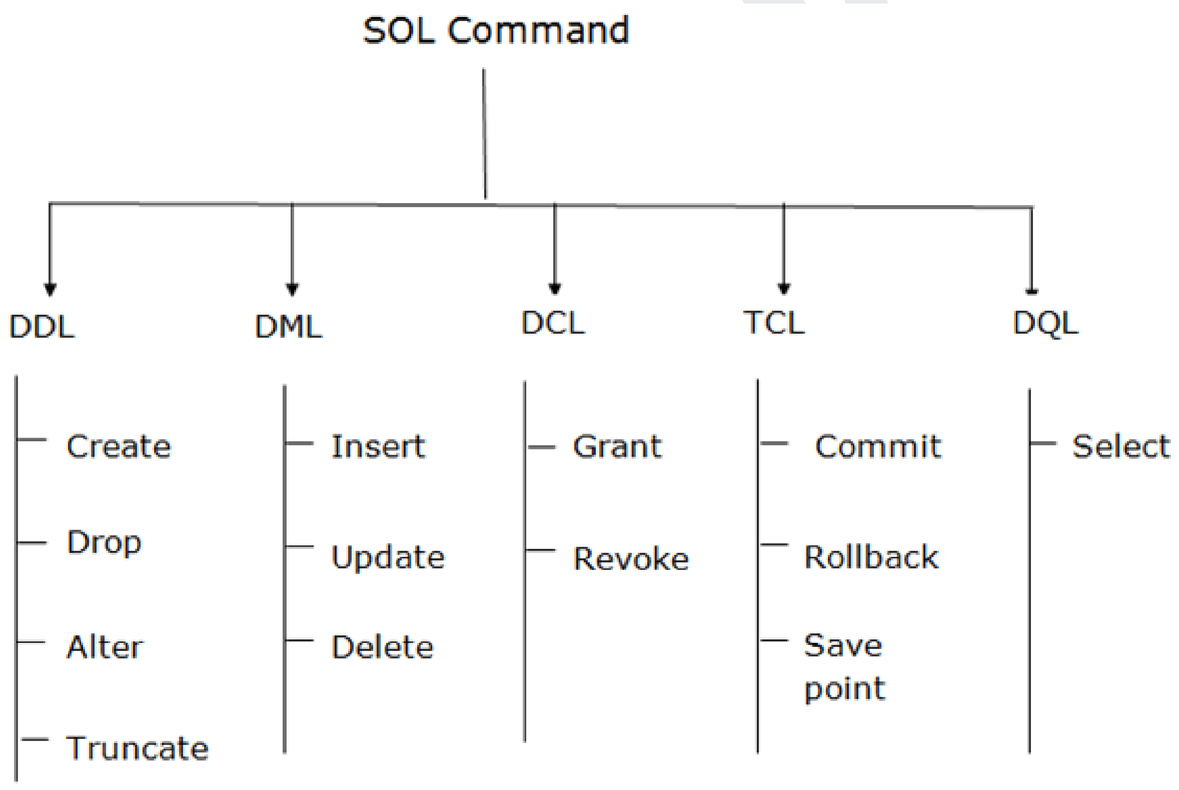
Let’s work with mysql database management system

SQL - structured query language to communicate with the database.

Mysql - Relational Database management system.
Mysql uses sql to interact with databases.
Note → sql is case in-sensitive (create and CREATE is same)

Single line comments start with `--`
Any text between -- and the end of the line will be ignored (will not be executed).
Multi-line comments start with `/*` and end with `*/`.



Data Definition Language (DDL)

CREATE → It is used to create a new table in the database.
DROP → It is used to delete both the structure and record stored in the table.
ALTER → It is used to alter the structure of the database.
-To add a new column in the table.
-To modify existing column in the table:
TRUNCATE → removes all rows from a table, but the table structure and its columns, constraints, indexes, and so on remain.

Data Manipulation Language(DML)

INSERT → It is used to insert data into the row of a table.

UPDATE → it is used to update or modify the value of a column in the table.

DELETE → It is used to remove one or more rows from a table.

Data Control Language(DCL)

Grant → It is used to give user access privileges to a database.

Revoke → It is used to take back permissions from the user.

Transaction Control Language(TCL)

Commit →it is used to save all the transactions to the database.

Rollback→it is used to undo transactions that have not already been saved to the database.

SAVEPOINT→It is used to roll the transaction back to a certain point without rolling back the entire transaction.

Data Query Language(DQL)

SELECT → It is used to select the attribute based on the condition described by WHERE clause.

Note → Database Name - **company**

→Table Name - **employee**

- **Create Database company ;** → create new database
- **Show databases ;** → show all databases
- **Use company ;** → use company database
- **Create table employee(Id int not null primary key auto_increment ,**

  name char(20) , age int , salary double (15) , department varchar(20)) ;

- **insert into employee (name , age , salary , department) values('chail', 22 , 10000 , Software);** → insert data in table , id will generated automatically

  If you are adding data sequentially you can direct use values with define column name →

- **insert into employee values(' chail ' , 20 , 10000 , Software);**
- **insert into employee values(' gaju ' , 22 , 20000 , IT);**
- **insert into employee values(' ram ' , 24 , 30000 , Electronics);**
- **insert into employee values(' komal ', 25 , 40000 , HR);**
- **select * from employee ;** → select all(*) rows from table
- **select name , age from employee ;** → select name and age row from table
- **Select distinct name from employee** → select only distinct name row from table
- **Select count(distinct name) from employee** → select only distinct name row from table
- `SELECT Count(*) AS DistinctCountries`
- `FROM (SELECT DISTINCT Country FROM Customers);`
- `SELECT * FROM Customers`
- `WHERE Country='Mexico';`
- **alter table employeeinfo add mobile int ;** → add new column(mobile)
- **alter table employeeinfo drop mobile ;** → delete mobile column
- **truncate table employee** → removes all rows from a table , structure remains as it is .
- **select * from employee limit 2 ;** → fetch top 2 records

- **Importants :**

```sql
SELECT * FROM Customers;

SELECT CustomerName, City FROM Customers;

SELECT DISTINCT Country FROM Customers;

SELECT COUNT(DISTINCT Country) FROM Customers;

SELECT Count(*) AS DistinctCountries FROM (SELECT DISTINCT Country FROM
Customers);

SELECT * FROM Customers WHERE Country='Mexico';

SELECT * FROM Customers WHERE CustomerID=1;

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

SELECT * FROM Customers WHERE City='Berlin' OR City='München';

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

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

SELECT * FROM Customers ORDER BY Country;

SELECT * FROM Customers ORDER BY Country DESC;

SELECT * FROM Customers ORDER BY Country, CustomerName;

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

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



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

UPDATE Customers SET ContactName = 'Alfred Schmidt' → this will give an error(sql_safe_update) , this query will set all contactName with Alfred Schmidt in whole table , this may be a virus who is trying to modify whole table for safety purpose mysql itself enable sql_safe__update=1 . now if you want forcefully run this query you will have to disable this safety by - set sql_safe_updates=0;

So whenever you want to update , give any condition statement with where keyword

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

Same case with delete

DELETE CustomerName from customers ; → error

DELETE FROM Customers WHERE CustomerName='Alfreds Futterkiste';

SELECT * FROM Customers LIMIT 3; → return top 3 record (mysql)

SELECT TOP 3 * FROM Customers; → return top 3 record(sql server/ms access)

SELECT * FROM Customers FETCH FIRST 3 ROWS ONLY;(oracle)

SELECT TOP 50 PERCENT * FROM Customers;(sql server/ms-access)

SELECT * FROM Customers FETCH FIRST 50 PERCENT ROWS ONLY;(oracle)

SELECT MIN(Price) FROM Products;

SELECT MIN(Price) AS SmallestPrice FROM Products;

SELECT MAX(Price) FROM Products;

SELECT MAX(Price) AS LargestPrice FROM Products;

SELECT COUNT(ProductID) FROM Products; → NULL values are not counted.

SELECT AVG(Price) FROM Products; → NULL values are ignored.

`SELECT SUM(Quantity) FROM OrderDetails;` → **NULL values are ignored.**

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

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

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

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

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

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

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

`SELECT * FROM Customers WHERE City LIKE '[bsp]%';` → **selects all customers with a City starting with "b", "s", or "p":**

`SELECT * FROM Customers WHERE City LIKE '[a-c]%';` → **selects all customers with a City starting with "a", "b", or "c":**

`SELECT * FROM Customers WHERE City LIKE '[!bsp]%';` → **select all customers with a City NOT starting with "b", "s", or "p":**

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

The `IN` operator is a shorthand for multiple `OR` conditions.

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

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

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

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

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

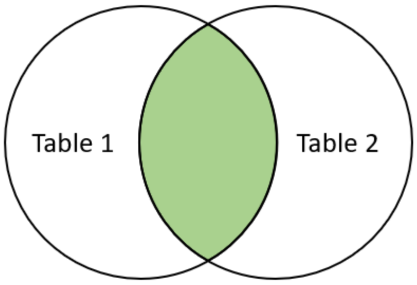
```sql
SELECT * FROM Orders WHERE OrderDate BETWEEN '1996-07-01' AND
'1996-07-31';
```

aliases are used to give a table, or a column in a table, a temporary name. An alias is created with the `AS` keyword.

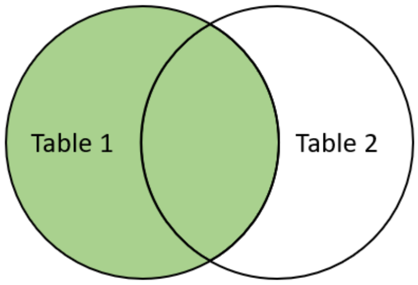
```sql
SELECT CustomerID AS ID, CustomerName AS Customer FROM Customers;
```

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

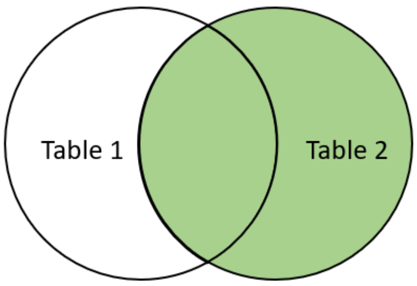
A `JOIN` clause is used to combine rows from two or more tables, based on a related column between them.



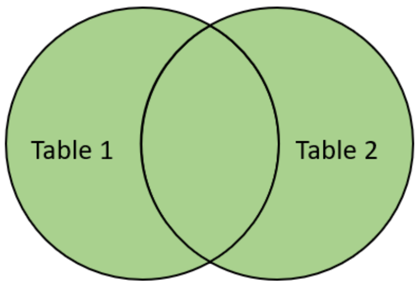
INNER JOIN



LEFT JOIN



RIGHT JOIN



FULL JOIN

INNER JOIN →

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

```
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);
```

LEFT JOIN →

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

RIGHT JOIN →

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

FULL JOIN →

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

SELF JOIN →

```
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;
```

UNION →

```
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;
```

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.

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

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 ORDER BY COUNT(CustomerID) DESC;
```

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.

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

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.

```
SELECT ProductName
FROM Products
```

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

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.

```
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 CustomersBackup FROM Customers;

SELECT * INTO CustomersBackup IN 'Backup.mdb' FROM Customers;

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

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.

```
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.

```
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;
```

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

Stored Procedure Syntax

```
CREATE PROCEDURE SelectAllCustomers

AS

SELECT * FROM Customers

GO;
```

Execute a Stored Procedure

```
EXEC SelectAllCustomers;
```

```
CREATE DATABASE testDB;

DROP DATABASE testDB;
```

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;
```

The following SQL statement creates a full back up of the existing database "testDB" to the D disk:

```
BACKUP DATABASE testDB

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



CREATE TABLE Persons (

  PersonID int,

  LastName varchar(255),

  FirstName varchar(255),

  Address varchar(255),

  City varchar(255)

);



CREATE TABLE TestTable AS

SELECT customername, contactname

FROM customers;
```

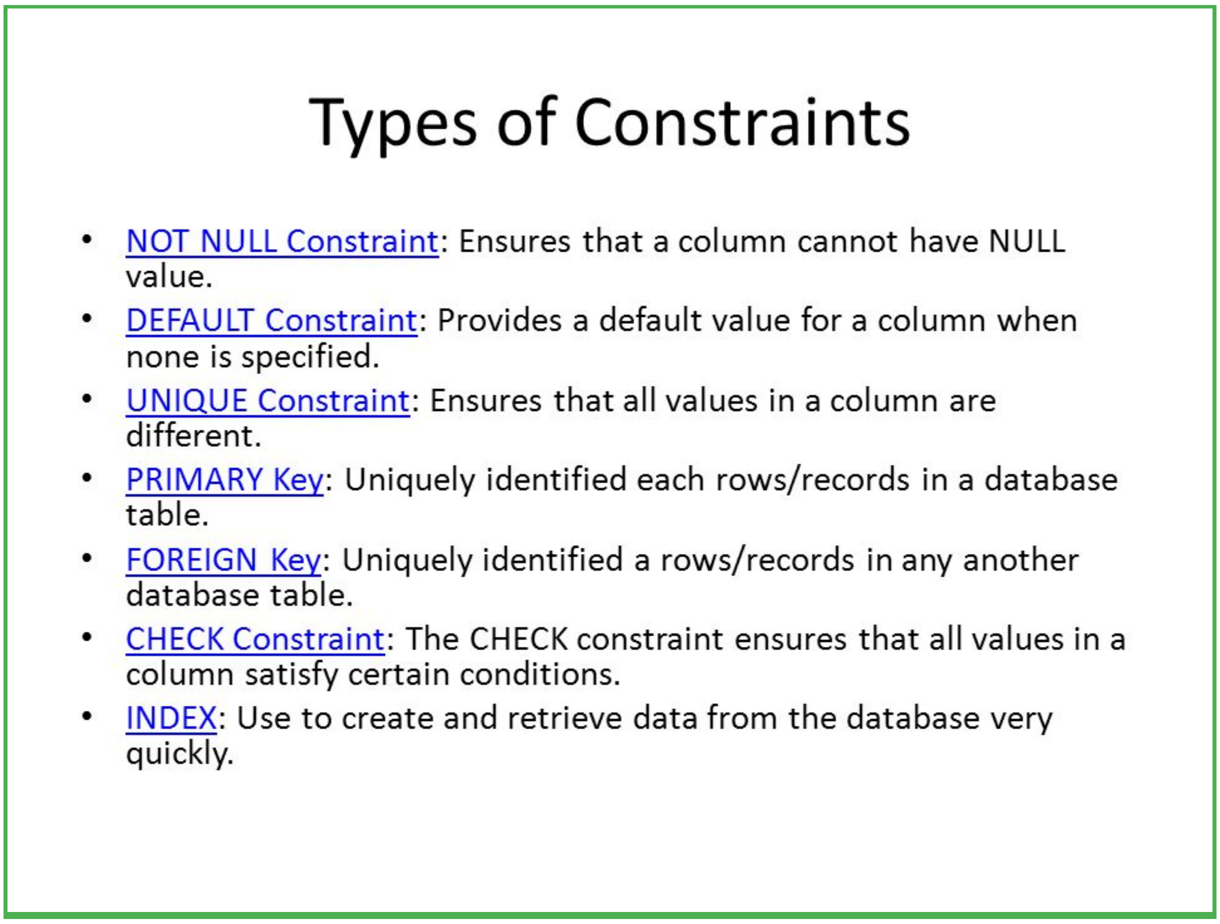
```
DROP TABLE Shippers;

TRUNCATE TABLE Shippers;

ALTER TABLE Customers ADD Email varchar(255);

ALTER TABLE Customers DROP COLUMN Email;

ALTER TABLE table_name MODIFY COLUMN column_name datatype;
```



Types of Constraints

- NOT NULL Constraint: Ensures that a column cannot have NULL value.
- DEFAULT Constraint: Provides a default value for a column when none is specified.
- UNIQUE Constraint: Ensures that all values in a column are different.
- PRIMARY Key: Uniquely identified each rows/records in a database table.
- FOREIGN Key: Uniquely identified a rows/records in any another database table.
- CHECK Constraint: The CHECK constraint ensures that all values in a column satisfy certain conditions.
- INDEX: Use to create and retrieve data from the database very quickly.

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

ALTER TABLE Persons ALTER COLUMN Age int NOT NULL;
```

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

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

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

```
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) DEFAULT 'Sandnes'
);
```

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.

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

```
DROP INDEX index_name ON table_name;

CREATE TABLE Persons (

    Personid int NOT NULL AUTO_INCREMENT,

  LastName varchar(255) NOT NULL,

  FirstName varchar(255),

  Age int,

    PRIMARY KEY (Personid)

);
```

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;
```

- 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

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

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 [Brazil Customers] AS

SELECT CustomerName, ContactName

FROM Customers

WHERE Country = 'Brazil';
```

SQL Injection

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):

SQL Injection Based on 1=1 is Always True

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

SQL Injection Based on ""="" is Always True

```
SELECT * FROM Users WHERE Name ="" or ""="" AND Pass ="" or ""=""
```

SQL Injection Based on Batched SQL Statements

```
SELECT * FROM Users WHERE UserId = 105; DROP TABLE Suppliers;
```

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.

SQL Hosting

If you want your web site to be able to store and retrieve data from a database, your web server should have access to a database-system that uses the SQL language.

If your web server is hosted by an Internet Service Provider (ISP), you will have to look for SQL hosting plans.

The most common SQL hosting databases are MS SQL Server, Oracle, MySQL, and MS Access.

SQL Data Types

Each column in a database table is required to have a name and a data type.

An SQL developer must decide what type of data that will be stored inside each column when creating a table. The data type is a guideline for SQL to understand what type of data is expected inside of each column, and it also identifies how SQL will interact with the stored data.

CHAR(size)	A FIXED length string (can contain letters, numbers, and special characters). The *size* parameter specifies the column length in characters - can be from 0 to 255. Default is 1
VARCHAR(size)	A VARIABLE length string (can contain letters, numbers, and special characters). The *size* parameter specifies the maximum string length in characters - can be from 0 to 65535
BINARY(size)	Equal to CHAR(), but stores binary byte strings. The *size* parameter specifies the column length in bytes. Default is 1
VARBINARY(size)	Equal to VARCHAR(), but stores binary byte strings. The *size* parameter specifies the maximum column length in bytes.
TINYBLOB	For BLOBs (Binary Large Objects). Max length: 255 bytes
TINYTEXT	Holds a string with a maximum length of 255 characters

TEXT(size)	Holds a string with a maximum length of 65,535 bytes
BLOB(size)	For BLOBs (Binary Large Objects). Holds up to 65,535 bytes of data
MEDIUMTEXT	Holds a string with a maximum length of 16,777,215 characters
MEDIUMBLOB	For BLOBs (Binary Large Objects). Holds up to 16,777,215 bytes of data
LONGTEXT	Holds a string with a maximum length of 4,294,967,295 characters
LONGBLOB	For BLOBs (Binary Large Objects). Holds up to 4,294,967,295 bytes of data
ENUM(val1, val2, val3, ...)	A string object that can have only one value, chosen from a list of possible values. You can list up to 65535 values in an ENUM list. If a value is inserted that is not in the list, a blank value will be inserted. The values are sorted in the order you enter them
SET(val1, val2, val3, ...)	A string object that can have 0 or more values, chosen from a list of possible values. You can list up to 64 values in a SET list

Numeric Data Types

Data type	**Description**
BIT(*size*)	A bit-value type. The number of bits per value is specified in *size*. The *size* parameter can hold a value from 1 to 64. The default value for *size* is 1.
TINYINT(*size*)	A very small integer. Signed range is from -128 to 127. Unsigned range is from 0 to 255. The *size* parameter specifies the maximum display width (which is 255)
BOOL	Zero is considered as false, nonzero values are considered as true.
BOOLEAN	Equal to BOOL
SMALLINT(*size*)	A small integer. Signed range is from -32768 to 32767. Unsigned range is from 0 to 65535. The *size* parameter specifies the maximum display width (which is 255)
MEDIUMINT(*size*)	A medium integer. Signed range is from -8388608 to 8388607. Unsigned range is from 0 to 16777215. The *size* parameter specifies the maximum display width (which is 255)
INT(*size*)	A medium integer. Signed range is from -2147483648 to 2147483647. Unsigned range is from 0 to 4294967295. The *size* parameter

	specifies the maximum display width (which is 255)
INTEGER(*size*)	Equal to INT(size)
BIGINT(*size*)	A large integer. Signed range is from -9223372036854775808 to 9223372036854775807. Unsigned range is from 0 to 18446744073709551615. The *size* parameter specifies the maximum display width (which is 255)
FLOAT(*size*, *d*)	A floating point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter. This syntax is deprecated in MySQL 8.0.17, and it will be removed in future MySQL versions
FLOAT(*p*)	A floating point number. MySQL uses the *p* value to determine whether to use FLOAT or DOUBLE for the resulting data type. If *p* is from 0 to 24, the data type becomes FLOAT(). If *p* is from 25 to 53, the data type becomes DOUBLE()
DOUBLE(*size*, *d*)	A normal-size floating point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter
DOUBLE PRECISION(*size*, *d*)	

DECIMAL(*size*, *d*)	An exact fixed-point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter. The maximum number for *size* is 65. The maximum number for *d* is 30. The default value for *size* is 10. The default value for *d* is 0.
DEC(*size*, *d*)	Equal to DECIMAL(size,d)

Note: All the numeric data types may have an extra option: UNSIGNED or ZEROFILL. If you add the UNSIGNED option, MySQL disallows negative values for the column. If you add the ZEROFILL option, MySQL automatically also adds the UNSIGNED attribute to the column.

Date and Time Data Types

Data type	**Description**
DATE	A date. Format: YYYY-MM-DD. The supported range is from '1000-01-01' to '9999-12-31'
DATETIME(*fsp*)	A date and time combination. Format: YYYY-MM-DD hh:mm:ss. The supported range is from '1000-01-01 00:00:00' to '9999-12-31 23:59:59'. Adding DEFAULT and ON UPDATE in the column definition to get automatic initialization and updating to the current date and time
TIMESTAMP(*fsp*)	A timestamp. TIMESTAMP values are stored as the number of seconds since the Unix epoch ('1970-01-01 00:00:00' UTC). Format: YYYY-MM-DD hh:mm:ss. The supported range is

	from '1970-01-01 00:00:01' UTC to '2038-01-09 03:14:07' UTC. Automatic initialization and updating to the current date and time can be specified using DEFAULT CURRENT_TIMESTAMP and ON UPDATE CURRENT_TIMESTAMP in the column definition
TIME(*fsp*)	A time. Format: hh:mm:ss. The supported range is from '-838:59:59' to '838:59:59'
YEAR	A year in four-digit format. Values allowed in four-digit format: 1901 to 2155, and 0000. MySQL 8.0 does not support year in two-digit format.