💻

Practical - MYSQL

@Sujal Samai

What is MySQL?

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

What is RDBMS?

RDBMS stands for Relational Database Management System. RDBMS is a program used to maintain a relational database. It is the basis for all modern database systems such as MySQL, Microsoft SQL Server, Oracle, and Microsoft Access. RDBMS uses SQL queries to access the data in the database.

What is a Database Table?

A table is a collection of related data entries, and it consists of columns and rows.

A column holds specific information about every record in the table.

A record (or row) is each individual entry that exists in a table.

What is a Relational Database?

A relational database defines database relationships in the form of tables. The tables are related to each other - based on data common to each.

What is SQL?

SQL is the standard language for dealing with Relational Databases. SQL is used to insert, search, update, and delete database records.

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

Semicolon is the standard way to separate each SQL statement in database systems that allow more than one SQL statement to be executed in the same call to the server.

SELECT Statement

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

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

SELECT DISTINCT

The `SELECT DISTINCT` statement is used to return only distinct (different) values. Inside a table, a column often contains many duplicate values; and sometimes you only want to list the different (distinct) values. Syntax:

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

WHERE Clause

The `WHERE` clause is used to filter records. It is used to extract only those records that fulfill a specified condition. The `WHERE`clause is not only used in `SELECT` statements, it is also used in `UPDATE`, `DELETE`, etc.! Syntax:

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
Eg: SELECT * FROM Customers WHERE Country = 'Mexico';
```

SQL requires single quotes around text values (most database systems will also allow double quotes). However, numeric fields should not be enclosed in quotes

AND, OR, NOT

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.

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

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

We can also combine the `AND`, `OR` and `NOT` operators.

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

ORDER BY

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

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

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

Can also do this
SELECT * FROM Customers
ORDER BY Country ASC, CustomerName DESC;
```

INSERT INTO

The `INSERT INTO` statement is used to insert new records in a table. 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, ...);
```

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, ...);
```

NULL values

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

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

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

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.

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

UPDATE

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

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!

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

DELETE

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

```
DELETE FROM table_name WHERE condition;

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;
```

LIMIT

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

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

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

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.

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

COUNT(), AVG() and SUM()

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

```
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;
```

LIKE OPERATOR

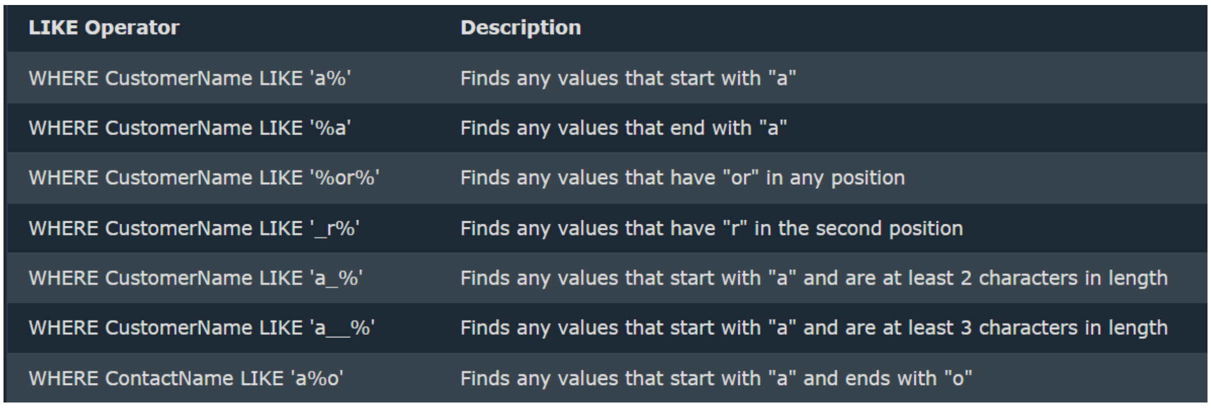
The `LIKE` operator is used in a `WHERE` clause to search for a specified pattern in a column.

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

- The percent sign (%) represents zero, one, or multiple characters
- The underscore sign (_) represents one, single character

The percent sign and the underscore can also be used in combinations!

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



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

Wildcard Characters

A wildcard character is used to substitute one or more characters in a string.

Wildcard characters are used with the `LIKE` operator. The `LIKE` operator is used in a `WHERE` clause to search for a specified pattern in a column.

Symbol	Description	Example
%	Represents zero or more characters	bl% finds bl, black, blue, and blob
_	Represents a single character	h_t finds hot, hat, and hit

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.

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN|NOT IN (value1, value2, ...);
or:
SELECT column_name(s)
FROM table_name
WHERE column_name IN (SELECT STATEMENT);
```

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 column_name(s)
FROM table_name
WHERE column_name BETWEEN| NOT BETWEEN value1 AND value2;
```

Aliases

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

Aliases are often used to make column names more readable. An alias only exists for the duration of that query. An alias is created with the `AS` keyword.

```
SELECT column_name AS alias_name
FROM table_name;

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

Aliases can be useful when:

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

UNION

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

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

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

NOTE: We can also include WHERE and ORDER BY Statements

GROUP BY

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

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

HAVING

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

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

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

EXISTS

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

//The following SQL statement returns TRUE and lists the suppliers
//with a product price less than 20:
Eg: SELECT SupplierName
FROM Suppliers
WHERE EXISTS (SELECT ProductName FROM Products
              WHERE Products.SupplierID = Suppliers.supplierID AND
              Price < 20);
```

ANY, ALL Operators

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

```
//lists the ProductName if it finds ANY records in the OrderDetails
//table has Quantity equal to 10
Eg: 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 ALL column_name(s)
FROM table_name
WHERE condition;
or
SELECT column_name(s)
FROM table_name
WHERE column_name operator ALL
  (SELECT column_name
  FROM table_name
  WHERE condition);

//lists the ProductName if ALL the records in the OrderDetails
//table has Quantity equal to 10.
Eg: SELECT ProductName
FROM Products
WHERE ProductID = ALL
  (SELECT ProductID
  FROM OrderDetails
  WHERE Quantity = 10);
```

INSERT INTO SELECT

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 matches.

Note: The existing records in the target table are unaffected.

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

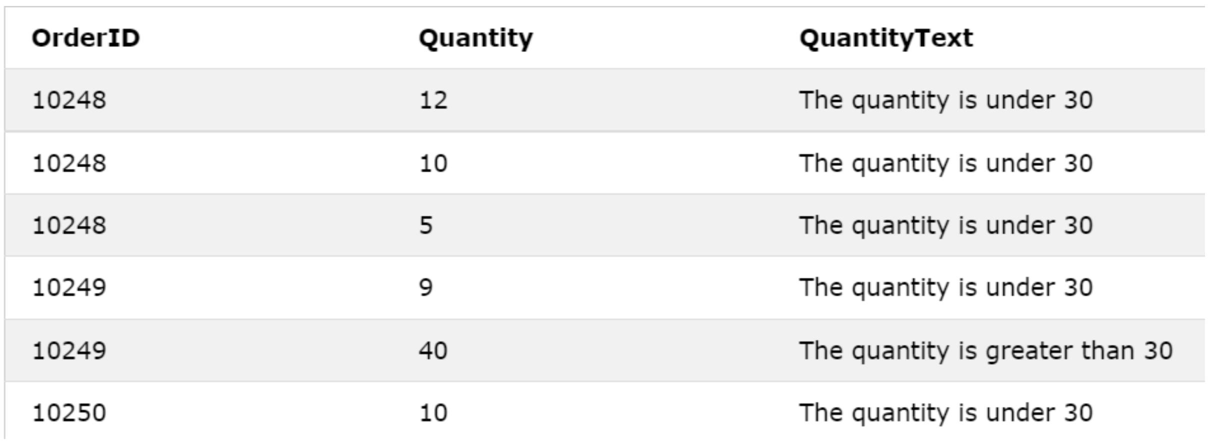
CASE Statement

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

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

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

```
Eg: 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;
Output:
```



OrderID	Quantity	QuantityText
10248	12	The quantity is under 30
10248	10	The quantity is under 30
10248	5	The quantity is under 30
10249	9	The quantity is under 30
10249	40	The quantity is greater than 30
10250	10	The quantity is under 30

JOINS

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

INNER JOIN

The `INNER JOIN` keyword selects records that have matching values in both tables.

Note: The `INNER JOIN` keyword selects all rows from both tables as long as there is a match between the columns. If there are records in the "Orders" table that do not have matches in "Customers", these orders will not be shown!

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

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

LEFT JOIN

The `LEFT JOIN` keyword returns all records from the left table (table1), and the matching records (if any) from the right table (table2).

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

RIGHT JOIN

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

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

CROSS JOIN

The `CROSS JOIN` keyword returns all records from both tables (table1 and table2).

```
SELECT column_name(s)
FROM table1
CROSS JOIN table2;
```

💡 **Note:** The `CROSS 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. If you add a `WHERE` clause (if table1 and table2 has a relationship), the `CROSS JOIN` will produce the same result as the `INNER JOIN` clause:

SELF JOIN

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;

Eg: 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;
```

CREATE

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

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

```
CREATE DATABASE databasename;

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

Creating Table from another Table

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

DROP

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

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

```
DROP DATABASE databasename;
DROP TABLE table_name;
```

TRUNCATE

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

```
TRUNCATE TABLE table_name;
```

ALTER TABLE

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 - ADD

ALTER TABLE table_name
DROP COLUMN column_name;   //Alter- DROP

ALTER TABLE table_name
MODIFY COLUMN column_name datatype;   //ALter- Modify
```

Constraints

Constraints can be specified when the table is created with the `CREATE TABLE` statement, or after the table is created with the `ALTER TABLE` statement. SQL constraints are used to specify rules for the data in a table.

Constraints are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the table. If there is any violation between the constraint and the data action, the action is aborted. Constraints can be column level or table level. Column level constraints apply to a column, and table level constraints apply to the whole table.

The following constraints are commonly used in SQL:

- `NOT NULL` - Ensures that a column cannot have a NULL value
- `UNIQUE` - Ensures that all values in a column are different
- `PRIMARY KEY` - A combination of a `NOT NULL` and `UNIQUE`. Uniquely identifies each row in a table
- `FOREIGN KEY` - Prevents actions that would destroy links between tables
- `CHECK` - Ensures that the values in a column satisfies a specific condition
- `DEFAULT` - Sets a default value for a column if no value is specified
- `CREATE INDEX` - Used to create and retrieve data from the database very quickly

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

NOT NULL Constraint

By default, a column can hold NULL values. The `NOT NULL` constraint enforces a column to NOT accept NULL values. This enforces a field to always contain a value, which means that you cannot insert a new record, or update a record without adding a value to this field.

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

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

UNIQUE Constraint

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.

```
Eg: CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    UNIQUE (ID)  //ID is unique
);
or
ALTER TABLE Persons
ADD UNIQUE (ID);

CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    CONSTRAINT UC_Person UNIQUE (ID,LastName)//ID, LastName is unique named as UC_person
);
or
ALTER TABLE Persons
ADD CONSTRAINT UC_Person UNIQUE (ID,LastName);

//DROP a UNIQUE Constraint
ALTER TABLE Persons
DROP INDEX UC_Person;
```

PRIMARY KEY Constraint

The `PRIMARY KEY` constraint uniquely identifies each record in a table. Primary keys must contain UNIQUE values, and cannot contain NULL values. A table can have only ONE primary key; and in the table, this primary key can consist of single or multiple columns (fields).

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    PRIMARY KEY (ID) or CONSTRAINT PK_Person PRIMARY KEY (ID,LastName)
);
//In the example above there is only ONE PRIMARY KEY (PK_Person).
```

```
//However, the VALUE of the primary key is made up of TWO COLUMNS (ID + LastName).
ALTER TABLE Persons
ADD PRIMARY KEY (ID);

ALTER TABLE Persons
ADD CONSTRAINT PK_Person PRIMARY KEY (ID,LastName);
//If you use ALTER TABLE to add a primary key, the primary key column(s)
//must have been declared to not contain NULL values (when the table was first created)
ALTER TABLE Persons
DROP PRIMARY KEY;
```

FOREIGN KEY Constraint

The `FOREIGN KEY` constraint is used to prevent actions that would destroy links between tables.

A `FOREIGN KEY` is a field (or collection of fields) in one table, that refers to the `PRIMARY KEY` in another table. The table with the foreign key is called the child table, and the table with the primary key is called the referenced or parent table.

The `FOREIGN KEY` constraint prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the parent table.

```
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)
);

//<CONSTRAINT FK_PersonOrder> - optional, used for creating Constraint
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;
```

CHECK Constraint

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,
    City varchar(255),
    CONSTRAINT CHK_Person CHECK (Age>=18 AND City='Sandnes')
);
//Again Constraint is optional

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

ALTER TABLE Persons
DROP CHECK CHK_PersonAge;
```

DEFAULT Constraint

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'
);

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

ALTER TABLE Persons
ALTER City SET DEFAULT 'Sandnes';

ALTER TABLE Persons
ALTER City DROP DEFAULT;
```

VIEWS

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.

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

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

A view is deleted with the `DROP VIEW` statement.

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

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.

```
delimiter //
CREATE PROCEDURE male_student()
BEGIN
SELECT* FROM information WHERE gender="M";
END //

//to display
```

```
call male_student();
END //
```

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.

In MySQL there are three main data types: string, numeric, and date and time.

1. String - CHAR, VARCHAR, BINARY, VARBINARY, TINYBLOB, TINYTEXT, TEXT, BLOB, MEDIUMTEXT, MEDIUMBLOB, LONGTEXT, LONGBLOB, ENUM, SET
2. Numeric - BIT, TINYINT, BOOL, BOOLEAN, SMALLINT, MEDIUMINT, INT, INTEGER, BIGINT, FLOAT, DOUBLE, DOUBLE PRECISION, DECIMAL, DEC
3. Date and Time - DATETIME, TIMESTAMP, TIME, YEAR