CHAPTER 4

Relational Database Query Languages

Introduction to SQL

- ✓ SQL is a standard database language used to access and manipulate data in databases.
- ✓ SQL stands for Structured Query Language.
- ✓ IBM developed the original version of SQL, originally called sequel ,as part of the system R project in the early 1970's.The sequel language has evolved since then, and its name has changed to SQL(Structured Query Language)
- ✓ By executing queries SQL can create, update, delete, and retrieve data in within a database management systems like MySQL, Oracle, PostgreSQL, etc.
- ✓ Overall SQL is a query language that communicates with databases.

The SQL language has several parts,

Data Definition Language (DDL): SQL provides a set of commands to define and modify the structure of a database, including creating tables, modifying table structure, and dropping tables.

Data Manipulation Language (DML): SQL provides a set of commands to manipulate data within a database, including adding, modifying, and deleting data.

SQL provides a rich set of commands for querying a database to retrieve data, including the ability to filter, sort, group, and join data from multiple tables.

Transaction Control: SQL supports transaction processing, which allows users to group a set of database operations into a single transaction that can be rolled back in case of failure.

Data Integrity: SQL includes features to enforce data integrity, such as the ability to specify constraints on the values that can be inserted or updated in a table, and to enforce referential integrity between tables.

User Access Control: SQL provides mechanisms to control user access to a database, including the ability to grant and revoke privileges to perform certain operations on the database.

Data types in SQL

char(n) : Fixed length character string, with user-specified length n.

varchar(n) : Variable length character strings, with user-specified maximum length n.

int : Integer (a finite subset of the integers that is machine-dependent).

smallint: Small integer (a machine-dependent subset of the integer domain type).

numeric(p,d): Fixed point number, with user-specified precision of p digits, with d digits to the right of decimal point. (ex., numeric(3,1), allows 44.5 to be stores exactly, but not 444.5 or 0.32)

real, double precision: Floating point and double-precision floating point numbers, with machine-dependent precision.

float(n): Floating point number, with user-specified precision of at least n digits.

Date and Time types in SQL

date: A calendar date containing a (four-digit) year, month, and day of the month

time: The time of day, in hours, minutes and seconds.

timestamp: A combination of date and time

Date and time values can be specified like this:

date: '2001-04-25'
time :'09:30:00'
timestamp '2001-04-25 10:29:01.45'

DDL and DML queries in DBMS

Data Definition Language (DDL)

- ✓ The Data Definition Language is made up of SQL commands that can be used to design the database structure.
- ✓ DDL refers to a set of SQL instructions for creating, modifying, and deleting database structures, but not data
- ✓ Popular DDL commands are: CREATE, DROP, ALTER and TRUNCATE.

CREATE: The database or its objects are created with this command (like table, views, store procedure, and triggers).

- ✓ A database is a systematic collection of data. To store data in a well-structured manner, the first step with SQL is to establish a database. To build a new database in SQL, use the CREATE DATABASE statement.
  Syntax: CREATE DATABASE db_name;

  Example:
  CREATE DATABASE student_db;
  The above example will create a database named student_db;

- ✓ We've already learned how to create databases. To save the information, we'll need a table. In SQL, the CREATE TABLE statement is used to make a table. A table is made up of rows and columns, as we all know. As a result, while constructing tables, we must give SQL all relevant information, such as the names of the columns, the type of data to be stored in the columns, the data size, and so on.

  Syntax:

  CREATE TABLE table_name(

  column1 data_type1,
  column2 data_type2,
  column3 data_type3,
  column4 data_type4,
  …..
  );
  Example:

```
CREATE TABLE student_info(
sid int,
name  varchar(30),
program varchar(30),
roll int);
```

The above command will create the table schema that look like:

sid	name	program	roll

DROP

- ✓ The DROP statement deletes existing objects such as databases, tables and views.

  For dropping table

  Syntax: DROP TABLE table_name;
  Example: DROP TABLE student_info;

  For dropping database

  Syntax: DROP DATABASE db_name;
  Example: DROP DATABASE student_db;

ALTER

- ✓ In an existing table, this command is used to add, delete or edit columns.
- ✓ It can also be used to create and remove constraints from a table that already exists.

To add Column in table

Syntax:

ALTER TABLE table_name

ADD column_name datatype;

Example:

ALTER TABLE student_info;

ADD address varchar(30);

To remove existing column from table

Syntax:

ALTER TABLE table_name
DROP COLUMN column_name;

Example:
ALTER TABLE student_info
DROP COLUMN roll;

To rename column of table

Syntax:

ALTER TABLE *table_name*
CHANGE COLUMN *old_name new_name datatype*;

Example:
ALTER TABLE student_info
CHANGE COLUMN address location varchar(30);

(Note: This syntax is for MariaDB and may vary upon different DBMS)

To modify data type of column

Syntax:

ALTER TABLE table_name
MODIFY COLUMN column_name datatype;

Example:

ALTER TABLE student_info

MODIFY COLUMN program char(20);

TRUNCATE

- ✓ This statement deletes all the rows from the table.
- ✓ This is different from the DROP command, the DROP command deletes the entire table along with the table schema, however TRUNCATE just deletes all the rows and leaves an empty table.

Syntax:

TRUNCATE TABLE table_name;

Example:

TRUNCATE TABLE student_info;

//Deletes all the rows from the table student_info.

By performing all the above operations, finally our table named student_info becomes

sid	name	program	location
int	varchar(30)	char(20)	varchar(30)

Data Manipulation Language (DML)

- ✓ The SQL commands that deal with manipulating data in a database are classified as DML (Data Manipulation Language)
- ✓ The popular commands that come under DML are INSERT,UPDATE,DELETE,SELECT
- ✓ This command is used to insert records in a table

INSERT

- ✓ When you are not inserting the data for all the columns and leaving some columns empty. In that case specify the column name and corresponding value. The non selected field will have NULL value inserted upon execution of the given query.

Syntax:

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

Example:

```
INSERT INTO student_info(sid,name,location)
VALUES(1,'Hari','Pokhara');
```

- ✓ When inserting the data for all the columns. No need to specify column name.

Syntax:

INSERT INTO table_name

VALUES (value1, value2, value3, ...);

Example:

```
INSERT INTO student_info

 VALUES(2,'Rita','Computer','Butwal');
```

UPDATE

- ✓ In SQL, the UPDATE statement is used to update data in an existing database table.

Syntax:

UPDATE table_name

SET column1 = value1, column2 = value2,…

WHERE condition;

Example:

```
UPDATE student_info
SET location='kathmandu'
WHERE sid=2;
```

DELETE

- ✓ DELETE statement is used to delete records from a table.
- ✓ Depending on the condition we set in the WHERE clause, we can delete a single record or numerous records.

Syntax:

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

Example1:

```
DELETE FROM student_info
WHERE location='kathmandu';
```

//Delete records of student from table named student_info whose location is Kathmandu

Example 2:

```
DELETE FROM  student_info;
```

//Delete all records from table named student_info;

SELECT

- ✓ SELECT command fetches the records from the specified table that matches the given condition, if no condition is provided, it fetches all the records from the table.

Syntax:

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

Here, column1, column2, ... are the column names of the table we want to select data from.

- ✓ If we want to apply conditions while selecting the data then syntax becomes

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

- ✓ If we want to select all the columns and rows available in the table, use the following syntax:

  SELECT * FROM table_name;

  Example:

  SELECT * FROM student_info;

  //Displays all the information of students from table named student_info

  SELECT name,program

  FROM student_info

  WHERE location='pokhara';

  //Displays name and program of students from table named student_info whose location is 'pokhara'

Note: Some authors grouped SELECT command as DQL(Data Query Language)

SQL constraints

- 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.

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

NOT NULL

- ✓ 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 without adding a value to this field.

❖ **create table with NOT NULL constraint**

Example:

```
CREATE TABLE Colleges (
college_id INT NOT NULL,
college_code VARCHAR(20),
college_name VARCHAR(50)
);
```

❖ **Add the NOT NULL constraint to a column in an existing table**

Example:

```
ALTER TABLE Colleges
MODIFY COLUMN college_id INT NOT NULL;
```

❖ **Remove NOT NULL Constraint**

Example:

```
ALTER TABLE Colleges
MODIFY college_id INT;
```

UNIQUE

- ✓ The UNIQUE constraint ensures that all values in a column are different.

❖ **Create a table with unique constraint**

Example

```
CREATE TABLE Colleges (
 college_id INT NOT NULL UNIQUE,
 college_code VARCHAR(20) UNIQUE,
 college_name VARCHAR(50)
);
```

❖ **Add the UNIQUE constraint to an existing column**

For single column

Example

```
ALTER TABLE Colleges
ADD UNIQUE (college_id);
```

For multiple columns

Example

```
ALTER TABLE Colleges
ADD UNIQUE Unique_College (college_id, college_code);
```

- ✓ Here, the SQL command adds the UNIQUE constraint to college_id and college_code columns in the existing Colleges table.
- ✓ Also, Unique_College is a name given to the UNIQUE constraint defined for college_id and college_code columns.

❖ **DROP a UNIQUE Constraint**

Example

```
ALTER TABLE Colleges
DROP INDEX Unique_College;
```

PRIMARY KEY

- ✓ The PRIMARY KEY constraint uniquely identifies each record in a table.
- ✓ Primary keys must contain UNIQUE values, and cannot contain NULL values.

❖ **Create table with PRIMARY KEY constraint**

Syntax:

```
CREATE TABLE table_name (
 column1 data_type,
  ..............................,
  [CONSTRAINT constraint_name] PRIMARY KEY (column1)
);
```

Example

```
CREATE TABLE Colleges (
  college_id INT,
  college_code VARCHAR(20) ,
  college_name VARCHAR(50),
CONSTRAINT CollegePK  PRIMARY KEY (college_id)
);
```

//Create Colleges table with primary key college_id

❖ **Add the PRIMARY KEY constraint to a column in an existing table**

Example

```
ALTER TABLE Colleges
ADD CONSTRAINT CollegePK PRIMARY KEY (college_id);
```

❖ **DROP a PRIMARY KEY Constraint**

Example

```
ALTER TABLE Colleges
DROP PRIMARY KEY;
```

DEFAULT

- ✓ the DEFAULT constraint is used to set a default value if we try to insert an empty value into a column.
- ✓ However if the user provides value then the particular value will be stored.

❖ **Default constraint while creating table**

The following example set default value of college_country column to 'Nepal'

Example:

```
CREATE TABLE Colleges (
  college_id INT PRIMARY KEY,
  college_code VARCHAR(20),
  college_country VARCHAR(20) DEFAULT 'Nepal'
);
```

❖ **Add the DEFAULT constraint to an existing column**

Example:

```
ALTER TABLE Colleges
ALTER college_country SET DEFAULT 'Nepal';
```

❖ **Remove DEFAULT Constraint**

Example:

```
ALTER TABLE Colleges
ALTER college_country DROP DEFAULT;
```

CHECK

- ✓ 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.

❖ **CHECK constraint while creating table**

Example:

Here we are Applying the CHECK constraint named amountCK the constraint makes sure that amount is greater than 0.

```
CREATE TABLE Orders (
order_id INT PRIMARY KEY,
amount INT,
CONSTRAINT amountCK CHECK (amount > 0)
);
```

❖ **Add CHECK Constraint in Existing Table**

Here we add CHECK constraint named amountCK the constraint makes sure that amount is greater than 0.

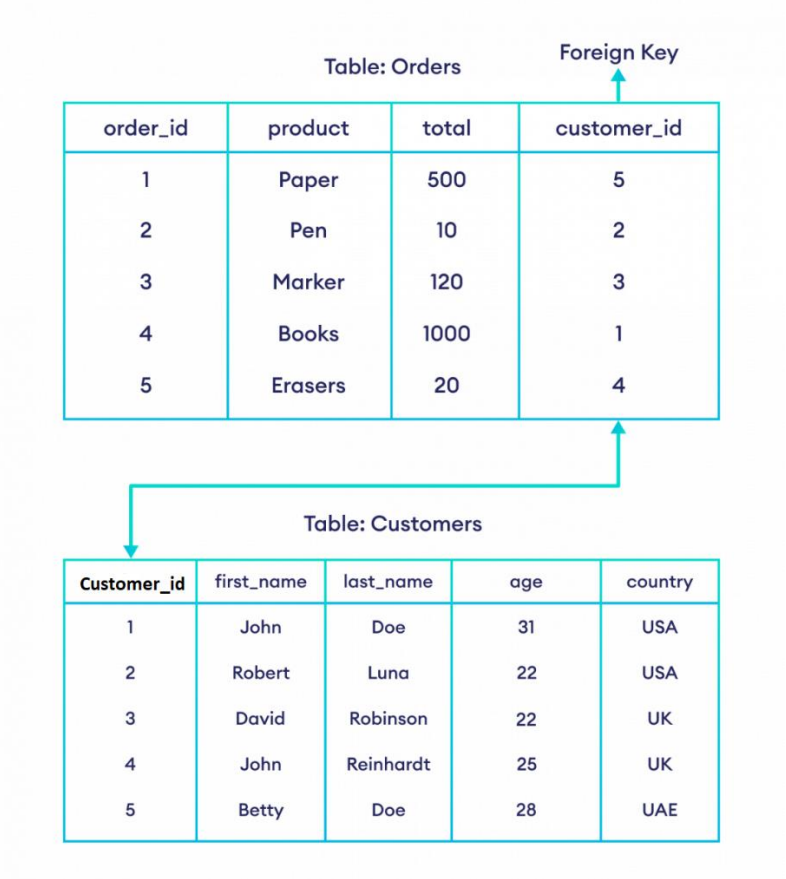
```
ALTER TABLE Orders
ADD CONSTRAINT amountCK CHECK (amount > 0);
```

❖ **Remove CHECK Constraint**

```
ALTER TABLE Orders
DROP CONSTRAINT amountCK;
```

FOREIGN KEY

The FOREIGN KEY constraint in SQL establishes a relationship between two tables by linking columns in one table to those in another.



- ✓ Here, the **customer_id** field in the Orders table is a FOREIGN KEY that references the **customer_id** field in the Customers table.
- ✓ This means that the value of the **customer_id** (of the Orders table) must be a value from the **customer_id** column (of the Customers table).

The syntax of the SQL FOREIGN KEY constraint is:

```
CREATE TABLE table_name (
   column1 data_type,
   column2 data_type,
   ……..,
  [CONSTRAINT CONSTRAINT_NAME] FOREIGN KEY (column_name)
   REFERENCES referenced_table_name (referenced_column_name)
);
```

Here,

- ✓ table_name is the name of the table where the FOREIGN KEY constraint is to be defined
- ✓ column_name is the name of the column where the FOREIGN KEY constraint is to be defined
- ✓ referenced_table_name and referenced_column_name are the names of the table and the column that the FOREIGN KEY constraint references
- ✓ [CONSTRAINT CONSTRAINT_NAME] is optional

Let us see with following example

- ✓ This table doesn't have a foreign key
- ✓ add foreign key to the customer_id field
- ✓ the foreign key references the id field of the Customers table

```
   -- this table doesn't have a foreign key

CREATE TABLE Customers (
 customer_id INT,
  first_name VARCHAR(40),
  last_name VARCHAR(40),
  age INT,
  country VARCHAR(10),
  CONSTRAINT CustomersPK PRIMARY KEY (customer_id)
);
-- add foreign key to the customer_id field
-- the foreign key references the id field of the Customers table
CREATE TABLE Orders (
  order_id INT,
  product VARCHAR(40),
  total INT,
  customer_id INT,
  CONSTRAINT OrdersPK PRIMARY KEY (order_id),
  CONSTRAINT CustomerOrdersFK FOREIGN KEY (customer_id) REFERENCES
Customers(customer_id)
);
```

Add the FOREIGN KEY constraint to an existing table

- ✓ add foreign key to the **customer_id** field of Orders the foreign key references the **customer_id** field of Customers

```
ALTER TABLE Orders
ADD FOREIGN KEY (customer_id) REFERENCES Customers(customer_id);
```

Remove a FOREIGN KEY Constraint

```
ALTER TABLE Orders
DROP FOREIGN KEY CustomerOrdersFK;
```

Operators in SQL

- An operator is a reserved word or a character that is used to query our database in a SQL expression.
- To query a database using operators, we use a WHERE clause.
- The operator manipulates the data and gives the result based on the operator's functionality.

Before starting with operators let us consider the following relation that we use to illustrate the examples of operators
Customers(customer_id,first_name,last_name,age,country);
Orders(order_id,product,total,customer_id);

Some operators available in SQL are:

Arithmetic Operators

- ✓ These operators are used to perform operations such as addition, multiplication, subtraction etc.
- ✓ Example. + (Addition) ,- (substraction), * (multiplication), / (division), % (modulus) etc.

```
UPDATE Orders
SET total=total+15;
```

This query increase the total amount of all records by 15.

Comparison Operators

- ✓ We can compare two values using comparison operators in SQL.
- ✓ These operators return either 1 (means true) or 0 (means false).
  Example:

Operator	Description
=	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<> , !=	Not equal to

```
SELECT *
FROM customers
WHERE age>20;
```

This query display the information of customers whose age is greater than 20

Logical operators

We can use logical operators to compare multiple SQL commands. These operators return either 1 (means true) or 0 (means false).
Some of the Logical operators available in SQL are,

AND
OR
NOT
BETWEEN
IN
LIKE

AND

- Returns the records if all the conditions separated by AND are TRUE

Example:

- ✓ *Display the first_name and last_name of all customers who live in 'Nepal' and have the last_name 'Paudel'*

```
SELECT first_name, last_name
FROM Customers
WHERE country = 'Nepal' AND last_name = 'Paudel';
```

OR

- Returns the records for which any of the conditions separated by OR is true

Example:

- ✓ *Display the first_name and last_name of all customers who either live in the 'Nepal' or have the last name 'Paudel'*

```
SELECT first_name, last_name
FROM Customers
WHERE country = 'Nepal' OR last_name = 'Paudel';
```

NOT

- Used to reverse the output of any logical operator

Example:

Display customers who don't live in the USA

```
SELECT first_name, last_name
FROM Customers
WHERE NOT country = 'USA';
```

Combining Multiple Operators

- ✓ It is also possible to combine multiple AND, OR and NOT operators in an SQL statement.

Display customers who live in either USA or UK and whose age is less than 26

```
SELECT  *
FROM Customers
WHERE (country = 'USA' OR country = 'UK') AND age < 26;
```

BETWEEN

- ❖ Returns the rows for which the value lies between the mentioned range.

Example:

Displays customers first_name,last_name, age from customers table whose age lies in the range 20-30

```
SELECT first_name,last_name,age
FROM Customers
WHERE age BETWEEN 20 AND 30;
```

Note: The **NOT BETWEEN** operator is used to exclude the rows that match the values in the range. It returns all the rows except the excluded rows.

IN

- ❖ Used to compare a value to a specified value in a list
- ❖ The IN operator selects values that match any one values given in the list

Example:

Select rows if the country lies in following list USA,UK,Nepal ,India,Pakistan

```
SELECT  *
FROM Customers
WHERE country IN ('USA', 'UK',’Nepal’,’India’,’Pakistan’);
```

Note: The **NOT IN** operator is used to exclude the rows that match values in the list. It returns all the rows except the excluded rows

LIKE

- ✓ The SQL LIKE operator is used with the WHERE clause to get a result set that matches the given string pattern.
- ✓ The pattern includes combination of wildcard characters and regular characters

Example :

```
SELECT  *
FROM Customers
WHERE last_name LIKE 'r%';
```

- ✓ Here, % (means zero or more characters) is a wildcard character.
- ✓ Hence, the SQL command selects customers whose last_name starts with r followed by zero or more characters after it.

Wildcard Characters

Symbol	Description	Example
%	Represents zero or more characters	bl% finds bl, black, blue,
_	Represents a single character	h_t finds hot, hat, and hit
[]	Represents any single character within the brackets	h[oa]t finds hot and hat, but not hit
^	Represents any character not in the brackets	h[^oa]t finds hit but not hot and hat
-	Represents any single character within the specified range	c[a-b]t finds cat and cbt

Here are some examples showing different LIKE operators with '%' and '_' wildcards:

LIKE Operator	Description
WHERE first_name LIKE 'a%'	Finds first_name that starts with "a"
WHERE first_name LIKE '%a'	Finds first_name that ends with "a"
WHERE first_name LIKE '%or%'	Finds first_name that have "or" in any position
WHERE first_name LIKE '_r%'	Finds first_name that have "r" in the second position
WHERE first_name LIKE 'a_ _%'	Finds first_name that starts with "a" and are at least 3 characters in length
WHERE first_name LIKE 'a%h'	Finds first_name that starts with "a" and ends with "h"

NULL values

- ✓ The term NULL in SQL is used to specify that a data value does not exist in the database.
- ✓ 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.

Some common reasons why a value may be NULL

- ❖ The value may not be provided during the data entry.
- ❖ The value is not yet known.

- ✓ 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.

IS NULL

The IS NULL operator is used to test for empty values (NULL values).

Syntax:

```
SELECT column_names
FROM table_name
WHERE column_name IS NULL;
```

Example:

The following SQL lists first_name and last_name of all customers with a NULL value in the "country" field

```
SELECT first_name,last_name
FROM Customers
WHERE country IS NULL;
```

IS NOT NULL

The IS NOT NULL operator is used to test for non-empty values (NOT NULL values).

Syntax

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

The following SQL lists first_name,last_name and country of all customers with a value in the "country " field:

```
SELECT first_name,last_name,country
FROM Customers
WHERE country IS NOT NULL;
```

SQL SELECT DISTINCT

- ✓ The SQL SELECT DISTINCT statement retrieves distinct values from a database table.

Example 1:

Select the unique ages from the Customers table

```
SELECT DISTINCT age
FROM Customers;
```

Example 2:

- ✓ select the unique countries from the customers table

```
SELECT DISTINCT country
FROM Customers;
```

SQL DISTINCT With Multiple Columns

- ✓ We can also use SELECT DISTINCT with multiple columns.

Select rows if the first name and country of a customer is unique

```
SELECT DISTINCT country, first_name
FROM Customers;
```

Rename operation

- ✓ The AS command is used to rename a column or table with an alias.
- ✓ An alias only exists for the duration of the query.
- ✓ We can also use aliases with more than one column**.**

Example1:

```
SELECT first_name AS name
FROM Customers;
```

Here, the SQL command selects the first_name column of Customers. However, the column name will change to name in the result set.

Example2:

```
SELECT customer_id AS cid, first_name AS name
FROM Customers;
```

Here, the SQL command selects customer_id as cid and first_name as name

Sorting Results

- ✓ 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.

ORDER BY Syntax

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

Example:

The following SQL statement selects all customers from the "Customers" table, sorted by the "Country" column:

```
SELECT *
FROM Customers
ORDER BY country;
```

The following SQL statement selects all customers from the "Customers" table, sorted DESCENDING by the "Country" column.

```
SELECT * FROM
Customers
ORDER BY country DESC;
```

ORDER BY Several Columns

The following SQL statement selects all customers from the "Customers" table, sorted by the "country" and the "first_name" column. This means that it orders by Country, but if some rows have the same Country, it orders them by first_name:

Example

```
SELECT * FROM Customers

ORDER BY country,first_name;
```

The following SQL statement selects all customers from the "Customers" table, sorted ascending by the "Country" and descending by the "first_name" column:

Example

```
SELECT * FROM Customers
ORDER BY country ASC, first_name DESC;
```

Aggregate functions

An aggregate function in SQL returns one value after calculating multiple values of a column

Let us consider the following relation

Employee(employee_id,name,deparment,position,salary);

COUNT()

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

Syntax:

```
SELECT COUNT(column_name)
FROM table_name
WHERE condition;
```

Example:

```
SELECT COUNT( DISTINCT employee_id)
FROM Employee;
```

AVG()

- ✓ The AVG() function returns the average value of a numeric column.

Syntax:

```
SELECT AVG(column_name)
FROM table_name
WHERE condition;
```

Example:

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

SUM()

- ✓ The SUM() function returns the total sum of a numeric column.

Syntax:

```
SELECT SUM(column_name)
FROM table_name
WHERE condition;
```

Example:

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

MIN()

- ✓ The MIN() function returns the smallest value of the selected column

Syntax:

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

Example:

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

MAX()

- ✓ The MAX() function returns the largest value of the selected column

Syntax:

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

Example:

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

GROUP BY and HAVING clause

GROUP BY

- ✓ The GROUP BY statement groups rows that have the same values into summary rows, like "find the number of Employees in each department".
- ✓ 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.

Let us consider the following table

Table: Employee

eid	name	address	dept_name	salary
1	Hari	Butwal	Civil	62000
2	Shyam	Kathmandu	Computer	80000
3	Sita	Pokhara	Civil	90000
4	Ramesh	Kathmandu	IT	32000
5	Riya	Pokhara	Computer	76000
6	Dinesh	Kathmandu	Civil	94000
7	Srijana	Butwal	IT	68000

As an illustration consider the query "find the average salary of employee in each department"

```
SELECT dept_name, avg(salary) as average_salary
FROM Employee
GROUP BY dept_name;
```

dept_name	average_salary
Civil	82000
Computer	78000
IT	50000

HAVING clause

- ✓ SQL HAVING clause is similar to the WHERE clause; they are both used to filter rows in a table based on conditions.
- ✓ However, the HAVING clause was included in SQL to filter grouped rows instead of single rows.
- ✓ These rows are grouped together by the GROUP BY clause, so, the HAVING clause must always be followed by the GROUP BY clause.
- ✓ It can be used with aggregate functions, whereas the WHERE clause cannot**.**

As an illustration consider the query "find the name department where the average salary is greater than 60000"

```
SELECT dept_name, avg(salary) as average_salary
FROM employee
GROUP BY dept_name
HAVING avg(salary)>60000;
```

dept_name	average_salary
Civil	82000
Computer	78000

HAVING clause vs WHERE clause

HAVING clause	WHERE clause
HAVING Clause is used to filter record from the groups based on the specified condition.	WHERE Clause is used to filter the records from the table based on the specified condition.
HAVING Clause cannot be used without GROUP BY Clause	WHERE Clause can be used without GROUP BY Clause
HAVING Clause can contain aggregate function	WHERE Clause cannot contain aggregate function
HAVING Clause can only be used with SELECT statement	WHERE Clause can be used with SELECT, UPDATE, DELETE statement.
HAVING Clause implements in column operation	WHERE Clause implements in row operations

Let's take a look at another example, for following relations

```
Customers(customer_id,first_name,last_name,country)
Orders(order_id,product,amount,customer_id)
```

We can write a WHERE clause to filter out rows where the value of amount in the Orders table is less than 500:

```
SELECT customer_id, amount
FROM Orders
WHERE amount < 500;
```

But with the HAVING clause, we can use an aggregate function like SUM to calculate the sum of amounts in the Orders table and get the total order value of less than 500 for each customer:

```
SELECT customer_id, SUM(amount) AS total
FROM Orders
GROUP BY customer_id
HAVING SUM(amount) < 500;
```

Sub Query (Inner Query/Nested Query)

- ✓ A Subquery or Inner query or a Nested query is a query within another SQL query and embedded within clauses, most commonly in the WHERE clause.
- ✓ It is used to return data from a table, and this data will be used in the main query as a condition to further restrict the data to be retrieved.
- ✓ Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN etc.

Consider the following relation

```
employee(emp_id,name,age,department,salary)
```

Subqueries with SELECT statement

Display information of employee whose salary is greater than average salary of all employees

```
select *
from employee
where salary>(select avg(salary) from employee);
```

Display the information of employees whose salary is greater than 26000

```
select *
from employee
where emp_id in (select emp_id from employee where salary>26000);
```

Display information of employee whose salary is greater than at least one employee of IT department.

```
select *
from employee
where salary>some(select salary from employee where department ='IT ');
```

Display information of employee whose salary is greater than that of all employee of IT department.

```
select *
from employee
where salary>all(select salary from employee where department ='IT ');
```

Subqueries with UPDATE statement

Increase salary of employees by 10% whose salary is greater than the average salary of all employees.

```
update employee
 set salary=salary*1.1
 where salary> (select avg(salary) from employee);
```

Subqueries with DELETE statement

Delete the information of employees whose salary is less than average salary of all employees

```
delete from employee where salary < (select avg(salary) from employee);
```

Subqueries with INSERT statement

The INSERT statement uses the data returned from the subquery to insert into another table.

Suppose we want to make each employee board member of company whose department is 'finance' and age>55

Consider a table Boardmember with similar structure as Employee table. Now to copy the records of employee table whose department is 'finance' and age>55 into the Boardmember table, we can use the following syntax.

```
insert into Boardmember
select *
 from employee
where emp_id in (select emp_id from employee  where department='finance' and age>55);
```

Set operations

- ✓ In SQL, set operation is used to combine the two or more SQL SELECT statements.
- ✓ They allow the results of multiple SELECT queries to be combined into single result set.
- ✓ SQL set operators enable the comparison of rows from multiple tables or a combination of results from multiple queries
- ✓ There are certain rules which must be followed to perform operations using SET operators in SQL. Rules are as follows:
  - ☞ The number of columns in the SELECT statement on which you want to apply the SQL set operators must be the same.
  - ☞ The order of columns must be in the same order.
  - ☞ The selected columns must have the same data type.

UNION

- ✓ The SQL Union operation is used to combine the result of two or more SQL SELECT queries.
- ✓ The union operation eliminates the duplicate rows from its result set.

Syntax:

```
SELECT column_name(s) FROM table_1
UNION
SELECT column_name(s) FROM table_2;
```

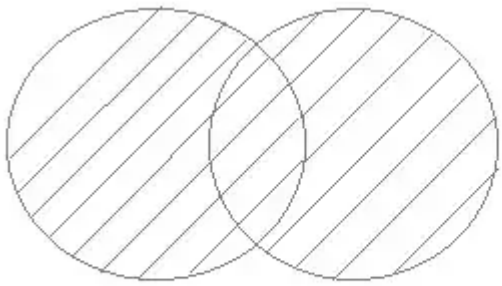


figure: pictorial representation of UNION operation

Let us consider the following two relations.

tbl_first

id	name
1	riya
2	durga
3	anish

tbl_second

id	name
3	anish
4	roshan
5	rojina

Example:

```
SELECT * FROM tbl_first
UNION
SELECT * FROM tbl_second;
```

Output:

id	name
1	riya
2	durga
3	anish
4	roshan
5	rojina

UNION ALL

- ✓ It is similar to Union but it also shows the duplicate rows.

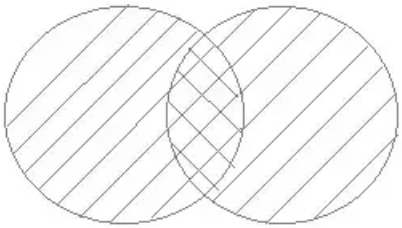


figure: Pictorial representation of UNION ALL operation

Let us consider the following two relations.

tbl_first

id	name
1	riya
2	durga
3	anish

tbl_second

id	name
3	anish
4	roshan
5	rojina

Example:

```
SELECT * FROM tbl_first
UNION ALL
SELECT * FROM tbl_second;
```

Output

id	name
1	riya
2	durga
3	anish
3	anish
4	roshan
5	rojina

INTERSECT

- ✓ The Intersect operation returns the common rows from both the SELECT statements.
- ✓ It has no duplicates and it arranges the data in ascending order by default.

Syntax:

```
SELECT column_name(s) FROM table_1
INTERSECT
SELECT column_name(s) FROM table_2;
```

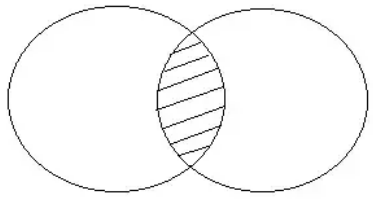


figure: pictorial representation of INTERSECT operation

Let us consider the following two relations.

tbl_first

id	name
1	riya
2	durga
3	anish

tbl_second

id	name
3	anish
4	roshan
5	rojina

Example:

```
SELECT * FROM tbl_first
INTERSECT
SELECT * FROM tbl_second;
```

Output

id	name
3	anish

EXCEPT

- ✓ EXCEPT operator is used to display the rows which are present in the first query but absent in the second query.
- ✓ It has no duplicates and data arranged in ascending order by default.

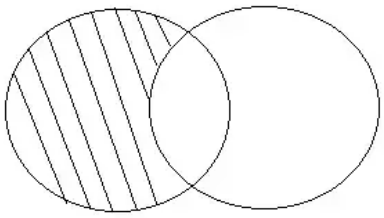


figure: pictorial representation of EXCEPT operation

Syntax:

```
SELECT column_name(s) FROM table_1
EXCEPT
SELECT column_name(s) FROM table_2;
```

Let us consider the following two relations.

tbl_first

id	name
1	riya
2	durga
3	anish

tbl_second

id	name
3	anish
4	roshan
5	rojina

Example:

```
SELECT * FROM tbl_first
EXCEPT
SELECT * FROM tbl_second;
```

Output

id	name
1	riya
2	durga

Join

- ✓ In SQL, a join is an operation that combines rows from two or more tables based on a related column between them.
- ✓ It allows you to retrieve data from multiple tables simultaneously by establishing a relationship between them.
- ✓ Joins are typically performed using the JOIN keyword in an SQL query.

There are different types of joins that can be used:

Before performing join operations, let us consider the following table

Employee

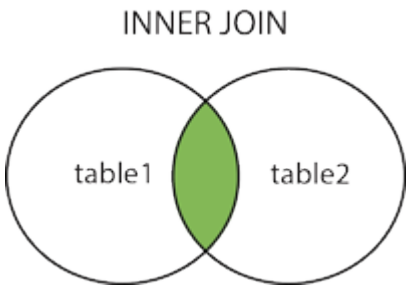
emp_id	emp_name	salary	dept_id
1	anish	25000	1
2	sita	55000	2
3	ronit	40000	4
4	riya	50000	5

Department

dept_id	dept_name
1	sales
2	marketing
3	finance
4	operations

Inner Join

- ✓ Returns only the rows that have matching values in both tables. It combines rows from the tables based on the specified join condition.
- ✓ It is the simple and most popular form of join and assumes as a default join.
- ✓ If we omit the INNER keyword with the JOIN query, we will get the same output.



Syntax:

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

Example 1:

```
SELECT  *
FROM Employee INNER JOIN Department
ON
Employee.dept_id=Department.dept_id;
```

Output:

emp_id	emp_name	salary	dept_id	dept_id	dept_name
1	anish	25000	1	1	sales
2	sita	55000	2	2	marketing
3	ronit	40000	4	4	operations

Example 2:

```
SELECT  Employee.emp_name,Department.dept_name
FROM Employee INNER JOIN Department
ON
Employee.dept_id=Department.dept_id;
```

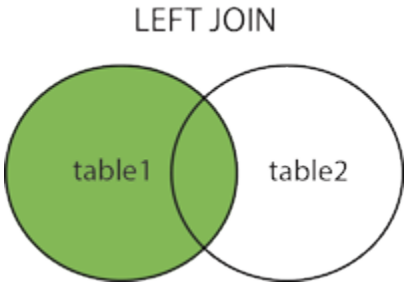
Output:

emp_name	dept_name
Anish	sales
Sita	marketing
ronit	operations

Outer JOIN

- ✓ An outer join is a type of join operation in SQL that includes unmatched rows from one or both tables in the join result.
- ✓ Unlike an inner join, which only returns matching rows, an outer join ensures that all rows from one table (or both tables) are included in the result set, even if there is no corresponding match in the other table.
  There are three types of outer joins:

1) LEFT JOIN (LEFT OUTER JOIN): Returns all rows from the left table and the matching rows from the right table. If there are no matching rows in the right table, NULL values are returned for the columns of the right table.



Syntax:

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

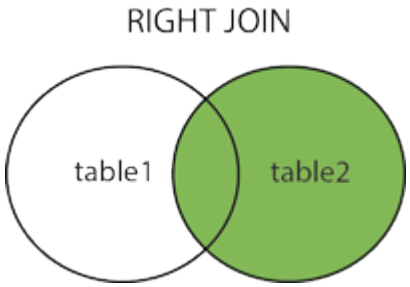
Example:

```
SELECT *
FROM Employee LEFT JOIN Department
ON
Employee.dept_id=Department.dept_id;
```

Output:

emp_id	emp_name	salary	dept_id	dept_id	dept_name
1	anish	25000	1	1	sales
2	sita	55000	2	2	marketing
3	ronit	40000	4	4	operations
4	riya	50000	5	NULL	NULL

2) RIGHT JOIN (RIGHT OUTER JOIN): Returns all rows from the right table and the matching rows from the left table. If there are no matching rows in the left table, NULL values are returned for the columns of the left table.



Syntax:

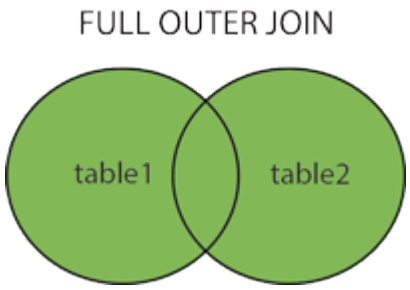
```
SELECT column_name(s)
FROM table1 RIGHT JOIN table2
ON
table1.column_name = table2.column_name;
```

Example:

```
SELECT *
FROM Employee RIGHT JOIN Department
ON
Employee.dept_id=Department.dept_id;
```

emp_id	emp_name	salary	dept_id	dept_id	dept_name
1	anish	25000	1	1	sales
2	sita	55000	2	2	marketing
NULL	NULL	NULL	NULL	3	finance
3	ronit	40000	4	4	operations

3)FULL JOIN (FULL OUTER JOIN): Returns all rows from both tables, regardless of whether they have a match or not. If there is no match, NULL values are returned for the columns of the table that does not have a match.



Syntax:

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

Example:

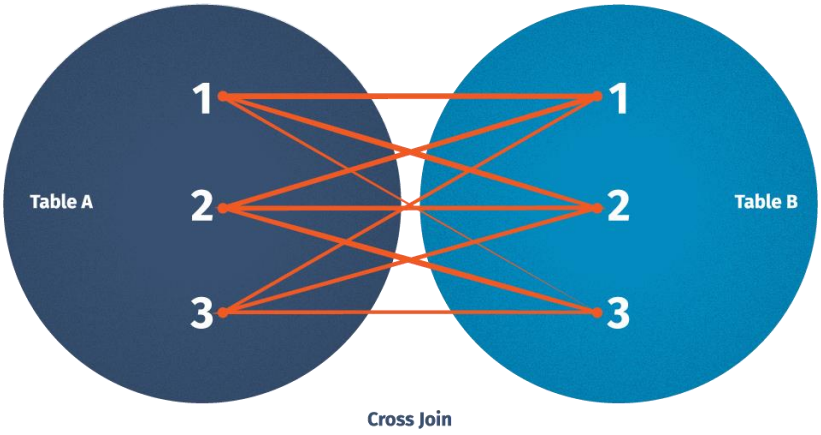
```
SELECT *
FROM Employee FULL JOIN DEPARTMENT
ON
Employee.dept_id=Department.dept_id;
```

Output

emp_id	emp_name	salary	dept_id	dept_id	dept_name
1	anish	25000	1	1	sales
2	sita	55000	2	2	marketing
3	ronit	40000	4	4	operations
4	riya	50000	5	NULL	NULL
Null	Null	Null	Null	3	finance

CROSS JOIN:

- ✓ Returns the Cartesian product of the two tables, which means it combines every row from the first table with every row from the second table.
- ✓ It does not require a join condition.
- ✓ Here each row is the combination of rows of both tables:



Syntax:

```
SELECT column(s) name
FROM table1 CROSS JOIN table2;
```

Example:

SELECT *
FROM Employee CROSS JOIN Department;

emp_id	emp_name	salary	dept_id	dept_id	dept_name
1	anish	25000	1	1	sales
2	sita	55000	2	1	sales
3	ronit	40000	4	1	sales
4	riya	50000	5	1	sales
1	anish	25000	1	2	marketing
2	sita	55000	2	2	marketing
3	ronit	40000	4	2	marketing
4	riya	50000	5	2	marketing
1	anish	25000	1	3	finance
2	sita	55000	2	3	finance
3	ronit	40000	4	3	finance
4	riya	50000	5	3	finance
1	anish	25000	1	4	operations
2	sita	55000	2	4	operations

3	ronit	40000	4	4	operations
4	riya	50000	5	4	operations

NATURAL JOIN

- ✓ The NATURAL JOIN is a type of join in SQL that automatically matches columns with the same name in the joined tables.
- ✓ It eliminates the need to specify the join condition explicitly.
- ✓ The resulting join will include only one instance of columns with the same name. It automatically eliminates duplicate columns from the join result.

Syntax:

```
SELECT column(s) name
FROM table1 NATURAL JOIN table2;
```

Example

```
SELECT *
 FROM Employee NATURAL JOIN Department;
```

Output

dept_id	emp_id	emp_name	salary	dept_name
1	1	anish	25000	sales
2	2	sita	55000	marketing
4	3	ronit	40000	operations

It's important to note that the NATURAL JOIN relies on columns having the same name and data types in both tables.

SELF JOIN:

- ✓ Joins a table with itself, treating it as two separate tables.
- ✓ It is useful when you want to compare rows within the same table.
- ✓ We can do this with the help of table name aliases to assign a specific name to each table's instance. The table aliases enable us to use the table's temporary name that we are going to use in the query.

Syntax:

```
SELECT column_name(s)
FROM table1 T1, table1 T2
WHERE condition;
```

T1 and T2 are different table aliases for the same table.
Example:

Let us consider the following table Employee

emp_id	emp_name	Salary	emp_supervisor
1	anish	25000	5
2	sita	55000	1
3	ronit	40000	4
4	riya	50000	1

The emp_supervisor column refers to the employee_id of the employee's supervisor. Now the above data shows:

- sita and riya supervisor is anish
- ronit supervisor is riya

In the following example, we will use the table EMPLOYEE twice and in order to do this we will use the alias of the table.
To get the list of employees and their supervisor the following SQL statement has used:

Example:

```
select t1.emp_name as Employee,t2.emp_name as Supervisor
 from Employee t1,Employee t2
where t1.emp_supervisor=t2.emp_id;
```

Output:

Employee	Supervisor
sita	anish
ronit	riya
Riya	anish

Query By Example (QBE)

- ✓ QBE stands for Query By Example and it was developed in 1970 by Moshe Zloof at IBM.
- ✓ It is based on Domain relational calculus.
- ✓ It is a graphical query language where we get a user interface and then we fill some required fields to get our proper result.
- ✓ In QBE we don't write complete queries like SQL or other database languages it comes with some blank so we need to just fill that blanks and we will get our required result.
- ✓ In SQL we will get an error if the query is not correct but in the case of QBE if the query is wrong either we get a wrong answer or the query will not be going to execute but we will never get an error.

Queries in QBE are expressed by using skeleton tables, which show the relational schema of database.
Skelton table looks like

Relation name	Column1	Column2	Column3	……	Column n
Tuple or row operation					

Let us consider the following schema

Sailors (sid, sname, age)
Boats(bid, bname,color)
Reserves(sid,bid,day)

To print the names and ages of all sailors, we would create the following example table:

Sailors	Sid	sname	Age
		P._N	P._A

- ✓ In above table _N and _A are variables.
- ✓ The use of variables is optional.
- ✓ We can write Just P.
- ✓ P. means print

Display all records of sailor

Sailors	sid	same	Age
P.			

OR

Sailors	sid	sname	Age
	P.	P.	P.

Display name of sailors whose age is greater than 50

Sailors	Sid	sname	age
		P.	>50

We can order the presentation of the answers through the use of the .AO (for ascending order) and .DO for descending order) commands in conjunction with P. An optional integer argument allows us to sort on more than one field.

For example, we can display the names and ages, of all sailors in ascending order by age, and for each age, in ascending order by name as follows.

Sailors	Sid	sname	age
		P.AO(2)	P.AO(1)

To find sailors with a reservation, we have to combine information from the Sailors and the Reserves relations. In particular we have to select tuples from the two relations with the same value in the join column sid. We do this by placing the same variable in the sid columns of the two example relation

Find the sailors who have reserved a boat for 8/24/96 and who are older than 25,we could write

Sailors	sid	sname	age
	_Id	P._S	>25

Reserves	sid	bid	day
	_Id		'8/24/96'

Stored procedure

- ✓ A stored procedure is a named collection of SQL statements that are precompiled and stored in a database.
- ✓ If the user has an SQL query that you write over and over again, keep it as a stored procedure and execute it.
- ✓ Users can also pass parameters to a stored procedure so that the stored procedure can act based on the parameter value that is given.
- ✓ Based on the statements in the procedure and the parameters we pass, it can perform one or multiple DML operations on the database, and return value, if any.
- ✓ Thus, it allows us to pass the same statements multiple times, thereby, enabling reusability.

Advantages

- ✓ **Reusability:** Once a stored procedure is created, it can be called multiple times from different parts of an application or by multiple users.
- ✓ **Improved performance**: Since stored procedures are precompiled and stored in the database, they can execute faster than sending individual SQL statements from an application to the database server. This is because the database server doesn't have to re-parse and optimize the code each time it is executed.
- ✓ **Reduced network traffic:** The server only passes the procedure name and parameter instead of the whole query, reducing network traffic.
- ✓ **Easy to modify:** WE can easily change the statements in a stored procedure as per necessary.
- ✓ **Security:** Stored procedures can provide an additional layer of security by allowing access to the underlying data through the procedure while restricting direct access to the tables.
- ✓ **Modularity and encapsulation:** Stored procedures enable the modularization and encapsulation of database logic, making it easier to maintain and update the database code.
- ✓ **Parameterization**: Stored procedures can accept input parameters, allowing you to pass values into the procedure at runtime. These parameters can be used within the SQL statements to make the procedure more flexible and reusable

Creating stored procedure

To create a stored procedure in **MySQL**, we can use the following syntax:

```
DELIMITER //
CREATE PROCEDURE procedure_name(parameter1 datatype, parameter2 datatype, ...)
BEGIN
  ----Statements using the input parametes
END //
DELIMITER ;
```

Note:

- ✓ **DELIMITER //** is used to change the delimiter temporarily so that you can use the semicolon ; within the procedure body without ending the entire statement prematurely.
- ✓ **DELIMITER ;** sets the delimiter back to the default semicolon ;

Executing stored procedure

To execute a stored procedure in MySQL, you can use the CALL statement followed by the name of the procedure and list of parameters if any. The syntax is as follows:

CALL procedure_name(parameter_list);

Creating stored procedure without parameters

Example:

```
DELIMITER //
CREATE PROCEDURE  getallEmployee ()
BEGIN
SELECT * FROM employee;
END //
DELIMITER ;
```

Now, we can execute the above stored procedure as follows

```
CALL getallEmployee();
```

Creating parameterized stored procedure

In SQL, a parameterized procedure is a type of stored procedure that can accept input parameters. These parameters can be used to customize the behavior of the procedure and perform operations based on the input values provided.

To create a parameterized stored procedure in MySQL, we can define input parameters within the procedure definition.

Example:

```
DELIMITER //
CREATE PROCEDURE getdepartmentEmployee (dept varchar(30))
BEGIN
SELECT *
FROM employee
WHERE department=dept;
END //
DELIMITER ;
```

To call this stored procedure, you can use the CALL statement and pass the parameter value:

```
CALL getdepartmentEmployee('civil');
```

Drop procedure

We can use the DROP PROCEDURE statement followed by the name of the procedure.

Here's the syntax:

```
DROP PROCEDURE procedure_name;
```

Example:

DROP PROCEDURE getdepartmentEmployee;