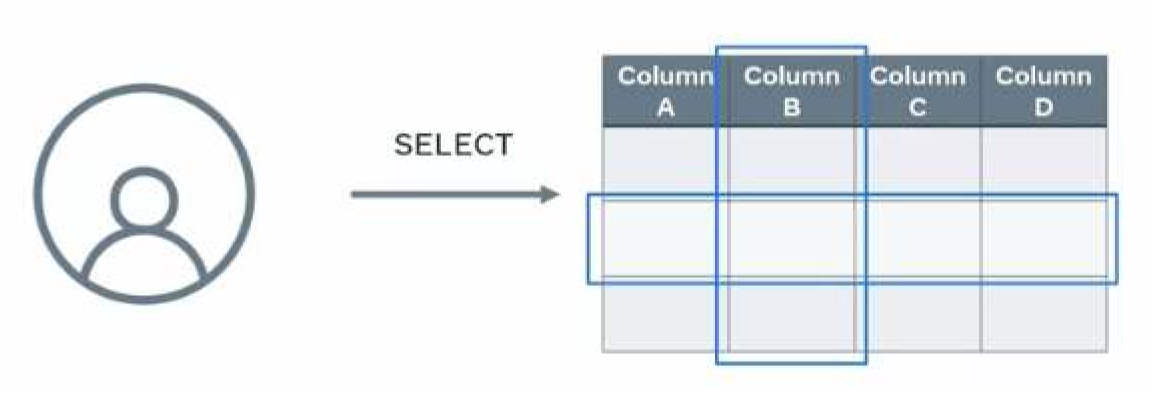
Column A	Column B	Column C	Column D
Updated Data	Data	Updated Data	Data
Data	Updated Data	Data	Updated Data
Updated Data	Data	Updated Data	Data

DML Delete command: Delete one or more rows of data from a table



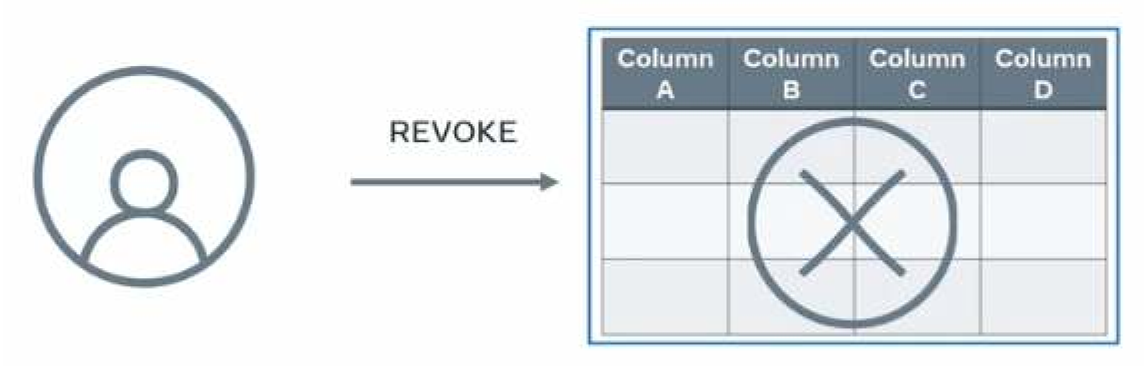
Column A	Column B	Column C	Column D
Data	Data	Data	Data
Data	Data	Data	Data

DQL Select command: Retrieve data from one or multiple tables letting you specify the data fields that you want based on preferred filter criteria



DCL commands: Using DCL commands, you control access to data stored in the database

Grant & Revoke DCL command: used to give users access privileges to data, and to revert access privileges already given to users



SQL syntax intro:

NOTE: SQL is case insensitive, which means SELECT and select have same meaning in SQL statements.

`Creat a database`

CREATE DATABASE database_name;

`Create a table`

CREATE TABLE table_name;

`Add data to a table`

INSERT INTO table_name (column_one, column_two, column_three, ...) VALUES (value1, value2, value3, ...);

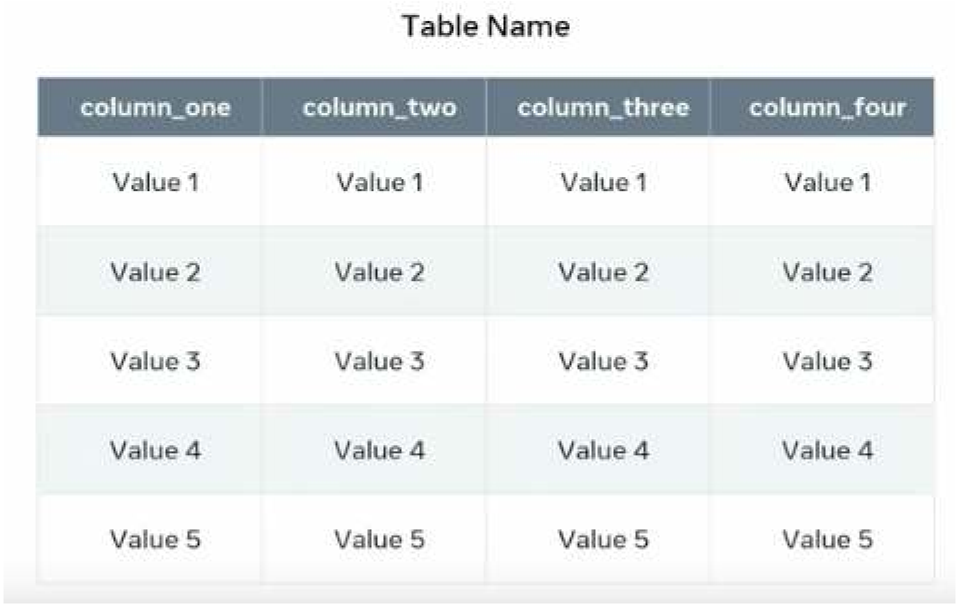
- ***INSERT INTO table_name***: Specifies the name of the table you want to insert data into. (column_one, column_two, column_three, ...) : Specifies the columns into which you want to insert data.
- ***VALUES (value1, value2, value3, ...)*** : Specifies the values you want to insert into the respective columns. 

  Table Name

column_one	column_two	column_three	column_four
Value 1	Value 1	Value 1	Value 1
Value 2	Value 2	Value 2	Value 2
Value 3	Value 3	Value 3	Value 3
Value 4	Value 4	Value 4	Value 4
Value 5	Value 5	Value 5	Value 5

  We will take it in detail soon

`Update data in a table`

UPDATE table_name SET column_name = 'new row name' WHERE column_name = 'old row name';

- ***UPDATE table_name***: Specifies the name of the table you want to update
- ***SET column_name*** = 'new row name': Specifies the column you want to update and the new value you want to assign to it.
- ***WHERE column_name*** = 'old row name': Optional condition that filters the rows to be updated. In this case, it restricts the update to rows where the column value matches the specified old row name.

`Delete data from a table`

DELETE FROM table_name;

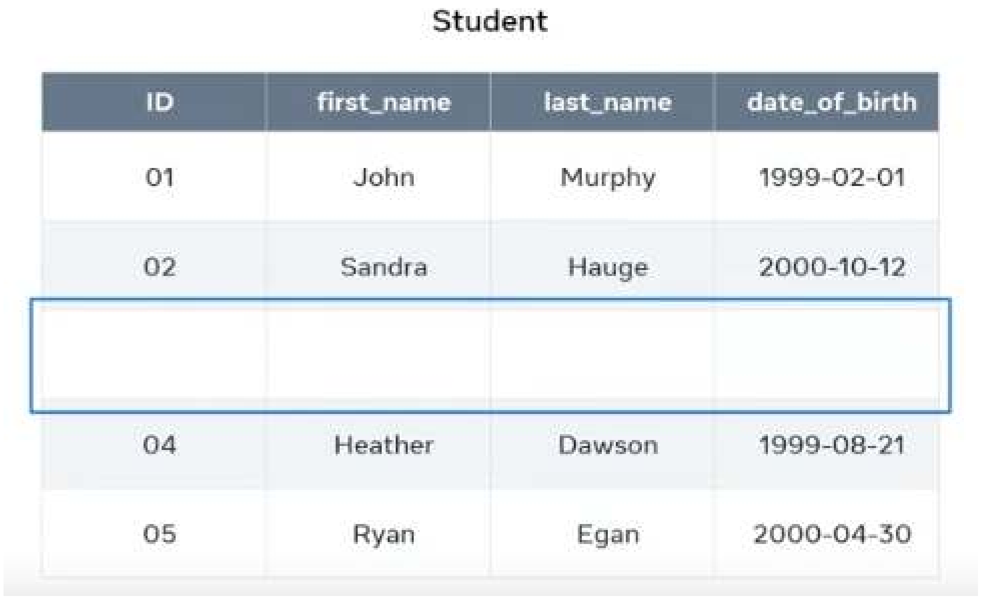
- ***DELETE FROM table_name***: Specifies the name of the table from which you want to delete all rows.

DELETE FROM table_name WHERE column_name = 'The_row' ;

- ***DELETE FROM table_name***: Specifies the name of the table from which you want to delete rows.
- ***WHERE column_name = 'The_row'***: Specifies the condition that determines which rows to delete. You need to replace column_name with the actual column name you want to use for the condition, and

Loading [MathJax]/extensions/Safe.js

'The_row' with the specific value or condition that identifies the rows you want to delete.



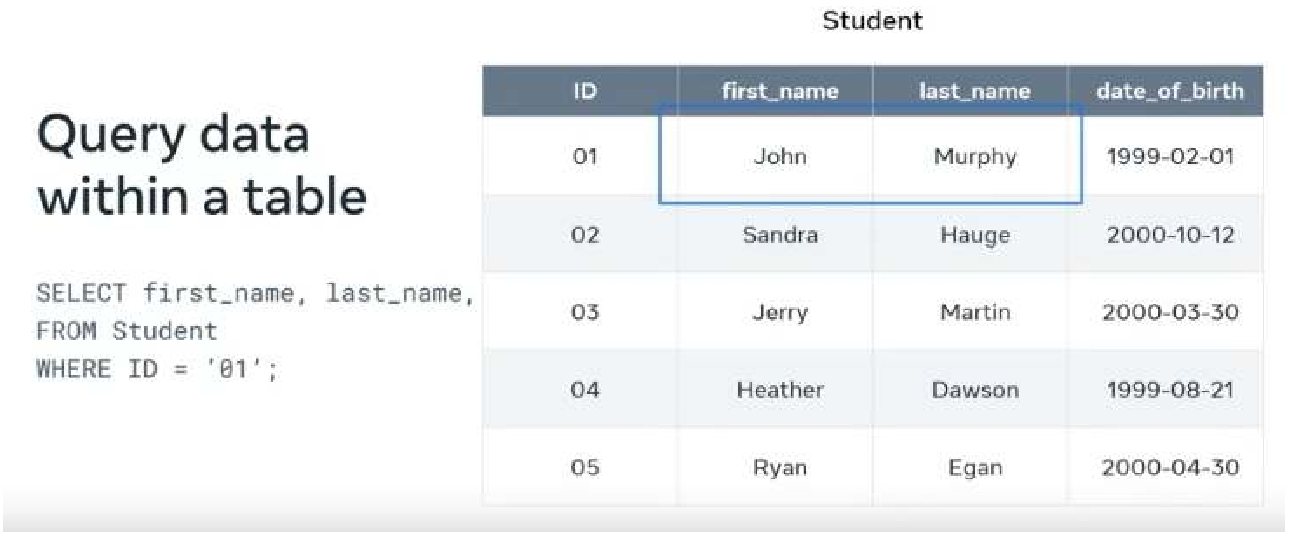
Student

ID	first_name	last_name	date_of_birth
01	John	Murphy	1999-02-01
02	Sandra	Hauge	2000-10-12
04	Heather	Dawson	1999-08-21
05	Ryan	Egan	2000-04-30

`Query data within a table`

SELECT column_one, column_two, ... FROM table_name WHERE column_name = 'The_row';

- ***SELECT column_one, column_two, ...***: Specifies the columns you want to retrieve data from. Replace column_one, column_two, etc. with the actual column names you want to select.
- ***FROM table_name***: Specifies the name of the table from which you want to retrieve data. Replace table_name with the actual name of the table you want to query.
- ***WHERE column_name = 'The_row'***: Specifies the condition that determines which rows to retrieve. Replace column_name with the actual column name you want to use for the condition, and 'The_row' with the specific value or condition that identifies the rows you want to select.



Query data within a table

```
SELECT first_name, last_name,
FROM Student
WHERE ID = '01';
```

Student

ID	first_name	last_name	date_of_birth
01	John	Murphy	1999-02-01
02	Sandra	Hauge	2000-10-12
03	Jerry	Martin	2000-03-30
04	Heather	Dawson	1999-08-21
05	Ryan	Egan	2000-04-30

Common SQL Commands

The main commands used in SQL. At a later stage you will explore relevant examples of how to use these commands with a detailed explanation of the SQL syntax for key operations such as to create, insert, update and delete data in the database.

The SQL Commands are grouped into four categories known as DDL, DML, DCL and TCL depending on

Loading [MathJax]/extensions/Safe.js y, namely the type of operation they're used to perform. Let's explore these commands in greater detail.

Data Definition Language (DDL)

The SQL DDL category provides commands for defining, deleting and modifying tables in a database. Use the following commands in this category.

CREATE Command

Purpose: To create the database or tables inside the database

Syntax to create a table with three columns:

`CREATE TABLE table_name (column_name1 datatype(size), column_name2 datatype(size), column_name3 datatype(size));`

DROP Command

Purpose: To delete a database or a table inside the database.

Syntax to drop a table:

`DROP TABLE table_name;`

ALTER Command

Purpose: To change the structure of the tables in the database such as changing the name of a table, adding a primary key to a table, or adding or deleting a column in a table.

1. Syntax to add a column into a table:

   `ALTER TABLE table_name ADD (column_name datatype(size));`

2. Syntax to add a primary key to a table:

   `ALTER TABLE table_name ADD primary key (column_name);`

TRUNCATE Command

Purpose: To remove all records from a table, which will empty the table but not delete the table itself.

Syntax to truncate a table:

`TRUNCATE TABLE table_name;`

COMMENT Command

Purpose: To add comments to explain or document SQL statements by using double dash (--) at the start of the line. Any text after the double dash will not be executed as part of the SQL statement. These comments are not there to build the database. They are only for your own use.

Syntax to COMMENT a line in SQL:

--Retrieve all data from a table `SELECT * FROM table_name;`

Data Manipulation Language (DML)

Loading [MathJax]/extensions/Safe.js

The SQL DML commands provide the ability to query, delete and update data in the database. Use the following commands in this category.

SELECT Command

Purpose: To retrieve data from tables in the database.

Syntax to select data from a table:

```
SELECT * FROM table_name;
```

INSERT Command

Purpose: To add records of data into an existing table. Syntax to insert data into three columns in a table:

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

UPDATE Command

Purpose: To modify or update data contained within a table in the database.

Syntax to update data in two columns:

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

DELETE Command

Purpose: To delete data from a table in the database.

Syntax to delete data:

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

Data Control Language (DCL)

You use DCL to deal with the rights and permissions of users of a database system. You can execute SQL commands to perform different types of operations such as create and drop tables. To do this, you need to have user rights set up. This is called user privileges. This category deals with advanced functions or operations in the database. Note that this category can have a generic description of the two main commands. Use the following commands in this category:

GRANT Command to provide the user of the database with the privileges required to allow users to access and manipulate the database.

REVOKE Command to remove permissions from any user.

Transaction Control Language (TCL)

The TCL commands are used to manage transactions in the database. These are used to manage the changes made to the data in a table by utilizing the DML commands. It also allows SQL statements to be grouped together into logical transactions. This category deals with advanced functions or operations in a database. Note that this category can have a generic description of the two main commands. Use the following commands in this category:

COMMIT Command to save all the work you have already done in the database.

ROLLBACK Command to restore a database to the last committed state.

Basic database structure

Table overview: The main objective of this reading is to examine tables in more depth in terms of the structure of a table, data types, primary and foreign keys and the role they play in a table, as well as table constraints.

To refresh your memory, a table is the most basic type of database object in relational databases. It is responsible for storing data in the database. Like any other table, a database table also consists of rows and columns.

Just like in a spreadsheet of data, there are rows that run horizontally. These rows represent each record. Rows in turn span multiple columns.

Columns run vertically. They are like the definition of each field. Each column has a name that describes the data that is stored in it. Examples of column names could include FirstName, LastName, ProductID, Price and so forth.

Where the row intersects with a column is where a cell is located. A cell is where you store an item of data.

What are data types?

Every column in a table has a data type. These data types are defined by SQL or Structured Query Language. A data type defines the type of value that can be stored in a table column.

For example, here are some of the data types that are available:

- Numeric data types such as INT, TINYINT, BIGINT, FLOAT, REAL.
- Date and time data types such as DATE, TIME, DATETIME.
- Character and string data types such as CHAR, VARCHAR.
- Binary data types such as BINARY, VARBINARY.
- Miscellaneous data types such as:
- Character Large Object (CLOB) for storing a large block of text in some form of text encoding.
- Binary Large Object (BLOB) for storing a collection of binary data such as images.

Here's an example of a table. This is the student table that stores data about a student such as:

- student ID,
- first name,
- last name,
- ,

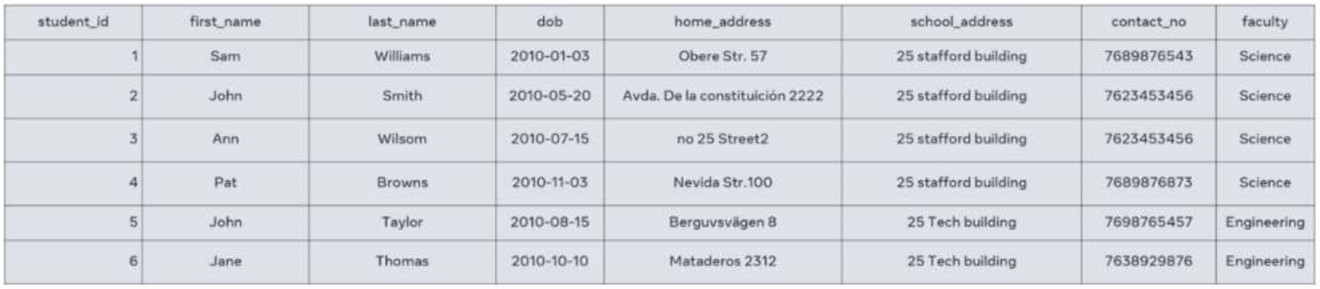
Loading [MathJax]/extensions/Safe.js

- home address,
- and faculty.

These are the table's columns.

There are also six rows within this table; one for each student. In other words, the table contains the records of six students.

Each cell in a row or record contains a piece of data such as student ID = 1, first name = Emily, last name = Williams and so on. Database table that contains different data types



student_id	first_name	last_name	dob	home_address	school_address	contact_no	faculty
1	Sam	Williams	2010-01-03	Obere Str. 57	25 stafford building	7689876543	Science
2	John	Smith	2010-05-20	Avda. De la constituición 2222	25 stafford building	7623453456	Science
3	Ann	Wilsom	2010-07-15	no 25 Street2	25 stafford building	7623453456	Science
4	Pat	Browns	2010-11-03	Nevida Str.100	25 stafford building	7689876873	Science
5	John	Taylor	2010-08-15	Berguvsvägen 8	25 Tech building	7698765457	Engineering
6	Jane	Thomas	2010-10-10	Mataderos 2312	25 Tech building	7638929876	Engineering

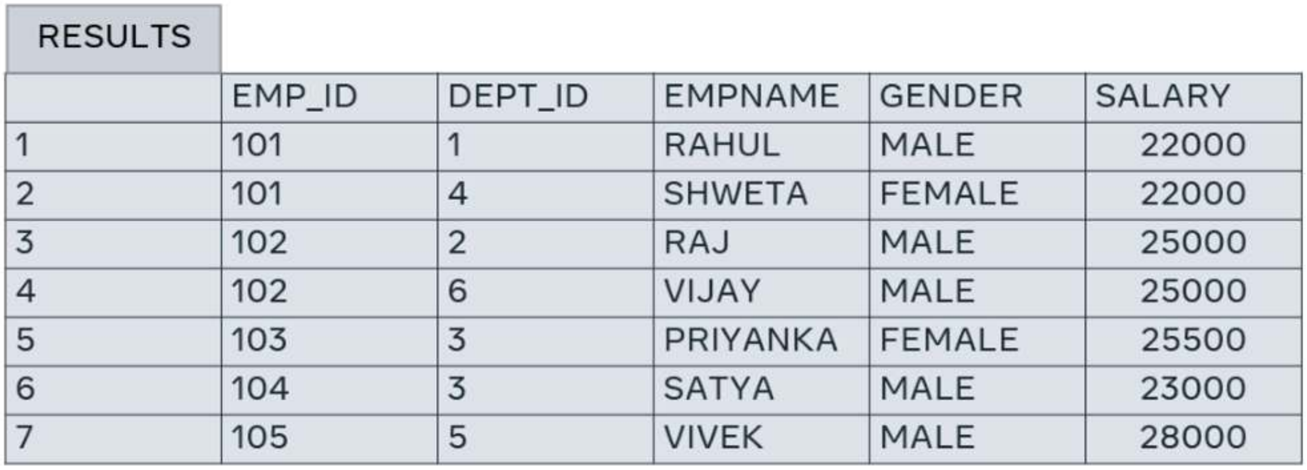
The student ID would probably have a data type of INT, for example. First name and last name would have a data type of VARCHAR and date of birth would have a data type of DATE.

What is a primary key?

In a table, there is a field or column that is known as a key which can uniquely identify a particular tuple (row) in a relation (table). This key is specifically known as a primary key.

For example, in the student table, the student ID allows you to uniquely identify a particular row. The other columns like first name, last name, date of birth and others could contain duplicate or repeating data for multiple students. Therefore, they can't be used to uniquely identify a given student record. So, the student ID is the primary key of the student table.

In some cases, the primary key can comprise more than one column or field. This happens when a single column cannot make a record in a table uniquely identifiable. For example, in the table below, the EMP_ID values aren't unique, so the column is not unique by itself. Thus, this column alone cannot be used as the primary key of this table. However, the EMP_ID and DEPT_ID columns together can make a record unique. Therefore, the primary key of this table is EMP_ID and DEPT_ID. This is also known as a composite primary key.

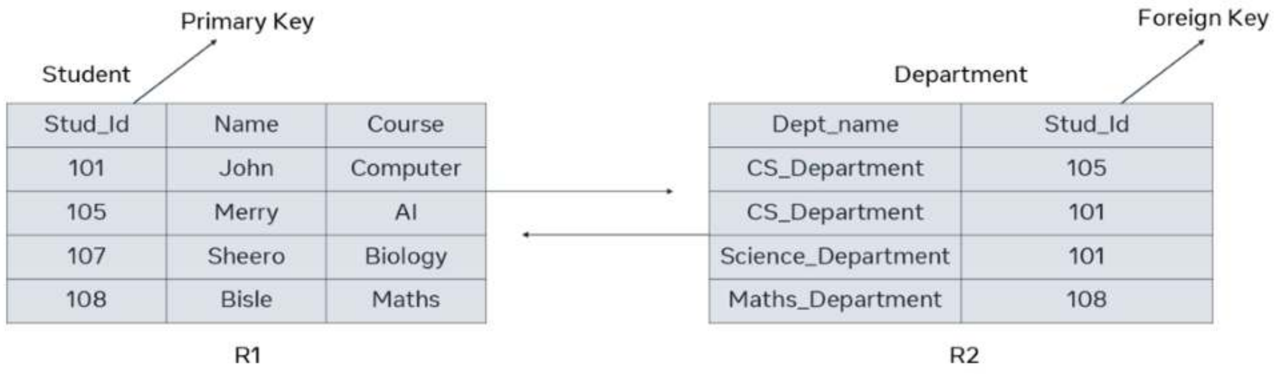


RESULTS

	EMP_ID	DEPT_ID	EMPNAME	GENDER	SALARY
1	101	1	RAHUL	MALE	22000
2	101	4	SHWETA	FEMALE	22000
3	102	2	RAJ	MALE	25000
4	102	6	VIJAY	MALE	25000
5	103	3	PRIYANKA	FEMALE	25500
6	104	3	SATYA	MALE	23000
7	105	5	VIVEK	MALE	28000

What is a foreign key?

Tables in a database do not stay isolated from each other. They need to have relationships between them. Tables are linked with one another through a key column (the primary key) of one table that's also present in the related table as a foreign key. For example, the student table and the department table are linked via the student ID which is the primary key of the student table that's also present in the Department table as a foreign key.



Stud_Id	Name	Course
101	John	Computer
105	Merry	AI
107	Sheero	Biology
108	Bisle	Maths

Dept_name	Stud_Id
CS_Department	105
CS_Department	101
Science_Department	101
Maths_Department	108

Integrity constraints

Every table in a database should abide by rules or constraints. These are known as integrity constraints.

There are three main integrity constraints:

1. Key constraints
2. Domain constraints
3. Referential integrity constraints

What are key constraints?

In every table there should be one or more columns or fields that can be used to fetch data from tables. In other words, a primary key. The key constraint specifies that there should be a column, or columns, in a table that can be used to fetch data for any row. This key attribute or primary key should never be NULL or the same for two different rows of data. For example, in the student table I can use the student ID to fetch data for each of the students. No value of student ID is null, and it is unique for every row, hence it can be the key attribute.

What are domain constraints?

Domain constraints refer to the rules defined for the values that can be stored for a certain column. For instance, you cannot store the home address of a student in the first name column. Similarly, a contact number cannot exceed ten digits.

What are referential integrity constraints?

When a table is related to another table via a foreign key column, then the referenced column value must exist in the other table. This means, according to the student and department examples, that values should exist in the student ID column in the student table because the two tables are related via the student ID column.

In this reading, you learned more about tables in a relational database as you explored the table in terms of its structure, data types, constraints, and the role of primary and foreign keys.

Database structure overview:The main objective of this reading is to cover the basic structure of a database. In other words, you will learn more about tables, fields (or attributes), records, keys and table relationships.

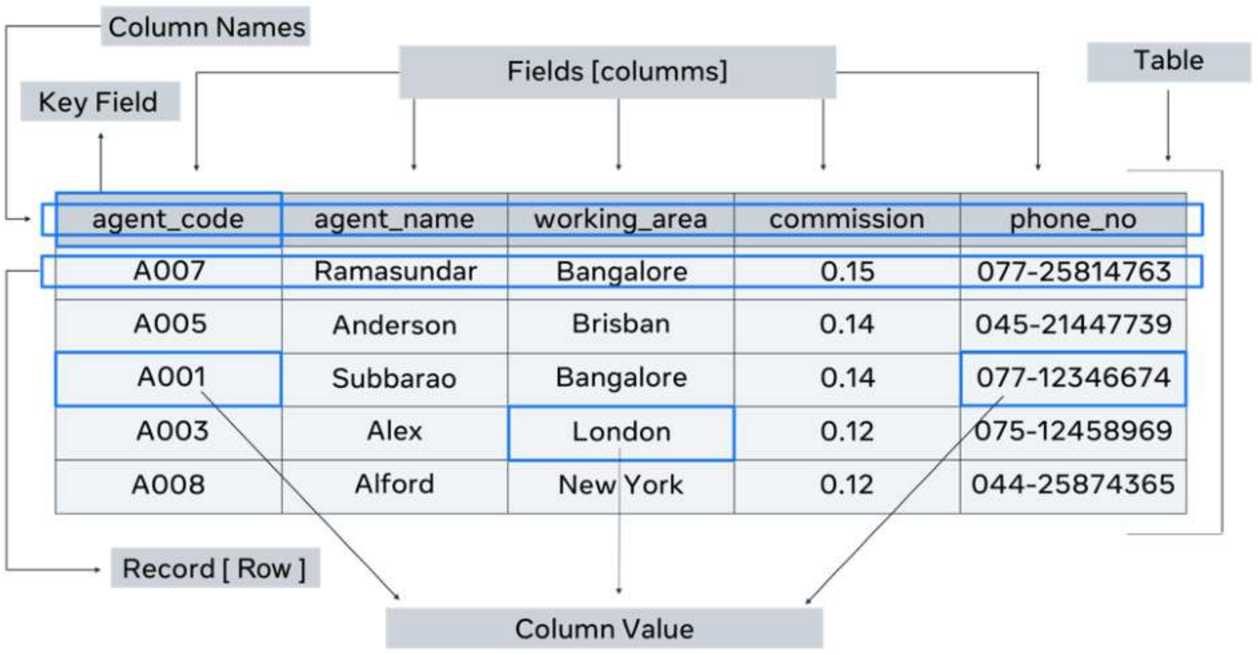
What is database structure?

Database structure refers to how data is arranged in a database. Within a database, related data are grouped into tables, each of which consists of rows (also called tuples) and columns, like in a spreadsheet.

The structure of a database consists of a set of key components. These include:

- Tables or entities, where the data is stored.
- Attributes which are details about the table or entity. In other words, attributes describe the table.
- Fields, which are columns used to capture attributes.
- A record, which is one row of details about a table or entity.
- And the primary key, which is a unique value for an entity.

This image shows the basic structural elements of a database table.



agent_code	agent_name	working_area	commission	phone_no
A007	Ramasundar	Bangalore	0.15	077-25814763
A005	Anderson	Brisban	0.14	045-21447739
A001	Subbarao	Bangalore	0.14	077-12346674
A003	Alex	London	0.12	075-12458969
A008	Alford	New York	0.12	044-25874365

Table

A table contains all the fields, attributes and records for a type of entity. A database will most probably contain more than one table.

Fields

Column headings are known as fields. Each field contains a different attribute. For every table, a unit of data is entered into each field. It’s also known as a column value. Each column has a data type. For example, the

olumn has a data type of text, and the “commission” column has a numeric data type.

Column value or unit of data

Each individual piece of data entered into a column is a unit of data. These units are also called data elements or column values.

Records

A record consists of a collection of data for each entity. It’s also known as a row in the table.

Data types

To keep the data consistent from one record to the next, an appropriate data type is assigned to each column. The data type of a column determines what type of data can be stored in each column.

Data types are also a way of classifying data values or column values. Different kinds of data values or column values require different amounts of memory to store them. Different operations can be performed on those column values based on their datatypes.

Some common data types used in databases are:

- Numeric data types such as INT, TINYINT, BIGINT, FLOAT and REAL.
- Date and time data types such as DATE, TIME and DATETIME.
- Character and string data types such as CHAR and VARCHAR.
- Binary data types such as BINARY and VARBINARY.
- And miscellaneous data types such as:
    - Character Large Object (CLOB), for storing a large block of text in some form of text encoding.
    - and Binary Large Object (BLOB), for storing a collection of binary data such as images.

Logical database structure

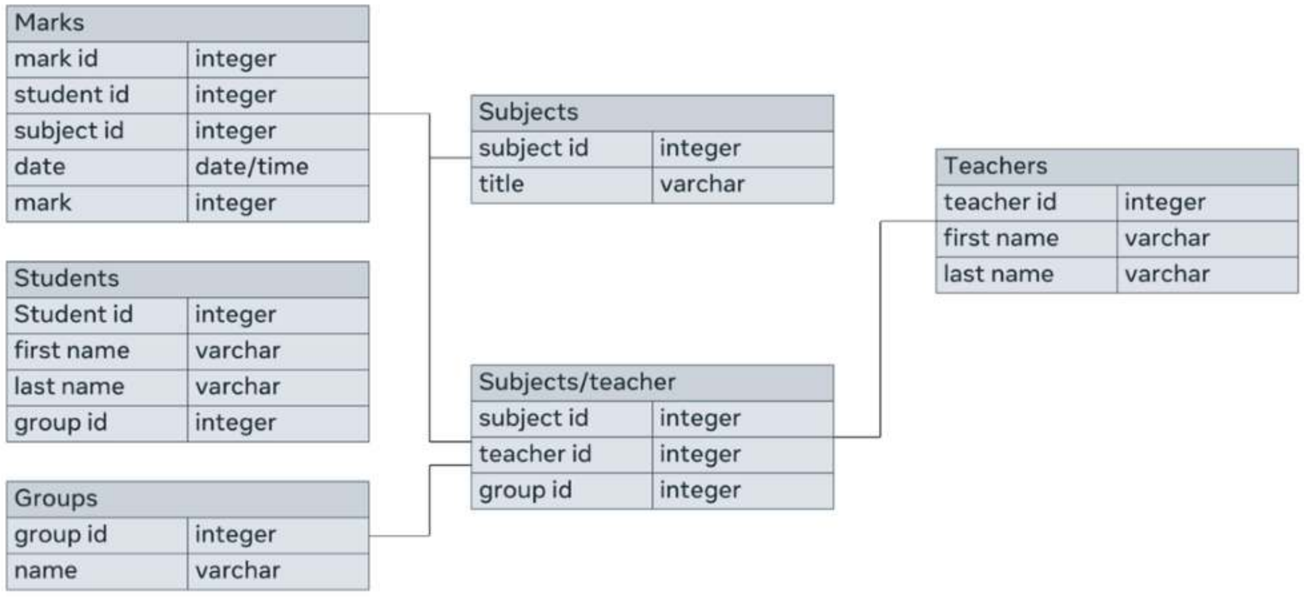
The logical structure of a database is represented using a diagram known as the Entity Relationship Diagram (ERD). It is a visual representation of how the database will be implemented into tables during physical database design, using a Database Management System (DBMS) like MySQL or Oracle, for example.

A part of the logical database structure is how relationships are established between entities. These relationships are established between the instances of the entities. Accordingly, there can be three ways in which entity instances can be related to each other:

- One-to-one relationships
- One-to-many relationships
- Many-to-many relationships

This is also known as cardinality of relationships. The logical database structure which is represented using an ERD also depicts these relationships.

Here's an example of an ERD that has all these elements.



Marks	
mark id	integer
student id	integer
subject id	integer
date	date/time
mark	integer

Students	
Student id	integer
first name	varchar
last name	varchar
group id	integer

Groups	
group id	integer
name	varchar

Subjects	
subject id	integer
title	varchar

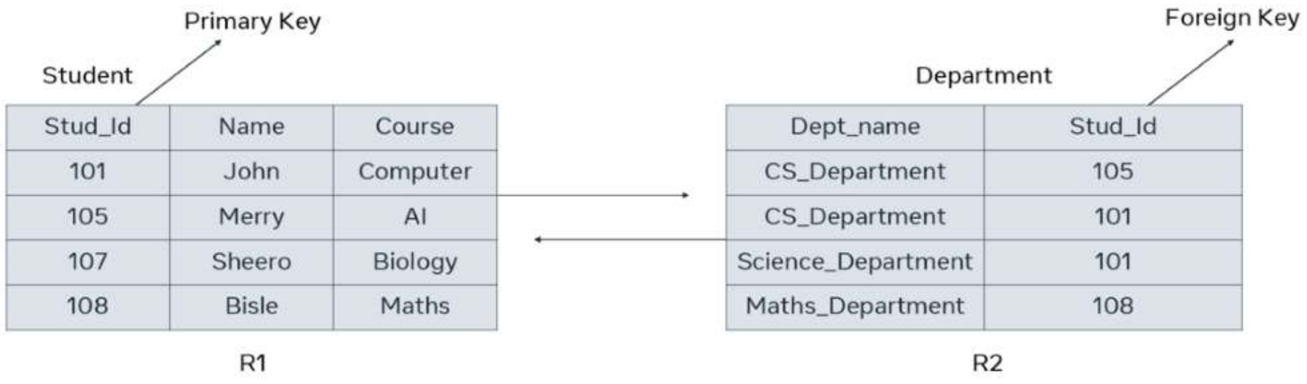
Subjects/teacher	
subject id	integer
teacher id	integer
group id	integer

Teachers	
teacher id	integer
first name	varchar
last name	varchar

Physical database structure

In the physical database structure, where entities are implemented as tables, the relationships are established using a field known as a foreign key. A foreign key is a field in one table that refers to a common field in another table (usually the primary key).

Let's take the example of a database that contains two tables: student and department. The student table has a primary key of "Stud_id", which is also present in the Department table as a foreign key. Therefore, the two tables are related to each other via the "Stud_id" field.



Stud_Id	Name	Course
101	John	Computer
105	Merry	AI
107	Sheero	Biology
108	Bisle	Maths

Dept_name	Stud_Id
CS_Department	105
CS_Department	101
Science_Department	101
Maths_Department	108

In this reading, you learned more about the basic database structure including tables, fields or attributes, records, keys and relationships between tables.

Create, Read, Update and Delete (CRUD) Operations

SQL data types

Data types: Datatypes tell a database management system how to interpret the value of a column

- Numric

- String
- Date & time

Numric data type: Datatypes that let columns store data as numbers within the database

- Integer: Represents a whole number value
- Decimal: Represent a number with a fraction value
- TINYINT: Represent small integer numrical values
- INT: Represent large numrical value

String data types: Used when storing data with mixed type of characters

- CHAR: Characters of fixed length. For fixed columns as String ID
- VARCHAR: Characters of variable length. For names or emails
- TEXT: Usually for feedback comments

Date & time are String

Database constraints: Limit the type of data that can be stored in a table

Violation example: An attempt to insert or upload invalid data

Constraints: Applied at column level

Rule: Applies to a specific column

FOREIGN KEY: Used to prevent actions that would destroy table links

NOT NULL: Preserves empty value fields, ensures data fields never left blank

DEFAULT: Assigns default values, sets a default value for a column if no value is specified

EXAMPLE:

CREATE TABLE employees (EmployeeID int NOT NULL, EmployeeName varchar(150) DEFAULT NULL, Department varchar(150) DEFAULT NULL, ContactNo varchar(12) DEFAULT NULL, Email varchar(100) DEFAULT NULL, AnnualSalary int DEFAULT NULL, PRIMARY KEY (EmployeeID));

- The CREATE TABLE statement is used to create a new table in the database.
- The table is named "employees".

  The table has several columns defined:

  - ***EmployeeID***: An integer column with the int data type that cannot contain NULL values (NOT NULL constraint).
  - ***EmployeeName***: A variable-length character column with the varchar(150) data type, allowing a maximum of 150 characters. It has a default value of NULL.
  - ***Department***: A variable-length character column with the varchar(150) data type, allowing a maximum of 150 characters. It has a default value of NULL.
  - ***ContactNo***: A variable-length character column with the varchar(12) data type, allowing a maximum of 12 characters. It has a default value of NULL.

    - ***Email***: A variable-length character column with the varchar(100) data type, allowing a maximum of 100 characters. It has a default value of NULL.
    - ***AnnualSalary***: An integer column with the int data type. It has a default value of NULL.
- The PRIMARY KEY constraint is applied to the EmployeeID column, indicating that it is the primary key for the table.

Create, Read, Update & Delete

- **create a database**: This statement is used to create a new database with the specified name. It creates a new container for tables, views, and other database objects.

CREATE DATABASE database_name;

- **drop a database**: This statement is used to delete an existing database and all its associated objects, including tables, views, stored procedures, and data.

DROP DATABASE database_name;

- **drop a table**: This statement is used to remove an existing table from the database, along with all its data and associated objects.

DROP TABLE table1_name;

- **create a table**: This statement is used to create a new table with the specified name and columns. You need to provide column names and their corresponding data types.

CREATE TABLE table1_name (column1_name DATATYPE...);

- **add a column to an existing table**: This statement is used to add a new column to an existing table. It allows you to extend the table's structure by specifying the column name and its data type.

ALTER TABLE table1_name ADD column2_name DATATYPE;

- **drop a column from an existing table**: This statement is used to remove a column from an existing table. It permanently deletes the specified column and its data from the table.

ALTER TABLE table1_name DROP COLUMN column2_name;

- **modify a column in an existing table**: This statement is used to modify the data type of a column in an existing table. It allows you to change the column's data type to a new one.

ALTER TABLE table1_name MODIFY column2_name DATATYPE;

- **insert a row with specific columns into a table**: This statement is used to insert a new row into a table. You provide the column names and their corresponding values for the new row.

INSERT INTO table1_name (column1_name, column2_name) VALUES (value1, value2);

- **insert multiple rows with specific columns into a table**: This statement is used to insert multiple rows into a table at once. It allows you to insert several rows with different values for the specified columns.

INSERT INTO table1_name (column1_name, column2_name) VALUES (value1, value2), (value1, value2);

- **select specific columns from a table**: This statement is used to retrieve data from a table. You specify the columns you want to select and the table from which you want to retrieve the data.

SELECT column1, column2 FROM table1;

- **insert data from one table's column into another table's column**: This statement is used to insert data from one table into another table. It selects the data from a specific column in the source table and inserts it into a specific column in the target table.

INSERT INTO target_table (column1_new) SELECT column2_exist FROM source_table;

- **update records in a table using the UPDATE statement**: This statement is used to update records in a table. It sets new values for specified columns in the table for records that match the specified condition.

UPDATE table1_name SET column1_name = 'column', column_number = '054323' WHERE column_record = 'John';

- **delete a single record from a table**: This statement is used to delete specific records from a table based on a specified condition. It removes the rows that match the condition from the table.

DELETE FROM table1_name WHERE column_record = 'record_name';

SQL Operators and Sorting and Filtering Data

SQL operators

Arithmetic operators

Arithmetic operators are useful when you want to perform mathematical operations on the data in tables while you retrieve them by writing SQL SELECT queries. In SQL, arithmetic operators are used to perform mathematical operations on data. To be more specific, they’re used with numerical data stored in database tables.

Arithmetic operators can be used in the SELECT clause as well as in the WHERE clause in a SQL SELECT statement. When an operator is used in the WHERE clause, it’s intended to perform the operations on specific rows only. This is because the WHERE clause in SQL is used to filter out data that a particular SQL statement is working on.

All arithmetic operators are used on numerical operands for performing:

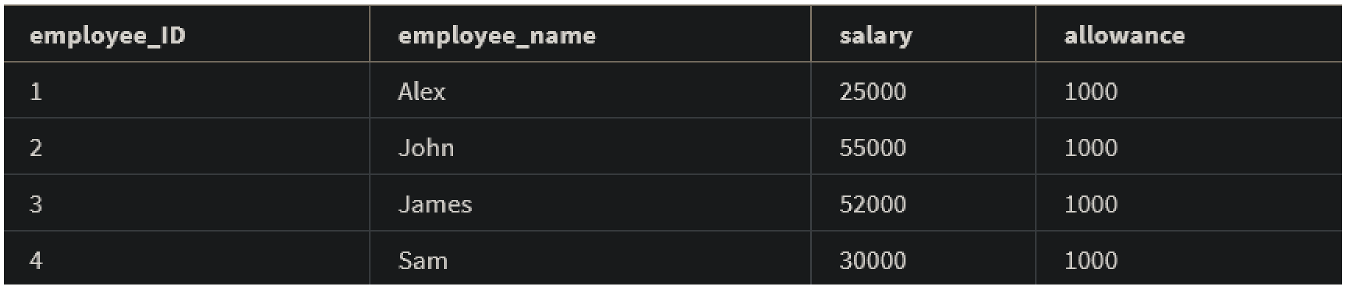
- Addition
- Subtraction
- Multiplication
- Division
- Modulus

Using the addition operator

The SQL addition operator performs the mathematical addition operation on numerical data within columns in a table. For example, if you want to add the values of two instances of numerical data from two separate columns in the table, then you need to specify the two columns as the first and second operand. The syntax is as follows:

```
SELECT column_name1 + column_name2 FROM table_name;
```

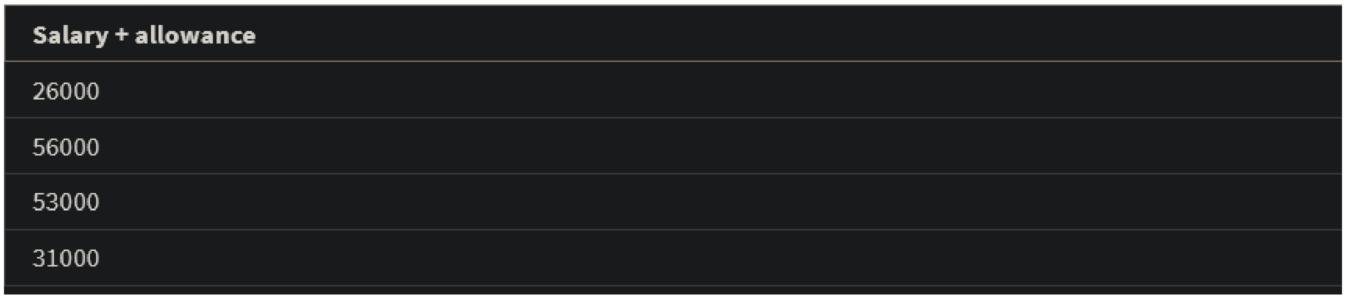
Let’s review an example. This is an employee table from a company database.



employee_ID	employee_name	salary	allowance
1	Alex	25000	1000
2	John	55000	1000
3	James	52000	1000
4	Sam	30000	1000

If you want to know the total salaries of all employees with the basic salary and the allowance added to it then you can use the addition operator. The SQL syntax for the addition operator is as follows:

```
SELECT salary + allowance FROM employee;
```

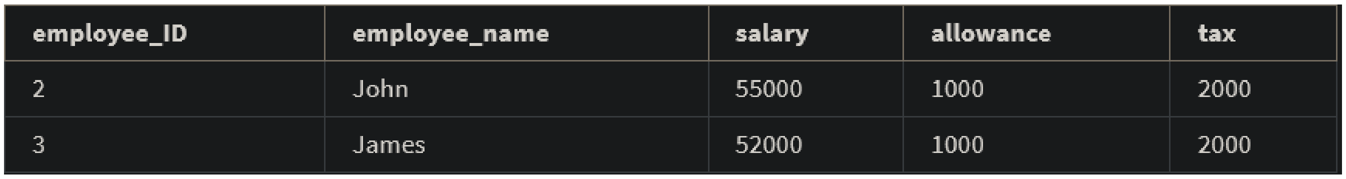


Salary + allowance
26000
56000
53000
31000

Now let’s review an example of how to use the multiplication operator in the WHERE clause.Let’s say you want to know who must pay an amount of tax equal to 4000, after doubling the current tax value.

The SELECT query gives the desired result, using the multiplication operator in the WHERE clause.

```
SELECT * FROM employee WHERE tax * 2 = 4000;
```



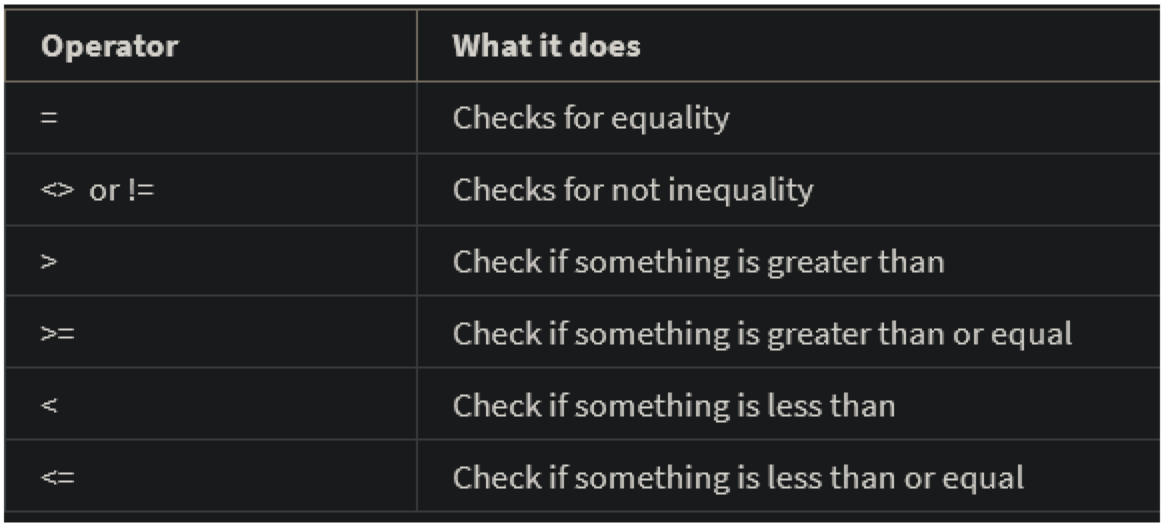
employee_ID	employee_name	salary	allowance	tax
2	John	55000	1000	2000
3	James	52000	1000	2000

SQL Comparison operator

Refreshing SQL comparison operators

As you’ve already learned, all the comparison operators that are used with other programming languages also work with SQL.

These comparison operators are:



Operator	What it does
=	Checks for equality
<> or !=	Checks for not inequality
>	Check if something is greater than
>=	Check if something is greater than or equal
<	Check if something is less than
<=	Check if something is less than or equal

Sorting & Filtering data

Types of ordering / sorting

You'll explore using the ORDER BY clause for sorting data.

The ORDER BY clause

The ORDER BY clause is useful when you want to sort or order the results obtained when running a SQL SELECT query. Data in a database become more meaningful when they are ordered or sorted in a specific order. Ordered data helps people make more accurate business decisions effectively and efficiently.

In SQL, there's the ORDER BY clause that can help you achieve this. If you run a SQL SELECT query, you get a set of unsorted results. If you want to sort them, you need to add the special ORDER BY clause into the SQL SELECT statement.

It can be used after the FROM clause as in:

```
SELECT * FROM Employee ORDER BY ;
```

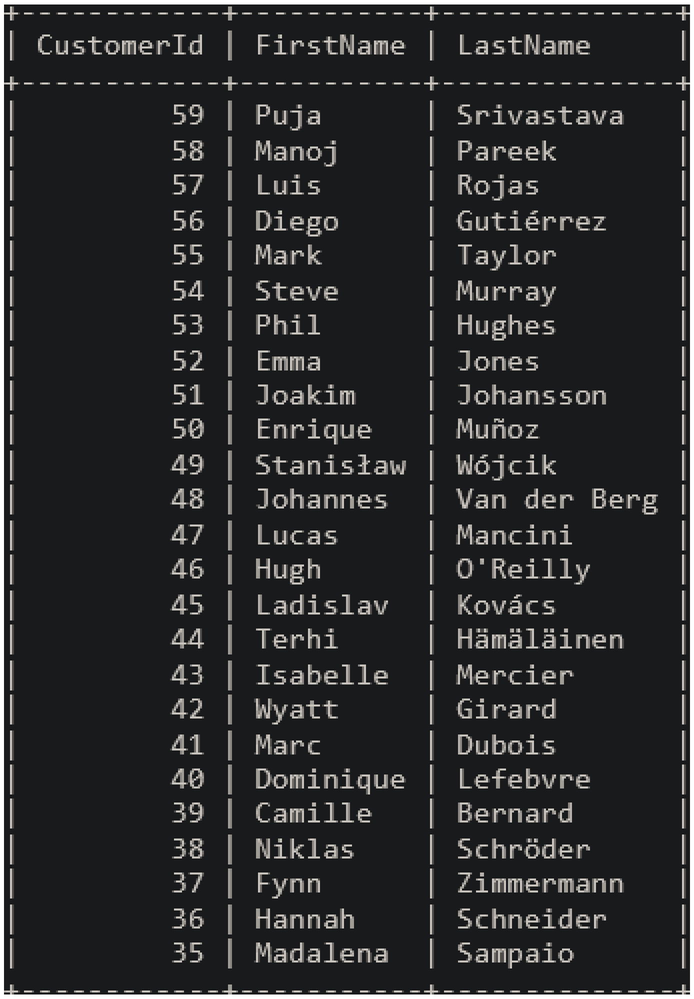
After the ORDER BY keyword, you need to specify the column name based on the data that needs to be sorted. Optionally, you can specify the keywords ASC or DESC after the column name. This is to indicate if the ordering should be in ascending or descending order.

Ascending and descending order are the two main types of ordering. If ASC or DESC are not specified, the data is sorted by default in ascending order. The ASC and DESC keywords sort the data based on the order by column, taking into consideration the data type of column or field, namely integer, numeric, text and dates.

Sorting by a single column:

Loading [MathJax]/extensions/Safe.js

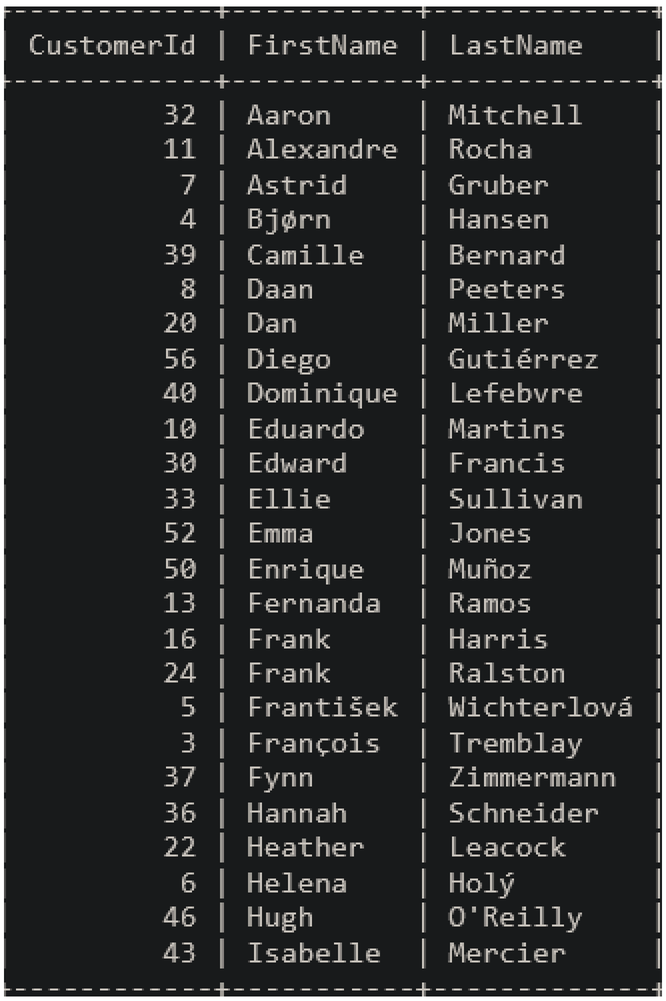
SELECT * FROM customers ORDER BY CustomerId DESC;



```
+------------+-----------+--------------+
| CustomerId | FirstName | LastName     |
+------------+-----------+--------------+
|         59 | Puja      | Srivastava   |
|         58 | Manoj     | Pareek       |
|         57 | Luis      | Rojas        |
|         56 | Diego     | Gutiérrez    |
|         55 | Mark      | Taylor       |
|         54 | Steve     | Murray       |
|         53 | Phil      | Hughes       |
|         52 | Emma      | Jones        |
|         51 | Joakim    | Johansson    |
|         50 | Enrique   | Muñoz        |
|         49 | Stanisław | Wójcik       |
|         48 | Johannes  | Van der Berg |
|         47 | Lucas     | Mancini      |
|         46 | Hugh      | O'Reilly     |
|         45 | Ladislav  | Kovács       |
|         44 | Terhi     | Hämäläinen   |
|         43 | Isabelle  | Mercier      |
|         42 | Wyatt     | Girard       |
|         41 | Marc      | Dubois       |
|         40 | Dominique | Lefebvre     |
|         39 | Camille   | Bernard      |
|         38 | Niklas    | Schröder     |
|         37 | Fynn      | Zimmermann   |
|         36 | Hannah    | Schneider    |
|         35 | Madalena  | Sampaio      |
+------------+-----------+--------------+
```

Now let’s examine how sorting happens for a text data typed column

SELECT * FROM customers ORDER BY FirstName;



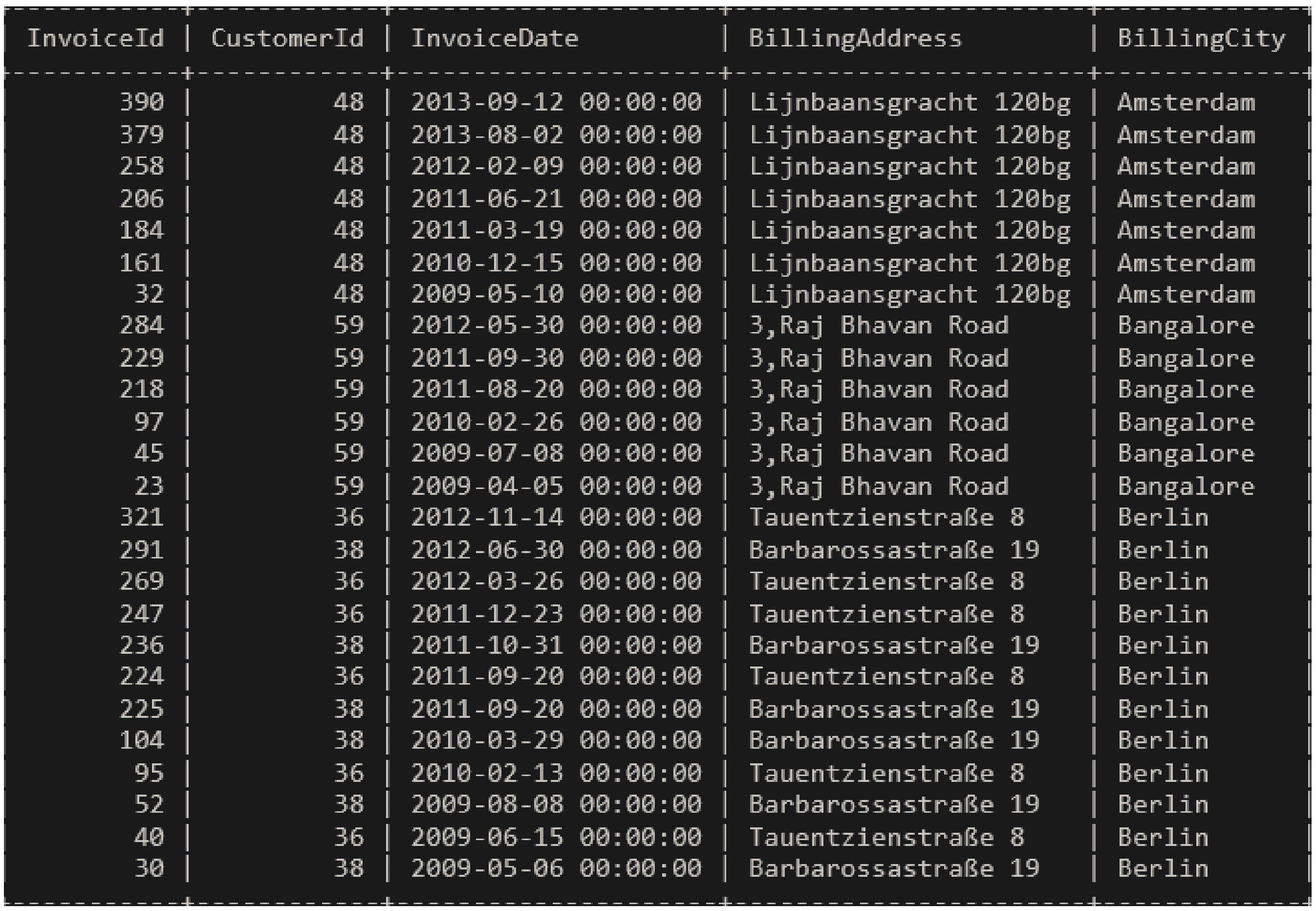
```
+------------+-----------+-------------+
| CustomerId | FirstName | LastName    |
+------------+-----------+-------------+
|         32 | Aaron     | Mitchell    |
|         11 | Alexandre | Rocha       |
|          7 | Astrid    | Gruber      |
|          4 | Bjørn     | Hansen      |
|         39 | Camille   | Bernard     |
|          8 | Daan      | Peeters     |
|         20 | Dan       | Miller      |
|         56 | Diego     | Gutiérrez   |
|         40 | Dominique | Lefebvre    |
|         10 | Eduardo   | Martins     |
|         30 | Edward    | Francis     |
|         33 | Ellie     | Sullivan    |
|         52 | Emma      | Jones       |
|         50 | Enrique   | Muñoz       |
|         13 | Fernanda  | Ramos       |
|         16 | Frank     | Harris      |
|         24 | Frank     | Ralston     |
|          5 | František | Wichterlová |
|          3 | François  | Tremblay    |
|         37 | Fynn      | Zimmermann  |
|         36 | Hannah    | Schneider   |
|         22 | Heather   | Leacock     |
|          6 | Helena    | Holý        |
|         46 | Hugh      | O'Reilly    |
|         43 | Isabelle  | Mercier     |
+------------+-----------+-------------+
```

Ordering by multiple columns

You can also sort data by multiple columns and apply different sort orders to them. Let’s say you want to sort invoice data by both billing city and invoice date. To do this, run the following query:

Loading [MathJax]/extensions/Safe.js

SELECT * FROM invoices ORDER BY BillingCity ASC, InvoiceDate DESC;



```
+-----------+------------+---------------------+------------------------+-------------+
| InvoiceId | CustomerId | InvoiceDate         | BillingAddress         | BillingCity |
+-----------+------------+---------------------+------------------------+-------------+
|       390 |         48 | 2013-09-12 00:00:00 | Lijnbaansgracht 120bg  | Amsterdam   |
|       379 |         48 | 2013-08-02 00:00:00 | Lijnbaansgracht 120bg  | Amsterdam   |
|       258 |         48 | 2012-02-09 00:00:00 | Lijnbaansgracht 120bg  | Amsterdam   |
|       206 |         48 | 2011-06-21 00:00:00 | Lijnbaansgracht 120bg  | Amsterdam   |
|       184 |         48 | 2011-03-19 00:00:00 | Lijnbaansgracht 120bg  | Amsterdam   |
|       161 |         48 | 2010-12-15 00:00:00 | Lijnbaansgracht 120bg  | Amsterdam   |
|        32 |         48 | 2009-05-10 00:00:00 | Lijnbaansgracht 120bg  | Amsterdam   |
|       284 |         59 | 2012-05-30 00:00:00 | 3,Raj Bhavan Road      | Bangalore   |
|       229 |         59 | 2011-09-30 00:00:00 | 3,Raj Bhavan Road      | Bangalore   |
|       218 |         59 | 2011-08-20 00:00:00 | 3,Raj Bhavan Road      | Bangalore   |
|        97 |         59 | 2010-02-26 00:00:00 | 3,Raj Bhavan Road      | Bangalore   |
|        45 |         59 | 2009-07-08 00:00:00 | 3,Raj Bhavan Road      | Bangalore   |
|        23 |         59 | 2009-04-05 00:00:00 | 3,Raj Bhavan Road      | Bangalore   |
|       321 |         36 | 2012-11-14 00:00:00 | Tauentzienstraße 8     | Berlin      |
|       291 |         38 | 2012-06-30 00:00:00 | Barbarossastraße 19    | Berlin      |
|       269 |         36 | 2012-03-26 00:00:00 | Tauentzienstraße 8     | Berlin      |
|       247 |         36 | 2011-12-23 00:00:00 | Tauentzienstraße 8     | Berlin      |
|       236 |         38 | 2011-10-31 00:00:00 | Barbarossastraße 19    | Berlin      |
|       224 |         36 | 2011-09-20 00:00:00 | Tauentzienstraße 8     | Berlin      |
|       225 |         38 | 2011-09-20 00:00:00 | Barbarossastraße 19    | Berlin      |
|       104 |         38 | 2010-03-29 00:00:00 | Barbarossastraße 19    | Berlin      |
|        95 |         36 | 2010-02-13 00:00:00 | Tauentzienstraße 8     | Berlin      |
|        52 |         38 | 2009-08-08 00:00:00 | Barbarossastraße 19    | Berlin      |
|        40 |         36 | 2009-06-15 00:00:00 | Tauentzienstraße 8     | Berlin      |
|        30 |         38 | 2009-05-06 00:00:00 | Barbarossastraße 19    | Berlin      |
+-----------+------------+---------------------+------------------------+-------------+
```

You'll notice that the data is sorted in ascending order of BillingCity. That's why the data in the BillingCity column is sorted in alphabetical order. The data of the InvoiceDatecolumn is in turn sorted in descending order.

For example, if you review the records with the billing city of Amsterdam, the invoice dates are ordered in descending order from largest to smallest date. Similarly, if you examine the other sets of data closely, you'll observe the same.

The main types of ordering in SQL are ASC, ascending, and DESC, descending. How the data is ordered in these two cases would depend on the data type of the field or column being used as the sort column.

WHERE Clause

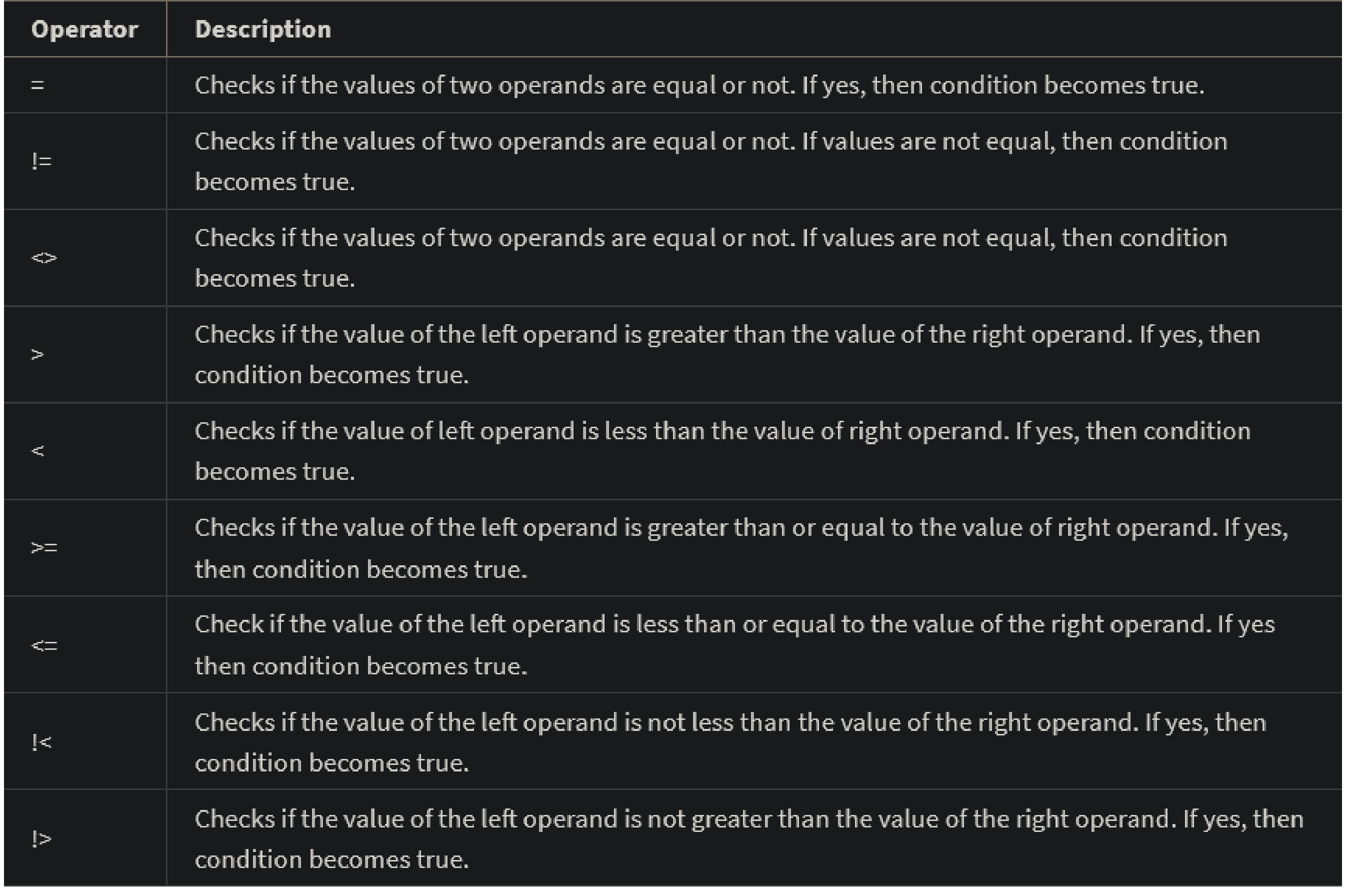
You'll explore the usage of the WHERE clause for filtering data

The WHERE clause

The WHERE clause is useful when you want to filter data in a table based on a given condition in the SQL statement.The WHERE clause in SQL is there for the purpose of filtering records and fetching only the necessary records. This can be used in SQL SELECT, UPDATE and DELETE statements.

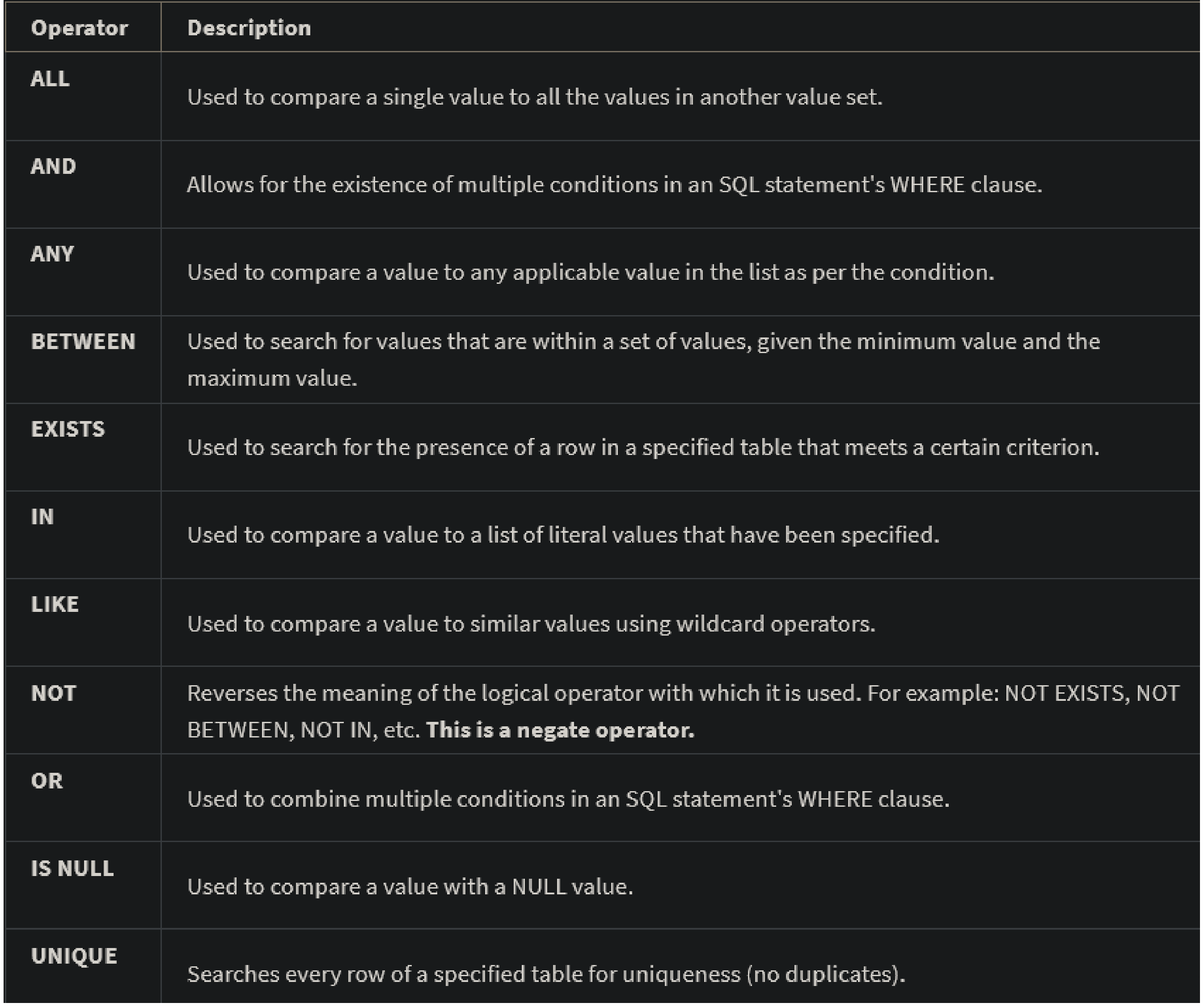
The filtering happens based on a condition. The condition can be written using any of the following comparison or logical operators. ***Comparison operators***

Loading [MathJax]/extensions/Safe.js



Operator	Description
=	Checks if the values of two operands are equal or not. If yes, then condition becomes true.
!=	Checks if the values of two operands are equal or not. If values are not equal, then condition becomes true.
<>	Checks if the values of two operands are equal or not. If values are not equal, then condition becomes true.
>	Checks if the value of the left operand is greater than the value of the right operand. If yes, then condition becomes true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then condition becomes true.
>=	Checks if the value of the left operand is greater than or equal to the value of right operand. If yes, then condition becomes true.
<=	Check if the value of the left operand is less than or equal to the value of the right operand. If yes then condition becomes true.
!<	Checks if the value of the left operand is not less than the value of the right operand. If yes, then condition becomes true.
!>	Checks if the value of the left operand is not greater than the value of the right operand. If yes, then condition becomes true.

Logical operators



Operator	Description
ALL	Used to compare a single value to all the values in another value set.
AND	Allows for the existence of multiple conditions in an SQL statement's WHERE clause.
ANY	Used to compare a value to any applicable value in the list as per the condition.
BETWEEN	Used to search for values that are within a set of values, given the minimum value and the maximum value.
EXISTS	Used to search for the presence of a row in a specified table that meets a certain criterion.
IN	Used to compare a value to a list of literal values that have been specified.
LIKE	Used to compare a value to similar values using wildcard operators.
NOT	Reverses the meaning of the logical operator with which it is used. For example: NOT EXISTS, NOT BETWEEN, NOT IN, etc. **This is a negate operator.**
OR	Used to combine multiple conditions in an SQL statement's WHERE clause.
IS NULL	Used to compare a value with a NULL value.
UNIQUE	Searches every row of a specified table for uniqueness (no duplicates).

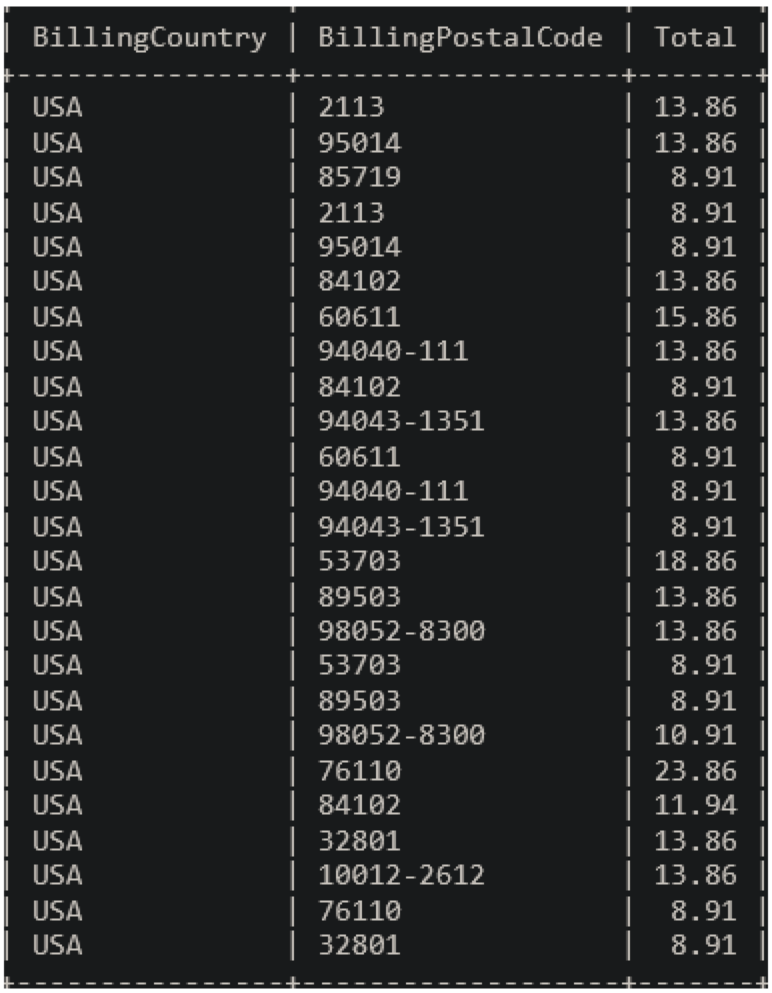
let's review an example:

The syntax required to use the AND operator in the WHERE clause of a SELECT statement is as follows:

SELECT column1, column2, columnN FROM table_name WHERE [condition1] AND [condition2]...AND [conditionN];SELECT

Loading [MathJax]/extensions/Safe.js

* FROM invoices WHERE Total > 2 AND BillingCountry = 'USA';



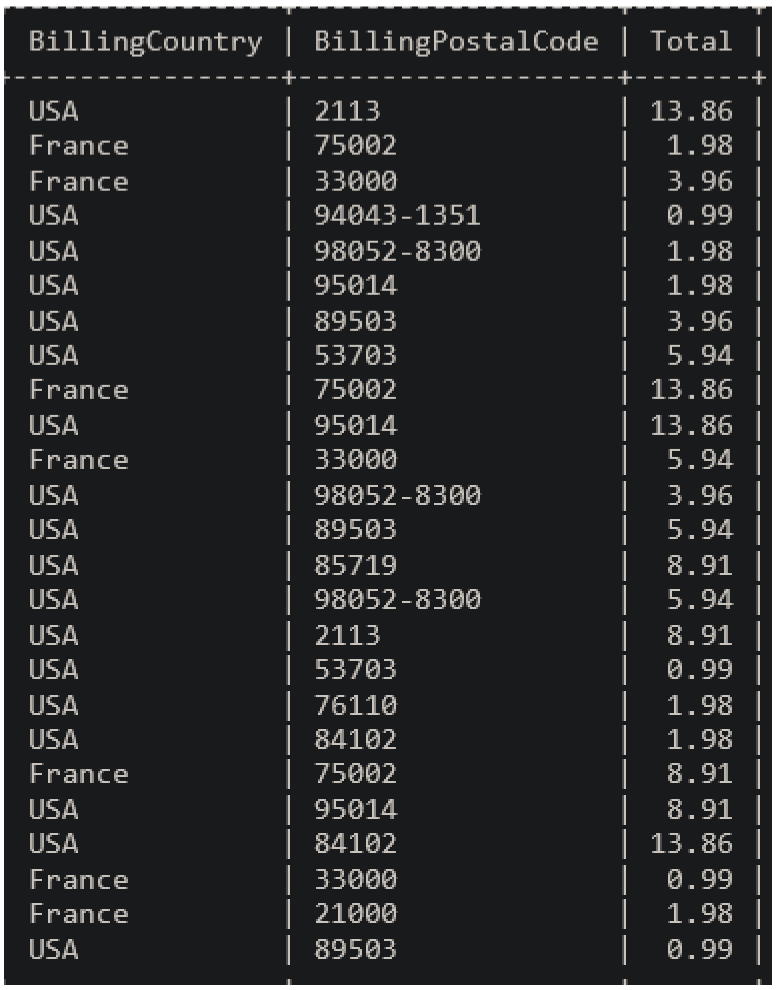
```
+----------------+-------------------+-------+
| BillingCountry | BillingPostalCode | Total |
+----------------+-------------------+-------+
| USA            | 2113              | 13.86 |
| USA            | 95014             | 13.86 |
| USA            | 85719             |  8.91 |
| USA            | 2113              |  8.91 |
| USA            | 95014             |  8.91 |
| USA            | 84102             | 13.86 |
| USA            | 60611             | 15.86 |
| USA            | 94040-111         | 13.86 |
| USA            | 84102             |  8.91 |
| USA            | 94043-1351        | 13.86 |
| USA            | 60611             |  8.91 |
| USA            | 94040-111         |  8.91 |
| USA            | 94043-1351        |  8.91 |
| USA            | 53703             | 18.86 |
| USA            | 89503             | 13.86 |
| USA            | 98052-8300        | 13.86 |
| USA            | 53703             |  8.91 |
| USA            | 89503             |  8.91 |
| USA            | 98052-8300        | 10.91 |
| USA            | 76110             | 23.86 |
| USA            | 84102             | 11.94 |
| USA            | 32801             | 13.86 |
| USA            | 10012-2612        | 13.86 |
| USA            | 76110             |  8.91 |
| USA            | 32801             |  8.91 |
+----------------+-------------------+-------+
```

Here, the AND operator is used as a conjunctive operator to combine the two conditions Total > 8 AND BillingCountrywhich is the USA. You'll receive the invoice records with a total bill value of more than $8 with the USA as billing country. This means that for a record to be included in the result, both the conditions should be true. Similarly, **the OR operator** can also be used to combine multiple conditions in the WHERE clause.

SELECT column1, column2, columnN FROM table_name WHERE [condition1] OR [condition2]...OR [conditionN] SELECT *

Loading [MathJax]/extensions/Safe.js

FROM invoices WHERE BillingCountry = 'USA' OR BillingCountry='France';



```
+----------------+------------------+-------+
 BillingCountry | BillingPostalCode | Total |
+----------------+------------------+-------+
 USA            | 2113              | 13.86 |
 France         | 75002             |  1.98 |
 France         | 33000             |  3.96 |
 USA            | 94043-1351        |  0.99 |
 USA            | 98052-8300        |  1.98 |
 USA            | 95014             |  1.98 |
 USA            | 89503             |  3.96 |
 USA            | 53703             |  5.94 |
 France         | 75002             | 13.86 |
 USA            | 95014             | 13.86 |
 France         | 33000             |  5.94 |
 USA            | 98052-8300        |  3.96 |
 USA            | 89503             |  5.94 |
 USA            | 85719             |  8.91 |
 USA            | 98052-8300        |  5.94 |
 USA            | 2113              |  8.91 |
 USA            | 53703             |  0.99 |
 USA            | 76110             |  1.98 |
 USA            | 84102             |  1.98 |
 France         | 75002             |  8.91 |
 USA            | 95014             |  8.91 |
 USA            | 84102             | 13.86 |
 France         | 33000             |  0.99 |
 France         | 21000             |  1.98 |
 USA            | 89503             |  0.99 |
```

SELECT DISTINCT clause in use

You'll explore the usage of SELECT DISTINCT to retrieve a unique set of values in a SELECT statement

The DISTINCT keyword

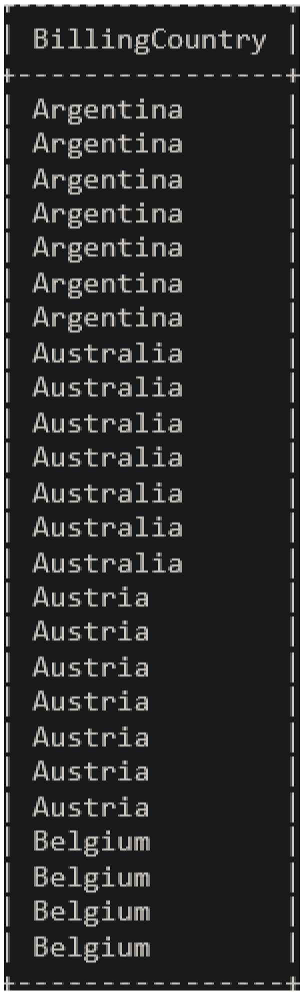
DISTINCT is useful for retrieving a set of unique values when there are duplicate column values in a table. It is used with the SELECT statement, so it's commonly referred to as SELECT DISTINCT. In short, what DISTINCT does is to findunique values within a column, or columns, of a table.

Let's look at some examples of how the DISTINCT keyword behaves using a few data retrieval scenarios from the table in the sample database.

Using SELECT DISTINCT on a single column

If there's a table named invoices with the same BillingCountryrepeated in many instances, you can run the following query to identify what they are:

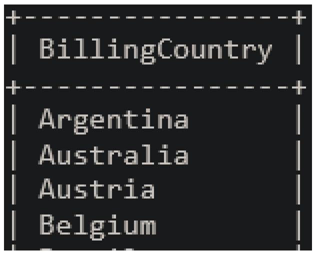
SELECT BillingCountry FROM invoices ORDER BY BillingCountry;



```
+----------------+
| BillingCountry |
+----------------+
| Argentina      |
| Argentina      |
| Argentina      |
| Argentina      |
| Argentina      |
| Argentina      |
| Argentina      |
| Australia      |
| Australia      |
| Australia      |
| Australia      |
| Australia      |
| Australia      |
| Australia      |
| Austria        |
| Austria        |
| Austria        |
| Austria        |
| Austria        |
| Austria        |
| Austria        |
| Belgium        |
| Belgium        |
| Belgium        |
| Belgium        |
+----------------+
```

When you look at the result, you'll notice that there are duplicate values in the BillingCountry column. How can you obtain a list of unique billing countries where the invoices have been raised? Let's change the SELECT statement by adding the DISTINCT keyword and then run it again.

SELECT DISTINCT BillingCountry FROM invoices ORDER BY BillingCountry;



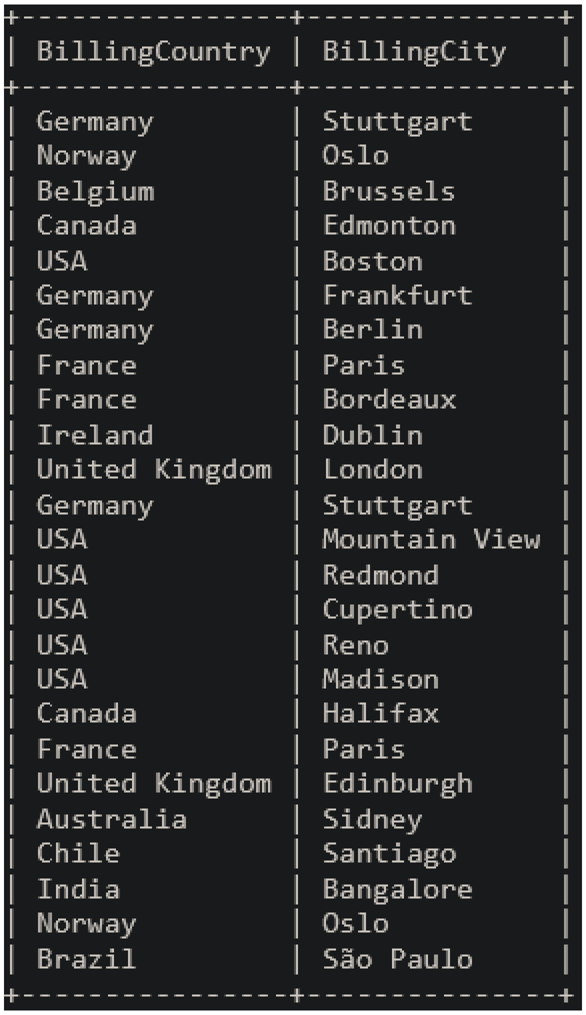
```
+----------------+
| BillingCountry |
+----------------+
| Argentina      |
| Australia      |
| Austria        |
| Belgium        |
```

This time, the duplicate values are gone and only a unique set of billing countries are returned as the result.

Using SELECT DISTINCT on multiple columns

Loading [MathJax]/extensions/Safe.js

SELECT DISTINCT BillingCountry, BillingCity FROM invoices;



```
+----------------+---------------+
| BillingCountry | BillingCity   |
+----------------+---------------+
| Germany        | Stuttgart     |
| Norway         | Oslo          |
| Belgium        | Brussels      |
| Canada         | Edmonton      |
| USA            | Boston        |
| Germany        | Frankfurt     |
| Germany        | Berlin        |
| France         | Paris         |
| France         | Bordeaux      |
| Ireland        | Dublin        |
| United Kingdom | London        |
| Germany        | Stuttgart     |
| USA            | Mountain View |
| USA            | Redmond       |
| USA            | Cupertino     |
| USA            | Reno          |
| USA            | Madison       |
| Canada         | Halifax       |
| France         | Paris         |
| United Kingdom | Edinburgh     |
| Australia      | Sidney        |
| Chile          | Santiago      |
| India          | Bangalore     |
| Norway         | Oslo          |
| Brazil         | São Paulo     |
+----------------+---------------+
```

The result is a unique set of billing cities retrieved for the billing countries. Basically, there are no duplicate values in the BillingCity column. In other words, when you do a DISTINCT of multiple columns, it looks for a combination of unique values in all those columns. In this example, all combinations of BillingCountryand BillingCity in the result are unique.

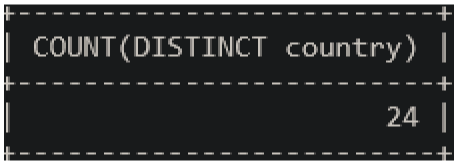
Using DISTINCT with SQL aggregate functions

DISTINCT can also be used with SQL aggregate functions like COUNT, AVG, MAX and so on. In this case, you must specify an expression that's written using some aggregate function. Therefore, it's not only column names that you can use DISTINCT with but also with expressions.

What if you want to find out the number of unique countries of the customers in the customer table? Run a SELECT statement that uses the aggregate function COUNT on the country column along with DISTINCT.

For example:

SELECT COUNT(DISTINCT country) FROM customers;



```
+-------------------------+
| COUNT(DISTINCT country) |
+-------------------------+
|                      24 |
+-------------------------+
```

Database Design

Loading [MathJax]/extensions/Safe.js

Exploring database schema

What is a database schema?

Designing the schema or structure of a database is the very first step in designing a database system. Database schema is about the structure of a database. In other words, how data is organized in a database. Data in a database is organized into tables that have columns and rows. Each column or field has a defined data type, and the tables are related to each other. The simplest way of understanding database schema is to think of it as the blueprint of a database. Before anyone can use a database to store and manipulate data, the database schema must first be designed. This process of database schema design is also known as data modeling.

Usually, the database schema is designed by database designers. The database schema is just the skeleton of the database, and it doesn't store any actual data. Once the designers have provided the database schema, the developers can understand how the data should be stored by the application that they are implementing.

Database schema can be broadly divided into three categories.

1. Conceptual or logical schema that defines entities, attributes and relationships.
2. Internal or physical schema that defines how data is stored in a secondary storage. In other words, the actual storage of data and access paths.
3. External or view schema that defines different user views.

Create a simple database Schema:

```
CREATE DATABASE shopping_cart_db; -- run the statment CREATE TABLE customer( customer_id INT, name
VARCHAR(100), address VARCHAR(255), email VARCHAR(100), phone VARCHAR(10), PRIMARY KEY (customer_id) );
CREATE TABLE product( product_id INT, name VARCHAR(100), price NUMERIC(8, 2), description VARCHAR(255),
PRIMARY KEY (product_id) ); CREATE TABLE cart_order ( order_id INT, customer_id INT, product_id INT, quantity INT ,
order_date DATE, PRIMARY KEY (order_id), FOREIGN KEY (customer_id) REFERENCES customer (customer_id),
FOREIGN KEY (product_id) REFERENCES product (product_id) );
```

Type of schema:

1. Logical Schema

The logical database schema illustrate relationships between different tables or entities Entity relationship (ER) modeling: Illustrating relationships between entity types

2. Physical Schema

This involve creating the actual structure of your database using code

Relational database design

What is the relational model?

The relational model is built around three main concepts which are:

- Data,

- Relationships,
- and constraints.

It describes a database as “a collection of inter-related relations (or tables)”. It is still a dominant model used for data storage and retrieval. In essence, it is a way of organizing or storing data in a database. SQL is the language that’s used to retrieve data from a relational database.

Types of relationships

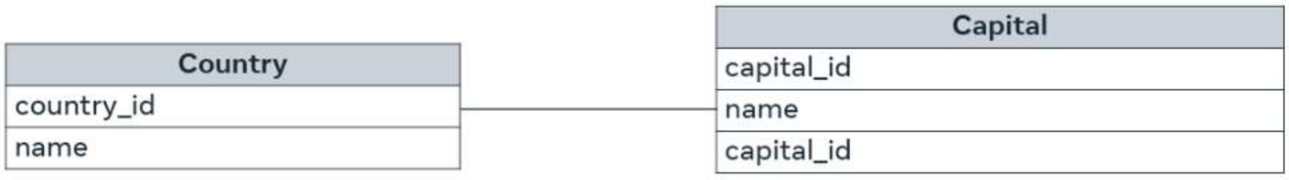
In the relational model, there are three types of relationships that can exist between tables.

1. One-to-one
2. One-to-many
3. Many-to-many

One-to-one

In order to understand one-to-one relationships, let’s take the example of two tables: Table A and Table B. A one-to-one (1:1) relationship means that each record in Table A relates to one, and only one, record in Table B. Likewise, each record in Table B relates to one, and only one, record in Table A.

Here is a diagram that illustrates the example:

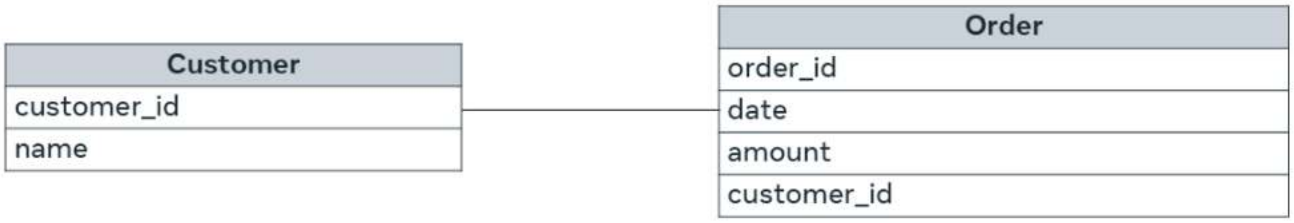


Here, every country has one, and only one, capital. And every capital belongs to one and only one country.

One-to-many

If there are two tables, Table A and Table B, a one-to-many (1:N) relationship means a record in Table A can relate to zero, one, or many records in Table B. Many records in Table B can relate to one record in Table A.

Let’s examine the following relationship between customers and orders.

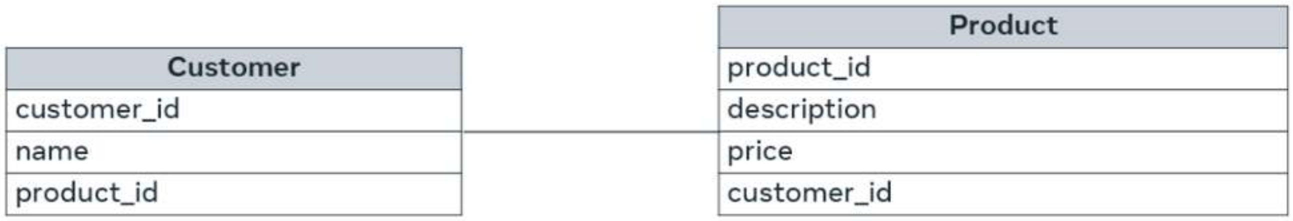


Here, each customer can place many orders. Many records in the order table can relate to only one record in the customer table.

Loading [MathJax]/extensions/Safe.js

Many-to-many

If there are two tables, Table A and Table B, a many-to-many (N:N) relationship means many records in Table A can relate to many records in Table B. And many records in Table B can relate to many records in Table A. Let’s examine this example of many-to-many relationship between customer and product with the use of a diagram. In the diagram, customers can purchase various products, and products can be purchased by many customers.



Usually, many-to-many relationships are not kept in a data model. They are broken down into two one-to-many relationships by introducing a junction or middle table.

To conclude, there are many benefits to the relational database model. This includes the ability to design and develop a meaningful system of information, and the ability to access and retrieve every single piece of data stored in the database.

Keys

Primary Key: A primary key is a unique identifier for a record in a database table. It uniquely identifies each row in the table, and no two rows can have the same primary key value. A primary key enforces entity integrity by ensuring uniqueness and serves as a reference point for relationships between tables.

```
CREATE TABLE employees ( employee_id INT PRIMARY KEY, employee_name VARCHAR(100), department
VARCHAR(100) );
```

In this example, the "employee_id" column is defined as the primary key for the "employees" table. Each row in the table will have a unique employee ID, and it serves as the primary identifier for each employee record.

Candidate Key: A candidate key is a column or set of columns in a table that can uniquely identify each row. Unlike a primary key, a table can have multiple candidate keys. One of the candidate keys is selected as the primary key, and the remaining candidate keys are called alternate keys.

```
CREATE TABLE students ( student_id INT PRIMARY KEY, student_name VARCHAR(100), email VARCHAR(100) UNIQUE,
phone VARCHAR(10) );
```

In this example, both the "student_id" and "email" columns are candidate keys for the "students" table. Each student record will have a unique student ID, and the email address will also be unique for each student. We selected the "student_id" as the primary key, and the "email" column becomes an alternate candidate key.

Composite Primary Key: A composite key is a primary key or candidate key that consists of two or more columns. It is used when a single column cannot uniquely identify a row, but the combination of multiple columns can. The combination of columns in the composite key must be unique.

```
CREATE TABLE orders ( order_id INT, customer_id INT, order_date DATE, PRIMARY KEY (order_id, order_date) );
```

In this example, the "orders" table has a composite key consisting of both "order_id" and "order_date". Together, these two columns uniquely identify each order. The combination of order ID and order date must be unique within the table, ensuring the uniqueness of each order record.

Foreign Key: A foreign key is a constraint in a relational database that establishes a link or relationship between two tables based on the values of one or more columns. It ensures referential integrity and maintains consistency between related data in different tables.

```
CREATE TABLE customers ( customer_id INT PRIMARY KEY, customer_name VARCHAR(100), address VARCHAR(255),
email VARCHAR(100) ); CREATE TABLE orders ( order_id INT PRIMARY KEY, order_date DATE, customer_id INT,
FOREIGN KEY (customer_id) REFERENCES customers (customer_id) );
```

In this example, we have two tables: "customers" and "orders". The "customers" table has a primary key defined on the "customer_id" column. The "orders" table has a primary key defined on the "order_id" column.

The "orders" table also includes a foreign key, "customer_id", which references the "customer_id" column in the "customers" table. This foreign key establishes a relationship between the two tables. It ensures that every value in the "customer_id" column of the "orders" table must exist in the "customer_id" column of the "customers" table.

Entity relationship diagrams (ERD)

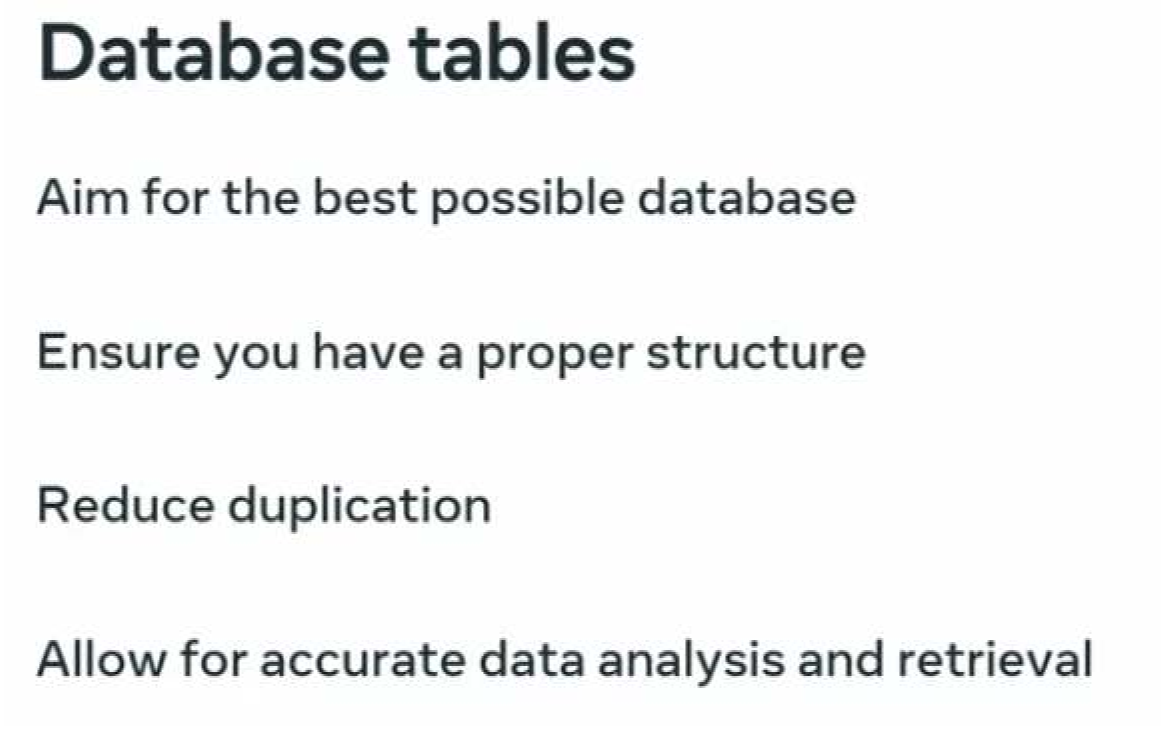
CHECK Lucid channel in YouTube

Database normalization

The normalization process aims to minimize data duplications, avoid errors during data modifications and simplify data queries from the database. The three fundamental normalization forms are known as:

- First Normal Form (1NF)
- Second Normal Form (2NF)
- Third Normal Form (3NF)

In this reading, you will learn how to apply the rules that ensure that a database meets the criteria of these three normal forms.



Database tables

Aim for the best possible database

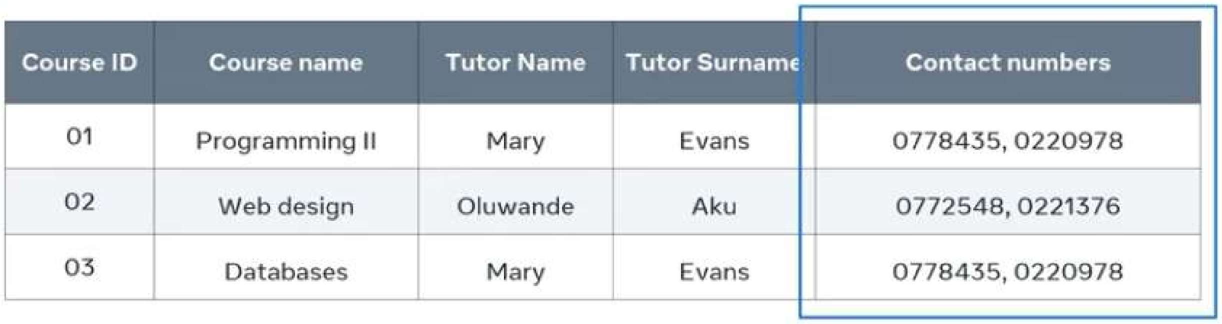
Ensure you have a proper structure

Reduce duplication

Allow for accurate data analysis and retrieval

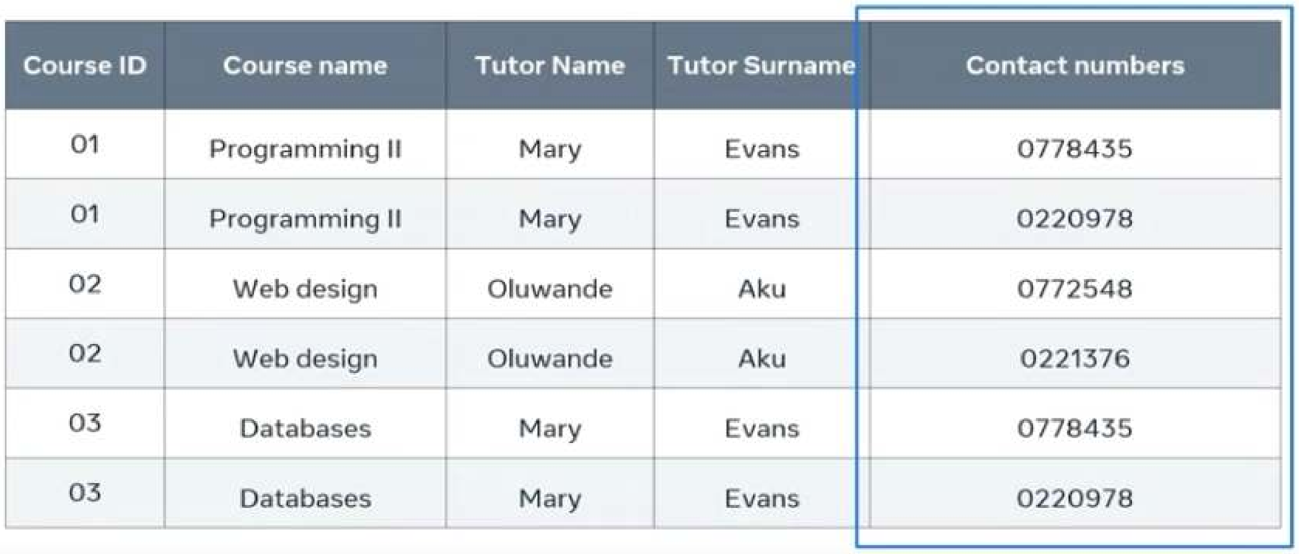
First normal form (1NF): Enforces data atomicity and eliminates unnecessary repeating groups of data in database (Remove Repeating Group, & multi valued row)

- Data atomicity: Ensurign that there is only one single instance value per column field.



Course ID	Course name	Tutor Name	Tutor Surname	Contact numbers
01	Programming II	Mary	Evans	0778435, 0220978
02	Web design	Oluwande	Aku	0772548, 0221376
03	Databases	Mary	Evans	0778435, 0220978

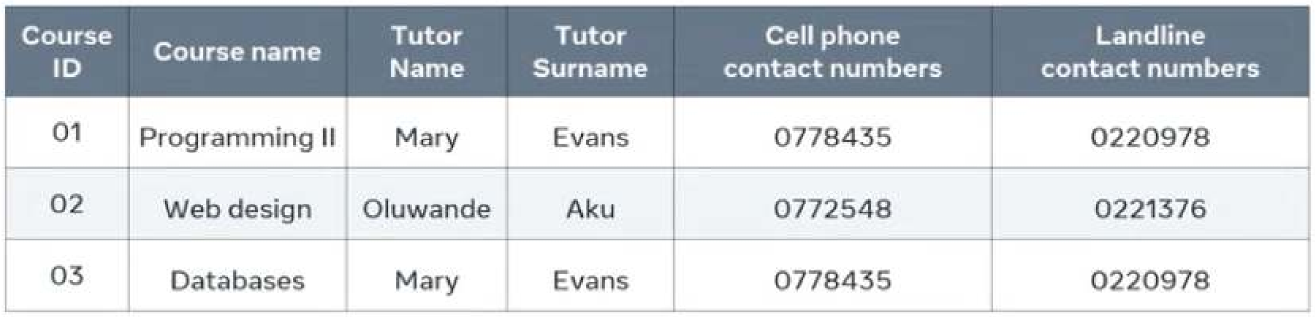
As you can see **above** in Contact numbers including multiple values in a single field. Therefore, we can solve this by putting each value in single field, like this:



Course ID	Course name	Tutor Name	Tutor Surname	Contact numbers
01	Programming II	Mary	Evans	0778435
01	Programming II	Mary	Evans	0220978
02	Web design	Oluwande	Aku	0772548
02	Web design	Oluwande	Aku	0221376
03	Databases	Mary	Evans	0778435
03	Databases	Mary	Evans	0220978

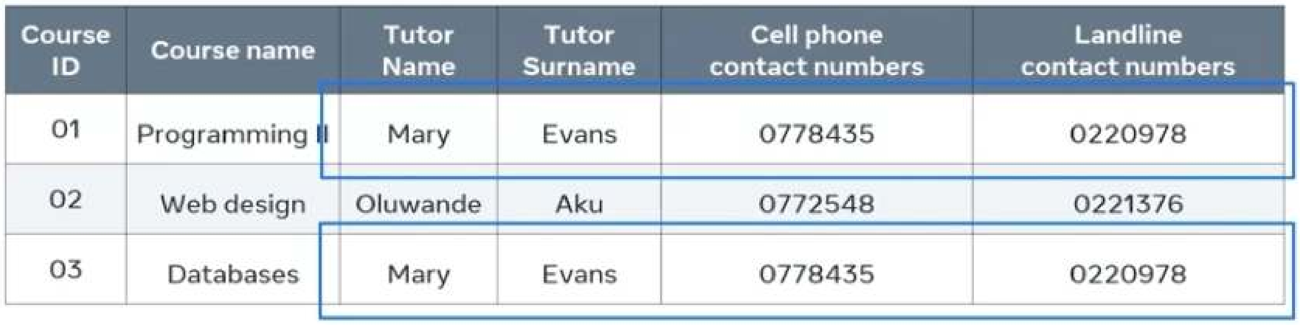
Loading [MathJax]/extensions/Safe.js

BUT this solution has also create another problem. The primary key is no longer unique, because multipule rows now has same course ID. So, we can solve this by putting each cContact number value for the same Course ID in different column, as below:



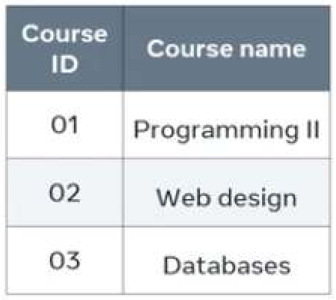
Course ID	Course name	Tutor Name	Tutor Surname	Cell phone contact numbers	Landline contact numbers
01	Programming II	Mary	Evans	0778435	0220978
02	Web design	Oluwande	Aku	0772548	0221376
03	Databases	Mary	Evans	0778435	0220978

BUT I still have the issue of unnecessary repeated groups of data as below:



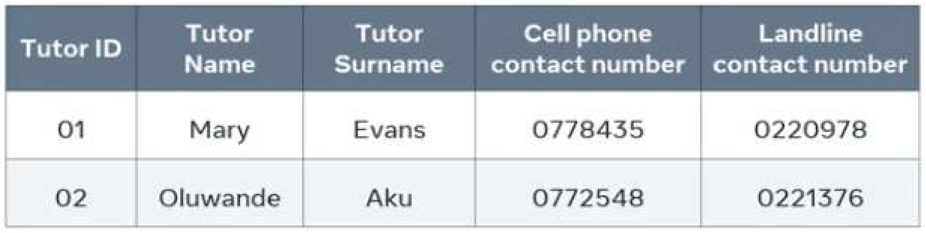
Course ID	Course name	Tutor Name	Tutor Surname	Cell phone contact numbers	Landline contact numbers
01	Programming II	Mary	Evans	0778435	0220978
02	Web design	Oluwande	Aku	0772548	0221376
03	Databases	Mary	Evans	0778435	0220978

To solve that we can split it to two tables and add Tutor ID to be the Primary key for Tutor table:



Course ID	Course name
01	Programming II
02	Web design
03	Databases

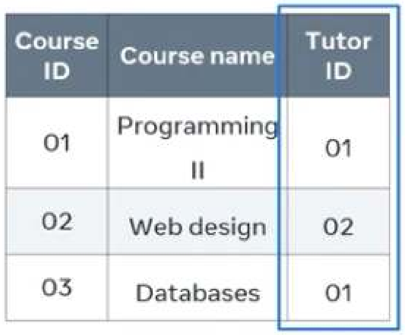
Course table



Tutor ID	Tutor Name	Tutor Surname	Cell phone contact number	Landline contact number
01	Mary	Evans	0778435	0220978
02	Oluwande	Aku	0772548	0221376

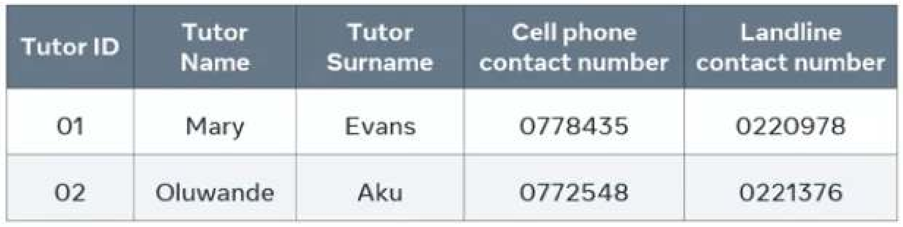
Tutor table

Finally we solve it, but we also need to provide a link between the two tables by using a foreign key



Course ID	Course name	Tutor ID
01	Programming II	01
02	Web design	02
03	Databases	01

Course table



Tutor ID	Tutor Name	Tutor Surname	Cell phone contact number	Landline contact number
01	Mary	Evans	0778435	0220978
02	Oluwande	Aku	0772548	0221376

Tutor table

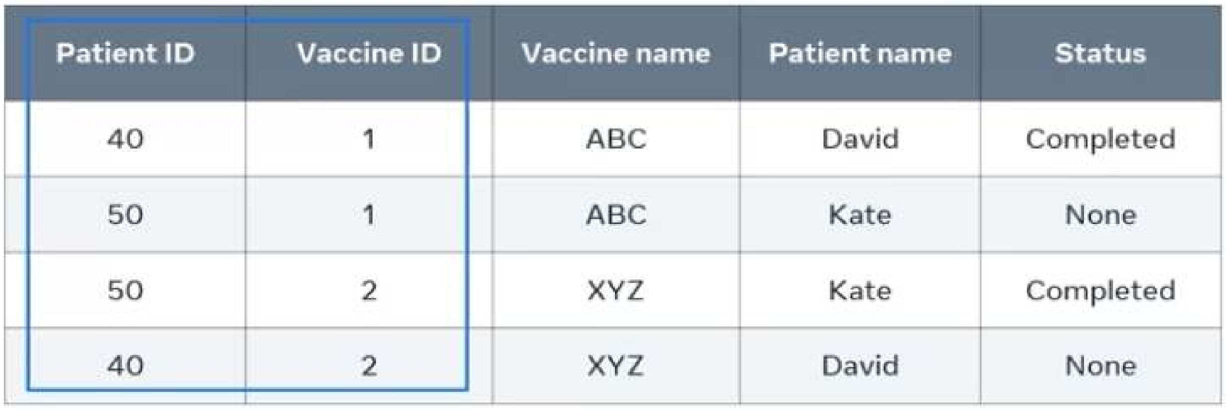
For more explination about Mary Evans :

Loading [MathJax]/extensions/Safe.js

Mary Evans is the assigned tutor for two of the courses. Her name appears twice in the table as do her contact details. These instances of data will continue to reappear if she's assigned more courses to teach. It's likely that our details will appear in other tables within a database system. This means I could have even more groups of repeated data. This creates another problem. If this user changes any of their details, then I'll have to update their details in this table and all others in which it appears. If I miss any of these tables, then I'll have inconsistency and invalid data within my database system.

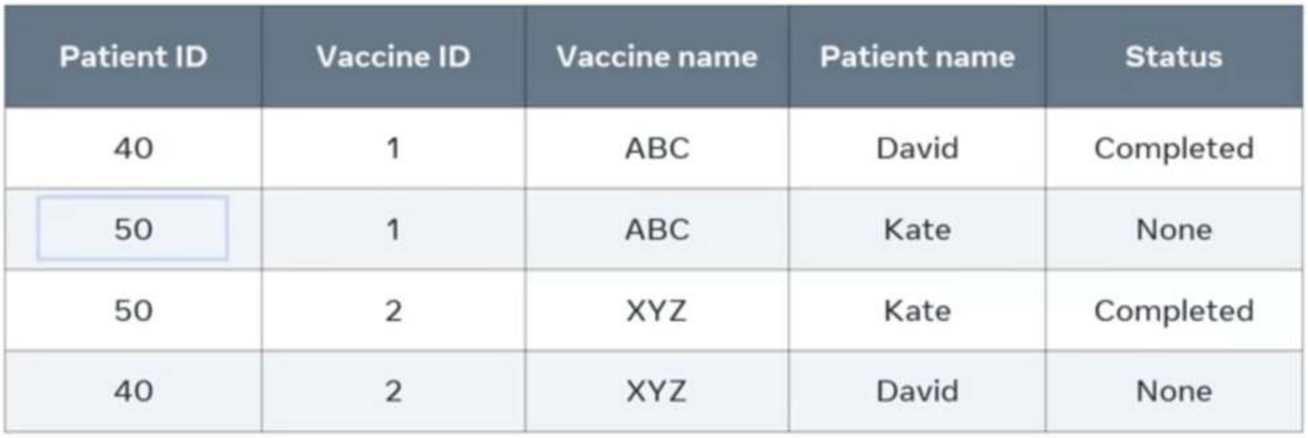
Before we begin withthe 2NF make sure you got 1NF. also we should know

- Functional dependency: The relationship between two attributes in a table
- Partial dependency: A table with a composite primary key



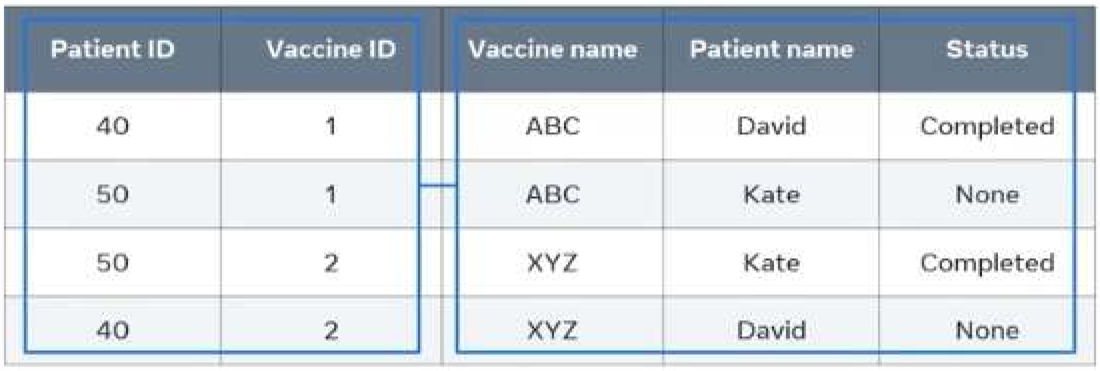
Patient ID	Vaccine ID	Vaccine name	Patient name	Status
40	1	ABC	David	Completed
50	1	ABC	Kate	None
50	2	XYZ	Kate	Completed
40	2	XYZ	David	None

Second normal form (2NF): Avoid any partial dependency relationships between data.(Remove Partial Dependency)



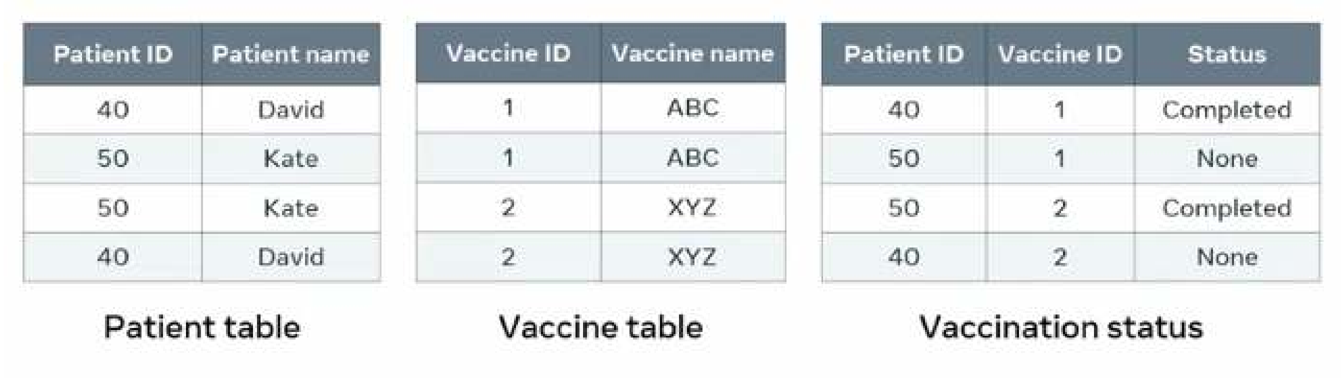
Patient ID	Vaccine ID	Vaccine name	Patient name	Status
40	1	ABC	David	Completed
50	1	ABC	Kate	None
50	2	XYZ	Kate	Completed
40	2	XYZ	David	None

First, I need to make all non-key columns dependent on all components of the primary key.



Patient ID	Vaccine ID	Vaccine name	Patient name	Status
40	1	ABC	David	Completed
50	1	ABC	Kate	None
50	2	XYZ	Kate	Completed
40	2	XYZ	David	None

By identify the entities included in the vaccination table. In this instance there are three entities: Vaccination status, as represented by the status column, vaccine, which is the vaccine ID and vaccine name columns, and patient, represented by the patient name and patient ID columns. I then break up the table into three separate tables as follows: Patient table, vaccine table, and vaccination status.



Patient ID	Patient name
40	David
50	Kate
50	Kate
40	David

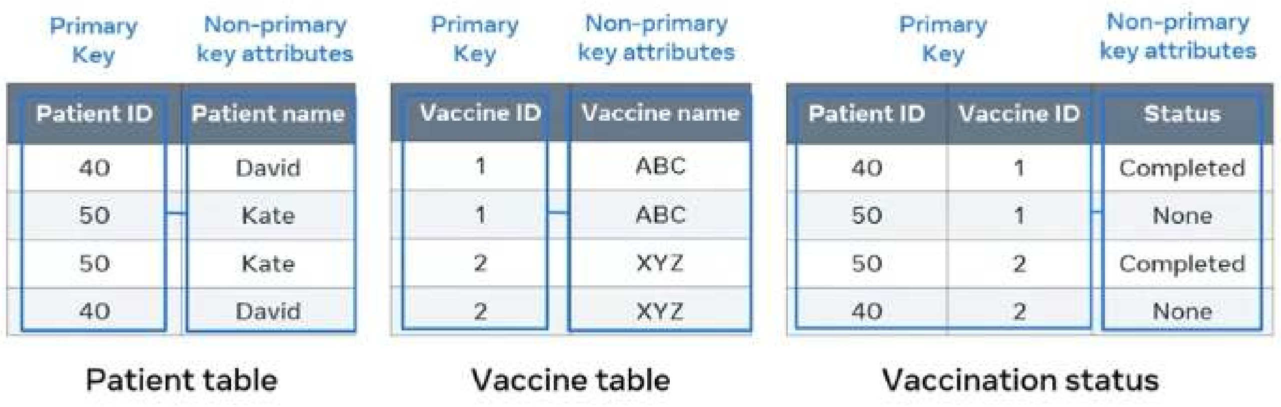
Patient table

Vaccine ID	Vaccine name
1	ABC
1	ABC
2	XYZ
2	XYZ

Vaccine table

Patient ID	Vaccine ID	Status
40	1	Completed
50	1	None
50	2	Completed
40	2	None

Vaccination status


Primary Key
Non-primary key attributes
Primary Key
Non-primary key attributes
Primary Key
Non-primary key attributes

Patient ID	Patient name
40	David
50	Kate
50	Kate
40	David

Patient table

Vaccine ID	Vaccine name
1	ABC
1	ABC
2	XYZ
2	XYZ

Vaccine table

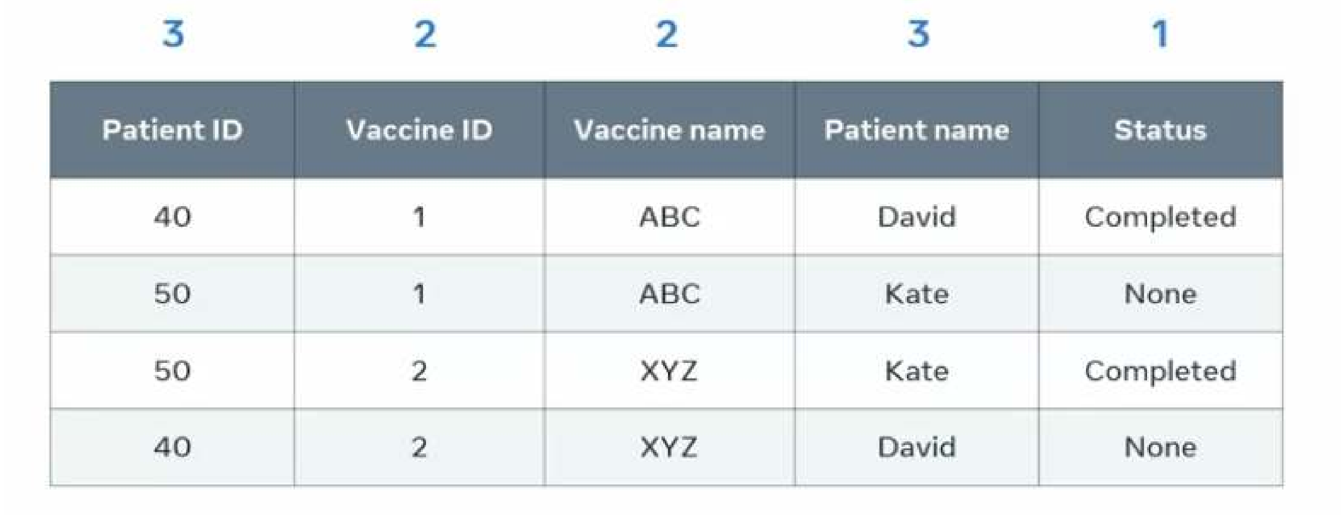
Patient ID	Vaccine ID	Status
40	1	Completed
50	1	None
50	2	Completed
40	2	None

Vaccination status

For more explination about Partial dependency:

Partial dependency is a concept in database normalization that occurs when a non-key attribute is functionally dependent on only a part of a candidate key, rather than the entire candidate key.

In other words, a partial dependency exists when a non-key attribute is determined by only a subset of the columns that make up the primary key of a table. This means that the non-key attribute is functionally dependent on a proper subset of the primary key, rather than the entire primary key.



3	2	2	3	1
Patient ID	Vaccine ID	Vaccine name	Patient name	Status
40	1	ABC	David	Completed
50	1	ABC	Kate	None
50	2	XYZ	Kate	Completed
40	2	XYZ	David	None

For example as you can see in the above figure, we have a composite primary key of Patient ID & Vaccine ID. But the issue that the Non-attribute key can depend just in one of the composite Primary key . this called Prtial dependency

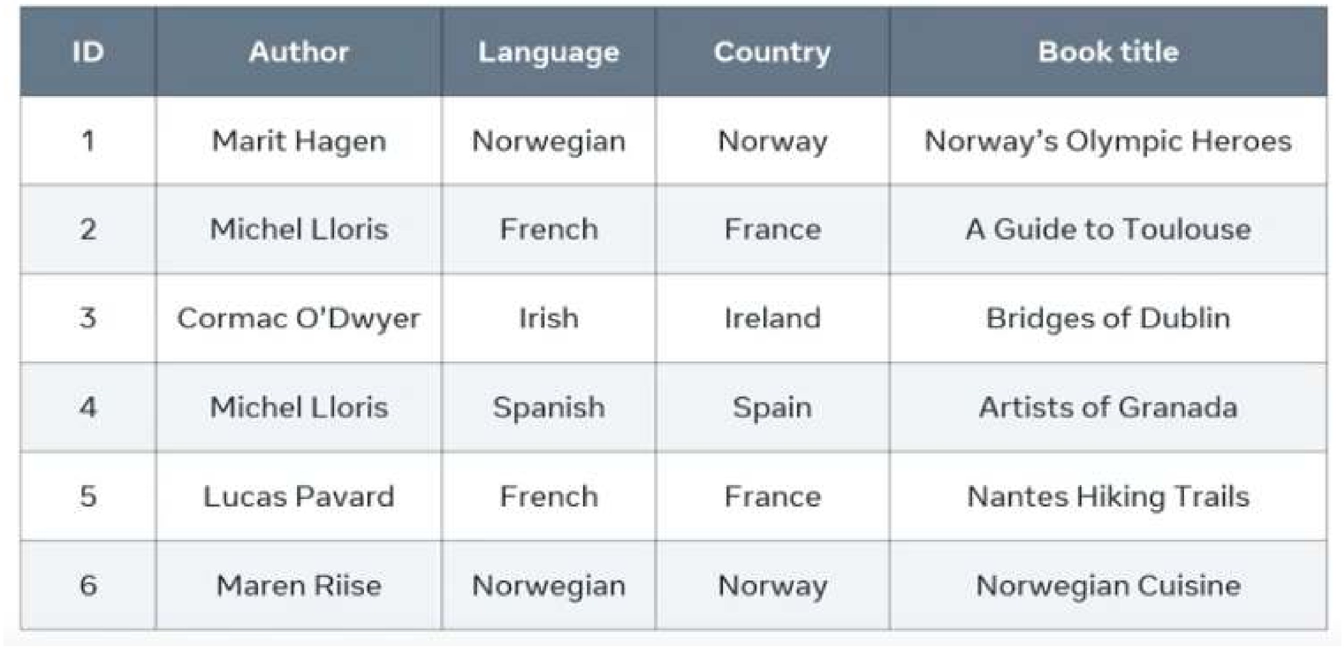
---

NOTE Database normalization is a progressive process, which means that the database relation cannot be in the third normal form if it not already applying thr rules of the first and second normal forms

Third normal form (3NF): It must have no transitive dependency.(Remove Transitive Dependancy)

This means that any non-key(Not primary or Foreign key) attribute in table may not be functionally dependent on another non-key attribute in the same table. In other word, a non-key attribute cannot depend upon one another

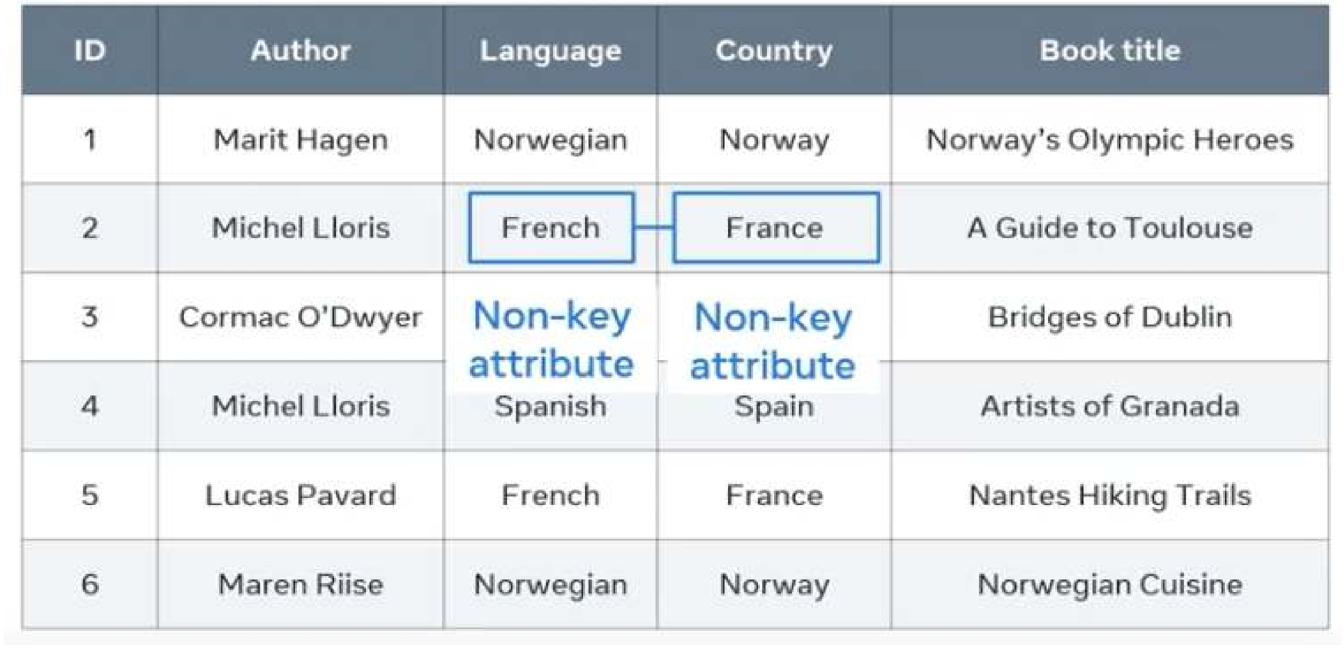
In the below figure ID is the only key or primary key that exists in the table. All other attributes are non-key attributes.



ID	Author	Language	Country	Book title
1	Marit Hagen	Norwegian	Norway	Norway's Olympic Heroes
2	Michel Lloris	French	France	A Guide to Toulouse
3	Cormac O'Dwyer	Irish	Ireland	Bridges of Dublin
4	Michel Lloris	Spanish	Spain	Artists of Granada
5	Lucas Pavard	French	France	Nantes Hiking Trails
6	Maren Riise	Norwegian	Norway	Norwegian Cuisine

Loading [MathJax]/extensions/Safe.js

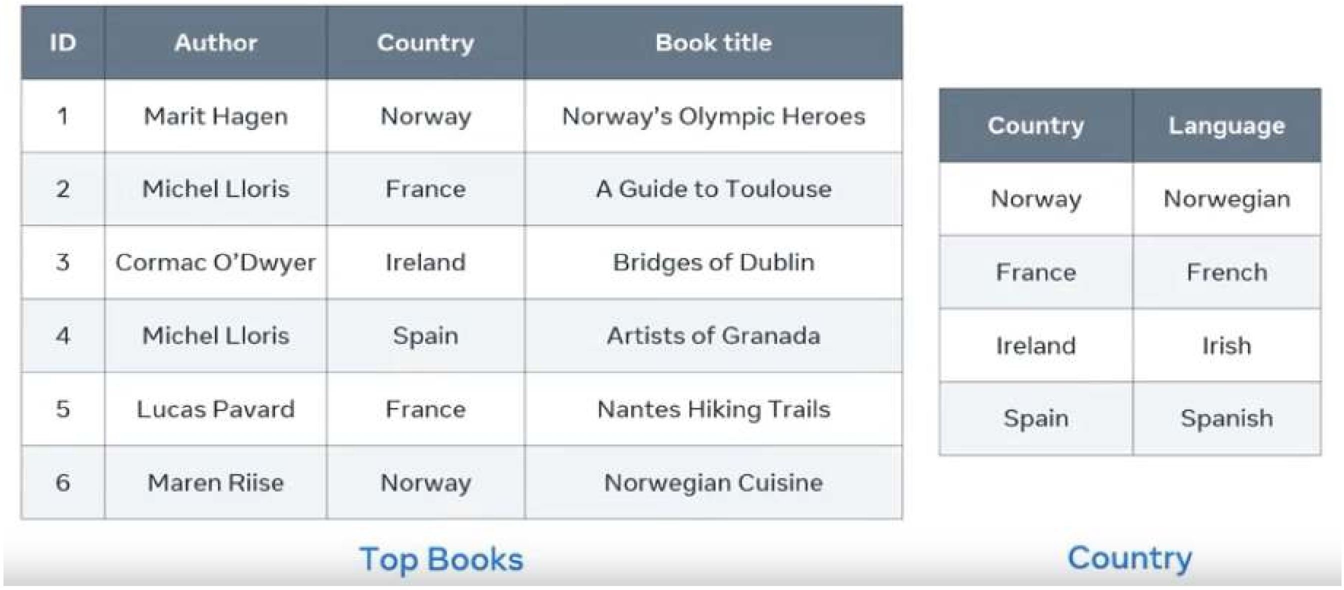
it's always possible to determine the country is France, if the language is French, and vice versa. This means that I have a transitive dependency in this relation. A non-key attribute depends on another non-key attribute.



ID	Author	Language	Country	Book title
1	Marit Hagen	Norwegian	Norway	Norway's Olympic Heroes
2	Michel Lloris	French	France	A Guide to Toulouse
3	Cormac O'Dwyer			Bridges of Dublin
4	Michel Lloris	Spanish	Spain	Artists of Granada
5	Lucas Pavard	French	France	Nantes Hiking Trails
6	Maren Riise	Norwegian	Norway	Norwegian Cuisine

Non-key attribute
Non-key attribute

So we need to solve this by splitting the table into two tables, and link the two tables by a foreign key which it Country column



ID	Author	Country	Book title
1	Marit Hagen	Norway	Norway's Olympic Heroes
2	Michel Lloris	France	A Guide to Toulouse
3	Cormac O'Dwyer	Ireland	Bridges of Dublin
4	Michel Lloris	Spain	Artists of Granada
5	Lucas Pavard	France	Nantes Hiking Trails
6	Maren Riise	Norway	Norwegian Cuisine

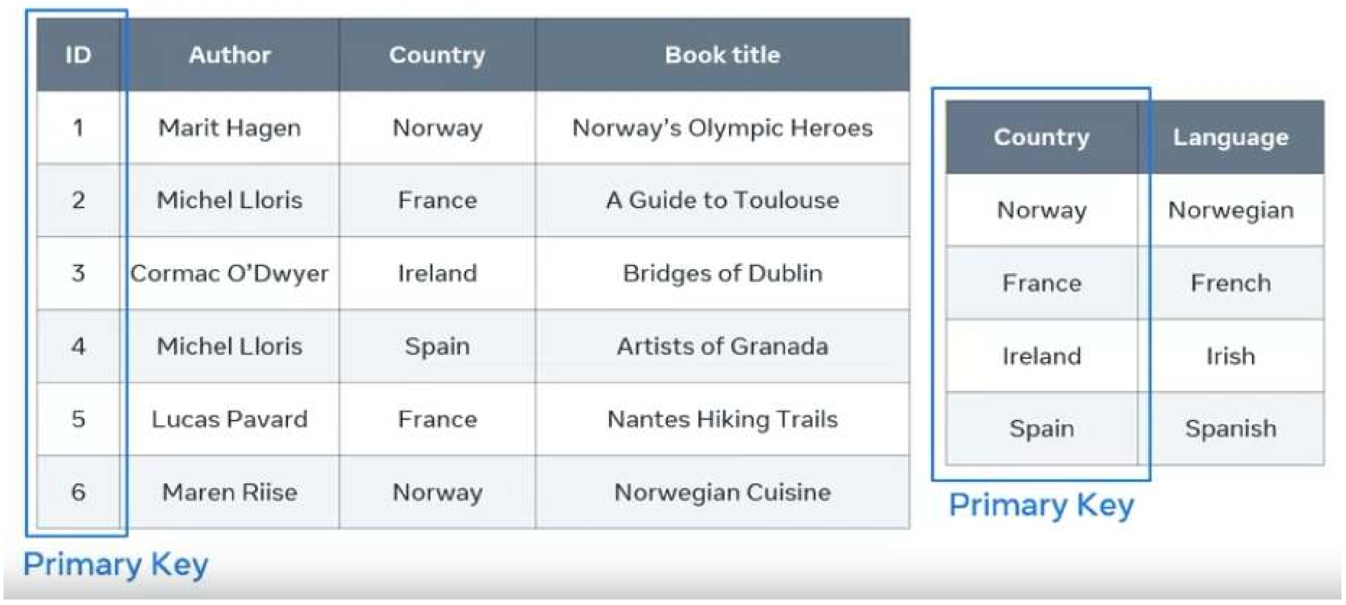
Top Books

Country	Language
Norway	Norwegian
France	French
Ireland	Irish
Spain	Spanish

Country

Loading [MathJax]/extensions/Safe.js

NOTE each table most have a primary key



ID	Author	Country	Book title
1	Marit Hagen	Norway	Norway's Olympic Heroes
2	Michel Lloris	France	A Guide to Toulouse
3	Cormac O'Dwyer	Ireland	Bridges of Dublin
4	Michel Lloris	Spain	Artists of Granada
5	Lucas Pavard	France	Nantes Hiking Trails
6	Maren Riise	Norway	Norwegian Cuisine

Primary Key

Country	Language
Norway	Norwegian
France	French
Ireland	Irish
Spain	Spanish

Primary Key

---

3. DATABASE STRUCTURE AND MANAGEMENT WITH MYSQL

Introduction to MySQL

Filtering data

Logical operation as specifiy multiple conditions

Here's an explanation of logical operations including AND, OR, NOT, IN, BETWEEN, LIKE, and REGEXP, along with examples in MySQL:

1. **AND**: The AND operator is used to combine multiple conditions, and it returns true only if all the conditions are true.

SELECT * FROM employees WHERE age > 30 AND salary > 50000;

This query selects all employees whose age is greater than 30 and salary is greater than 50000.

1. **OR**: The OR operator is used to combine multiple conditions, and it returns true if at least one of the conditions is true.

SELECT * FROM employees WHERE department = 'HR' OR department = 'Finance';

This query selects all employees from the HR department or Finance department.

1. **NOT**: The NOT operator is used to negate a condition or expression, returning the opposite boolean value.

Loading [MathJax]/extensions/Safe.js

SELECT * FROM employees WHERE NOT department = 'Sales';

1. **IN**: The IN operator is used to match a value against a list of specified values.

SELECT * FROM employees WHERE department IN ('Sales', 'Marketing');

This query selects all employees from the Sales or Marketing department.

1. **BETWEEN**: The BETWEEN operator is used to match a value within a specified range.

SELECT * FROM employees WHERE age BETWEEN 25 AND 35;

This query selects all employees whose age is between 25 and 35 (inclusive).

1. **LIKE**: The LIKE operator is used for pattern matching using wildcard characters (% and _).

-- WHRER last_name LIKE 'b%'.... here is search for any last_name that start with b -- WHRER last_name LIKE '%b'.... here is search for any last_name that end with b -- WHRER last_name LIKE '%b%'.... here is search for any last_name that have b -- WHERE last_name LIKE 'b____y'...That means sdearch for any last name that begin whith b and after b we have exactly 4 char followed by y --WHERE EmployeeName LIKE 'a%i' -- Finds EmployeeName values that start with "a" and end with "i". Matches with the record that has an ID of 2.

1. **REGEXP**: The REGEXP operator is used for pattern matching using regular expressions.(same as LIKE but with more usefull features such as: ^ = start , $ = end)

-- WHERE last_name REGEXP 'field|mac|rose' ...here we want to search about any last name that have feild or mac or rose -- WHERE last_name REGEXP '^field|mac|rose' ...here we want to search about any last name that either (start with feild) or have mac or have rose -- WHERE last_name REGEXP 'field$|mac|rose' ...here we want to search about any last name that either (end with feild) or have mac or have rose -- WHERE last_name REGEXP '[gim]e'... here we search for any last name that have g or i or m before e. ex: ge , ie , me -- WHERE last_name REGEXP 'e[gim]'... here we search for any last name that have g or i or m after e. ex: eg , ei , em -- WHERE last_name REGEXP '[a-h]e'... here we search for any last name that have char from a to h before e ex: he, ce, ae, ee, ge ...

EXAMPLE:

CREATE DATABASE luckyshrub_db; USE luckyshrub_db; CREATE TABLE employees (EmployeeID int NOT NULL, EmployeeName varchar(150) DEFAULT NULL, Department varchar(150) DEFAULT NULL, ContactNo varchar(12) DEFAULT NULL, Email varchar(100) DEFAULT NULL, AnnualSalary int DEFAULT NULL, PRIMARY KEY (EmployeeID)); INSERT INTO employees VALUES (1,'Seamus Hogan', 'Recruitment', '351478025', 'Seamus.h@luckyshrub.com',50000), (2,'Thomas Eriksson', 'Legal', '351475058', 'Thomas.e@ luckyshrub.com',75000), (3,'Simon Tolo', 'Marketing', '351930582','Simon.t@ luckyshrub.com',40000), (4,'Francesca Soffia', 'Finance', '351258569','Francesca.s@ luckyshrub.com',45000), (5,'Emily Sierra', 'Customer Service', '351083098','Emily.s@ luckyshrub.com',35000), (6,'Maria Carter', 'Human Resources', '351022508','Maria.c@ luckyshrub.com',55000), (7,'Rick Griffin', 'Marketing', '351478458','Rick.G@luckyshrub.com',50000); -- Task 1: Use the AND operator to find employees who earn an annual salary of $50,000 or more attached to the Marketing department. SELECT * FROM employees WHERE AnnualSalary >= 50000 AND Department = 'Marketing'; -- Task 2: Use the NOT operator to find employees not earning over $50,000 across all departments. SELECT * FROM employees WHERE NOT AnnualSalary >= 50000; -- Task 3: Use the IN operator to find Marketing, Finance, and Legal employees whose annual salary is below $50,000. SELECT * FROM employees WHERE AnnualSalary < 50000 AND Department IN('Marketing', 'Finance', 'Legal'); -- Task 4: Use the BETWEEN operator to find employees who earn annual salaries between $10,000 and $50,000. SELECT * FROM employees WHERE AnnualSalary BETWEEN 10000 AND 50000; -- Task 5: Use the LIKE operator to find employees whose names start with ‘S’ and are at least 4 characters in length. SELECT * FROM employees WHERE EmployeeName LIKE 'S___%';

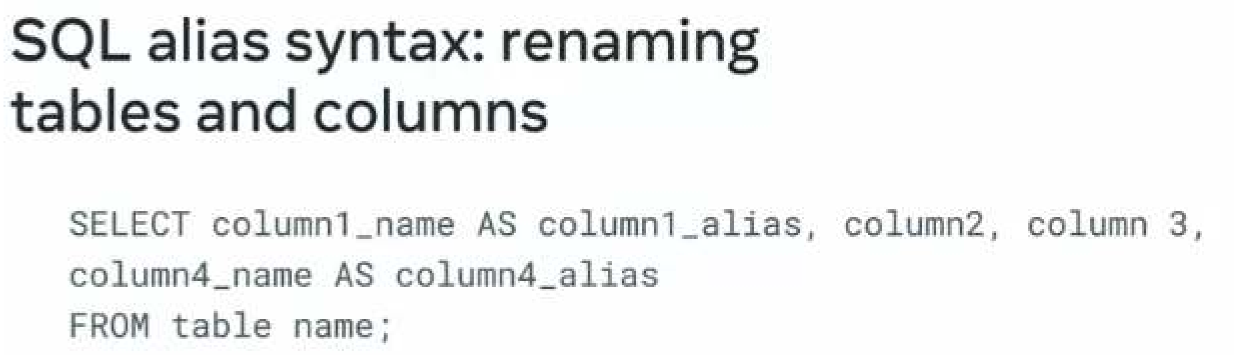
Joining Table

Loading [MathJax]/extensions/Safe.js

MySQL aliases

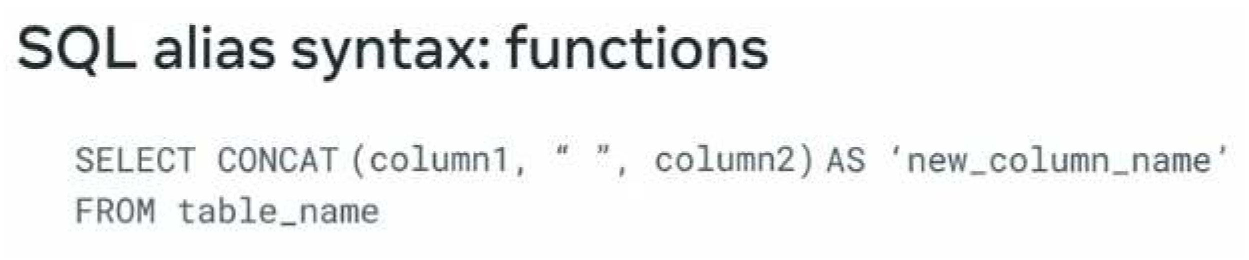
SQL aliases: Provide temporary names within the database

An aliase can rename a tables or columns, also it can be use with a concatenation function to combine an output into one column instead of two and you can use an alianse to create distinct table names when dealing with multiple tables



SQL alias syntax: renaming tables and columns

```
SELECT column1_name AS column1_alias, column2, column 3,
column4_name AS column4_alias
FROM table name;
```

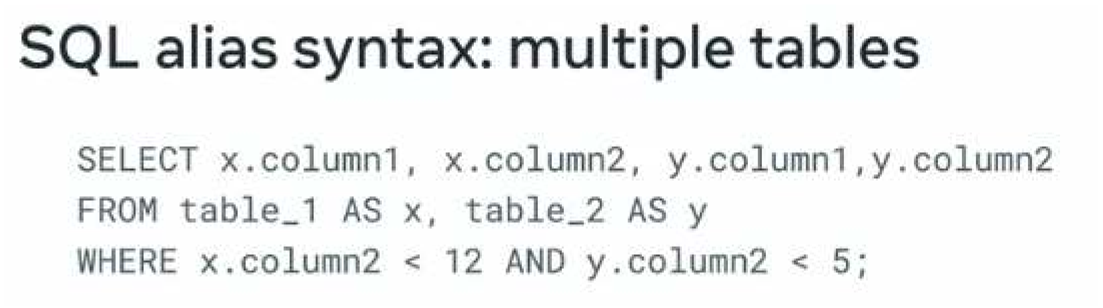


SQL alias syntax: functions

```
SELECT CONCAT(column1, " ", column2) AS 'new_column_name'
FROM table_name
```

IT combined the columns information into one column and by using AS we select a new name

In concat the columns must be between a quotaton " "



SQL alias syntax: multiple tables

```
SELECT x.column1, x.column2, y.column1,y.column2
FROM table_1 AS x, table_2 AS y
WHERE x.column2 < 12 AND y.column2 < 5;
```

JOIN

Join: A join in a database links records of data betwenn one or multiple tables on a common column between them.

There are four different types of joins supported in MySQL that are covered in this lesson.

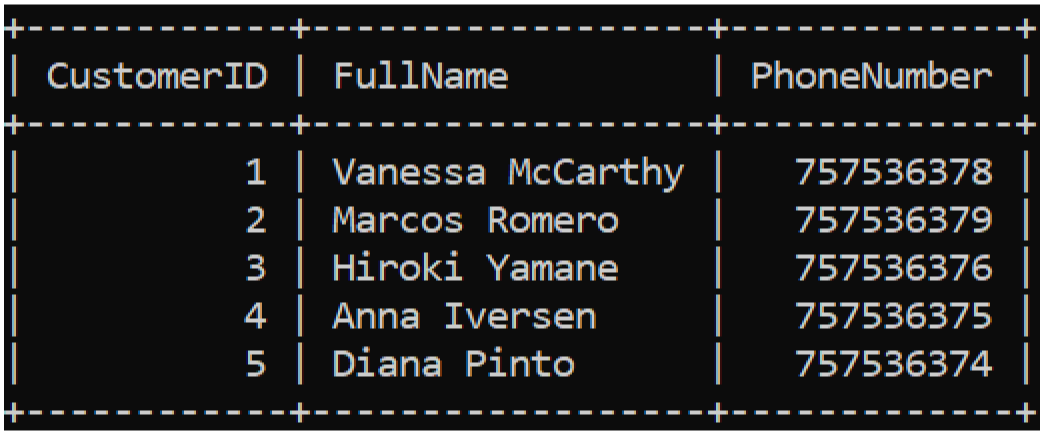
- INNER JOIN
- LEFT JOIN
- RIGHT JOIN
- SELF JOIN

Loading [MathJax]/extensions/Safe.js

To explain the difference between these types of JOINS, let's look at the Little Lemon restaurant database, which includes two tables.

The first is the Customers table with the following columns:

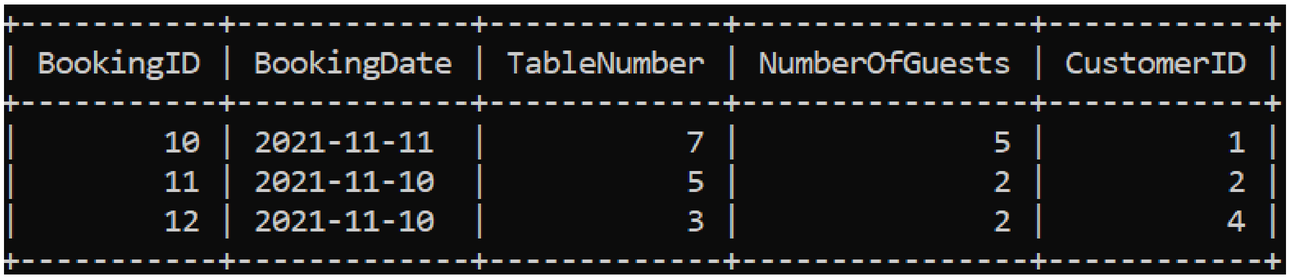
- CustomerID,
- FullName
- and PhoneNumber columns, as shown below:



```
+------------+-----------------+-------------+
| CustomerID | FullName        | PhoneNumber |
+------------+-----------------+-------------+
|          1 | Vanessa McCarthy |   757536378 |
|          2 | Marcos Romero   |   757536379 |
|          3 | Hiroki Yamane   |   757536376 |
|          4 | Anna Iversen    |   757536375 |
|          5 | Diana Pinto     |   757536374 |
+------------+-----------------+-------------+
```

The second is the bookings table with the columns:

- BookingID,
- BookingDate,
- TableNumber,
- NumberOfGuests
- and CustomerID columns.



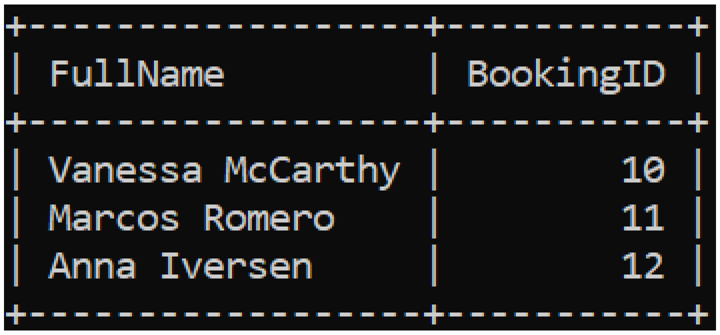
```
+-----------+-------------+-------------+----------------+------------+
| BookingID | BookingDate | TableNumber | NumberOfGuests | CustomerID |
+-----------+-------------+-------------+----------------+------------+
|        10 | 2021-11-11  |           7 |              5 |          1 |
|        11 | 2021-11-10  |           5 |              2 |          2 |
|        12 | 2021-11-10  |           3 |              2 |          4 |
+-----------+-------------+-------------+----------------+------------+
```

INNER JOIN

This type of JOIN returns records of data that have matching values in the joined tables. For example, assume that you want to return the full name and booking ID of customers who made bookings. In this situation, you can use the INNER JOIN clause to extract records of data from the Customers and the Bookings tables based on the matching customer ID value as follows.

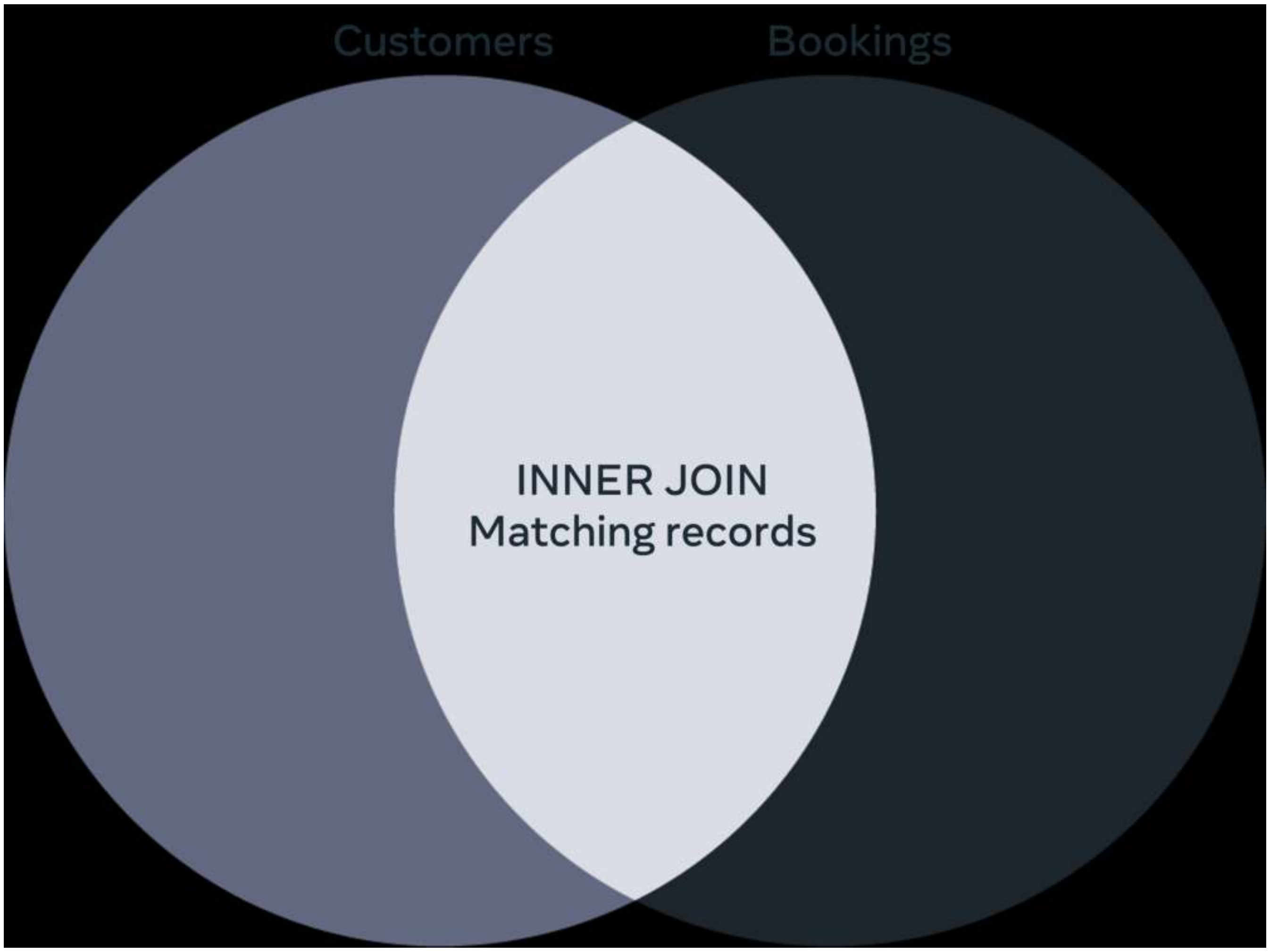
SELECT Customers.FullName, Bookings.BookingID FROM Customers INNER JOIN Bookings ON Customers.CustomerID =

Bookings.CustomerID;

The outuput result shown below 

```
+------------------+-----------+
| FullName         | BookingID |
+------------------+-----------+
| Vanessa McCarthy |        10 |
| Marcos Romero    |        11 |
| Anna Iversen     |        12 |
+------------------+-----------+
```

The INNER JOIN is illustrated in the following Venn diagram.



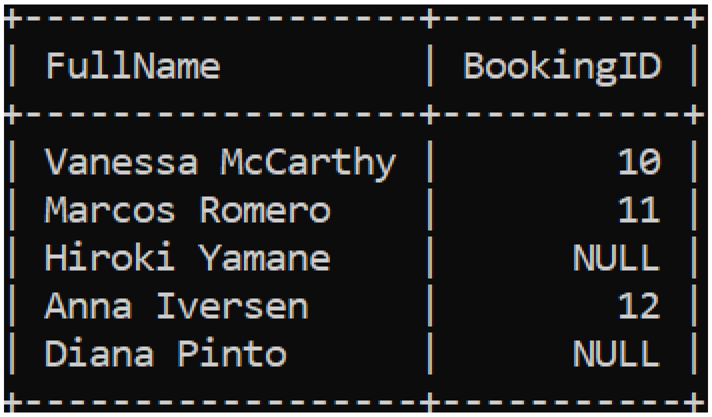
LEFT JOIN

You can use the LEFT JOIN clause to extract the full names and the booking IDs from the Customers and the Bookings tables as follows:

SELECT Customers.FullName, Bookings.BookingID FROM Customers LEFT JOIN Bookings ON Customers.CustomerID =

Loading [MathJax]/extensions/Safe.js

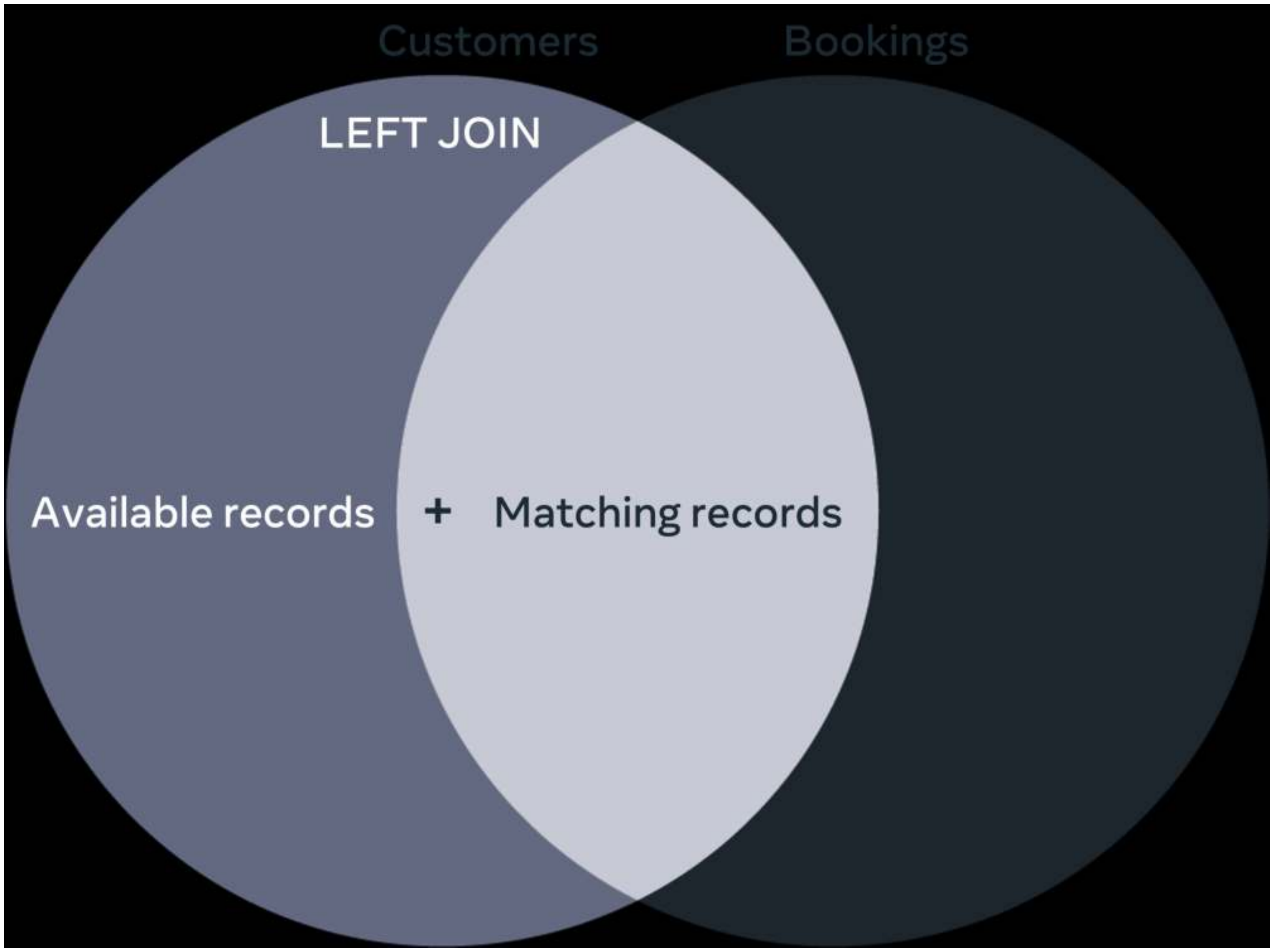
Bookings.CustomerID;

The results of this query are as follows: 

```
+-------------------+-----------+
| FullName          | BookingID |
+-------------------+-----------+
| Vanessa McCarthy  |        10 |
| Marcos Romero     |        11 |
| Hiroki Yamane     |      NULL |
| Anna Iversen      |        12 |
| Diana Pinto       |      NULL |
+-------------------+-----------+
```

The LEFT JOIN returns all common records in a similar way to the INNER JOIN, plus all queried records from the left table regardless of whether there is a match in the right table or not. If there are no matching records in the right table, then null values will be inserted for the bookings IDs.

The LEFT JOIN is illustrated in the following Venn diagram.



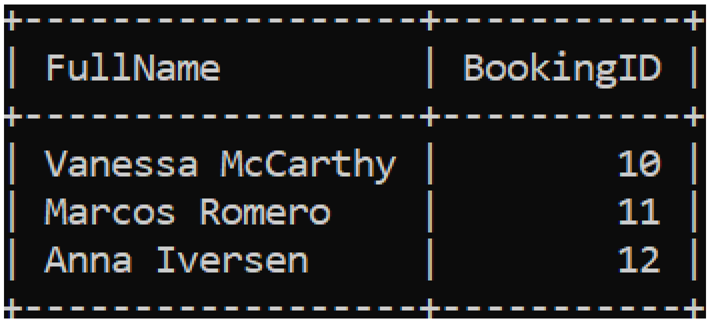
RIGHT JOIN

You can use the RIGHT JOIN clause to extract the full names and the booking IDs from the Customers and the Bookings tables as follows:

SELECT Customers.FullName, Bookings.BookingID FROM Customers RIGHT JOIN Bookings ON Customers.CustomerID =

Loading [MathJax]/extensions/Safe.js

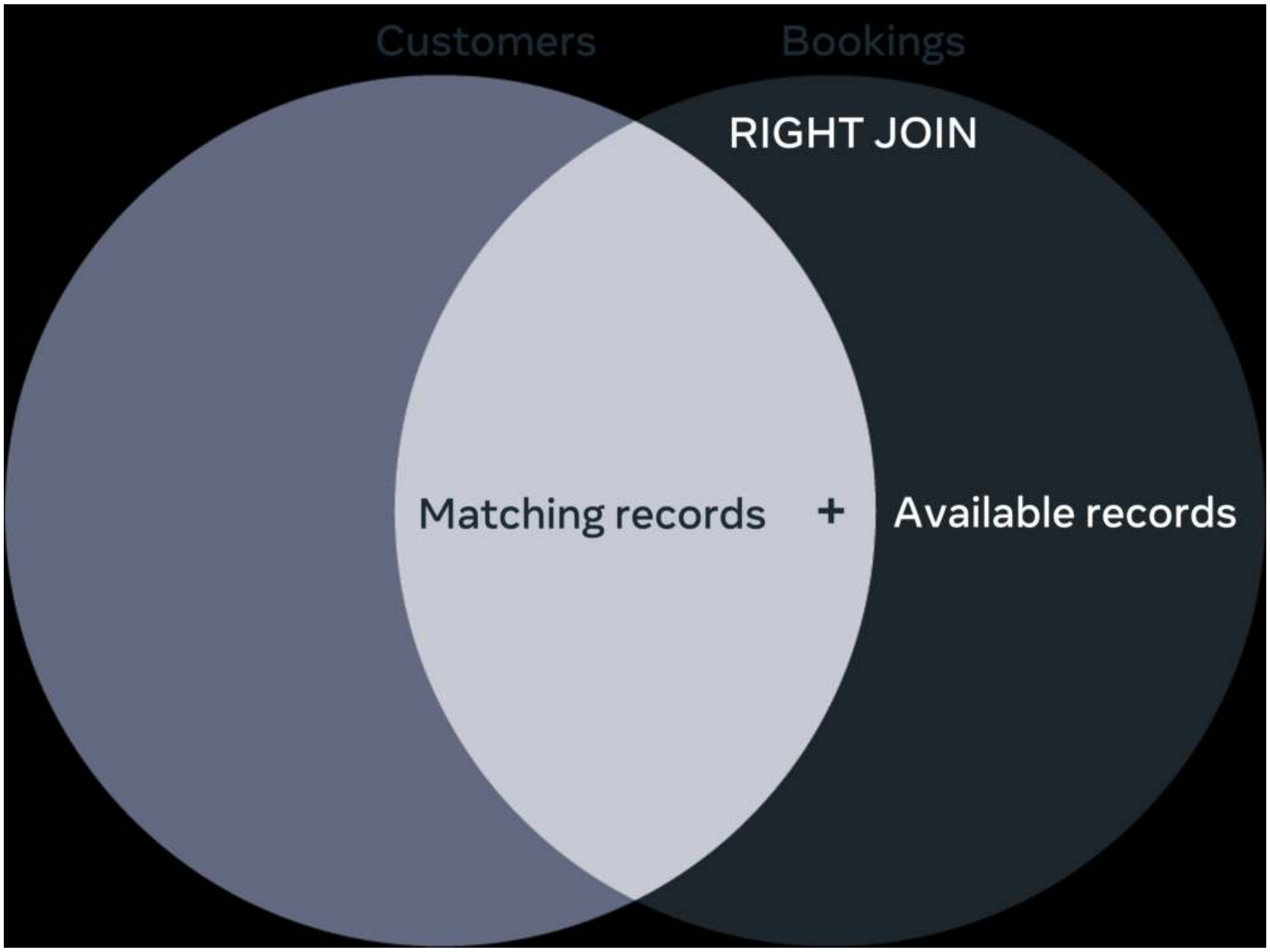
Bookings.CustomerID;

The output of this query is as follows: 

```
+-------------------+-----------+
| FullName          | BookingID |
+-------------------+-----------+
| Vanessa McCarthy  |        10 |
| Marcos Romero     |        11 |
| Anna Iversen      |        12 |
+-------------------+-----------+
```

The RIGHT JOIN returns all common records in a similar way to the INNER JOIN, plus all queried records from the right table regardless of whether there is a match in the left table or not. If there are no matching records in the left table, then null values will be inserted for the customers full names.

The RIGHT JOIN is illustrated in the following Venn diagram.

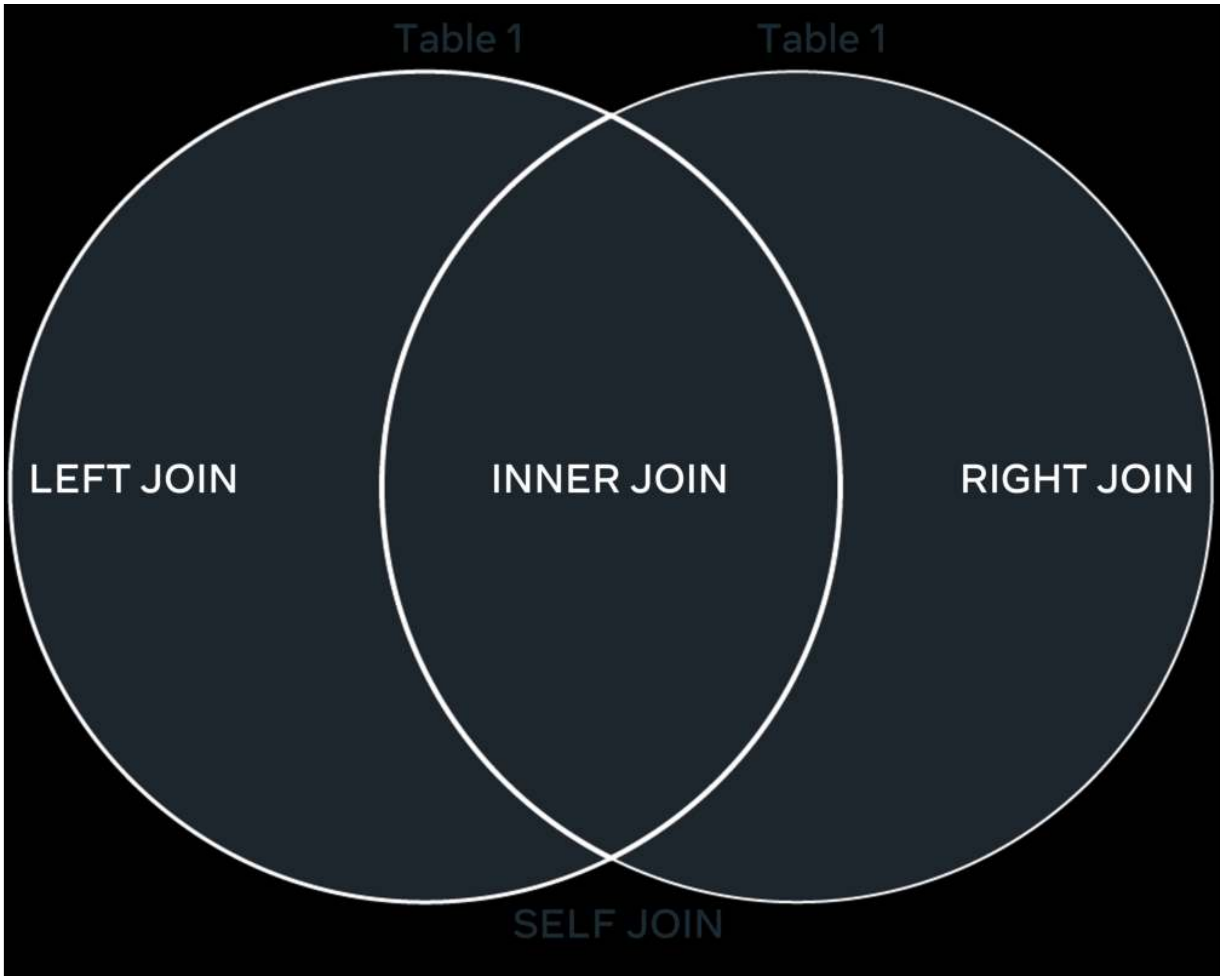


SELF JOIN

This is a special case where you need to join a table with itself to get specific information existing in the same table. In this case you may choose the INNER JOIN, LEFT JOIN or RIGHT JOIN presented earlier to query the required data.

Loading [MathJax]/extensions/Safe.js

The SELF JOIN is illustrated in the following Venn diagram.



UNION operator

Combines results sets from multiple statements in the same query

NOTE !:

1. EVERY SELECT STATEMANT MUST HAVE THE SAME NUMBER OF COLUMNS
2. ALL RELATED COLUMNS MUST HAVE THE SAME DATA TYPE
3. ALL RELATED COLUMNS MUST HAVE THE SAME ORDER

SELECT col1, col2 FROM table1 UNION -- only return distinct values from the target table -- if you want to return all values : UNION ALL: IT UNSURE THAT DUPLICATED VALUES ARE RETAINED IN THE OUTOUT OF A SELECT query TAHT TARGET TWO SEPARATE TABLES SELECT col1, col2 FROM table2;

Grouping Data

GROUP BY

Groups rows in a table based on given columns, into summary rows (Distinct value), or subgroups

It also fequently used with aggregate functions

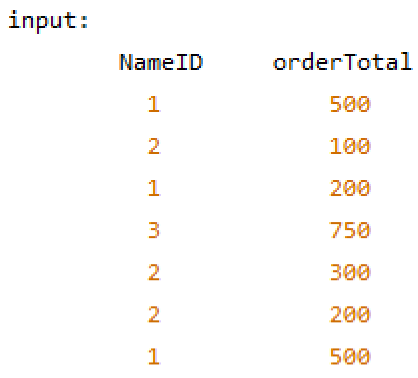
Loading [MathJax]/extensions/Safe.js WHERE filter_condition -- WHERE must be placed before GROUP BY clause GROUP BY col1, col2;

Aggregate functions:

1. SUM(): USED TO ADD VALUES OF GIVEN COLUMNS TOGETHER AND RETURN A SINGLE VALUE
2. AVERAGE(): USED TO DETERMINE THE AVERAGE OF COLUMN VALUES
3. MAX(): WHICH RETURNS THE MAXIMUM VALUE OF ONE OR MORE GIVEN COLUMNS
4. MIN(): DETERMINES THE MINIMUM VALUE OF ONE OR MORE GIVEN COLUMNS
5. COUNT(): USED TO COUNT THE NUMBER OF INSTANCES THAT A GIVEN COLUMN VALUE OCCURS

group by syntax with aggregate functions:

SELECT col1, col2, col3, MAX(col1) FROM tbl GROUP BY col1, col2, col3;

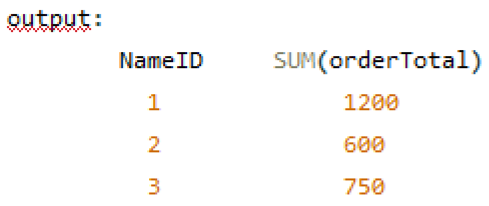


input:

NameID	orderTotal
1	500
2	100
1	200
3	750
2	300
2	200
1	500

EXAMPLE:

SELECT NameID, SUM(orderTotal) FROM orders GROUP BY Department;



output:

NameID	SUM(orderTotal)
1	1200
2	600
3	750

HAVING

Specifies a filter condition for groups of rows or aggregates

HAVING is used to filter the data of the group by bcuz WHERE cannot filter it

In the absence of GROUP BY clause, the HAVING clause behave just like the WHERE clause

SELECT Deparment, SUM(orderTotal) AS total -- it is better to use alias FROM orders GROUP BY Department HAVING total > 2000;

exercise: Operators and clauses

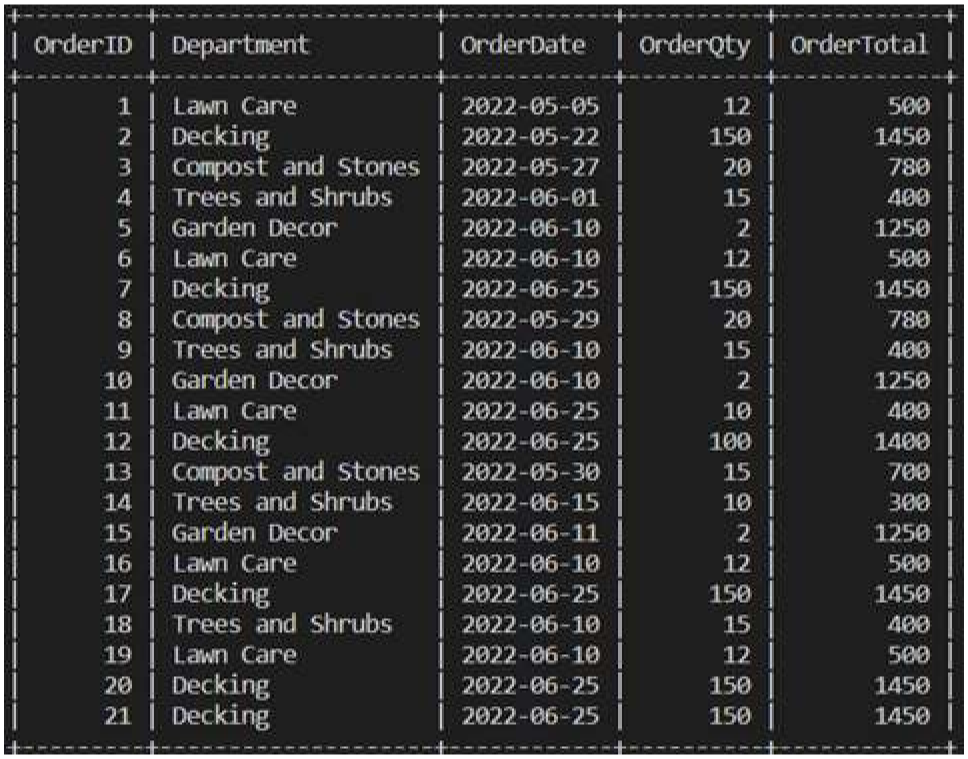
Lucky Shrub gardening center's database stores data on their employees, customer orders and the details of orders handled by employees.

CREATE DATABASE luckyshrub_db; USE luckyshrub_db; CREATE TABLE employees (EmployeeID int NOT NULL, EmployeeName varchar(150) DEFAULT NULL, Department varchar(150) DEFAULT NULL, ContactNo varchar(12) DEFAULT NULL, Email varchar(100) DEFAULT NULL, AnnualSalary int DEFAULT NULL, PRIMARY KEY (EmployeeID)); CREATE TABLE orders (OrderID int NOT NULL, Department varchar(100) DEFAULT NULL, OrderDate date DEFAULT NULL, OrderQty int DEFAULT NULL, OrderTotal int DEFAULT NULL, PRIMARY KEY (OrderID)); CREATE TABLE employee_orders

loyeeID int NOT NULL, Status VARCHAR(150), HandlingCost int DEFAULT NULL, PRIMARY

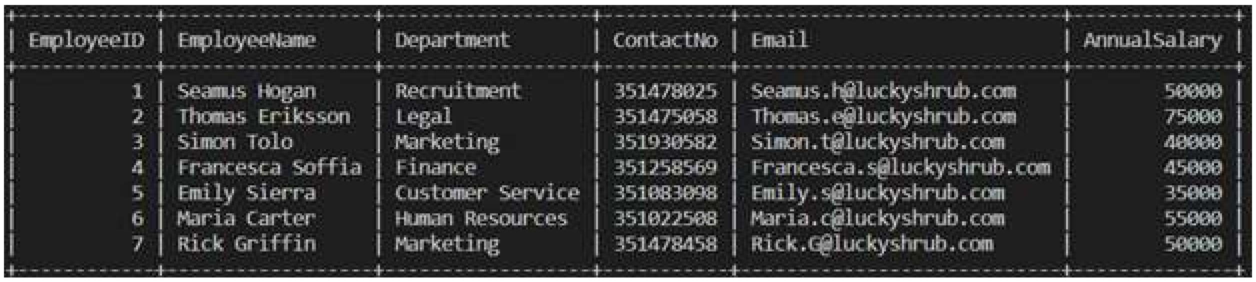
```
KEY (EmployeeID,OrderID), FOREIGN KEY (EmployeeID) REFERENCES employees(EmployeeID), FOREIGN KEY
(OrderID) REFERENCES orders(OrderID) ); INSERT INTO employees VALUES (1,'Seamus Hogan', 'Recruitment',
'351478025', 'Seamus.h@luckyshrub.com',50000), (2,'Thomas Eriksson', 'Legal', '351475058',
'Thomas.e@luckyshrub.com',75000), (3,'Simon Tolo', 'Marketing', '351930582','Simon.t@luckyshrub.com',40000),
(4,'Francesca Soffia', 'Finance', '351258569','Francesca.s@luckyshrub.com',45000), (5,'Emily Sierra', 'Customer Service',
'351083098','Emily.s@luckyshrub.com',35000), (6,'Maria Carter', 'Human Resources',
'351022508','Maria.c@luckyshrub.com',55000), (7,'Rick Griffin', 'Marketing', '351478458','Rick.G@luckyshrub.com',50000);
INSERT INTO orders VALUES(1,'Lawn Care','2022-05-05',12,500), (2,'Decking','2022-05-22',150,1450), (3,'Compost and
Stones','2022-05-27',20,780), (4,'Trees and Shrubs','2022-06-01',15,400), (5,'Garden Decor','2022-06-10',2,1250), (6,'Lawn
Care','2022-06-10',12,500), (7,'Decking','2022-06-25',150,1450), (8,'Compost and Stones','2022-05-29',20,780), (9,'Trees and
Shrubs','2022-06-10',15,400), (10,'Garden Decor','2022-06-10',2,1250), (11,'Lawn Care','2022-06-25',10,400),
(12,'Decking','2022-06-25',100,1400), (13,'Compost and Stones','2022-05-30',15,700), (14,'Trees and Shrubs','2022-06-
15',10,300), (15,'Garden Decor','2022-06-11',2,1250), (16,'Lawn Care','2022-06-10',12,500), (17,'Decking','2022-06-
25',150,1450), (18,'Trees and Shrubs','2022-06-10',15,400), (19,'Lawn Care','2022-06-10',12,500), (20,'Decking','2022-06-
25',150,1450), (21,'Decking','2022-06-25',150,1450); INSERT INTO employee_orders VALUES(1,3,"In Progress",200),
(1,5,"Not Recieved",300), (1,4,"Not Recieved",250), (2,3,"Completed",200), (2,5,"Completed",300), (2,4,"In Progress",250),
(3,3,"In Progress",200), (3,5,"Not Recieved",300), (3,4,"Not Recieved",250), (4,3,"Completed",200), (4,5,"In Progress",300),
(4,4,"In Progress",250), (5,3,"Completed",200), (5,5,"In Progress",300), (5,4,"Not Recieved",250), (11,3,"Completed",200),
(11,5,"Completed",300), (11,4,"Not Recieved",250), (14,3,"Completed",200), (14,5,"Not Recieved",300), (14,4,"Not
Recieved",250);
```

The Lucky Shrub database contains a table called orders that stores data about orders placed by different departments.



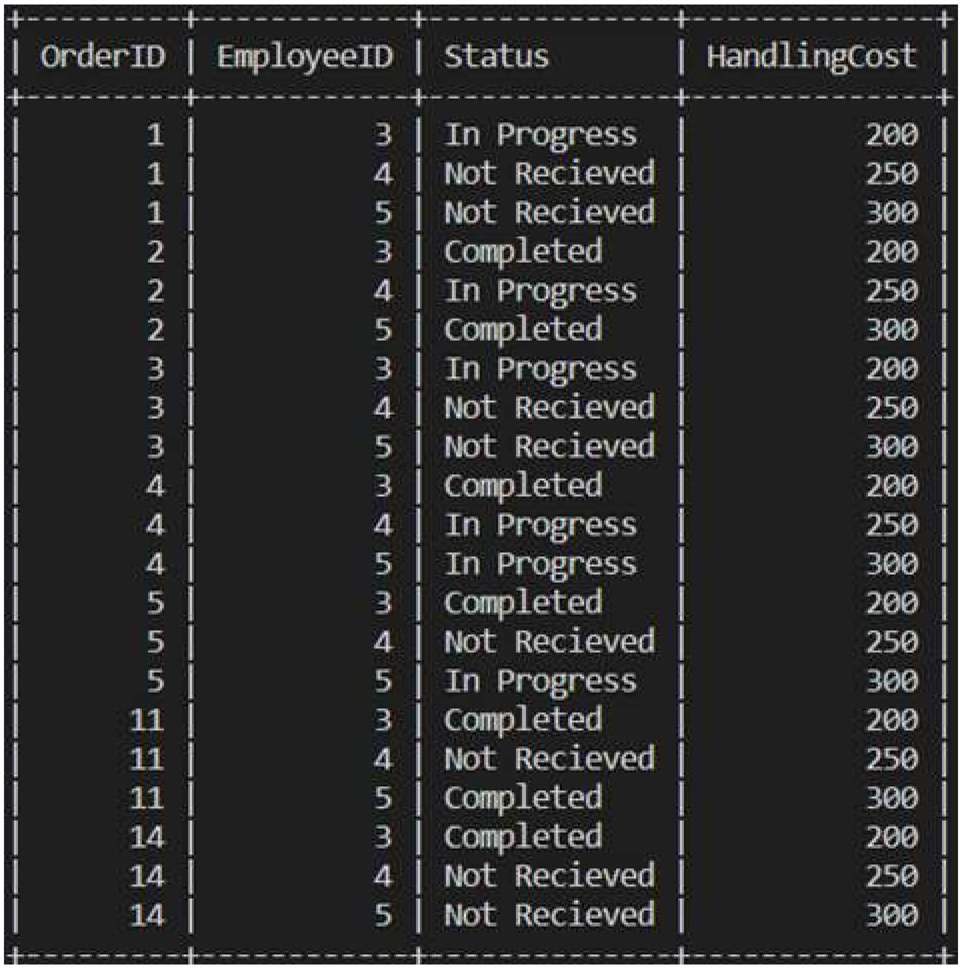
```
+---------+--------------------+------------+----------+------------+
| OrderID | Department         | OrderDate  | OrderQty | OrderTotal |
+---------+--------------------+------------+----------+------------+
|       1 | Lawn Care          | 2022-05-05 |       12 |        500 |
|       2 | Decking            | 2022-05-22 |      150 |       1450 |
|       3 | Compost and Stones | 2022-05-27 |       20 |        780 |
|       4 | Trees and Shrubs   | 2022-06-01 |       15 |        400 |
|       5 | Garden Decor       | 2022-06-10 |        2 |       1250 |
|       6 | Lawn Care          | 2022-06-10 |       12 |        500 |
|       7 | Decking            | 2022-06-25 |      150 |       1450 |
|       8 | Compost and Stones | 2022-05-29 |       20 |        780 |
|       9 | Trees and Shrubs   | 2022-06-10 |       15 |        400 |
|      10 | Garden Decor       | 2022-06-10 |        2 |       1250 |
|      11 | Lawn Care          | 2022-06-25 |       10 |        400 |
|      12 | Decking            | 2022-06-25 |      100 |       1400 |
|      13 | Compost and Stones | 2022-05-30 |       15 |        700 |
|      14 | Trees and Shrubs   | 2022-06-15 |       10 |        300 |
|      15 | Garden Decor       | 2022-06-11 |        2 |       1250 |
|      16 | Lawn Care          | 2022-06-10 |       12 |        500 |
|      17 | Decking            | 2022-06-25 |      150 |       1450 |
|      18 | Trees and Shrubs   | 2022-06-10 |       15 |        400 |
|      19 | Lawn Care          | 2022-06-10 |       12 |        500 |
|      20 | Decking            | 2022-06-25 |      150 |       1450 |
|      21 | Decking            | 2022-06-25 |      150 |       1450 |
+---------+--------------------+------------+----------+------------+
```

There’s also the employees table that stores data about Lucky Shrub’s employees.



```
+------------+------------------+------------------+-----------+----------------------------+--------------+
| EmployeeID | EmployeeName     | Department       | ContactNo | Email                      | AnnualSalary |
+------------+------------------+------------------+-----------+----------------------------+--------------+
|          1 | Seamus Hogan     | Recruitment      | 351478025 | Seamus.h@luckyshrub.com    |        50000 |
|          2 | Thomas Eriksson  | Legal            | 351475058 | Thomas.e@luckyshrub.com    |        75000 |
|          3 | Simon Tolo       | Marketing        | 351930582 | Simon.t@luckyshrub.com     |        40000 |
|          4 | Francesca Soffia | Finance          | 351258569 | Francesca.s@luckyshrub.com |        45000 |
|          5 | Emily Sierra     | Customer Service | 351083098 | Emily.s@luckyshrub.com     |        35000 |
|          6 | Maria Carter     | Human Resources  | 351022508 | Maria.c@luckyshrub.com     |        55000 |
|          7 | Rick Griffin     | Marketing        | 351478458 | Rick.G@luckyshrub.com      |        50000 |
+------------+------------------+------------------+-----------+----------------------------+--------------+
```

Finally, there's an employee_orders table that stores data about the employees who are handling the orders.



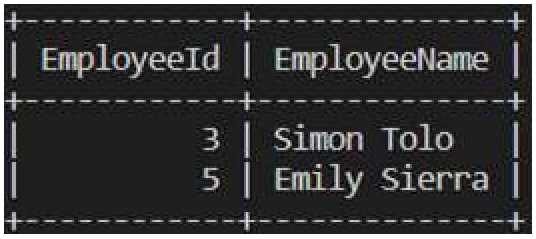
```
+---------+------------+--------------+--------------+
| OrderID | EmployeeID | Status       | HandlingCost |
+---------+------------+--------------+--------------+
|       1 |          3 | In Progress  |          200 |
|       1 |          4 | Not Recieved |          250 |
|       1 |          5 | Not Recieved |          300 |
|       2 |          3 | Completed    |          200 |
|       2 |          4 | In Progress  |          250 |
|       2 |          5 | Completed    |          300 |
|       3 |          3 | In Progress  |          200 |
|       3 |          4 | Not Recieved |          250 |
|       3 |          5 | Not Recieved |          300 |
|       4 |          3 | Completed    |          200 |
|       4 |          4 | In Progress  |          250 |
|       4 |          5 | In Progress  |          300 |
|       5 |          3 | Completed    |          200 |
|       5 |          4 | Not Recieved |          250 |
|       5 |          5 | In Progress  |          300 |
|      11 |          3 | Completed    |          200 |
|      11 |          4 | Not Recieved |          250 |
|      11 |          5 | Completed    |          300 |
|      14 |          3 | Completed    |          200 |
|      14 |          4 | Not Recieved |          250 |
|      14 |          5 | Not Recieved |          300 |
+---------+------------+--------------+--------------+
```

Task 1 solution: Use the ANY operator to identify all employees with the Order Status status 'Completed'.

```
SELECT EmployeeId, EmployeeName  FROM employees  WHERE EmployeeID = ANY (SELECT EmployeeID FROM employee_orders WHERE Status='Completed');
```

This query returns all employee records that have an EmployeeID which matches ANY of the EmployeeIDs returned from the subquery.

There are two employee records in the results set:



```
+------------+--------------+
| EmployeeId | EmployeeName |
+------------+--------------+
|          3 | Simon Tolo   |
|          5 | Emily Sierra |
+------------+--------------+
```

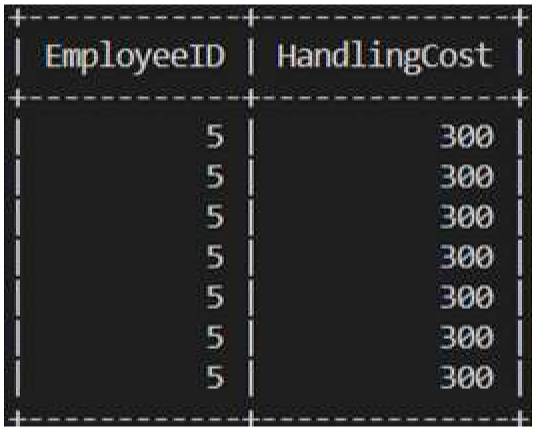
Task 2 solution: Use the ALL operator to identify the IDs of employees who earned a handling cost of "more than 20% of the order value" from all orders they have handled.

```
SELECT EmployeeID,HandlingCost  FROM employee_orders  WHERE HandlingCost > ALL(SELECT ROUND(OrderTotal/100 * 20) FROM orders);
```

This query returns all records from the employee_orders table that have a handling cost more than ALL the 20% values returned from the subquery.

Loading [MathJax]/extensions/Safe.js

The result that's returned by this query contains seven records, all from the same employee ID:

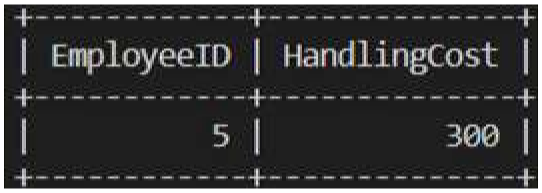

```
+------------+--------------+
| EmployeeID | HandlingCost |
+------------+--------------+
|          5 |          300 |
|          5 |          300 |
|          5 |          300 |
|          5 |          300 |
|          5 |          300 |
|          5 |          300 |
|          5 |          300 |
+------------+--------------+
```
The same employee ID is repeated here because this employee has handled 7 orders. For all those orders, the employee has earned a handling cost of $300.

Task 3 solution: Use the GROUP BY clause to summarize the duplicate records with the same column values into a single record by grouping them based on those columns.

SELECT EmployeeID,HandlingCost   FROM employee_orders   WHERE HandlingCost > ALL(SELECT ROUND(OrderTotal/100 * 20) FROM orders) GROUP BY EmployeeID,HandlingCost;

Note that the data has been grouped by the two columns that have the same (or duplicate) values, which are EmployeeID and HandlingCost.

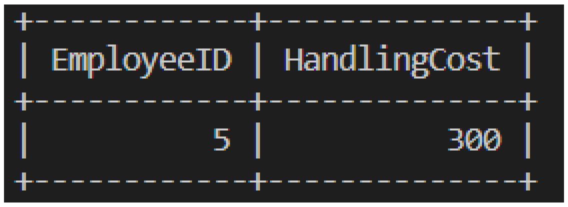
The result of this query contains one record:


```
+------------+--------------+
| EmployeeID | HandlingCost |
+------------+--------------+
|          5 |          300 |
+------------+--------------+
```
The data now has been summarized to a single record.

Task 4 solution: Use the HAVING clause to filter the grouped data in the subquery that you wrote in task 3 to filter the 20% OrderTotal values to only retrieve values that are more than $100.

This query is written as follows:

SELECT EmployeeID,HandlingCost FROM employee_orders WHERE HandlingCost > ALL(SELECT ROUND(OrderTotal/100 * 20) AS twentyPercent FROM orders GROUP BY OrderTotal HAVING twentyPercent > 100) GROUP BY EmployeeID,HandlingCost;


```
+------------+--------------+
| EmployeeID | HandlingCost |
+------------+--------------+
|          5 |          300 |
+------------+--------------+
```

Reference sheet: Operators and clauses

GROUP BY clause

Use the GROUP BY clause in a SELECT statement to group rows in a table(s) based on a given column(s) into summary rows or subgroups.

It is placed after the FROM clause. If there is a WHERE clause in your SELECT statement, it should be placed after the WHERE clause. After the GROUP BY keyword, place a list of comma-separated column names by which you want to group the data.

HAVING clause

If you also want to filter your grouped data, use the HAVING clause. You should be aware that the WHERE clause cannot filter grouped data. The HAVING clause should appear after the GROUP BY clause. In the HAVING clause, you can specify the filter condition(s) that needs to be applied to your grouped data.

ANY operator

The ANY operator lets you perform a comparison between a single column value and a range of other values. The range of values comes from the execution of a subquery.

The syntax of a statement that uses an ANY operator is as follows:

SELECT column_name(s) FROM table_name WHERE column_name comparison operator ANY   (SELECT column_name FROM table_name   WHERE condition);

The ANY operator returns a boolean value following a comparison operation. It returns a TRUE value if ANY subquery values meet the given condition. In other words, the condition will be TRUE if the operation is true for any of the values in the range.

In this syntax, the ANY operator should be preceded by a column name and a comparison operator that operates on the column name against the set of values.

Standard comparison operators like =, <>, !=, >, >=, <, or <= can be used here.

ALL operator

Use the ALL operator for the same purpose as the ANY operator. However, the way it works is a little bit different. It returns a boolean value as a result of performing a comparison operation. It returns TRUE only if ALL subquery values meet the given condition. In other words, the condition will be TRUE only if the operation is true for all values in the range.

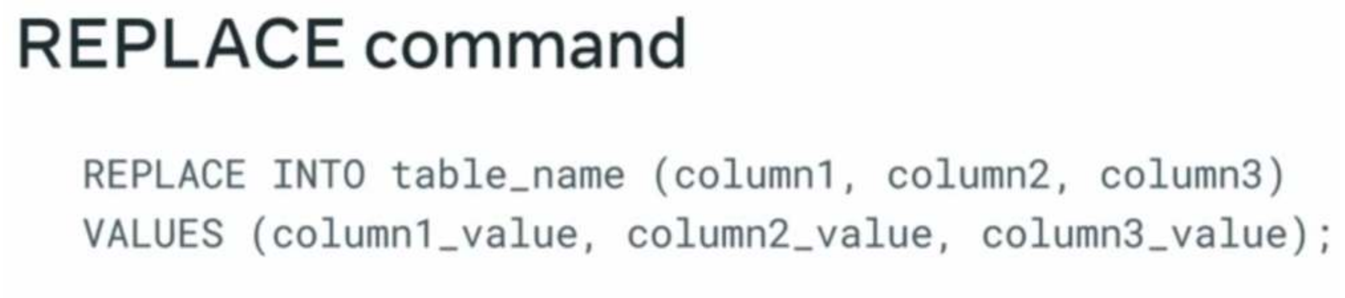
The syntax for the ALL operator is as follows:

SELECT column_name(s) FROM table_name WHERE column_name operator ALL   (SELECT column_name FROM table_name WHERE condition);

Updating Databases and Working with Views

MySQL REPLACE statement

REPLACE command: Insert or update data in a table by deleting and replacing existing records



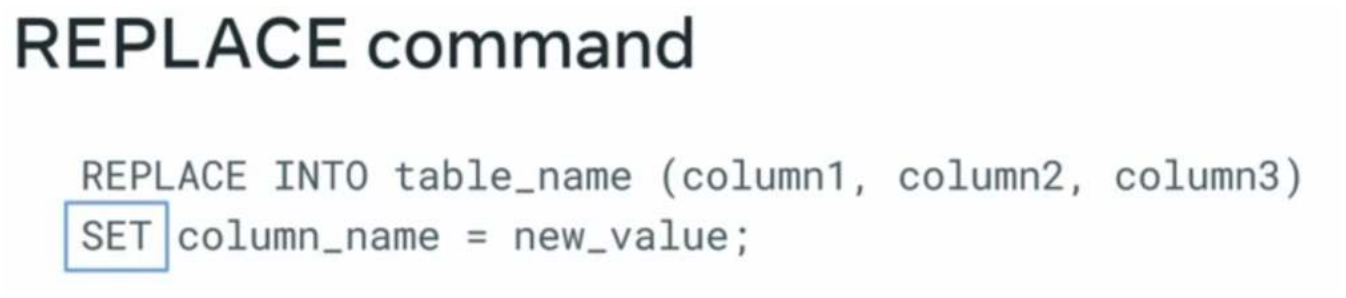
REPLACE command

```
REPLACE INTO table_name (column1, column2, column3)
VALUES (column1_value, column2_value, column3_value);
```

Loading [MathJax]/extensions/Safe.js

It similar to INSERT syntax

Yoe can also use Replace command with the SET keyword



REPLACE command

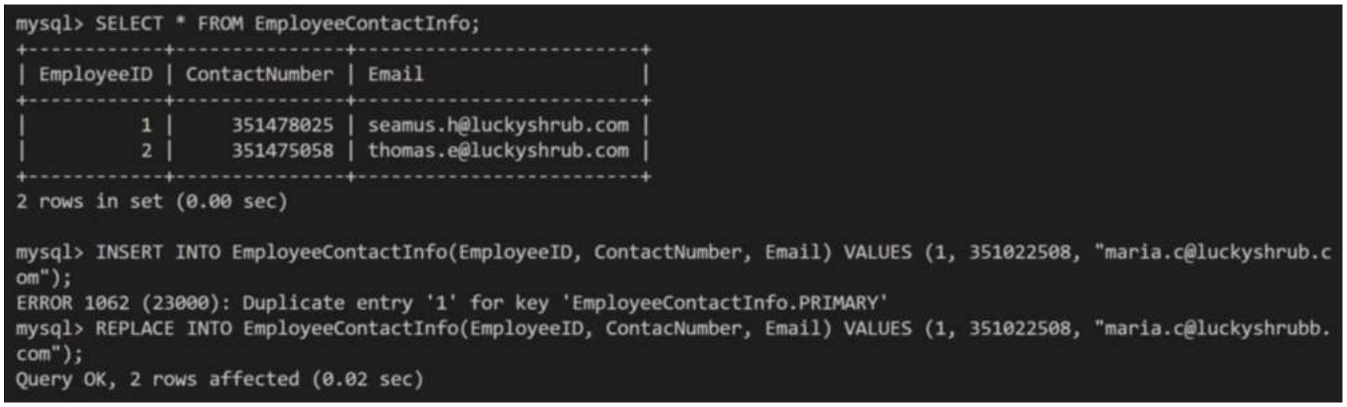
```
REPLACE INTO table_name (column1, column2, column3)
SET column_name = new_value;
```

How the Replace command works?

1. Replace command checks for the primary or UNIQUE key of existing records
2. Replace command adds new data if no matching key is found
3. Replace command Deletes and replaces the existing record if a match is found

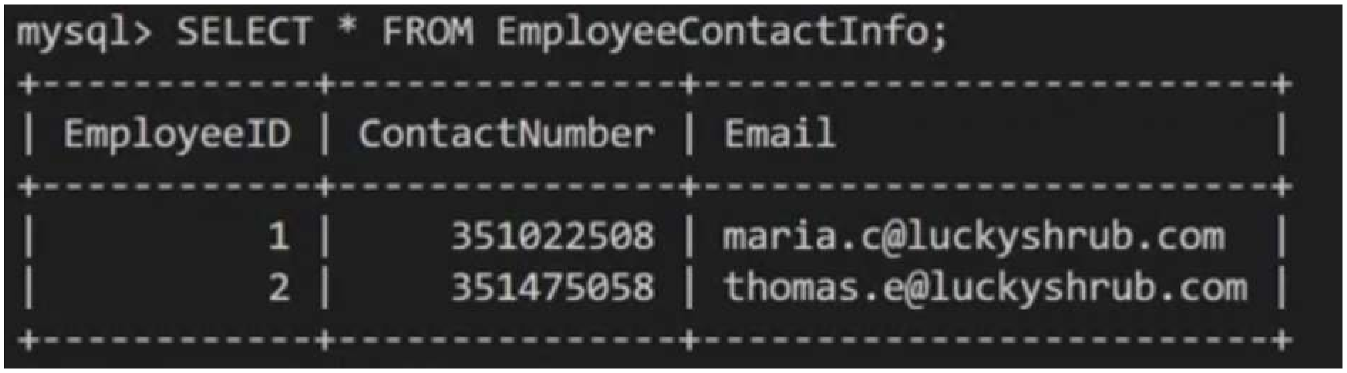
---

As you see below, we got an error message when we tried to change the information of EmloeeID 1 by using INSERT INTO becuz EmployeeID is a primary column that we cannot change its values. Therefore, instead of INSERT we used REPLASE INTO to change it



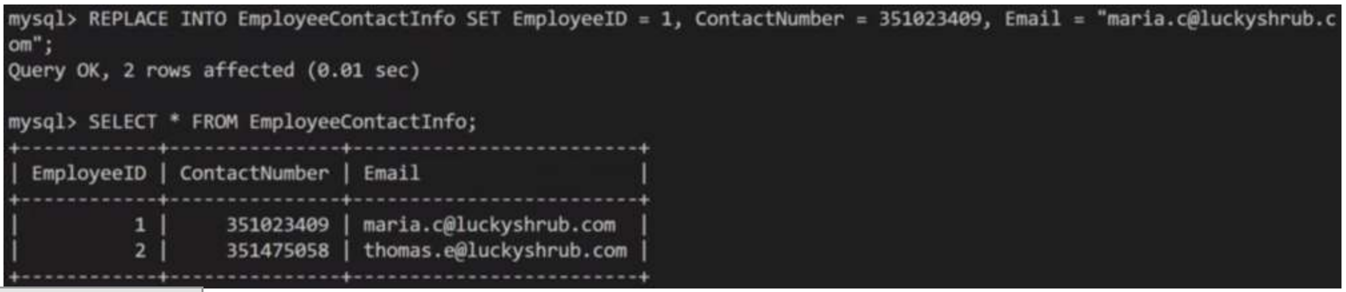
```
mysql> SELECT * FROM EmployeeContactInfo;
+------------+---------------+------------------------+
| EmployeeID | ContactNumber | Email                  |
+------------+---------------+------------------------+
|          1 |     351478025 | seamus.h@luckyshrub.com |
|          2 |     351475058 | thomas.e@luckyshrub.com |
+------------+---------------+------------------------+
2 rows in set (0.00 sec)

mysql> INSERT INTO EmployeeContactInfo(EmployeeID, ContactNumber, Email) VALUES (1, 351022508, "maria.c@luckyshrub.c
om");
ERROR 1062 (23000): Duplicate entry '1' for key 'EmployeeContactInfo.PRIMARY'
mysql> REPLACE INTO EmployeeContactInfo(EmployeeID, ContacNumber, Email) VALUES (1, 351022508, "maria.c@luckyshrubb.
com");
Query OK, 2 rows affected (0.02 sec)
```

..



```
mysql> SELECT * FROM EmployeeContactInfo;
+------------+---------------+-------------------------+
| EmployeeID | ContactNumber | Email                   |
+------------+---------------+-------------------------+
|          1 |     351022508 | maria.c@luckyshrub.com  |
|          2 |     351475058 | thomas.e@luckyshrub.com |
+------------+---------------+-------------------------+
```

BUT what if we want to change EmployeeContactInfo ?



```
mysql> REPLACE INTO EmployeeContactInfo SET EmployeeID = 1, ContactNumber = 351023409, Email = "maria.c@luckyshrub.c
om";
Query OK, 2 rows affected (0.01 sec)

mysql> SELECT * FROM EmployeeContactInfo;
+------------+---------------+-------------------------+
| EmployeeID | ContactNumber | Email                   |
+------------+---------------+-------------------------+
|          1 |     351023409 | maria.c@luckyshrub.com  |
|          2 |     351475058 | thomas.e@luckyshrub.com |
+------------+---------------+-------------------------+
```

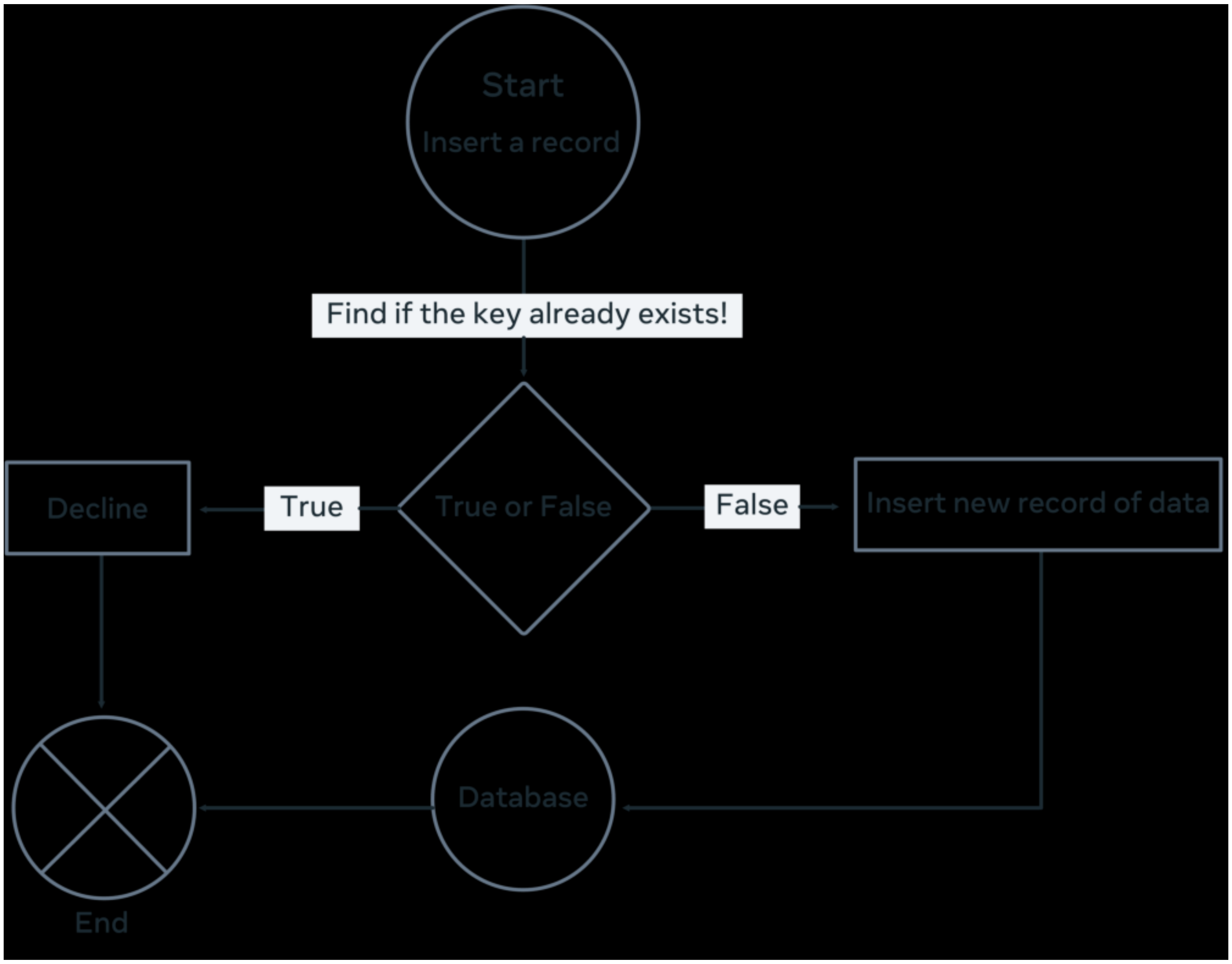
Loading [MathJax]/extensions/Safe.js

if we did not write the email it will retrun NULL

Diffirence between INSERT, UPDATE, & REPLACE:

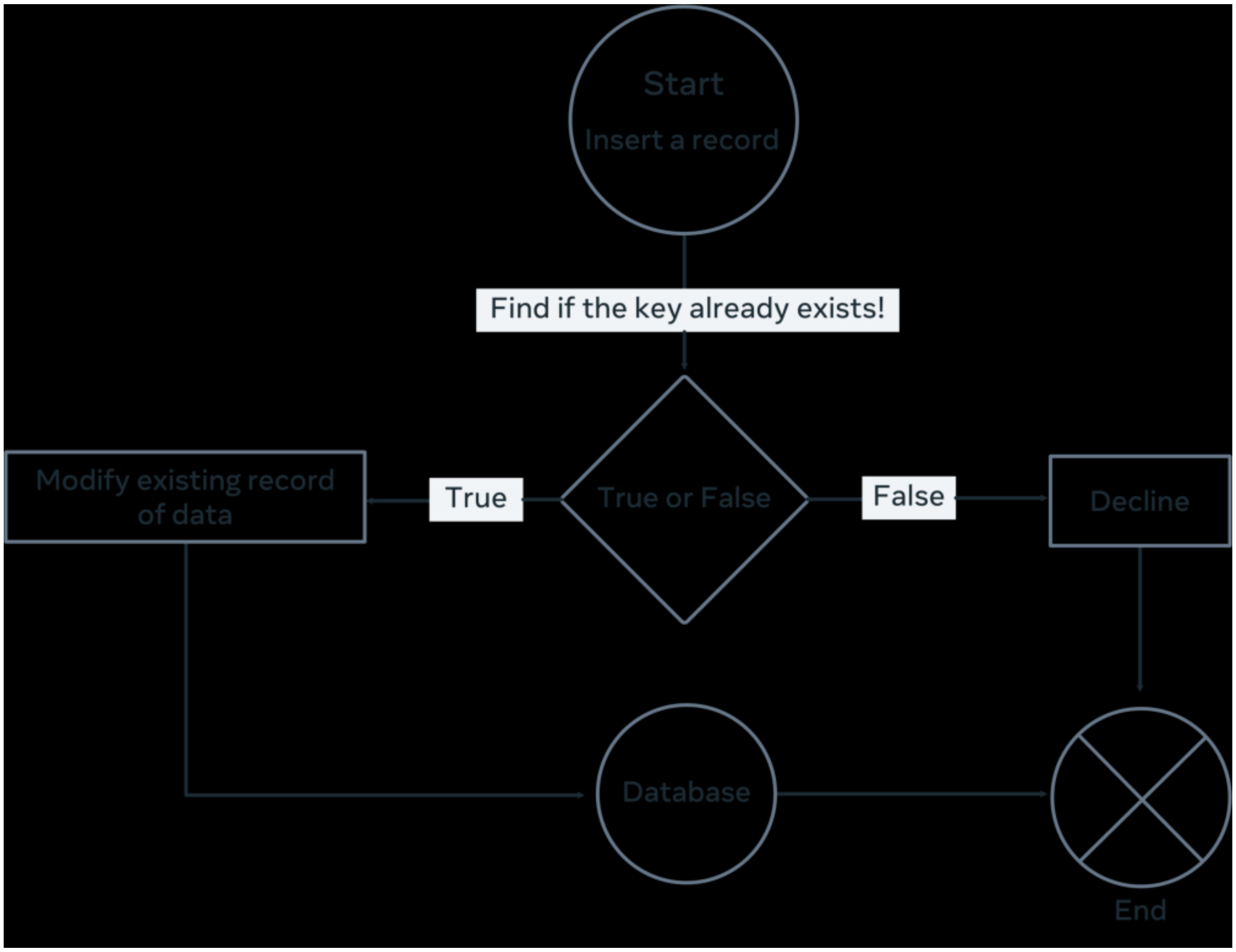
1. How the INSERT INTO statement works? The INSERT INTO statement attempts to insert a new record of data. It checks if the unique key already exists in the table. If YES or TRUE, the insert process is declined, and MySQL generates an error message.

Suppose a value of NO or FALSE is returned. In that case, the insert process is completed, and the new data record is added to the database. A flowchart that demonstrates how the INSERT INTO statement works is illustrated below.



1. How the UPDATE statement works? The update statement attempts to modify an existing record with new data. It checks if the unique key already exists in the table. Suppose a value of NO or FALSE is returned. In that case, the update process is declined, and MySQL generates an error message.

Suppose it returns a value of YES or TRUE. In that case, the update process is completed, and the existing data record is modified with the new data. A flowchart demonstrating how the UPDATE statement works illustrated below. Diagram for update statement


Start
Insert a record
Find if the key already exists!
Modify existing record of data
True
True or False
False
Decline
Database
End

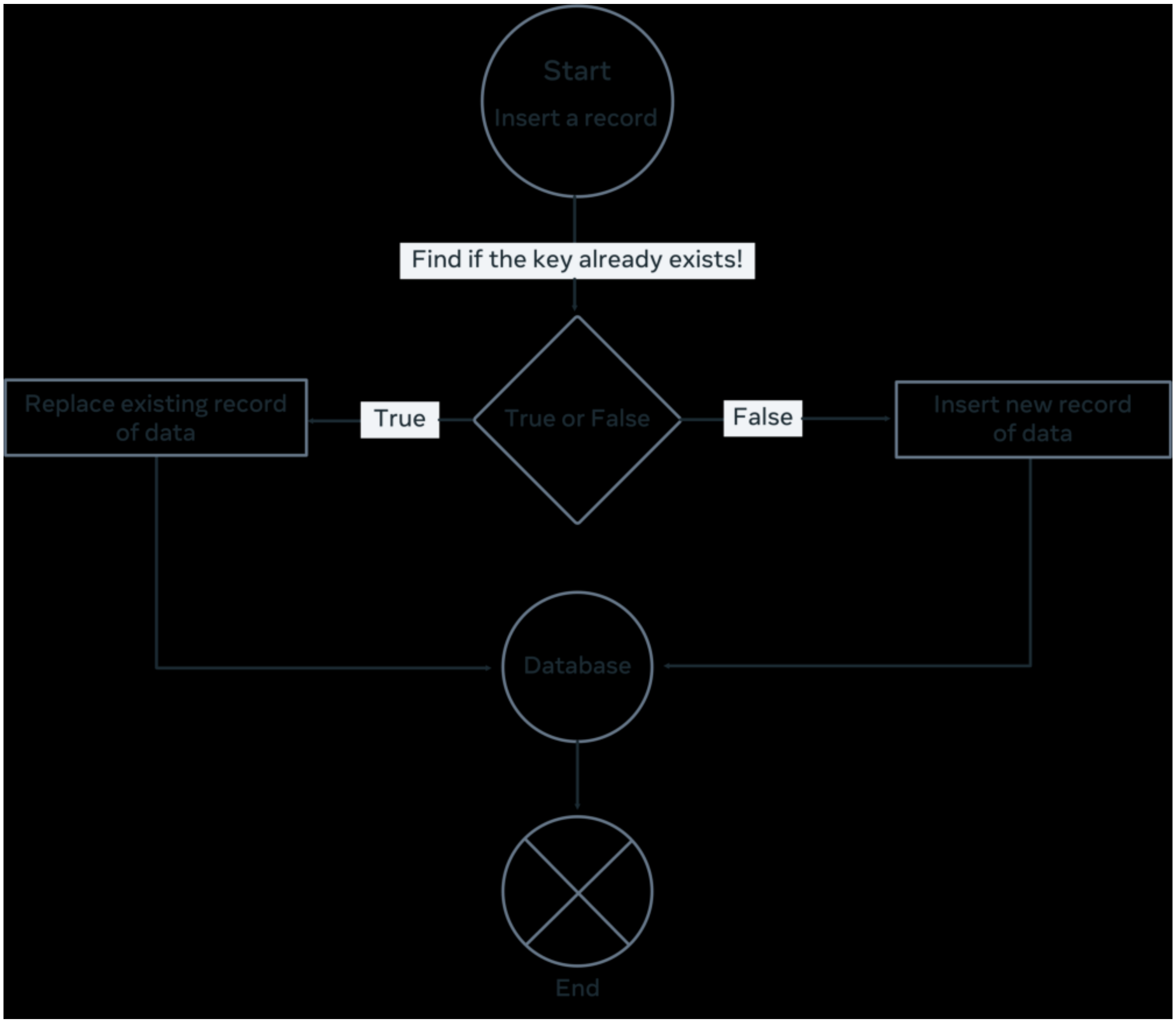
1. How the REPLACE INTO statement works? The REPLACE statement checks whether the intended data record's unique key value already exists in the table before inserting it as a new record or updating it.

The REPLACE INTO statement attempts to insert a new record or modify an existing record. In both cases, it checks whether the unique key of the proposed record already exists in the table. Suppose a value of NO or FALSE is returne. In that case, the REPLACE statement inserts the record similar to the INSERT INTO statement.

Suppose the key value already exists in the table (in other words, a duplicate key). In that case, the REPLACE statement deletes the existing record of data and replaces it with a new record of data. This happens regardless of whether you use the first or the second REPLACE statement syntax.

Loading [MathJax]/extensions/Safe.js

A flowchart outlining how the REPLACE INTO statement works is illustrated below.


Start
Insert a record
Find if the key already exists!
True or False
True
Replace existing record of data
False
Insert new record of data
Database
End

Constraints in MySQL

MySQL constraints: Used to ensure that each value in a column cell is unique

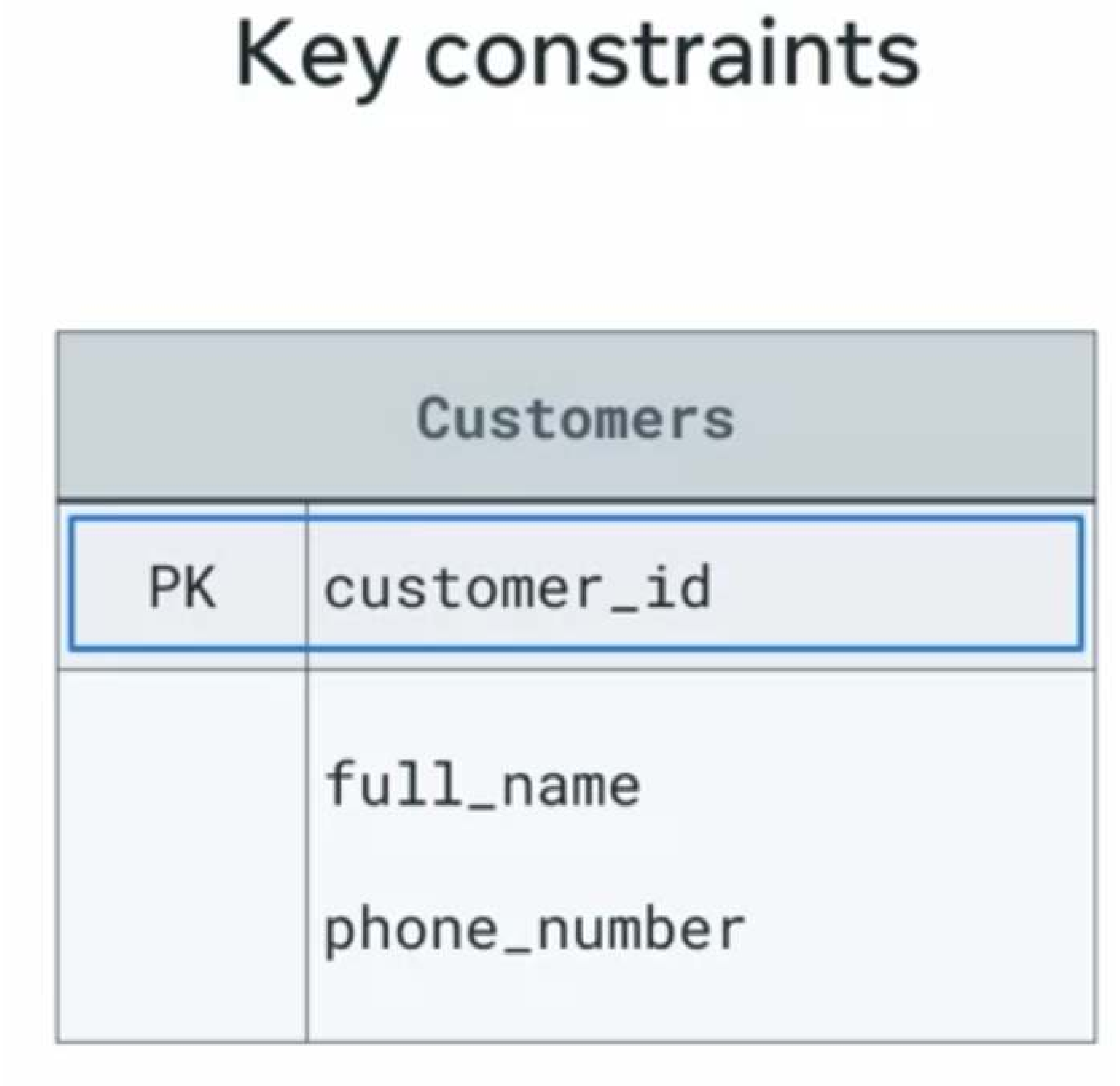
Main types of constrainnts in MySQL:

1. Key constraints: Apply rules to key types
2. Domain contraints: Govern values stored for a specific column
3. Referential integrity constraints: Establish rules for referential keys

let's explore each of these three:

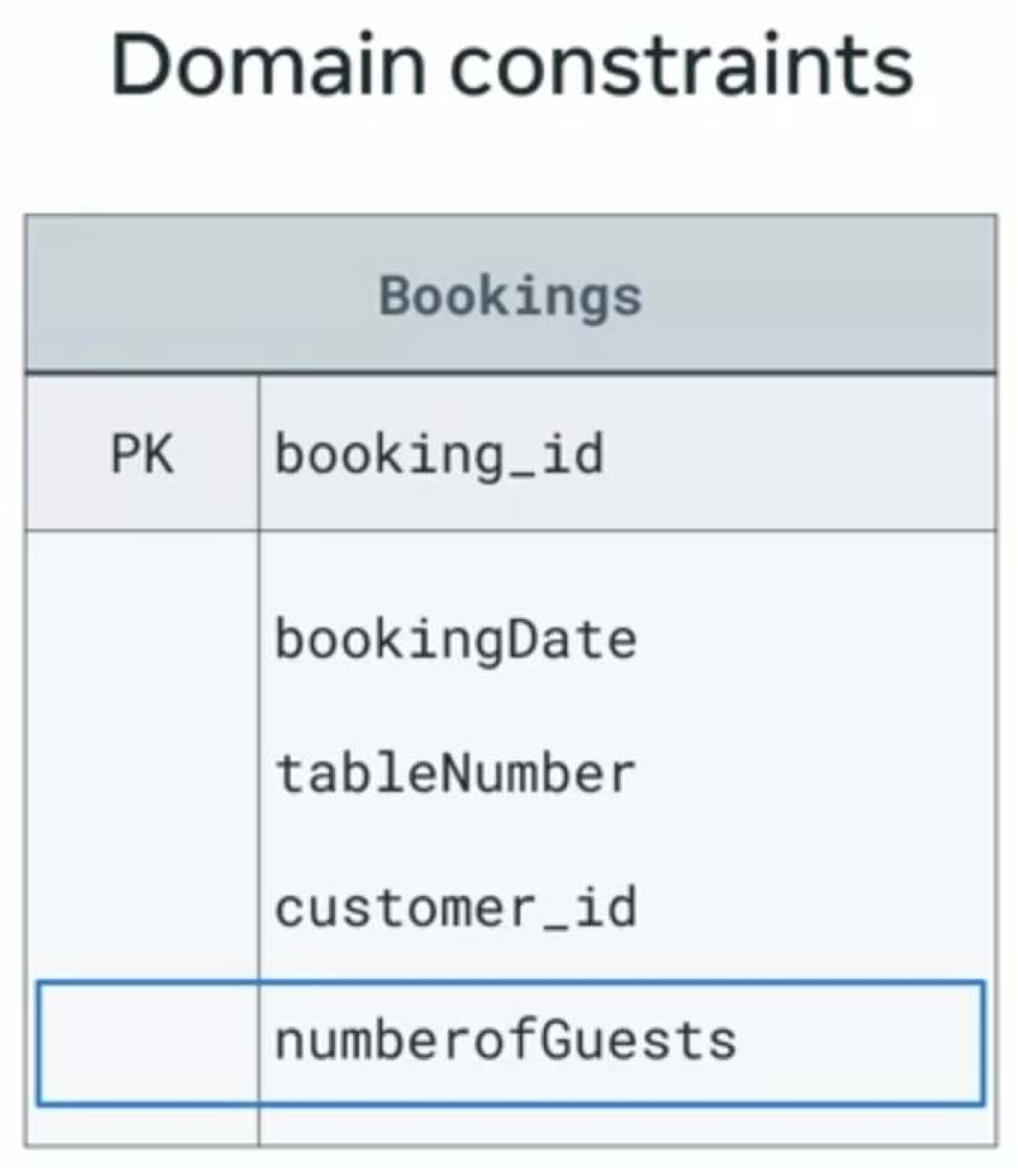
- As you see below, customer_id is a primary key. This column values must always be unique and it can never accept no value. In other word, all customer id must be unique

Loading [MathJax]/extensions/Safe.js


Key constraints

Customers	
PK	customer_id
	full_name phone_number

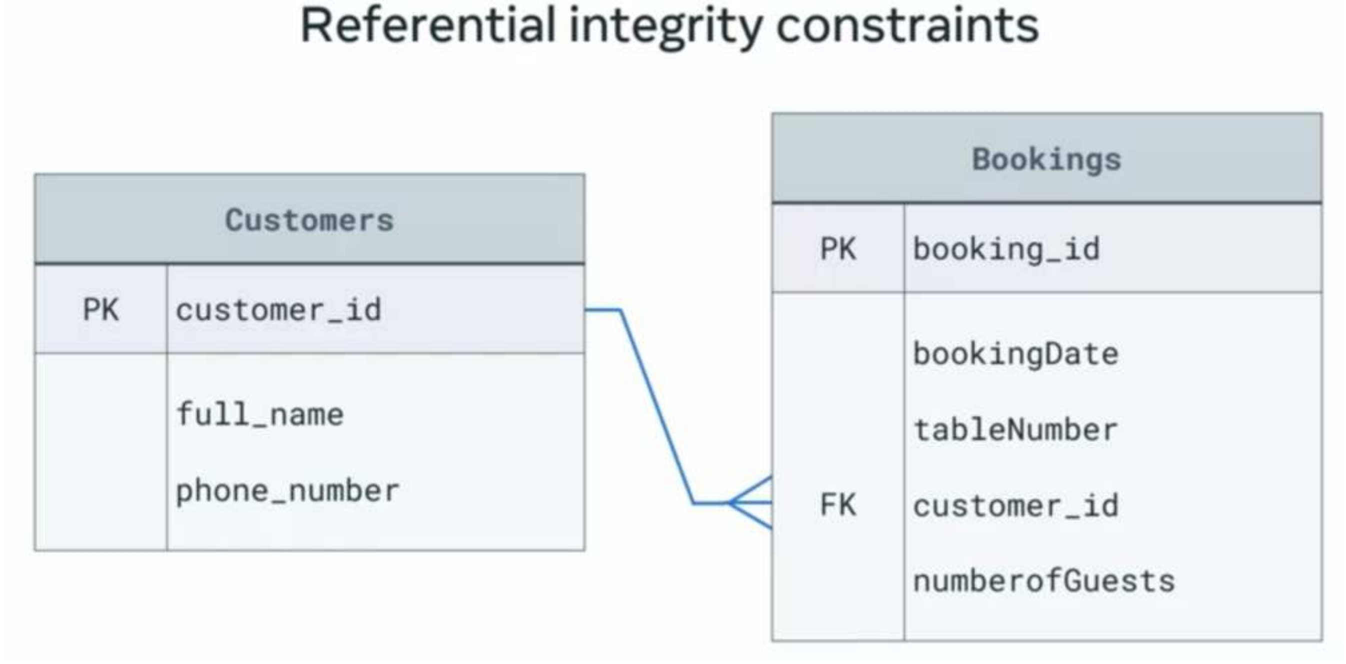
- Now let's look at domain constraints, SIIUUU restaurant can only facilitate a maximum of eight guests per booking. So, they enact the SQL check constrain on the number of guests column. This limits the value range that can be placed in the column, which means the table rejects any numric values greater than eight



Domain constraints

Bookings	
PK	booking_id
	bookingDate tableNumber customer_id
	numberofGuests

- Finally, in referential integrity constraints there are two types of tables, a referencing table that holds a primary key and a reference table that contains a foreign key. The value of the foreign key column that exists in the referecing table must always exist in the referenced table. Otherwise, a connection cannot be established between the two tables.



Summary

- Key Constraints

There are different types of keys in a relational database. For example, each table must have a primary key that maintains table integrity. The primary key ensures no duplications of records in the same table. Also, it allows identifying each data record using the primary key value. Therefore, it must be unique in each row of the table, and it is not allowed to contain null values.

For example, each citizen living in Denmark must have a unique personal number, which can be used to access different types of state services.

- Domain Constraints

Domain constraints refer to special rules defined for the values that can be stored for a certain column. To apply this, you must specify what data values are allowed and which ones should be rejected.

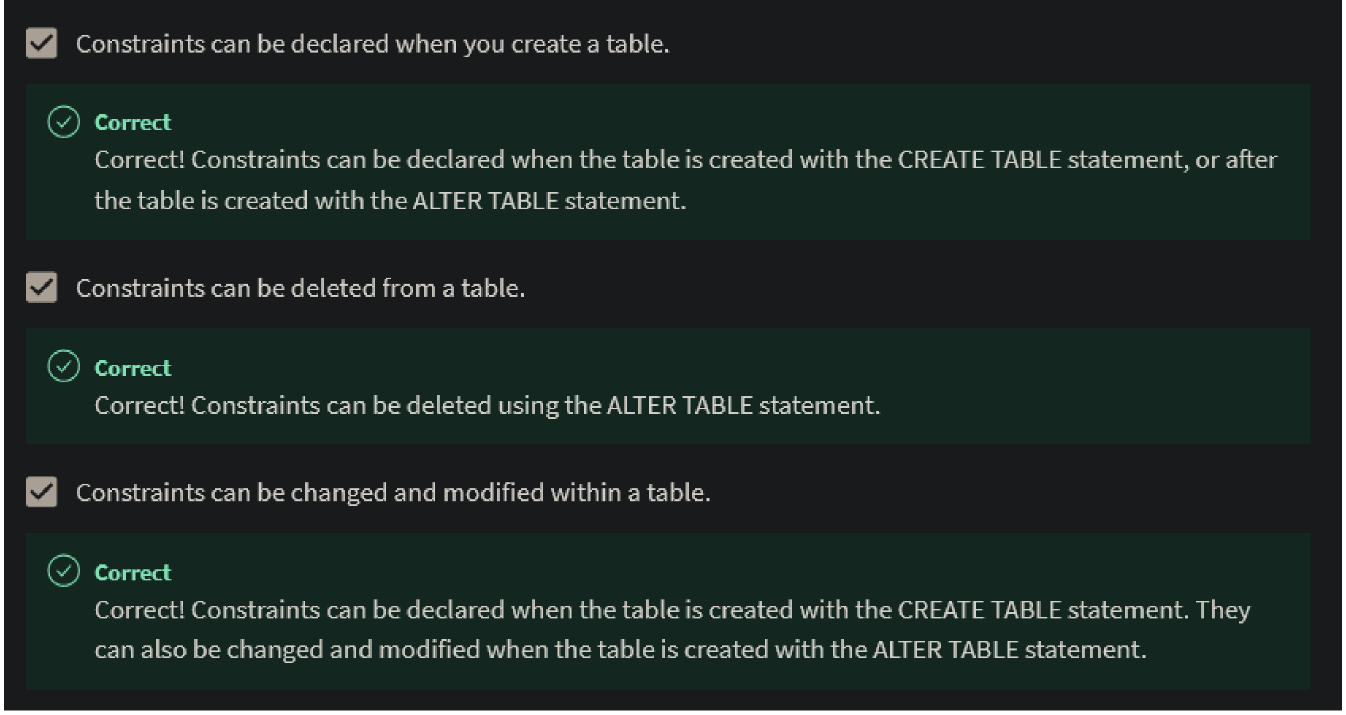
For example, you can define a valid range number for users to rate a streaming service that offers a wide variety of TV shows and movies. This range could be a number between 3 and 10, in which case the user will not be able to insert a value that is more than 10 and less than 3.

- Referential Integrity Constraints

In a relational database, tables are connected via a foreign key in one table linked to a primary key (or a unique key) in another table.

This implies that the value of the foreign key column in the 'referencing' table must also exist in the referenced table. Otherwise, you will end up with a problem as the "connection" between the records of the tables will cease.

Therefore, maintaining referential integrity requires that a foreign key value must have a matching primary key value to link the records of different related tables.


☑ Constraints can be declared when you create a table.

Correct

Correct! Constraints can be declared when the table is created with the CREATE TABLE statement, or after the table is created with the ALTER TABLE statement.

☑ Constraints can be deleted from a table.

Correct

Correct! Constraints can be deleted using the ALTER TABLE statement.

☑ Constraints can be changed and modified within a table.

Correct

Correct! Constraints can be declared when the table is created with the CREATE TABLE statement. They can also be changed and modified when the table is created with the ALTER TABLE statement.

Exercise: Working with constraints

Prerequisites

To complete this lab, you must have the Mangata and Gallo database in MySQL, so that you can create the three tables within it. If you do not have the database, then create it in MySQL using the following instructions.

The code to create and use the database is as follows:

1: Create database

CREATE DATABASE Mangata_Gallo;

2: Use database

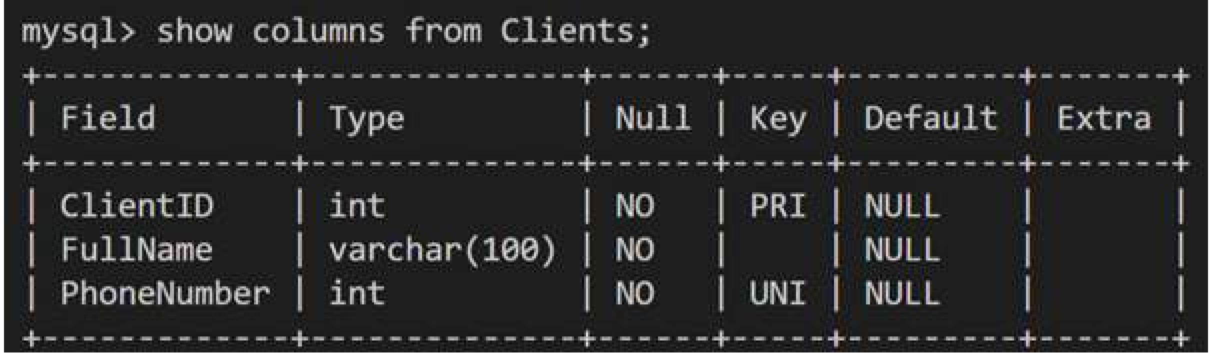
USE Mangata_Gallo;

Task 1: Create the Clients table with the following columns and constraints.

- ClientID: INT, NOT NULL and PRIMARY KEY
- FullName: VARCHAR(100) NOT NULL
- PhoneNumber: INT, NOT NULL and UNIQUE

The expected structure of the table should be the same as the following screenshot (assuming that you have created and populated the tables correctly.)

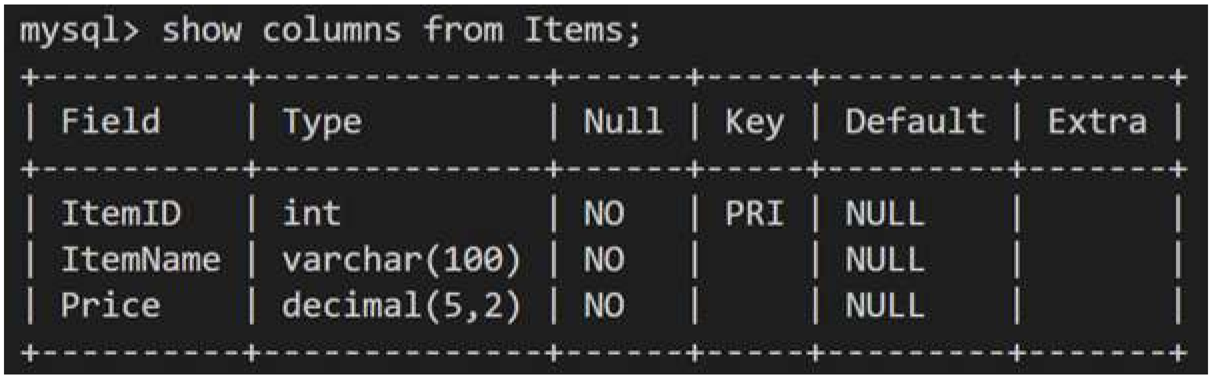
Loading [MathJax]/extensions/Safe.js


```
mysql> show columns from Clients;
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| ClientID    | int          | NO   | PRI | NULL    |       |
| FullName    | varchar(100) | NO   |     | NULL    |       |
| PhoneNumber | int          | NO   | UNI | NULL    |       |
+-------------+--------------+------+-----+---------+-------+
```

Task 2: Create the Items table with the following attributes and constraints:

- ItemID: INT, NOT NULL and PRIMARY KEY
- ItemName: VARCHAR(100) and NOT NULL
- Price: Decimal(5,2) and NOT NULL

The expected structure of the table should be the same as the following screenshot (assuming that you have created and populated the tables correctly.)

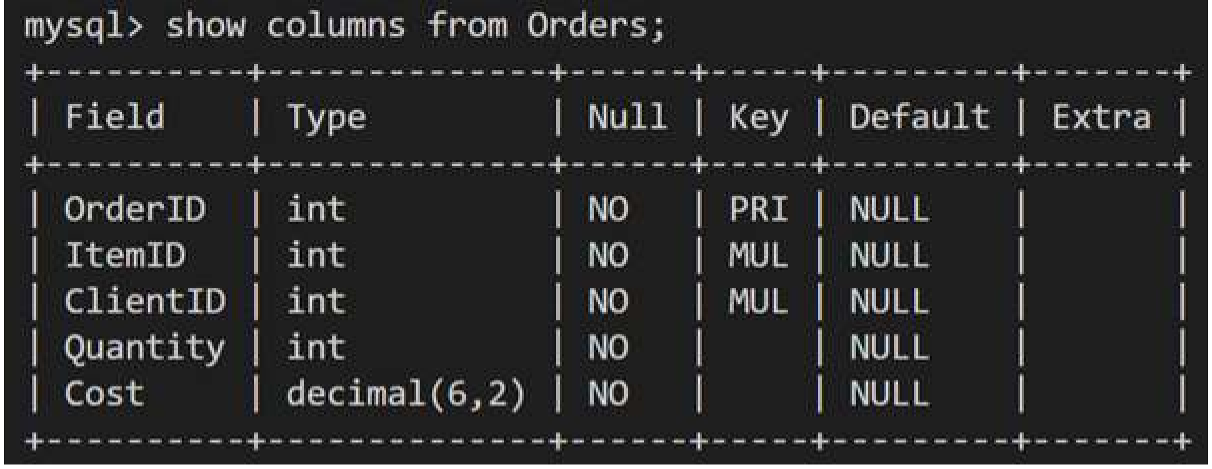

```
mysql> show columns from Items;
+----------+--------------+------+-----+---------+-------+
| Field    | Type         | Null | Key | Default | Extra |
+----------+--------------+------+-----+---------+-------+
| ItemID   | int          | NO   | PRI | NULL    |       |
| ItemName | varchar(100) | NO   |     | NULL    |       |
| Price    | decimal(5,2) | NO   |     | NULL    |       |
+----------+--------------+------+-----+---------+-------+
```

Task 3: Create the Orders table with the following constraints.

- OrderID: INT, NOT NULL and PRIMARY KEY
- ClientID: INT, NOT NULL and FOREIGN KEY
- ItemID: INT, NOT NULL and FOREIGN KEY
- Quantity: INT, NOT NULL and maximum allowed items in each order 3 only
- COST Decimal(6,2) and NOT NULL

The expected structure of the table should be the same as the following screenshot (assuming that you have created and populated the tables correctly.) details showing orders and details

Loading [MathJax]/extensions/Safe.js



```
mysql> show columns from Orders;
+----------+--------------+------+-----+---------+-------+
| Field    | Type         | Null | Key | Default | Extra |
+----------+--------------+------+-----+---------+-------+
| OrderID  | int          | NO   | PRI | NULL    |       |
| ItemID   | int          | NO   | MUL | NULL    |       |
| ClientID | int          | NO   | MUL | NULL    |       |
| Quantity | int          | NO   |     | NULL    |       |
| Cost     | decimal(6,2) | NO   |     | NULL    |       |
+----------+--------------+------+-----+---------+-------+
```

CREATE DATABASE Mangata_Gallo; USE Mangata_Gallo; CREATE TABLE Clints (ClintID INT PRIMARY KEY, FullName VARCHAR(100) NOT NULL, PhoneNumber INT NOT NULL UNIQUE); CREATE TABLE Items (ItemID INT PRIMARY KEY, ItemName VARCHAR(100) NOT NULL, PRICE DECIMAL(5,2) NOT NULL); CREATE TABLE Orders (OrderID PRIMARY KEY, ItemID INT NOT NULL, ClientID INT NOT NULL, Quantity INT NOT NULL CHECK (Quantity < 4), Cost DECIMAL(6,2) NOT NULL, FOREIGN KEY (ClientID) REFERENCES Clients (ClientID), FOREIGN KEY (ItemID) REFERENCES Items (ItemID));

ON DELETE & ON UPDATE

CASCADE, SET NULL, and RESTRICT (NO ACTION) are different referential actions that can be specified in MySQL when defining foreign key constraints. These actions determine what should happen to related rows in a child table when a corresponding row in the parent table is deleted or updated. Here's an explanation of each action with examples:

1. **CASCADE**: When a foreign key constraint is defined with CASCADE, it means that if a row in the parent table is deleted or updated, the corresponding rows in the child table will be automatically deleted or updated as well. This action propagates the changes from the parent table to the child table.

Example: Consider two tables, orders and order_items, with a foreign key relationship where order_items references orders:

CREATE TABLE orders (order_id INT PRIMARY KEY); CREATE TABLE order_items (item_id INT PRIMARY KEY, order_id INT, item_name VARCHAR(50), FOREIGN KEY (order_id) REFERENCES orders(order_id) ON DELETE CASCADE ON UPDATE CASCADE);

If a row in the orders table is deleted or updated, all the corresponding rows in the order_items table will also be deleted or updated accordingly.

1. **SET NULL**: When a foreign key constraint is defined with SET NULL, it means that if a row in the parent table is deleted or updated, the foreign key values in the child table will be set to NULL. This action allows the child table to have NULL values in the foreign key column(s).

Example: Consider two tables, users and comments, with a foreign key relationship where comments references users:

CREATE TABLE users (user_id INT PRIMARY KEY); CREATE TABLE comments (comment_id INT PRIMARY KEY, user_id INT, comment_text VARCHAR(100), FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE SET NULL ON UPDATE SET NULL);

If a row in the users table is deleted or updated, the corresponding user_id value(s) in the comments table will be set to NULL.

1. **RESTRICT (NO ACTION)**: When a foreign key constraint is defined with RESTRICT (or NO ACTION, which is equivalent), it means that if a row in the parent table is deleted or updated, the corresponding operation will be restricted if there are dependent rows in the child table. In other words, the deletion or update will fail, and an error will be thrown.

Example: Consider two tables, categories and products, with a foreign key relationship where products references categories:

CREATE TABLE categories (category_id INT PRIMARY KEY); CREATE TABLE products (product_id INT PRIMARY KEY, category_id INT, product_name VARCHAR(50), FOREIGN KEY (category_id) REFERENCES categories(category_id) ON DELETE RESTRICT ON UPDATE RESTRICT);

If there are any rows in the products table that have a corresponding category_id value in the categories table, you won't be able to delete or update the corresponding row in the categories table unless the dependent rows in the products table are first removed or updated to reference a different category.

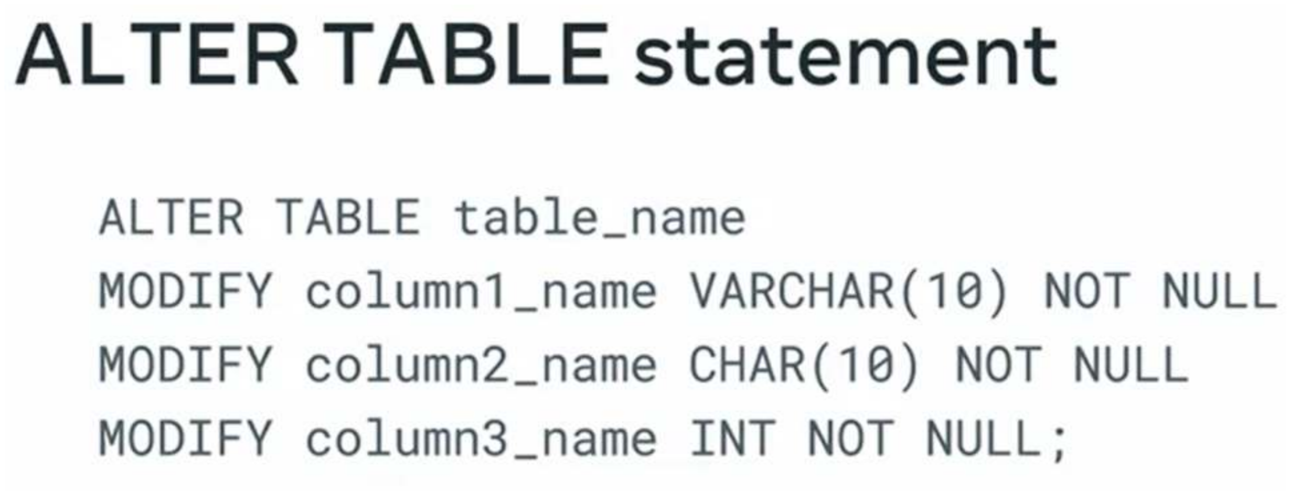
Changing table structure

MySql ALTER Table

ALTER TABLE statement: Makes changes to a table in a database by altering columns or constraints

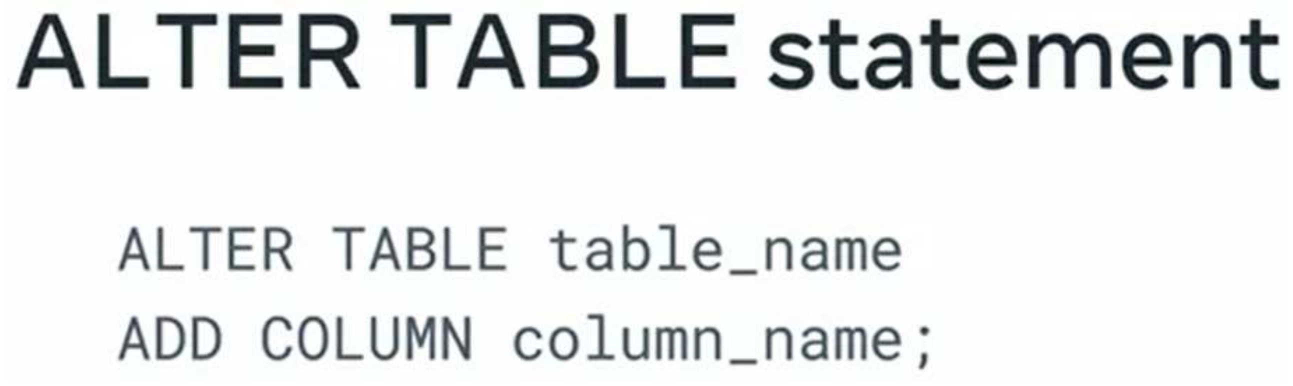
Overview of some common commands used with ALTER TABLE:

1. MODIFY: Make changes to specific columns in a table
2. ADD: Add a column to a table
3. DROP: Drop a column from a table
4. RENAME: Rename columns and table


ALTER TABLE statement

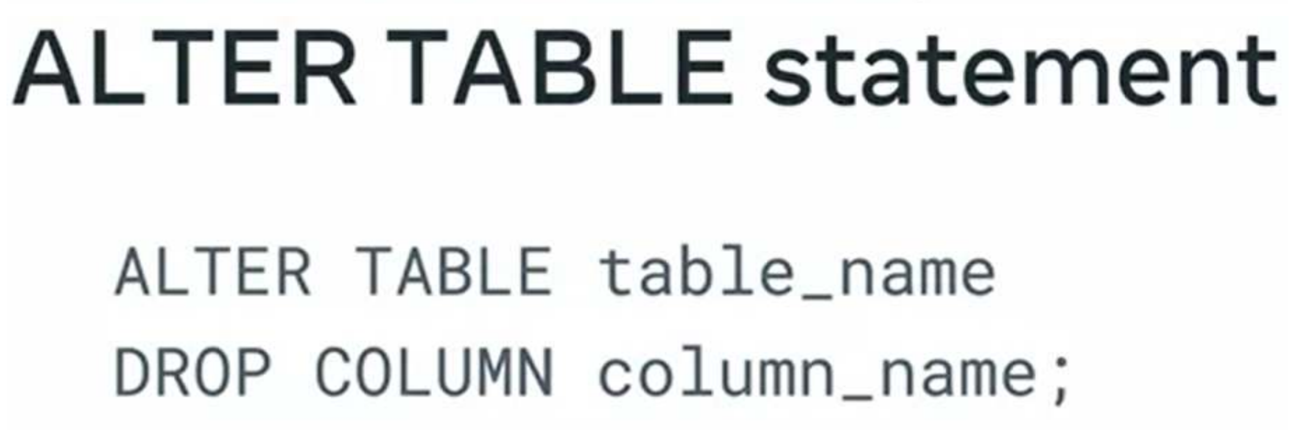
```
ALTER TABLE table_name
MODIFY column1_name VARCHAR(10) NOT NULL
MODIFY column2_name CHAR(10) NOT NULL
MODIFY column3_name INT NOT NULL;
```

Loading [MathJax]/extensions/Safe.js



ALTER TABLE statement

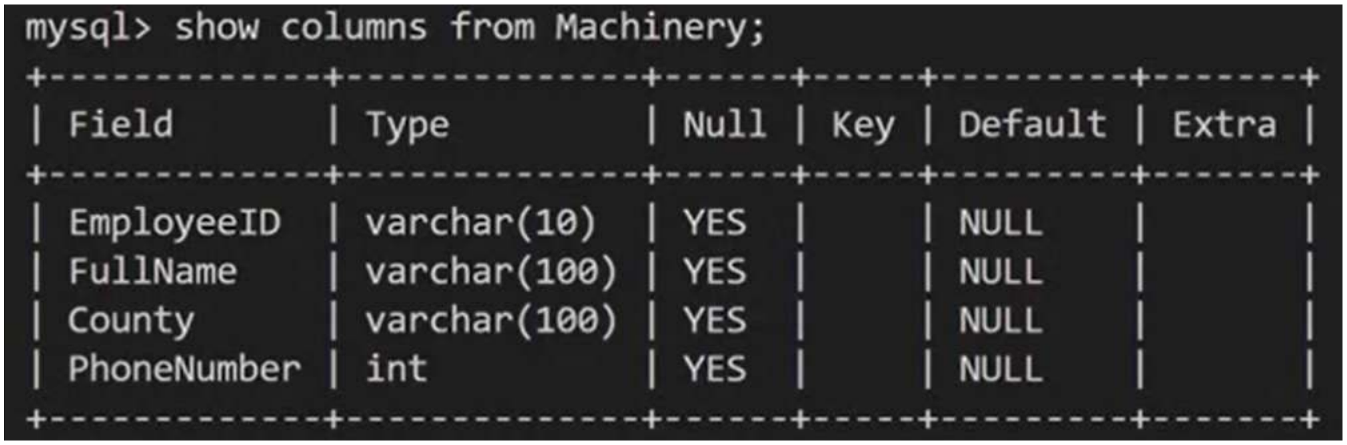
```
ALTER TABLE table_name
ADD COLUMN column_name;
```



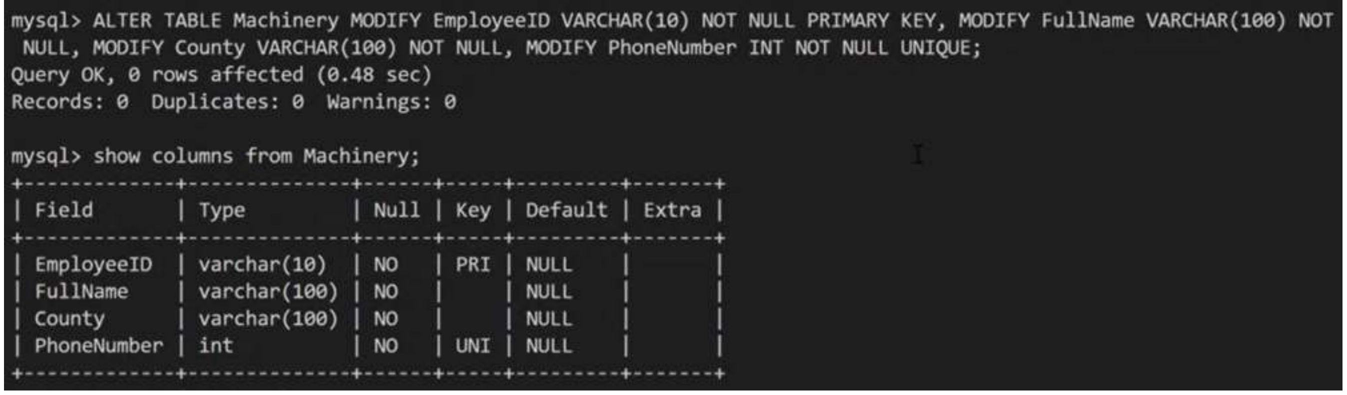
ALTER TABLE statement

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

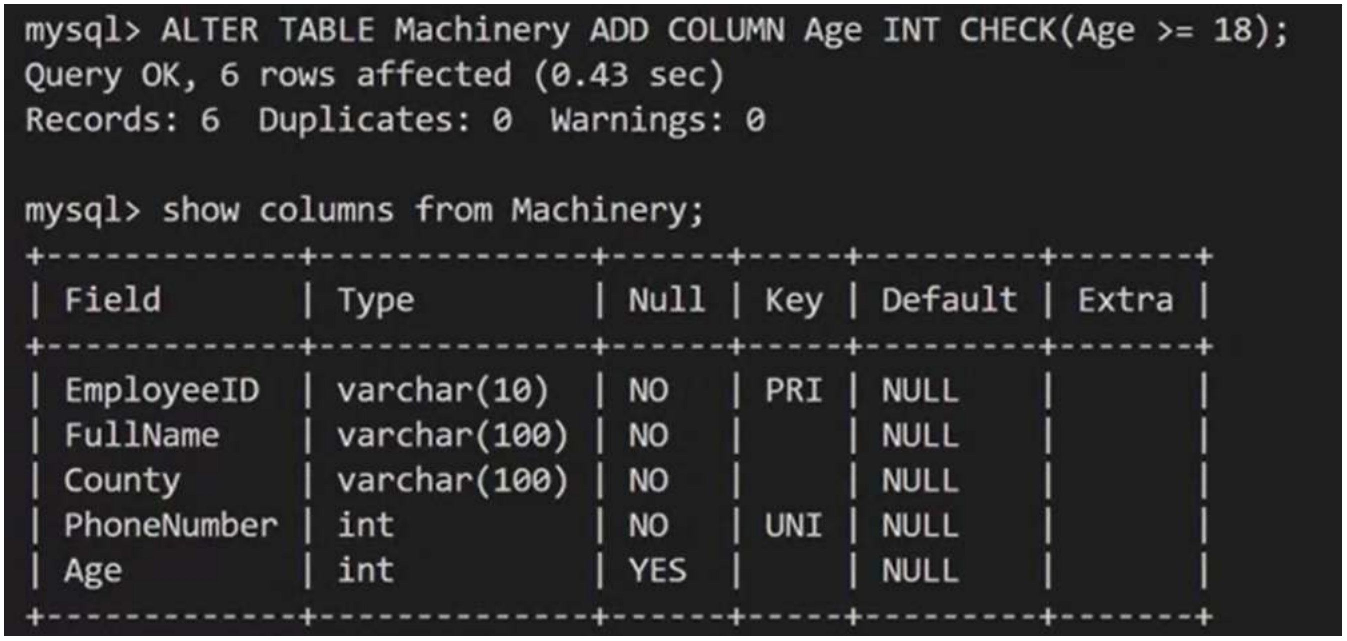
EXAMPLE:



```
mysql> show columns from Machinery;
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| EmployeeID  | varchar(10)  | YES  |     | NULL    |       |
| FullName    | varchar(100) | YES  |     | NULL    |       |
| County      | varchar(100) | YES  |     | NULL    |       |
| PhoneNumber | int          | YES  |     | NULL    |       |
+-------------+--------------+------+-----+---------+-------+
```



```
mysql> ALTER TABLE Machinery MODIFY EmployeeID VARCHAR(10) NOT NULL PRIMARY KEY, MODIFY FullName VARCHAR(100) NOT
 NULL, MODIFY County VARCHAR(100) NOT NULL, MODIFY PhoneNumber INT NOT NULL UNIQUE;
Query OK, 0 rows affected (0.48 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> show columns from Machinery;
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| EmployeeID  | varchar(10)  | NO   | PRI | NULL    |       |
| FullName    | varchar(100) | NO   |     | NULL    |       |
| County      | varchar(100) | NO   |     | NULL    |       |
| PhoneNumber | int          | NO   | UNI | NULL    |       |
+-------------+--------------+------+-----+---------+-------+
```

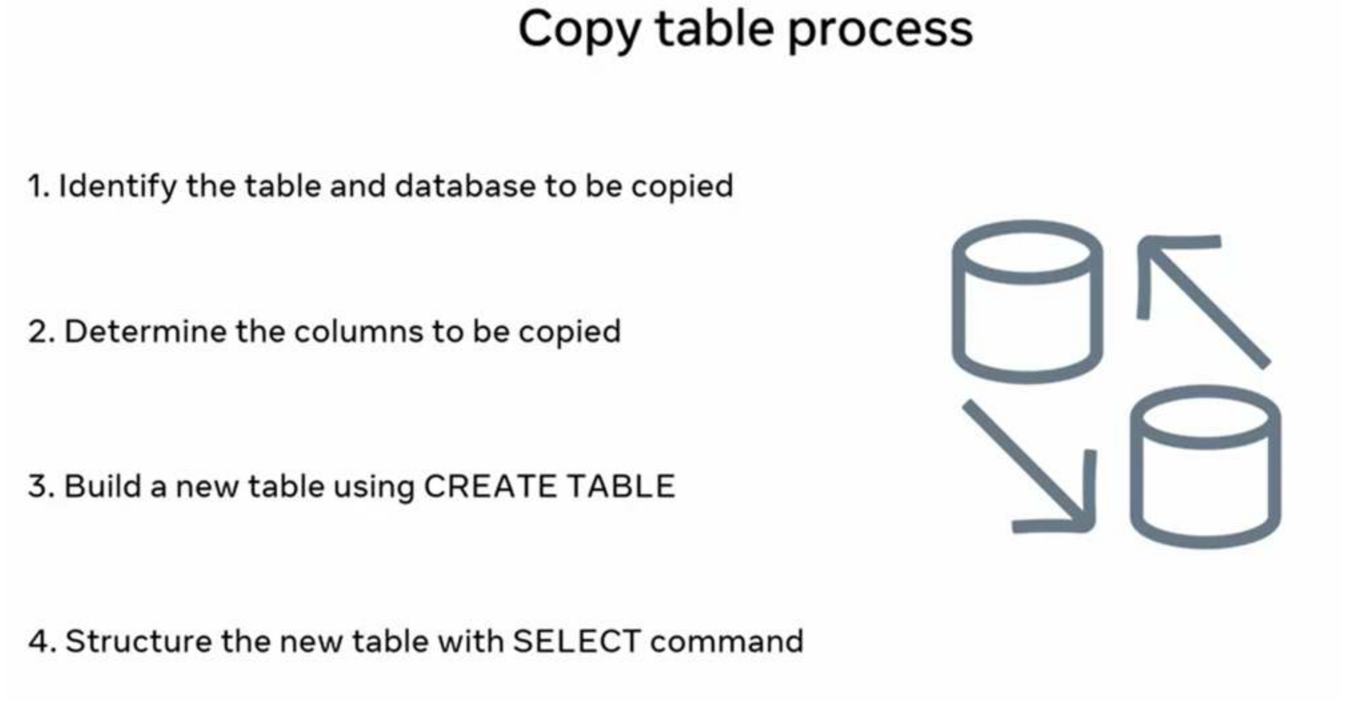


```
mysql> ALTER TABLE Machinery ADD COLUMN Age INT CHECK(Age >= 18);
Query OK, 6 rows affected (0.43 sec)
Records: 6  Duplicates: 0  Warnings: 0

mysql> show columns from Machinery;
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| EmployeeID  | varchar(10)  | NO   | PRI | NULL    |       |
| FullName    | varchar(100) | NO   |     | NULL    |       |
| County      | varchar(100) | NO   |     | NULL    |       |
| PhoneNumber | int          | NO   | UNI | NULL    |       |
| Age         | int          | YES  |     | NULL    |       |
+-------------+--------------+------+-----+---------+-------+
```

MySQL COPY TABLE

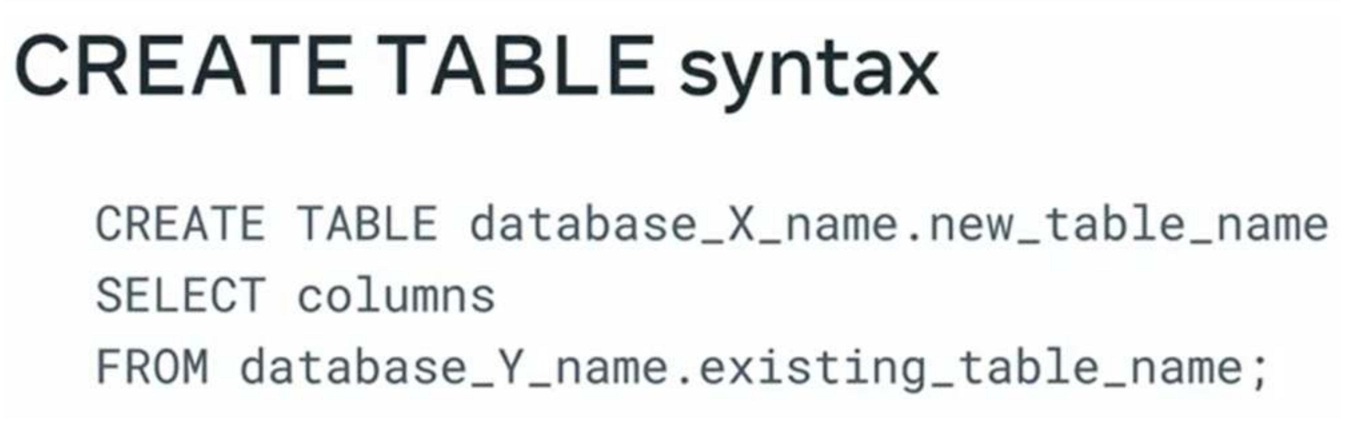
COPY TABLE **Overview**

1. Copy an existing table's data to a new table
2. Copy a table with constraints
3. Copy existing table data to a new database



Copy table process

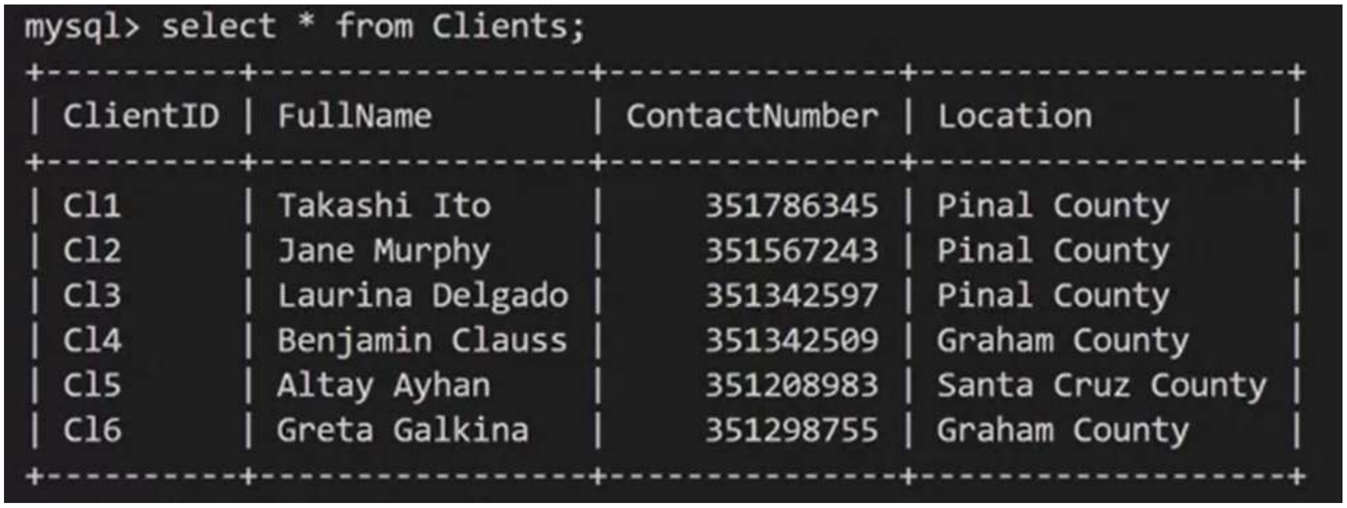
1. Identify the table and database to be copied
2. Determine the columns to be copied
3. Build a new table using CREATE TABLE
4. Structure the new table with SELECT command



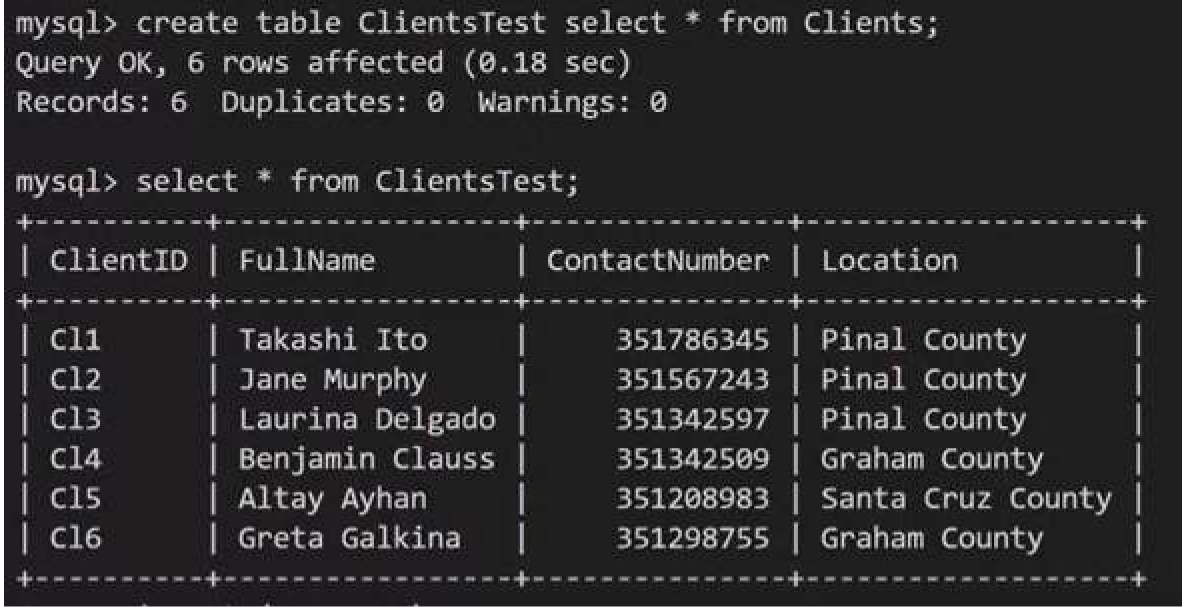
CREATE TABLE syntax

```
CREATE TABLE database_X_name.new_table_name
SELECT columns
FROM database_Y_name.existing_table_name;
```

EXAMPLE:

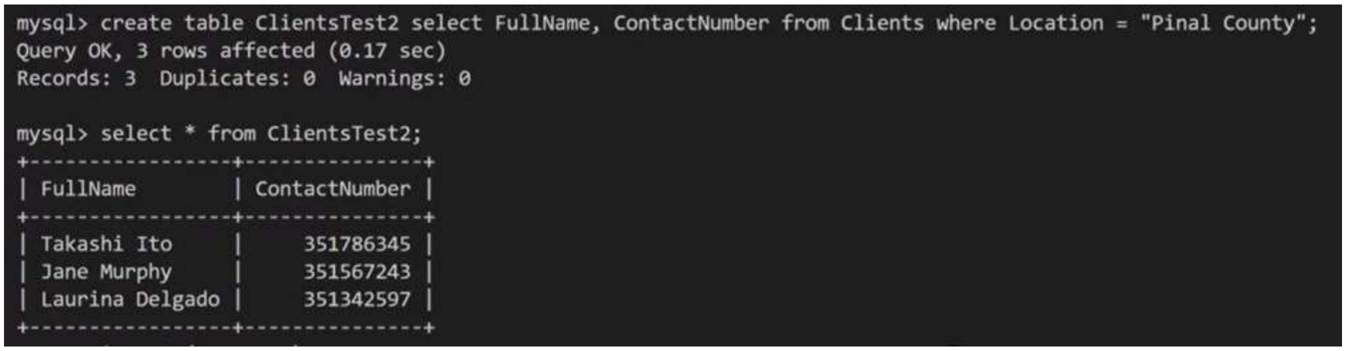


```
mysql> select * from Clients;
+----------+-----------------+---------------+-------------------+
| ClientID | FullName        | ContactNumber | Location          |
+----------+-----------------+---------------+-------------------+
| Cl1      | Takashi Ito     |     351786345 | Pinal County      |
| Cl2      | Jane Murphy     |     351567243 | Pinal County      |
| Cl3      | Laurina Delgado |     351342597 | Pinal County      |
| Cl4      | Benjamin Clauss |     351342509 | Graham County     |
| Cl5      | Altay Ayhan     |     351208983 | Santa Cruz County |
| Cl6      | Greta Galkina   |     351298755 | Graham County     |
+----------+-----------------+---------------+-------------------+
```

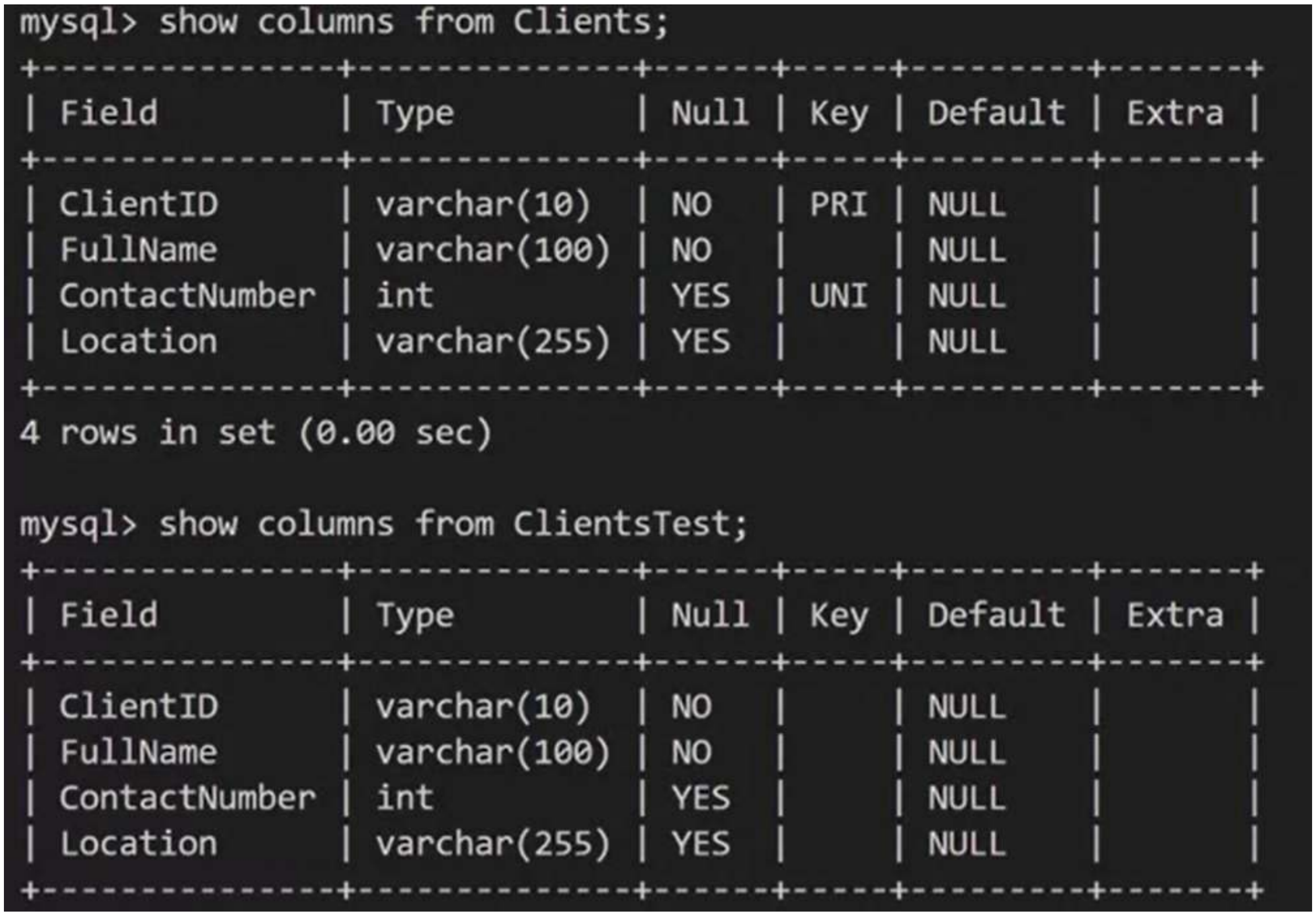


```
mysql> create table ClientsTest select * from Clients;
Query OK, 6 rows affected (0.18 sec)
Records: 6  Duplicates: 0  Warnings: 0

mysql> select * from ClientsTest;
+----------+-----------------+---------------+-------------------+
| ClientID | FullName        | ContactNumber | Location          |
+----------+-----------------+---------------+-------------------+
| Cl1      | Takashi Ito     |     351786345 | Pinal County      |
| Cl2      | Jane Murphy     |     351567243 | Pinal County      |
| Cl3      | Laurina Delgado |     351342597 | Pinal County      |
| Cl4      | Benjamin Clauss |     351342509 | Graham County     |
| Cl5      | Altay Ayhan     |     351208983 | Santa Cruz County |
| Cl6      | Greta Galkina   |     351298755 | Graham County     |
+----------+-----------------+---------------+-------------------+
```

Below we will copy just a FullNAME & Contact number whose live in Pinal County



```
mysql> create table ClientsTest2 select FullName, ContactNumber from Clients where Location = "Pinal County";
Query OK, 3 rows affected (0.17 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> select * from ClientsTest2;
+-----------------+---------------+
| FullName        | ContactNumber |
+-----------------+---------------+
| Takashi Ito     |     351786345 |
| Jane Murphy     |     351567243 |
| Laurina Delgado |     351342597 |
+-----------------+---------------+
```

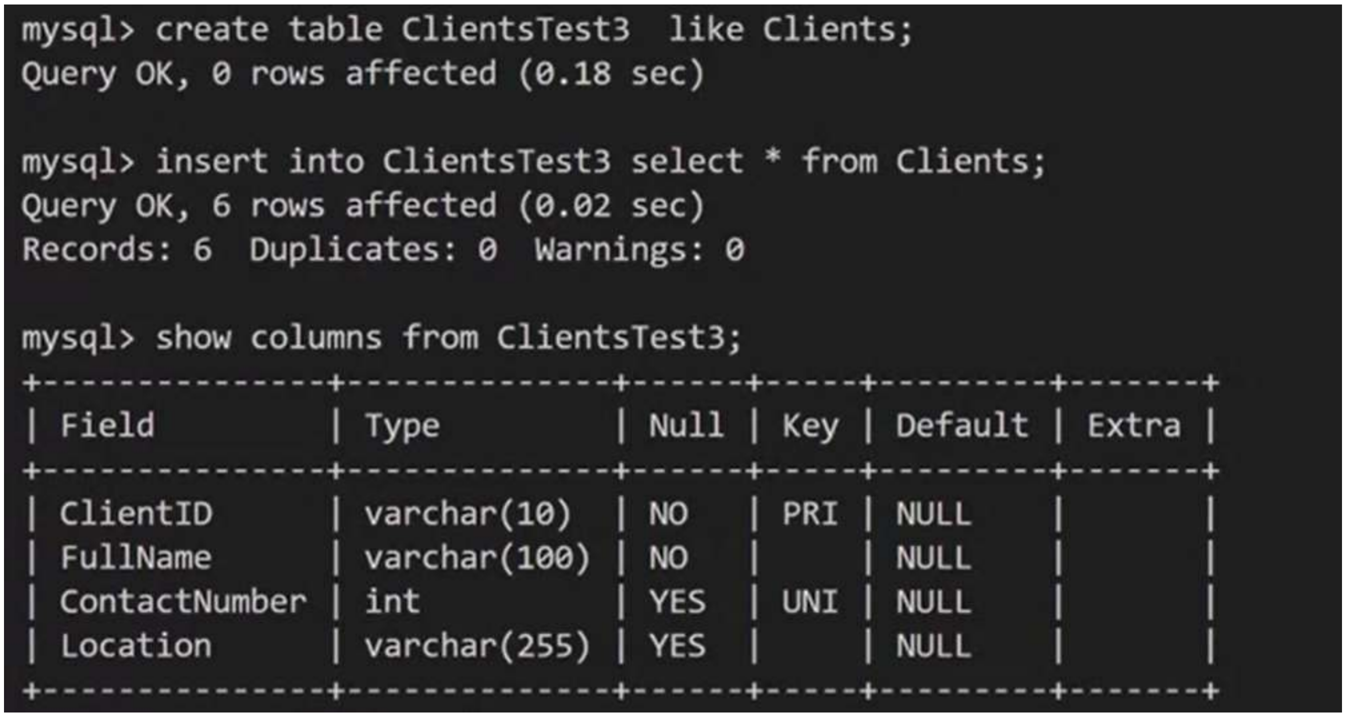


```
mysql> show columns from Clients;
+---------------+--------------+------+-----+---------+-------+
| Field         | Type         | Null | Key | Default | Extra |
+---------------+--------------+------+-----+---------+-------+
| ClientID      | varchar(10)  | NO   | PRI | NULL    |       |
| FullName      | varchar(100) | NO   |     | NULL    |       |
| ContactNumber | int          | YES  | UNI | NULL    |       |
| Location      | varchar(255) | YES  |     | NULL    |       |
+---------------+--------------+------+-----+---------+-------+
4 rows in set (0.00 sec)

mysql> show columns from ClientsTest;
+---------------+--------------+------+-----+---------+-------+
| Field         | Type         | Null | Key | Default | Extra |
+---------------+--------------+------+-----+---------+-------+
| ClientID      | varchar(10)  | NO   |     | NULL    |       |
| FullName      | varchar(100) | NO   |     | NULL    |       |
| ContactNumber | int          | YES  |     | NULL    |       |
| Location      | varchar(255) | YES  |     | NULL    |       |
+---------------+--------------+------+-----+---------+-------+
```

As you see above, the table is missing the primary and unique keys defined in the original table. So, we

Loading [MathJax]/extensions/Safe.js

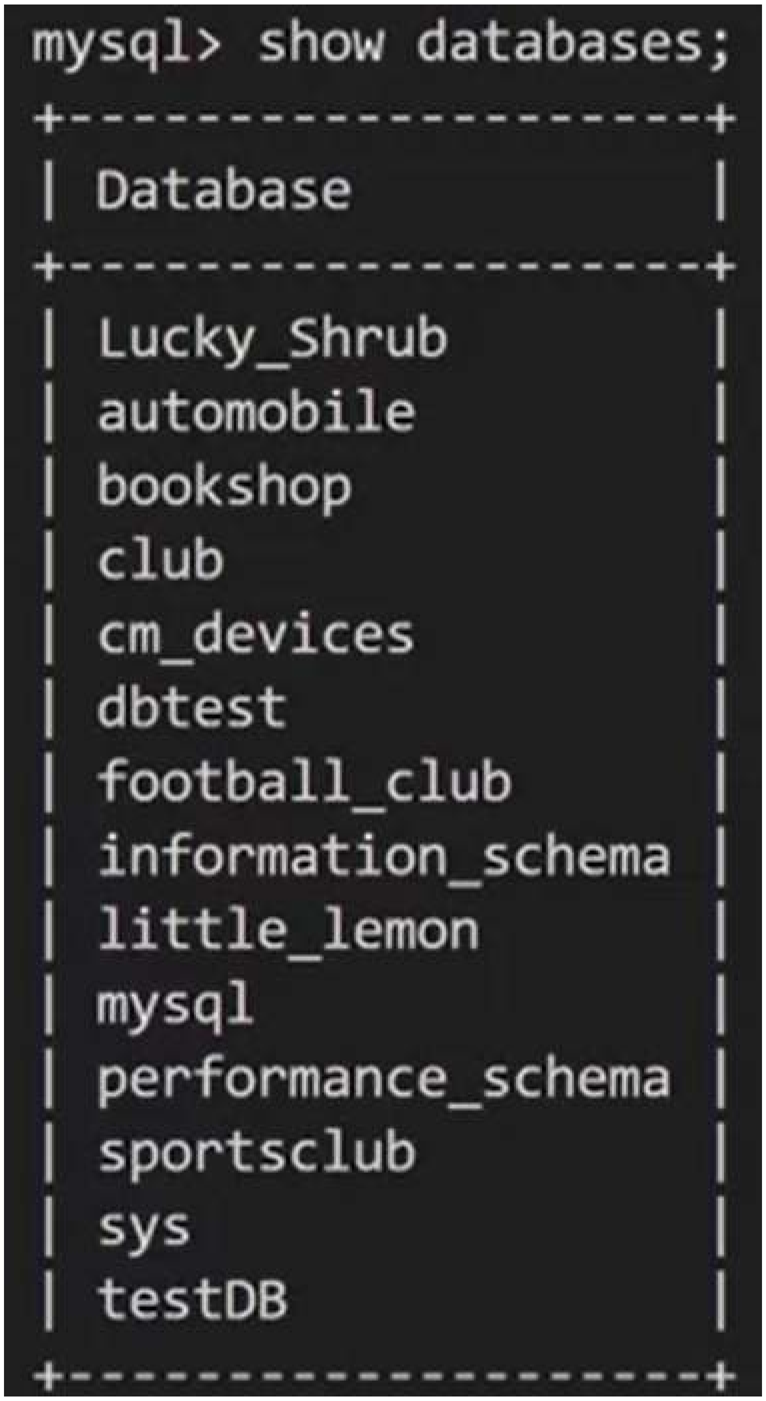
need to copy these keys by using the following statement



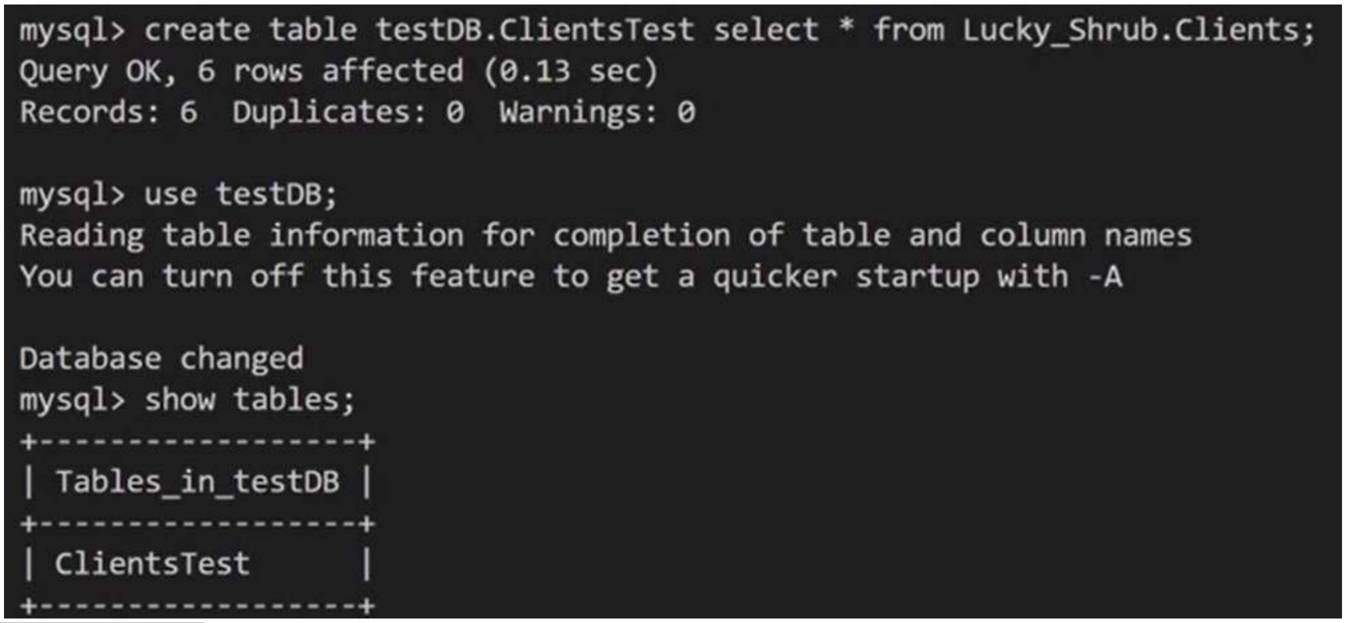
```
mysql> create table ClientsTest3  like Clients;
Query OK, 0 rows affected (0.18 sec)

mysql> insert into ClientsTest3 select * from Clients;
Query OK, 6 rows affected (0.02 sec)
Records: 6  Duplicates: 0  Warnings: 0

mysql> show columns from ClientsTest3;
+---------------+--------------+------+-----+---------+-------+
| Field         | Type         | Null | Key | Default | Extra |
+---------------+--------------+------+-----+---------+-------+
| ClientID      | varchar(10)  | NO   | PRI | NULL    |       |
| FullName      | varchar(100) | NO   |     | NULL    |       |
| ContactNumber | int          | YES  | UNI | NULL    |       |
| Location      | varchar(255) | YES  |     | NULL    |       |
+---------------+--------------+------+-----+---------+-------+
```

The like key word create an exact copy of the original table

Our final task is to copy the clint table from our database to another new database



```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| Lucky_Shrub        |
| automobile         |
| bookshop           |
| club               |
| cm_devices         |
| dbtest             |
| football_club      |
| information_schema |
| little_lemon       |
| mysql              |
| performance_schema |
| sportsclub         |
| sys                |
| testDB             |
+--------------------+
```



```
mysql> create table testDB.ClientsTest select * from Lucky_Shrub.Clients;
Query OK, 6 rows affected (0.13 sec)
Records: 6  Duplicates: 0  Warnings: 0

mysql> use testDB;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables;
+------------------+
| Tables_in_testDB |
+------------------+
| ClientsTest      |
+------------------+
```

Loading [MathJax]/extensions/Safe.js

Exercise: Changing table structure

Prerequisites

To complete this lab, you need to have the Mangata and Gallo database so that you can create the 'Staff' table inside it. If you don't have this database, complete the following steps to create it.

1: Create database CREATE DATABASE Mangata_Gallo;

2: Use database USE Mangata_Gallo;

This activity has two main objectives:

1. Create the 'Staff' table in Mangata and Gallo database.
2. Make the necessary changes to the table structure.

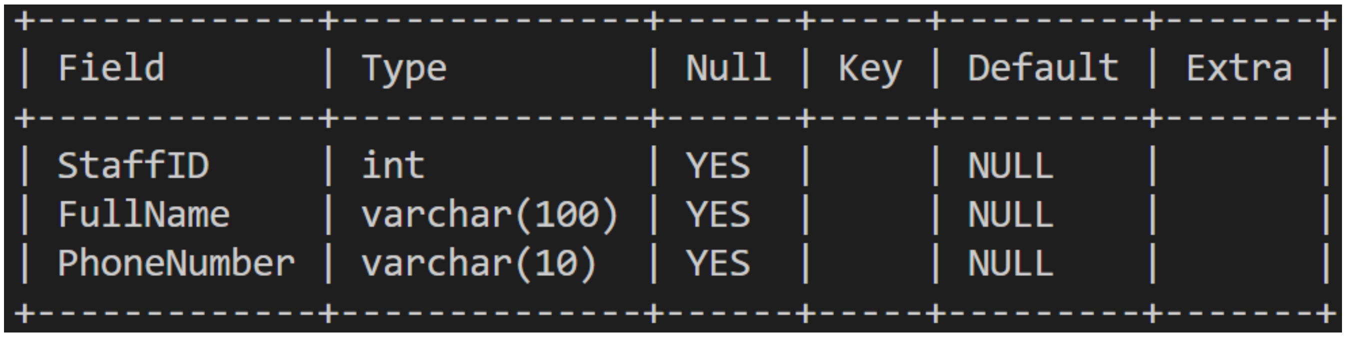
Exercise Instructions

Please attempt the following tasks.

Task 1 Write a SQL statement that creates the Staff table with the following columns.

- StaffID: INT
- FullName: VARCHAR(100)
- PhoneNumber: VARCHAR(10)

The table structure should be the same as the following screenshot (assuming that you have created and populated the table correctly.)



```
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| StaffID     | int          | YES  |     | NULL    |       |
| FullName    | varchar(100) | YES  |     | NULL    |       |
| PhoneNumber | varchar(10)  | YES  |     | NULL    |       |
+-------------+--------------+------+-----+---------+-------+
```

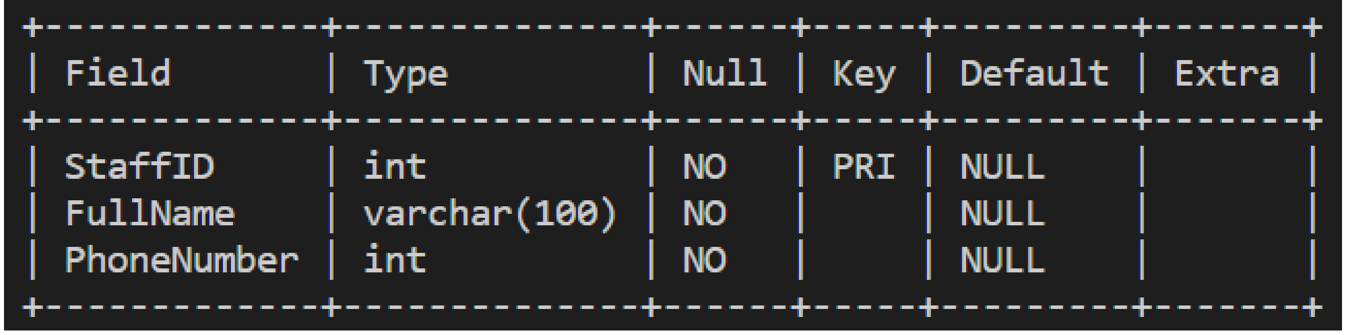
Task 2

Write a SQL statement to apply the following constraints to the Staff table:

- StaffID: INT NOT NULL and PRIMARY KEY
- FullName: VARCHAR(100) and NOT NULL
- PhoneNumber: INT NOT NULL

The expected output result should be the same as the following screenshot (assuming that you have created and populated the tables correctly.)

Loading [MathJax]/extensions/Safe.js



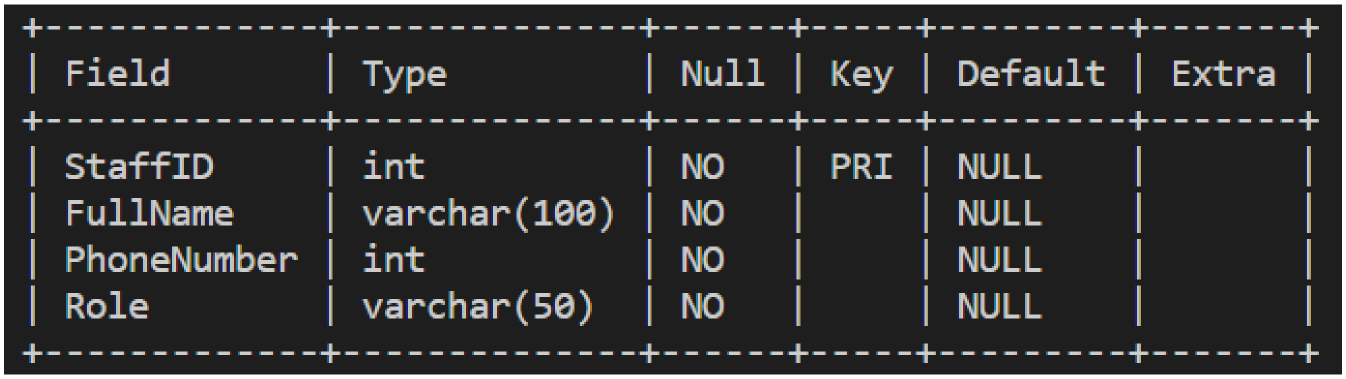
```
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| StaffID     | int          | NO   | PRI | NULL    |       |
| FullName    | varchar(100) | NO   |     | NULL    |       |
| PhoneNumber | int          | NO   |     | NULL    |       |
+-------------+--------------+------+-----+---------+-------+
```

Task 3

Write a SQL statement that adds a new column called 'Role' to the Staff table with the following constraints:

- Role: VARCHAR(50) and NOT NULL

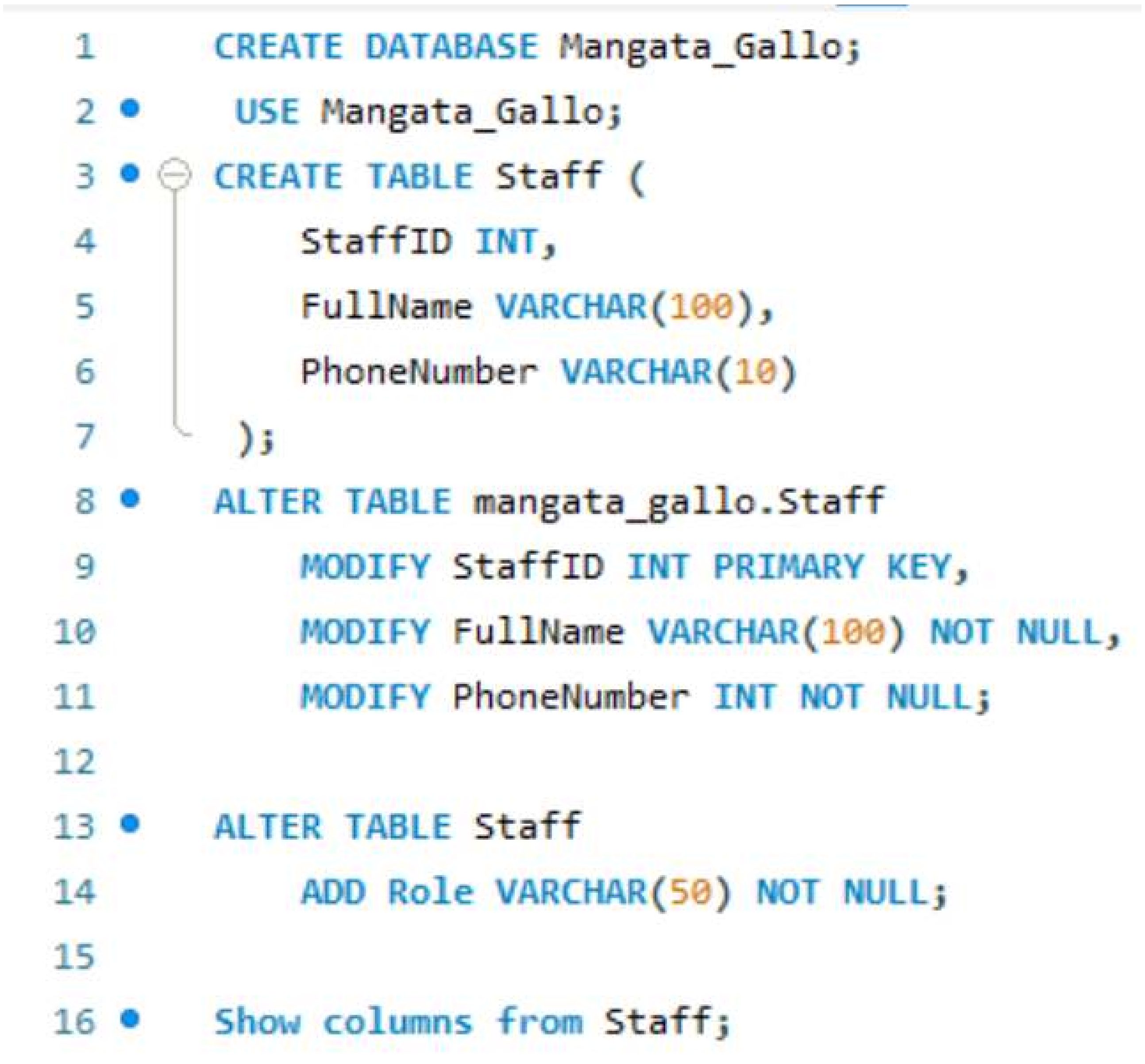
The expected output result should be the same as the following screenshot (assuming that you have created and populated the tables correctly.)



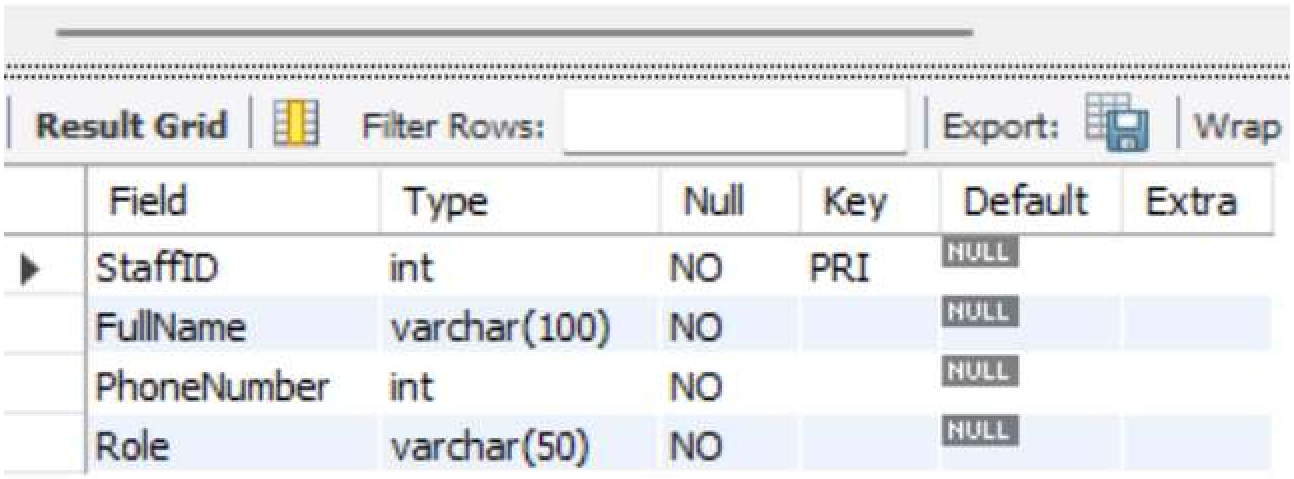
```
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| StaffID     | int          | NO   | PRI | NULL    |       |
| FullName    | varchar(100) | NO   |     | NULL    |       |
| PhoneNumber | int          | NO   |     | NULL    |       |
| Role        | varchar(50)  | NO   |     | NULL    |       |
+-------------+--------------+------+-----+---------+-------+
```

Loading [MathJax]/extensions/Safe.js

Answer:



```
CREATE DATABASE Mangata_Gallo;
 USE Mangata_Gallo;
CREATE TABLE Staff (
    StaffID INT,
    FullName VARCHAR(100),
    PhoneNumber VARCHAR(10)
 );
ALTER TABLE mangata_gallo.Staff
    MODIFY StaffID INT PRIMARY KEY,
    MODIFY FullName VARCHAR(100) NOT NULL,
    MODIFY PhoneNumber INT NOT NULL;

ALTER TABLE Staff
    ADD Role VARCHAR(50) NOT NULL;

Show columns from Staff;
```



Field	Type	Null	Key	Default	Extra
StaffID	int	NO	PRI	NULL	
FullName	varchar(100)	NO		NULL	
PhoneNumber	int	NO		NULL	
Role	varchar(50)	NO		NULL	

Reference sheet: ALTER TABLE statement and its uses

MySQL. You will repeatedly need it when developing your database.

Loading [MathJax]/extensions/Safe.js

You will find that the table you initially created in the database needs adjusting. Maybe it has some missing columns or constraints, or it may include some irrelevant ones, or you may have misnamed the table or the column, so you need to fix it.

Use the ALTER TABLE statement to solve all these problems, allowing you to change an existing table structure. It will enable you to add columns, delete columns, change column type, change column constraints, and even rename columns and the table itself.

Commands to use with the ALTER Statement

ADD

To add a column to an existing table, you can use the ADD command followed by the column name and data type. The syntax is as follows:

```
ALTER TABLE table_name ADD column_name datatype;
```

Example:

```
ALTER TABLE Employees ADD Email VARCHAR(255);
```

MODIFY

To modify a column, you can use the MODIFY command followed by the column that you want to modify.

Syntax:

```
ALTER TABLE table_name MODIFY COLUMN column_name datatype;
```

Example:

```
ALTER TABLE Employees MODIFY COLUMN Email VARCHAR(50);
```

Adding a FOREIGN KEY to an existing table using ADD

To create a link between one table and another table, you can use the FOREIGN KEY and REFERENCES commands along with the ADD command.

Syntax:

```
ALTER TABLE table_name ADD FOREIGN KEY (primary_key_column_name) REFERENCES
link_table_name(reference_column_name);
```

Example:

```
ALTER TABLE Orders ADD FOREIGN KEY (CustomerID) REFERENCES
Customer(CustomerID);
```

DROP COLUMN

To remove/delete an existing column from a table, you can use the DROP COLUMN command followed by the column that you want to modify.

Syntax:

```
ALTER TABLE table_name DROP COLUMN column_name;
```

Example:

```
ALTER TABLE Employees DROP COLUMN Email;
```

CHANGE

You can use the CHANGE command with the ALTER statement to rename a column.

Syntax:

```
ALTER TABLE table_name CHANGE from_column to_column datatype;
```

Example:

```
ALTER TABLE Employees CHANGE Email BusinessEmail VARCHAR(50);
```

RENAME

The RENAME command can be used to change a table name, followed by the new name that needs to be given to the table.

Syntax:

```
ALTER TABLE table_name RENAME new_table_name;
```

Example:

```
ALTER TABLE OrderStatus RENAME OrderDeliveryStatus;
```

Subqueries

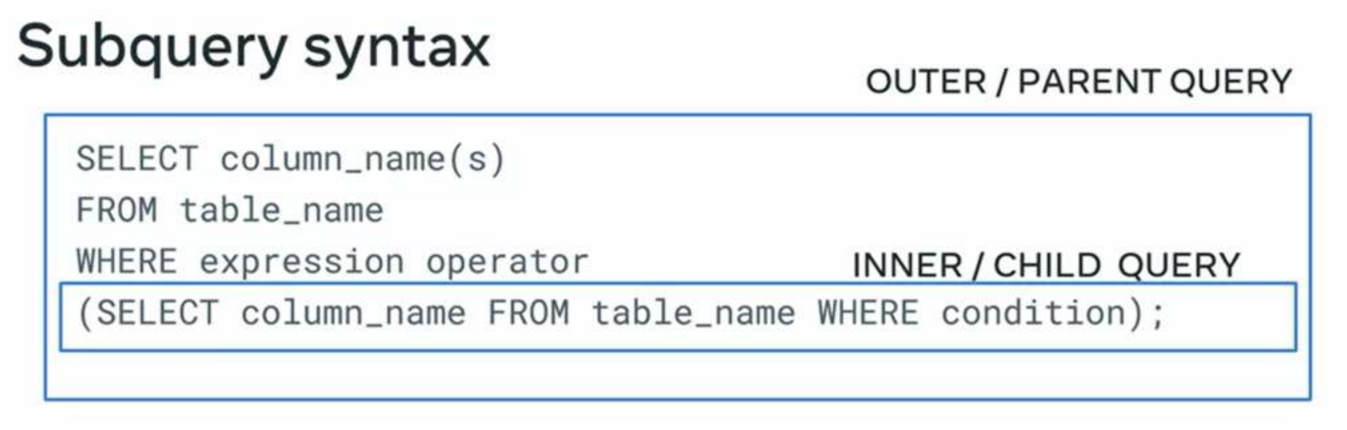
Subquery overview

1. Recognize a subquery and its syntax
2. Identify when to use a subquery
3. Explain how subqueries are used in data retrieval
4. Subqueries & complex comparison operators

Subquery - An inner query placed wihtin an outer query

Inner query is the child query

Outer query is the parent query


Subquery syntax

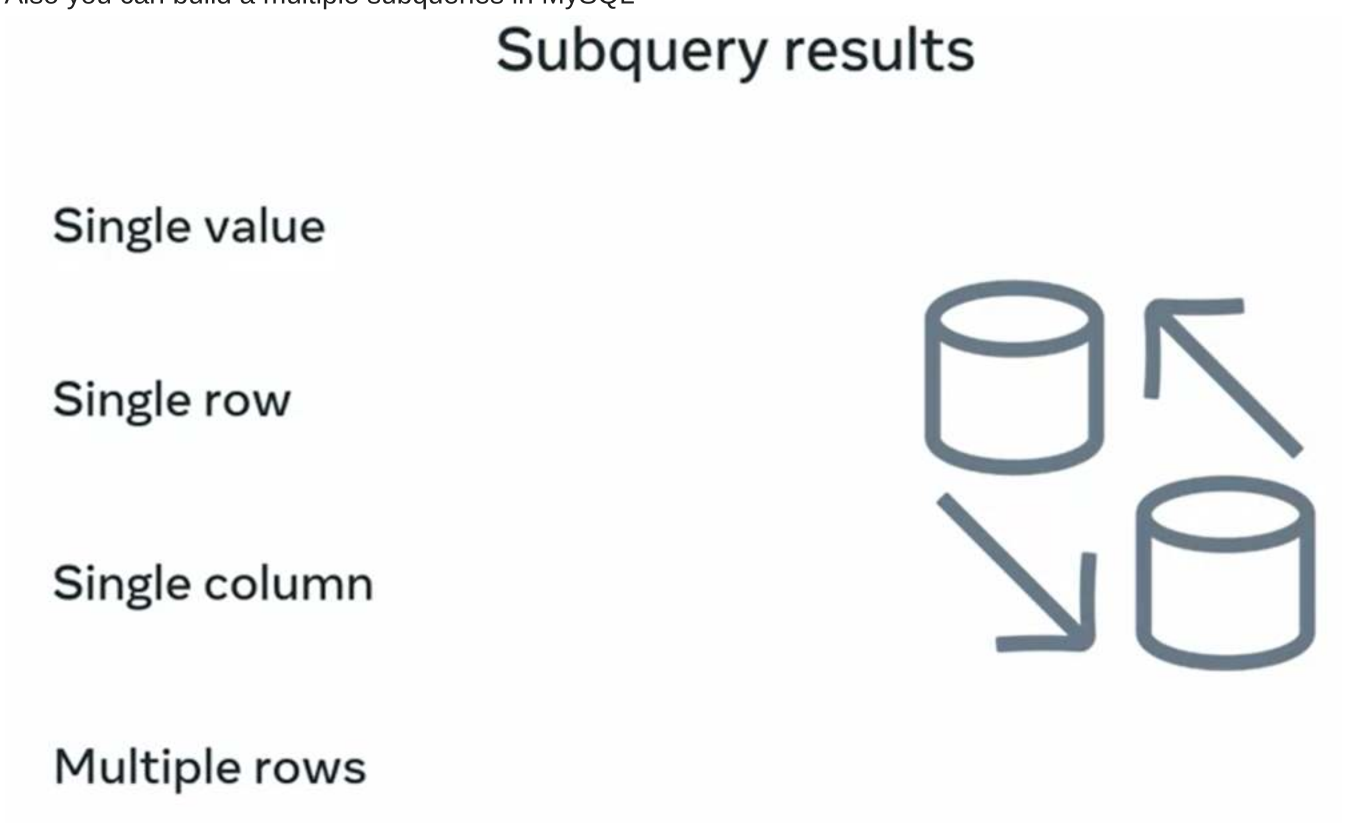
OUTER / PARENT QUERY

```
SELECT column_name(s)
FROM table_name
WHERE expression operator
(SELECT column_name FROM table_name WHERE condition);
```

INNER / CHILD QUERY

The inner query or subquery executes first and then passed to the outer or parent query

Also you can build a multiple subqueries in MySQL


Subquery results

Single value

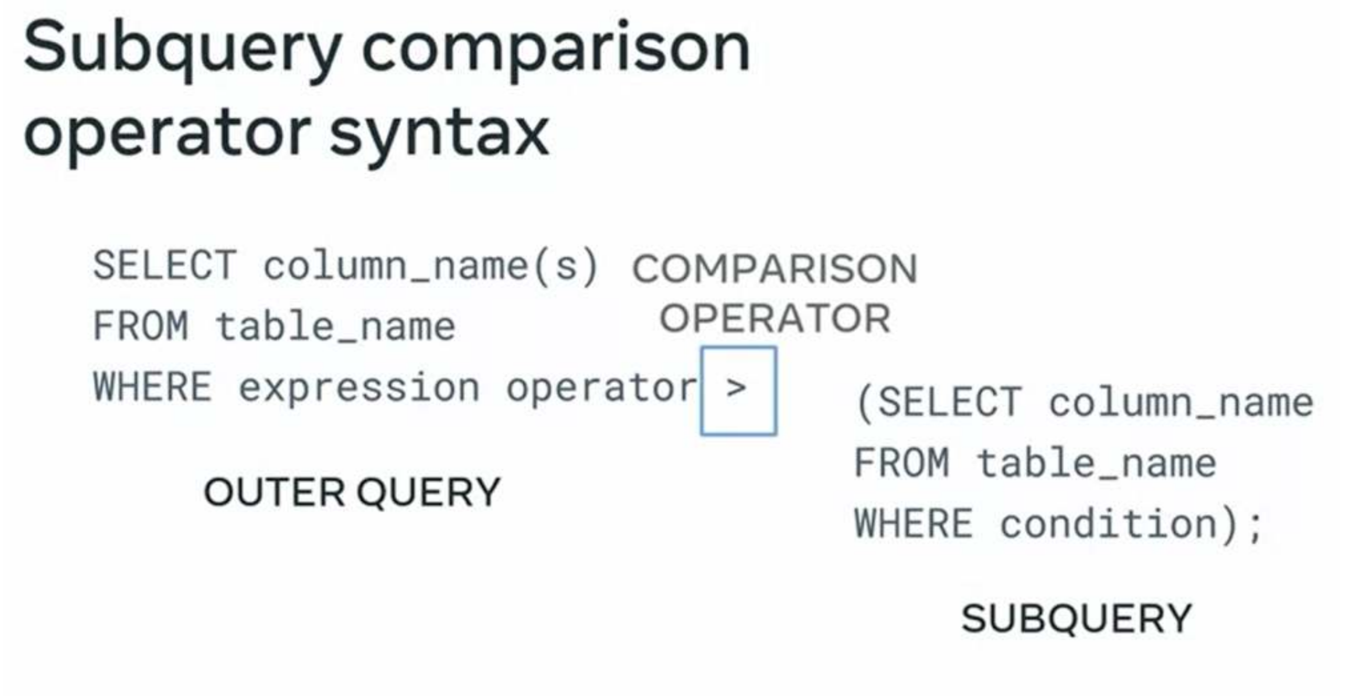
Single row

Single column

Multiple rows

Loading [MathJax]/extensions/Safe.js

A key advantage of subquery is that you can compare it against other values using a comparison operator

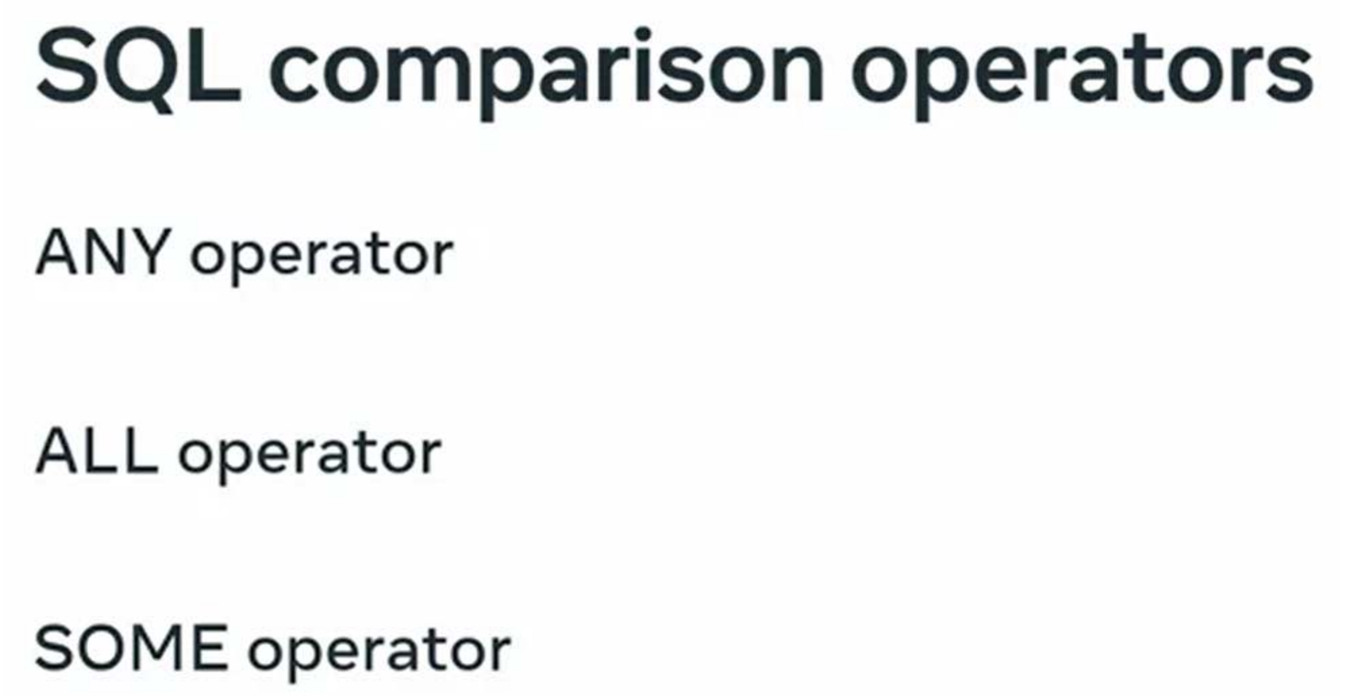


Subquery comparison operator syntax

```
SELECT column_name(s)
FROM table_name
WHERE expression operator >   (SELECT column_name
                               FROM table_name
                               WHERE condition);
```

COMPARISON OPERATOR
OUTER QUERY
SUBQUERY

Exploring subquery with complex comparison operators

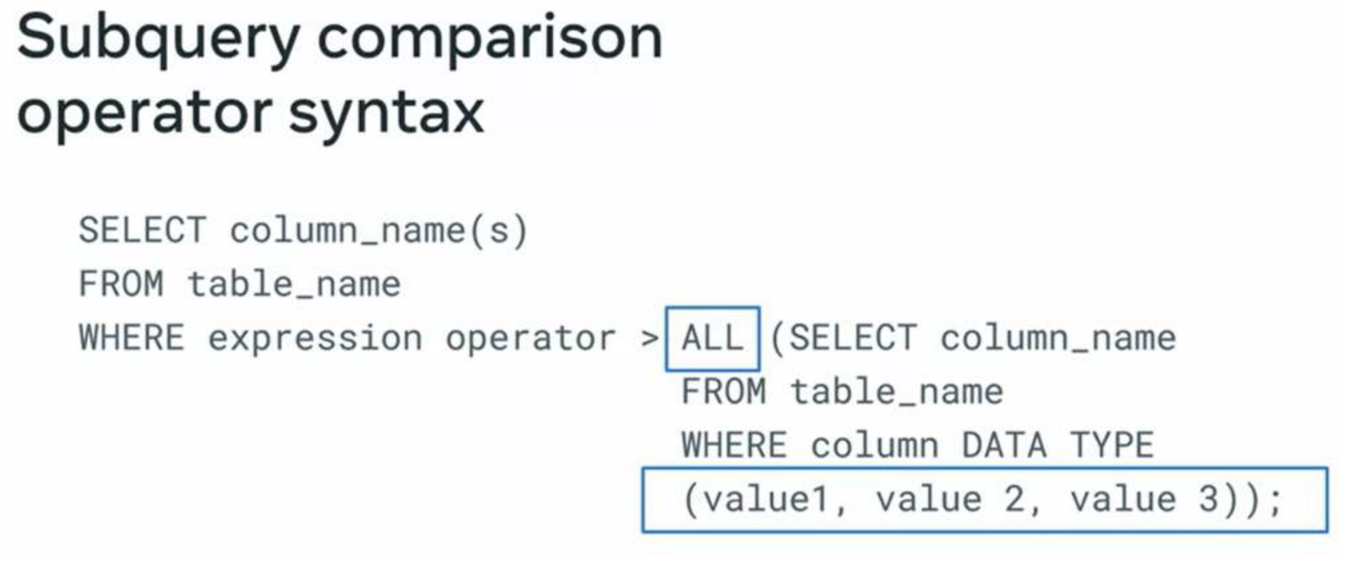


SQL comparison operators

ANY operator

ALL operator

SOME operator



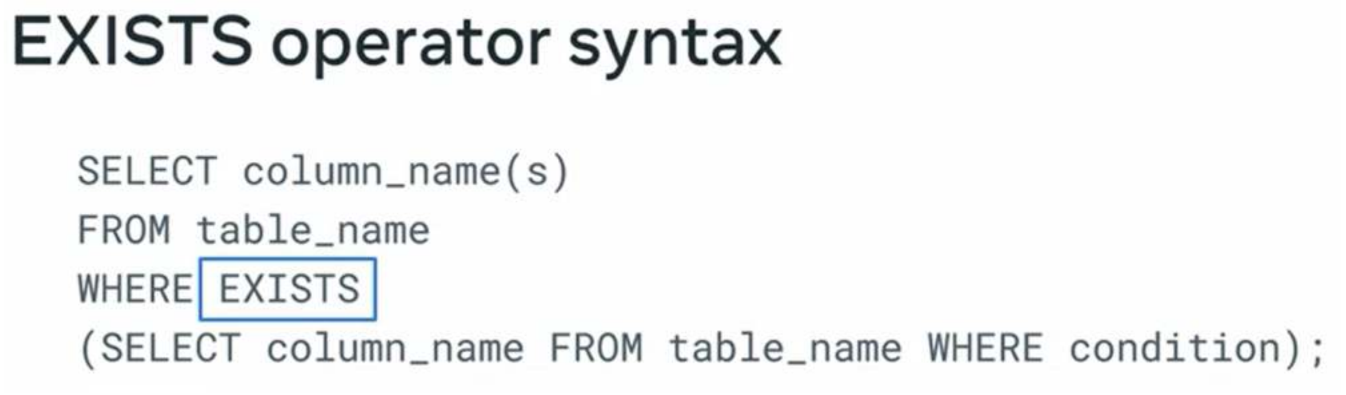
Subquery comparison operator syntax

```
SELECT column_name(s)
FROM table_name
WHERE expression operator > ALL (SELECT column_name
                                FROM table_name
                                WHERE column DATA TYPE
                                (value1, value 2, value 3));
```

Subqueries can also be used with the **EXISTS** & **NOT EXISTS** operators

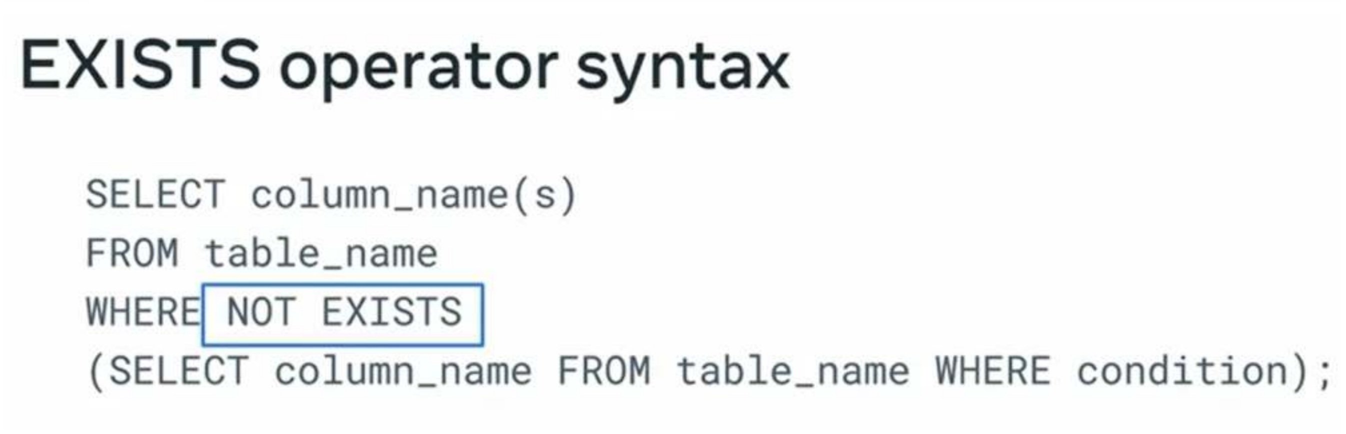
EXISTS - The EXISTS operator tests for the existence of rows in the results set returned by the sub-query. It return true if the sub-query returns one or more records

NOT EXISTS - Checks for the non-existence or absence of results from the sub-query. It returns true when the sub-query does not return any row of result



EXISTS operator syntax

```
SELECT column_name(s)
FROM table_name
WHERE EXISTS
(SELECT column_name FROM table_name WHERE condition);
```



EXISTS operator syntax

```
SELECT column_name(s)
FROM table_name
WHERE NOT EXISTS
(SELECT column_name FROM table_name WHERE condition);
```

Exercise: Working with subqueries

Prerequisites

To complete this lab, you need to have created the Little Lemon database in MySQL. You must also have populated and created the tables with relevant data inside the Little Lemon database.

If you have not created these tables in the database, the code to create the database and the tables are as follows:

1. Create database

```
CREATE DATABASE littlelemon_db;
```

1. Use database

```
USE littlelemon_db;
```

1. Create the MenuItems table

```
CREATE TABLE MenuItems (
  ItemID INT,
  Name VARCHAR(200),
  Type VARCHAR(100),
  Price INT,
  PRIMARY KEY (ItemID)
);
```

1. Create the Menus table

```
CREATE TABLE Menus (
  MenuID INT,
  ItemID INT,
  Cuisine VARCHAR(100),
  PRIMARY KEY (MenuID,ItemID)
);
```

1. Create the Bookings table

```
CREATE TABLE Bookings (
  BookingID INT,
  TableNo INT,
  GuestFirstName VARCHAR(100),
  GuestLastName VARCHAR(100),
  BookingSlot TIME,
  EmployeeID INT,
  PRIMARY KEY (BookingID)
);
```

1. Create the TableOrders table

```
CREATE TABLE TableOrders (
  OrderID INT,
  TableNo INT,
  MenuID INT,
  BookingID INT,
  BillAmount INT,
  Quantity INT,
```

Loading [MathJax]/extensions/Safe.js

```
  PRIMARY KEY (OrderID,TableNo)
);
```

1. Insert data

```
 INSERT INTO MenuItems VALUES(1,'Olives','Starters', 5),
(2,'Flatbread','Starters', 5),
(3, 'Minestrone', 'Starters', 8),
(4, 'Tomato bread','Starters', 8),
(5, 'Falafel', 'Starters', 7),
(6, 'Hummus', 'Starters', 5),
(7, 'Greek salad', 'Main Courses', 15),
(8, 'Bean soup', 'Main Courses', 12),
(9, 'Pizza', 'Main Courses', 15),
(10,'Greek yoghurt','Desserts', 7),
(11, 'Ice cream', 'Desserts', 6),
(12, 'Cheesecake', 'Desserts', 4),
(13, 'Athens White wine', 'Drinks', 25),
(14, 'Corfu Red Wine', 'Drinks', 30),
(15, 'Turkish Coffee', 'Drinks', 10),
(16, 'Turkish Coffee', 'Drinks', 10),
(17, 'Kabasa', 'Main Courses', 17);
```

```
 INSERT INTO Menus VALUES(1, 1, 'Greek'), (1, 7, 'Greek'), (1, 10, 'Greek'),
(1, 13, 'Greek'), (2, 3, 'Italian'), (2, 9, 'Italian'), (2, 12, 'Italian'), (2,
15, 'Italian'), (3, 5, 'Turkish'), (3, 17, 'Turkish'), (3, 11, 'Turkish'), (3,
16, 'Turkish');
```

```
 INSERT INTO Bookings VALUES(1,12,'Anna','Iversen','19:00:00',1),
(2, 12, 'Joakim', 'Iversen', '19:00:00', 1), (3, 19, 'Vanessa', 'McCarthy',
'15:00:00', 3),
(4, 15, 'Marcos', 'Romero', '17:30:00', 4), (5, 5, 'Hiroki', 'Yamane',
'18:30:00', 2),
(6, 8, 'Diana', 'Pinto', '20:00:00', 5);
```

```
 INSERT INTO TableOrders VALUES(1, 12, 1, 1, 2, 86), (2, 19, 2, 2, 1, 37), (3,
15, 2, 3, 1, 37), (4, 5, 3, 4, 1, 40), (5, 8, 1, 5, 1, 43);
```

There are two main objectives in this activity:

1. Working with single row, multiple row and correlated subqueries.

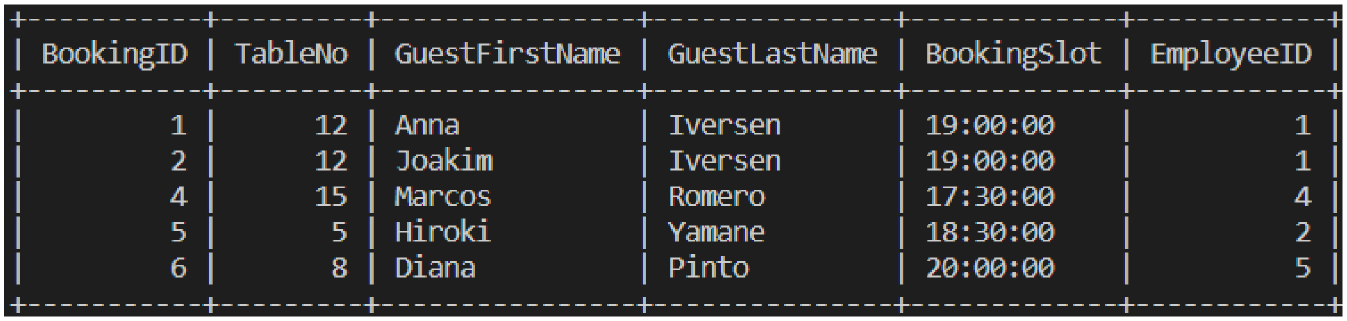
2. Using the comparison operators and the ALL and NOT EXISTS operators with subqueries.

Exercise Instructions

Please attempt the following tasks.

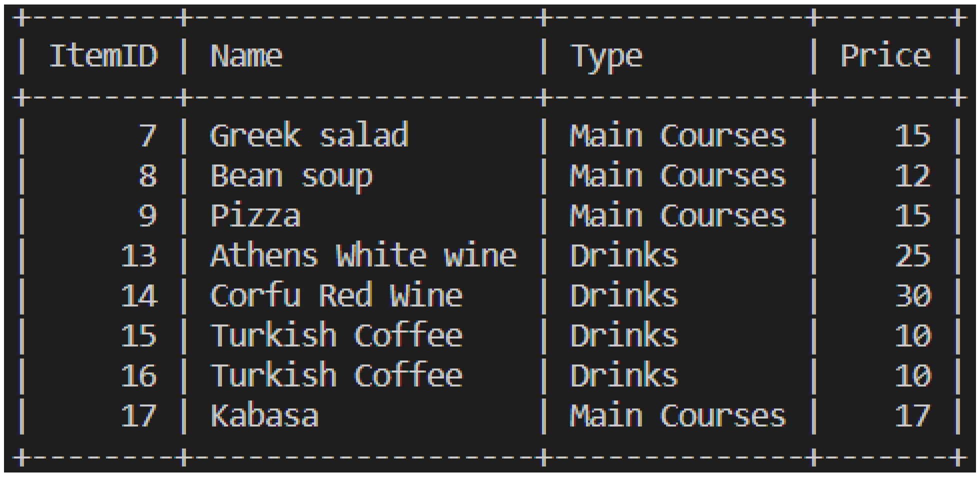
Task 1: Write a SQL SELECT query to find all bookings that are due after the booking of the guest ‘Vanessa McCarthy’.

The expected output result should be the same as the following screenshot (assuming that you have created and populated the orders tables correctly).


```
+-----------+---------+----------------+---------------+-------------+------------+
| BookingID | TableNo | GuestFirstName | GuestLastName | BookingSlot | EmployeeID |
+-----------+---------+----------------+---------------+-------------+------------+
|         1 |      12 | Anna           | Iversen       | 19:00:00    |          1 |
|         2 |      12 | Joakim         | Iversen       | 19:00:00    |          1 |
|         4 |      15 | Marcos         | Romero        | 17:30:00    |          4 |
|         5 |       5 | Hiroki         | Yamane        | 18:30:00    |          2 |
|         6 |       8 | Diana          | Pinto         | 20:00:00    |          5 |
+-----------+---------+----------------+---------------+-------------+------------+
```

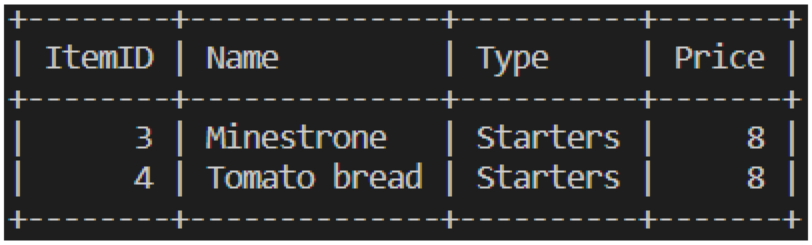
Task 2: Write a SQL SELECT query to find the menu items that are more expensive than all the 'Starters' and 'Desserts' menu item types.

The expected output result should be the same as the following screenshot (assuming that you have created and populated the orders tables correctly).


```
+--------+-------------------+--------------+-------+
| ItemID | Name              | Type         | Price |
+--------+-------------------+--------------+-------+
|      7 | Greek salad       | Main Courses |    15 |
|      8 | Bean soup         | Main Courses |    12 |
|      9 | Pizza             | Main Courses |    15 |
|     13 | Athens White wine | Drinks       |    25 |
|     14 | Corfu Red Wine    | Drinks       |    30 |
|     15 | Turkish Coffee    | Drinks       |    10 |
|     16 | Turkish Coffee    | Drinks       |    10 |
|     17 | Kabasa            | Main Courses |    17 |
+--------+-------------------+--------------+-------+
```

Task 3: Write a SQL SELECT query to find the menu items that costs the same as the starter menu items that are Italian cuisine.

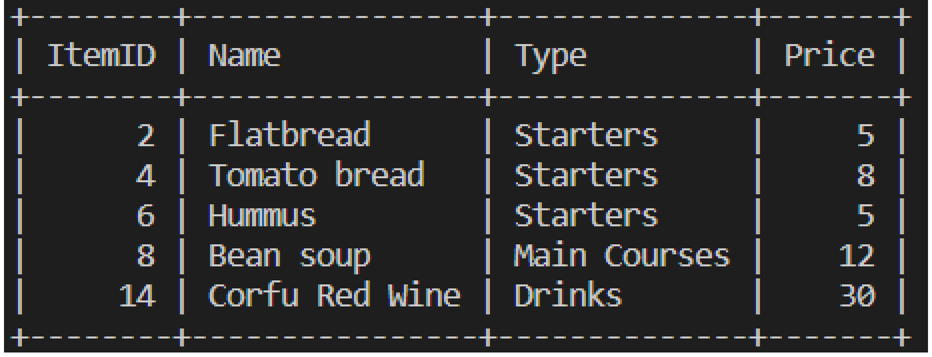
The expected output result should be the same as the following screenshot (assuming that you have created and populated the orders tables correctly).


```
+--------+--------------+----------+-------+
| ItemID | Name         | Type     | Price |
+--------+--------------+----------+-------+
|      3 | Minestrone   | Starters |     8 |
|      4 | Tomato bread | Starters |     8 |
+--------+--------------+----------+-------+
```

Task 4: Write a SQL SELECT query to find the menu items that were not ordered by the guests who placed bookings.

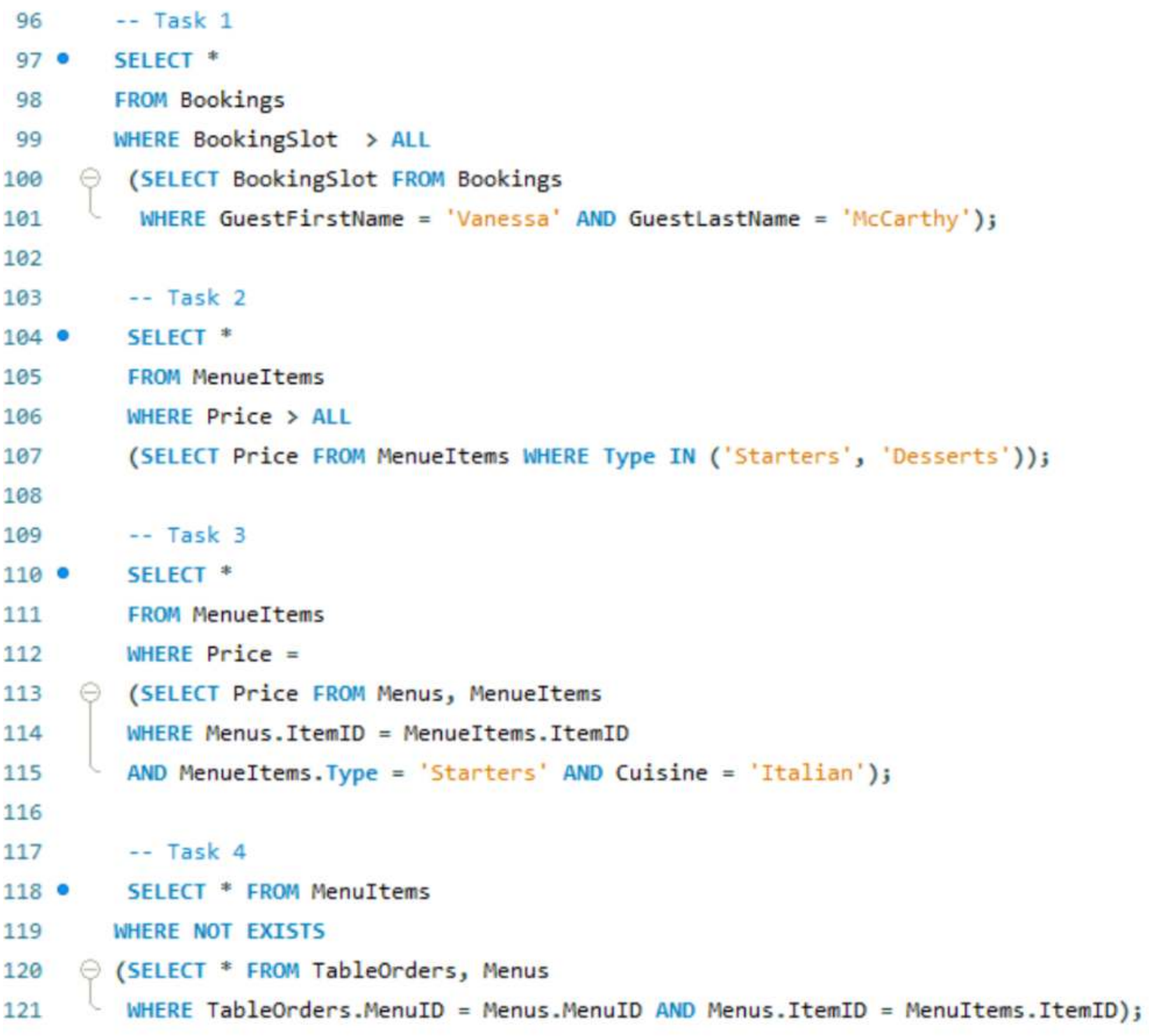
The expected output result should be the same as the following screenshot (assuming that you have created and populated the orders tables correctly).

Loading [MathJax]/extensions/Safe.js



```
+--------+----------------+---------------+-------+
| ItemID | Name           | Type          | Price |
+--------+----------------+---------------+-------+
|      2 | Flatbread      | Starters      |     5 |
|      4 | Tomato bread   | Starters      |     8 |
|      6 | Hummus         | Starters      |     5 |
|      8 | Bean soup      | Main Courses  |    12 |
|     14 | Corfu Red Wine | Drinks        |    30 |
+--------+----------------+---------------+-------+
```

Answer:



```sql
-- Task 1
SELECT *
FROM Bookings
WHERE BookingSlot  > ALL
 (SELECT BookingSlot FROM Bookings
  WHERE GuestFirstName = 'Vanessa' AND GuestLastName = 'McCarthy');

 -- Task 2
 SELECT *
 FROM MenueItems
 WHERE Price > ALL
 (SELECT Price FROM MenueItems WHERE Type IN ('Starters', 'Desserts'));

 -- Task 3
 SELECT *
 FROM MenueItems
 WHERE Price =
 (SELECT Price FROM Menus, MenueItems
 WHERE Menus.ItemID = MenueItems.ItemID
 AND MenueItems.Type = 'Starters' AND Cuisine = 'Italian');

 -- Task 4
 SELECT * FROM MenuItems
WHERE NOT EXISTS
(SELECT * FROM TableOrders, Menus
 WHERE TableOrders.MenuID = Menus.MenuID AND Menus.ItemID = MenuItems.ItemID);
```

Reading exercise: Practicing subqueries

Based in Chicago, Illinois, Little Lemon is a family-owned Mediterranean restaurant focused on traditional recipes served with a modern twist.

You can use the provided SQL scripts to create and set up their database.

Instructions

To complete this exercise, you can either keep the MySQL terminal open from the previous lab, or use MySQL on your own machine. To install MySQL on your own machine you can follow the instructions provided in the link in the additional resources item in the first module of this course.

Prerequisites

Here is the code to create and use the littlelemon_db database:

```
CREATE DATABASE littlelemon_db;
```

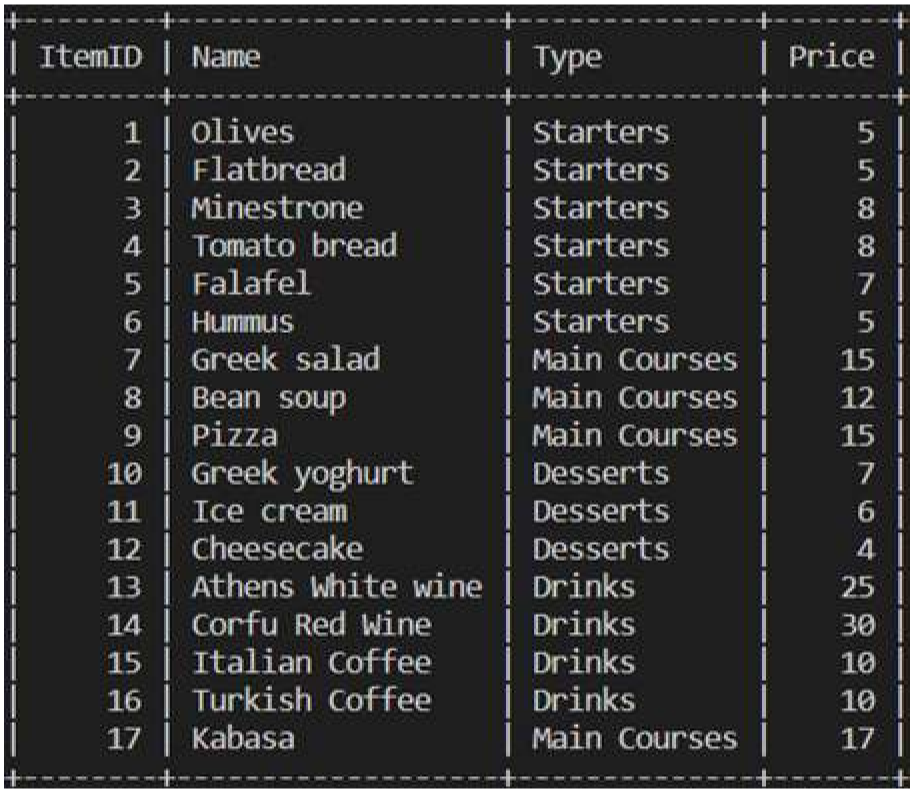
```
USE littlelemon_db;
```

And the code to populate the MenuItems table with relevant data.

```
CREATE TABLE MenuItems (
  ItemID int NOT NULL,
  Name varchar(200) DEFAULT NULL,
  Type varchar(100) DEFAULT NULL,
  Price int DEFAULT NULL,
  PRIMARY KEY (ItemID)
);
```

```
INSERT INTO MenuItems VALUES
(1,'Olives','Starters',5),
(2,'Flatbread','Starters',5),
(3,'Minestrone','Starters',8),
(4,'Tomato bread','Starters',8),
(5,'Falafel','Starters',7),
(6,'Hummus','Starters',5),
(7,'Greek salad','Main Courses',15),
(8,'Bean soup','Main Courses',12,),
(9,'Pizza','Main Courses',15),
(10,'Greek yoghurt','Desserts',7),
(11,'Ice cream','Desserts',6),
(12,'Cheesecake','Desserts',4),
(13,'Athens White wine','Drinks',25),
(14,'Corfu Red Wine','Drinks',30),
(15,'Turkish Coffee','Drinks',10),
(16,'Turkish Coffee','Drinks',10),
(17,'Kabasa','Main Courses',17);
```

The content of the MenuItems table.



```
+--------+--------------------+--------------+-------+
| ItemID | Name               | Type         | Price |
+--------+--------------------+--------------+-------+
|      1 | Olives             | Starters     |     5 |
|      2 | Flatbread          | Starters     |     5 |
|      3 | Minestrone         | Starters     |     8 |
|      4 | Tomato bread       | Starters     |     8 |
|      5 | Falafel            | Starters     |     7 |
|      6 | Hummus             | Starters     |     5 |
|      7 | Greek salad        | Main Courses |    15 |
|      8 | Bean soup          | Main Courses |    12 |
|      9 | Pizza              | Main Courses |    15 |
|     10 | Greek yoghurt      | Desserts     |     7 |
|     11 | Ice cream          | Desserts     |     6 |
|     12 | Cheesecake         | Desserts     |     4 |
|     13 | Athens White wine  | Drinks       |    25 |
|     14 | Corfu Red Wine     | Drinks       |    30 |
|     15 | Italian Coffee     | Drinks       |    10 |
|     16 | Turkish Coffee     | Drinks       |    10 |
|     17 | Kabasa             | Main Courses |    17 |
+--------+--------------------+--------------+-------+
```

The code to create the LowCostMenuItems table.

```
(ItemID INT, Name VARCHAR(200), Price INT, PRIMARY KEY(ItemID));
```

Tasks

Task 1: Find the minimum and the maximum average prices at which the customers can purchase food and drinks.

Hint: In this task, you must write subqueries using the FROM clause. Your subquery would find the average prices of menu items by their type. The subquery result will be the input for the outer query to find the minimum and maximum average prices.

Task 2: Insert data of menu items with a minimum price based on the 'Type' into the LowCostMenuItems table.

Hint: In this task, you must write subqueries in INSERT statements. Your subquery would find the details of menu items with a minimum price based on the 'Type' of menu item. In other words, GROUP BY Type. Then you can insert the data retrieved from the subquery into the LowCostMenuItems table using an INSERT INTO SELECT statement.

Task 3: Delete all the low-cost menu items whose price is more than the minimum price of menu items that have a price between $5 and $10.

Hint: You need to write subqueries in DELETE statements in this task. Your subquery will be placed in the WHERE clause of the DELETE statement to find the minimum prices of menu items that have a price between $5 and $10. Use the ALL operator in the outer query to find matches from the values returned from the subquery. Delete those records with matching prices from the LowCostMenuItems table.

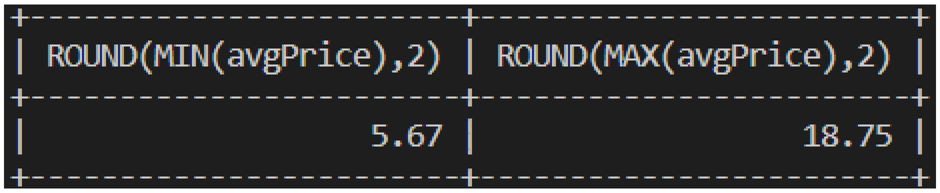
Task 1 solution: Find the minimum and the maximum average prices at which the customers can purchase food and drinks.

SELECT Type, AVG(Price) AS avgPrice FROM MenuItems GROUP BY Type;

It can, in turn, be the source from which data (average price) is used to find the min and max average prices. So, you write this outer SELECT statement and add the above query as a subquery in its FROM clause as follows:

SELECT ROUND(MIN(avgPrice),2), ROUND(MAX(avgPrice),2) FROM (SELECT Type,AVG(Price) AS avgPrice FROM MenuItems GROUP BY Type) AS aPrice;

The highlighted subquery in the FROM clause behaves as a temporary table in this case, and it becomes the source or target table for the outer query.

The result is as follows: 

```
+-----------------------+-----------------------+
| ROUND(MIN(avgPrice),2) | ROUND(MAX(avgPrice),2) |
+-----------------------+-----------------------+
|                  5.67 |                 18.75 |
+-----------------------+-----------------------+
```

So, the minimum average price at which customers can buy food/drink from Little Lemon is 5.67 and the maximum average price is $ 18.75.

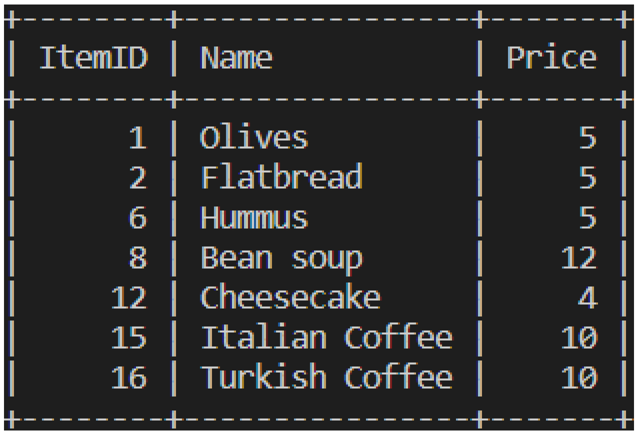
Task 2 solution: Insert data of menu items with a minimum price based on the 'Type' into the LowCostMenuItems table.

Here, the SELECT statement in INSERT INTO, contains the subquery.

INSERT INTO LowCostMenuItems SELECT ItemID,Name,Price FROM MenuItems WHERE Price =ANY(SELECT MIN(Price) FROM MenuItems GROUP BY Type);

The outer SELECT query uses the ANY operator to find ANY matches from the values returned from the subquery – which gives a list of minimum prices of menu items based on their types. If it finds any matches, it will insert them into the LowCostMenuItems table.

The data inserted in the LowCostMenuItems looks like this once you run this:



```
+--------+----------------+-------+
| ItemID | Name           | Price |
+--------+----------------+-------+
|      1 | Olives         |     5 |
|      2 | Flatbread      |     5 |
|      6 | Hummus         |     5 |
|      8 | Bean soup      |    12 |
|     12 | Cheesecake     |     4 |
|     15 | Italian Coffee |    10 |
|     16 | Turkish Coffee |    10 |
+--------+----------------+-------+
```

Task 3 solution: Delete all the low-cost menu items whose price is more than the minimum price of menu items that have a price between $5 and $10.

DELETE FROM LowCostMenuItems WHERE Price > ALL(SELECT MIN(Price) as p FROM MenuItems GROUP BY Type HAVING p BETWEEN 5 AND 10);

The outer query uses the ALL operator to find matches from the values returned from the subquery. This provides a list of minimum prices of menu items whose price is between $5 and $10. If it finds any matches

Loading [MathJax]/extensions/Safe.js

then it deletes those records with matching prices from the LowCostMenuItems table.

Reference sheet: Subqueries

What is a sub-query?

A sub-query is **a query within another query**. In other words, an inner query. In a sub-query, a SQL SELECT query is placed as a part of another query called an outer query. Think of the inner query as the child query and the outer query as the parent query.

Place a sub-query in the SELECT, FROM, or WHERE clause. You will find it used in the WHERE clause most of the time. Also, always place a subquery within a pair of parentheses. There can be sub-queries that return a single value, a single row, a single column, or multiple rows of one or more columns. When writing subqueries, it is possible to use comparison operators such as =, >, >=, <, <=, != (or <>). In addition, you can also use ALL, ANY and SOME operators followed by a comparison operator.

Types of sub-queries

Subqueries in the FROM clause

You already know that in SQL, in the FROM clause, you specify the table name in which the data in question resides. When you use a sub-query in the FROM clause, the result returned from a sub-query is used as a temporary table.

This table is referred to as a derived table as well.

Here’s what the syntax of a sub-query in the FROM clause looks like:

`SELECT some_column, some_column FRO**M (Subquery) AS alias;`

**Sub-queries in INSERT statements

You already know that you can use INSERT INTO SELECT statements. That means to use SELECT statements in an INSERT statement when you want to retrieve some data and insert them into another table.

The SELECT statement here can contain a sub-query.

Here’s what the syntax of a sub-query in an INSERT statement looks like:

`INSERT INTO table1 SELECT column1,column2,column3 FROM table2 WHERE some_column some_operator(Subquery);`

Sub-queries in UPDATE statements

An update statement can also use a subquery. In the UPDATE statement’s WHERE clause, you can use a subquery to filter out the records you want to update.

Here’s what the syntax of a subquery in an UPDATE statement looks like:

`UPDATE table1 SET column1 = some_value WHERE some_column some_operator(Sub-query);`

Sub-queries in DELETE statements

You can also use sub-queries in your DELETE statement. Once again, in the WHERE clause of the DELETE statement, you can use a subquery to filter out the records you want to delete.

Here’s what the syntax of a subquery in a DELETE statement looks like:

```
DELETE FROM table1 WHERE some_column some_operator(Subquery);
```

Virtual Tables

MySQL CREATE VIEW

Views: Vitual tables used for accessing and manipulating data with MySQL

Why do you need to use virtual tables?

You have a database with a base table with 7 columns named: column 1, column 2, column 3, column 4, column 5, column 6 and column 7.

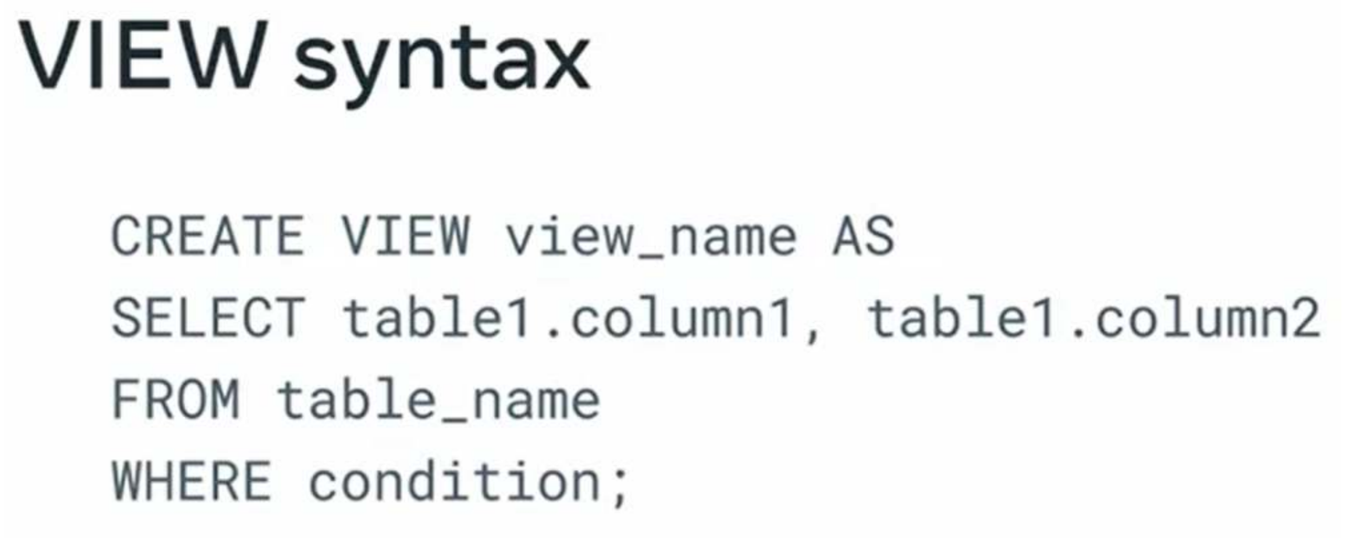
However, you are only interested in viewing and analyzing information in columns 3, column 4 and column 5. In this case, you can create a virtual table that contains the three required columns. This virtual table utilizes the data that exists in the corresponding columns in the base table, as presented in the following illustration.

Common use cases for views:

1. Createe a subset: Focus on a subset of a table's data
2. Combined data: Combine data from multiple tables into one\

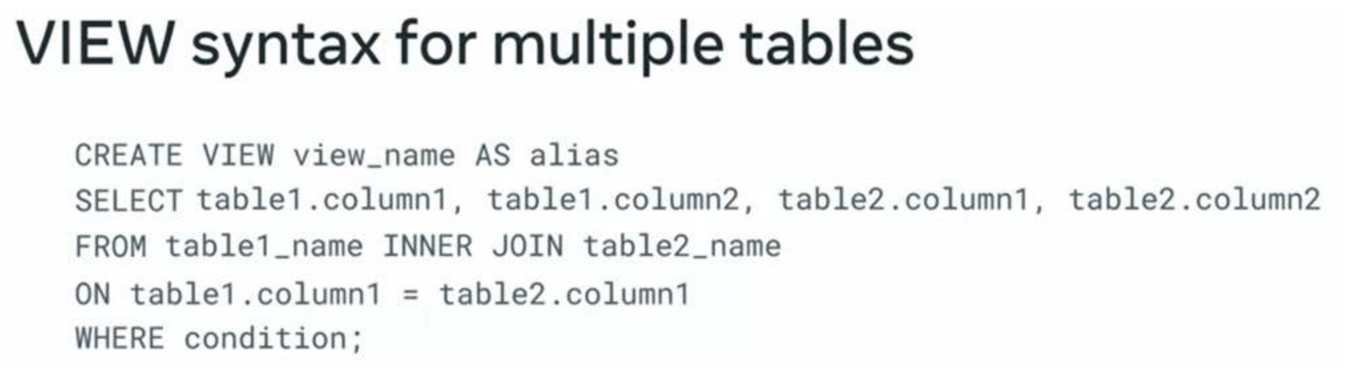
Creating a virtual table:

1. Create the vitual table
2. Select the table's columns
3. Specify a table to extract data from
4. Set the conditions
5. Set data order and filtering rule



VIEW syntax

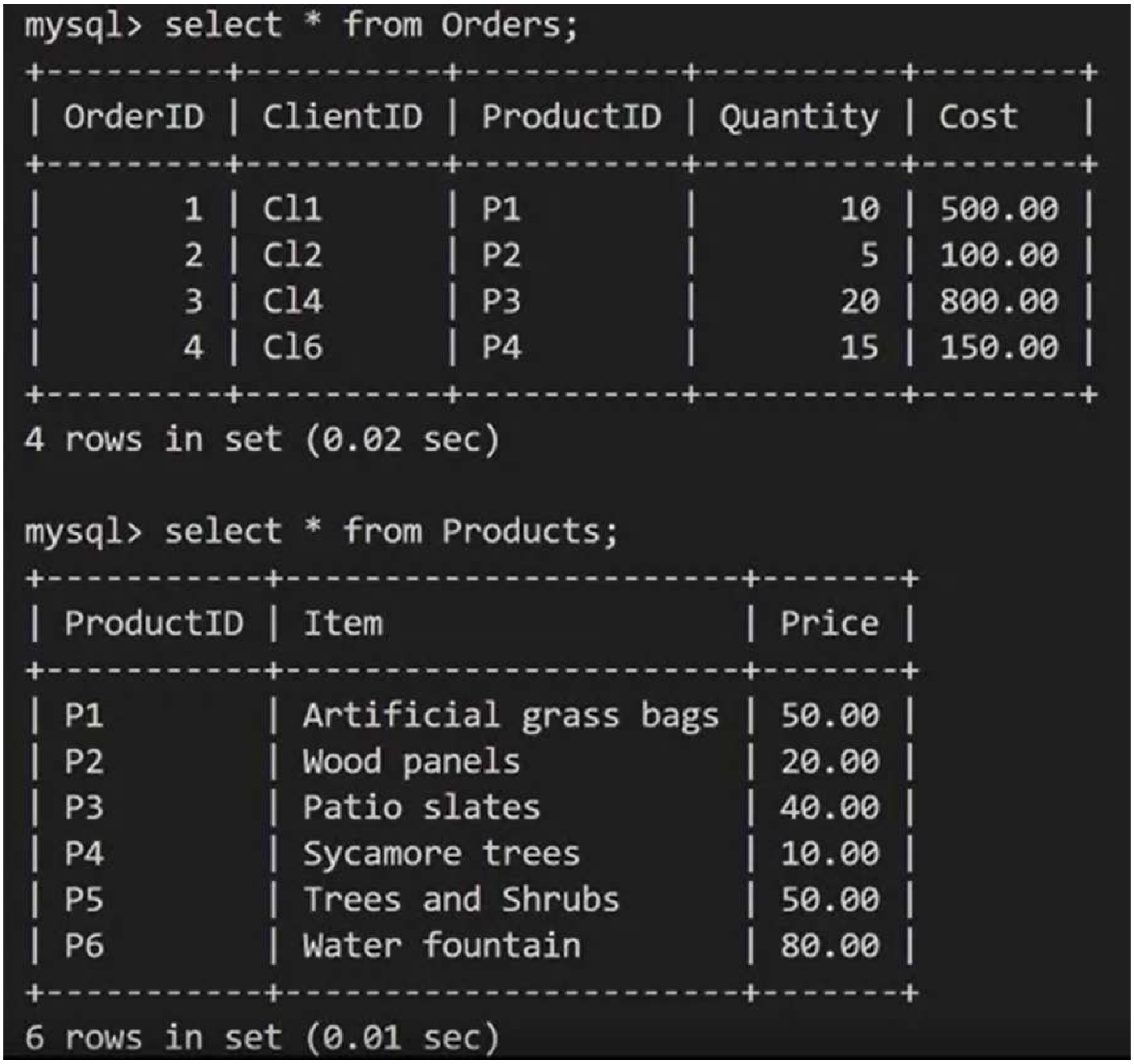
```
CREATE VIEW view_name AS
SELECT table1.column1, table1.column2
FROM table_name
WHERE condition;
```



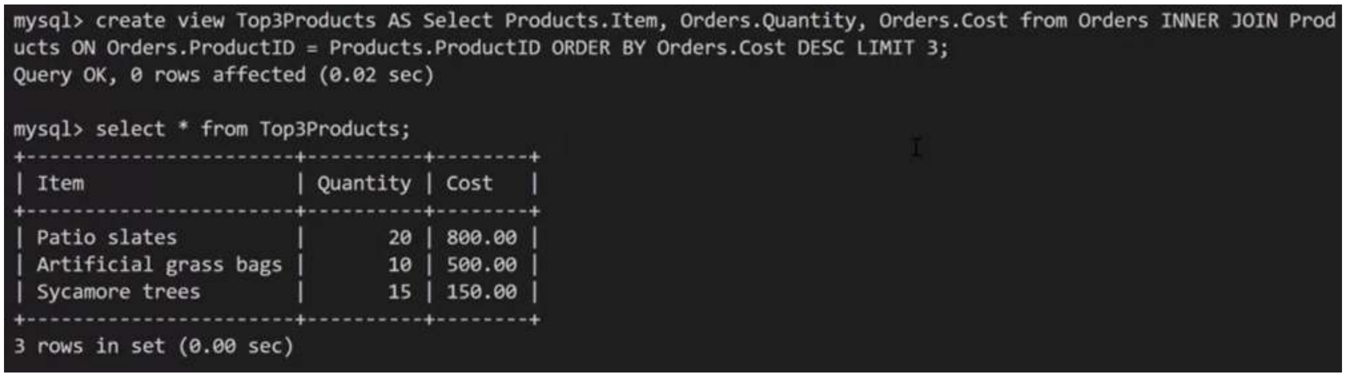
VIEW syntax for multiple tables

```
CREATE VIEW view_name AS alias
SELECT table1.column1, table1.column2, table2.column1, table2.column2
FROM table1_name INNER JOIN table2_name
ON table1.column1 = table2.column1
WHERE condition;
```

Let's take an example:



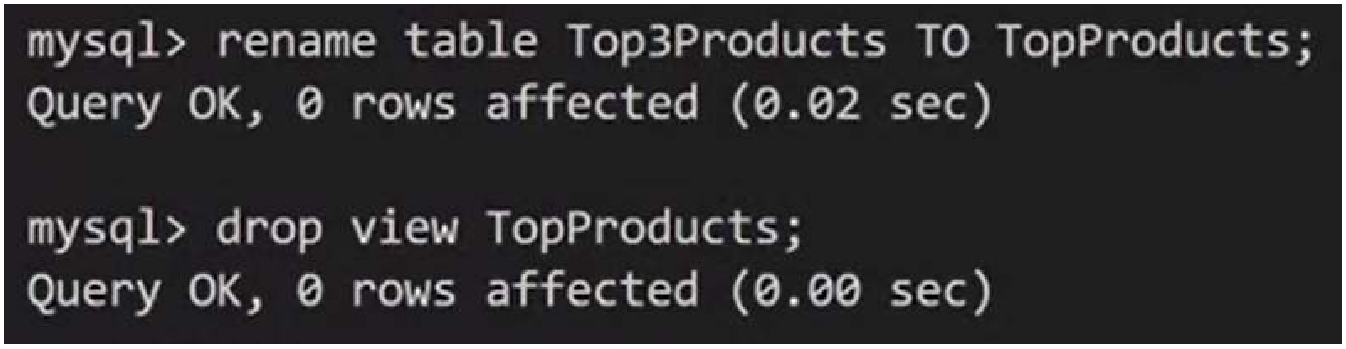
```
mysql> select * from Orders;
+---------+----------+-----------+----------+--------+
| OrderID | ClientID | ProductID | Quantity | Cost   |
+---------+----------+-----------+----------+--------+
|       1 | Cl1      | P1        |       10 | 500.00 |
|       2 | Cl2      | P2        |        5 | 100.00 |
|       3 | Cl4      | P3        |       20 | 800.00 |
|       4 | Cl6      | P4        |       15 | 150.00 |
+---------+----------+-----------+----------+--------+
4 rows in set (0.02 sec)

mysql> select * from Products;
+-----------+----------------------+-------+
| ProductID | Item                 | Price |
+-----------+----------------------+-------+
| P1        | Artificial grass bags | 50.00 |
| P2        | Wood panels          | 20.00 |
| P3        | Patio slates         | 40.00 |
| P4        | Sycamore trees       | 10.00 |
| P5        | Trees and Shrubs     | 50.00 |
| P6        | Water fountain       | 80.00 |
+-----------+----------------------+-------+
6 rows in set (0.01 sec)
```



```
mysql> create view Top3Products AS Select Products.Item, Orders.Quantity, Orders.Cost from Orders INNER JOIN Prod
ucts ON Orders.ProductID = Products.ProductID ORDER BY Orders.Cost DESC LIMIT 3;
Query OK, 0 rows affected (0.02 sec)

mysql> select * from Top3Products;
+-----------------------+----------+--------+
| Item                  | Quantity | Cost   |
+-----------------------+----------+--------+
| Patio slates          |       20 | 800.00 |
| Artificial grass bags |       10 | 500.00 |
| Sycamore trees        |       15 | 150.00 |
+-----------------------+----------+--------+
3 rows in set (0.00 sec)
```

Loading [MathJax]/extensions/Safe.js

To rename or drop the view:



```
mysql> rename table Top3Products TO TopProducts;
Query OK, 0 rows affected (0.02 sec)

mysql> drop view TopProducts;
Query OK, 0 rows affected (0.00 sec)
```

Functions in MySQL

A code that performs an operation and returns a result, often with the use of parameters.

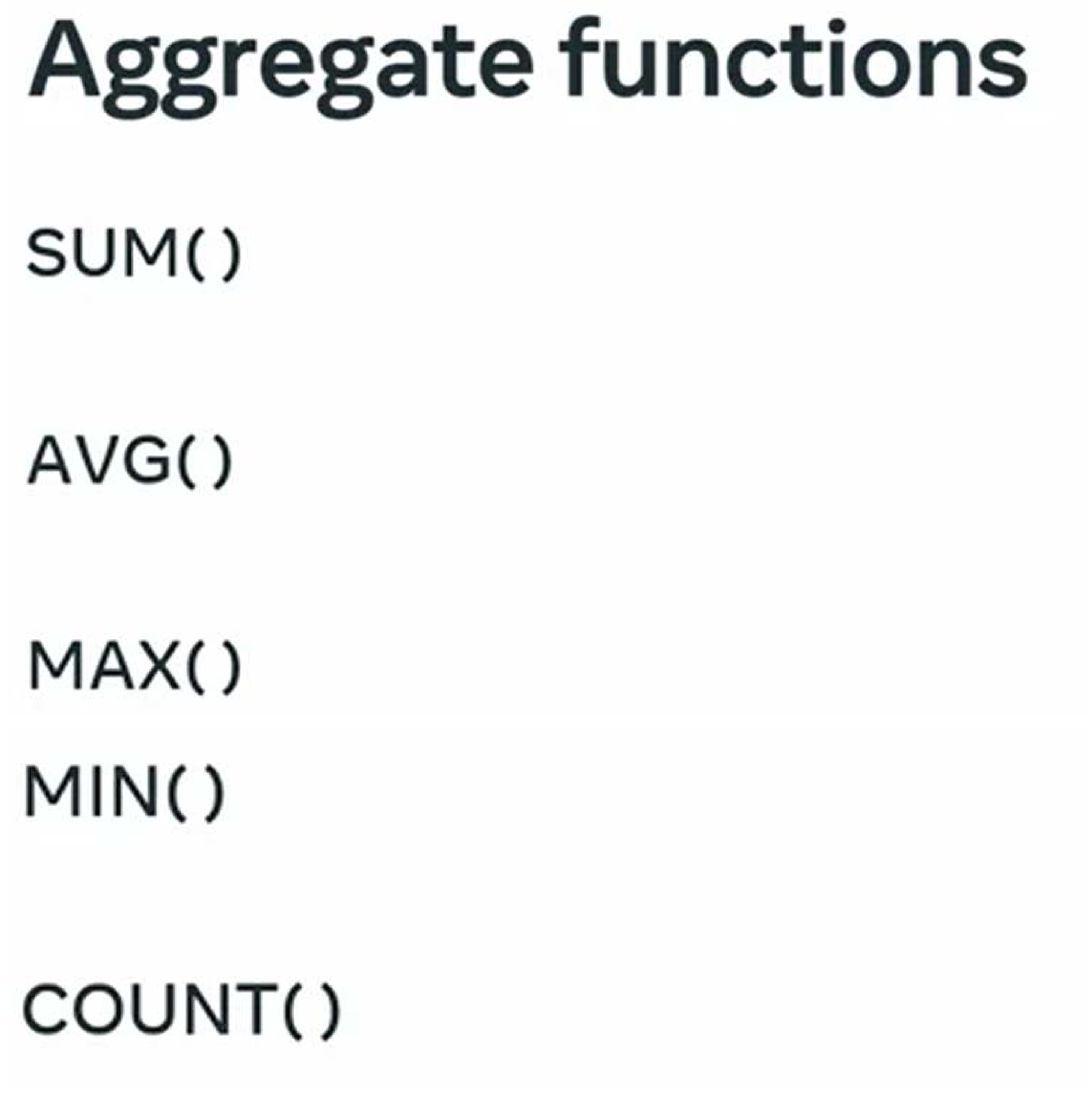
Functions overview:

1. Numric functions
2. String functions
3. Date functions
4. Comparison functions
5. Control flow functions

$Numrice functions$- can be divided into two categories:

1. Aggregate functions: Perform tasks on a set of values
2. Math functions: Perform basic mathmatical tasks on data

Loading [MathJax]/extensions/Safe.js



Aggregate functions

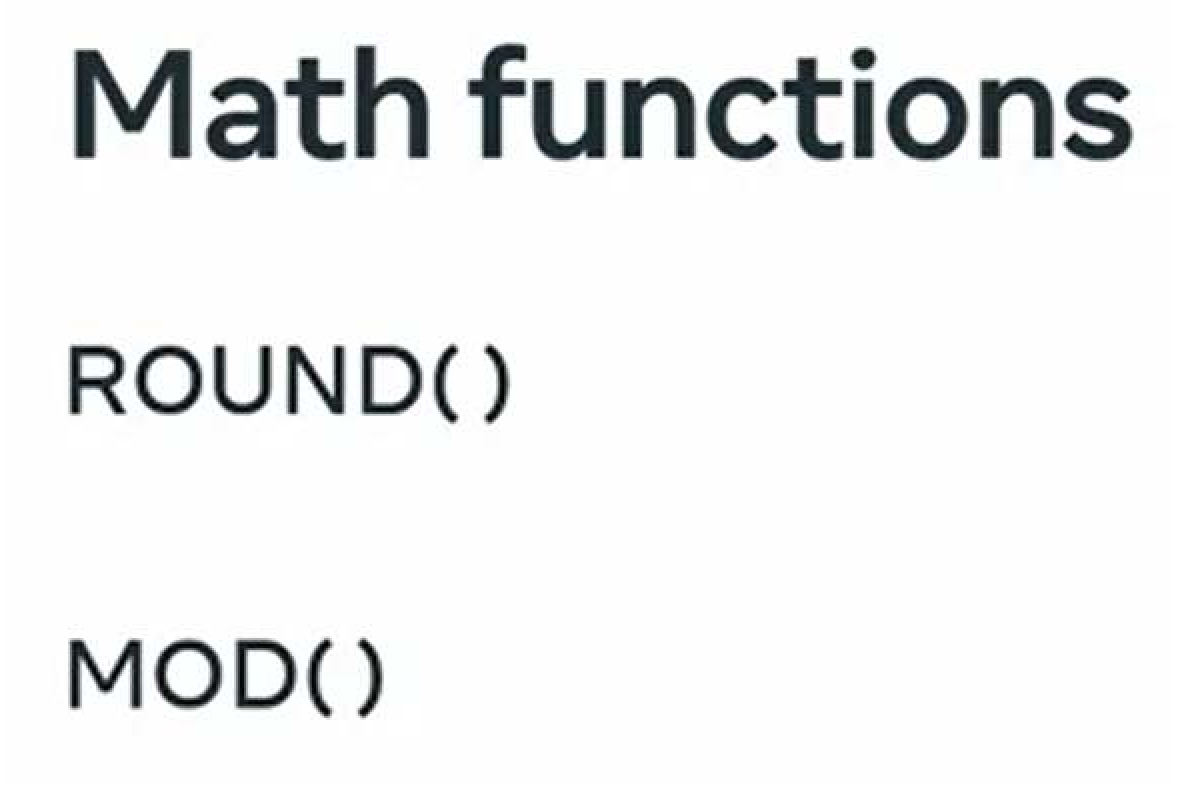
SUM( )

AVG( )

MAX( )

MIN( )

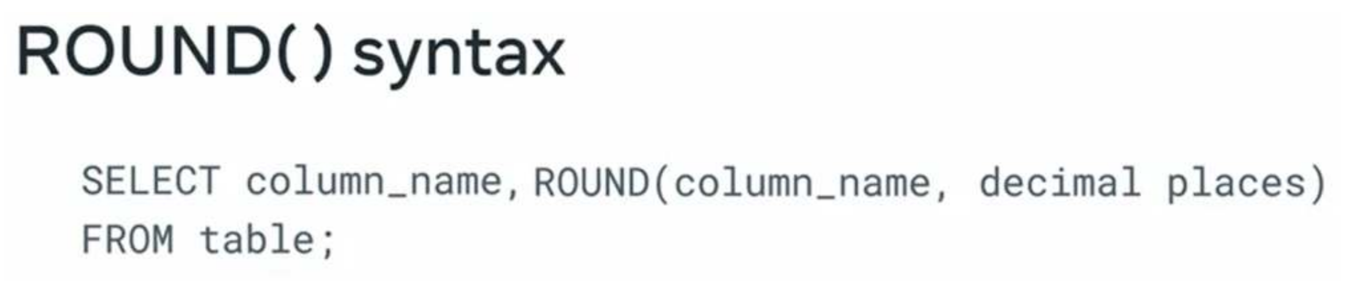
COUNT( )



Math functions

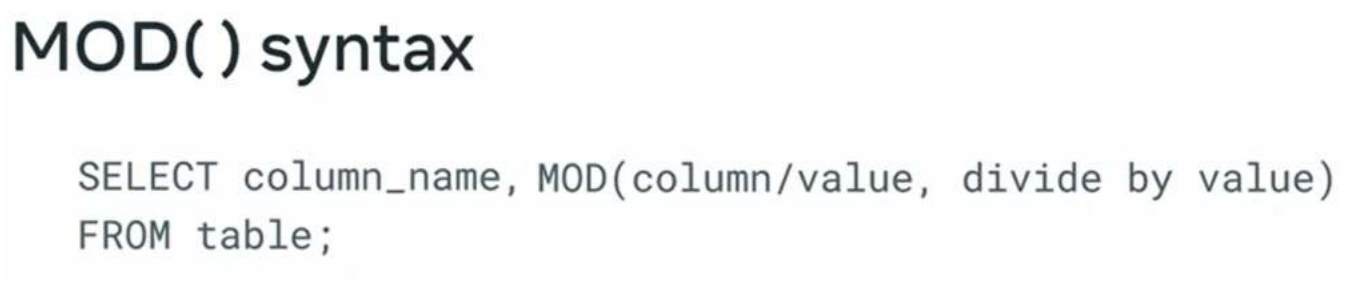
ROUND()

MOD()



ROUND() syntax

```
SELECT column_name, ROUND(column_name, decimal places)
FROM table;
```



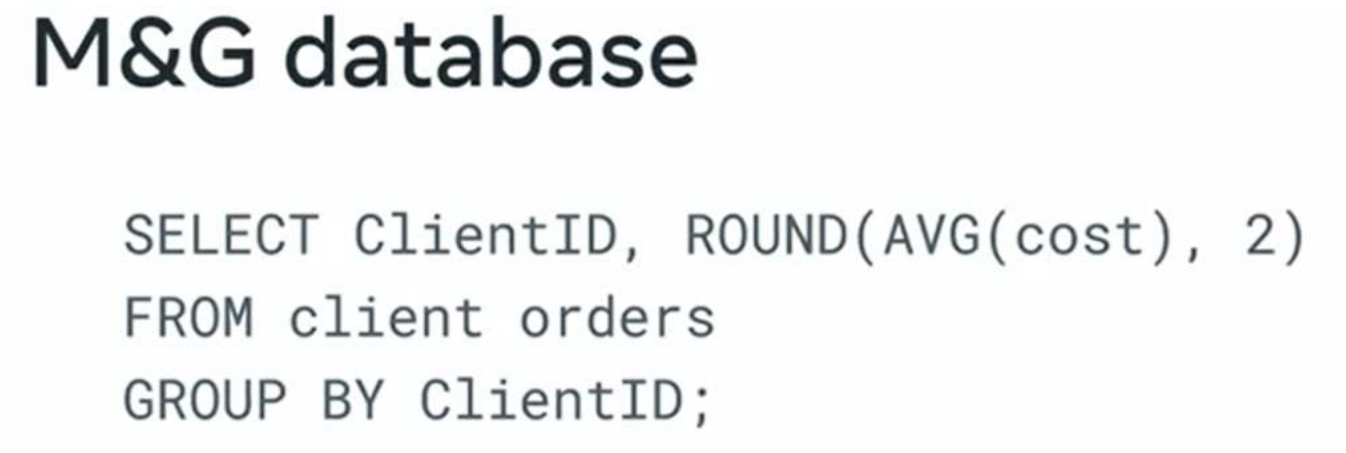
MOD() syntax

```
SELECT column_name, MOD(column/value, divide by value)
FROM table;
```

MOD(): Returns the remainder of one number divided by another

---

you can use both Math and Aggregate functuons(**M&G**)



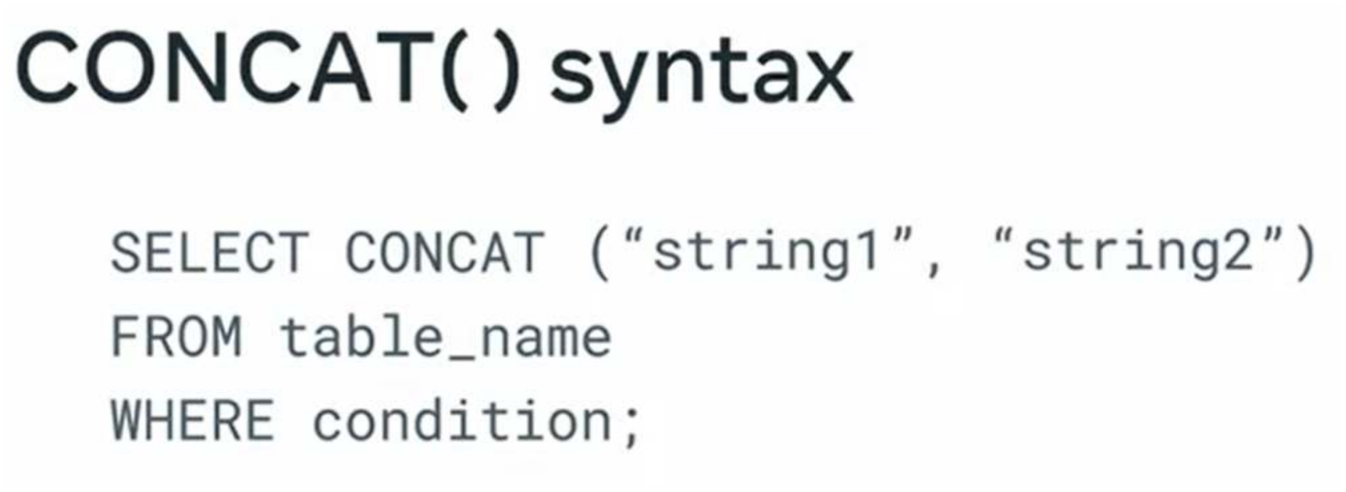
M&G database

```
SELECT ClientID, ROUND(AVG(cost), 2)
FROM client orders
GROUP BY ClientID;
```

$String functions$ - Perform different operations on string values in a database

String functions:

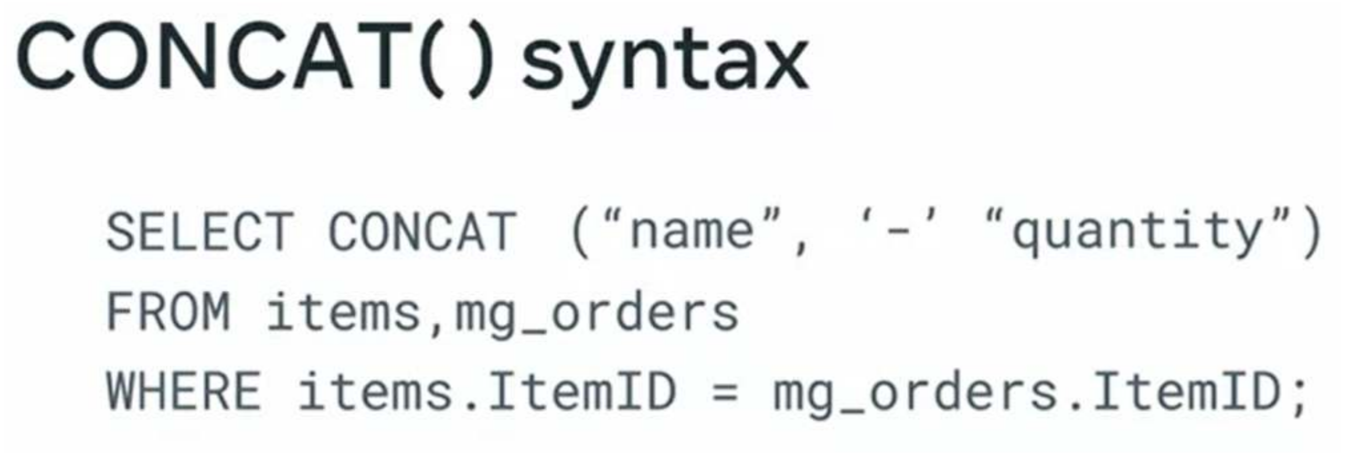
1. CONCAT() : Used to add several strings together
2. SUBSTR() : Extracts a segment of a string from a parent string
3. UCASE() : Convert a string to uppercase
4. LCASE() : Converts a string to lowercase



CONCAT() syntax

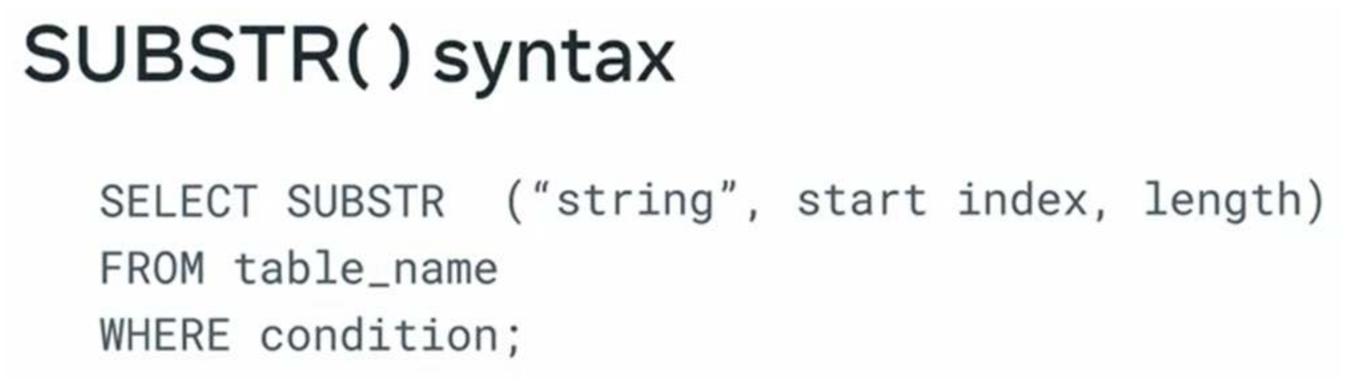
```
SELECT CONCAT (“string1”, “string2”)
FROM table_name
WHERE condition;
```

A more complex example of the concatenation function might involve extracting string values from two separate tables:



CONCAT() syntax

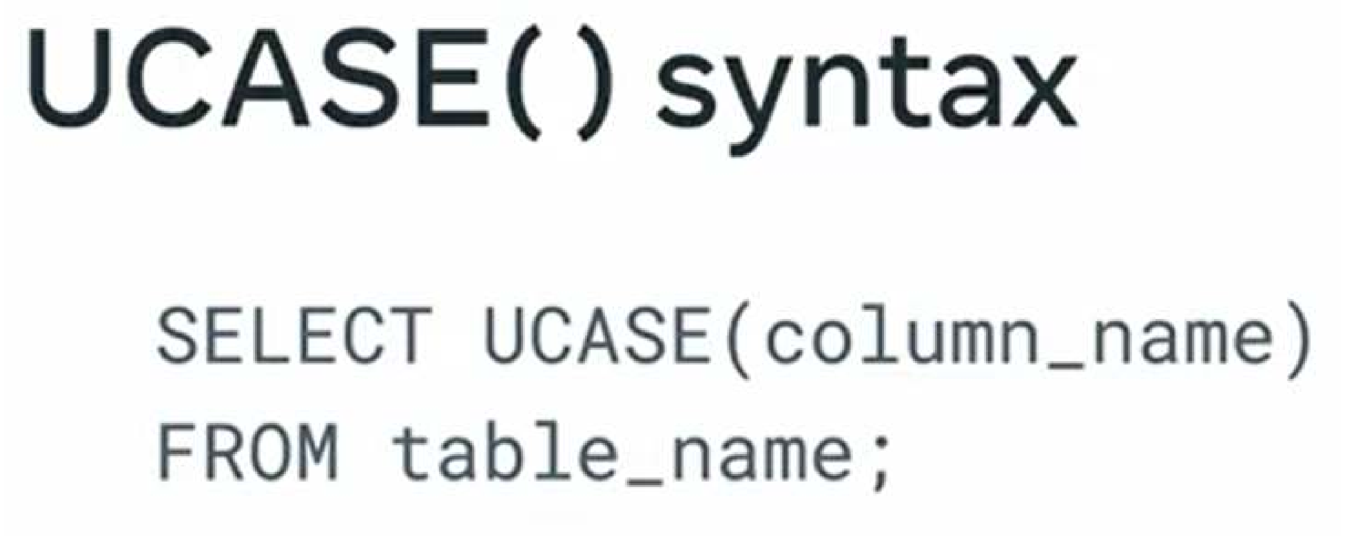
```
SELECT CONCAT (“name”, ‘-’ “quantity”)
FROM items,mg_orders
WHERE items.ItemID = mg_orders.ItemID;
```



SUBSTR() syntax

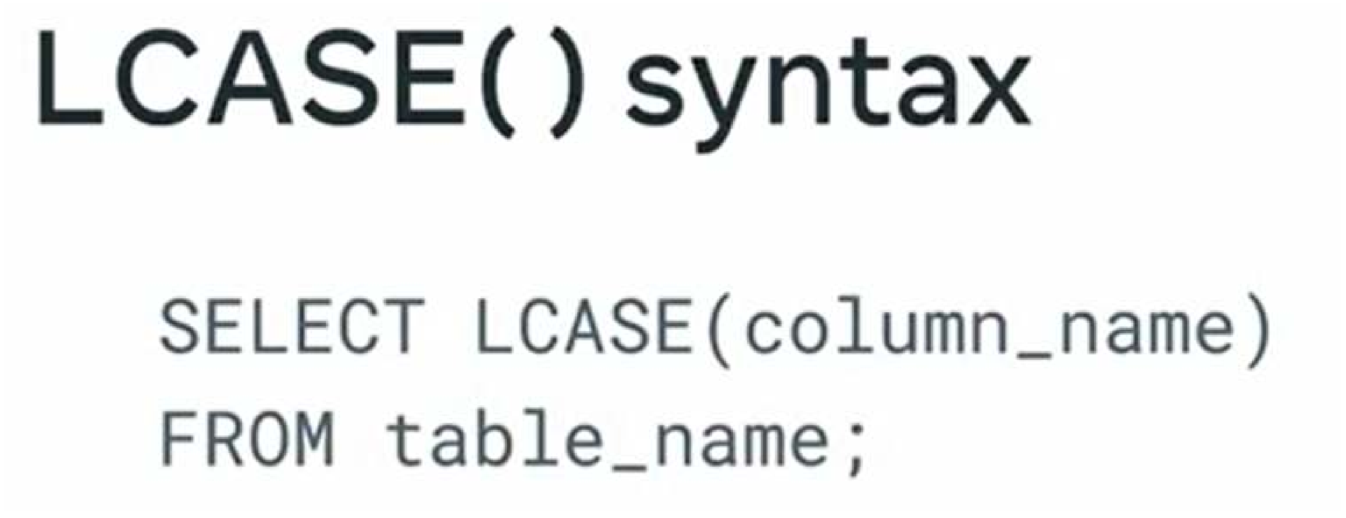
```
SELECT SUBSTR (“string”, start index, length)
FROM table_name
WHERE condition;
```

Loading [MathJax]/extensions/Safe.js



UCASE() syntax

```
SELECT UCASE(column_name)
FROM table_name;
```



LCASE() syntax

```
SELECT LCASE(column_name)
FROM table_name;
```

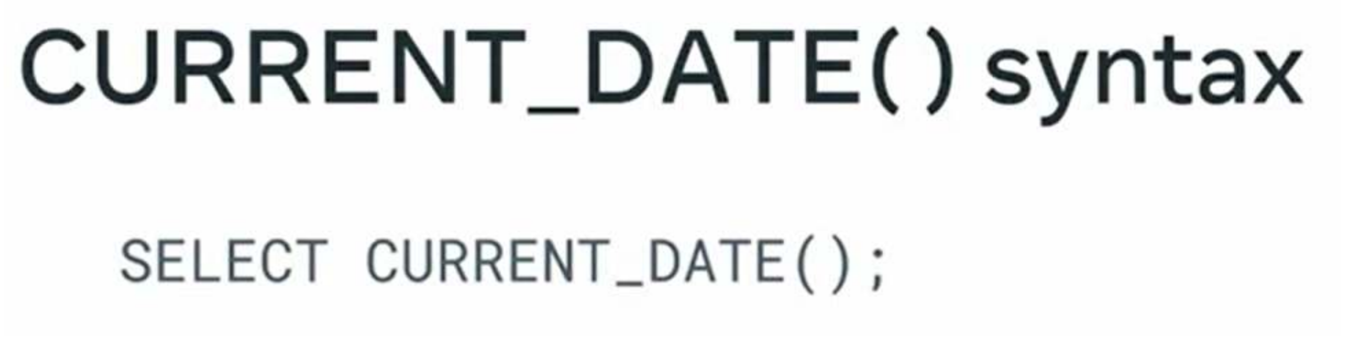
$Date functions$ - Extract outputs that contain information on time and date values

Date functions:

1. CURRENT_DATE() : Returns the date in year, month date format
2. CURRENT_TIME() : Returns the time in hours, minutes, seconds format
3. DATE_FORMAT() : Used to format a date according to a given format
4. DATEDIFF : Identifies the number of days between two date values

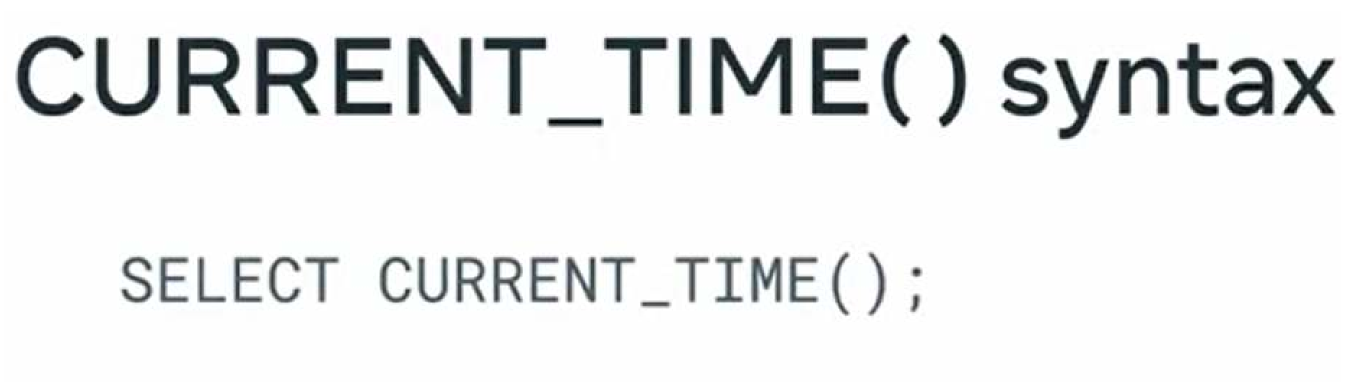
---

Loading [MathJax]/extensions/Safe.js



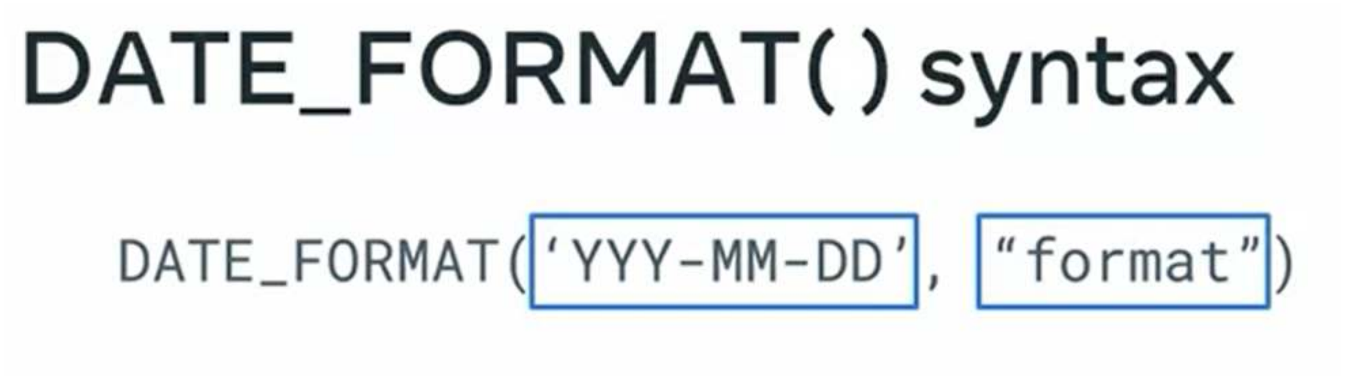
CURRENT_DATE() syntax

```
SELECT CURRENT_DATE();
```



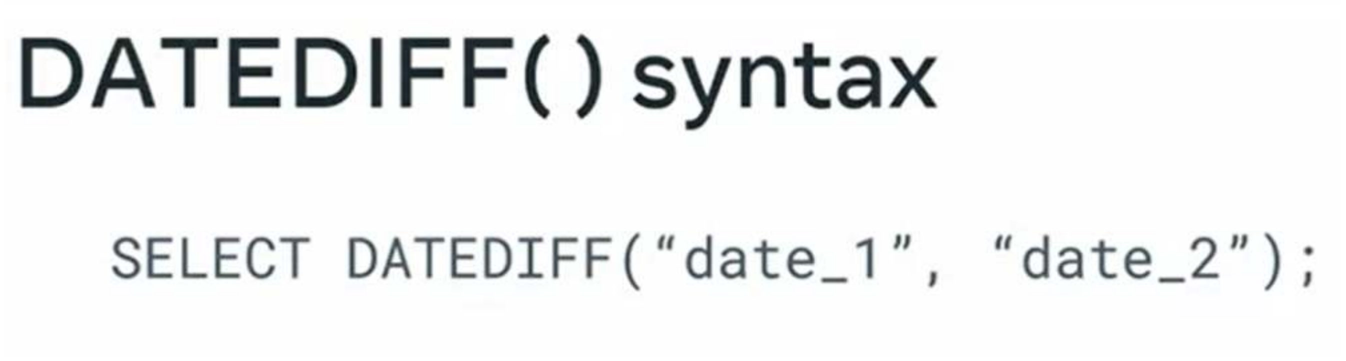
CURRENT_TIME() syntax

```
SELECT CURRENT_TIME();
```



DATE_FORMAT() syntax

```
DATE_FORMAT(‘YYY-MM-DD’, “format”)
```



DATEDIFF() syntax

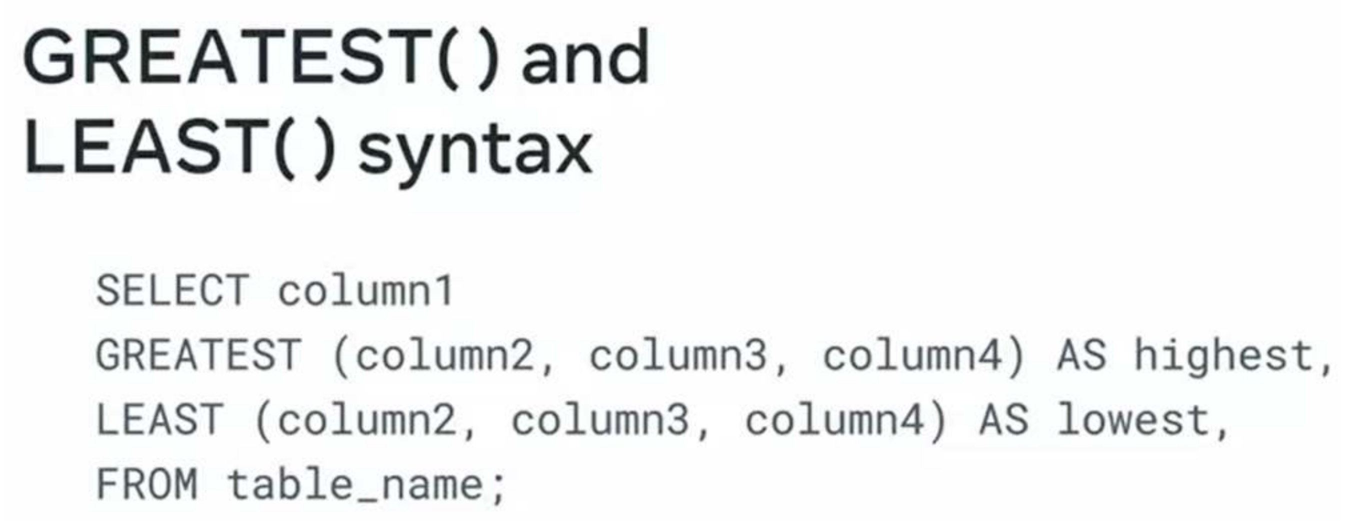
```
SELECT DATEDIFF(“date_1”, “date_2”);
```

$Comparison functions$ - Compare values within a database

Comparison functions:

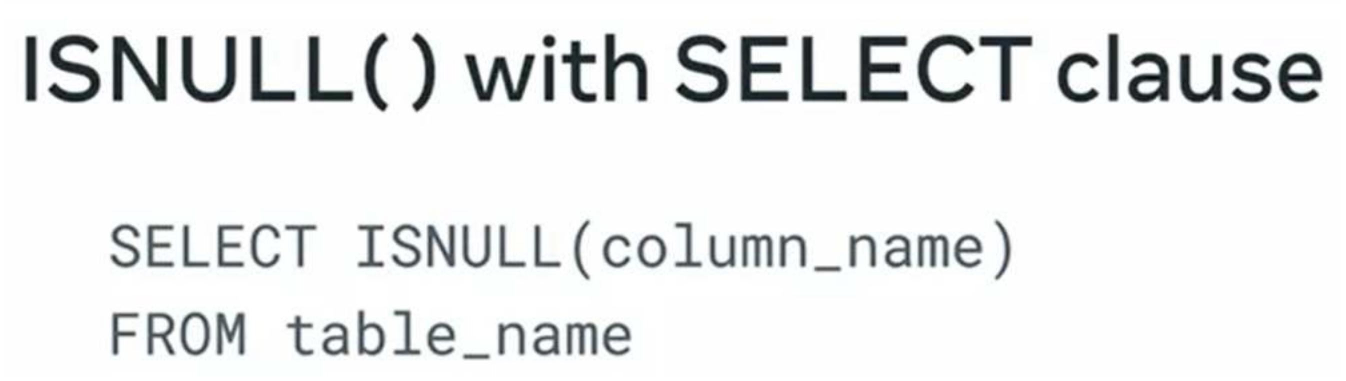
1. GREATEST() : Find the highest value
2. LEAST() : Determines the lowest values
3. ISNULL() : Used as an alternative to the equals operator to test if a value is null

---



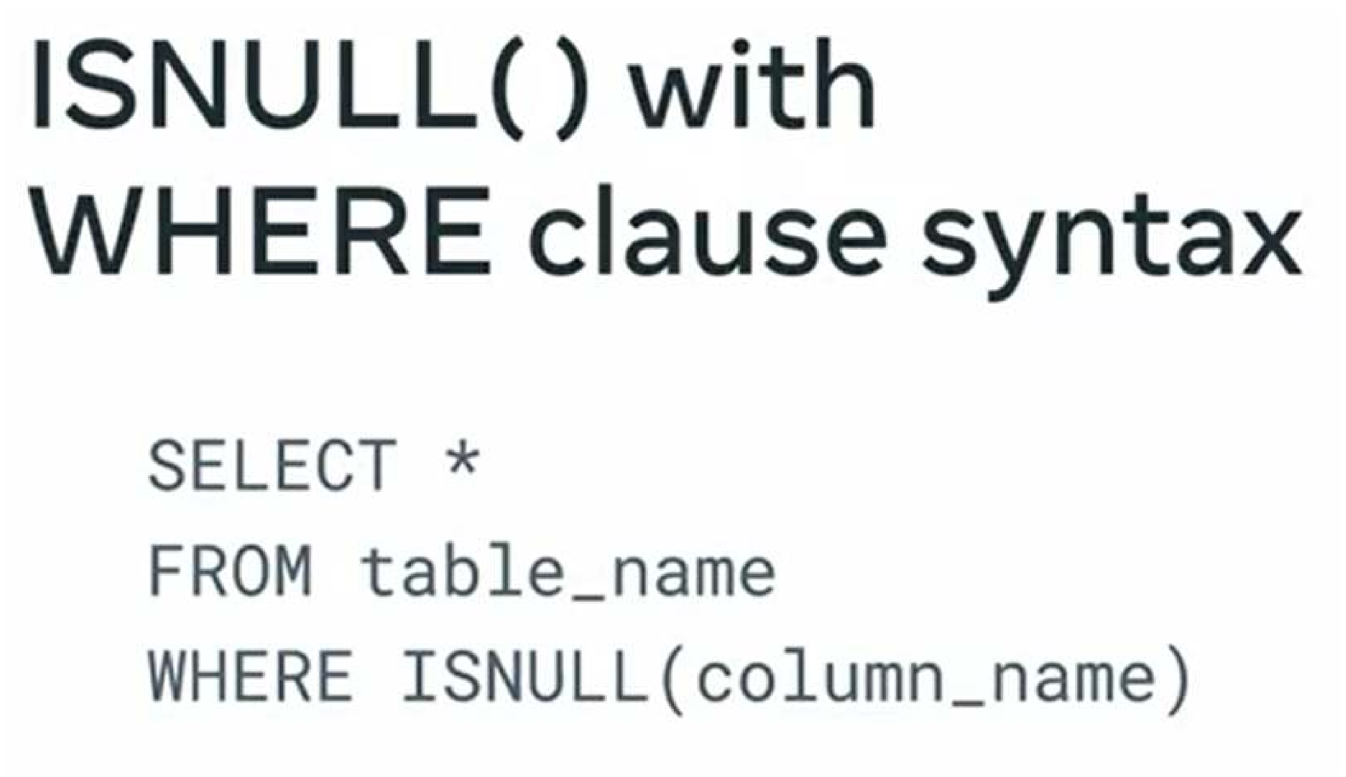
GREATEST() and LEAST() syntax

```
SELECT column1
GREATEST (column2, column3, column4) AS highest,
LEAST (column2, column3, column4) AS lowest,
FROM table_name;
```



ISNULL() with SELECT clause

```
SELECT ISNULL(column_name)
FROM table_name
```



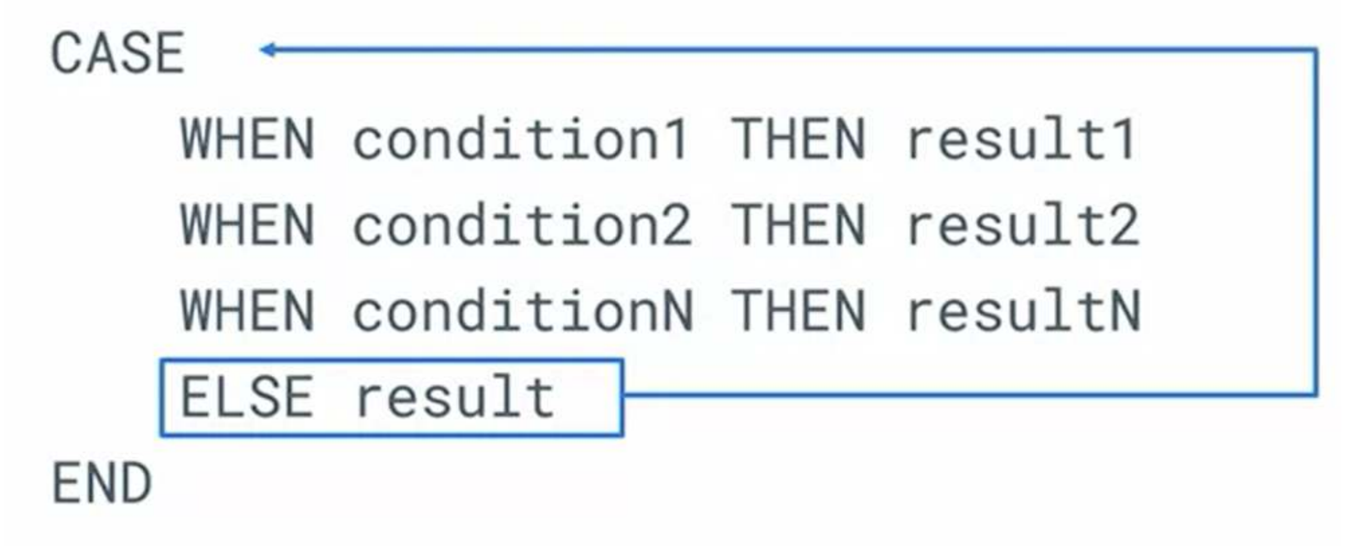
ISNULL() with WHERE clause syntax

```
SELECT *
FROM table_name
WHERE ISNULL(column_name)
```

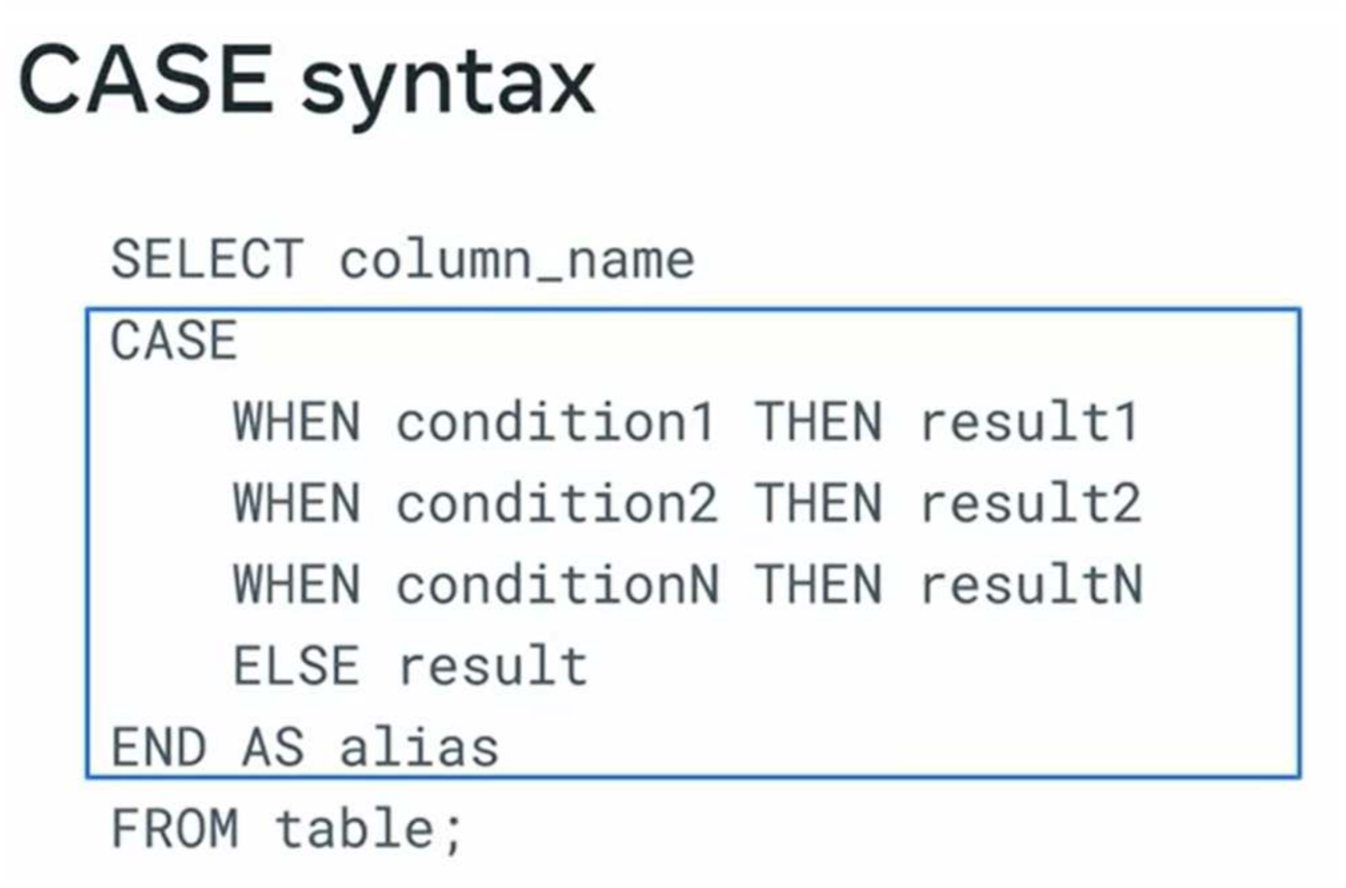
$Control flow functions$ - Evaluate conditions and determine the execution path of a query

The most common control flow function used in a MySQL is the CASE function

CASE function: The case function runs through a set of conditions contained within a case block and returns a value when the first condition is met.



```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    WHEN conditionN THEN resultN
    ELSE result
END
```



CASE syntax

```
SELECT column_name
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    WHEN conditionN THEN resultN
    ELSE result
END AS alias
FROM table;
```

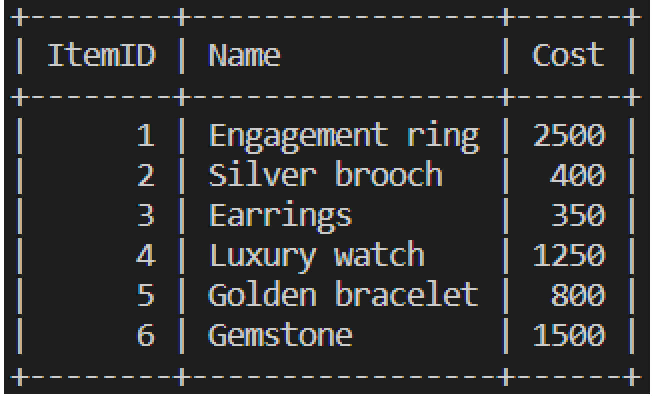
Exercise: Working with MySQL functions

Lab Instructions: Working with MySQL Functions

Mangata and Gallo (also known as M&G) is a jewelry store that specializes in special occasions like engagements, weddings and anniversaries. In this lab, you are going to complete a series of tasks to make it easier for M&G staff to format and filter data using MySQL string, Math, Date and Comparison functions for their reports.

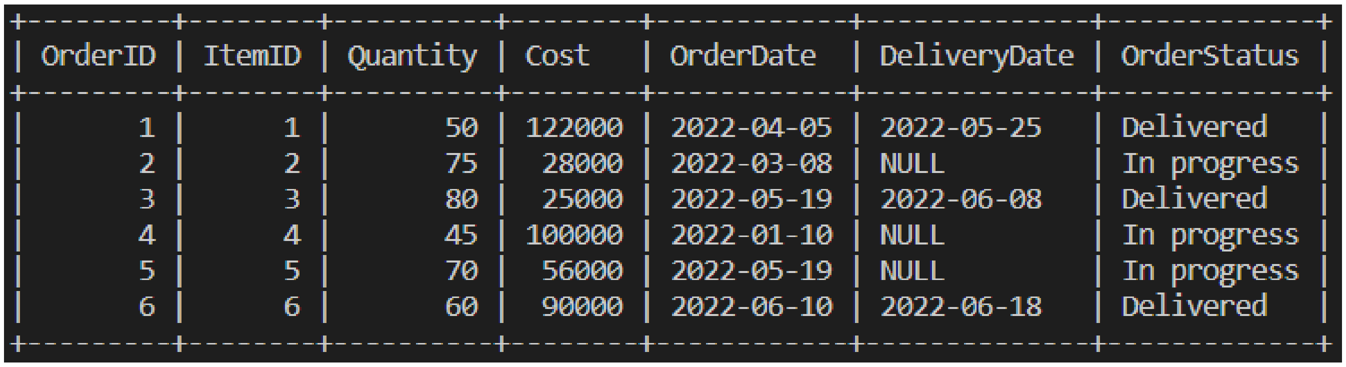
The data used in this lab comes from the following item and mg_orders tables:

Loading [MathJax]/extensions/Safe.js

item table 

```
+--------+-----------------+------+
| ItemID | Name            | Cost |
+--------+-----------------+------+
|      1 | Engagement ring | 2500 |
|      2 | Silver brooch   |  400 |
|      3 | Earrings        |  350 |
|      4 | Luxury watch    | 1250 |
|      5 | Golden bracelet |  800 |
|      6 | Gemstone        | 1500 |
+--------+-----------------+------+
```

mg_orders table



```
+---------+--------+----------+--------+------------+--------------+-------------+
| OrderID | ItemID | Quantity | Cost   | OrderDate  | DeliveryDate | OrderStatus |
+---------+--------+----------+--------+------------+--------------+-------------+
|       1 |      1 |       50 | 122000 | 2022-04-05 | 2022-05-25   | Delivered   |
|       2 |      2 |       75 |  28000 | 2022-03-08 | NULL         | In progress |
|       3 |      3 |       80 |  25000 | 2022-05-19 | 2022-06-08   | Delivered   |
|       4 |      4 |       45 | 100000 | 2022-01-10 | NULL         | In progress |
|       5 |      5 |       70 |  56000 | 2022-05-19 | NULL         | In progress |
|       6 |      6 |       60 |  90000 | 2022-06-10 | 2022-06-18   | Delivered   |
+---------+--------+----------+--------+------------+--------------+-------------+
```

Prerequisites

To complete this lab, you must have created the M&G jewelry store database in MySQL. This includes the item and mg_orders tables, which must be populated with relevant data.

The code required to create the database and the tables are listed below.

1: Create the database `CREATE DATABASE jewelrystore_db;`

2: Use the database `USE jewelrystore_db;`

3: Create the item table `CREATE TABLE item(ItemID INT, Name VARCHAR(150), Cost INT, PRIMARY KEY(ItemID));`

4: Insert data into the item table `INSERT INTO item VALUES(1,'Engagement ring',2500),(2,'Silver brooch',400),(3,'Earrings',350),(4,'Luxury watch',1250),(5,'Golden bracelet',800), (6,'Gemstone',1500);`

5: Create the mg_orders table `CREATE TABLE mg_orders(OrderID INT, ItemID INT, Quantity INT, Cost INT, OrderDate DATE, DeliveryDate DATE, OrderStatus VARCHAR(50), PRIMARY KEY(OrderID));`

6: Insert data into the mg_orders table `INSERT INTO mg_orders VALUES(1,1,50,122000,'2022-04-05','2022-05-25', 'Delivered'),(2,2,75,28000,'2022-03-08',NULL, 'In progress'), (3,3,80,25000,'2022-05-19','2022-06-08', 'Delivered'),(4,4,45,100000,'2022-01-10',NULL, 'In progress'),(5,5,70,56000,'2022-05-19',NULL, 'In progress'),(6,6,60,90000,'2022-06-10','2022-06-18', 'Delivered');`

The main objectives of this activity:

Loading [MathJax]/extensions/Safe.js

- Work with MySQL String, Math, Date and Comparison functions.

Exercise Instructions

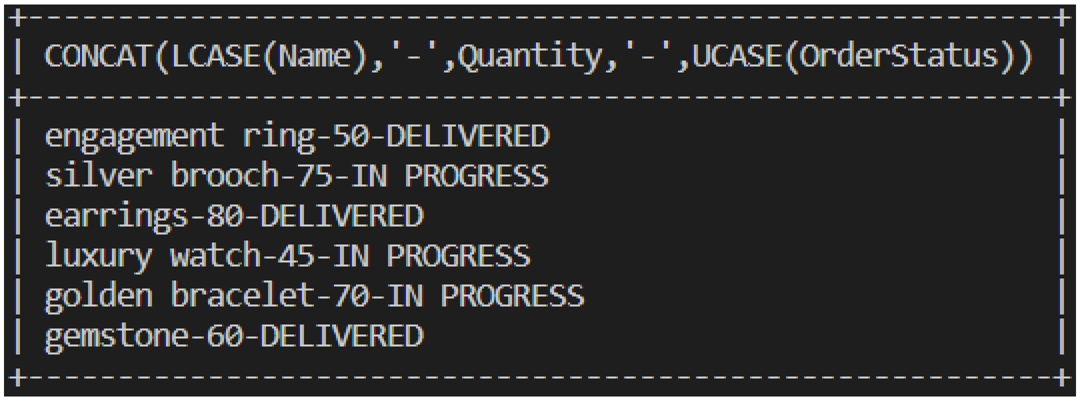
Please attempt the following tasks:

Task 1: Write a SQL SELECT query using appropriate MySQL string functions to list items, quantities and order status in the following format:

- Item name–quantity–order status

Item name should be in lower case. Order status should be in upper case.

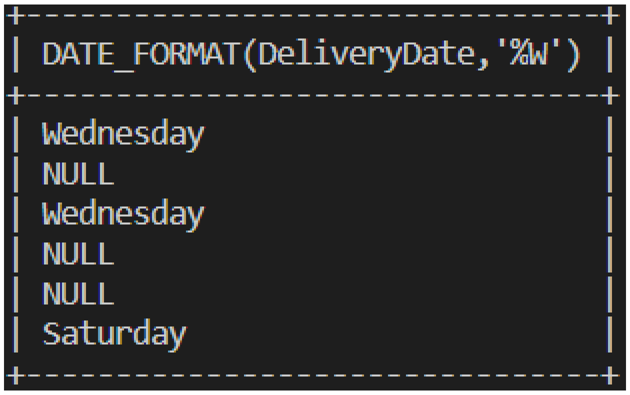
The expected output result should be similar to the following screenshot (if you have the same data set populated in the orders table).



```
+-----------------------------------------------------------+
| CONCAT(LCASE(Name),'-',Quantity,'-',UCASE(OrderStatus))  |
+-----------------------------------------------------------+
| engagement ring-50-DELIVERED                              |
| silver brooch-75-IN PROGRESS                              |
| earrings-80-DELIVERED                                     |
| luxury watch-45-IN PROGRESS                               |
| golden bracelet-70-IN PROGRESS                            |
| gemstone-60-DELIVERED                                     |
+-----------------------------------------------------------+
```

Task 2: Write a SQL SELECT query using an appropriate date function and a format string to find the name of the weekday on which M&G's orders are to be delivered.

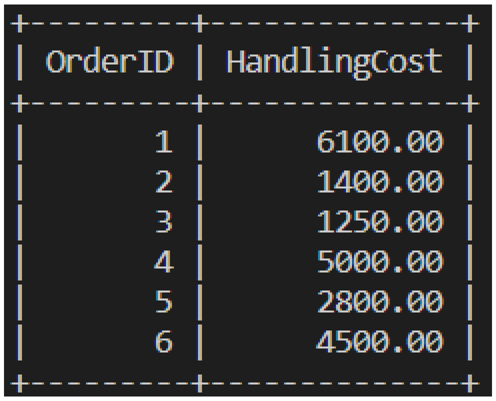
The expected output result should be like the following screenshot (if you have same data set populated in the orders table).



```
+--------------------------------+
| DATE_FORMAT(DeliveryDate,'%W') |
+--------------------------------+
| Wednesday                      |
| NULL                           |
| Wednesday                      |
| NULL                           |
| NULL                           |
| Saturday                       |
+--------------------------------+
```

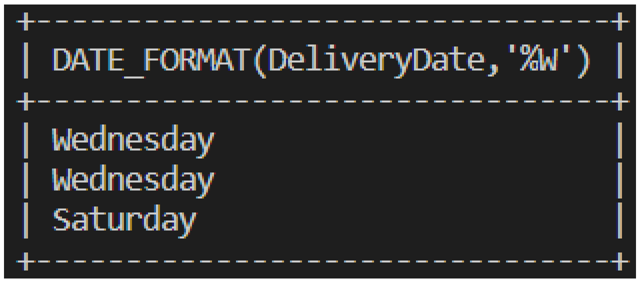
Task 3: Write a SQL SELECT query that calculates the cost of handling each order. This should be 5% of the total order cost. Use an appropriate math function to round that value to 2 decimal places.

Loading [MathJax]/extensions/Safe.js

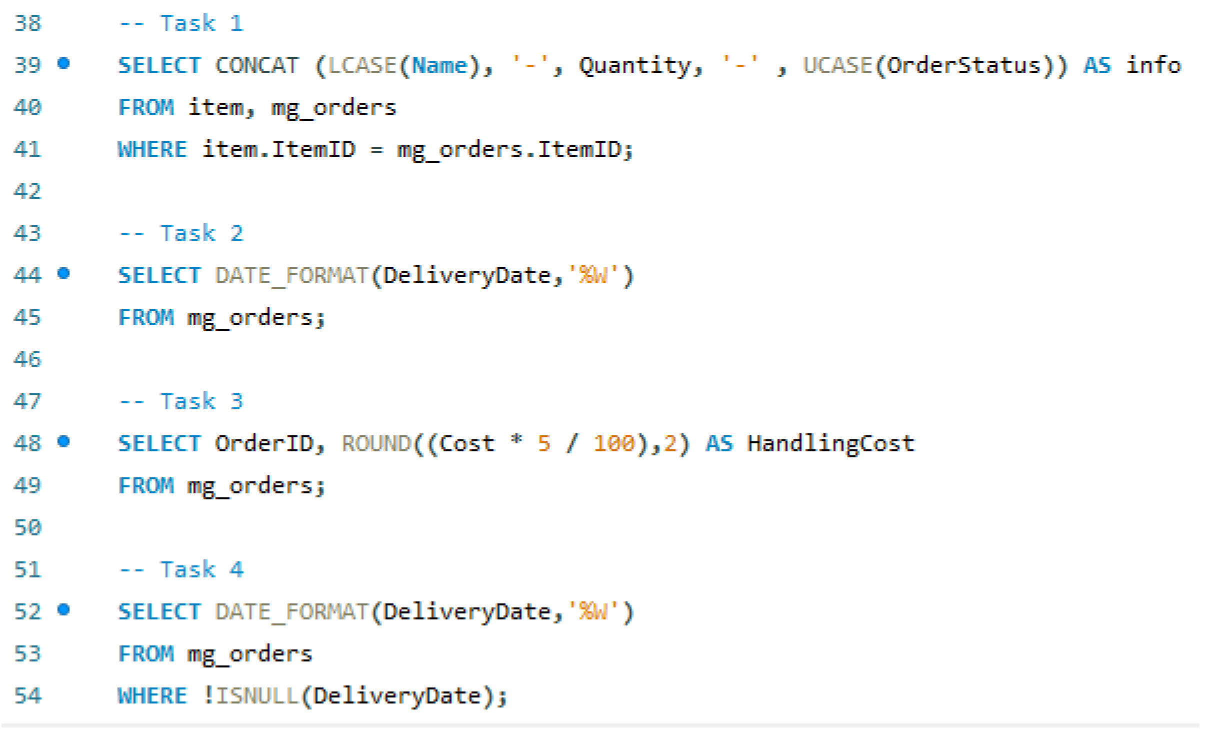
The expected output result should be like the following screenshot (if you have same data set populated in the orders table). 

```
+---------+--------------+
| OrderID | HandlingCost |
+---------+--------------+
|       1 |      6100.00 |
|       2 |      1400.00 |
|       3 |      1250.00 |
|       4 |      5000.00 |
|       5 |      2800.00 |
|       6 |      4500.00 |
+---------+--------------+
```

Task 4: Review the query that you wrote in the second task. Use an appropriate comparison function to filter out the records that do not have a NULL value in the delivery date column.

The expected output result should be like the following screenshot (if you have same data set populated in the orders table). 

```
+-------------------------------+
| DATE_FORMAT(DeliveryDate,'%W') |
+-------------------------------+
| Wednesday                     |
| Wednesday                     |
| Saturday                      |
+-------------------------------+
```

ANSWER: 

```
-- Task 1
SELECT CONCAT (LCASE(Name), '-', Quantity, '-' , UCASE(OrderStatus)) AS info
FROM item, mg_orders
WHERE item.ItemID = mg_orders.ItemID;

-- Task 2
SELECT DATE_FORMAT(DeliveryDate,'%W')
FROM mg_orders;

-- Task 3
SELECT OrderID, ROUND((Cost * 5 / 100),2) AS HandlingCost
FROM mg_orders;

-- Task 4
SELECT DATE_FORMAT(DeliveryDate,'%W')
FROM mg_orders
WHERE !ISNULL(DeliveryDate);
```

Reference sheet: Functions in MySQL

What is a function in MySQL?

A function is a code that performs an operation and returns a result. Functions are used to manipulate data in a database table.

MySQL has many built-in functions that fall under different categories. These include:

Loading [MathJax]/extensions/Safe.js

- String manipulation functions
- Date and time functions
- Numeric functions
- Comparison functions
- And control flow functions

At this stage in the lesson, you should be familiar with basic examples of each of these types of functions. So let's take a few moments to explore other examples.

Numeric functions

MySQL numeric functions can be broadly divided into mathematical and aggregate functions. You've reviewed numerous examples of many MySQL aggregate functions. You've also looked at many mathematical functions. In this reading, you'll explore a few more mathematical functions.

Mathematical functions allow you to perform mathematical tasks on numeric data.

Let's look at two meaningful and useful mathematical functions: CEIL and FLOOR.

The CEIL function returns the smallest integer value, which is not less than the passed value. For example, passing a value of 1.23 to the CEIL function returns a value of 2.

It returns NULL if the value passed is NULL. The syntax of the CEIL function is as follows:

```
SELECT CEIL(VALUE);
```

Example: The following code returns 16 as it is the smallest integer value that is greater than or equal to 15.50.

```
SELECT CEIL(15.50);
```

The FLOOR function does the opposite of CEIL. It returns the largest integer value not greater than the passed value. It returns NULL if the passed value is NULL.

Its syntax is the same as the CEIL function. It's just a matter of replacing the CEIL function with the FLOOR function.

```
SELECT FLOOR(VALUE);
```

Example: The following code returns 15 as it is the largest integer value that is less than or equal to 15.50.

```
SELECT FLOOR(15.50);
```

String functions

Now let's look at some more String functions like FORMAT, LENGTH and REPLACE.

The FORMAT function formats the number passed into a format like '#,###,###.##', rounded to the specified number of decimal places. It returns the result as a string.

Here's the syntax:

```
SELECT FORMAT(number_to_be_formatted, number_of_decimal_places);
```

Example: The following code returns 3,750.75 as it formats the number (3750.753, 2) as "#,###.##" and rounds it within two decimal places.

```
FORMAT(3750.753, 2)
```

If the number of decimal places is 0, the result has no decimal point or fractional part. If the number to be formatted or the number of decimal places is NULL, then the function returns NULL.

Date functions

You've explored MySQL date functions such as CURRENT_DATE, CURRENT_TIME, DATE_FORMAT and DATEDIFF. Now let's explore a few more essential date functions with some examples.

The ADDDATE function is used to perform arithmetic with dates. It comes in two forms.

1. ADDDATE(date, INTERVAL expr unit)
2. ADDDATE(date, days)

The first argument specifies the starting date or datetime value in the first form. The second argument is an expression that determines the interval value to be added to the starting date.

It has three parts:

1. INTERVAL is a keyword
2. expr represents a quantity
3. and unit represents the unit for interpreting the quantity; such as HOUR, DAY, or WEEK

The syntax for the first form is as follows:

```
SELECT ADDDATE(date, INTERVAL expr unit);
```

In the second form, the first argument is the same and the second argument is the integer, or number of days, to be added to the given date in the first argument.

Example: the following code returns 2020-05-15 because it adds 5 days to the specified date.

```
SELECT ADDDATE("2020-05-10", INTERVAL 5 DAY);
```

The syntax for the second form is as follows:

```
SELECT ADDDATE(date, days);
```

Example: The following code returns 2020-06-25 because it adds 5 days to the specified date.

```
SELECT ADDDATE("2020-06-15", 10);
```

The QUARTER function is also a very versatile function that returns the quarter of the year for the given date. The return value is in the range 1 to 4, or NULL if date is NULL.

The syntax is as follows:

```
SELECT QUARTER(date_value);
```

The DATE, MONTH, YEAR and DAY functions extract the date, month, year and 'day of the month' parts respectively, of a given date or date time expression.

Example: The following code returns a value of 3. The QUARTER() function returns the quarter of the year for the given date value.

```
SELECT QUARTER("2020-09-15");
```

Comparison functions

You may know of a few comparison functions of MySQL like GREATEST, LEAST and ISNULL.

COALESCE is another comparison function that takes several arguments and returns the first non-NULL argument. In case all arguments are NULL, the COALESCE function returns NULL. You can think of this function as a NULL checking function.

The syntax for this function is as follows:

```
SELECT COALESCE (value1, value2);
```

This function is particularly useful when you replace a column value with a NULL value with some other value.

Example: The following code returns Coursera, because the COALESCE() function returns the first non-null value in a list.

```
SELECT COALESCE(NULL, 'Coursera', NULL, 'Database');
```

Control flow functions

MySQL also has a function that lets you evaluate conditions and decide how the query should be executed accordingly. You should be familiar with the CASE function in MySQL and IFNULL, another function. It accepts two arguments and returns the first argument if it is not NULL. Otherwise, the IFNULL function returns the second argument. The two arguments can be literal values or expressions.

The syntax is as follows:

```
SELECT IFNULL(evaluated_expression, alternative_value);
```

Example: The following code returns Coursera as the value because the evaluated expression is NULL.

```
SELECT IFNULL(NULL, "Coursera");
```

There’s also the NULLIF function that takes in two values or expressions. If they're equal then it returns NULL. Otherwise, it returns the first value or expression.

This is an example of the syntax:

```
SELECT NULLIF(expression1, expression2);
```

Example: The following NULLIF() function returns 10, as it compares the two expressions (10 and 15) and returns NULL if they are equal. Otherwise, the first expression is returned.

Loading [MathJax]/extensions/Safe.js

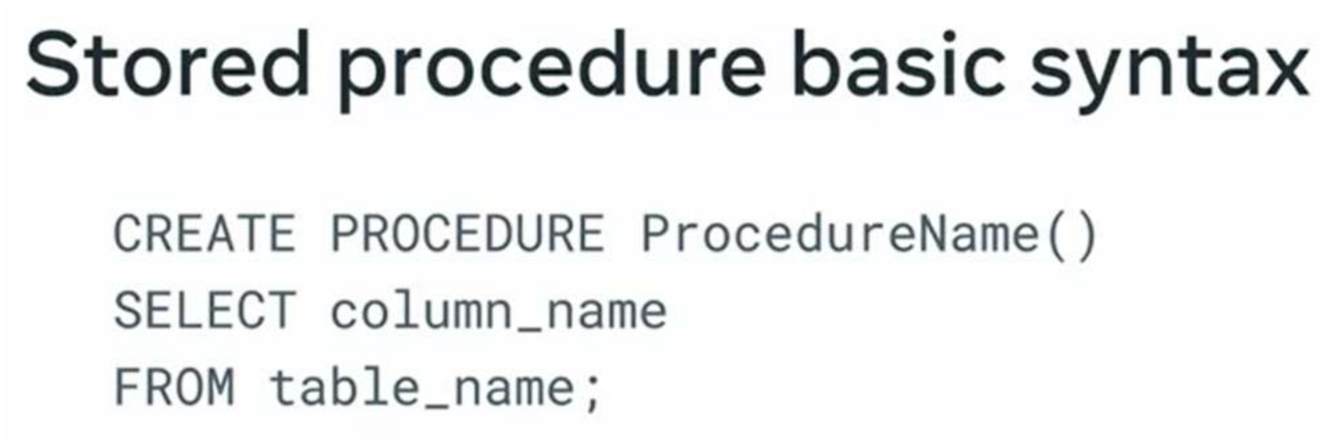
```
SELECT NULLIF(10, 15);
```

Procedures

Stored procedures: Reusable block of code invoked on command

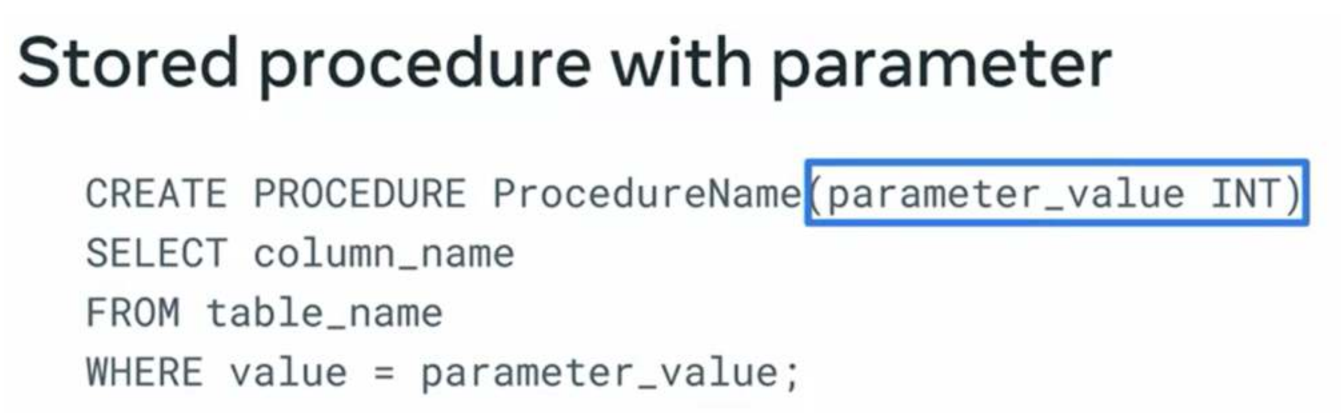
Benefits of stored procedures:

1. Consistency
2. Reusability
3. Efficiency



Stored procedure basic syntax

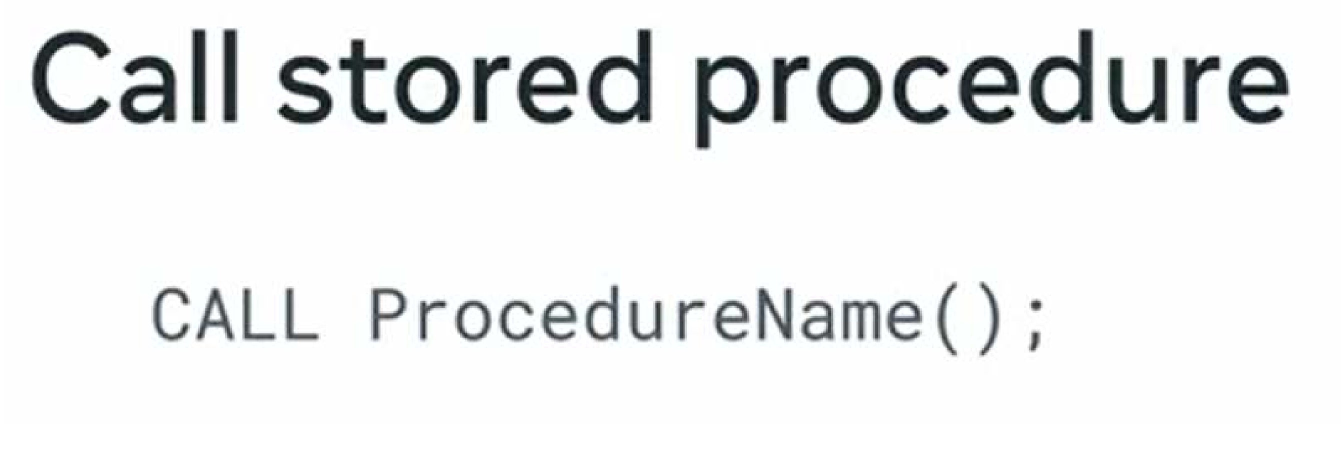
```
CREATE PROCEDURE ProcedureName()
SELECT column_name
FROM table_name;
```



Stored procedure with parameter

```
CREATE PROCEDURE ProcedureName(parameter_value INT)
SELECT column_name
FROM table_name
WHERE value = parameter_value;
```

Once you've created the stored procedure, the next step is to invoke it by using the call command.

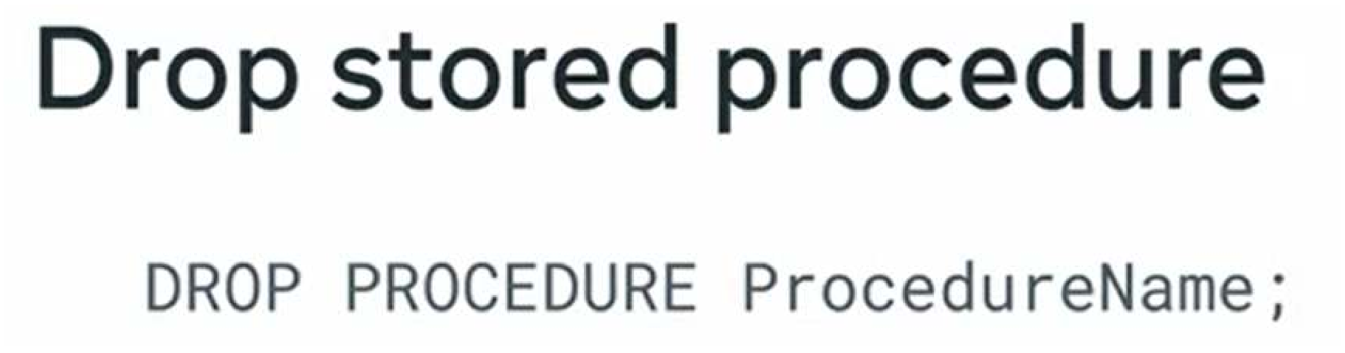


Call stored procedure

```
CALL ProcedureName();
```

Loading [MathJax]/extensions/Safe.js

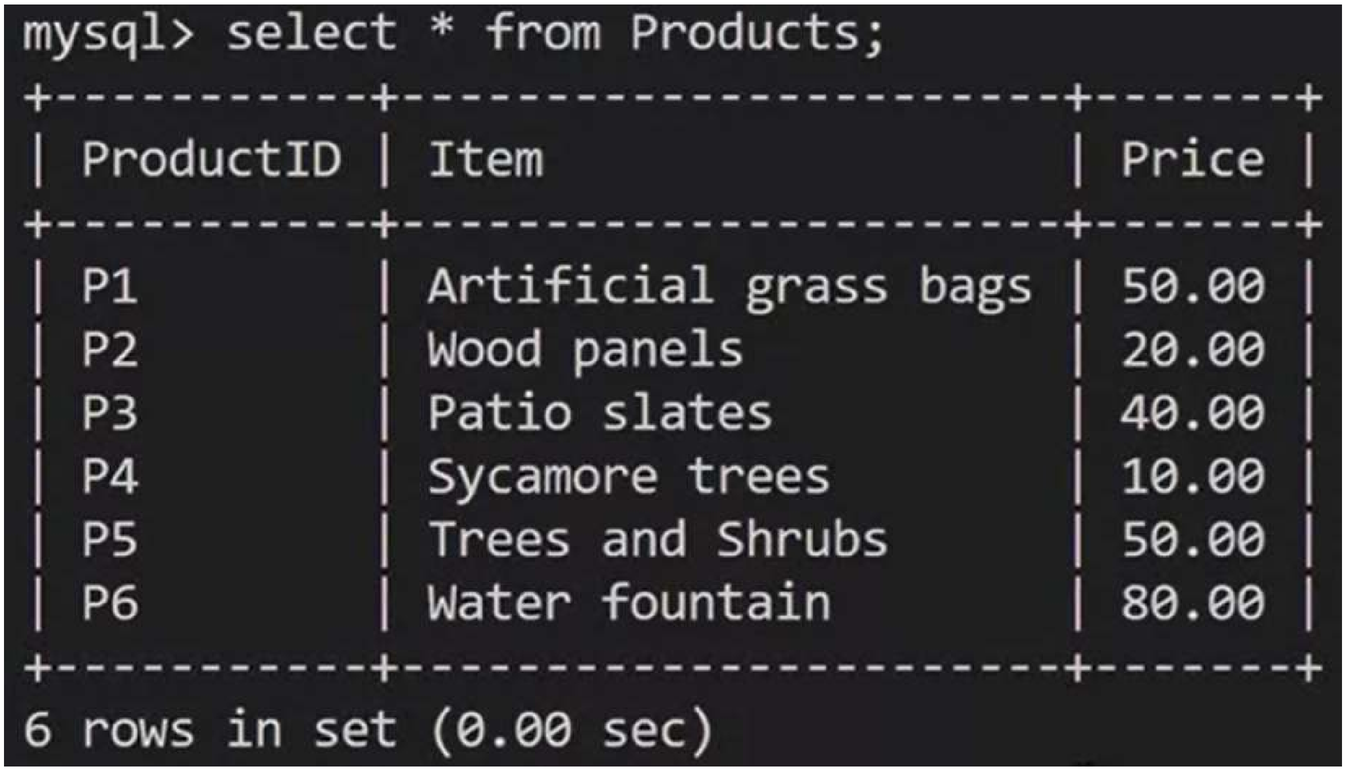
To delete a stored procedure:



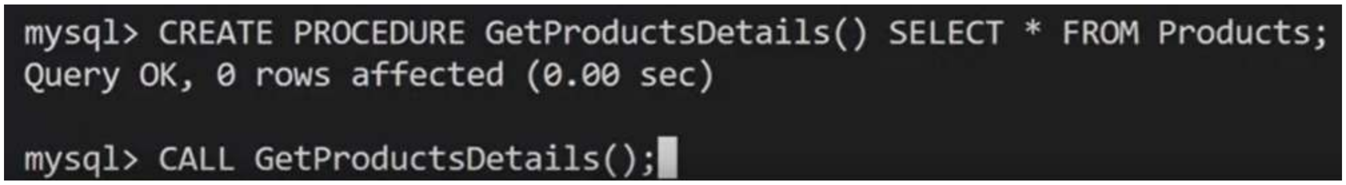
Drop stored procedure

```
DROP PROCEDURE ProcedureName;
```

EXAMPLE:

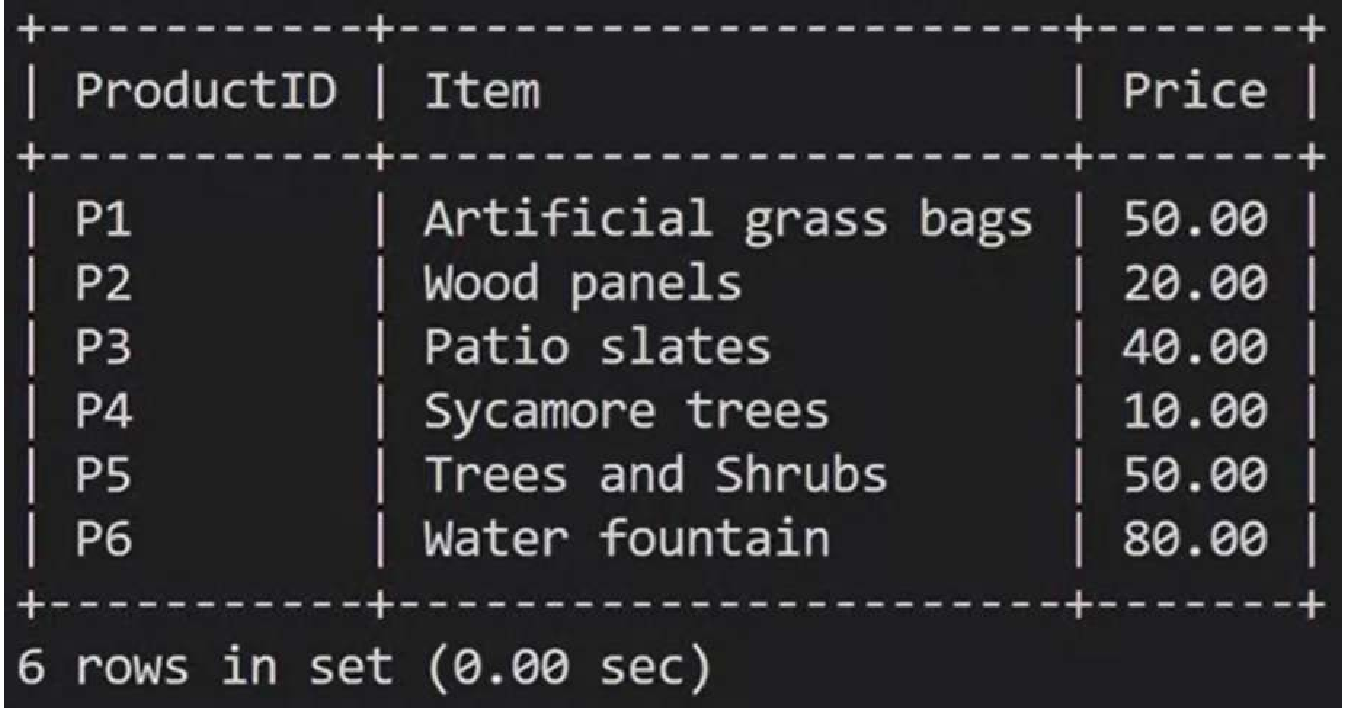


```
mysql> select * from Products;
+-----------+------------------------+-------+
| ProductID | Item                   | Price |
+-----------+------------------------+-------+
| P1        | Artificial grass bags  | 50.00 |
| P2        | Wood panels            | 20.00 |
| P3        | Patio slates           | 40.00 |
| P4        | Sycamore trees         | 10.00 |
| P5        | Trees and Shrubs       | 50.00 |
| P6        | Water fountain         | 80.00 |
+-----------+------------------------+-------+
6 rows in set (0.00 sec)
```

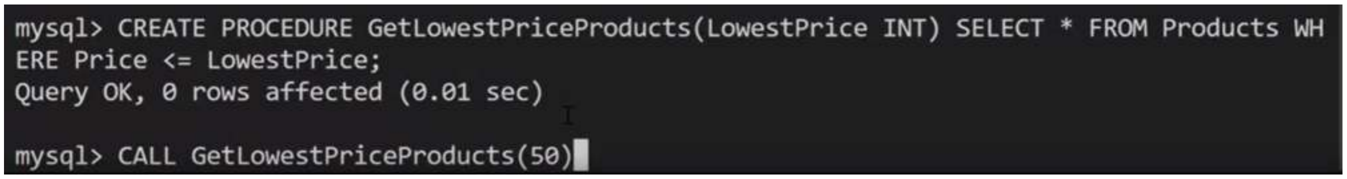


```
mysql> CREATE PROCEDURE GetProductsDetails() SELECT * FROM Products;
Query OK, 0 rows affected (0.00 sec)

mysql> CALL GetProductsDetails();
```

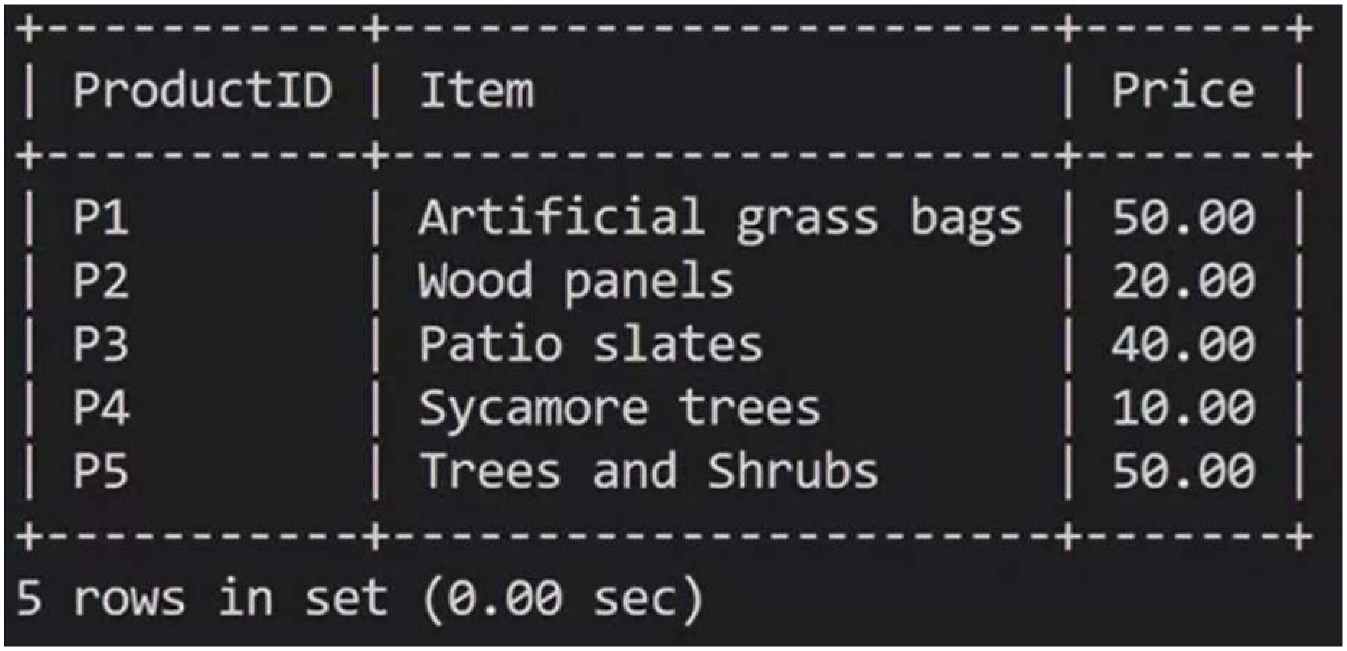
Loading [MathJax]/extensions/Safe.js



```
+-----------+------------------------+-------+
| ProductID | Item                   | Price |
+-----------+------------------------+-------+
| P1        | Artificial grass bags  | 50.00 |
| P2        | Wood panels            | 20.00 |
| P3        | Patio slates           | 40.00 |
| P4        | Sycamore trees         | 10.00 |
| P5        | Trees and Shrubs       | 50.00 |
| P6        | Water fountain         | 80.00 |
+-----------+------------------------+-------+
6 rows in set (0.00 sec)
```



```
mysql> CREATE PROCEDURE GetLowestPriceProducts(LowestPrice INT) SELECT * FROM Products WH
ERE Price <= LowestPrice;
Query OK, 0 rows affected (0.01 sec)

mysql> CALL GetLowestPriceProducts(50)
```



```
+-----------+------------------------+-------+
| ProductID | Item                   | Price |
+-----------+------------------------+-------+
| P1        | Artificial grass bags  | 50.00 |
| P2        | Wood panels            | 20.00 |
| P3        | Patio slates           | 40.00 |
| P4        | Sycamore trees         | 10.00 |
| P5        | Trees and Shrubs       | 50.00 |
+-----------+------------------------+-------+
5 rows in set (0.00 sec)
```

In [ ]:

Loading [MathJax]/extensions/Safe.js