Theory1

September 14, 2025

Theory SQL is a standard language for storing, manipulating and retrieving data in databases.

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

The 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.

SQL Statements:

The following SQL statement returns all records from a table named "Customers":

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

–Syntax:

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

[SQL keywords are NOT case sensitive: select is the same as SELECT]

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

Select all the different countries from the "Customers" table:

```
SELECT DISTINCT Country FROM Customers;
```

Syntax:

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

The WHERE clause is used to filter records.

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

–Syntax:

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

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

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

– Syntax:

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

–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:

```
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:

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

The WHERE clause can contain one or many AND operators.

The AND operator is used to filter records based on more than one condition.

```
SELECT * FROM Customers WHERE Country = 'Spain' AND CustomerName LIKE 'G%';
```

–Syntax:

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

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.

SQL OR:

– Syntax:

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

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

– Syntax:

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

Example:

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

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

```
SELECT * FROM Customers WHERE CustomerID NOT BETWEEN 10 AND 60;
```

```
SELECT * FROM Customers WHERE City NOT IN ('Paris', 'London');
```

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

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

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

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

```
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');
```

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.

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

– UPDATE Syntax:

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
```

```
WHERE condition;
```

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

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

– DELETE Syntax:

```
DELETE FROM table_name WHERE condition;

DELETE FROM Customers WHERE CustomerName='Alfreds Futterkiste';

DELETE FROM table_name;
```

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

```
DROP TABLE Customers;
```

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.

– Select only the first 3 records of the Customers table:

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

Not all database systems support the SELECT TOP clause.

MySQL supports the LIMIT clause to select a limited number of records, while Oracle uses FETCH FIRST n ROWS ONLY and ROWNUM.

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

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

```
SELECT * FROM Customers
FETCH FIRST 3 ROWS ONLY;
```

```
SELECT TOP 50 PERCENT * FROM Customers;
```

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()).

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

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

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

```
SELECT MIN(Price) AS SmallestPrice, CategoryID
FROM Products
GROUP BY CategoryID;
```

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

```
SELECT COUNT(ProductID)
FROM Products
WHERE Price > 20;
```

```
SELECT COUNT(DISTINCT Price)
FROM Products;
```

```
SELECT COUNT(*) AS [Number of records]
FROM Products;
```

```
SELECT COUNT(*) AS [Number of records], CategoryID
FROM Products
GROUP BY CategoryID;
```

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

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:

- A. The percent sign % represents zero, one, or multiple characters

- B. The underscore sign _ represents one, single character

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

```
SELECT * FROM Customers
WHERE city LIKE 'L_nd__';
```

```
SELECT * FROM Customers
WHERE city LIKE '%L%';
```

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

```
SELECT * FROM Customers
WHERE CustomerName LIKE 'b%s';
```

–Return all customers that have “r” in the second position:

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

Wildcard Characters

Symbol Description

- % –> Represents zero or more characters
- _ –> Represents a single character
- [] –> Represents any single character within the brackets
- ^ –> Represents any character not in the brackets
- - –> Represents any single character within the specified range
- {} –> Represents any escaped character

–Return all customers starting with either “b”, “s”, or “p”:

```
SELECT * FROM Customers
WHERE CustomerName LIKE '[bsp]%';
```

– Return all customers starting with “a”, “b”, “c”, “d”, “e” or “f”:

```
SELECT * FROM Customers
WHERE CustomerName LIKE '[a-f]%';
```

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

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

```
SELECT * FROM Customers
WHERE CustomerID NOT IN (SELECT CustomerID FROM Orders);
```

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.

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

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

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.

```
SELECT CustomerID AS ID
FROM Customers;
```

```
SELECT CustomerID ID
FROM Customers;
```

```
SELECT ProductName AS [My Great Products]
FROM Products;
```

```
SELECT ProductName AS "My Great Products"
FROM Products;
```

```
SELECT CustomerName, Address + ', ' + PostalCode + ' ' + City + ', ' + Country AS Address
FROM Customers;
```

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

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

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

Different Types of SQL JOINs:

Here are the different types of the JOINs in SQL:

- A. **(INNER) JOIN:** Returns records that have matching values in both tables
- B. **LEFT (OUTER) JOIN:** Returns all records from the left table, and the matched records from the right table
- C. **RIGHT (OUTER) JOIN:** Returns all records from the right table, and the matched records from the left table
- D. **FULL (OUTER) JOIN:** Returns all records when there is a match in either left or right table

JOIN and INNER JOIN will return the same result.

INNER is the default join type for JOIN, so when you write JOIN the parser actually writes INNER JOIN.

```
SELECT Products.ProductID, Products.ProductName, Categories.CategoryName
FROM Products
JOIN Categories ON Products.CategoryID = Categories.CategoryID
```

```
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);
```

```
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).

The RIGHT JOIN keyword returns all records from the right table (table2), and the matching records from the left table (table1).

The result is 0 records from the left side, if there is no match.

```
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).

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

Tip: FULL OUTER JOIN and FULL JOIN are the same.

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

A self join is a regular join, but the table is joined with itself.

```
SELECT column_name(s)
FROM table1 T1, table1 T2
WHERE condition;
```

T1 and T2 are different table aliases for the same table.

```
SELECT A.CustomerName AS CustomerName1, B.CustomerName AS CustomerName2, A.City
FROM Customers A, Customers B
WHERE A.CustomerID <> B.CustomerID
AND A.City = B.City
ORDER BY A.City;
```

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

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

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;
```

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

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

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

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

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

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

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

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

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

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

```
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 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.

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.supplierI
```

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

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

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

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

```
SELECT *
INTO newtable [IN externaldb]
FROM oldtable
WHERE condition;
```

The new table will be created with the column-names and types as defined in the old table. You can create new column names using the AS clause.

```
SELECT * INTO CustomersBackup2017
FROM Customers;
```

– The following SQL statement uses the IN clause to copy the table into a new table in another database:

```
SELECT * INTO CustomersBackup2017 IN 'Backup.mdb'
FROM Customers;
```

– The following SQL statement copies only a few columns into a new table:

```
SELECT CustomerName, ContactName INTO CustomersBackup2017
FROM Customers;
```

Tip: SELECT INTO can also be used to create a new, empty table using the schema of another. Just add a WHERE clause that causes the query to return no data:

```
SELECT * INTO newtable
```

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.

– 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;
```

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

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;
```

```
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);
```

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

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

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

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

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;

EXEC procedure_name;
```

```
CREATE PROCEDURE SelectAllCustomers
AS
SELECT * FROM Customers;

EXEC SelectAllCustomers;
```

Stored Procedure With One Parameter:

The following SQL statement creates a stored procedure that selects Customers from a particular City from the “Customers” table:

```
CREATE PROCEDURE SelectAllCustomers @City nvarchar(30)
```

```
AS
SELECT * FROM Customers WHERE City = @City

EXEC SelectAllCustomers @City = 'London';
```

```
CREATE PROCEDURE SelectAllCustomers @City nvarchar(30), @PostalCode nvarchar(10)
AS
SELECT * FROM Customers WHERE City = @City AND PostalCode = @PostalCode

EXEC SelectAllCustomers @City = 'London', @PostalCode = 'WA1 1DP';
```

Single line Comment: –

Multiline Comment: /* */

```
CREATE DATABASE databasename;
```

```
DROP DATABASE databasename;
```

Once a database is dropped, you can check it in the list of databases with the following SQL command: `SHOW DATABASES;`

The BACKUP DATABASE statement is used in SQL Server to create a full back up of an existing SQL database.

```
BACKUP DATABASE databasename
TO DISK = 'filepath';
```

A differential back up only backs up the parts of the database that have changed since the last full database backup.

```
BACKUP DATABASE databasename
TO DISK = 'filepath'
WITH DIFFERENTIAL;
```

Example:

```
BACKUP DATABASE testDB
TO DISK = 'D:\backups\testDB.bak';
```

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

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

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.

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

```
DROP TABLE table_name;
```

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

```
TRUNCATE TABLE table_name;
```

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 table_name
ADD column_name datatype;
```

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

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

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

```
ALTER TABLE table_name
RENAME COLUMN old_name to new_name;
```

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

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

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

To rename a column in a table in SQL Server, use the following syntax:

EXEC sp_rename 'table_name.old_name', 'new_name', 'COLUMN';

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255) NOT NULL,
    Age int
);
```

The UNIQUE constraint ensures that all values in a column are different.

Both the UNIQUE and PRIMARY KEY constraints provide a guarantee for uniqueness for a column or set of columns.

A PRIMARY KEY constraint automatically has a UNIQUE constraint.

However, you can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table.

```
CREATE TABLE Persons (
    ID int NOT NULL UNIQUE,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int
);
```

To name a UNIQUE constraint, and to define a UNIQUE constraint on multiple columns, use the following SQL syntax:

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    CONSTRAINT UC_Person UNIQUE (ID,LastName)
);
```

```
ALTER TABLE Persons
ADD UNIQUE (ID);
```

```
ALTER TABLE Persons
ADD CONSTRAINT UC_Person UNIQUE (ID,LastName);
```

To drop a UNIQUE constraint, use the following SQL:

```
ALTER TABLE Persons
DROP INDEX UC_Person;
```

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    PRIMARY KEY (ID)
);
```

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    CONSTRAINT PK_Person PRIMARY KEY (ID,LastName)
);
```

```
ALTER TABLE Persons
ADD PRIMARY KEY (ID);
```

```
ALTER TABLE Persons
ADD CONSTRAINT PK_Person PRIMARY KEY (ID,LastName);
```

```
ALTER TABLE Persons
DROP PRIMARY KEY;
```

```
ALTER TABLE Persons
DROP CONSTRAINT PK_Person;
```

```
CREATE TABLE Orders (
    OrderID int NOT NULL,
    OrderNumber int NOT NULL,
    PersonID int,
    PRIMARY KEY (OrderID),
    FOREIGN KEY (PersonID) REFERENCES Persons(PersonID)
);
```

To allow naming of a FOREIGN KEY constraint, and for defining a FOREIGN KEY constraint on multiple columns, use the following SQL syntax:

```
CREATE TABLE Orders (
    OrderID int NOT NULL,
    OrderNumber int NOT NULL,
    PersonID int,
    PRIMARY KEY (OrderID),
    CONSTRAINT FK_PersonOrder FOREIGN KEY (PersonID)
    REFERENCES Persons(PersonID)
);
```

```
ALTER TABLE Orders
ADD FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);
```

```
ALTER TABLE Orders
ADD CONSTRAINT FK_PersonOrder
FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);
```

```
ALTER TABLE Orders
DROP FOREIGN KEY FK_PersonOrder;
```

The CHECK constraint is used to limit the value range that can be placed in a column.

If you define a CHECK constraint on a column it will allow only certain values for this column.

If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row.

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    CHECK (Age>=18)
);
```

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    City varchar(255),
    CONSTRAINT CHK_Person CHECK (Age>=18 AND City='Sandnes')
);
```

```
ALTER TABLE Persons
ADD CHECK (Age>=18);
```

```
ALTER TABLE Persons
ADD CONSTRAINT CHK_PersonAge CHECK (Age>=18 AND City='Sandnes');
```

```
ALTER TABLE Persons
DROP CONSTRAINT CHK_PersonAge;
```

```
ALTER TABLE Persons
DROP CHECK CHK_PersonAge;
```

The DEFAULT constraint is used to set a default value for a column.

The default value will be added to all new records, if no other value is specified.

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    City varchar(255) DEFAULT 'Sandnes'
);
```

The DEFAULT constraint can also be used to insert system values, by using functions like GETDATE():

```
CREATE TABLE Orders (
    ID int NOT NULL,
    OrderNumber int NOT NULL,
    OrderDate date DEFAULT GETDATE()
);
```

```
ALTER TABLE Persons
ALTER City SET DEFAULT 'Sandnes';
```

```
ALTER TABLE Persons
ALTER City DROP DEFAULT;
```

The CREATE INDEX statement is used to create indexes in tables.

Indexes are used to retrieve data from the database more quickly than otherwise. The users cannot see the indexes, they are just used to speed up searches/queries

Note: Updating a table with indexes takes more time than updating a table without (because the indexes also need an update).
So, only create indexes on columns that will be frequently searched against.

Creates an index on a table. Duplicate values are allowed:

```
CREATE INDEX index_name
ON table_name (column1, column2, ...);
```

Creates a unique index on a table. Duplicate values are not allowed:

```
CREATE UNIQUE INDEX index_name
ON table_name (column1, column2, ...);
```

The SQL statement below creates an index named "idx_lastname" on the "LastName" column in the "Persons" table:

```
CREATE INDEX idx_lastname
ON Persons (LastName);
```

If you want to create an index on a combination of columns, you can list the column names within the parentheses, separated by commas:

```
CREATE INDEX idx_pname
ON Persons (LastName, FirstName);
```

DROP INDEX index_name ON table_name;

```
ALTER TABLE table_name
DROP INDEX index_name;
```

Auto-increment allows a unique number to be generated automatically when a new record is inserted into a table.

Often this is the primary key field that we would like to be created automatically every time a new record is inserted.

```
CREATE TABLE Persons (
   Personid int NOT NULL AUTO_INCREMENT,
   LastName varchar(255) NOT NULL,
   FirstName varchar(255),
   Age int,
   PRIMARY KEY (Personid)
);
```

MySQL uses the AUTO_INCREMENT keyword to perform an auto-increment feature.

By default, the starting value for AUTO_INCREMENT is 1, and it will increment by 1 for each new record.

To let the AUTO_INCREMENT sequence start with another value, use the following SQL statement:

```
ALTER TABLE Persons AUTO_INCREMENT=100;
```

–To insert a new record into the “Persons” table, we will NOT have to specify a value for the “Personid” column (a unique value will be added automatically):

```
INSERT INTO Persons (FirstName,LastName)
VALUES ('Lars','Monsen');
```

– The following SQL statement defines the “Personid” column to be an auto-increment primary key field in the “Persons” table:

```
CREATE TABLE Persons (
    Personid int IDENTITY(1,1) PRIMARY KEY,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int
);
```

The MS SQL Server uses the IDENTITY keyword to perform an auto-increment feature.

In the example above, the starting value for IDENTITY is 1, and it will increment by 1 for each new record.

Tip: To specify that the “Personid” column should start at value 10 and increment by 5, change it to IDENTITY(10,5).

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

Example:

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

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

– 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;
```

– A view is deleted with the DROP VIEW statement.

```
DROP VIEW view_name;
```

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

SQL injection is a code injection technique that might destroy your database.

SQL injection is one of the most common web hacking techniques.

SQL injection is the placement of malicious code in SQL statements, via web page input.

SQL injection usually occurs when you ask a user for input, like their username/userid, and instead of a name/id, the user gives you an SQL statement that you will unknowingly run on your database.

Look at the following example which creates a SELECT statement by adding a variable (txtUserId) to a select string.

The variable is fetched from user input (getRequestString):

```
txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```

he original purpose of the code was to create an SQL statement to select a user, with a given user id.

If there is nothing to prevent a user from entering “wrong” input, the user can enter some “smart” input like this:

HTML Inputbox : UserId: 105 OR 1=1

Then, the SQL statement will look like this:

```
SELECT * FROM Users WHERE UserId = 105 OR 1=1;
```

The SQL above is valid and will return ALL rows from the “Users” table, since OR 1=1 is always TRUE.

Does the example above look dangerous? What if the “Users” table contains names and passwords?

The SQL statement above is much the same as this:

```
SELECT UserId, Name, Password FROM Users WHERE UserId = 105 or 1=1;
```

Use SQL Parameters for Protection:

To protect a web site from SQL injection, you can use SQL parameters.

SQL parameters are values that are added to an SQL query at execution time, in a controlled manner.

ASP.NET Razor Example

```
txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = @0";
db.Execute(txtSQL,txtUserId);
```