- Structured Query Language (SQL)
- SQL is a standard language for storing, manipulating and retrieving data in databases.
- With help of SQL, you can C**RUD** operations
  - Select: **R**etrieve Date,
  - Insert: **C**reate Data
  - Update: **U**pdate Data
  - Delete: **D**elete Data

- Database is collection of tables.
- SQL is not case sensitive.
- Database Table Schema:
- Column data types.
- Use Semicolon (;) at end of each query.
- **Primary key?**
  - PK is a column in the Table that uniquely identifies each record and the value is never duplicated in the same table
  - PK cannot contain NULL Values
- **Foreign key?**
  - Is existence of PK in another table
  - Is a key used to link two tables together
  - It can accept Null Values
  - We can have more than one Foreign Key in a table.

SELECT

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

SELECT DISTINCT

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

- The following SQL statement lists the number of different (distinct) customer countries:

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

WHERE Clause

- is used to filter records.
- The WHERE clause is used to extract only those records that fulfill a specified condition.

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

- The WHERE clause is not only used in SELECT statement, it is also used in UPDATE, DELETE statement, etc.!
- Where Clause with texts (Data stored in Tables are case sensetive)

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

- Where clause for Numeric fileds

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

Operators in The WHERE Clause

=	Equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
<>	Not equal. This operator may be written as !=
BETWEEN	Between a certain range
LIKE	Search for a pattern
IN	To specify multiple possible values for a column

The SQL AND, OR and NOT Operators

- The WHERE clause can be combined with AND, OR, and NOT operators.
- The AND and OR operators are used to filter records based on more than one condition:
- The AND operator displays a record if all the conditions separated by AND are TRUE.
- The OR operator displays a record if any of the conditions separated by OR is TRUE.
- The NOT operator displays a record if the condition(s) is NOT TRUE.
- We normally don't use Where clause after another where clause, instead we use AND.
- In SQL, you combine all of "AND"s together. Order of AND OR AND is interrupting.

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

- **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;
```

- **Examples:**
  - 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 following SQL statement selects all fields from "Customers" where country is "Germany" AND city must be "Berlin" OR "München" (use parenthesis to form complex expressions):

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

The SQL 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;
```

- **Examples**:
  - **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:** 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 NOT CategoryID 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;
```

  - **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;
```

  - **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';
```

SQL 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);
```

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

```
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":

```
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:

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

The SQL 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.
- ORDER BY Syntax

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

- **Examples**
  - 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;
```

  - 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;
```

SQL Aliases

- SQL 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 the query.
- 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;
```

- **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:** It requires double quotation marks or square brackets 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, Address + ', ' + PostalCode + ' ' + City + ', ' + Country AS Address
FROM Customers;
```

SQL NULL Values

- A field with a NULL value is a field with no value.
- NULL value cannot be compared with other NULL values. Hence, It is not possible to test for NULL values with comparison operators, such as =, <, or <>.
- 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.
- 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!
- 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;
```

- **Examples:**
  - **IS NULL Operator:** The following SQL lists all customers with a NULL value in the "Address" field:

```
FROM Customers
WHERE Address IS NULL;
```

  - **IS NOT NULL Operator**: The following SQL lists all customers with a value in the "Address" field:

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

SQL 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.
- **Syntax**
  - MIN() Syntax

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

  - MAX() Syntax

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

- **Examples**:
  - **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;
```

The SQL COUNT(), AVG() and SUM() Functions

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

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

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

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

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

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

- **Examples:**
  - **COUNT() Example**: The following SQL statement finds the number of products. **Note:** NULL values are not counted.

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

  - **AVG() Example**: The following SQL statement finds the average price of all products. **Note:** NULL values are ignored.

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

  - **SUM() Example**: The following SQL statement finds the sum of the "Quantity" fields in the "OrderDetails" table. **Note:** NULL values are ignored.

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

SQL 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 represents a 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;
```

- Here are some examples showing different LIKE operators with '%' and '_' wildcards:
  - 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__%' 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"
- **Examples**
  - 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%';
```

SQL Comments

- Comments are used to explain sections of SQL statements, or to prevent execution of SQL statements.
- **Single Line Comments**
  - Single line comments start with --.
  - Any text between -- and the end of the line will be ignored (will not be executed).
  - The following example uses a single-line comment as an explanation:

```
--Select all:
SELECT * FROM Customers;
```

  - The following example uses a single-line comment to ignore the end of a line:

```
SELECT * FROM Customers -- WHERE City='Berlin';
```

  - The following example uses a single-line comment to ignore a statement:

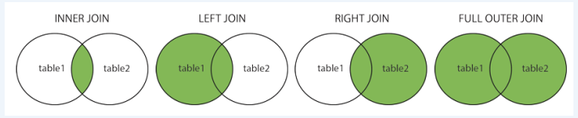
```
--SELECT * FROM Customers;
SELECT * FROM Products;
```

- **Multi-line Comments**
  - Multi-line comments start with /* and end with */.
  - Any text between /* and */ will be ignored.
  - The following example uses a multi-line comment as an explanation:

```
/*Select all the columns
of all the records
in the Customers table:*/
SELECT * FROM Customers;
```

SQL JOIN

- A JOIN clause is used to combine rows from two or more tables, based on a related column between them.
- different types of the JOINs in SQL:
  - **(INNER) JOIN**: Returns records that have matching values in both tables
  - **LEFT (OUTER) JOIN**: Returns all records from the left table, and the matched records from the right table
  - **RIGHT (OUTER) JOIN**: Returns all records from the right table, and the matched records from the left table
  - **FULL (OUTER) JOIN**: Returns all records when there is a match in either left or right table



SQL INNER JOIN Keyword

- The INNER JOIN keyword selects records that have matching values in both tables.
- **INNER JOIN Syntax**

```
SELECT column_name(s)
FROM table1
INNER JOIN table2
ON table1.column_name = table2.column_name;
```

- **Example**: The following SQL statement selects all orders with customer information:

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

- **JOIN Three Tables**
  - The following SQL statement selects all orders with customer and shipper information:

```
SELECT Orders.OrderID, Customers.CustomerName, Shippers.ShipperName
FROM ((Orders
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID)
INNER JOIN Shippers ON Orders.ShipperID = Shippers.ShipperID);
```

SQL LEFT JOIN Keyword

- The LEFT JOIN keyword returns all records from the left table (table1), and the matched records from the right table (table2). The result is NULL from the right side, if there is no match.
- In some databases LEFT JOIN is called LEFT OUTER JOIN.
- **LEFT JOIN Syntax**

```
SELECT column_name(s)
FROM table1
LEFT JOIN table2
ON table1.column_name = table2.column_name;
```

- **Example:** The following SQL statement will select all customers, and any orders they might have:

```
SELECT Customers.CustomerName, Orders.OrderID
FROM Customers
LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID
ORDER BY Customers.CustomerName;
```

- **Note:** The LEFT JOIN keyword returns all records from the left table (Customers), even if there are no matches in the right table (Orders).

SQL RIGHT JOIN Keyword

- The RIGHT JOIN keyword returns all records from the right table (table2), and the matched records from the left table (table1). The result is NULL from the left side, when there is no match.
- **Note:** In some databases RIGHT JOIN is called RIGHT OUTER JOIN
- **RIGHT JOIN Syntax**

```
SELECT column_name(s)
FROM table1
RIGHT JOIN table2
ON table1.column_name = table2.column_name;
```

- **Example**: The following SQL statement will return all employees, and any orders they might have placed:

```
SELECT Orders.OrderID, Employees.LastName, Employees.FirstName
FROM Orders
RIGHT JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID
ORDER BY Orders.OrderID;
```

- **Note:** The RIGHT JOIN keyword returns all records from the right table (Employees), even if there are no matches in the left table (Orders).

SQL FULL OUTER JOIN Keyword

- The FULL OUTER JOIN keyword returns all records when there is a match in left (table1) or right (table2) table records.
- **Note:** FULL OUTER JOIN can potentially return very large result-sets!
- **Tip:** FULL OUTER JOIN and FULL JOIN are the same.
- **Syntax**

```
SELECT column_name(s)
FROM table1
FULL OUTER JOIN table2
ON table1.column_name = table2.column_name
WHERE condition;
```

- **Example**: The following SQL statement selects all customers, and all orders:

```
SELECT Customers.CustomerName, Orders.OrderID
FROM Customers
FULL OUTER JOIN Orders ON Customers.CustomerID=Orders.CustomerID
ORDER BY Customers.CustomerName;
```

- **Note:** The FULL OUTER JOIN keyword returns all matching records from both tables whether the other table matches or not. So, if there are rows in "Customers" that do not have matches in "Orders", or if there are rows in "Orders" that do not have matches in "Customers", those rows will be listed as well.

The SQL UNION Operator

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

- Each SELECT statement within UNION must have the same number of columns
- The columns must also have similar data types
- The columns in each SELECT statement must also be in the same order
- **Syntax**
  - **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: **Note:** The column names in the result-set are usually equal to the column names in the first SELECT statement in the UNION.

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

- **Examples**
  - **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!

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

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

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

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

  - **UNION ALL With WHERE**: The following SQL statement returns the German cities (duplicate values also) from both the "Customers" and the "Suppliers" table:

```
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:

```
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".