What Can SQL do?

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

RDBMS

RDBMS stands for Relational Database Management System.

RDBMS is the basis for SQL, and for all modern database systems such as MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access.

The data in RDBMS is stored in database objects called tables. A table is a collection of related data entries and it consists of columns and rows.

- `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 SQL SELECT DISTINCT Statement

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

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

Count Distinct

By using the `DISTINCT` keyword in a function called `COUNT`, we can return the number of different countries.

The SQL WHERE Clause

The `WHERE` clause is used to filter records.

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

Syntax

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

The SQL ORDER BY

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

Syntax

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

Ex:

```
SELECT * FROM Products
ORDER BY Price DESC;
```

Order Alphabetically

For string values the `ORDER BY` keyword will order alphabetically:

ORDER BY Several Columns

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:

Example

```
SELECT * FROM Customers

ORDER BY Country, CustomerName
```

Using Both ASC and DESC

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

Example

```
SELECT * FROM Customers

ORDER BY Country ASC, CustomerName DESC;
```

The SQL AND Operator

The `WHERE` clause can contain one or many `AND` operators.

Syntax

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

AND vs OR

The `AND` operator displays a record if *all* the conditions are TRUE.

The `OR` operator displays a record if *any* of the conditions are TRUE.

The SQL OR Operator

The `WHERE` clause can contain one or more `OR` operators.

The `OR` operator is used to filter records based on more than one condition, like if you want to return all customers from Germany but also those from Spain:

Syntax

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

The NOT Operator

The NOT operator is used in combination with other operators to give the opposite result, also called the negative result.

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

Syntax

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

NOT LIKE

Example

Select customers that does not start with the letter 'A':

```
SELECT * FROM Customers

WHERE CustomerName NOT LIKE 'A%';
```

NOT BETWEEN

Example

Select customers with a customerID not between 10 and 60:

```
SELECT * FROM Customers

WHERE CustomerID NOT BETWEEN 10 AND 60;
```

The SQL 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, ...);
```

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):

Example

```
INSERT INTO Customers (CustomerName, City, Country)

VALUES ('Cardinal', 'Stavanger', 'Norway');
```

Insert Multiple Rows

It is also possible to insert multiple rows in one statement.

To insert multiple rows of data, we use the same `INSERT INTO` statement, but with multiple values:

Example

```
INSERT INTO Customers (CustomerName, ContactName, Address, City,
PostalCode, Country)


VALUES


('Cardinal', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger', '4006',
'Norway'),


('Greasy Burger', 'Per Olsen', 'Gateveien 15', 'Sandnes', '4306',
'Norway'),


('Tasty Tee', 'Finn Egan', 'Streetroad 19B', 'Liverpool', 'L1 0AA',
'UK')
```

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

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;
```

The SQL 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;
```

Note: Be careful when updating records in a table! Notice the `WHERE` clause in the `UPDATE` statement. The `WHERE` clause specifies which record(s) that should be updated. If you omit the `WHERE` clause, all records in the table will be updated!

The SQL 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!

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;
```

Delete a Table

To delete the table completely, use the `DROP TABLE` statement:

Example

Remove the Customers table:

```
DROP TABLE Customers;
```

The SQL SELECT TOP Clause

The `SELECT TOP` clause is used to specify the number of records to return.

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

Example

Select only the first 3 records of the Customers table:

```
SELECT TOP 3 * FROM Customers;
```

SQL Aggregate Functions

An aggregate function is a function that performs a calculation on a set of values, and returns a single value.

Aggregate functions are often used with the `GROUP BY` clause of the `SELECT` statement. The `GROUP BY` clause splits the result-set into groups of values and the aggregate function can be used to return a single value for each group.

The most commonly used SQL aggregate functions are:

- `MIN()` - returns the smallest value within the selected column
- `MAX()` - returns the largest value within the selected column
- `COUNT()` - returns the number of rows in a set
- `SUM()` - returns the total sum of a numerical column
- `AVG()` - returns the average value of a numerical column

Aggregate functions ignore null values (except for `COUNT()`).

The 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 sign `_` represents one, single character

Example

Get your own SQL Server

Select all customers that starts with the letter "a":

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

The 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.

Example

Get your own SQL Server

Return all customers from 'Germany', 'France', or 'UK'

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

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.

Example

Selects all products with a price between 10 and 20:

```
SELECT * FROM Products

WHERE Price BETWEEN 10 AND 20;
```

NOT BETWEEN

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

Example

```
SELECT * FROM Products

WHERE Price NOT BETWEEN 10 AND 20;
```

BETWEEN Text Values

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

Example

```
SELECT * FROM Products

WHERE ProductName BETWEEN 'Carnarvon Tigers' AND 'Mozzarella di
Giovanni'

ORDER BY ProductName;
```

BETWEEN Dates

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

Example

```
SELECT * FROM Orders

WHERE OrderDate BETWEEN #07/01/1996# AND #07/31/1996#;
```

OR:

Example

```
SELECT * FROM Orders

WHERE OrderDate BETWEEN '1996-07-01' AND '1996-07-31';
```

The 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.

Example

Get your own SQL Server

Return all customers from 'Germany', 'France', or 'UK'

```
SELECT * FROM Customers

WHERE Country IN ('Germany', 'France', 'UK');
```

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.

Example

Get your own SQL Server

Selects all products with a price between 10 and 20:

```
SELECT * FROM Products

WHERE Price BETWEEN 10 AND 20;
```

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 that query.

An alias is created with the `AS` keyword.

Example

```
SELECT CustomerID AS ID

FROM Customer
```

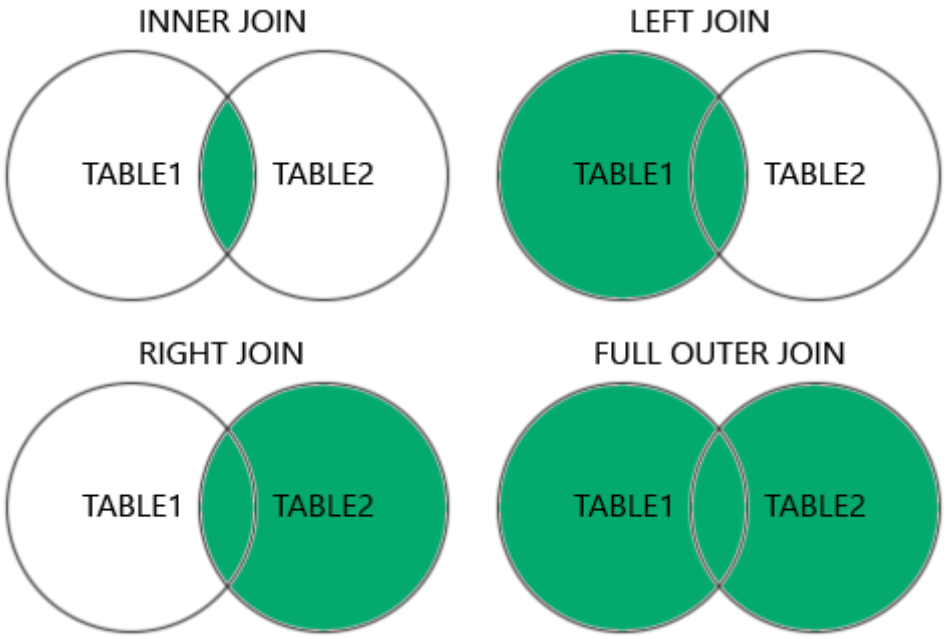
SQL JOIN

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

Different Types of SQL JOINs

Here are the 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



Example

Get your own SQL Server

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

The SQL 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;
```

The SQL 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.

Example

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

The SQL HAVING Clause

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

```
SELECT COUNT(CustomerID), Country

FROM Customers

GROUP BY Country

HAVING COUNT(CustomerID) >5
```

The SQL 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.

Example

```
SELECT SupplierName

FROM Suppliers

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

The SQL 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 SQL ANY Operator

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);
```

The SQL ALL Operator

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);
```

The SQL SELECT INTO Statement

The `SELECT INTO` statement copies data from one table into a new table.

SELECT INTO Syntax

Copy all columns into a new table:

```
SELECT *

INTO newtable [IN externaldb]

FROM oldtable

WHERE condition;
```

SQL SELECT INTO Examples

The following SQL statement creates a backup copy of Customers:

```
SELECT * INTO CustomersBackup2017

FROM Customers;

SELECT * INTO CustomersGermany

FROM Customers

WHERE Country = 'Germany';
```

The SQL 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;
```

Example

Copy only the German suppliers into "Customers":

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

The SQL CASE Expression

The `CASE` expression 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;
```

Example

Get your own SQL Server

```
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;
```

SQL IFNULL(), ISNULL(), COALESCE(), and NVL() 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.

Solutions

MySQL

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

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

FROM Products;
```

or we can use the COALESCE() function, like this:

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

```
FROM Products;
```

SQL Server

The SQL Server ISNULL() function lets you return an alternative value when an expression is NULL:

```
SELECT ProductName, UnitPrice * (UnitsInStock + ISNULL(UnitsOnOrder,
0))

FROM Products;
```

What is a Stored Procedure?

A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.

So if you have an SQL query that you write over and over again, save it as a stored procedure, and then just call it to execute it.

You can also pass parameters to a stored procedure, so that the stored procedure can act based on the parameter value(s) that is passed.

Stored Procedure Syntax

```
CREATE PROCEDURE procedure_name

AS

sql_statement

GO;
```

Execute a Stored Procedure

```
EXEC procedure_name;
```

Example

Get your own SQL Server

```
CREATE PROCEDURE SelectAllCustomers

AS

SELECT * FROM Customers

GO;
```

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:

Example

```
-- Select all:

SELECT * FROM Customers;
```