**SQL Classes**

Constraints -

Allows us to impose constraints over a column or a set of columns.

**UNIQUE -**Does not allow duplicates but allows NULL values.

**NOT NULL -** Does not allow NULL values.

1. **use `DemoDB`;**
3. **drop table `Person`;**
5. **create table `Person` (**
6. **`name` char(50) unique not null,**
7. **`address` char(200),**
8. **`dob` date not null**
9. **);**
11. **insert into `Person` values('A','Addr1', '2001-03-21');**
13. **-- Insert will fail as name should be unique 'A' is already present.**
14. **insert into `Person` values('A', 'Addr2', '2001-03-22');**
16. **-- Insert will fail as name should not be null.**
17. **insert into `Person` values(null, 'Addr2', '2001-03-22');**

20. **select \* from `Person`;**

#### Identity Column -

Also called as a surrogate key, plays critical role in foreign key relationships, you can think of it as a sequence number associated with the record. Usually selected for auto\_increment.

1. **use `DemoDB`;**
3. **create table `Course` (**
4. **`id` int unique auto\_increment,**
5. **`name` varchar(100)**
6. **);**
8. **-- In the below insert commands we are providing value only for**
9. **-- the name field. Because id filed is selected for auto\_increment**
10. **-- its value is auto incremented, normally it starts with 1**
11. **-- and incremented by 1 for each new insert.**
13. **insert into `Course`(`name`) values('A');**
14. **insert into `Course`(`name`) values('B');**
16. **select \* from `Course`;**
18. **-- Adjusting the auto\_increment to the desired value.**
19. **alter table `Course` auto\_increment = 100;**

A Key uniquely identifies a record. And there may be more than one could be present in the table, they are called as candidate keys. Out of which one is selected as primary key and rest of them are considered as secondary keys.

**PRIMARY KEY  -**A combination of UNIQUE + NOT NULL. Note only one PRIMARY KEY is allowed per table.

1. **use DemoDB;**
3. **create table UserProfile(**
4. **userProfileID int primary key auto\_increment,**
5. **firstName varchar(50) not null,**
6. **lastName varchar(50) not null,**
7. **middleName varchar(50),**
8. **userName varchar(20) unique not null,**
9. **password varchar(20),**
10. **email varchar(50) unique not null,**
11. **isActive bit not null default 1,**
12. **isLocked bit not null default 0,**
13. **createdDTTM datetime not null default now(),**
14. **lastModifiedDTTM datetime**
15. **);**
17. **-- Keys**
18. **-- 1. userProfileID**
19. **-- 2. userName**
20. **-- 3. email**
21. **-- Usually identity column is selected for primary key i.e. userProfileID**
22. **-- in this case. Identity column is the one which is not impacted by the**
23. **-- change in the business rules because it is not a business column.**
24. **-- Others such as userName and email are business columns and today**
25. **-- they may be unique, but may not be at later point of time,**
26. **-- for example business rule might change to only email is unique.**
27. **-- So, in order to avoid the impact, the surrogate key**
28. **-- i.e. the identity column is normally**
29. **-- selected for primary key.**
31. **-- Full form is describe table, retrieves table description.**
33. **desc UserProfile;**

#### Referential Integrity or Foreign Key -

Foreign key allows us to refer to key column(s) of one table/relation from other table/relation. This will establish a relation between the two tables and also enforces the constraint that only existing key values are allowed to be inserted into the referring field(s) or it should be NULL.

1. **-- Select the Database.**
3. **USE `DemoDB`;**
5. **-- Create**
7. **CREATE TABLE `Dept` (**
8. **`DeptID` INT PRIMARY KEY,**
9. **`Name` VARCHAR(50),**
10. **`Location` VARCHAR(50)**
11. **);**
13. **CREATE TABLE `Employee`(**
14. **`EmpID` INT PRIMARY KEY,**
15. **`FirstName` VARCHAR(50),**
16. **`LastName` VARCHAR(50),**
17. **`DeptID` INT,**
18. **`Salary` NUMERIC(10,2),**
19. **CONSTRAINT fk\_DeptID FOREIGN KEY(`DeptID`) REFERENCES `Dept`(`DeptId`)**
20. **);**

23. **-- Sample Department Records**
25. **INSERT INTO `Dept`**
26. **VALUES(101, 'Inventory', 'Loc1');**
28. **INSERT INTO `Dept`**
29. **VALUES(102, 'Sales', 'Loc2');**
31. **-- Sample Employee Records**
33. **INSERT INTO `Employee`**
34. **VALUES(1, 'A','A', 101, 11000);**
36. **INSERT INTO `Employee`**
37. **VALUES(2, 'B','B', 102, 12000);**
39. **-- NULL value for DeptID is allowed in Employee table**
40. **-- because foreign key allows NULL values.**
41. **-- If there is a need to restrict NULL values then you**
42. **-- can apply NOT NULL constraint for DeptID column in Employee.**
44. **INSERT INTO `Employee`**
45. **VALUES(3, 'C','C', NULL, 21000);**
47. **INSERT INTO `Employee`**
48. **VALUES(4, 'D','D', 102, 22000);**
50. **-- Below insert will fail because 103 is not an existing DeptID**
51. **-- in Dept table.**

#### Relationship Cardinality -

Cardinality of relationship represents the number of entities participate in the relation. It could be categorized into three types

1. **one-to-one**

If you apply UNIQUE constraint on a FOREIGN KEY, that makes it a one-to-one relation. This is normally used for creating extension tables with additional attributes with one-to-one relation with a record in main table.

1. **one-to-many (or) many-to-one**

This is the natural cardinality achieved through simple foreign key relation. For example consider the employee and department relation where many employees work for a department. This itself if many-to-one.

1. **many-to-many**

For example consider student and course relation, a student can join many courses and a course can have many students which makes it a many-to-many relation. This relation is captured using a junction table as shown in the below example.

1. **USE `DemoDB`;**
3. **-- Tables**
5. **CREATE TABLE `Course`(**
6. **`CourseID` INT PRIMARY KEY AUTO\_INCREMENT,**
7. **`Name` VARCHAR(50) NOT NULL,**
8. **`Fee` NUMERIC(10, 2) NOT NULL**
9. **);**
11. **CREATE TABLE `Student`(**
12. **`StudentID` INT PRIMARY KEY AUTO\_INCREMENT,**
13. **`FirstName` VARCHAR(50) NOT NULL,**
14. **`LastName` VARCHAR(50) NOT NULL**
15. **);**
17. **-- A junction table to capture many-to-many relation between**
18. **-- Course and Student.**
20. **CREATE TABLE `CourseEnrollments`(**
21. **`CourseID` INT NOT NULL ,**
22. **`StudentID` INT NOT NULL,**
23. **CONSTRAINT fk\_courseid FOREIGN KEY(`CourseID`) REFERENCES `Course`(`CourseID`),**
24. **CONSTRAINT fk\_studentid FOREIGN KEY(`StudentID`) REFERENCES `Student`(`StudentID`)**
25. **);**
27. **-- Sample insertions**
29. **INSERT INTO `Course`(`Name`, `Fee`)**
30. **VALUES('SQL',30);**
32. **INSERT INTO `Course`(`Name`,`Fee`)**
33. **VALUES('DS', 40);**
35. **INSERT INTO `Student`(`FirstName`,`LastName`)**
36. **VALUES('a','a');**
38. **INSERT INTO `Student`(`FirstName`,`LastName`)**
39. **VALUES('b','b');**

42. **INSERT INTO `CourseEnrollments`**
43. **VALUES(1,1);**
45. **INSERT INTO `CourseEnrollments`**
46. **VALUES(1,2);**
48. **INSERT INTO `CourseEnrollments`**
49. **VALUES(2,1);**
51. **-- Query**
53. **SELECT \* FROM `Student`;**
54. **SELECT \* FROM `Course`;**
55. **SELECT \* FROM `CourseEnrollments`;**
57. **-- Drop CourseEnrollments before dropping Course and Student.**
59. **DROP TABLE `CourseEnrollments`;**
60. **DROP TABLE `Course`;**
61. **DROP TABLE `Student`;**

#### Aggregate Functions -

**sum() -**to calculate the sum of values.

**min() -**to find the minimum value.

**max() -**for getting the maximum value.

**avg() -**for getting the average value.

**count() -**for counting the records.

1. **-- Select the Database.**
3. **USE `DemoDB`;**
5. **-- Drop the existing table if any and continue.**
7. **-- Create**
9. **CREATE TABLE `Student` (**
10. **`StudentID` INT PRIMARY KEY AUTO\_INCREMENT,**
11. **`Name` VARCHAR(50),**
12. **`Course` VARCHAR(10),**
13. **`Score` NUMERIC(5,2)**
14. **);**
16. **-- Insert Sample Records**
18. **INSERT INTO `Student`(`Name`, `Course`, `Score`)**
19. **VALUES('A', 'CS', 80);**
21. **INSERT INTO `Student`(`Name`, `Course`, `Score`)**
22. **VALUES('B', 'CS', 60);**
24. **INSERT INTO `Student`(`Name`, `Course`, `Score`)**
25. **VALUES('C', 'IT', 70);**
27. **INSERT INTO `Student`(`Name`, `Course`, `Score`)**
28. **VALUES('D', 'IT', 85);**
30. **INSERT INTO `Student`(`Name`, `Course`, `Score`)**
31. **VALUES('E', 'ECE', 88);**

34. **-- Retrieve all records**
36. **SELECT \* FROM `Student`;**

39. **-- Retrieves the total score and displays the resultant**
40. **-- column as Total which is called as an alias name.**
42. **SELECT SUM(`Score`) AS `Total` FROM `Student`;**
44. **-- Display the minimum score**
46. **SELECT MIN(`Score`) AS `Minimum` FROM `Student`;**
48. **-- Display the maximum score**
50. **SELECT MAX(`Score`) AS `Maximum` FROM `Student`;**
52. **-- Display the average score**
54. **SELECT AVG(`Score`) AS `Average` FROM `Student`;**
56. **-- Count(\*) returns the record count, because here it is entire Student table**
57. **-- you will get total student count.**
59. **SELECT COUNT(\*) AS `TotalStudents` FROM `Student`;**
61. **-- Retrieves the record count for which the Score is greater than 80.**
63. **SELECT COUNT(\*) AS `TopScorers` FROM `Student`**
64. **WHERE `Score` > 80;**

#### Using GROUP BY in Query

Group by helps us to group records in the table based on the selected fields. The equal values for the selected fields will form a group. And if the aggregate functions such as SUM, MIN, MAX etc. are used along with GROUP BY they perform the corresponding operation on each group instead of entire table.

Note that while using GROUP BY, the display column list can only contain the grouped columns along with aggregate operations or pseudo columns such as direct values.

1. **-- Select the Database.**
3. **USE `DemoDB`;**
5. **-- Drop the existing Student table if any.**
7. **-- Create**
9. **CREATE TABLE `Student` (**
10. **`StudentID` INT PRIMARY KEY AUTO\_INCREMENT,**
11. **`Name` VARCHAR(50),**
12. **`Course` VARCHAR(10),**
13. **`Score` NUMERIC(5,2)**
14. **);**
16. **-- Sample Data**
18. **INSERT INTO `Student`(`Name`, `Course`, `Score`)**
19. **VALUES('A', 'CS', 80);**
21. **INSERT INTO `Student`(`Name`, `Course`, `Score`)**
22. **VALUES('B', 'CS', 60);**
24. **INSERT INTO `Student`(`Name`, `Course`, `Score`)**
25. **VALUES('C', 'IT', 70);**
27. **INSERT INTO `Student`(`Name`, `Course`, `Score`)**
28. **VALUES('D', 'IT', 85);**
30. **INSERT INTO `Student`(`Name`, `Course`, `Score`)**
31. **VALUES('E', 'ECE', 88);**
33. **-- Basic Query**
35. **SELECT \* FROM `Student`;**
37. **-- Get total score here SUM operates on entire table i.e. all scores are added.**
39. **SELECT SUM(`Score`) AS `Total Score` FROM `Student`;**
41. **-- But when used with GROUP BY these operations are applied**
42. **-- for each group(all equal values for selected attributes e.g. Course here**
43. **-- will form a group).**
44. **-- Displays the Total Score, Average Score, Minimum, Maximum score for each Course.**
46. **SELECT**
47. **`Course`,**
48. **SUM(`Score`) AS `Total Score`,**
49. **AVG(`Score`) AS `Average`,**
50. **MIN(`Score`) AS `Minimum`,**
51. **MAX(`Score`) AS `Maximum`**
52. **FROM `Student`**
53. **GROUP BY `Course`;**

56. **-- Below query is invalid, as non group by column Name is used in the display**
57. **-- column list.**
59. **SELECT**
60. **`Name`,**
61. **`Course`,**
62. **SUM(`Score`) AS `Total Score`**
63. **FROM `Student`**
64. **GROUP BY `Course`;**

#### HAVING clause -

HAVING clause allows us to further filter the records obtained as a result of GROUP BY and helps us further filter the groups.

1. **-- With out HAVING clause the below select command displays all the courses along with**
2. **-- the selected aggregated results.**
4. **-- With HAVING only those groups are selected with match the given condition**
5. **-- i.e. AVG(`Score`) > 75 in this case.**
7. **-- Display the Total Score, Average Score, Minimum, Maximum score for each group**
8. **-- whose Average score is greater than 75.**
10. **SELECT**
11. **`Course`,**
12. **SUM(`Score`) AS `Total Score`,**
13. **AVG(`Score`) AS `Average`,**
14. **MIN(`Score`) AS `Minimum`,**
15. **MAX(`Score`) AS `Maximum`**
16. **FROM `Student`**
17. **GROUP BY `Course`**
18. **HAVING AVG(`Score`) > 75;**

21. **-- Display the groups which are having 2 or more students.**
23. **-- COUNT(\*) returns the number of records present in that group**
24. **-- and >= 2 ensures that there are at least there are two or more records.**
26. **SELECT `Course` FROM `Student`**
27. **GROUP BY `Course`**
28. **HAVING COUNT(\*) >= 2;**

#### Cartesian Product -

Consider two tables A(X, Y) and B(Y, Z) and

A = { (a, 1), (c, 3) }  and B = { (1, p), (2, q) }

A x B = { ((a, 1), (1, p)) , ((a, 1), (2, q)), ((c, 3), (1, p)), ((c, 3), (2, q)) }

The above will be the result of the command select \* from A, B;

Now in order to filter only those records with match the Y attribute value i.e. A.Y  = B.Y

select \* from A, B where A.Y = B.Y;

Then this picks only one record from the above.

{ ((a, 1), (1, p)) }

This is referred to as **CROSS JOIN** and A.Y = B.Y is referred to as**JOIN CONDITION**.

1. -- Select the Database.
3. USE `DemoDB`;
5. -- Create
7. CREATE TABLE `Dept`(
8. `DeptID` INT PRIMARY KEY,
9. `Name` VARCHAR(50)
10. );
12. CREATE TABLE `Employee`(
13. `EmpID` INT PRIMARY KEY,
14. `Name` VARCHAR(50),
15. `DeptID` INT REFERENCES Dept
16. );
18. -- Sample records
19. -- Department Records
21. INSERT INTO `Dept`
22. VALUES(101, 'Inventory');
24. INSERT INTO `Dept`
25. VALUES(102, 'Sales');
27. -- Employee Records
29. INSERT INTO `Employee`
30. VALUES(1, 'A', 101);
32. INSERT INTO `Employee`
33. VALUES(2, 'B', 102);
35. INSERT INTO `Employee`
36. VALUES(3, 'C', NULL);
38. -- Individual table queries.
40. SELECT \* FROM `Dept`;
41. SELECT \* FROM `Employee`;

44. -- Cartesian product
46. SELECT \* FROM `Employee`,`Dept`;
48. -- Cross Join to pick only those records whose DeptID matches.
50. SELECT \* FROM `Employee`, `Dept`
51. WHERE `Employee`.`DeptID` = `Dept`.`DeptID`;

54. -- Query with alias
56. SELECT \* FROM `Employee` as e, `Dept` as d
57. WHERE e.`DeptID` = d.`DeptID`;

60. -- Same is achieved through INNER JOIN

63. SELECT \* FROM `Employee` as e
64. INNER JOIN `Dept` as d ON (e.`DeptID` = d.`DeptID`);

#### JOIN -

Is the process of combining the information from multiple tables based on some condition( i.e. join condition).

Consider two tables A(X, Y) and B(Y, Z) and  JOINED over Y on equality of Y i.e. A.Y = B.Y,

A = { (a, 1), (c, 3) }  and B = { (1, p), (2, q) }

**INNER JOIN** -

Picking only those records which match the JOIN condition is called INNER JOIN.

**A** INNER JOIN **B**=> bring all the records from A and B which match the JOIN CONDITION.

i.e. { ((a, 1), (1, p)) }

But there may be some records which might not be selected as they did not satisfy the JOIN condition what about them? In order bring those records we need to use OUTER JOIN.

**OUTER JOIN -**

Includes those records which did not meet the JOIN condition.

**A** LEFT OUTER JOIN **B**=> bring all the records from A even there is no matching records in B

i.e. { ((a, 1), (1, p)), ((c, 3), (null, null)) }

**A** RIGHT OUTER JOIN **B**=> bring all the records from B even there is no matching records in A

i.e. { ((a, 1), (1, p)), ((null, null), (2, q)) }

**A** FULL OUTER JOIN **B**=> bring all the records from both A and B even when the condition did not match. It is nothing but the UNION Of the above two.

i.e. { ((a, 1), (1, p)), ((c, 3), (null, null)), ((null, null), (2, q)) }

1. **-- Inner join - Retrieves only matching Employee and Dept records**
3. **SELECT \* FROM `Employee` as e**
4. **INNER JOIN `Dept` as d ON (e.DeptID = d.DeptID);**
6. **-- Outer Join**
8. **-- 1. Left outer Join**
9. **-- Retrieves all employees irrespective of whether they have a matching**
10. **-- department record or not.**
12. **SELECT \* FROM `Employee` as e**
13. **LEFT JOIN `Dept` as d ON (e.DeptID = d.DeptID);**

16. **-- 2. Right outer Join**
17. **-- Retrieves all department records irrespective of whether they have a matching**
18. **-- department record or not.**
20. **SELECT \* FROM `Employee` as e**
21. **RIGHT JOIN `Dept` as d ON (e.DeptID = d.DeptID);**
23. **-- 3. Full outer Join**
25. **-- Below query is valid in MS Sql Server; but not in MySql**
26. **-- SELECT \* FROM Employee as e**
27. **-- FULL JOIN Dept as d ON (e.DeptID = d.DeptID);**
29. **-- Full Join Simulation in MySQL**
31. **SELECT \* FROM `Employee` as e**
32. **LEFT JOIN `Dept` as d ON (e.DeptID = d.DeptID)**
34. **UNION**
36. **SELECT \* FROM `Employee` as e**
37. **RIGHT JOIN `Dept` as d ON (e.DeptID = d.DeptID);**

#### Subqueries -

Subquery is a query written inside another query. The outcome of the inner query is used as an input to the outer query.

1. **-- Select the Database.**
3. **USE `DemoDB`;**
5. **-- Drop the existing tables if any..**
7. **-- Create**
9. **CREATE TABLE `Dept`(**
10. **`DeptID` INT PRIMARY KEY AUTO\_INCREMENT,**
11. **`Name` VARCHAR(50) NOT NULL,**
12. **`Location` VARCHAR(50) NOT NULL**
13. **);**
15. **CREATE TABLE `Employee` (**
16. **`EmpID` INT PRIMARY KEY AUTO\_INCREMENT,**
17. **`FirstName` VARCHAR(50) NOT NULL,**
18. **`LastName` VARCHAR(50) NOT NULL,**
19. **`DeptID` INT REFERENCES Dept,**
20. **`Salary` NUMERIC(10,2) NOT NULL CHECK (Salary > 0)**
21. **);**
23. **-- Sample Records**
25. **-- Here DeptID is autogenerated you can see it is selected**
26. **-- for auto\_increment.**
28. **INSERT INTO `Dept`(`Name`,`Location`)**
29. **VALUES('Inventory', 'Loc1');**
31. **INSERT INTO `Dept`(`Name`,`Location`)**
32. **VALUES('Sales', 'Loc2');**
34. **INSERT INTO `Dept`(`Name`,`Location`)**
35. **VALUES('HR', 'Loc2');**
37. **-- Below insert command is very useful in SQL in Project**
38. **-- When you are inserting master data and if you do not know**
39. **-- the generated ID then the query could be used to get the**
40. **-- generated ID and given as input to the insert.**
41. **-- Here the output of the select command is given as input for**
42. **-- DeptID column value.**
43. **-- NOTE - You need to make sure that it only retrieves one value**
44. **-- if it retrieves multiple values then it lead to failure.**
46. **INSERT INTO `Employee`(`FirstName`, `LastName`, `DeptID`, `Salary`)**
47. **VALUES('A','A', (SELECT `DeptID` FROM `Dept` WHERE `Name`='Inventory'), 11000);**
49. **INSERT INTO `Employee`(`FirstName`, `LastName`, `DeptID`, `Salary`)**
50. **VALUES('B','B', (SELECT `DeptID` FROM `Dept` WHERE `Name`='Sales'), 12000);**
52. **INSERT INTO `Employee`(`FirstName`, `LastName`, `DeptID`, `Salary`)**
53. **VALUES('C','C', (SELECT `DeptID` FROM `Dept` WHERE `Name`='HR'), 21000);**
55. **INSERT INTO `Employee`(`FirstName`, `LastName`, `DeptID`, `Salary`)**
56. **VALUES('D','D', (SELECT `DeptID` FROM `Dept` WHERE `Name`='HR'), 22000);**
58. **-- Basic queries**
60. **SELECT \* FROM `Dept`;**
61. **SELECT \* FROM `Employee`;**
63. **-- Display employees who work for HR department**
64. **-- Here the sub query returns the department ID for the HR**
65. **-- which is then used as input for retrieving employee**
66. **-- records for that department.**
68. **SELECT \* FROM `Employee`**
69. **WHERE `DeptID` = (**
70. **SELECT `DeptID` FROM `Dept`**
71. **WHERE `Name` = 'HR' );**
73. **-- Display employees whose department location is 'Loc2'**
75. **SELECT \* FROM `Employee`**
76. **WHERE `DeptID` IN (**
77. **SELECT `DeptID` FROM `Dept`**
78. **WHERE `Location` = 'Loc2' );**

#### Correlated Subquery -

A subquery is said to be correlated if it refers to the columns of the outer query.

1. **USE `DemoDB`;**
3. **-- Normal sub query and if you observe the subquery output is static**
4. **-- i.e. it just returns the HR department ID and the same value could**
5. **-- be used for the entire outer query.**
6. **-- it is almost like select \* from Employee where DeptID = 3;**
8. **select \* from Employee where DeptId = (select DeptId from Dept where Name = 'HR');**
10. **-- Translation of above query using correlated subquery.**
12. **-- In case of this query the reference to e.DeptId is made which is an outer**
13. **-- query column. And this column value changes for each record in the Employee table**
14. **-- and hence the subquery need to be evaluated for each record. And this is slightly**
15. **-- in efficient.**
17. **-- Here it the inner query check for the existence of a record with employees departmentId**
18. **-- whose name is HR. If record exists then Employee record is selected otherwise not.**
20. **select \* from Employee as e**
21. **where**
22. **exists (select \* from Dept where DeptId = e.DeptId and Name = 'HR');**
24. **-- Translation using JOIN.**
25. **-- Most of the times JOIN might turns out to be a better option.**
27. **select e.\* from Employee as e**
28. **INNER JOIN Dept as d ON (e.DeptId = d.DeptId and d.Name = 'HR');**