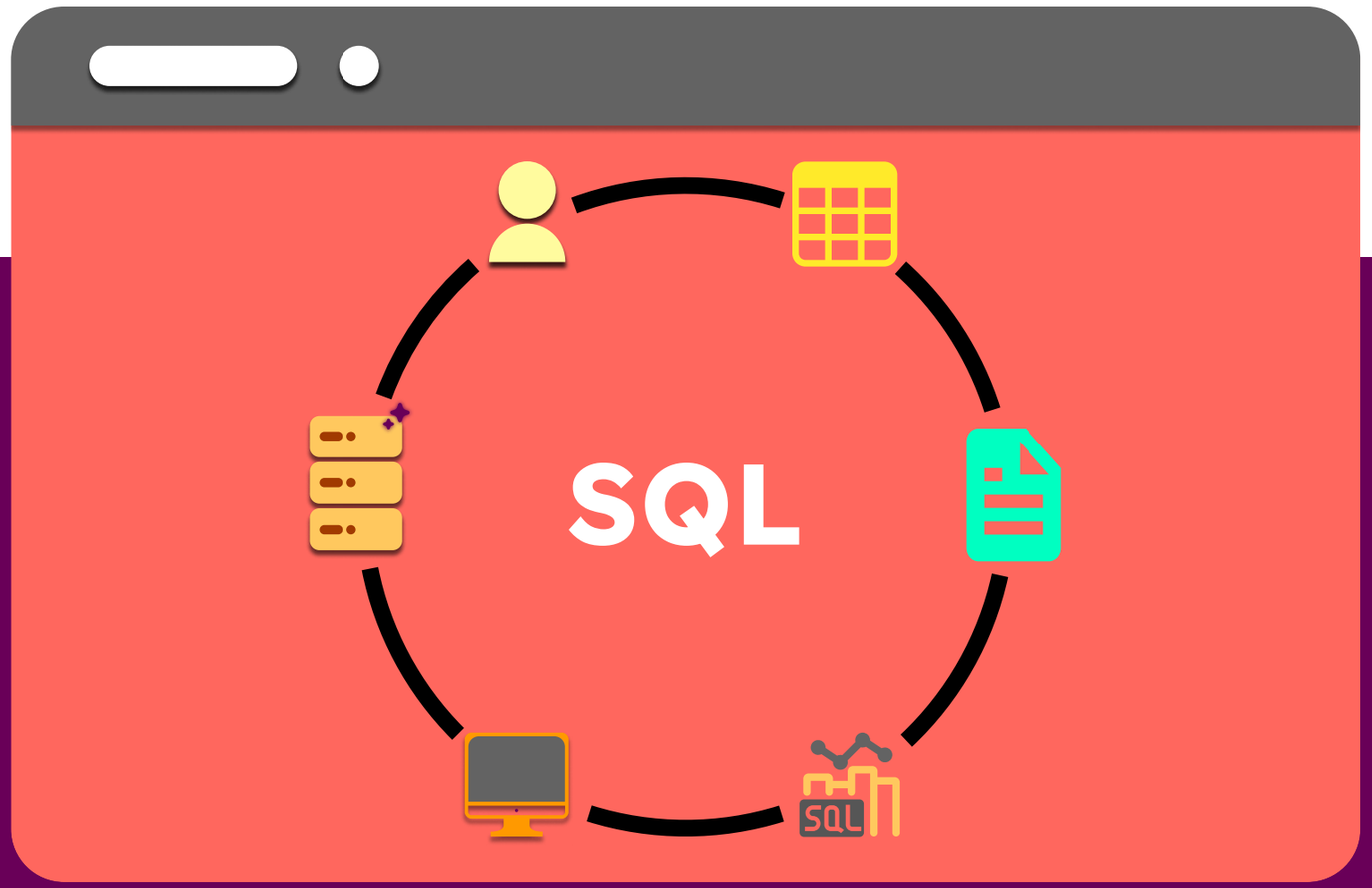
Brij Kishore Pandey

A COMPREHENSIVE GUIDE

# SQL (Structured Query Language)




MASTER SQL IN THE SHORTEST POSSIBLE TIME

swipe

# WHAT IS SQL?

SQL is a database computer language used to manage data stored in relational databases. SQL stands for Structured Query Language.

SQL was developed in the 1970s by IBM Computer Scientists. It became a standard of the ANSI in 1986, and of the ISO in 1987.

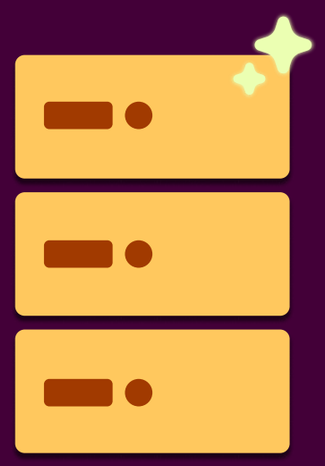
All relational databases like MySQL, MS Access, Oracle, Sybase, Postgres and SQL Server use SQL as their standard database language.

# SQL COMMANDS

## DDL

These are SQL commands that are used to define the database schema. DDL stands for Data Definition Language.

DDL COMMANDS - CREATE, DROP, ALTER, TRUNCATE



## DML

These are SQL commands that are used to manipulate the data present in the database. DML stands for Data Manipulation Language.

DML COMMANDS - INSERT, UPDATE, DELETE

# SQL COMMANDS

## DCL

These are SQL commands that deal with rights, permissions, and other controls of the database system. DCL stands for Data Control Language.

DCL COMMANDS - GRANT, REVOKE

## TCL

These are SQL commands that deal with the transactions happening in the database. TCL stands for Transaction Control Language.

TCL COMMANDS - COMMIT, ROLLBACK

# SQL COMMANDS

## DQL

These are SQL commands that are used to retreive data present in the database. DQL stands for Data Query Language.

DQL COMMANDS - SELECT

# DATABASE QUERIES

## CREATE DATABASE

This statement is used to create a new SQL database.

```
CREATE DATABASE DATABASENAME;
```

**EXAMPLE**

```
CREATE DATABASE COMPANY;
```

## USE DATABASE

This statement is used to select an SQL database for usage.

```
USE DATABASENAME;
```

**EXAMPLE**

```
USE COMPANY;
```

# DATABASE QUERIES

## LIST DATABASES

This statement is used to list down all the available databases.

```
SHOW DATABASES;
```

## DROP DATABASE

This statement is used to delete an existing database along with all the data such as tables, views, indexes, etc.

```
DROP DATABASE DATABASENAME;
```

**EXAMPLE**

```
DROP DATABASE COMPANY;
```

# CREATE TABLE

This statement is used to create a new table in a database.

```
CREATE TABLE TABLENAME(
COLUMN-1 DATATYPE,
COLUMN-2 DATATYPE,...,
COLUMN-N DATATYPE);
```

**EXAMPLE**

```
CREATE TABLE EMPLOYEES(
EMPID INT,
FIRSTNAME VARCHAR(70),
LASTNAME VARCHAR(70),
CITY VARCHAR(50)
SALARY DECIMAL(10, 2));
```

This creates an empty table named EMPLOYEES with all the fields mentioned.

# INSERT VALUES

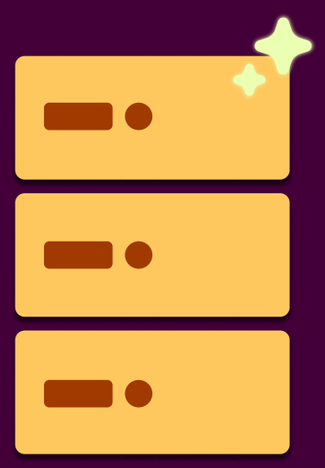
This statement is used to insert new records in a table.

```
INSERT INTO TABLENAME VALUES(
VALUE1, VALUE2,..., VALUE3);
```

**EXAMPLE**

```
INSERT INTO EMPLOYEES VALUES(
10000, 'FRANCES', 'BERGER', '
ALABAMA', 52750);

INSERT INTO EMPLOYEES VALUES(
10001, 'MICHAEL', '
THOMPSON', 'FLORIDA', 48450);
```



Here, we have inserted two records into the table. You must enter values in the order of the fields.

# SELECT QUERY

This statement is used to retrieve data and display it as a table. These tables are called result-sets.

## DISPLAY ALL COLUMNS

```
SELECT * FROM TABLENAME;
```

## EXAMPLE

```
SELECT * FROM EMPLOYEES;
```

## OUTPUT

| EMPID | FIRSTNAME | LASTNAME | CITY | SALARY |
|---|---|---|---|---|
| 10000 | FRANCES | BERGER | ALABAMA | 52750.00 |
| 10001 | MICHAEL | THOMPSON | FLORIDA | 48450.00 |

Here, the employees table is displayed with the inserted records. You can specify the column names instead of asterisk (*) to display specific columns of a table.

# EMPLOYEES TABLE

Consider this table for the next few concepts explained.

**EXAMPLE TABLE**

| EMPID | FIRSTNAME | LASTNAME | CITY | SALARY |
|---|---|---|---|---|
| 10000 | MITCHELL | JOHNSON | ALABAMA | 52750.00 |
| 10001 | MICHAEL | WILLIAMS | FLORIDA | 48450.00 |
| 10002 | JAMES | BROWN | ALABAMA | 26320.00 |
| 10003 | WILLIAM | SMITH | OHIO | 40000.00 |
| 10004 | OSCAR | JONES | GEORGIA | 35850.00 |
| 10005 | JAKE | BROWN | NEVADA | 43700.00 |
| 10006 | JOHN | THOMPSON | FLORIDA | 55000.00 |
| 10007 | DAVID | SMITH | TEXAS | 23450.00 |
| 10008 | MICHAEL | SMITH | GEORGIA | 38900.00 |
| 10009 | SHANE | JOHNSON | OHIO | 32000.00 |

# SELECT DISTINCT

This statement is used to return only distinct (different) values.

```
SELECT DISTINCT COLUMN1,
COLUMN2, ... FROM TABLENAME;
```

CONSIDER THE EMPLOYEES TABLE

QUERY

```
SELECT DISTINCT CITY FROM
EMPLOYEES;
```

OUTPUT

| CITY |
| --- |
| ALABAMA |
| FLORIDA |
| OHIO |
| GEORGIA |
| NEVADA |
| TEXAS |

# WHERE CLAUSE

This statement is used to filter records. It is used to extract only those records that fulfill a specified condition.

```
SELECT COLUMN1, COLUMN2, ..
FROM TABLENAME WHERE CONDITION;
```

CONSIDER THE EMPLOYEES TABLE

**QUERY**

```
SELECT * FROM EMPLOYEES WHERE
SALARY > 50000;
```

**OUTPUT**

| EMPID | FIRSTNAME | LASTNAME | CITY | SALARY |
|---|---|---|---|---|
| 10000 | MITCHELL | JOHNSON | ALABAMA | 52750.00 |
| 10006 | JOHN | THOMPSON | FLORIDA | 55000.00 |

# ORDER BY

This statement is used to sort the result set in ascending or descending order.

```
SELECT * FROM TABLENAME
ORDER BY COLUMN1, COLUMN2, ASC|
DESC;
```

**QUERY**

```
SELECT * FROM EMPLOYEES
ORDER BY SALARY;
```

CONSIDER THE EMPLOYEES TABLE - OUTPUT IN NEXT PAGE

By default, the ORDER BY statement sorts the data in ascending order. You can specify the DESC keyword to sort it in descending order.

# ORDER BY

## OUTPUT

| EMPID | FIRSTNAME | LASTNAME | CITY | SALARY |
|---|---|---|---|---|
| 10007 | DAVID | SMITH | TEXAS | 23450.00 |
| 10002 | JAMES | BROWN | ALABAMA | 26320.00 |
| 10009 | SHANE | JOHNSON | OHIO | 32000.00 |
| 10004 | OSCAR | JONES | GEORGIA | 35850.00 |
| 10008 | MICHAEL | SMITH | GEORGIA | 38900.00 |
| 10003 | WILLIAM | SMITH | OHIO | 40000.00 |
| 10005 | JAKE | BROWN | NEVADA | 43700.00 |
| 10001 | MICHAEL | WILLIAMS | FLORIDA | 48450.00 |
| 10000 | MITCHELL | JOHNSON | ALABAMA | 52750.00 |
| 10006 | JOHN | THOMPSON | FLORIDA | 55000.00 |

# AND OPERATOR

The AND operator is used to filter records based on more than one condition. The records must match both the condition.

```
SELECT COLUMNS FROM
TABLENAME WHERE
CONDITION1 AND CONDITION2;
```

CONSIDER THE EMPLOYEES TABLE

EXAMPLE

```
SELECT EMPID FROM EMPLOYEES
WHERE LASTNAME = 'SMITH' AND
SALARY > 30000;
```

OUTPUT

| EMPID |
| --- |
| 10003 |
| 10008 |

# OR OPERATOR

The OR operator is used to filter records based on more than one condition. The records must match any of the conditions.

```
SELECT COLUMNS FROM
TABLENAME WHERE
CONDITION1 OR CONDITION2;
```

CONSIDER THE EMPLOYEES TABLE

EXAMPLE

```
SELECT FIRSTNAME FROM EMPLOYEES
WHERE CITY = 'NEVADA' OR
CITY = 'TEXAS';
```

OUTPUT

| FIRSTNAME |
| --- |
| JAKE |
| DAVID |

# NOT OPERATOR

The NOT operator is used in combo with other operators to give the opposite result. It is also called as the negative result.

```
SELECT COLUMNS FROM TABLENAME
WHERE NOT CONDITION;
```

CONSIDER THE EMPLOYEES TABLE

EXAMPLE

```
SELECT * FROM EMPLOYEES
WHERE NOT SALARY > 30000;
```

OUTPUT

| EMPID | FIRSTNAME | LASTNAME | CITY | SALARY |
|---|---|---|---|---|
| 10002 | JAMES | BROWN | ALABAMA | 26320.00 |
| 10007 | DAVID | SMITH | TEXAS | 23450.00 |

# AGGREGATE FUNCTIONS

An aggregate function is a function that performs a calculation on a set of values, and returns a single value.

Aggregate functions are often used with the GROUP BY clause of the SELECT statement.

The GROUP BY clause splits the result-set into groups of values and the aggregate function can be used to return a single value for each group.

The most commonly used SQL aggregate functions are: MIN, MAX, COUNT, SUM, AVG.

# MIN

The MIN function returns the smallest value of the selected column.

```
SELECT MIN(COLUMNNAME) FROM
TABLENAME WHERE CONDITION;
```

CONSIDER THE EMPLOYEES TABLE

EXAMPLE

```
SELECT MIN(SALARY) FROM
EMPLOYEES;
```

OUTPUT

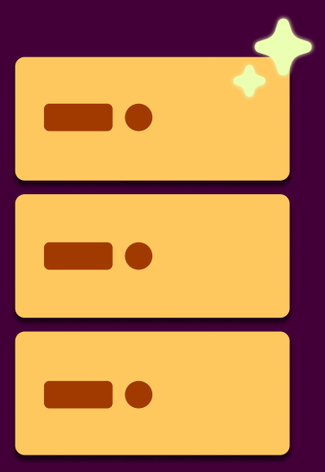
| MIN(SALARY) |
| --- |
| 23450.00 |

# MAX

The MAX function returns the largest value of the selected column.

```
SELECT MAX(COLUMNNAME) FROM
TABLENAME WHERE CONDITION;
```

CONSIDER THE EMPLOYEES TABLE



EXAMPLE

```
SELECT MAX(SALARY) FROM
EMPLOYEES;
```

OUTPUT

| MAX(SALARY) |
| --- |
| 55000.00 |

# COUNT

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

```
SELECT COUNT(COLUMNNAME) FROM
TABLENAME WHERE CONDITION;
```

CONSIDER THE EMPLOYEES TABLE

EXAMPLE

```
SELECT COUNT(*) FROM EMPLOYEES
WHERE CITY = 'FLORIDA';
```

OUTPUT

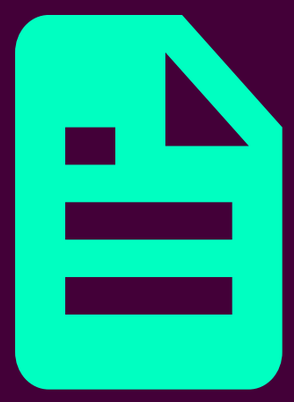
| COUNT(*) |
| --- |
| 2 |

# SUM

The SUM function returns the total sum of a numeric column.

```
SELECT SUM(COLUMNNAME) FROM
TABLENAME WHERE CONDITION;
```

CONSIDER THE EMPLOYEES TABLE



EXAMPLE

```
SELECT SUM(SALARY)
FROM EMPLOYEES
WHERE SALARY > 50000;
```

OUTPUT

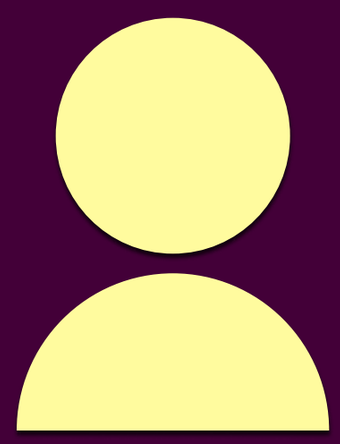
| SUM(SALARY) |
| --- |
| 107750.00 |

# AVG

The AVG function returns the average of a numeric column.

```
SELECT AVG(COLUMNNAME) FROM
TABLENAME WHERE CONDITION;
```

CONSIDER THE EMPLOYEES TABLE



**EXAMPLE**

```
SELECT AVG(SALARY)FROM
EMPLOYEES;
```

**OUTPUT**

| AVG(SALARY) |
| --- |
| 39642 |

# GROUP BY

The GROUP BY statement groups rows that have the same values into summary rows.

The GROUP BY statement is often used with aggregate functions to group the result set by one or more columns.

```
SELECT COLUMNNAME(S) FROM
TABLENAME
WHERE CONDITION GROUP BY
COLUMN_NAME(S);
```

CONSIDER THE EMPLOYEES TABLE

EXAMPLE

```
SELECT COUNT(EMPID), CITY
FROM EMPLOYEES GROUP BY CITY;
```

# GROUP BY

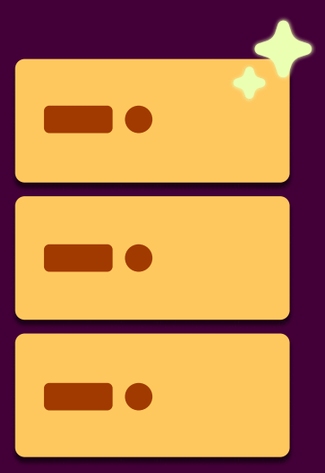
## OUTPUT

| COUNT(EMPID) | CITY |
|---|---|
| 2 | ALABAMA |
| 2 | FLORIDA |
| 2 | OHIO |
| 2 | GEORGIA |
| 1 | NEVADA |
| 1 | TEXAS |

# HAVING CLAUSE

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

```
SELECT COLUMNNAME(S) FROM
TABLENAME WHERE CONDITION
GROUP BY
COLUMNNAME(S) HAVING CONDITION;
```



## CONSIDER THE EMPLOYEES TABLE

### EXAMPLE

```
SELECT COUNT(EMPID), CITY FROM
EMPLOYEES GROUP BY CITY HAVING
COUNT(EMPID) < 2 ;
```

### OUTPUT

| COUNT(EMPID) | CITY |
|---|---|
| 1 | NEVADA |
| 1 | TEXAS |

# LIKE OPERATOR

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column. There are two wildcards often used with the LIKE operator.

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

```
SELECT COLUMN-1,COLUMN-2
FROM TABLENAME WHERE
COLUMN-N LIKE PATTERN;
```

**EXAMPLE**

```
SELECT * FROM EMPLOYEES
WHERE FIRSTNAME LIKE 'J%';
```

# LIKE OPERATOR

## OUTPUT

| EMPID | FIRSTNAME | LASTNAME | CITY | SALARY |
|---|---|---|---|---|
| 10002 | JAMES | BROWN | ALABAMA | 26320.00 |
| 10005 | JAKE | BROWN | NEVADA | 43700.00 |
| 10006 | JOHN | THOMPSON | FLORIDA | 55000.00 |

Here, the query selects all the first names that start with J. You can use these wildcards in different formats to achieve different results.

# IN OPERATOR

The IN operator allows you to specify multiple values in a WHERE clause. The IN operator is a shorthand for multiple OR conditions.

```
SELECT COLUMNNAME(S)
FROM TABLENAME
WHERE COLUMNNAME IN (VALUES 1,
2, ...);
```

## CONSIDER THE EMPLOYEES TABLE

### QUERY

```
SELECT * FROM EMPLOYEES
WHERE CITY IN ('OHIO', 'NEVADA');
```

### OUTPUT

| EMPID | FIRSTNAME | LASTNAME | CITY | SALARY |
|---|---|---|---|---|
| 10003 | WILLIAM | SMITH | OHIO | 40000.00 |
| 10005 | JAKE | BROWN | NEVADA | 43700.00 |
| 10009 | SHANE | JOHNSON | OHIO | 32000.00 |

# BETWEEN OPERATOR

The BETWEEN operator selects values within a given range. The values can be numbers, text, or dates. It is inclusive: begin and end values are included.

```
SELECT COLUMNNAME(S) FROM TABLENAME
WHERE COLUMNNAME
BETWEEN VALUE1 AND VALUE2;
```

CONSIDER THE EMPLOYEES TABLE

QUERY

```
SELECT * FROM EMPLOYEES WHERE
SALARY BETWEEN 20000 AND 32000;
```

OUTPUT

| EMPID | FIRSTNAME | LASTNAME | CITY | SALARY |
|---|---|---|---|---|
| 10002 | JAMES | BROWN | ALABAMA | 26320.00 |
| 10007 | DAVID | SMITH | TEXAS | 23450.00 |
| 10009 | SHANE | JOHNSON | OHIO | 32000.00 |

# UPDATE QUERY

This statement is used to modify the existing records in a table.

```
UPDATE TABLENAME SET
COLUMN1 = VALUE1, COLUMN2 =
VALUE2 WHERE CONDITION;
```

CONSIDER THE EMPLOYEES TABLE

QUERY

```
UPDATE EMPLOYEES
SET FIRSTNAME = 'LUKE'
WHERE EMPID = 10003;
```

UPDATED ROW

| EMPID | FIRSTNAME | LASTNAME | CITY | SALARY |
|---|---|---|---|---|
| 10003 | LUKE | SMITH | OHIO | 40000.00 |

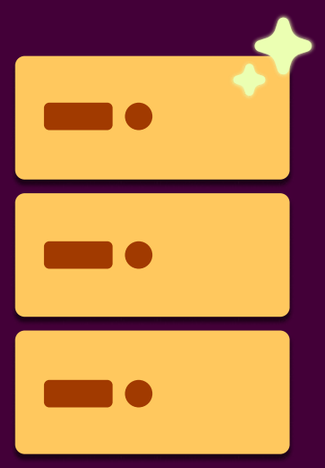
The updated row is only displayed here. You must use the select query to view the full result.

# DELETE QUERY

This statement is used to delete existing records in a table.

```
DELETE FROM TABLENAME WHERE
CONDITION;
```

CONSIDER THE EMPLOYEES TABLE



QUERY

```
DELETE FROM EMPLOYEES WHERE
EMPID = 10008;
```

DELETED ROW

| EMPID | FIRSTNAME | LASTNAME | CITY | SALARY |
|---|---|---|---|---|
| 10008 | MICHAEL | SMITH | GEORGIA | 38900.00 |

The deleted row is just displayed here. You must use the select query to view the full result.

# DROP TABLE

This statement is used to drop an existing table in a database.

```
DROP TABLE TABLENAME;
```

**EXAMPLE**

```
DROP TABLE EMPLOYEES;
```

**TRUNCATE TABLE**

This statement is used to delete the data inside a table, but not the table itself.

```
TRUNCATE TABLE TABLENAME;
```

**EXAMPLE**

```
TRUNCATE TABLE EMPLOYEES;
```

# ALTER TABLE

This statement is used to add, delete, or modify columns in an existing table and also used to add and drop various constraints on an existing table.

## ADD COLUMN

```
ALTER TABLE TABLENAME ADD
COLUMNNAME DATATYPE;
```

### EXAMPLE

```
ALTER TABLE EMPLOYEES ADD
BONUS DECIMAL(10, 2);
```

### EXTRA COLUMN ADDED

| EMPID | FIRSTNAME | LASTNAME | CITY | SALARY | BONUS |
|---|---|---|---|---|---|

# ALTER TABLE

## DROP COLUMN

```
ALTER TABLE TABLENAME DROP
COLUMN COLUMNNAME;
```

### EXAMPLE

```
ALTER TABLE EMPLOYEES DROP
COLUMN BONUS;
```

### BONUS COLUMN DELETED

| EMPID | FIRSTNAME | LASTNAME | CITY | SALARY |
|---|---|---|---|---|

## RENAME COLUMN

```
ALTER TABLE TABLENAME RENAME
COLUMN OLDNAME TO NEWNAME;
```

# ALTER TABLE

EXAMPLE

```
ALTER TABLE EMPLOYEES RENAME
COLUMN FIRSTNAME TO FNAME;
```

FIRSTNAME COLUMN RENAMED

| EMPID | FNAME | LASTNAME | CITY | SALARY |
|---|---|---|---|---|

# MODIFY DATATYPE

```
ALTER TABLE TABLENAME MODIFY
COLUMN COLUMNNAME DATATYPE;
```

EXAMPLE

```
ALTER TABLE EMPLOYEES MODIFY
COLUMN CITY VARCHAR(100);
```

This query will modify and increase the number of characters for the city column.

# 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.

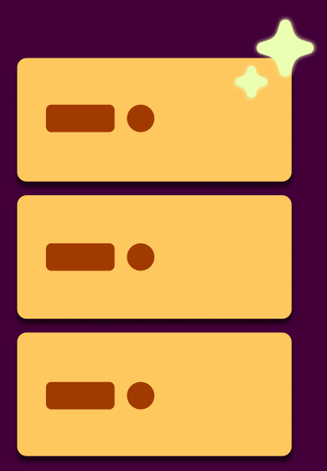
Constraints can be column level or table level. Column level constraints apply to a column, and table level constraints apply to the whole table.

# CONSTRAINTS

## NOT NULL

This constraint enforces a column to not accept NULL values.

It enforces a field to always contain a value, which means that you cannot insert a new record, or update a record without adding a value to this field.



### EXAMPLE

```
CREATE TABLE EMPLOYEES (
EMPID INT NOT NULL,
FIRSTNAME VARCHAR(70),
LASTNAME VARCHAR(70),
CITY VARCHAR(50), SALARY
DECIMAL(10, 2) NOT NULL);
```

The EMPID and SALARY field is NOT NULL here.

# CONSTRAINTS

## UNIQUE

This constraint ensures that all values in a column are different.

Both the UNIQUE and PRIMARY KEY constraints provide a guarantee for uniqueness for a column or set of columns.

### EXAMPLE

```
ALTER TABLE EMPLOYEES
ADD UNIQUE (EMPID);
```

Here, we have used ALTER to add a UNIQUE constraint to the EMPID column. The EMPID column can not take duplicate values.

# CONSTRAINTS

## PRIMARY KEY

This constraint uniquely identifies each record in a table. Primary keys must contain UNIQUE values, and cannot contain NULL values.

A table can have only one primary key, and in the table, this primary key can consist of single or multiple columns.

**EXAMPLE**

```
CREATE TABLE EMPLOYEES (EMPID
INT, FIRSTNAME VARCHAR(70),
LASTNAME VARCHAR(70), CITY
VARCHAR(50), SALARY DECIMAL(10,
2),PRIMARY KEY (EMPID));
```

Here, EMPID column is the primary key of the table.

# CONSTRAINTS

## CHECK

This 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.

### EXAMPLE

```
CREATE TABLE EMPLOYEES (EMPID
INT, FIRSTNAME VARCHAR(70),
LASTNAME VARCHAR(70), AGE INT
CHECK (AGE >= 18));
```

Here, we have defined a CHECK constraint for the AGE column where the age must be greater or equal to 18. Anything below that wouldn't be accepted.

# CONSTRAINTS

## DEFAULT

This constraint is used to set a default value for a column. The default value will be added to all new records, if no other value is specified.

### EXAMPLE

```
CREATE TABLE EMPLOYEES (EMPID
INT, FIRSTNAME VARCHAR(70),
LASTNAME VARCHAR(70), CITY
VARCHAR(50) DEFAULT 'ALABAMA',
SALARY DECIMAL(10, 2));
```

Here, the CITY column would be filled with the value ALABAMA if there are no values explicitly specified.

# EXAMPLE TABLES

Consider these two tables for the next few concepts explained.

## PERSONS TABLE

| PID | FIRSTNAME | LASTNAME | AGE |
|---|---|---|---|
| 1 | MARCO | JANSEN | 28 |
| 2 | ASHIF | BASHA | 24 |
| 3 | MONISH | KUMAR | 26 |

## ORDERS TABLE

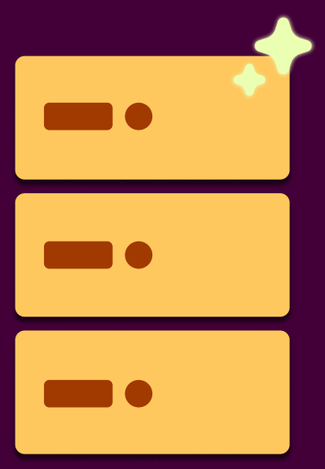
| OID | ORDERNO | PID |
|---|---|---|
| 1 | 34578 | 2 |
| 2 | 45876 | 1 |
| 3 | 55900 | 2 |
| 4 | 22089 | 3 |

# CONSTRAINTS

## FOREIGN KEY

This constraint is used to prevent actions that would destroy links between tables.

The **FOREIGN KEY** is a field (or collection of fields) in one table, that refers to the **PRIMARY KEY** in another table.



CONSIDER BOTH THE EXAMPLE TABLES

QUERY

```
ALTER TABLE ORDERS ADD
FOREIGN KEY (PID)
REFERENCES PERSONS(PID);
```

# ANY OPERATOR

The ANY operator returns a boolean value as a result and returns TRUE if any of the subquery values meet the condition.

This means that the condition will be true if the operation is true for any of the values in the range.

```
SELECT * FROM TABLENAME WHERE
COLUMNNAME OPERATOR ANY (SELECT
COLUMNNAME FROM TABLENAME
WHERE CONDITION);
```

**QUERY**

```
SELECT FIRSTNAME FROM PERSONS
WHERE PID = ANY(SELECT PID
FROM ORDERS WHERE ORDERNO >
40000);
```

# ANY OPERATOR

**OUTPUT**

| FIRSTNAME |
| --- |
| MARCO |
| ASHIF |

Here, the ORDERNO is greater than 40000 for PID 1 and 2 in the ORDERS table. So, it displays their respective first names from the PERSONS table.

# ALL OPERATOR

The ALL operator returns a boolean value as a result and returns TRUE if all of the subquery values meet the condition.

ALL means that the condition will be true only if the operation is true for all values in the range.

```
SELECT * FROM TABLENAME WHERE
COLUMNNAME OPERATOR ALL (SELECT
COLUMNNAME FROM TABLENAME
WHERE CONDITION);
```

**QUERY**

```
SELECT FIRSTNAME FROM PERSONS
WHERE PID = ALL(SELECT PID
FROM ORDERS WHERE ORDERNO >
40000);
```

# ALL OPERATOR

## OUTPUT

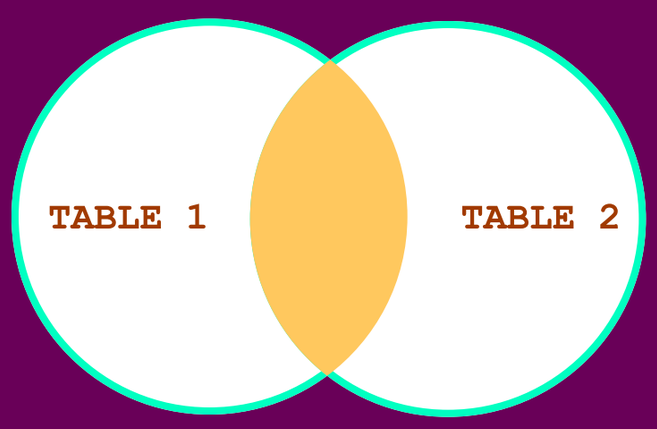
```
EMPTY TABLE
```

Here, the condition does not match all the records in the ORDERS table. So, it displays an empty table with no records.

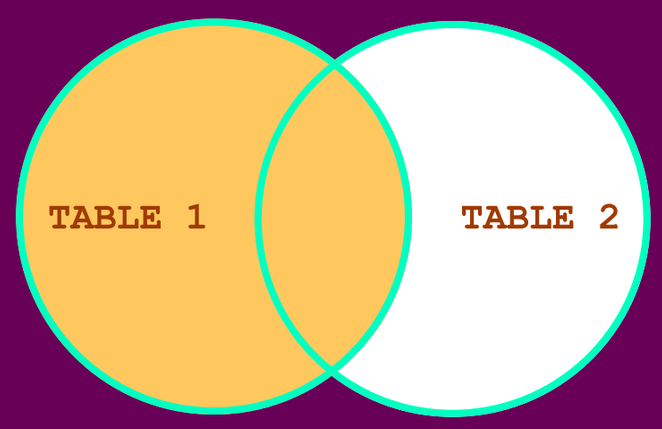
# SQL JOINS

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

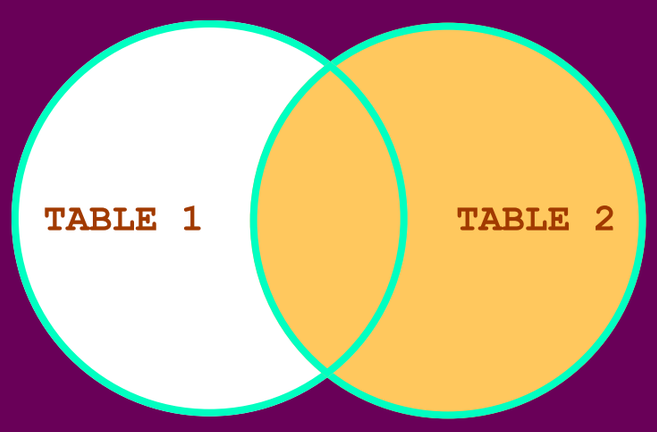
## TYPES OF JOINS



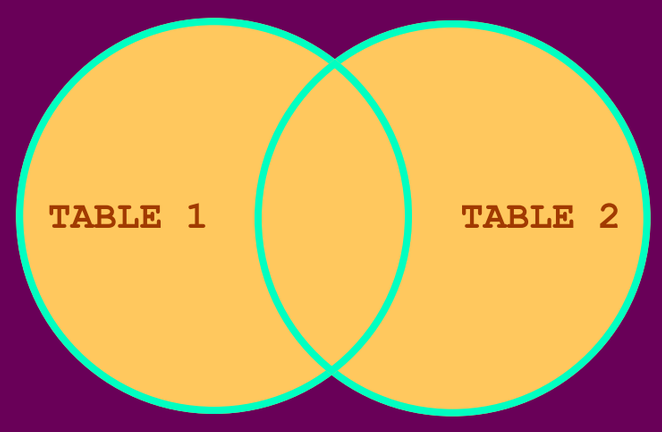

INNER JOIN




LEFT JOIN




RIGHT JOIN



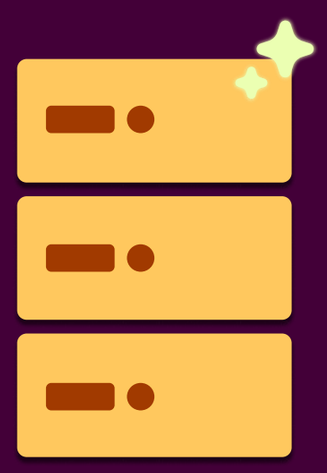

FULL OUTER JOIN

# INNER JOIN

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

```
SELECT COLUMNNAME(S) FROM
TABLE1 INNER JOIN
TABLE2 ON TABLE1.COLNAME =
TABLE2.COLNAME;
```

CONSIDER BOTH THE EXAMPLE TABLES



QUERY

```
SELECT FIRSTNAME, AGE, OID
FROM PERSONS
INNER JOIN ORDERS ON
PERSONS.PID = ORDERS.PID;
```

# INNER JOIN

## OUTPUT

| FIRSTNAME | AGE | OID |
|---|---|---|
| MARCO | 28 | 2 |
| ASHIF | 24 | 3 |
| ASHIF | 24 | 1 |
| MONISH | 26 | 4 |

# LEFT JOIN

The **LEFT JOIN** keyword returns all records from the left table, and the matching records from the right table. The result is **0 records** from the right side, if there is no match.

```
SELECT COLUMNNAME(S) FROM
TABLE1 LEFT JOIN
TABLE2 ON TABLE1.COLNAME =
TABLE2.COLNAME;
```

**CONSIDER BOTH THE EXAMPLE TABLES**

**QUERY**

```
SELECT LASTNAME, ORDERS.PID,
ORDERNO FROM PERSONS LEFT JOIN
ORDERS ON PERSONS.PID =
ORDERS.PID;
```

# LEFT JOIN

## OUTPUT

| LASTNAME | PID | ORDERNO |
|---|---|---|
| JANSEN | 1 | 45876 |
| BASHA | 2 | 34578 |
| BASHA | 2 | 55900 |
| KUMAR | 3 | 22089 |

# RIGHT JOIN

The RIGHT JOIN keyword returns all records from the right table, and the matching records from the left table. The result is 0 records from the left side, if there is no match.

```
SELECT COLUMNNAME(S) FROM
TABLE1 RIGHT JOIN
TABLE2 ON TABLE1.COLNAME =
TABLE2.COLNAME;
```

CONSIDER BOTH THE EXAMPLE TABLES

QUERY

```
SELECT LASTNAME, ORDERS.PID,
ORDERNO FROM PERSONS RIGHT JOIN
ORDERS ON PERSONS.PID =
ORDERS.PID;
```

# RIGHT JOIN

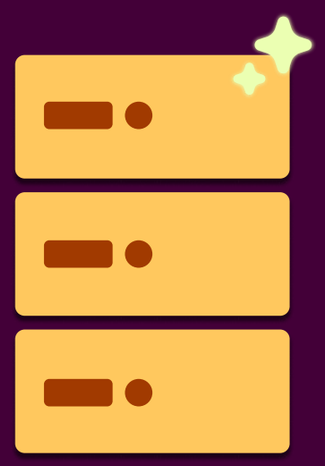
## OUTPUT

| LASTNAME | PID | ORDERNO |
|---|---|---|
| BASHA | 2 | 34578 |
| JANSEN | 1 | 45876 |
| BASHA | 2 | 55900 |
| KUMAR | 3 | 22089 |

# FULL OUTER JOIN

The FULL OUTER JOIN or FULL JOIN keyword returns all records when there is a match in left or right table records.

```
SELECT COLUMNNAME(S) FROM
TABLE1 FULL OUTER JOIN
TABLE2 ON TABLE1.COLNAME =
TABLE2.COLNAME WHERE CONDITION;
```



CONSIDER BOTH THE EXAMPLE TABLES

QUERY

```
SELECT LASTNAME, ORDERS.PID,
ORDERNO FROM PERSONS RIGHT JOIN
ORDERS ON PERSONS.PID =
ORDERS.PID;
```

# FULL OUTER JOIN

## OUTPUT

| LASTNAME | PID | ORDERNO |
|---|---|---|
| JANSEN | 1 | 45876 |
| BASHA | 2 | 34578 |
| BASHA | 2 | 55900 |
| KUMAR | 3 | 22089 |

# ALIASES

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

An alias only exists for the duration of that query. An alias is created with the AS keyword.

```
SELECT COLUMNNAME AS ALIASNAME
FROM TABLENAME;
```

**QUERY**

```
SELECT OID AS ID FROM ORDERS;
```

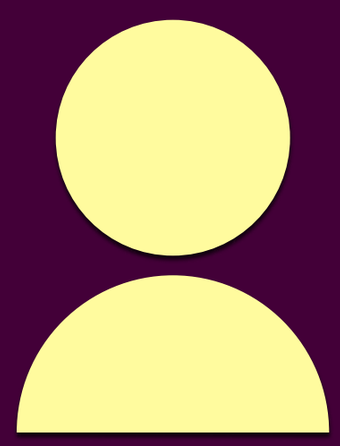
**ORDERS TABLE**

| ID |
|---|
| 1 |
| 2 |
| 3 |
| 4 |

# STORED PROCEDURE

A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.

If you have an SQL query that you write over and over again, save it as a stored procedure, and then just call it to execute it.



You can also pass parameters to a stored procedure, so that the stored procedure can act based on the parameter value(s) that is passed.

# STORED PROCEDURE

```
CREATE PROCEDURE PROCEDURENAME
AS
SQLSTATEMENT
GO;
```

## QUERY

```
CREATE PROCEDURE ALLPERSONS
AS
SELECT * FROM PERSONS
GO;
```

## EXECUTE STORED PROCEDURE

```
EXEC ALLPERSONS;
```

## OUTPUT

| PID | FIRSTNAME | LASTNAME | AGE |
|---|---|---|---|
| 1 | MARCO | JANSEN | 28 |
| 2 | ASHIF | BASHA | 24 |
| 3 | MONISH | KUMAR | 26 |

# For More Interesting Content

## Brij Kishore Pandey

# Follow Me On LinkedIn

https://www.linkedin.com/in/brijpandeyji/