COMPLETE INTERVIEW PREPARATION

TOPIC 1: REUNITE PROJECT

TOPIC 2: FPL PROJECT

TOPIC 3: RESTAURANT PROJECT

TOPIC 4: ADVANCED JAVA

TOPIC 5: ADVANCED SPRING FRAMEWORK

TOPIC 6: COLLECTIONS FRAMEWORK

TOPIC 7: ADVANCED MULTITHREADING

TOPIC 8: HIBERNATE FRAMEWORK

TOPIC 9: GIT COMMANDS

TOPIC 10: LINUX COMMANDS

TOPIC 11: ADVANCED MYSQL

What is MySQL?

MySQL is a relational database management system
MySQL is open-source
MySQL is free
MySQL is ideal for both small and large applications
MySQL is very fast, reliable, scalable, and easy to use
MySQL is cross-platform
MySQL is compliant with the ANSI SQL standard
MySQL was first released in 1995
MySQL is developed, distributed, and supported by Oracle Corporation
MySQL is named after co-founder Monty Widenius's daughter: My

Some of The Most Important SQL Commands

SELECT - extracts data from a database
UPDATE - updates data in a database
DELETE - deletes data from a database
INSERT INTO - inserts new data into a database
CREATE DATABASE - creates a new database
ALTER DATABASE - modifies a database
CREATE TABLE - creates a new table
ALTER TABLE - modifies a table
DROP TABLE - deletes a table
CREATE INDEX - creates an index (search key)
DROP INDEX - deletes an index

The MySQL SELECT Statement

The SELECT statement is used to select data from a database.
The data returned is stored in a result table, called the result-set.

SELECT Syntax

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

Here, column1, column2, ... are the field names of the table you want to select data from. If you want to select all the fields available in the table, use the following syntax:

```
SELECT * FROM table_name;
```

Demo Database

Below is a selection from the "Customers" table in the Northwind sample database:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

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

The MySQL SELECT DISTINCT Statement

The SELECT DISTINCT statement is used to return only distinct (different) values.

Inside a table, a column often contains many duplicate values; and sometimes you only want to list the different (distinct) values.

SELECT DISTINCT Syntax

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

The following SQL statement counts and returns the number of different (distinct) countries in the "Customers" table:

```
SELECT COUNT(DISTINCT Country) FROM Customers;
```

The MySQL WHERE Clause

The WHERE clause is used to filter records.

It is used to extract only those records that fulfill a specified condition.

WHERE Syntax

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

WHERE Clause Example
The following SQL statement selects all the customers from "Mexico":

```
SELECT * FROM Customers
WHERE Country = 'Mexico';
```

Text Fields vs. Numeric Fields

SQL requires single quotes around text values (most database systems will also allow double quotes).

However, numeric fields should not be enclosed in quotes:

```
SELECT * FROM Customers
WHERE CustomerID = 1;
```

Operators in The WHERE Clause

The following operators can be used in the WHERE clause:

Operator	Description
=	Equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
<>	Not equal. Note: In some versions it is written as !=
BETWEEN	Between a certain range
LIKE	Search for a pattern
IN	To specify multiple possible values for a column

The MySQL AND, OR and NOT Operators

The WHERE clause can be combined with AND, OR, and NOT operators.

AND, OR operators filter records based on more than one condition:
AND operator displays a record if all conditions separated by AND are TRUE.
OR operator displays a record if any conditions separated by OR is TRUE.
NOT operator displays a record if the condition(s) is NOT TRUE.

AND Syntax

```
SELECT column1, column2, ...
FROM table_name
WHERE condition1 AND condition2 AND condition3 ...;
```

OR Syntax

```
SELECT column1, column2, ...
FROM table_name
WHERE condition1 OR condition2 OR condition3 ...;
```

NOT Syntax

```
SELECT column1, column2, ...
FROM table_name
WHERE NOT condition;
```

Demo Database

Below is a selection from the "Customers" table in the Northwind sample database:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

AND Example

The following SQL statement selects all fields from "Customers" where country is "Germany" AND city is "Berlin":

```
SELECT * FROM Customers
WHERE Country = 'Germany' AND City = 'Berlin';
```

OR Example

The following SQL statement selects all fields from "Customers" where city is "Berlin" OR "Stuttgart":

```
SELECT * FROM Customers
WHERE City = 'Berlin' OR City = 'Stuttgart';
```

The following SQL statement selects all fields from "Customers" where country is "Germany" OR "Spain":

```
SELECT * FROM Customers
WHERE Country = 'Germany' OR Country = 'Spain';
```

NOT Example

The following SQL statement selects all fields from "Customers" where country is NOT "Germany":

```
SELECT * FROM Customers
WHERE NOT Country = 'Germany';
```

Combining AND, OR and NOT

You can also combine the AND, OR and NOT operators.

The following SQL statement selects all fields from "Customers" where country is "Germany" AND city must be "Berlin" OR "Stuttgart" (use parentheses to form complex expressions):

```
SELECT * FROM Customers
WHERE Country = 'Germany' AND (City = 'Berlin' OR City =
'Stuttgart');
```

The following SQL statement selects all fields from "Customers" where country is NOT "Germany" and NOT "USA":

```
SELECT * FROM Customers
WHERE NOT Country = 'Germany' AND NOT Country = 'USA';
```

The MySQL ORDER BY Keyword

The ORDER BY keyword is used to sort the result-set in ascending or descending order.

The ORDER BY keyword sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword.

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column1, column2, ... ASC|DESC;
```

Demo Database:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

ORDER BY Example

The following SQL statement selects all customers from the "Customers" table, sorted by the "Country" column:

```
SELECT * FROM Customers
ORDER BY Country;
```

ORDER BY DESC Example

The following SQL statement selects all customers from the "Customers" table, sorted DESCENDING by the "Country" column:

```
SELECT * FROM Customers
ORDER BY Country DESC;
```

ORDER BY Several Columns Example

The following SQL statement selects all customers from the "Customers" table, sorted by the "Country" and the "CustomerName" column. This means that it orders by Country, but if some rows have the same Country, it orders them by CustomerName:

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

ORDER BY Several Columns Example 2

The following SQL statement selects all customers from the "Customers" table, sorted ascending by the "Country" and descending by the "CustomerName" column:

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

The MySQL INSERT INTO Statement

The INSERT INTO statement is used to insert new records in a table.

INSERT INTO Syntax

It is possible to write the INSERT INTO statement in two ways:

1. Specify both the column names and the values to be inserted:

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

2. If you are adding values for all the columns of the table, you do not need to specify the column names in the SQL query. However, make sure the order of the values is in the same order as the columns in the table. Here, the INSERT INTO syntax would be as follows:

```
INSERT INTO table_name
VALUES (value1, value2, value3, ...);
```

Demo Database:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

INSERT INTO Example

The following SQL statement inserts a new record in the "Customers" table:

```
INSERT INTO Customers (CustomerName, ContactName, Address, City,
PostalCode, Country) VALUES ('Cardinal', 'Tom B. Erichsen', 'Skagen
21', 'Stavanger', '4006', 'Norway');
```

The selection from the "Customers" table will now look like this:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
89	White Clover Markets	Karl Jablonski	305 - 14th Ave. S. Suite 3B	Seattle	98128	USA
90	Wilman Kala	Matti Karttunen	Keskuskatu 45	Helsinki	21240	Finland
91	Wolski	Zbyszek	ul. Filtrowa 68	Walla	01-012	Poland
92	Cardinal	Tom B. Erichsen	Skagen 21	Stavanger	4006	Norway

Insert Data Only in Specified Columns
It is also possible to only insert data in specific columns.
The following SQL statement will insert a new record, but only insert data in the "CustomerName", "City", and "Country" columns (CustomerID will be updated automatically):

```
INSERT INTO Customers (CustomerName, City, Country)
VALUES ('Cardinal', 'Stavanger', 'Norway');
```

The following will be the results:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
89	White Clover Markets	Karl Jablonski	305 - 14th Ave. S. Suite 3B	Seattle	98128	USA
90	Wilman Kala	Matti Karttunen	Keskuskatu 45	Helsinki	21240	Finland
91	Wolski	Zbyszek	ul. Filtrowa 68	Walla	01-012	Poland
92	Cardinal	null	null	Stavanger	null	Norway

MySQL NULL Values

What is a NULL Value?

A field with a NULL value is a field with no value.

If a field in a table is optional, it is possible to insert a new record or update a record without adding a value to this field. Then, the field will be saved with a NULL value.

Note: A NULL value is different from a zero value or a field that contains spaces. A field with a NULL value is one that has been left blank during record creation!

How to Test for NULL Values?

It is not possible to test for NULL values with comparison operators, such as =, <, or <>.

We will have to use the IS NULL and IS NOT NULL operators instead.

IS NULL Syntax

```
SELECT column_names
FROM table_name
WHERE column_name IS NULL;
```

IS NOT NULL Syntax

```
SELECT column_names
FROM table_name
WHERE column_name IS NOT NULL;
```

Demo Database:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

Following SQL lists all customers with a NULL value in the "Address" field:

```
SELECT CustomerName, ContactName, Address
FROM Customers
WHERE Address IS NULL;
```

Tip: Always use IS NULL to look for NULL values.

Following SQL lists all customers with a value in the "Address" field:

```
SELECT CustomerName, ContactName, Address
FROM Customers
WHERE Address IS NOT NULL;
```

The MySQL UPDATE Statement

The UPDATE statement is used to modify the existing records in a table.

UPDATE Syntax

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

Demo Database:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

UPDATE Table

The following SQL statement updates the first customer (CustomerID = 1) with a new contact person and a new city.

```
UPDATE Customers
SET ContactName = 'Alfred Schmidt', City = 'Frankfurt'
WHERE CustomerID = 1;
```

The following will be the results:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Alfred Schmidt	Obere Str. 57	Frankfurt	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK

UPDATE Multiple Records

It is the WHERE clause that determines how many records will be updated.

The following SQL statement will update the PostalCode to 00000 for all records where country is "Mexico":

```
UPDATE Customers
SET PostalCode = 00000
WHERE Country = 'Mexico';
```

The following will be the results:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Alfred Schmidt	Obere Str. 57	Frankfurt	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	00000	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	00000	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK

Update Warning!

Be careful when updating records. If you omit the WHERE clause, ALL records will be updated!

```
UPDATE Customers
SET PostalCode = 00000;
```

The MySQL DELETE Statement

The DELETE statement is used to delete existing records in a table.

DELETE Syntax

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

Note: Be careful when deleting records in a table! Notice the WHERE clause in the DELETE statement. The WHERE clause specifies which record(s) should be deleted. If you omit the WHERE clause, all records in the table will be deleted!

Demo Database:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

SQL DELETE Example

The following SQL statement deletes the customer "Alfreds Futterkiste" from the "Customers" table:

```
DELETE FROM Customers WHERE CustomerName='Alfreds Futterkiste';
```

The "Customers" table will now look like this:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

Delete All Records

It is possible to delete all rows in a table without deleting the table. This means that the table structure, attributes, and indexes will be intact:

```
DELETE FROM table_name;
```

The following SQL statement deletes all rows in the "Customers" table, without deleting the table:

```
DELETE FROM Customers;
```

The MySQL LIMIT Clause

The LIMIT clause is used to specify the number of records to return.

The LIMIT clause is useful on large tables with thousands of records. Returning a large number of records can impact performance.

```
SELECT column_name(s)
FROM table_name
WHERE condition
LIMIT number;
```

Demo Database:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

MySQL LIMIT Examples

The following SQL statement selects the first three records from the "Customers" table:

```
SELECT * FROM Customers
LIMIT 3;
```

What if we want to select records 4 - 6 (inclusive)?

MySQL provides a way to handle this: by using OFFSET.

SQL query below says "return only 3 records, start on record 4 (OFFSET 3)":

```
SELECT * FROM Customers
LIMIT 3 OFFSET 3;
```

ADD a WHERE CLAUSE

The following SQL statement selects the first three records from the "Customers" table, where the country is "Germany":

```
SELECT * FROM Customers
WHERE Country='Germany'
LIMIT 3;
```

MySQL MIN() and MAX() Functions

The MIN() function returns the smallest value of the selected column.

The MAX() function returns the largest value of the selected column.

MIN() Syntax:

```
SELECT MIN(column_name)
FROM table_name
WHERE condition;
```

MAX() Syntax:

```
SELECT MAX(column_name)
FROM table_name
WHERE condition;
```

Demo Database:

ProductID	ProductName	SupplierID	CategoryID	Unit	Price
1	Chais	1	1	10 boxes x 20 bags	18
2	Chang	1	1	24 - 12 oz bottles	19
3	Aniseed Syrup	1	2	12 - 550 ml bottles	10
4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	22
5	Chef Anton's Gumbo Mix	2	2	36 boxes	21.35

MIN() Example

The following SQL statement finds the price of the cheapest product:

```
SELECT MIN(Price) AS SmallestPrice
FROM Products;
```

MAX() Example

The following SQL statement finds the price of the most expensive product:

```
SELECT MAX(Price) AS LargestPrice
FROM Products;
```

MySQL COUNT(), AVG() and SUM() Functions

The COUNT() function returns the number of rows that matches a specified criterion.

COUNT() Syntax:

```
SELECT COUNT(column_name)
FROM table_name
WHERE condition;
```

The AVG() function returns the average value of a numeric column.

AVG() Syntax:

```
SELECT AVG(column_name)
FROM table_name
WHERE condition;
```

The SUM() function returns the total sum of a numeric column.

SUM() Syntax:

```
SELECT SUM(column_name)
FROM table_name
WHERE condition;
```

Demo Database:

ProductID	ProductName	SupplierID	CategoryID	Unit	Price
1	Chais	1	1	10 boxes x 20 bags	18
2	Chang	1	1	24 - 12 oz bottles	19
3	Aniseed Syrup	1	2	12 - 550 ml bottles	10
4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	22
5	Chef Anton's Gumbo Mix	2	2	36 boxes	21.35

COUNT() Example

The following SQL statement finds the number of products:

```
SELECT COUNT(ProductID)
FROM Products;
```

Note: NULL values are not counted.

AVG() Example

The following SQL statement finds the average price of all products:

```
SELECT AVG(Price)
FROM Products;
```

Demo Database:

OrderDetailID	OrderID	ProductID	Quantity
1	10248	11	12
2	10248	42	10
3	10248	72	5
4	10249	14	9
5	10249	51	40

SUM() Example

The following SQL statement finds the sum of the "Quantity" fields in the "OrderDetails" table:

```
SELECT SUM(Quantity)
FROM OrderDetails;
```

Note: NULL values are ignored.

The MySQL LIKE Operator

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

There are two wildcards often used in conjunction with the LIKE operator:

The percent sign (%) represents zero, one, or multiple characters
The underscore sign (_) represents one, single character
The percent sign and the underscore can also be used in combinations!

LIKE Syntax

```
SELECT column1, column2, ...
FROM table_name
WHERE columnN LIKE pattern;
```

Tip: You can also combine any number of conditions using AND or OR operators.

Here are some examples showing different LIKE operators with '%' and '_' wildcards:

LIKE Operator	Description
WHERE CustomerName LIKE 'a%'	Finds any values that start with "a"
WHERE CustomerName LIKE '%a'	Finds any values that end with "a"
WHERE CustomerName LIKE '%or%'	Finds any values that have "or" in any position
WHERE CustomerName LIKE '_r%'	Finds any values that have "r" in the second position
WHERE CustomerName LIKE 'a_%'	Finds any values that start with "a" and are at least 2 characters in length
WHERE CustomerName LIKE 'a__%'	Finds any values that start with "a" and are at least 3 characters in length
WHERE ContactName LIKE 'a%o'	Finds any values that start with "a" and ends with "o"

Demo Database:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

The following SQL statement selects all customers with a CustomerName starting with "a":

```
SELECT * FROM Customers
WHERE CustomerName LIKE 'a%';
```

The following SQL statement selects all customers with a CustomerName ending with "a":

```
SELECT * FROM Customers
WHERE CustomerName LIKE '%a';
```

The following SQL statement selects all customers with a CustomerName that have "or" in any position:

```
SELECT * FROM Customers
WHERE CustomerName LIKE '%or%';
```

The following SQL statement selects all customers with a CustomerName that have "r" in the second position:

```
SELECT * FROM Customers
WHERE CustomerName LIKE '_r%';
```

The following SQL statement selects all customers with a CustomerName that starts with "a" and are at least 3 characters in length:

```
SELECT * FROM Customers
WHERE CustomerName LIKE 'a__%';
```

The following SQL statement selects all customers with a ContactName that starts with "a" and ends with "o":

```
SELECT * FROM Customers
WHERE ContactName LIKE 'a%o';
```

The following SQL statement selects all customers with a CustomerName that does NOT start with "a":

```
SELECT * FROM Customers
WHERE CustomerName NOT LIKE 'a%';
```

The MySQL IN Operator

The IN operator allows you to specify multiple values in a WHERE clause.

The IN operator is a shorthand for multiple OR conditions.

IN Syntax

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (value1, value2, ...);
```

or:

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (SELECT STATEMENT);
```

Demo Database:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

IN Operator Examples

The following SQL statement selects all customers that are located in "Germany", "France" or "UK":

```sql
SELECT * FROM Customers
WHERE Country IN ('Germany', 'France', 'UK');
```

The following SQL statement selects all customers that are NOT located in "Germany", "France" or "UK":

```sql
SELECT * FROM Customers
WHERE Country NOT IN ('Germany', 'France', 'UK');
```

The following SQL statement selects all customers that are from the same countries as the suppliers:

```sql
SELECT * FROM Customers
WHERE Country IN (SELECT Country FROM Suppliers);
```

The MySQL BETWEEN Operator

The BETWEEN operator selects values within a given range. The values can be numbers, text, or dates.

The BETWEEN operator is inclusive: begin and end values are included.

Between Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE column_name BETWEEN value1 AND value2;
```

Demo Database:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

BETWEEN Example

The following SQL statement selects all products with a price between 10 and 20:

```
SELECT * FROM Products
WHERE Price BETWEEN 10 AND 20;
```

NOT BETWEEN Example

To display the products outside the range of the previous example, use NOT BETWEEN:

```
SELECT * FROM Products
WHERE Price NOT BETWEEN 10 AND 20;
```

BETWEEN with IN Example

The following SQL statement selects all products with a price between 10 and 20. In addition; do not show products with a CategoryID of 1,2, or 3:

```
SELECT * FROM Products
WHERE Price BETWEEN 10 AND 20
AND CategoryID NOT IN (1,2,3);
```

BETWEEN Text Values Example

The following SQL statement selects all products with a ProductName between "Carnarvon Tigers" and "Mozzarella di Giovanni":

```
SELECT * FROM Products
WHERE ProductName BETWEEN 'Carnarvon Tigers' AND 'Mozzarella di
Giovanni' ORDER BY ProductName;
```

The following SQL statement selects all products with a ProductName between "Carnarvon Tigers" and "Chef Anton's Cajun Seasoning":

```
SELECT * FROM Products
WHERE ProductName BETWEEN "Carnarvon Tigers" AND "Chef Anton's Cajun
Seasoning" ORDER BY ProductName;
```

NOT BETWEEN Text Values Example

The following SQL statement selects all products with a ProductName not between "Carnarvon Tigers" and "Mozzarella di Giovanni":

```
SELECT * FROM Products
WHERE ProductName NOT BETWEEN 'Carnarvon Tigers' AND 'Mozzarella di
Giovanni' ORDER BY ProductName;
```

Sample Database:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10248	90	5	7/4/1996	3
10249	81	6	7/5/1996	1
10250	34	4	7/8/1996	2
10251	84	3	7/9/1996	1
10252	76	4	7/10/1996	2

BETWEEN Dates Example

The following SQL statement selects all orders with an OrderDate between '01-July-1996' and '31-July-1996':

```
SELECT * FROM Orders
WHERE OrderDate BETWEEN '1996-07-01' AND '1996-07-31';
```

MySQL Aliases

Aliases are used to give a table, or a column in a table, a temporary name.

Aliases are often used to make column names more readable.

An alias only exists for the duration of that query.

An alias is created with the AS keyword.

Alias Column Syntax:

```
SELECT column_name AS alias_name
FROM table_name;
```

Alias Table Syntax:

```
SELECT column_name(s)
FROM table_name AS alias_name;
```

Sample Database 1:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10354	58	8	1996-11-14	3
10355	4	6	1996-11-15	1
10356	86	6	1996-11-18	2

Sample Database 2:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK

Alias for Columns Examples

The following SQL statement creates two aliases, one for the CustomerID column and one for the CustomerName column:

```
SELECT CustomerID AS ID, CustomerName AS Customer
FROM Customers;
```

The following SQL statement creates two aliases, one for the CustomerName column and one for the ContactName column.

Note: Single or double quotation marks are required if the alias name contains spaces:

```
SELECT CustomerName AS Customer, ContactName AS "Contact Person"
FROM Customers;
```

The following SQL statement creates an alias named "Address" that combine four columns (Address, PostalCode, City and Country):

```
SELECT CustomerName, CONCAT_WS(', ', Address, PostalCode, City,
Country) AS Address
FROM Customers;
```

Alias for Tables Example

The following SQL statement selects all the orders from the customer with CustomerID=4 (Around the Horn). We use the "Customers" and "Orders" tables, and give them the table aliases of "c" and "o" respectively (Here we use aliases to make the SQL shorter):

```
SELECT o.OrderID, o.OrderDate, c.CustomerName
FROM Customers AS c, Orders AS o
WHERE c.CustomerName='Around the Horn' AND c.CustomerID=o.CustomerID;
```

The following SQL statement is the same as above, but without aliases:

```
SELECT Orders.OrderID, Orders.OrderDate, Customers.CustomerName
FROM Customers, Orders
WHERE Customers.CustomerName='Around the Horn' AND
Customers.CustomerID=Orders.CustomerID;
```

Aliases can be useful when:

- There are more than one table involved in a query.
- Functions are used in the query.
- Column names are big or not very readable.
- Two or more columns are combined together.

MySQL Joins:

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

Let's look at a selection from the "Orders" table:

OrderID	CustomerID	OrderDate
10308	2	1996-09-18
10309	37	1996-09-19
10310	77	1996-09-20

Then, look at a selection from the "Customers" table:

CustomerID	CustomerName	ContactName	Country
1	Alfreds Futterkiste	Maria Anders	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mexico

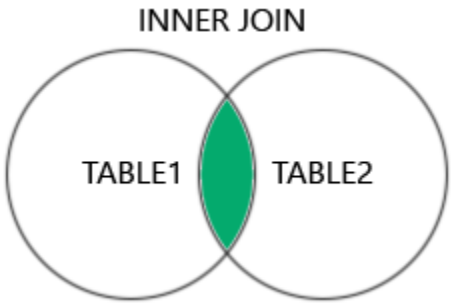
```
SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate
FROM Orders
INNER JOIN Customers ON Orders.CustomerID=Customers.CustomerID;
```

Results:

OrderID	CustomerName	OrderDate
10308	Ana Trujillo Emparedados y helados	9/18/1996
10365	Antonio Moreno Taquería	11/27/1996
10383	Around the Horn	12/16/1996
10355	Around the Horn	11/15/1996
10278	Berglunds snabbköp	8/12/1996

Types of Joins in MySQL:

INNER JOIN: Returns records that have matching values in both tables.



Inner Join Syntax:

```
SELECT columns_from_both_tables
FROM table1
INNER JOIN table2
ON table1.column1 = table2.column2
```

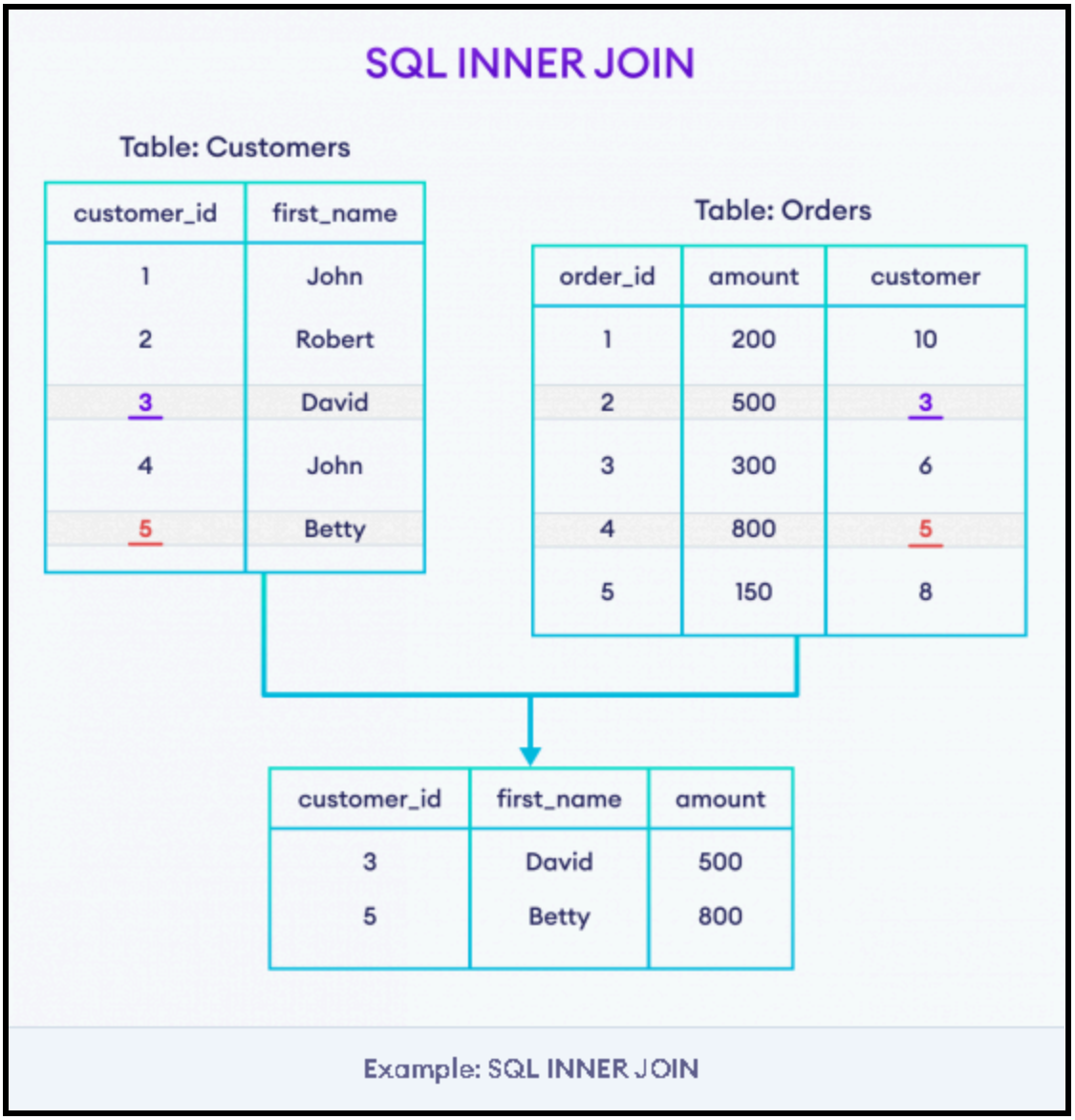
Here, table1 and table2 - two tables that are to be joined
column1 and column2 - columns common to in table1 and table2

Example 1: SQL INNER JOIN

```
-- join the Customers and Orders tables
-- with customer_id and customer fields

SELECT Customers.customer_id, Customers.first_name, Orders.amount
FROM Customers
INNER JOIN Orders
ON Customers.customer_id = Orders.customer;
```

Here, the SQL command selects the specified rows from both tables if the values of customer_id (of the Customers table) and customer (of the Orders table) are a match.



As you can see, INNER JOIN excludes all the rows that are not common between two tables.

Note: We can also use SQL JOIN instead of INNER JOIN. Basically, these two clauses are the same.

Example 2: Join Two Tables With a Matching Field

```
-- join Categories and Products tables
-- with their matching fields cat_id

SELECT Categories.cat_name, Products.prod_title
FROM Categories
INNER JOIN Products
ON Categories.cat_id = Products.cat_id;
```

Here, the SQL command selects common rows between Categories and Products tables with the matching field cat_id.

The result set has the cat_name column from Categories and the prod_title column from Products.

SQL INNER JOIN With AS Alias

We can use AS aliases inside INNER JOIN to make our query short and clean. For example,

```
-- use alias C for Categories table
-- use alias P for Products table

SELECT C.cat_name, P.prod_title
FROM Categories AS C
INNER JOIN Products AS P
ON C.cat_id= P.cat_id;
```

Here, the SQL command performs an inner join on the Categories and Products tables while assigning the aliases C and P to them, respectively.

SQL INNER JOIN With Three Tables

We can also join more than two tables using INNER JOIN. For example,

```
-- join three tables: Customers, Orders, and Shippings

SELECT C.customer_id, C.first_name, O.amount, S.status
FROM Customers AS C
INNER JOIN Orders AS O
ON C.customer_id = O.customer
INNER JOIN Shippings AS S
ON C.customer_id = S.customer;
```

Here, the SQL command joins Customers and Orders tables based on customer_id (from the Customers table) and customer (from the Orders table) and joins Customers and Shippings tables based on customer_id (from the Customers table) and customer (from the Shippings table)

The command returns the rows where there is a match between column values in both join conditions.

Note: For this command to run, there must be a customer_id column in each individual table. The column names can be different as long as they have common data.

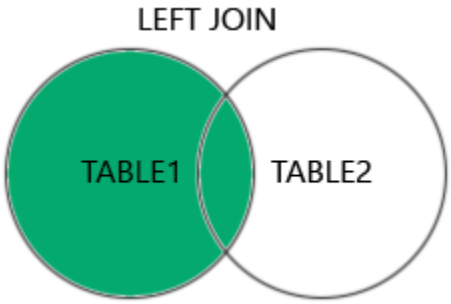
INNER JOIN With WHERE Clause

```
-- join Customers and Orders table
-- with customer_id and customer fields

SELECT Customers.customer_id, Customers.first_name, Orders.amount
FROM Customers
INNER JOIN Orders
ON Customers.customer_id = Orders.customer
WHERE Orders.amount >= 500;
```

Here, the SQL command joins two tables and selects rows where the amount is greater than or equal to 500.

LEFT JOIN: Returns all records from the left table, and the matched records from the right table.



SQL LEFT JOIN Syntax

```
SELECT columns_from_both_tables
FROM table1
LEFT JOIN table2
ON table1.column1 = table2.column2
```

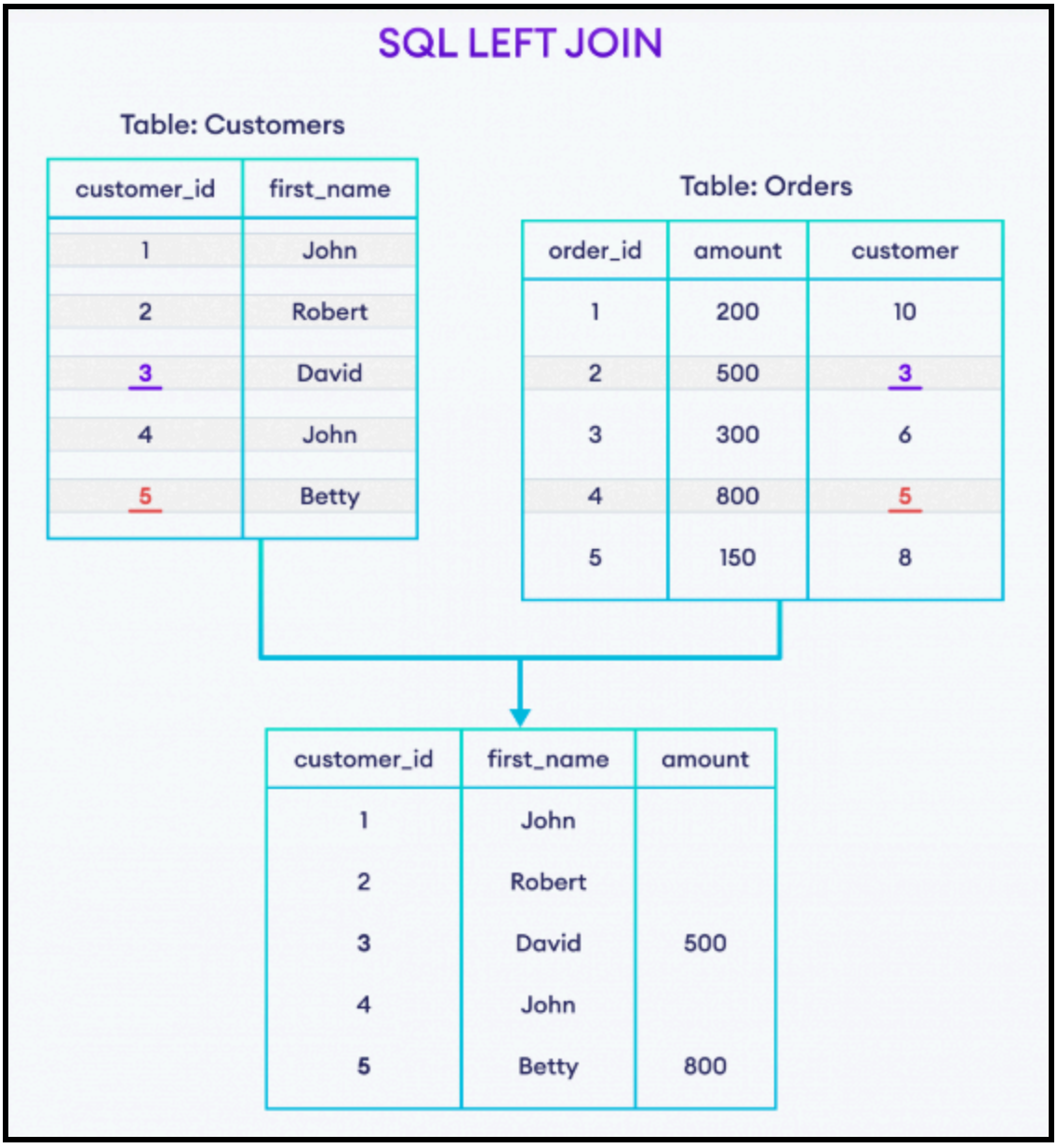
Here, table1 is the left table to be joined
table2 is the right table to be joined
column1 and column2 are the common columns in the two tables

The SQL LEFT JOIN combines two tables based on a common column. It then selects records having matching values in these columns and the remaining rows from the left table.

Example:

```
SELECT Customers.customer_id, Customers.first_name, Orders.item
FROM Customers
LEFT JOIN Orders
ON Customers.customer_id = Orders.customer_id;
```

Here, the code left joins the Customers and Orders tables based on customer_id, which is common to both tables.



Here, the SQL command combines data from the Customers and Orders tables.

The query selects the customer_id and first_name from Customers and the amount from Orders.

Hence, the result includes rows where customer_id from Customers matches customer from Orders.

JOIN With AS Alias

We can use AS aliases inside LEFT JOIN to make our query short and clean.

```
-- use alias C for Categories table
-- use alias P for Products table

SELECT C.cat_name, P.prod_title
FROM Categories AS C
LEFT JOIN Products AS P
ON C.cat_id= P.cat_id;
```

Here, the command left joins the Categories and Products tables while assigning the aliases C and P to them, respectively.

NULL Values in LEFT JOIN

When performing a LEFT JOIN, it's common to encounter NULL values in the result set for rows in the right table that don't have a matching row in the left table.

To handle these NULL values, we can use the COALESCE() function. For example,

```
-- display order status for each customer, handling NULL values

SELECT c.customer_id, COALESCE(s.status, 'No Orders') AS order_status
FROM Customers c
LEFT JOIN Shippings s ON c.customer_id = s.customer
ORDER BY c.customer_id;
```

In this command, we're joining the Customers table with the Shippings table. The COALESCE() function ensures that if the status column is NULL, it displays No Orders.

This way, we always have an order status for each customer.

LEFT JOIN With WHERE Clause

We can use the LEFT JOIN statement with an optional WHERE clause. For example,

```
SELECT Customers.customer_id, Customers.first_name, Orders.amount
FROM Customers
LEFT JOIN Orders
ON Customers.customer_id = Orders.customer
WHERE Orders.amount >= 500;
```

Here, the SQL command joins the Customers and Orders tables and selects rows where the amount is greater than or equal to 500.

LEFT JOIN Multiple Tables

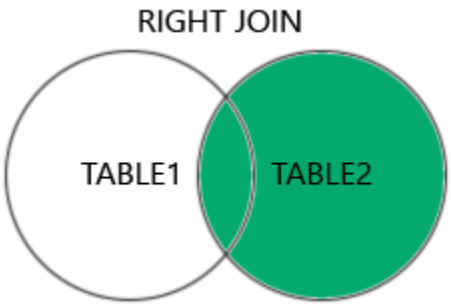
We can also use LEFT JOIN to combine more than two tables.

```
-- join Customers, Orders, and Shippings tables
SELECT C.customer_id, C.first_name, O.amount, S.status
FROM Customers C
LEFT JOIN Orders O
ON C.customer_id = O.customer_id
LEFT JOIN Shippings S
ON C.customer_id = S.customer;
```

This command will join three tables and return a row for each customer, including their order amounts and shipping status, if available.

Note: To learn more about joining multiple tables, visit SQL JOIN Multiple Tables.

RIGHT JOIN: Returns all records from the right table, and the matched records from the left table.



RIGHT JOIN SYNTAX

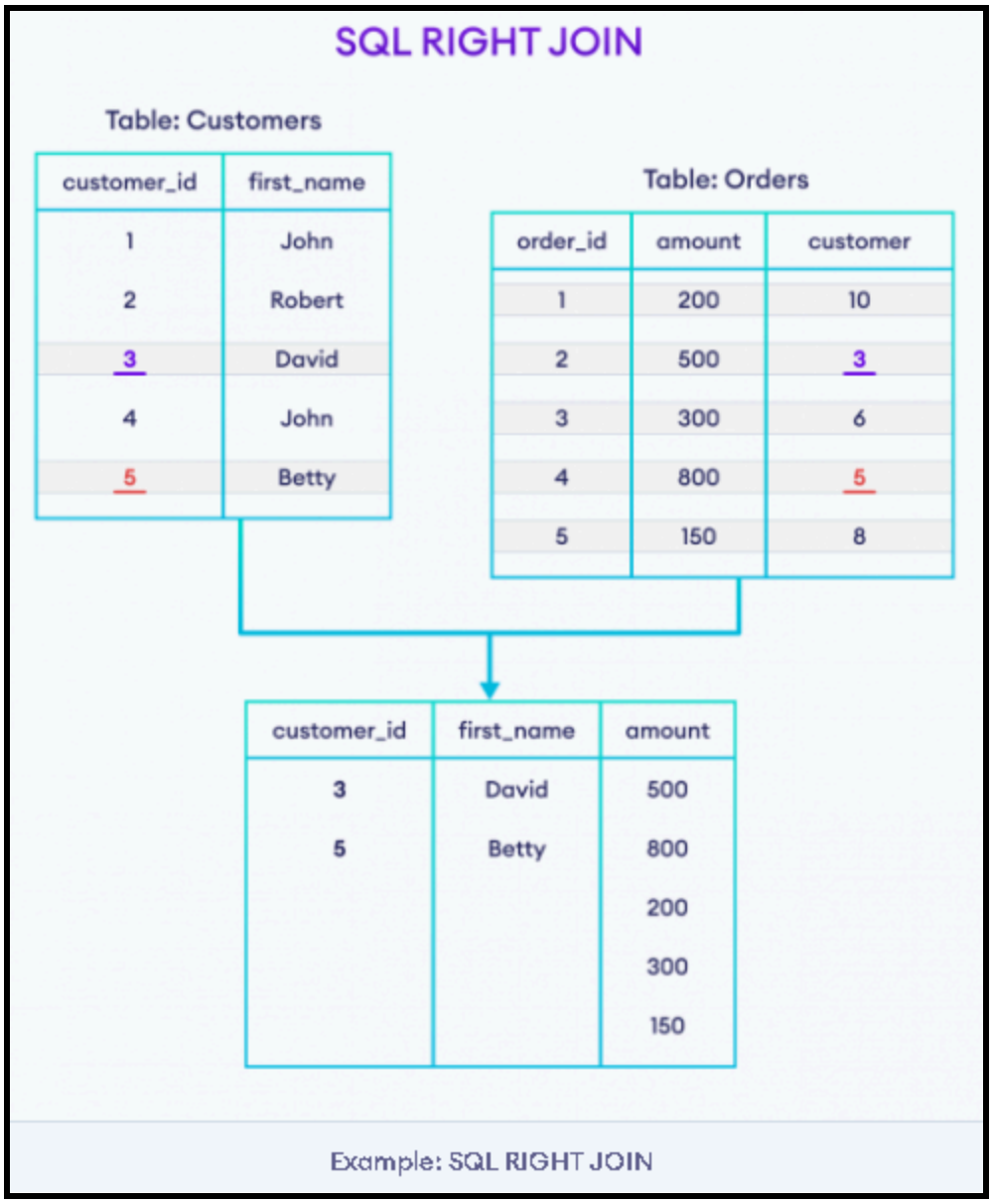
```
SELECT columns_from_both_tables
FROM table1
RIGHT JOIN table2
ON table1.column1 = table2.column2
```

Here,table1 is the left table to be joined
table2 is the right table to be joined
column1 and column2 are the related columns in the two tables

Example: SQL RIGHT JOIN

```
-- join Customers and Orders tables
-- based on customer_id of Customers and customer of Orders
-- Customers is the left table
-- Orders is the right table

SELECT Customers.customer_id, Customers.first_name, Orders.amount
FROM Customers
RIGHT JOIN Orders
ON Customers.customer_id = Orders.customer;
```



Here, the SQL command selects the customer_id and first_name columns (from the Customers table) and the amount column (from the Orders table).

And, the result set will contain those rows where there is a match between customer_id (of the Customers table) and customer (of the Orders table), along with all the remaining rows from the Orders table.

RIGHT JOIN With WHERE Clause

The SQL RIGHT JOIN statement can have an optional WHERE clause. For example,

```
SELECT Customers.customer_id, Customers.first_name, Orders.amount
FROM Customers
RIGHT JOIN Orders
ON Customers.customer_id = Orders.customer
WHERE Orders.amount >= 500;
```

Here, the SQL command joins the Customers and Orders tables and selects rows where the amount is greater than or equal to 500.

SQL RIGHT JOIN With AS Alias

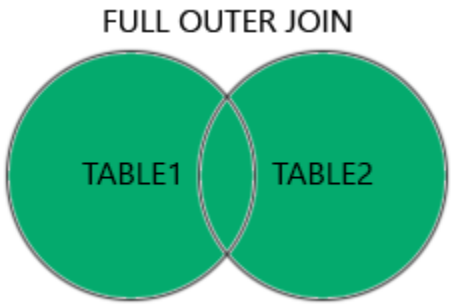
We can use AS aliases inside RIGHT JOIN to make our SQL code short and clean. For example,

```
-- use alias C for Categories table
-- use alias P for Products table

SELECT C.category_name, P.product_title
FROM Categories AS C
RIGHT JOIN Products AS P
ON C.cat_id = P.cat_id;
```

Here, the SQL command performs a right join on the Categories and Products tables while assigning the aliases C and P to them, respectively.

FULL OUTER JOIN: The FULL OUTER JOIN keyword returns all records when there is a match in left (table1) or right (table2) table records.



FULL OUTER JOIN SYNTAX

```
SELECT columns
FROM table1
FULL OUTER JOIN table2
ON table1.column1 = table2.column2;
```

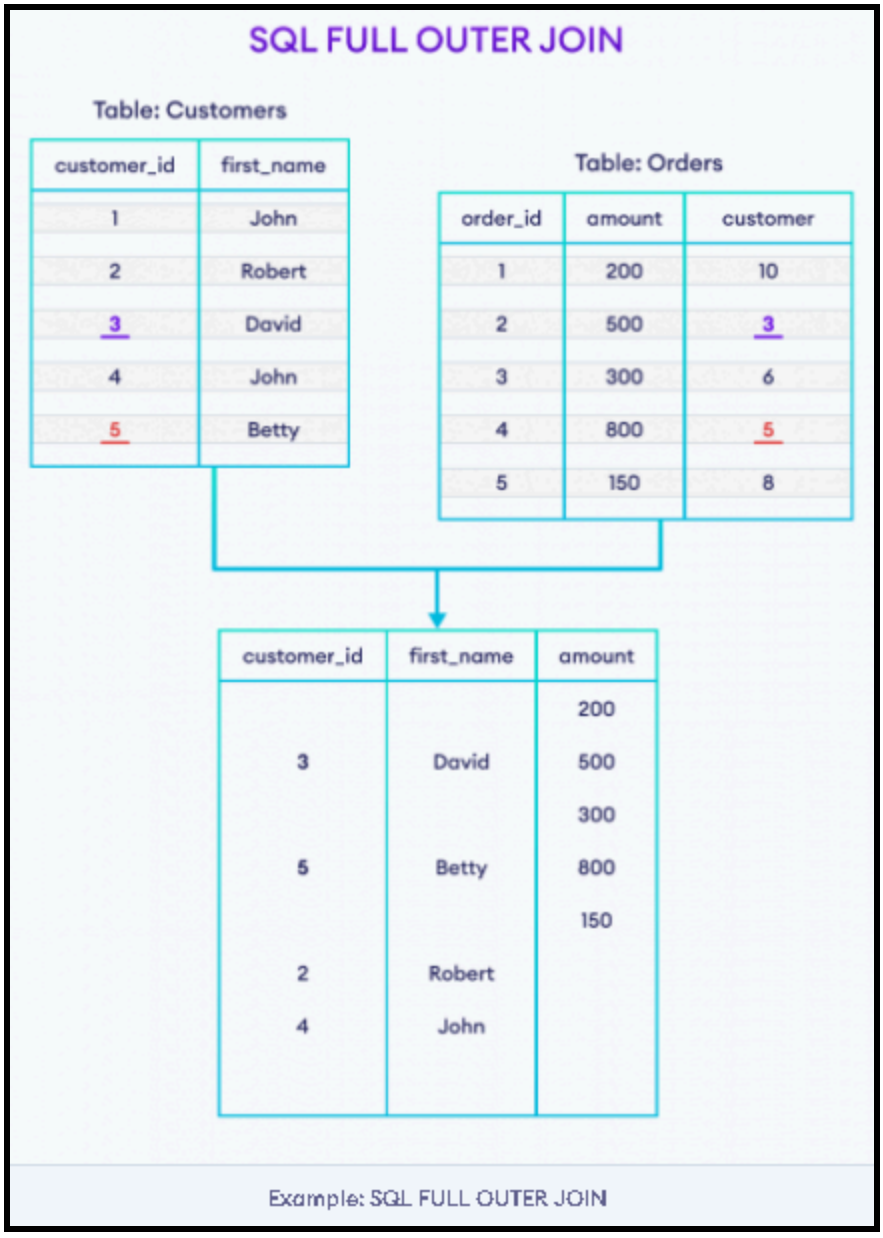
Here, table1 and table2 are the tables to be joined
column1 and column2 are the related columns in the two tables

Example: SQL OUTER Join

```
SELECT Customers.customer_id, Customers.first_name, Orders.amount
FROM Customers
FULL OUTER JOIN Orders
ON Customers.customer_id = Orders.customer;
```

Here, the SQL command selects the customer_id and first_name columns (from the Customers table) and the amount column (from the Orders table).

The result set will contain all rows of both the tables, regardless of whether there is a match between customer_id (of the Customers table) and customer (of the Orders table).


SQL FULL OUTER JOIN
Table: Customers
customer_id
first_name
1
John
2
Robert
3
David
4
John
5
Betty
Table: Orders
order_id
amount
customer
1
200
10
2
500
3
3
300
6
4
800
5
5
150
8
customer_id
first_name
amount
200
3
David
500
300
5
Betty
800
150
2
Robert
4
John
Example: SQL FULL OUTER JOIN

FULL OUTER JOIN With WHERE Clause

The SQL FULL OUTER JOIN statement can have an optional WHERE clause. For example,

```
SELECT Customers.customer_id, Customers.first_name, Orders.amount
FROM Customers
FULL OUTER JOIN Orders
ON Customers.customer_id = Orders.customer
WHERE Orders.amount >= 500;
```

Here, the SQL command joins two tables and selects rows where the amount is greater than or equal to 500.

SQL FULL OUTER JOIN With AS Alias

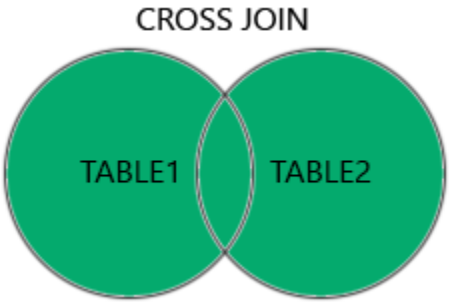
We can use AS aliases inside FULL OUTER JOIN to make our query short and clean. For example,

```
-- use alias C for Categories table
-- use alias P for Products table

SELECT C.category_name, P.product_title
FROM Categories AS C
FULL OUTER JOIN Products AS P
ON C.category_id = P.cat_id;
```

Here, the SQL command performs a full outer join on the Categories and Products tables while assigning the aliases C and P to them, respectively.

CROSS JOIN: Returns all records from both tables.



CROSS JOIN Syntax:

```
SELECT column1, column2, ...
FROM table1
CROSS JOIN table2;
```

Here, column1 and column2 - the table columns
table1 and table2 - the names of the tables we want to combine

Example: SQL CROSS JOIN

```
SELECT Customers.customer_id, Customers.first_name, Orders.order_id
FROM Customers
CROSS JOIN Orders;
```

Here, the SQL command performs a CROSS JOIN operation between the Customers and Orders tables.

This creates a Cartesian product of all customer IDs and first names with order IDs.

The result is a combination of every customer with every order.

CROSS JOIN With Multiple Tables

We can also perform CROSS JOIN with more than two tables. For example,

```
SELECT Customers.customer_id, Orders.item, Shippings.status
FROM Customers
CROSS JOIN Orders
CROSS JOIN Shippings;
```

Here, the SQL command combines rows from the Customers, Orders, and Shippings tables to create a Cartesian product.

SQL CROSS JOIN With Aliases

We can use aliases with table names to make our snippet short and clean. For example,

```
SELECT c.customer_id, o.item, s.status
FROM Customers c
CROSS JOIN Orders o
CROSS JOIN Shippings s;
```

In this example, c, o, and s are aliases for the Customers, Orders, and Shippings tables, respectively.

These aliases make the query more concise and readable.
Cross join :Cross Joins produce results that consist of every combination of rows from two or more tables. That means if table A has 3 rows and table B has 2 rows, a CROSS JOIN will result in 6 rows. There is no relationship established between the two tables – you literally just produce every possible combination.

Full outer Join : A FULL OUTER JOIN is neither "left" nor "right"— it's both! It includes all the rows from both of the tables or result sets participating in the JOIN. When no matching rows exist for rows on the "left" side of the JOIN, you see Null values from the result set on the "right." Conversely, when no matching rows exist for rows on the "right" side of the JOIN, you see Null values from the result set on the "left."

SELF JOIN: In SQL, the Self JOIN operation allows us to join a table with itself, creating a relationship between rows within the same table

Self JOIN Syntax

```
SELECT columns
FROM table1 T1,
JOIN table1 T2 ON
WHERE condition;
```

Here,columns - specifies the columns we want to retrieve
table1 T1 and table1 T2 - two instances T1 and T2 for the same table table1
JOIN - connects two tables and is usually followed by an ON command that specifies the common columns used for linking the two tables.
condition - specifies the condition specifying how the two instances of the same table should be joined

Example: SQL Self JOIN

```
-- retrieve Customers with the Same Country and Different Customer
IDs

SELECT
    c1.first_name,
    c1.country,
    c2.first_name
FROM Customers c1
JOIN Customers c2 ON c1.country = c2.country
WHERE c1.customer_id <> c2.customer_id;
```

Here, the SQL command uses self-join on the Customers table to find customers who have the same country but distinct customer IDs.

The MySQL UNION Operator

The UNION operator is used to combine the result-set of two or more SELECT statements.

Every SELECT statement within UNION must have the same number of columns
The columns must also have similar data types
The columns in every SELECT statement must also be in the same order

UNION Syntax

```
SELECT column_name(s) FROM table1
UNION
SELECT column_name(s) FROM table2;
```

UNION ALL Syntax

The UNION operator selects only distinct values by default. To allow duplicate values, use UNION ALL:

```
SELECT column_name(s) FROM table1
UNION ALL
SELECT column_name(s) FROM table2;
```

Note: The column names in the result-set are usually equal to the column names in the first SELECT statement.

Databases:

Below is a selection from the "Customers" table:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico

And a selection from the "Suppliers" table:

SupplierID	SupplierName	ContactName	Address	City	PostalCode	Country
1	Exotic Liquid	Charlotte Cooper	49 Gilbert St.	London	EC1 4SD	UK
2	New Orleans Cajun Delights	Shelley Burke	P.O. Box 78934	New Orleans	70117	USA
3	Grandma Kelly's Homestead	Regina Murphy	707 Oxford Rd.	Ann Arbor	48104	USA

SQL UNION Example

The following SQL statement returns the cities (only distinct values) from both the "Customers" and the "Suppliers" table:

```
SELECT City FROM Customers
UNION
SELECT City FROM Suppliers
ORDER BY City;
```

Note: If some customers or suppliers have the same city, each city will only be listed once, because UNION selects only distinct values. Use UNION ALL to also select duplicate values!

SQL UNION ALL Example

The following SQL statement returns the cities (duplicate values also) from both the "Customers" and the "Suppliers" table:

```sql
SELECT City FROM Customers
UNION ALL
SELECT City FROM Suppliers
ORDER BY City;
```

SQL UNION With WHERE

The following SQL statement returns the German cities (only distinct values) from both the "Customers" and the "Suppliers" table:

```sql
SELECT City, Country FROM Customers
WHERE Country='Germany'
UNION
SELECT City, Country FROM Suppliers
WHERE Country='Germany'
ORDER BY City;
```

SQL UNION ALL With WHERE

The following SQL statement returns the German cities (duplicate values also) from both the "Customers" and the "Suppliers" table:

```sql
SELECT City, Country FROM Customers
WHERE Country='Germany'
UNION ALL
SELECT City, Country FROM Suppliers
WHERE Country='Germany'
ORDER BY City;
```

Another UNION Example

The following SQL statement lists all customers and suppliers:

```sql
SELECT 'Customer' AS Type, ContactName, City, Country FROM Customers
UNION
SELECT 'Supplier', ContactName, City, Country FROM Suppliers;
```

Notice the "AS Type" above - it is an alias. SQL Aliases are used to give a table or a column a temporary name. An alias only exists for the duration of the query. So, here we have created a temporary column named "Type", that list whether the contact person is a "Customer" or a "Supplier".

The MySQL GROUP BY Statement

The GROUP BY statement groups rows that have the same values into summary rows, like "find the number of customers in each country".

The GROUP BY statement is often used with aggregate functions (COUNT(), MAX(), MIN(), SUM(), AVG()) to group the result-set by one or more columns.

GROUP BY Syntax

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
ORDER BY column_name(s);
```

Demo Database

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

MySQL GROUP BY Examples

The following SQL statement lists the number of customers in each country:

```
SELECT COUNT(CustomerID), Country
FROM Customers
```

```
GROUP BY Country;
```

The following SQL statement lists the number of customers in each country, sorted high to low:

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country
ORDER BY COUNT(CustomerID) DESC;
```

Demo Database

Below is a selection from the "Orders" table in the Northwind sample database:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10248	90	5	1996-07-04	3
10249	81	6	1996-07-05	1
10250	34	4	1996-07-08	2

And a selection from the "Shippers" table:

ShipperID	ShipperName
1	Speedy Express
2	United Package
3	Federal Shipping

GROUP BY With JOIN Example

The following SQL statement lists the number of orders sent by each shipper:

```
SELECT Shippers.ShipperName, COUNT(Orders.OrderID) AS NumberOfOrders
FROM Orders
LEFT JOIN Shippers ON Orders.ShipperID = Shippers.ShipperID
GROUP BY ShipperName;
```

The MySQL HAVING Clause

The HAVING clause was added to SQL because the WHERE keyword cannot be used with aggregate functions.

HAVING Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
HAVING condition
ORDER BY column_name(s);
```

Demo Database:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

MySQL HAVING Examples

The following SQL statement lists the number of customers in each country. Only include countries with more than 5 customers:

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country
HAVING COUNT(CustomerID) > 5;
```

The following SQL statement lists the number of customers in each country, sorted high to low (Only include countries with more than 5 customers):

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country
HAVING COUNT(CustomerID) > 5
ORDER BY COUNT(CustomerID) DESC;
```

Demo Database

Below is a selection from the "Orders" table in the Northwind sample database:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10248	90	5	1996-07-04	3
10249	81	6	1996-07-05	1
10250	34	4	1996-07-08	2

And a selection from the "Employees" table:

EmployeeID	LastName	FirstName	BirthDate	Photo	Notes
1	Davolio	Nancy	1968-12-08	EmpID1.pic	Education includes a BA....
2	Fuller	Andrew	1952-02-19	EmpID2.pic	Andrew received his BTS....
3	Leverling	Janet	1963-08-30	EmpID3.pic	Janet has a BS degree....

More HAVING Examples

The following SQL statement lists the employees that have registered more than 10 orders:

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

The following SQL statement lists if the employees "Davolio" or "Fuller" have registered more than 25 orders:

```sql
SELECT Employees.LastName, COUNT(Orders.OrderID) AS NumberOfOrders
FROM Orders
INNER JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID
WHERE LastName = 'Davolio' OR LastName = 'Fuller'
GROUP BY LastName
HAVING COUNT(Orders.OrderID) > 25;
```

The MySQL EXISTS Operator

The EXISTS operator is used to test for the existence of any record in a subquery.

The EXISTS operator returns TRUE if the subquery returns one or more records.

EXISTS Syntax:

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

Demo Database

Below is a selection from the "Products" table in the Northwind sample database:

ProductID	ProductName	SupplierID	CategoryID	Unit	Price
1	Chais	1	1	10 boxes x 20 bags	18
2	Chang	1	1	24 - 12 oz bottles	19
3	Aniseed Syrup	1	2	12 - 550 ml bottles	10
4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	22
5	Chef Anton's Gumbo Mix	2	2	36 boxes	21.35

And a selection from the "Suppliers" table:

SupplierID	SupplierName	ContactName	Address	City	PostalCode	Country
1	Exotic Liquid	Charlotte Cooper	49 Gilbert St.	London	EC1 4SD	UK
2	New Orleans Cajun Delights	Shelley Burke	P.O. Box 78934	New Orleans	70117	USA
3	Grandma Kelly's Homestead	Regina Murphy	707 Oxford Rd.	Ann Arbor	48104	USA
4	Tokyo Traders	Yoshi Nagase	9-8 Sekimai Musashino-shi	Tokyo	100	Japan

MySQL EXISTS Examples

The following SQL statement returns TRUE and lists the suppliers with a product price less than 20:

```
SELECT SupplierName
FROM Suppliers
WHERE EXISTS (SELECT ProductName FROM Products WHERE
Products.SupplierID = Suppliers.supplierID AND Price < 20);
```

The following SQL statement returns TRUE and lists the suppliers with a product price equal to 22:

```
SELECT SupplierName
FROM Suppliers
WHERE EXISTS (SELECT ProductName FROM Products WHERE
Products.SupplierID = Suppliers.supplierID AND Price = 22);
```

The MySQL ANY and ALL Operators

The ANY and ALL operators allow you to perform a comparison between a single column value and a range of other values.

The ANY operator: returns a boolean value as a result
returns TRUE if ANY of the subquery values meet the condition
ANY means that the condition will be true if the operation is true for any of the values in the range.

ANY Syntax:

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

Note: The operator must be a standard comparison operator (=, <>, !=, >, >=, <, or <=).

The ALL operator: returns a boolean value as a result
returns TRUE if ALL of the subquery values meet the condition
is used with SELECT, WHERE and HAVING statements
ALL means that the condition will be true only if the operation is true for all values in the range.

ALL Syntax With SELECT:

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

ALL Syntax With WHERE or HAVING:

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

Note: The operator must be a standard comparison operator (=, <>, !=, >, >=, <, or <=).

Demo Database

Below is a selection from the **"Products"** table in the Northwind sample database:

ProductID	ProductName	SupplierID	CategoryID	Unit	Price
1	Chais	1	1	10 boxes x 20 bags	18
2	Chang	1	1	24 - 12 oz bottles	19
3	Aniseed Syrup	1	2	12 - 550 ml bottles	10
4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	22
5	Chef Anton's Gumbo Mix	2	2	36 boxes	21.35
6	Grandma's Boysenberry Spread	3	2	12 - 8 oz jars	25
7	Uncle Bob's Organic Dried Pears	3	7	12 - 1 lb pkgs.	30
8	Northwoods Cranberry Sauce	3	2	12 - 12 oz jars	40
9	Mishi Kobe Niku	4	6	18 - 500 g pkgs.	97

And a selection from the **"OrderDetails"** table:

OrderDetailID	OrderID	ProductID	Quantity
1	10248	11	12
2	10248	42	10
3	10248	72	5
4	10249	14	9
5	10249	51	40
6	10250	41	10
7	10250	51	35
8	10250	65	15
9	10251	22	6
10	10251	57	15

SQL ANY Examples

The following SQL statement lists the ProductName if it finds ANY records in the OrderDetails table has Quantity equal to 10 (this will return TRUE because the Quantity column has some values of 10):

```
SELECT ProductName
FROM Products
WHERE ProductID = ANY
  (SELECT ProductID
  FROM OrderDetails
  WHERE Quantity = 10);
```

The following SQL statement lists the ProductName if it finds ANY records in the OrderDetails table has Quantity larger than 99 (this will return TRUE because the Quantity column has some values larger than 99):

```
SELECT ProductName
FROM Products
WHERE ProductID = ANY
  (SELECT ProductID
  FROM OrderDetails
  WHERE Quantity > 99);
```

The following SQL statement lists the ProductName if it finds ANY records in the OrderDetails table has Quantity larger than 1000 (this will return FALSE because the Quantity column has no values larger than 1000):

```
SELECT ProductName
FROM Products
WHERE ProductID = ANY
  (SELECT ProductID
  FROM OrderDetails
  WHERE Quantity > 1000);
```

SQL ALL Examples

The following SQL statement lists ALL the product names:

```
SELECT ALL ProductName
FROM Products
WHERE TRUE;
```

The following SQL statement lists the ProductName if ALL the records in the OrderDetails table have Quantity equal to 10. This will of course return FALSE because the Quantity column has many different values (not only the value of 10):

```
SELECT ProductName
FROM Products
WHERE ProductID = ALL
  (SELECT ProductID
  FROM OrderDetails
  WHERE Quantity = 10);
```

The MySQL INSERT INTO SELECT Statement

The INSERT INTO SELECT statement copies data from one table and inserts it into another table.
The INSERT INTO SELECT statement requires that the data types in source and target tables match.
Note: The existing records in the target table are unaffected.

INSERT INTO SELECT Syntax:
Copy all columns from one table to another table:

```
INSERT INTO table2
SELECT * FROM table1
WHERE condition;
```

Copy only some columns from one table into another table:

```
INSERT INTO table2 (column1, column2, column3, ...)
SELECT column1, column2, column3, ...
FROM table1
WHERE condition;
```

Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Customers" table:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico

And a selection from the "Suppliers" table:

SupplierID	SupplierName	ContactName	Address	City	Postal Code	Country
1	Exotic Liquid	Charlotte Cooper	49 Gilbert St.	Londona	EC1 4SD	UK
2	New Orleans Cajun Delights	Shelley Burke	P.O. Box 78934	New Orleans	70117	USA
3	Grandma Kelly's Homestead	Regina Murphy	707 Oxford Rd.	Ann Arbor	48104	USA

MySQL INSERT INTO SELECT Examples

The following SQL statement copies "Suppliers" into "Customers" (the columns that are not filled with data, will contain NULL):

```
INSERT INTO Customers (CustomerName, City, Country)
SELECT SupplierName, City, Country FROM Suppliers;
```

The following SQL statement copies "Suppliers" into "Customers" (fill all columns):

```
INSERT INTO Customers (CustomerName, ContactName, Address, City,
PostalCode, Country)
SELECT SupplierName, ContactName, Address, City, PostalCode, Country
FROM Suppliers;
```

The following SQL statement copies only the German suppliers into "Customers":

```
INSERT INTO Customers (CustomerName, City, Country)
SELECT SupplierName, City, Country FROM Suppliers
WHERE Country='Germany';
```

The MySQL CASE Statement

The CASE statement goes through conditions and returns a value when the first condition is met (like an if-then-else statement). So, once a condition is true, it will stop reading and return the result. If no conditions are true, it returns the value in the ELSE clause.

If there is no ELSE part and no conditions are true, it returns NULL.

CASE Syntax:

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

Demo Database

Below is a selection from the "OrderDetails" table in the Northwind sample database:

OrderDetailID	OrderID	ProductID	Quantity
1	10248	11	12
2	10248	42	10
3	10248	72	5
4	10249	14	9
5	10249	51	40

MySQL CASE Examples

The following SQL goes through conditions and returns a value when the first condition is met:

```
SELECT OrderID, Quantity,
CASE
    WHEN Quantity > 30 THEN 'The quantity is greater than 30'
    WHEN Quantity = 30 THEN 'The quantity is 30'
    ELSE 'The quantity is under 30'
END AS QuantityText
FROM OrderDetails;
```

The following SQL will order the customers by City. However, if City is NULL, then order by Country:

```
SELECT CustomerName, City, Country
FROM Customers
ORDER BY
(CASE
    WHEN City IS NULL THEN Country
    ELSE City
END);
```

MySQL IFNULL() and COALESCE() Functions

Look at the following "Products" table:

P_Id	ProductName	UnitPrice	UnitsInStock	UnitsOnOrder
1	Jarlsberg	10.45	16	15
2	Mascarpone	32.56	23	
3	Gorgonzola	15.67	9	20

Suppose that the "UnitsOnOrder" column is optional, and may contain NULL values.
Look at the following SELECT statement:

```
SELECT ProductName, UnitPrice * (UnitsInStock + UnitsOnOrder)
FROM Products;
```

In the example above, if any of the "UnitsOnOrder" values are NULL, the result will be NULL.

MySQL IFNULL() Function:
The MySQL IFNULL() function lets you return an alternative value if an expression is NULL.

The example below returns 0 if the value is NULL:

```
SELECT ProductName, UnitPrice * (UnitsInStock + IFNULL(UnitsOnOrder,
0))
FROM Products;
```

MySQL COALESCE() Function
Or we can use the COALESCE() function, like this:

```
SELECT ProductName, UnitPrice * (UnitsInStock +
COALESCE(UnitsOnOrder, 0))
FROM Products;
```

The MySQL CREATE DATABASE Statement

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

Syntax:

```
CREATE DATABASE databasename;
```

CREATE DATABASE Example

The following SQL statement creates a database called "testDB":

```
CREATE DATABASE testDB;
```

Tip: Make sure you have admin privilege before creating any database. Once a database is created, you can check it in the list of databases with the following SQL command:

```
SHOW DATABASES;
```

The MySQL DROP DATABASE Statement

The DROP DATABASE statement is used to drop an existing SQL database.

Syntax:

```
DROP DATABASE databasename;
```

Note: Be careful before dropping a database. Deleting a database will result in loss of complete information stored in the database!

DROP DATABASE Example
The following SQL statement drops the existing database "testDB":

```
DROP DATABASE testDB;
```

Tip: Make sure you have admin privilege before dropping any database.

The MySQL CREATE TABLE Statement

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

Syntax:

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

The column parameters specify the names of the columns of the table.

The datatype parameter specifies the type of data the column can hold (e.g. varchar, integer, date, etc.).

Tip: For an overview of the available data types, go to our complete Data Types Reference.

MySQL CREATE TABLE Example

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

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

The PersonID column is of type int and will hold an integer.

The LastName, FirstName, Address, and City columns are of type varchar and will hold characters, and the maximum length for these fields is 255 characters.

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

PersonID	LastName	FirstName	Address	City

Tip: The empty "Persons" table can now be filled with data with the SQL INSERT INTO statement.

Create Table Using Another Table

A copy of an existing table can also be created using CREATE TABLE.

The new table gets the same column definitions. All columns or specific columns can be selected.

If you create a new table using an existing table, the new table will be filled with the existing values from the old table.

Syntax:

```
CREATE TABLE new_table_name AS
    SELECT column1, column2,...
    FROM existing_table_name
    WHERE ....;
```

The following SQL creates a new table called "TestTables" (which is a copy of the "Customers" table):

```
CREATE TABLE TestTable AS
SELECT customername, contactname
FROM customers;
```

The MySQL DROP TABLE Statement

The DROP TABLE statement is used to drop an existing table in a database.

```
DROP TABLE table_name;
```

Note: Be careful before dropping a table. Deleting a table will result in loss of complete information stored in the table!

MySQL DROP TABLE Example

The following SQL statement drops the existing table "Shippers":

```
DROP TABLE Shippers;
```

MySQL TRUNCATE TABLE

The TRUNCATE TABLE statement is used to delete the data inside a table, but not the table itself.

```
TRUNCATE TABLE table_name;
```

MySQL ALTER TABLE Statement

The ALTER TABLE statement is used to add, delete, or modify columns in an existing table.

The ALTER TABLE statement is also used to add and drop various constraints on an existing table.

ALTER TABLE - ADD Column
To add a column in a table, use the following syntax:

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

The following SQL adds an "Email" column to the "Customers" table:

```
ALTER TABLE Customers
ADD Email varchar(255);
```

ALTER TABLE - DROP COLUMN

To delete a column in a table, use the following syntax (notice that some database systems don't allow deleting a column):

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

The following SQL deletes the "Email" column from the "Customers" table:

```
ALTER TABLE Customers
DROP COLUMN Email;
```

ALTER TABLE - MODIFY COLUMN

To change the data type of a column in a table, use the following syntax:

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

MySQL ALTER TABLE Example

Look at the "Persons" table:

ID	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

Now we want to add a column named "DateOfBirth" in the "Persons" table.

We use the following SQL statement:

```
ALTER TABLE Persons
ADD DateOfBirth date;
```

Notice that the new column, "DateOfBirth", is of type date and is going to hold a date. The data type specifies what type of data the column can hold. For a complete reference of all the data types available in MySQL, go to our complete Data Types reference.

The "Persons" table will now look like this:

ID	LastName	FirstName	Address	City	DateOfBirth
1	Hansen	Ola	Timoteivn 10	Sandnes	
2	Svendson	Tove	Borgvn 23	Sandnes	
3	Pettersen	Kari	Storgt 20	Stavanger	

Change Data Type Example

Now we want to change the data type of the column named "DateOfBirth" in the "Persons" table.

We use the following SQL statement:

```
ALTER TABLE Persons
MODIFY COLUMN DateOfBirth year;
```

Notice that the "DateOfBirth" column is now of type year and is going to hold a year in a two- or four-digit format.

DROP COLUMN Example

Next, we want to delete the column named "DateOfBirth" in the "Persons" table.

We use the following SQL statement:

```
ALTER TABLE Persons
DROP COLUMN DateOfBirth;
```

The "Persons" table will now look like this:

ID	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

MySQL Dates

The most difficult part when working with dates is to be sure that the format of the date you are trying to insert matches the format of the date column in the database.

As long as your data contains only the date portion, your queries will work as expected. However, if a time portion is involved, it gets more complicated.

MySQL Date Data Types

MySQL comes with the following data types for storing a date or a date/time value in the database:

DATE - format YYYY-MM-DD
DATETIME - format: YYYY-MM-DD HH:MI:SS
TIMESTAMP - format: YYYY-MM-DD HH:MI:SS
YEAR - format YYYY or YY

Note: The date data type is set for a column when you create a new table in your database!

Working with Dates

Look at the following table:

Orders Table

OrderId	ProductName	OrderDate
1	Geitost	2008-11-11
2	Camembert Pierrot	2008-11-09
3	Mozzarella di Giovanni	2008-11-11
4	Mascarpone Fabioli	2008-10-29

Now we want to select the records with an OrderDate of "2008-11-11" from the table above.

We use the following SELECT statement:

```
SELECT * FROM Orders WHERE OrderDate='2008-11-11';
```

The result-set will look like this:

OrderId	ProductName	OrderDate
1	Geitost	2008-11-11
3	Mozzarella di Giovanni	2008-11-11

Note: Two dates can easily be compared if there is no time component involved!

Now, assume that the "Orders" table looks like this (notice the added time-component in the "OrderDate" column):

OrderId	ProductName	OrderDate
1	Geitost	2008-11-11 13:23:44
2	Camembert Pierrot	2008-11-09 15:45:21
3	Mozzarella di Giovanni	2008-11-11 11:12:01
4	Mascarpone Fabioli	2008-10-29 14:56:59

If we use the same SELECT statement as above:

```
SELECT * FROM Orders WHERE OrderDate='2008-11-11';
```

we will get no result! This is because the query is looking only for dates with no time portion.

Tip: To keep your queries simple and easy to maintain, do not use time-components in your dates, unless you have to!

MySQL VIEW

In SQL, a view is a virtual table based on the result-set of an SQL statement.

A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

You can add SQL statements and functions to a view and present the data as if the data were coming from one single table.

A view is created with the CREATE VIEW statement.

CREATE VIEW Syntax:

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

Note: A view always shows up-to-date data! The database engine recreates the view, every time a user queries it.

MySQL CREATE VIEW Examples

The following SQL creates a view that shows all customers from Brazil:

```
CREATE VIEW [Brazil Customers] AS
SELECT CustomerName, ContactName
FROM Customers
WHERE Country = 'Brazil';
```

We can query the view above as follows:

```
SELECT * FROM [Brazil Customers];
```

The following SQL creates a view that selects every product in the "Products" table with a price higher than the average price:

```
CREATE VIEW [Products Above Average Price] AS
SELECT ProductName, Price
FROM Products
WHERE Price > (SELECT AVG(Price) FROM Products);
```

We can query the view above as follows:

```
SELECT * FROM [Products Above Average Price];
```

MySQL Updating a View

A view can be updated with the CREATE OR REPLACE VIEW statement.

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

The following SQL adds the "City" column to the "Brazil Customers" view:

```
CREATE OR REPLACE VIEW [Brazil Customers] AS
SELECT CustomerName, ContactName, City
FROM Customers
WHERE Country = 'Brazil';
```

MySQL Dropping a View

A view is deleted with the DROP VIEW statement.

DROP VIEW Syntax:

```
DROP VIEW view_name;
```

The following SQL drops the "Brazil Customers" view:

```
DROP VIEW [Brazil Customers];
```

SQL Constraints

In a database table, we can add rules to a column known as constraints. These rules control the data that can be stored in a column.

For example, if a column has NOT NULL constraint, it means the column cannot store NULL values.

The constraints used in SQL are as follows:

1. NOT NULL CONSTRAINT:

NOT NULL Constraint Syntax:

```
CREATE TABLE table_name (
  column_name data_type NOT NULL
);
```

Here, table_name is the name of the table to be created
column_name is the name of the column where the constraint is to be implemented
data_type is the data type of the column such as INT, VARCHAR, etc.

Note: The NOT NULL constraint is used to add a constraint to a table column whereas IS NULL and NOT NULL are used with the WHERE clause to select rows from the table.

Remove NOT NULL Constraint

```
ALTER TABLE Colleges
MODIFY college_id INT;
```

NOT NULL Constraint With Alter Table

We can also add the NOT NULL constraint to a column in an existing table using the ALTER TABLE command. For example,

```
ALTER TABLE Colleges
MODIFY COLUMN college_id INT NOT NULL;
```

Here, the SQL command adds the NOT NULL constraint to the college_id column in the existing Colleges table.

Error Due to NOT NULL Constraint

We must enter a value into columns with the NOT NULL constraint. Otherwise, SQL will give us an error.

For example, the college_id column of our Colleges table has the NOT NULL constraint. So, we will get an error if we enter records into the table without supplying a value to college_id.

```
-- gives error due to NOT NULL constraint
INSERT INTO Colleges (college_code, college_name)
VALUES ('NYC', "US");
```

2. SQL UNIQUE CONSTRAINT

Syntax:

```
CREATE TABLE table_name (
  column_name data_type UNIQUE
);
```

Here,table_name is the name of the table to be created
column_name is the name of the column where the constraint is to be implemented
data_type is the data type of the column such as INT, VARCHAR, etc.
Create UNIQUE Constraint
We can implement the UNIQUE constraint at the time of table creation. For example,

```
-- create a table with unique constraint
CREATE TABLE Colleges (
  college_id INT NOT NULL UNIQUE,
  college_code VARCHAR(20) UNIQUE,
  college_name VARCHAR(50)
);
```

```
-- insert values to Colleges table
INSERT INTO Colleges(college_id, college_code, college_name)
VALUES (1, "ARD12", "Star Public School"), (2, "ARD13", "Galaxy
School");
```

Here, both college_id and college_code have the UNIQUE constraint.

The INSERT INTO command runs successfully as we have inserted unique values to college_id and college_code.

UNIQUE Constraint With Alter Table

We can also add the UNIQUE constraint to an existing column using the ALTER TABLE command. For example,

For a Single Column

```
-- add unique constraint to an existing column
ALTER TABLE Colleges
ADD UNIQUE (college_id);
```

Here, the SQL command adds the UNIQUE constraint to the colleges_id column in the existing Colleges table.

For Multiple Columns

```
-- add unique constraint to multiple columns
ALTER TABLE Colleges
ADD UNIQUE Unique_College (college_id, college_code);
```

Here, the SQL command adds the UNIQUE constraint to college_id and college_code columns in the existing Colleges table.

Also, Unique_College is a name given to the UNIQUE constraint defined for college_id and college_code columns.

Note: Our online SQL editor doesn't support this action as it is based on SQLite.

Error When Inserting Duplicate Values

We will get an error if we try to insert duplicate values in a column with the UNIQUE constraint.

```
-- create a table named colleges
CREATE TABLE Colleges (
  college_id INT NOT NULL UNIQUE,
  college_code VARCHAR(20) UNIQUE,
  college_name VARCHAR(50)
);
```

```
-- insert values to Colleges table
-- college_code has duplicate values
INSERT INTO Colleges(college_id, college_code, college_name)
VALUES (1, "ARD12", "Star Public School"), (2, "ARD12", "Galaxy
School");
```

Here, we are trying to insert ARD12 into the college_code column in two different rows. Hence, the INSERT INTO command results in an error.

CREATE UNIQUE INDEX for Unique Values

If we want to create indexes for unique values in a column, we use the CREATE UNIQUE INDEX constraint. For example,

```
-- create unique index
CREATE UNIQUE INDEX college_index
ON Colleges(college_code);
```

Here, the SQL command creates a unique index named college_index on the Colleges table using the college_code column.

Note: Creating an index does not alter the original data in the table.

3. PRIMARY KEY CONSTRAINT

SQL PRIMARY KEY Syntax:

```
CREATE TABLE table_name (
  column1 data_type,
  ...,
  [CONSTRAINT constraint_name] PRIMARY KEY (column1)
);
```

Here, table_name is the name of the table to be created
column1 is the name of the column where the PRIMARY KEY constraint is to be defined constraint_name is the arbitrary name given to the constraint
[...] signifies that the code inside is optional.
Note: Although naming a constraint using [CONSTRAINT constraint_name] is optional, doing so makes it easier to make changes to and delete the constraint.

Primary Key Error:

In SQL, we will get an error If we try to insert NULL or duplicate values in the primary key column.

NOT NULL Constraint Error

We get this error when we supply the primary key with a NULL value. This is because primary keys need to obey the NOT NULL constraint. For example,

```
-- NOT NULL Constraint Error
-- the value of primary key (college_id) is NULL
INSERT INTO Colleges(college_id, college_code, college_name)
VALUES ("ARD12", "Star Public School");
```

Here, the SQL command gives us an error because we have supplied a NULL value to the primary key college_id in the Colleges table.

Fix the NOT NULL Constraint Error

```
-- Insertion Success
-- the value of primary key (college_id) is 1
INSERT INTO Colleges(college_id, college_code, college_name)
VALUES (1, "ARD12", "Star Public School");
```

Here, the SQL command runs without errors because we have supplied the value 1 to the primary key i.e. college_id.

UNIQUE Constraint Error

We get this error when we supply duplicate values to the primary key, which violates its UNIQUE constraint. For example,

```
INSERT INTO Colleges(college_id, college_code, college_name)
VALUES (1, "ARD12", "Star Public School");
```

```
-- UNIQUE Constraint Error
-- the value of college_id is not unique
INSERT INTO Colleges(college_id, college_code, college_name)
VALUES (1, "ARD12", "Star Public School");
```

Here, the SQL command gives us an error because we have inserted the duplicate value 1 into the primary key college_id.

Fix the UNIQUE Constraint Error

```
INSERT INTO Colleges(college_id, college_code, college_name)
VALUES (1, "ARD12", "Star Public School");
```

```
-- Insertion Success
INSERT INTO Colleges(college_id, college_code, college_name)
VALUES (2, "ARD12", "Star Public School");
```

Here, the SQL command runs without errors because we have supplied unique values 1 and 2 to college_id.

Note: There can only be one primary key in a table. However, that single primary key can contain multiple columns.

Primary Key With Multiple Columns

A primary key may also be made up of multiple columns. For example,

```
CREATE TABLE Colleges (
  college_id INT,
  college_code VARCHAR(20),
  college_name VARCHAR(50),
  CONSTRAINT CollegePK PRIMARY KEY (college_id, college_code)
);
```

Here, the PRIMARY KEY constraint named CollegePK is made up of the college_id and college_code columns.

This means that the combination of college_id and college_code must be unique and these two columns cannot contain NULL values.

Primary Key Constraint With Alter Table

We can also add the PRIMARY KEY constraint to a column in an existing table using the ALTER TABLE command. For example,

For a Single Column

```
ALTER TABLE Colleges
ADD PRIMARY KEY (college_id);
```

Here, the SQL command adds the PRIMARY KEY constraint to the college_id column in the existing Colleges table.

For Multiple Columns

```
ALTER TABLE Colleges
ADD CONSTRAINT CollegePK PRIMARY KEY (college_id, college_code);
```

Here, the SQL command adds the PRIMARY KEY constraint to the college_id and college_id columns in the existing Colleges table.

Note: This command is not supported by our online SQL editor as it is based on SQLite.

Auto Increment Primary Key

```
-- AUTO_INCREMENT keyword auto increments the value
CREATE TABLE Colleges (
  college_id INT AUTO_INCREMENT,
  college_code VARCHAR(20) NOT NULL,
  college_name VARCHAR(50),
  CONSTRAINT CollegePK PRIMARY KEY (college_id)
);

-- insert record without college_id
INSERT INTO Colleges(college_code, college_name)
VALUES ("ARD13", "Star Public School");
PostgreSQL

-- SERIAL keyword auto increments the value
CREATE TABLE Colleges (
  college_id INT SERIAL,
  college_code VARCHAR(20) NOT NULL,
  college_name VARCHAR(50),
  CONSTRAINT CollegePK PRIMARY KEY (college_id)
);

-- insert record without college_id
INSERT INTO Colleges(college_code, college_name)
VALUES ("ARD13", "Star Public School");
```

Remove Primary Key Constraint

We can remove the PRIMARY KEY constraint in a table using the DROP clause. For example,

```
ALTER TABLE Colleges
DROP PRIMARY KEY;
```

Here, the SQL command removes the PRIMARY KEY constraint from the Colleges table.

4. FOREIGN KEY CONSTRAINT:

Foreign Key Syntax in SQL
The syntax of the SQL FOREIGN KEY constraint is:

```
CREATE TABLE table_name (
    column1 data_type,
    column2 data_type,
    ...,
    FOREIGN KEY (column_name)
    REFERENCES referenced_table_name (referenced_column_name)
);
```

Here, table_name is the name of the table where the FOREIGN KEY constraint is to be defined
column_name is the name of the column where the FOREIGN KEY constraint is to be defined
referenced_table_name and referenced_column_name are the names of the table and the column that the FOREIGN KEY constraint references

Referencing Columns in Another Table with FOREIGN KEY

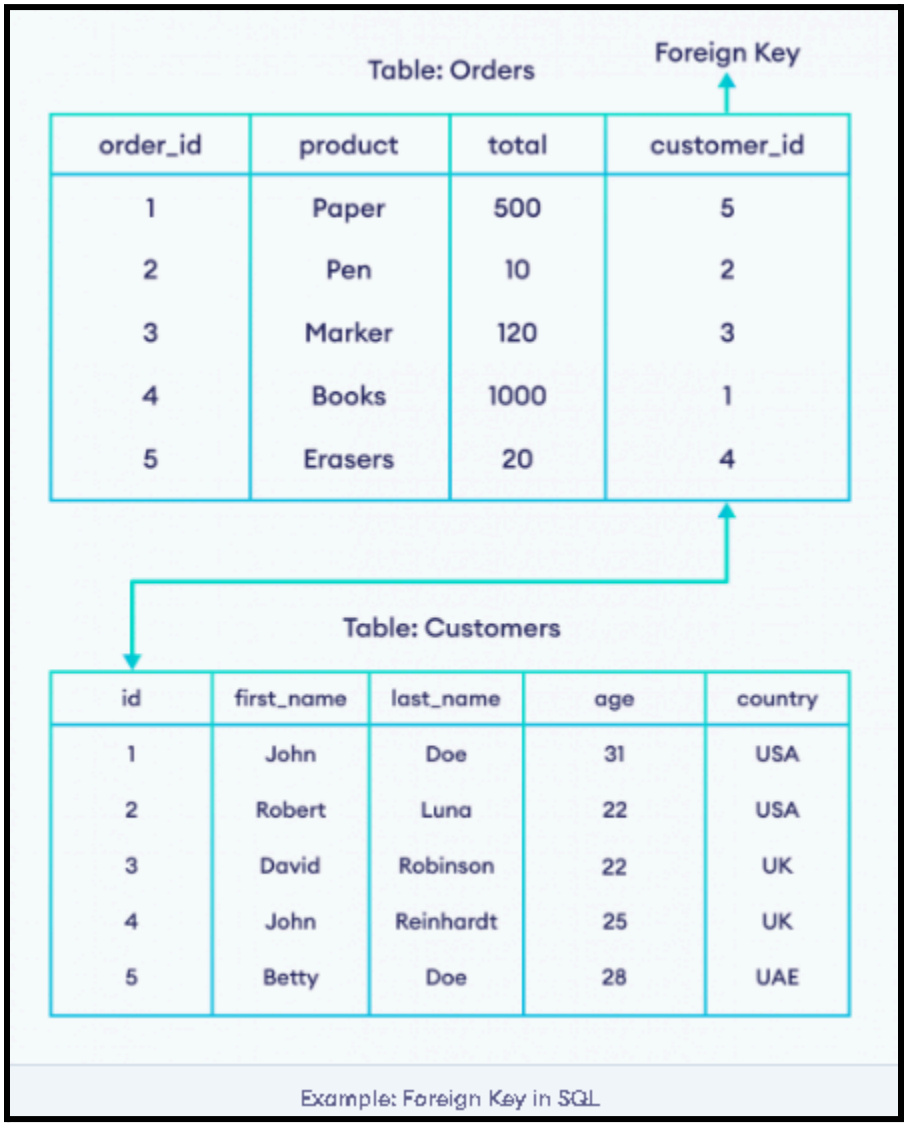
The FOREIGN KEY constraint in SQL establishes a relationship between two tables by linking columns in one table to those in another. For example,

(Please refer the diagram below before reading further)

Here, the customer_id field in the Orders table is a FOREIGN KEY that references the customer_id field in the Customers table.

This means that the value of the customer_id (of the Orders table) must be a value from the customer_id column (of the Customers table).

Note: The foreign key can be referenced to any column in the parent table. However, it is a general practice to reference the foreign key to the primary key of the parent table.



Creating Foreign Key

Now, let's see how we can create foreign key constraints in a database.

```
-- this table doesn't have a foreign key
CREATE TABLE Customers (
  id INT,
  first_name VARCHAR(40),
  last_name VARCHAR(40),
  age INT,
  country VARCHAR(10),
```

```
  CONSTRAINT CustomersPK PRIMARY KEY (id)
);

-- add foreign key to the customer_id field
-- the foreign key references the id field of the Customers table
CREATE TABLE Orders (
  order_id INT,
  product VARCHAR(40),
  total INT,
  customer_id INT,
  CONSTRAINT OrdersPK PRIMARY KEY (order_id),
  FOREIGN KEY (customer_id) REFERENCES Customers(id)
);
```

Here, the value of the customer_id column in the Orders table references the row in another table named Customers with its id column.

Note: The above code works in all major database systems. However, there may be an alternate syntax to create foreign keys depending on the database. Refer to their respective database documentation for more information.

Inserting Records in Table With Foreign Key

```
-- insert record into table with no foreign key first
INSERT INTO Customers
VALUES
(1, 'John', 'Doe', 31, 'USA'),
(2, 'Robert', 'Luna', 22, 'USA');

-- insert record into table with foreign key constraint in
customer_id column
-- Insertion Success
INSERT INTO Orders
VALUES
(1, 'Keyboard', 400, 2),
(2, 'Mouse', 300, 2),
(3, 'Monitor', 12000, 1);
```

Here, the query is successfully sql-executed as the rows we are trying to insert in the Orders table have valid values in the customer_id column, which has a FOREIGN KEY constraint in the Customers table.

Insertion Failure in Foreign Key

An insertion failure occurs when a value is entered into a table's foreign key column that does not match any value in the primary key column of the related table. For example,

```
-- insert record into table with no foreign key first
INSERT INTO Customers
VALUES
(1, 'John', 'Doe', 31, 'USA'),
(2, 'Robert', 'Luna', 22, 'USA');

-- insert record into table with a foreign key
-- insertion error because customer with id of 7 does not exist
INSERT INTO Orders
VALUES (4, 'Keyboard', 400, 7);
```

Here, the insertion of rows into the Orders table fails because 7 is not a valid customer_id value in the Customers table. Thus, this query fails the FOREIGN KEY constraint.

Why use Foreign Key?

Foreign keys are an important part of relational databases, and we use them for the following reasons:

To Normalize Data

The FOREIGN KEY constraint helps us to normalize data in multiple tables and reduce redundancy. This means that a database can have multiple tables that are related to each other.

Prevent Wrong Data From Insertion

If two database tables are related through a field (attribute), using FOREIGN KEY makes sure that wrong data is not inserted in that field. This helps eliminate bugs at the database level.

Foreign Key With Alter Table

It is possible to add the FOREIGN KEY constraint to an existing table using the ALTER TABLE command. For example,

```
CREATE TABLE Customers (
  id INT,
  first_name VARCHAR(40),
  last_name VARCHAR(40),
  age INT,
  country VARCHAR(10),
  CONSTRAINT CustomersPK PRIMARY KEY (id)
);

CREATE TABLE Orders (
  order_id INT,
  item VARCHAR(40),
  amount INT,
  customer_id INT,
  CONSTRAINT OrdersPK PRIMARY KEY (order_id)
);

-- add foreign key to the customer_id field of Orders
-- the foreign key references the id field of Customers
ALTER TABLE Orders
ADD FOREIGN KEY (customer_id) REFERENCES Customers(id);
```

Note: This action is not supported on our online SQL editor as it is based on SQLite.

Multiple Foreign Keys in a Table

A database table can also have multiple foreign keys.

For instance, let's say that we need to record all transactions where each user is a buyer and a seller.

```
-- this table doesn't have a foreign key
CREATE TABLE Users (
  id INT PRIMARY KEY,
  first_name VARCHAR(40),
  last_name VARCHAR(40),
  age INT,
  country VARCHAR(10)
);

-- add foreign key to buyer and seller fields
-- foreign key references the id field of the Users table
CREATE TABLE Transactions (
  transaction_id INT PRIMARY KEY,
  amount INT,
  seller INT,
  buyer INT,
  CONSTRAINT fk_seller FOREIGN KEY (seller) REFERENCES Users(id),
  CONSTRAINT fk_buyer FOREIGN KEY (buyer) REFERENCES Users(id)
);
```

Here, the SQL command creates two foreign keys (buyer and seller) in the Transactions table.

Note: As with other constraints, naming a FOREIGN KEY constraint using CONSTRAINT constraint_name is optional. But doing so makes it easier to make changes to or delete the constraint. This is especially helpful when defining multiple constraints.

5. CHECK CONSTRAINT:

CHECK Constraint Syntax
The syntax of the SQL CHECK constraint is:

```
CREATE TABLE table_name (
  column_name data_type CHECK(condition)
);
```

Here, table_name is the name of the table to be created
column_name is the name of the column where the constraint is to be implemented
data_type is the data type of the column such as INT, VARCHAR, etc.
condition is the condition that needs to be checked
Note: The CHECK constraint is used to validate data while insertion only. To check if the row exists or not, visit SQL EXISTS.

Example 1: SQL CHECK Constraint Success

```
-- apply the CHECK constraint to the amount column
CREATE TABLE Orders (
  order_id INT PRIMARY KEY,
  amount INT CHECK (amount > 0)
);

-- amount equal to 100
-- record is inserted
INSERT INTO Orders(amount) VALUES(100);
```

Here, we have created a table named Orders with a CHECK constraint that requires the amount value to be greater than 0.

When trying to insert a record with an amount value of 100, the insertion process was successful because the value satisfies the CHECK constraint condition.

Example 2: SQL CHECK Constraint Failure

```
-- apply the CHECK constraint to the amount column
CREATE TABLE Orders (
  order_id INT PRIMARY KEY,
  amount INT CHECK (amount > 0)
);

-- amount equal to -5
-- results in an error
INSERT INTO Orders(amount) VALUES(-5);
```

Here, we have created a table named Orders with a CHECK constraint that requires the amount value to be greater than 0.

When trying to insert a record with an amount value of -5, the insertion process failed because the value doesn't satisfy the CHECK constraint condition.

Create Named CHECK Constraint

It's a good practice to create named constraints so that it is easier to alter and drop constraints.

Here's an example to create a named CHECK constraint:

```
-- create a named constraint named amountCK
-- the constraint makes sure that amount is greater than 0
CREATE TABLE Orders (
  order_id INT PRIMARY KEY,
  amount INT,
  CONSTRAINT amountCK CHECK (amount > 0)
);
```

Here, amountCK is the name given to the CHECK constraint.

CHECK Constraint in Existing Table

We can add the CHECK constraint to an existing table by using the ALTER TABLE clause. For example, let's add the CHECK constraint to the amount column of an existing Orders table.

```
-- add CHECK constraint without name
ALTER TABLE Orders
ADD CHECK (amount > 0);
```

Here's how we can add a named CHECK constraint. For example,

```
-- add CHECK constraint named amountCK
ALTER TABLE Orders
ADD CONSTRAINT amountCK CHECK (amount > 0);
```

Notes: If we try to add the CHECK constraint amount > 0 to a column that already has value less than 0, we will get an error.
The ALTER TABLE command is not supported by our online SQL editor since it is based on SQLite.

Remove CHECK Constraint

We can remove the CHECK constraint using the DROP clause. For example,

```
-- remove CHECK constraint named amountCK
ALTER TABLE Orders
DROP CHECK amountCK;
```

6. DEFAULT CONSTRAINT:

The syntax of the SQL DEFAULT constraint is:

```
CREATE TABLE table_name (
  column_name data_type DEFAULT default_value
);
```

Here, table_name is the name of the table to be created
column_name is the name of the column where the constraint is to be implemented
data_type is the data type of the column such as INT, VARCHAR, etc.
default_value is the value that the inserted empty values are replaced with

Example: SQL DEFAULT Constraint

```
-- don't add any value to college_country column
-- thus default value 'US ' is inserted to the column
INSERT INTO Colleges (college_id, college_code)
VALUES (1, 'ARP76');

-- insert 'UAE' to the college_country column
INSERT INTO Colleges (college_id, college_code, college_country)
VALUES (2, 'JWS89', 'UAE');
```

Here, the default value of the college_country column is set to US. So, when we try to insert a NULL value to the college_country column, it is replaced with US by default.

But when we set college_country to UAE, the default value is ignored and the value of the column is set as UAE.

DEFAULT Constraint With Alter Table

We can also add the DEFAULT constraint to an existing column using the ALTER TABLE command. For example,

```
ALTER TABLE College
ALTER college_country SET DEFAULT 'US';
```

Here, the default value of the college_country column is set to US if NULL is passed during insertion.

Remove Default Constraint

We can use the DROP clause to remove the DEFAULT constraint in a column. For example,

```
ALTER TABLE College
ALTER college_country DROP DEFAULT;
```

Here, the SQL command removes the DEFAULT constraint from the college_country column.

7. CREATE INDEX CONSTRAINT:

The syntax of the SQL CREATE INDEX statement is:

```
CREATE INDEX index_name
ON table_name (column_name1, column_name2, ...);
```

Here, index_name is the name given to the index
table_name is the name of the table on which the index is to be created
column_name1, column_name2, ... are the names of the columns to be included in the index

CREATE UNIQUE INDEX for Unique Values

If you want to create indexes for unique values in a column, we use the CREATE UNIQUE INDEX constraint. For example,

```
-- create table
CREATE TABLE Colleges (
  college_id INT PRIMARY KEY,
  college_code VARCHAR(20) NOT NULL,
  college_name VARCHAR(50)
);

-- create unique index
CREATE UNIQUE INDEX college_index
ON Colleges(college_code);
```

Here, the SQL command creates a unique index named college_index on the Colleges table using the college_code column.

Note: Although the index is created for only unique values, the original data in the table remains unaltered.

Remove Index From Tables

To remove INDEX from a table, we can use the DROP INDEX command. For example,

```
ALTER TABLE Colleges
DROP INDEX college_index;
```

Here, the SQL command removes an index named college_index from the Colleges table.

Note: Deleting an index in SQL means only the index is deleted. The data in the original table remains unaltered.

8. COMPOSITE KEY CONSTRAINT:

A composite key is a unique identifier for each row in a table and is formed by combining two or more columns in a table.

Example:

```
CREATE TABLE CustomerOrderShippings (
    customer_id INT,
    order_id INT,
    shipping_id INT,
    PRIMARY KEY (customer_id, order_id, shipping_id)
);
```

Here, customer_id, order_id, and shipping_idtogether form a composite primary key of the CustomerOrderShippings table.

Using Composite Keys in Relationships

Composite keys are often used to create relationships between tables in a database. For example,

```
CREATE TABLE OrderShippings (
    order_id INT,
    shipping_id INT,
    PRIMARY KEY (order_id, shipping_id)
);
```

In this example, the composite key consists of the order_id and shipping_id columns, uniquely identifying each relationship between an order and its shipping details.

Note: Composite keys are particularly useful when a single column does not contain enough unique data to serve as a primary key.

Composite Key With Foreign Keys

Composite keys can also be used in conjunction with foreign keys to enforce referential integrity in a database.

Let's take a look at an example.

```
CREATE TABLE OrderDetails (
    customer_id INT,
    order_id INT,
    item_name VARCHAR(100),
    PRIMARY KEY (customer_id, order_id),
    FOREIGN KEY (customer_id) REFERENCES Customers(customer_id),
    FOREIGN KEY (order_id) REFERENCES Orders(order_id)
);
```

Here, in the OrderDetails table, the combination of customer_id and order_id serves two roles:

Composite Primary Key: Together, they uniquely identify each record in the OrderDetails table, ensuring no two records are the same.
Composite Foreign Key: customer_id links to the Customers table and order_id links to the Orders table.
This setup links order details to both specific customers and specific orders.

Inserting Records With Composite Keys

Inserting records into a table (see SQL INSERT INTO) with a composite key is similar to inserting into any other table. For example,

```
-- create table with composite keys
CREATE TABLE CustomerOrderShippings (
    customer_id INT,
    order_id INT,
    shipping_id INT,
    PRIMARY KEY (customer_id, order_id, shipping_id)
);

-- insert into the table
```

```
INSERT INTO CustomerOrderShippings (customer_id, order_id,
shipping_id) VALUES
(1, 4, 5),
(4, 2, 2);
```

Here, the SQL command inserts two records into the CustomerOrderShippings table. Each record is a combination of customer_id, order_id, and shipping_id:
(1, 4, 5) - represents a record where customer_id is 1, order_id is 4, and shipping_id is 5.
(4, 2, 2) - represents another record where customer_id is 4, order_id is 2, and shipping_id is 2.
These combinations form composite keys that uniquely identify each row in the table.

SQL Data Types - MySQL Data Types (w3schools.com)

SQL Functions - MySQL Functions (w3schools.com)

XXXXXXXXXXXXXXXXXXXXXXX END XXXXXXXXXXXXXXXXXXXXXXX

TOPIC 12: JENKINS, DOCKER, KUBERNETES

TOPIC 13: FINANCE CALCULATOR PROJECT

TOPIC 14: OS BASICS

TOPIC 15: CN BASICS

TOPIC 16: SEN BASICS (AGILE)