Database

- Database is a software which stores the data in an organized form.

- Database examples are : MySQL, Oracle, PostgreSQL, SQL Server etc
- Note : SQL is a query language which is used to manipulate database
  (SQL is not a database)

- Download : Download from "https://www.mysql.com/downloads/"

=> Type of SQL Commands :-

1. DDL (Data Defination Language) :- Create, Alter, Drop, Rename, Truncate, Comment

2. DML (Data Maniplulation Language) :- Insert, Delete, Update, Merge, Call

3. DQL (Data Query Language) :- Select

4. TCC (Transaction Control Commands) :- Commit, Rollback, Savepoint

5. DCL (Data Control Language) :- Alter password, Grant Access, Revoke Access etc

6. DAC (Data Administration Commands) :- Start Audit, Stop Audit etc

=> MySQL Commands :-

1. CREATE DATABASE datebase_name; (it will create new database)

2. SHOW DATABASES; (it will display all the databases)

3. DROP DATABASE database_name; (it will delete the provided database)

4. USE database_name; (it will select the provided database)

5. CREATE TABLE table_name(column_name data-type(size), column_name data-type(size), - , -); (it will create table with provided column names)

6. SHOW TABLES; (it will display all the tables in selected database)

7. DESC tablename; (it will display the table details)

8. ALTER TABLE tablename ADD column_name data-type(size); (it will add new column in the table)

9. ALTER TABLE tablename MODIFY COLUMN column_name data-type(size); (it will modify the column)

10. ALTER TABLE tablename DROP COLUMN column_name; (it will delete the provided column)

11. DROP TABLE tablename; (it will delete the table)

12. INSERT INTO tablename VALUES('-', '-', '-', '-', '-');

13. INSERT INTO tablename(column_name, column_name, column_name) VALUES('-', '-', '- ');

14. SELECT * FROM tablename;

15. SELECT column_name, column_name FROM tablename;

16. SELECT * FROM tablename WHERE column_name='value';

17. SELECT * FROM tablename WHERE column_name='value' AND/OR column_name='value';

18. SELECT column_name, column_name FROM tablename WHERE column_name='value' AND/OR column_name='value';

19. UPDATE tablename SET column_name='value', column_name='value' WHERE column_name='value';

20. DELETE FROM tablename WHERE column_name='value';

List of Constraints in MySQL

- NOT NULL
- UNIQUE
- DEFAULT
- CHECK
- FOREIGN KEY
- PRIMARY KEY

Data Table without Constraints

Id	Name	Age	Gender	Phone	City
1	Ram Kumar	17	Male	4022155	Agra
2	Salman Khan	19		4033244	Agra
3	Meera Khan	20	Female	4022155	Agra
	Sarita Kumari	18	Female	4066899	Agra
5	Anil Kapoor	19	Male	4188733	Agra

NOT NULL
UNIQUE

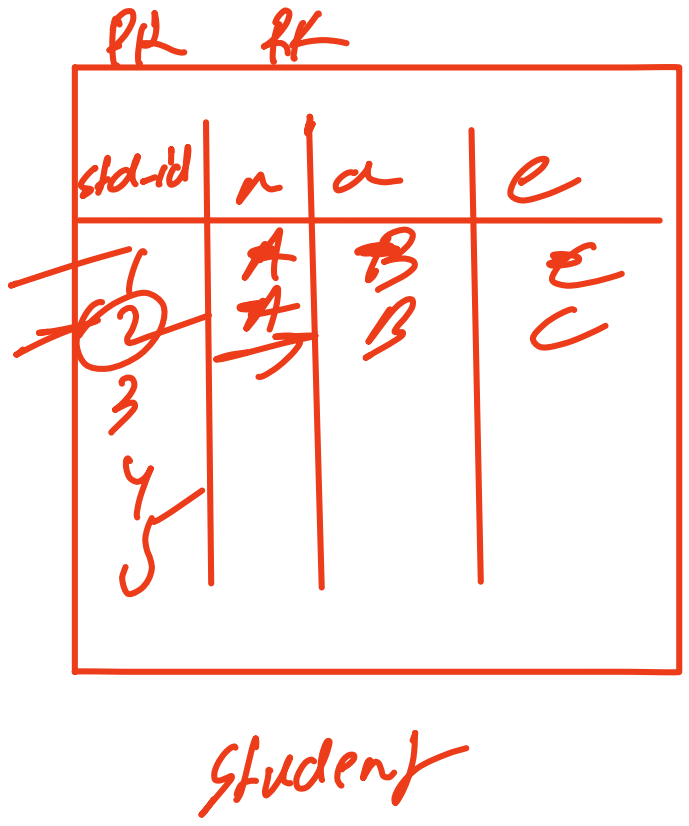
CHECK (age >= 18)
(only that data will be added whose age is greater than 18)

NOT NULL

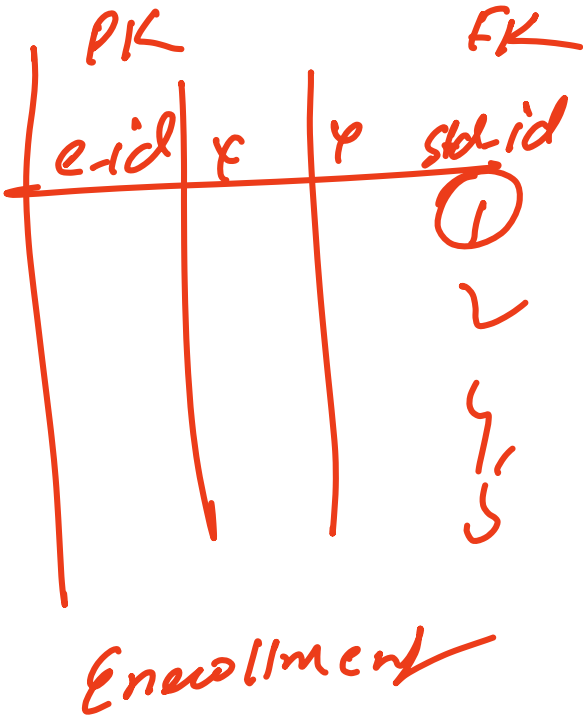
UNIQUE

DEFAULT 'Agra' -> if no value is provided, take agra as default

```
CREATE TABLE table_name (
    id INT NOT NULL UNIQUE,
    name VARCHAR(50) NOT NULL,
    age INT NOT NULL CHECK (age >= 18),
    gender VARCHAR(10) NOT NULL,
    phone VARCHAR(10) NOT NULL UNIQUE,
    city VARCHAR(10) NOT NULL DEFAULT 'Agra',
);
```



Student



Enrollment

```sql
CREATE TABLE personal(
    id INT NOT NULL UNIQUE,
    name VARCHAR(50) NOT NULL,
    age INT NOT NULL CHECK (age >= 18),
    gender VARCHAR(1) NOT NULL,
    phone VARCHAR(10) NOT NULL UNIQUE,
    city VARCHAR(15) NOT NULL DEFAULT 'Agra'
);
```

* If No constraints are provided

```sql
INSERT INTO personal(id, name, age, gender, phone, city)
VALUES
(1 , "Ram Kumar", 19, "M", "4022155", "Agra");

(2 , "Sarita", 18, "4015155", "Agra");
```

-> give warning since no default value for gender given but data still added in the table if NOT NULL constraint is not used

```sql
(3 , "Salman Khan", 20, "4015155");
```

-> set default value agra in city if default constraint is used

Primary Key:

In MySQL, a primary key is a column or a set of columns in a database table that uniquely identifies each row in that table

-Each value in the primary key column(s) must be unique within the table.

-Primary key columns cannot contain NULL values.

Syntax :

The following are the syntax used to create a primary key in MySQL.

-If we want to create only one primary key column into the table, use the below syntax:

```
CREATE TABLE table_name( col1 datatype PRIMARY KEY, col2 datatype, ... );
```

-If we want to create more than one primary key column into the table, use the below syntax:

```
CREATE TABLE table_name ( col1 col_definition, col2 col_definition, ...
CONSTRAINT [constraint_name] PRIMARY KEY (column_name(s)) );
```

Example:

```
CREATE TABLE Student
( StudentID VARCHAR(10) PRIMARY KEY, FirstName VARCHAR(25) NOT NULL, LastName VARCHAR(25)
NOT NULL, DateOfBirth DateTime NOT NULL, Gender VARCHAR(25) NOT NULL, Email VARCHAR(30)
UNIQUE NOT NULL, Phone VARCHAR(25) NOT NULL );
```

OR

```
CREATE TABLE Test(first_name VARCHAR(20), last_name VARCHAR(20), age INT, email
VARCHAR(40) ,
state VARCHAR(20) , constraint primary_key PRIMARY KEY(first_name, last_name)) ;
DESC Test ;

INSERT INTO Test VALUES('Arun', 'Shamrma', 28, 'arunkumarkv29@gmail.com', 'Delhi')  ;
INSERT INTO Test VALUES('sourav', 'kumar', 30, 'souravkumar@gmail.com', 'hyderabad')  ;
INSERT INTO Test VALUES('Arun', 'kumar', 30, 'souravkumar@gmail.com', 'hyderabad')  ;
INSERT INTO Test VALUES('Arun', 'Sharma', 30, 'souravkumar@gmail.com', 'hyderabad')  ;

SELECT * FROM Test ;
```

```
CREATE TABLE Std( id INT PK,
first_name VARCHAR(20) NOTNULL,
- - - - /

ALTER TABLE Std
ADD PK std-id

DESC Std
```

1. **Uniqueness** - A primary key ensures that each record in a table is unique. No two rows can have the same primary key value.

2. **Not Null** - A primary key column cannot contain NULL values. Every record must have a value for the primary key.

3. **Single or Composite Key** - A primary key can be a single column or a combination of multiple columns (composite key) to uniquely identify a record.

4. **Automatically Indexed** - Primary keys are automatically indexed in most databases, improving search performance.

5. **One per Table** - Each table can have only one primary key, though it may consist of multiple columns (composite key).

6. **Cannot be Changed Easily** - Once assigned, a primary key should not be changed frequently because it acts as the unique identifier for a record.

7. **Used for Relationships** - A primary key is used to establish relationships between tables. It is referenced as a foreign key in other tables.

8. **Enforced by the Database** - The database automatically enforces the uniqueness and not null constraints of a primary key.

```sql
CREATE TABLE Students (
    student_id INT PRIMARY KEY,  -- Unique and Not Null
    name VARCHAR(50),
    age INT
);
```

```sql
CREATE TABLE Orders (
    order_id INT,
    product_id INT,
    quantity INT,
    PRIMARY KEY (order_id, product_id)  -- Composite Key
);
```

Primary Key Using ALTER TABLE Statement:

This statement allows us to do the modification into the existing table. When the table does not have a primary key, this statement is used to add the primary key to the column of an existing table.

Syntax: Following are the syntax of the ALTER TABLE statement to create a primary key in MySQL:

```
-ALTER TABLE table_name
ADD PRIMARY KEY(column_list);
```

Example :The following statement creates a table "Person" that has no primary key column into the table definition.

```
-CREATE TABLE Person(Person_ID int, Name varchar(45), Age int,
City varchar(25) );
```

After creating a table, if we want to add a primary key to this table, we need to execute the ALTER TABLE statement as below:

```
-ALTER TABLE Persons
ADD PRIMARY KEY(Person_ID);
```

DROP Primary Key:

The ALTER TABLE statement also allows us to drop the primary key from the table. The following syntax is used to drop the primary key:

```
ALTER TABLE table_name
DROP PRIMARY KEY;
```

Example:

```
ALTER TABLE Person
        DROP PRIMARY KEY;
```

Clause:

WHERE Clause: Filters the rows returned by a query based on a specified condition.
-Select * from student where StudentId='S101' ;

ORDER BY Clause: Specifies the sorting order for the result set.
Example-Sort in ascending order:
-SELECT StudentID, FirstName,LastName from student ORDER BY StudentID;
Example-Sort in descending order:
-SELECT StudentID, FirstName,LastName from student ORDER BY StudentID desc;

LIMIT (or TOP) Clause: Restricts the number of rows returned by a query.
Example : **Select * from student limit 2;**

==========

UPDATE query:

```
# SET SQL_SAFE_UPDATES = 0;
 -UPDATE table_name
  SET column1 = value1, column2 = value2, ...
  WHERE condition;
```

Update All Records in a Table:

```
-UPDATE employee
 SET salary=10000;
```

Update a Specific Record by Matching a Column Value:

```
-UPDATE employee
 SET salary=20000
 WHERE eid=3;
```

==========

Aggregate Functions:

1. COUNT() : Counts the number of rows in a specified column or the number of rows that match a condition.
-Let's explore with an example: Count all records in a table.

```
==> SELECT count(*) FROM employee;
==> SELECT COUNT(*) AS count_of_employee FROM employee ;
==> SELECT count(*) FROM employee WHERE salary>30000;
```

2. SUM() • Calculates the sum of values in a numeric column.
Let's explore with an example: Calculate the total salary of all employees.

```
==> SELECT sum(salary) FROM employee;
```

3. AVG() • Computes the average (mean) of values in a numeric column.
Let's explore with an example: Calculate the average salary of emploee.

```
==> SELECT avg(salary) FROM employee;
```

4. MAX() ● Retrieves the maximum value from a column.
Let's explore with an example: Find the highest salary among all employees.
==> SELECT max(salary) FROM employee;

5. MIN() ● Retrieves the minimum value from a column.
Let's explore with an example: Find the lowest salary among all employees. ==>
==>SELECT min(salary) FROM employee;

Foreign Key In MySQL:

-A foreign key is a column or a set of columns that establishes a link between two tables in a relational database.
-The foreign key in one table references the primary key of another table, creating a relationship between them.

Syntax for Defining a Foreign Key:

To create a foreign key constraint when defining a table, you can use the following syntax:
CREATE TABLE child_table (... foreign_key_column data_type, ... FOREIGN KEY (foreign_key_column) REFERENCES parent_table(primary_key_column) ...);

Joining Table : A "joining table" in a database is often referred to as a "junction table" or an "association table."
- It's a table used in a relational database to manage many-to-many relationships between two other tables
-The joining table contains foreign keys from both of the related tables.
For example, if you have a database for a student management system, you might have a "Course" table and a "Instructor" table. Since one Instructor can teach many courses, and many courses are taught by one Instructor, you would use a joining table to link them together, indicating which Instructor is associated with which Courses.

Here's a simple representation of such tables:

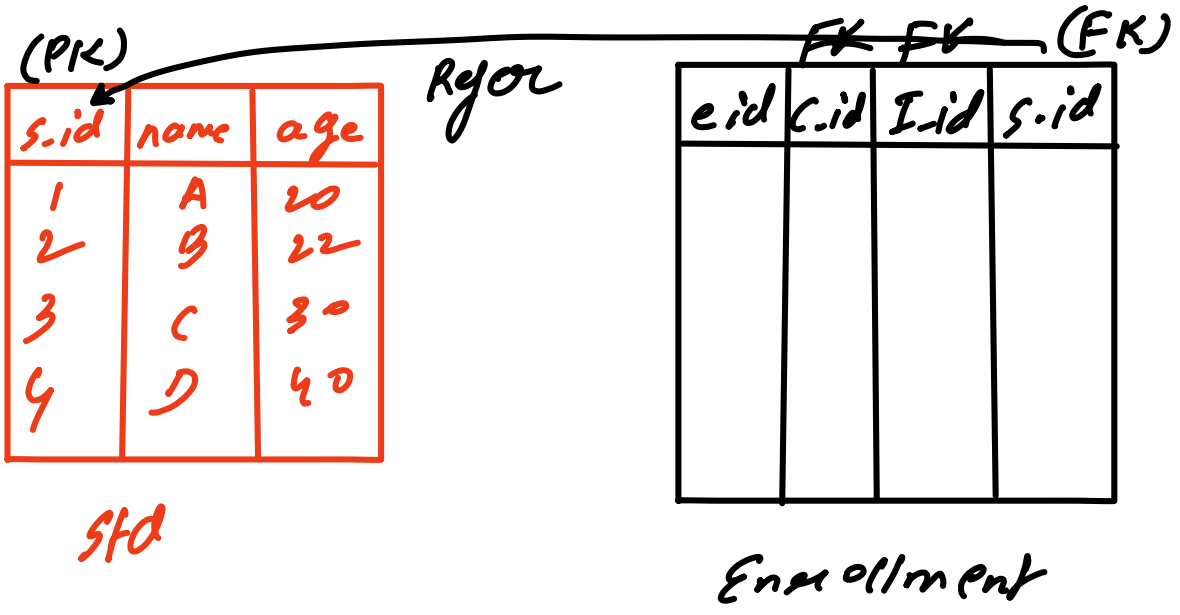
Student
- Attributes: StudentID (Primary Key) FirstName LastName DateOfBirth Gender Email Phone
- Relationships: One Student can enroll in more than one Course (One-to-Many) Course

1 : M

Course
- Attributes: CourseID (Primary Key) CourseTitle Credits
- Relationships: Many Course is taught by one Instructor (Many-to-One)

M : 1

#



CREATE TABLE enrollment

(e_id INT PK, s_id INT, c_id INT,

FOREIGN Key (s_id) references students (s_id)

1 : 1 ✗

1 : M
M : 1
} optional

M : M ⇒ junction

Instructor

- Attributes: InstructorID (Primary Key) FirstName LastName Email
- Relationships: One Instructor teaches many Courses (One-to-Many) One Instructor has many Students (One-to-Many) Enrollment

Enrollment

- Attributes: EnrollmentID (Primary Key) EnrollmentDate StudentID(Foreign key) CourseID(Foreign Key) InstructorID(Foreign key)
- Relationships: One Student maps to Many Enrolment ids (One-to-Many) Many Enrolment ids map to one Course (Many-to-One)

Score

- Attributes: ScoreID (Primary Key) CourseID (Foreign key) StudentID (Foreign Key) DateOfExam CreditObtained
- Relationships: Many ScoreIDs will map to one Student (Many-to-one) Many ScoresIDs will map to one Course (Many-to-one)

Feedback

- Attributes: FeedbackID (Primary Key) StudentID (Foreign key) Date InstructorName Feedback
- Relationships: One Student will maps to Many Feedbacks (One-to-Many)

Instructor Table:

```
CREATE TABLE Instructor ( InstructorID VARCHAR(10) PRIMARY KEY, Email VARCHAR(30) UNIQUE NOT NULL, FirstName VARCHAR(30) NOT NULL, LastName VARCHAR(30) );

DESC Instructor;

INSERT INTO Instructor
(InstructorID ,Email,FirstName,LastName) VALUES
('I101','sunil@example.com','Sunil','Rawat'),
('I102','nida@example.com','Nida','Fatima'),
('I103','shiv@example.com','Shiv','Kumar');

SELECT * FROM Instructor;
```

Course Table :

```
CREATE TABLE Course ( CourseID VARCHAR(10) PRIMARY KEY, CourseTitle VARCHAR(30) NOT NULL, Credits INT NOT NULL );

DESC Course;

INSERT INTO Course (CourseID,CourseTitle,Credits) VALUES ('C101','Math101',12), ('C102','History101',13), ('C103','Computer Science101',11);

SELECT * FROM Course;
```

Enrollment Table: (Junction table)

We are creating a third table as a junction table to represent the relationship between the two related tables.This table typically contains three columns that serve as foreign keys, each referencing a primary key from the related tables.

For our example, here we have created a "Enrollment" junction table with columns "StudentID" and "CourseID" and "InstructorID" along with EnrollmentId and EnrollmentDate.

```
CREATE TABLE Enrollment ( EnrollmentID VARCHAR(10) PRIMARY KEY, StudentID VARCHAR(10) NOT NULL, CourseID VARCHAR(10) NOT NULL, InstructorID VARCHAR(10) NOT NULL,
FOREIGN KEY (StudentID) REFERENCES Student(StudentID),
FOREIGN KEY (CourseID) REFERENCES Course(CourseID),
FOREIGN KEY (InstructorID) REFERENCES Instructor(InstructorID) );

desc Enrollment;

INSERT INTO Enrollment (EnrollmentID,StudentID, CourseID,InstructorID) VALUES ('E1001','S101','C101','I101'), ('E1002','S102','C101', 'I101'), ('E1003','S103','C102','I102');

SELECT * FROM Enrollment;
```

Score Table :

```
CREATE TABLE Score( ScoreID VARCHAR(10) PRIMARY KEY, StudentID VARCHAR(10) NOT NULL, CourseID VARCHAR(10) NOT NULL, FOREIGN KEY (StudentID) REFERENCES Student(StudentID), FOREIGN KEY (CourseID) REFERENCES Course(CourseID), CreditObtained VARCHAR(10) NOT NULL, DateOfExam DateTime NOT NULL );

DESC Score;

INSERT INTO Score
(ScoreID,StudentID,CourseID,CreditObtained,DateOfExam)VALUES
('SC101','S101','C101','12','2022-10-10'),
('SC102','S102','C101','10','2022-10-10'),
('SC103','S104','C102',11,'2023-09-10');

SELECT * FROM Score;
```

Student table:

```
CREATE TABLE Student ( StudentID VARCHAR(10) PRIMARY KEY, FirstName VARCHAR(25) NOT NULL, LastName VARCHAR(25) NOT NULL,
DateOfBirth Date NOT NULL, Gender VARCHAR(25) NOT NULL, Email VARCHAR(30) UNIQUE NOT NULL, Phone VARCHAR(25) NOT NULL );

INSERT INTO Student (StudentID, FirstName,LastName, DateOfBirth, Gender,  Email,Phone) VALUES
('S101','John', 'Doe','2000-10-10','M', 'john@example.com','9878457945'),
('S102','Jane', 'Smith','2013-08-08','M', 'jane@example.com','9977457745'),
('S103','Alice', 'Johnson','2011-09-08','F', 'alice@example.com','9876457845'),
('S104','Jim', 'Doe','2011-07-08','F', 'jim.doe@india.com','9876457845'),
('S105','Peter', 'Parker','2011-06-05','F', 'p_parker@example.com','9876457845') ;

SELECT * FROM Student ;
```

Joins :

SQL JOIN is a SQL operation used to combine rows from two or more tables based on a related column between them.

Types of Joins:

1. INNER JOIN: This type of join returns only the rows that have matching values in both tables. Rows from the tables that don't have a match in the other table are excluded from the result set.

```
SELECT * FROM table1
INNER JOIN table2 ON table1.column_name = table2.column_name;
```

```
SELECT Student.StudentId, Course.CourseTitle,Score.CreditObtained FROM Score
INNER JOIN Course ON Course.CourseId=Score.CourseId
INNER JOIN Student ON Student.StudentId=Score.StudentId;
```

2. LEFT JOIN (or LEFT OUTER JOIN): This type of join returns all the rows from the left table (table1), and the matched rows from the right table (table2). If there's no match in the right table, NULL values are returned for the columns from the right table.

```
SELECT * FROM table1
LEFT JOIN table2 ON table1.column_name = table2.column_name;
```

```
SELECT * FROM Course
LEFT JOIN Score ON Score.CourseId=Course.CourseId;
```

3. RIGHT JOIN (or RIGHT OUTER JOIN): This is the opposite of a LEFT JOIN. It returns all the rows from the right table and the matched rows from the left table. If there's no match in the left table, NULL values are returned for the columns from the left table.

```
SELECT * FROM table1
RIGHT JOIN table2 ON table1.column_name = table2.column_name;
```

```
SELECT * FROM Score
RIGHT JOIN Course ON Score.CourseId=Course.CourseId;
```

4. SELF JOIN: This is used to join a table with itself. It can be useful when you have hierarchical or recursive data structures within a single table.

```
CREATE TABLE Customer (customer_id INT, name VARCHAR(50), Address VARCHAR(50)) ;
INSERT INTO Customer VALUES
(101, 'saurabh', 'abc'),
(102, 'anil', 'def'),
(103, 'aparana', 'xyz');

SELECT * FROM Customer ;

CREATE TABLE Payment (payment_id INT, customer_id INT, amount VARCHAR(50),
modeofpayment VARCHAR(50)) ;
INSERT INTO Payment VALUES
(1, 101, '5000', 'Debit Card'),
(1, 102, '3000', 'Credit Card');

INSERT INTO Payment VALUES
(5, 108, '1000', 'UPI'),
(6, 107, '2000', 'Cash');

SELECT * FROM Payment ;

SELECT * FROM Customer
INNER JOIN Payment
ON Customer.customer_id = Payment.customer_id ;

SELECT * FROM Customer  AS c
INNER JOIN Payment AS p
ON c.customer_id = p.customer_id ;

SELECT * FROM Customer
LEFT JOIN Payment
ON Customer.customer_id = Payment.customer_id ;

SELECT * FROM Customer AS c
LEFT JOIN Payment AS p
ON c.customer_id = p.customer_id ;

SELECT * FROM Customer
RIGHT JOIN Payment
ON Customer.customer_id = Payment.customer_id ;

SELECT * FROM Customer AS c
RIGHT JOIN Payment AS p
ON c.customer_id = p.customer_id ;
```

```
CREATE TABLE EmployeeTable (emp_id INT PRIMARY KEY, name
VARCHAR(50), manager_id INT) ;
INSERT INTO EmployeeTable VALUES
(1,'munna', 2),
(2,'divya', 3),
(3,'ashish', 4),
(4,'anil', 4) ;

SELECT * FROM EmployeeTable ;

SELECT Table1.name AS EmpName, Table2.name AS ManagerName
FROM EmployeeTable As Table1
JOIN EmployeeTable AS Table2
ON Table1.manager_id = Table2.emp_id ;
```

FOREIGN KEY Implementation

Referenced table

rollno	name	state
1	Sachin	mumbai
2	virat	banglore
3	Dhoni	Chennai
4	dhawan	punjab
5	rohit	mumbai
NULL	NULL	NULL

```
CREATE TABLE STUDENT (rollno INT PRIMARY KEY AUTO_INCREMENT , name VARCHAR(50), state VARCHAR(40)) ;
INSERT INTO Student VALUES(1, 'Sachin', 'mumbai');
INSERT INTO Student(name, state) VALUES('virat', 'banglore'), ('Dhoni', 'Chennai'), ('dhawan', 'punjab');
```

```
CREATE TABLE Course (course_id INT PRIMARY KEY AUTO_INCREMENT , course_name VARCHAR(50), rollno INT,
FOREIGN KEY(rollno) REFERENCES Student(rollno)) ;
INSERT INTO Course VALUES (101, 'java', 1) ;
INSERT INTO Course (course_name, rollno) VALUES ( 'python', 2) ;
```

course_id	course_name	rollno
101	java	1
102	python	2
104	c	3
NULL	NULL	NULL

Referencing table

```
INSERT INTO Course (course_name, rollno) VALUES ( 'c++', 5) ; -- Error Code: 1452. Cannot add or update a child row: a foreign key constraint fails
```

* if rollno is not present in parent table i.e., Student table then we cannot add it in to Child Table

```
DELETE FROM Student
WHERE rollno=1 ; -- Error Code: 1451. Cannot delete or update a parent row: a foreign key constraint fails
```

* If perticular rollno is present in Child then we cannot delete it from parent

```
INSERT INTO Student(name, state) VALUES('rohit', 'mumbai') ;
INSERT INTO Course (course_name, rollno) VALUES ( 'c',3) ;
```

```
SELECT Student.name, Course.course_name FROM Student
INNER JOIN Course
ON Student.rollno = Course.rollno ;
```

name	course_name
Sachin	java
virat	python
Dhoni	c

```
UPDATE Student
SET rollno=10
WHERE rollno=3 ;
```

-- Error Code: 1451. Cannot delete or update a parent row: a foreign key constraint fails

```
UPDATE STUDENT
SET name = 'Arun'
WHERE rollno = 1 ;
```

-> can update other data except primary key

```
UPDATE Course
SET rollno=10
WHERE rollno=1 ;
```

* we cannot update in parent or child table data if that roll no is present in both

Solution of above problem: ON UPDATE CASCADE or ON DELETE CASCADE

```
CREATE TABLE Course (course_id INT PRIMARY KEY AUTO_INCREMENT , course_name VARCHAR(50), rollno INT,
FOREIGN KEY(rollno) REFERENCES Student(rollno)
ON DELETE CASCADE
ON UPDATE CASCADE) ;
```

```
UPDATE Student
SET rollno=13          * This will update in both the table and same with Delete
WHERE rollno=3 ;
```

```
DELETE FROM Student Where rollno=1 ;   *This will delete from both table
DELETE FROM Course Where rollno=2 ;    *This will delete from Course table only
```

```
CREATE TABLE Course (course_id INT PRIMARY KEY AUTO_INCREMENT , course_name VARCHAR(50), rollno INT,
FOREIGN KEY(rollno) REFERENCES Student(rollno)
ON DELETE SET NULL          --> this will set null value at that roll no and rest remain same, on delete the entire row
ON UPDATE SET NULL) ;
```

Referenced Table(Parent Table)

1. Insert: No Violation, i.e., we can insert values
2. Delete : May Cause Violation
   for ex: is data is not parent in child table then we can delete but if data is present in child then we cannot delete
3. Update: It may Cause Violation

Referencing Table(Child Table)

1. Insert: May cause Violation, if the data is not present in the parent table then we cannot insert in child
2. Delete : No Violation--> Student is enrolled but not taking any course
3. Update: It may Cause Violation

- we can update Course_id or Course_name but rollno we cannot update

GROUP BY Clause:

the GROUP BY clause is used to group rows that have the same values in specified columns into summary rows, often used with aggregate functions like COUNT(), SUM(), AVG(), MAX(), and MIN(). It allows you to perform calculations on each group of data independently.

std_id	std_name	subject
1	Amit	java
2	Rohit	c++
3	Sumit	java
4	Raj	python
5	Bhavek	python
6	Rahul	java
7	Gourav	C++
NULL	NULL	NULL

```
SELECT subject --> use the same column used with GROUP BY and we can also use Aggregate fun also(count, sum, max, min, avg)
FROM Student
GROUP BY subject ;
```

subject
java
c++
python

```
SELECT subject, COUNT(*)
FROM Student
GROUP BY subject ;
```

subject	COUNT(*)
java	3
c++	2
python	2

---

std_id	std_name	subject	salary
1	Amit	java	20000
2	Rohit	c++	50000
3	Sumit	java	30000
4	Raj	python	10000
5	Bhavek	python	10000
6	Rahul	java	25000
7	Gourav	C++	50000
NULL	NULL	NULL	NULL

```
SELECT subject, SUM(salary) AS Salary_of_emp
FROM Student
GROUP BY subject ;
```

subject	Salary_of_emp
java	75000
c++	100000
python	20000

```
SELECT subject, SUM(salary)
FROM Student
GROUP BY subject
```

subject	SUM(salary)
c++	100000
java	75000
python	20000

```
SELECT subject, SUM(salary)
FROM Student
GROUP BY subject
ORDER BY salary ;
```

==>Error Code: 1055. Expression #1 of ORDER BY clause is not in GROUP BY clause and contains nonaggregated column

```
SELECT subject, SUM(salary)
FROM Student
GROUP BY subject
ORDER BY AVG(salary) ;
```

==> Aggregate function can be use with ORDER BY here

subject	SUM(salary)
python	20000
java	75000
c++	100000

* In GROUP BY , With SELECT use the same column name used with GROUP BY, or you can use 5 aggregate function

* In GROUP BY, with order by also, use thge same columnas used with group by

```
SELECT subject, SUM(salary) AS total
FROM Student
GROUP BY subject
ORDER BY total ;
```

subject	total
python	20000
java	75000
c++	100000

HAVING Clause:

The HAVING clause in SQL is used to filter records after the GROUP BY clause has been applied. It allows you to filter the results based on aggregate functions like SUM(), COUNT(), AVG(), etc. It is similar to the WHERE clause, but WHERE filters rows before grouping, while HAVING filters groups after aggregation.

Basic Syntax:

```
SELECT column_name, aggregate_function(column_name)
FROM table_name
GROUP BY column_name
HAVING condition;
```

Example 1: Using HAVING with COUNT()

Suppose you have a table called orders with columns customer_id and order_amount, and you want to find customers who have placed more than 3 orders.

```
SELECT customer_id, COUNT(order_id) AS total_orders
FROM orders
GROUP BY customer_id
HAVING COUNT(order_id) > 3;
```

Explanation:

GROUP BY customer_id: Groups the orders by each customer.
COUNT(order_id): Counts the number of orders for each customer.
HAVING COUNT(order_id) > 3: Filters the groups where the customer has placed more than 3 orders.

Example 2: Using HAVING with SUM()
Let's say you want to find customers who have made total purchases greater than $5000.

```
SELECT customer_id, SUM(order_amount) AS total_spent
FROM orders
GROUP BY customer_id
HAVING SUM(order_amount) > 5000;
```

Explanation:
SUM(order_amount): Sums up the total amount spent by each customer.
HAVING SUM(order_amount) > 5000: Only shows customers whose total purchases exceed $5000.

Example 3: Combining WHERE and HAVING
If you want to first filter records before grouping, you can use the WHERE clause in combination with HAVING. For instance, if you want to find customers who made orders greater than $100 and whose total purchases exceed $5000:

```
SELECT customer_id, SUM(order_amount) AS total_spent
FROM orders
WHERE order_amount > 100
GROUP BY customer_id
HAVING SUM(order_amount) > 5000;
```

Explanation:
WHERE order_amount > 100: Filters orders where the individual order amount is greater than $100.
HAVING SUM(order_amount) > 5000: After grouping, filters customers whose total spending exceeds $5000.

Subqueries

emp_id	emp_name	dept	salary
101	Rohit	Developer	500000
102	Virat	Marketing	400000
103	Sachin	HR	250000
104	Ashwin	Developer	150000
105	Rahul	HR	250000
106	Hardik	Marketing	200000
107	Rishabh	HR	300000
NULL	NULL	NULL	NULL

```
CREATE TABLE Employee(emp_id int PRIMARY KEY auto_increment, emp_name VARCHAR(20) NOT NULL, dept VARCHAR(20) NOT NULL, salary INT NOT NULL) ;

INSERT INTO Employee VALUES (101, 'Rohit', 'Developer' , 500000) ;
INSERT INTO Employee (emp_name, dept, salary) VALUES ('Virat', 'Marketing' , 400000) ;
INSERT INTO Employee (emp_name, dept, salary) VALUES ('Sachin', 'HR' , 250000),
('Ashwin', 'Developer' , 150000),
('Rahul', 'HR' , 250000),
('Hardik', 'Marketing' , 200000) ,
('Rishabh', 'HR' , 300000);
```

Q1. Write a SQL Query to find the max salary of from the employee table.

```
SELECT MAX(salary) FROM Employee ;
```

MAX(salary)
500000

Q2. Write a SQL Query to find the name to the employee whose salary is maximum.

emp_name
Rohit

```
SELECT emp_name FROM Employee
WHERE salary = (SELECT MAX(salary) FROM Employee ) ;
```

outer query

Inner Query

Q3. Write a Query to display the second highest salary

find the max salary from the salaries which doen nort include the max salary

```
SELECT salary from Employee
WHERE salary != (SELECT MAX(salary) FROM Employee) ;
```

salary
400000
250000
150000
250000
200000
300000

above query give the salary which does not include max salary

now find the max salary from this data which does not include the max salary

```
SELECT MAX(salary) from Employee
WHERE salary != (SELECT MAX(salary) FROM Employee) ;
```

MAX(salary)
400000

Q4. Write a SQL Query to find the name to the employee who is taking second highest salary

```
SELECT emp_name FROM Employee
WHERE salary = (SELECT MAX(salary) from Employee WHERE salary !=
(SELECT MAX(salary) FROM Employee) ) ;
```

emp_name
Virat

Q5. Write a query to display all the department name along with the number of employees working in it

```
SELECT dept, COUNT(*)
FROM Employee
GROUP BY dept ;
```

dept	COUNT(*)
Developer	2
Marketing	2
HR	3

Q6. Write a query to display all the dept name where no. of employees are less than 3.

```
SELECT dept, COUNT(*) AS count_of_employee
FROM Employee
GROUP BY dept
HAVING count_of_employee<3 ;
```

dept	count_of_employee
Developer	2
Marketing	2

OR

```
SELECT dept
FROM Employee
GROUP BY dept
HAVING COUNT(*)<3 ;
```

dept
Developer
Marketing

Q7. Write a query to display the name of the employee who works in the dept where no. of employee is less than 3.

```
SELECT emp_name FROM Employee
WHERE dept IN (SELECT dept FROM Employee GROUP BY dept
HAVING COUNT(*)<3 ) ;
```

emp_name
Rohit
Virat
Ashwin
Hardik

Q8. Write a query to display the highest salary dept wise and name of the employee taking that salary

```
SELECT dept, MAX(salary) FROM Employee
GROUP BY dept ;
```

--> this will give the dept wise max salary

dept	MAX(salary)
Developer	500000
Marketing	400000
HR	300000

```
SELECT emp_name FROM Employee
WHERE salary IN (SELECT MAX(salary) FROM Employee  GROUP BY dept) ;
```

emp_name
Rohit
Virat
Rishabh

OR

```
SELECT emp_name,dept, salary FROM Employee
WHERE salary IN (SELECT MAX(salary) FROM Employee  GROUP BY dept) ;
```

emp_name	dept	salary
Rohit	Developer	500000
Virat	Marketing	400000
Rishabh	HR	300000