

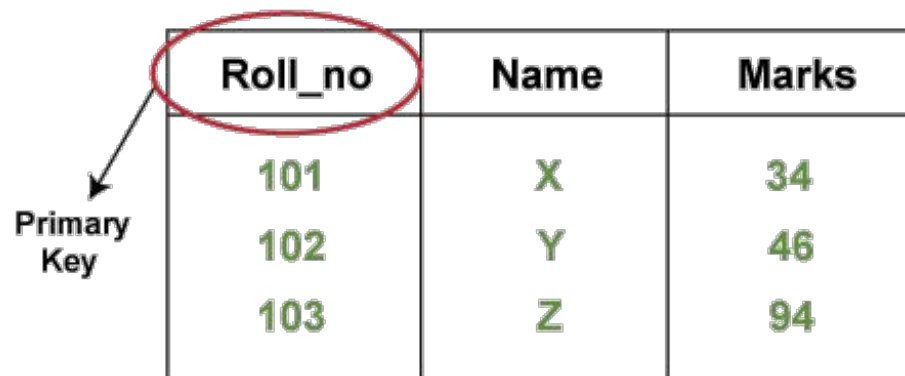
## Primary Key

- **Uniqueness:** Each primary key value must be unique per table row.
- **Immutable:** Primary keys should not change once set.
- **Simplicity:** Ideal to keep primary keys as simple as possible.
- **Non-Intelligent:** They shouldn't contain meaningful information.
- **Indexed:** Primary keys are automatically indexed for faster data retrieval.
- **Referential Integrity:** They serve as the basis for foreign keys in other tables.
- **Data Type:** Common types are integer or string.

**STUDENT\_DETAILS**

| Roll_no | Name | Marks |
|---------|------|-------|
| 101     | X    | 34    |
| 102     | Y    | 46    |
| 103     | Z    | 94    |

Primary Key

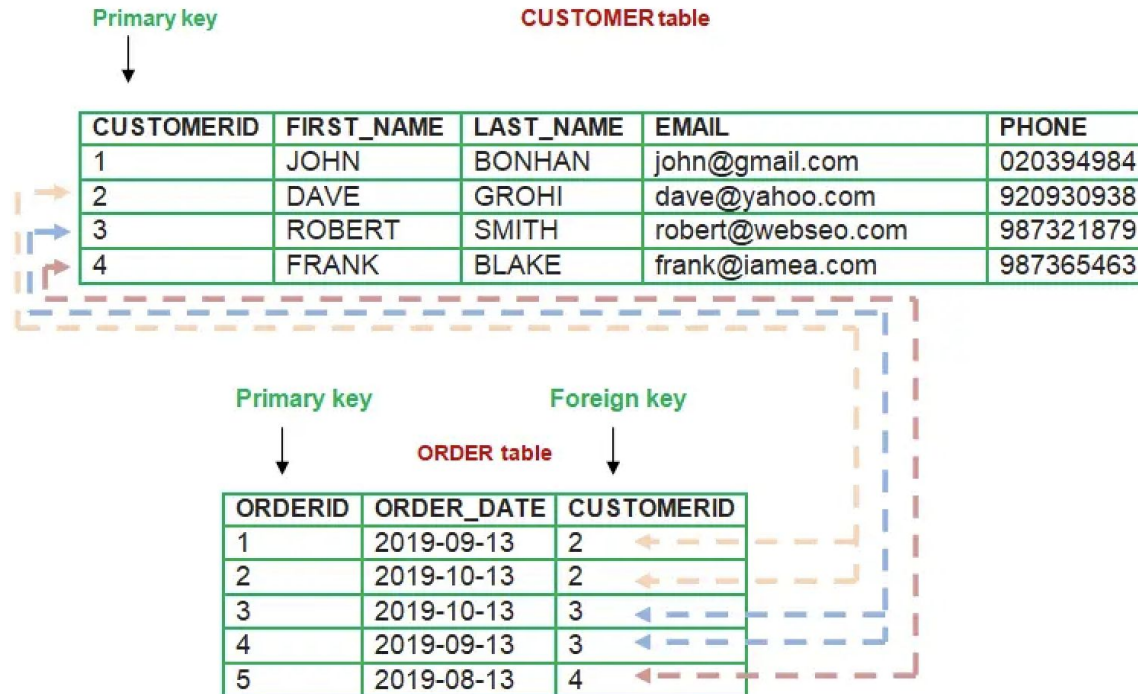


## Foreign Key

- **Referential Integrity:** Foreign keys link records between tables, maintaining data consistency.
- **Nullable:** Foreign keys can contain null values unless specifically restricted.
- **Match Primary Keys:** Each foreign key value must match a primary key value in the parent table, or be null.
- **Ensure Relationships:** They define the relationship between tables in a database.
- **No Uniqueness:** Foreign keys don't need to be unique.

## Foreign Key - Example

A primary key-foreign key relationship




## Delete command

The DELETE command in SQL is used to remove existing records from a table. Here's a basic syntax:

```
DELETE FROM table_name WHERE condition;
```

For example, to delete a record from a Students table where ID equals 5:

```
DELETE FROM Students WHERE ID = 5;
```

 **Be careful:** if you run the DELETE command without a WHERE clause, it will delete all records from the table.

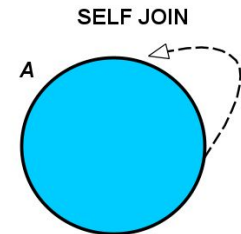
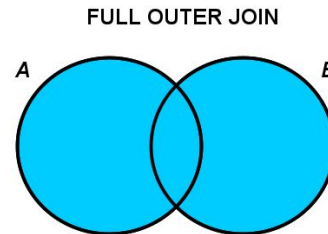
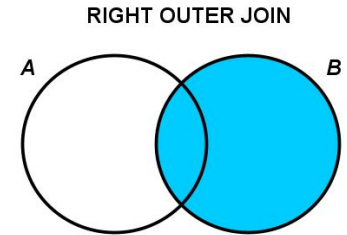
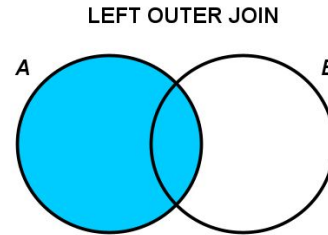
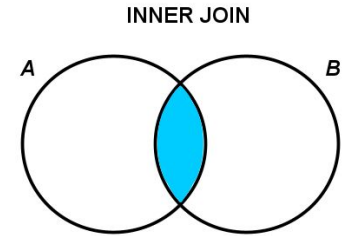
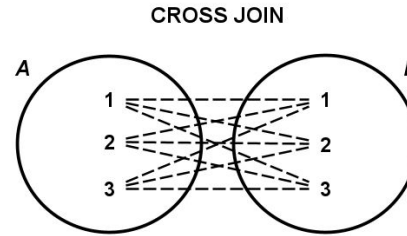
# Drop Vs Truncate vs Delete

|                              | DROP  | TRUNCATE   | DELETE   |
|------------------------------|---|--|--|
| <b>Purpose</b>               | Completely removes the entire table structure from the database | Removes all rows from a table, but the table structure remains | Removes specific rows based on a condition or all rows from a table, but the table structure remains |
| <b>Transaction Control</b>   | Cannot be rolled back   | Cannot be rolled back  | Can be rolled back   |
| <b>Space Reclaiming</b>      | Releases the object and its space                               | Frees the space containing the table                           | Doesn't free up space, but leaves empty space for future use   |
| <b>Speed</b>                 | Fastest as it removes all data and structure                    | Faster than DELETE as it doesn't log individual row deletions  | Slowest as it logs individual row deletions  |
| <b>Referential Integrity</b> | Not checked   | Checked  | Checked  |
| <b>Where Clause</b>          | Not applicable  | Not applicable   | Applicable   |
| <b>Command Type</b>          | DDL (Data Definition Language)                                  | DDL (Data Definition Language)                                 | DML (Data Manipulation Language)   |

# Joins

SQL joins are used to combine rows from two or more tables, based on a related column between them. Here are the main types of SQL joins:

- Inner Join
- Left Join
- Right Join
- Full Join
- Cross Join



# Inner Join

Returns records that have matching values in both tables.

## Syntax:

```
SELECT Customers.customer_id,  
       Customers.first_name,  
       Orders.amount  
FROM Customers  
INNER JOIN Orders  
ON Orders.customer = Customers.customer_id;
```


## SQL INNER JOIN

Table: Customers

| customer_id | first_name |
|-------------|------------|
| 1           | John       |
| 2           | Robert     |
| <u>3</u>    | David      |
| 4           | John       |
| <u>5</u>    | Betty      |

Table: Orders

| order_id | amount | customer |
|----------|--------|----------|
| 1        | 200    | 10       |
| 2        | 500    | <u>3</u> |
| 3        | 300    | 6        |
| 4        | 800    | <u>5</u> |
| 5        | 150    | 8        |



| customer_id | first_name | amount |
|-------------|------------|--------|
| 3           | David      | 500    |
| 5           | Betty      | 800    |

# Left Join

Returns all records from the left table (table1), and the matched records from the right table (table2). If no match, the result is NULL on the right side.

## SQL LEFT JOIN

### Syntax:


```
SELECT Customers.customer_id,  
       Customers.first_name,  
       Orders.amount  
FROM Customers  
LEFT JOIN Orders  
ON Orders.customer = Customers.customer_id;
```

Table: Customers

| customer_id | first_name |
|-------------|------------|
| 1           | John       |
| 2           | Robert     |
| 3           | David      |
| 4           | John       |
| 5           | Betty      |

Table: Orders

| order_id | amount | customer |
|----------|--------|----------|
| 1        | 200    | 10       |
| 2        | 500    | 3        |
| 3        | 300    | 6        |
| 4        | 800    | 5        |
| 5        | 150    | 8        |



| customer_id | first_name | amount |
|-------------|------------|--------|
| 1           | John       |        |
| 2           | Robert     |        |
| 3           | David      | 500    |
| 4           | John       |        |
| 5           | Betty      | 800    |



# Right Join

Returns all records from the right table (table2), and the matched records from the left table (table1). If no match, the result is NULL on the left side.

## SQL RIGHT JOIN

### Syntax:

```
SELECT Customers.customer_id,  
       Customers.first_name,  
       Orders.amount  
FROM Customers  
LEFT JOIN Orders  
ON Orders.customer = Customers.customer_id;
```

Table: Customers

| customer_id | first_name |
|-------------|------------|
| 1           | John       |
| 2           | Robert     |
| <u>3</u>    | David      |
| 4           | John       |
| <u>5</u>    | Betty      |

Table: Orders

| order_id | amount | customer |
|----------|--------|----------|
| 1        | 200    | 10       |
| 2        | 500    | <u>3</u> |
| 3        | 300    | 6        |
| 4        | 800    | <u>5</u> |
| 5        | 150    | 8        |



| customer_id | first_name | amount |
|-------------|------------|--------|
| 3           | David      | 500    |
| 5           | Betty      | 800    |
|             |            | 200    |
|             |            | 300    |
|             |            | 150    |

# Full Join

Returns all records when there is a match in either left (table1) or right (table2) table records.

## Syntax:

```
SELECT Customers.customer_id,  
       Customers.first_name,  
       Orders.amount  
FROM Customers  
FULL OUTER JOIN Orders  
ON Orders.customer = Customers.customer_id;
```

## SQL FULL OUTER JOIN

Table: Customers

| customer_id | first_name |
|-------------|------------|
| 1           | John       |
| 2           | Robert     |
| 3           | David      |
| 4           | John       |
| 5           | Betty      |

Table: Orders

| order_id | amount | customer |
|----------|--------|----------|
| 1        | 200    | 10       |
| 2        | 500    | 3        |
| 3        | 300    | 6        |
| 4        | 800    | 5        |
| 5        | 150    | 8        |



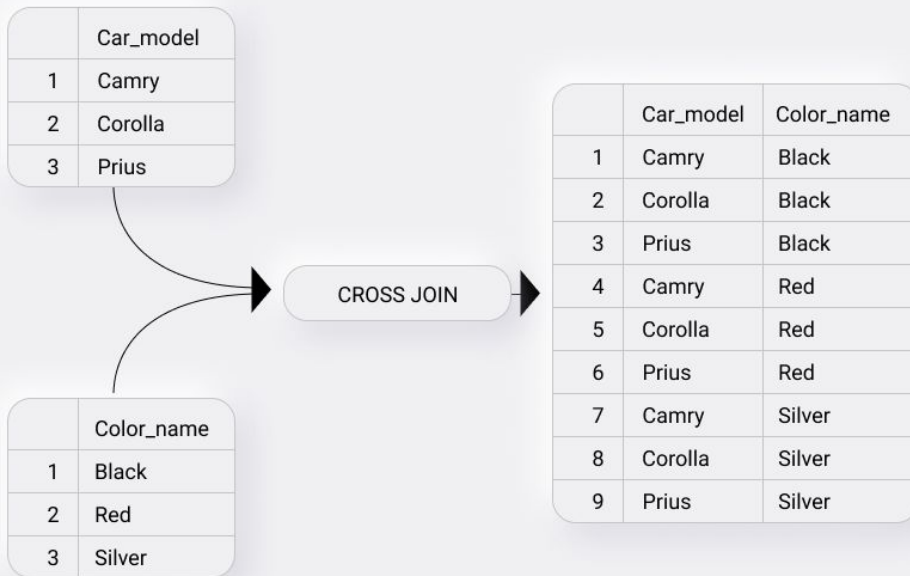
| customer_id | first_name | amount |
|-------------|------------|--------|
| 3           | David      | 200    |
|             |            | 500    |
|             |            | 300    |
| 5           | Betty      | 800    |
|             |            | 150    |
| 2           | Robert     |        |
| 4           | John       |        |

# Cross Join

Returns the Cartesian product of the sets of records from the two or more joined tables when no WHERE clause is used with CROSS JOIN.

## Syntax:

```
SELECT Model.car_model,  
       Color.color_name  
FROM Model  
Cross JOIN Color;
```



## Self Join

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

Let's take a look at an example. Consider the table **Employees** :

| Id | FullName      | Salary | ManagerId |
|----|---------------|--------|-----------|
| 1  | John Smith    | 10000  | 3         |
| 2  | Jane Anderson | 12000  | 3         |
| 3  | Tom Lanon     | 15000  | 4         |
| 4  | Anne Connor   | 20000  |           |
| 5  | Jeremy York   | 9000   | 1         |

**Now, to show the name of the manager for each employee in the same row, we can run the following query:**

```
SELECT
    employee.Id,
    employee.FullName,
    employee.ManagerId,
    manager.FullName as ManagerName
FROM Employees employee
JOIN Employees manager
ON employee.ManagerId = manager.Id
```

| Id | FullName      | ManagerId | ManagerName |
|----|---------------|-----------|-------------|
| 1  | John Smith    | 3         | Tom Lanon   |
| 2  | Jane Anderson | 3         | Tom Lanon   |
| 3  | Tom Lanon     | 4         | Anne Connor |
| 5  | Jeremy York   | 1         | John Smith  |

# Group By WITH ROLLUP

The GROUP BY clause in MySQL is used to group rows that have the same values in specified columns into aggregated data. The WITH ROLLUP option allows you to include extra rows that represent subtotals and grand totals.

## Input Table

| payment_amount | payment_date | store_id |
|----------------|--------------|----------|
| 1200.99        | 2018-01-18   | 1        |
| 189.23         | 2018-02-15   | 1        |
| 3002.43        | 2018-02-25   | 2        |
| 33.43          | 2018-03-03   | 3        |
| 7382.10        | 2019-01-11   | 2        |
| 382.92         | 2019-02-18   | 1        |
| 100.99         | 2019-03-07   | 1        |
| 322.34         | 2019-03-29   | 2        |
| 2929.14        | 2020-01-03   | 2        |
| 211.65         | 2020-02-02   | 1        |
| 499.02         | 2020-02-19   | 3        |
| 994.11         | 2020-03-14   | 1        |
| 500.73         | 2021-01-06   | 3        |
| 394.93         | 2021-01-22   | 2        |
| 3332.23        | 2021-02-23   | 3        |
| 9499.49        | 2021-03-10   | 3        |

- Rows with a Payment Year value but a NULL Store represent subtotals for each Payment Year.
- The row with NULL in both the Payment Year and Store columns represents the grand total payment\_amount sum.

## Output Table

| SUM(payment_amount) | Payment Year | Store |
|---------------------|--------------|-------|
| 30975.73            | NULL         | NULL  |
| 4426.08             | 2018         | NULL  |
| 1390.22             | 2018         | 1     |
| 3002.43             | 2018         | 2     |
| 33.43               | 2018         | 3     |
| 8188.35             | 2019         | NULL  |
| 483.91              | 2019         | 1     |
| 7704.44             | 2019         | 2     |
| 4633.92             | 2020         | NULL  |
| 1205.76             | 2020         | 1     |
| 2929.14             | 2020         | 2     |
| 499.02              | 2020         | 3     |
| 13727.38            | 2021         | NULL  |
| 394.93              | 2021         | 2     |
| 13332.45            | 2021         | 3     |

## Group By WITH ROLLUP - Query

```
SELECT
    SUM(payment_amount),
    YEAR(payment_date) AS 'Payment Year',
    store_id AS 'Store'
FROM payment
GROUP BY YEAR(payment_date), store_id WITH ROLLUP
ORDER BY YEAR(payment_date), store_id;
```

## Views

A view in SQL is a virtual table based on the result-set of an SQL statement. It 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.

Here are some key points about views:

- You can add SQL functions, WHERE, and JOIN statements to a view and display the data as if the data were coming from one single table.
- A view always shows up-to-date data. The database engine recreates the data every time a user queries a view.
- Views can be used to encapsulate complex queries, presenting users with a simpler interface to the data.
- They can be used to restrict access to sensitive data in the underlying tables, presenting only non-sensitive data to users.



## Syntax to create Views

```
CREATE VIEW View_Products AS  
SELECT ProductName, Price  
FROM Products  
WHERE Price > 30;
```

### Employee

| EmployeeID | Ename | DeptID | Salary |
|------------|-------|--------|--------|
| 1001       | John  | 2      | 4000   |
| 1002       | Anna  | 1      | 3500   |
| 1003       | James | 1      | 2500   |
| 1004       | David | 2      | 5000   |
| 1005       | Mark  | 2      | 3000   |
| 1006       | Steve | 3      | 4500   |
| 1007       | Alice | 3      | 3500   |

```
CREATE VIEW emp_view AS  
SELECT DeptID, AVG(Salary)  
FROM Employee  
GROUP BY DeptID;
```

Create View of  
grouped records  
on Employee  
table

### emp\_view

| DeptID | AVG(Salary) |
|--------|-------------|
| 1      | 3000.00     |
| 2      | 4000.00     |
| 3      | 4250.00     |