SQL COMMANDS

- SQL-LIKE CLAUSE
- SQL - Order By
- SQL - Group By
- SQL - Distinct Keyword
- SQL - Sorting Results
- SQL- CONSTRAINTS
- ARITHMETIC OPERATORS
- AS CLAUSE

SQL-LIKE CLAUSE

- The SQL **LIKE** clause is used to compare a value to similar values using wildcard operators. There are two wildcards used in conjunction with the LIKE operator:
  - **The percent sign (%)**
  - **The underscore (_)**

Syntax:

The basic syntax of % and _ is as follows:

SELECT FROM table_name **WHERE** column **LIKE** 'XXXX%'

or

SELECT FROM table_name **WHERE** column **LIKE** '%XXXX%'

or

SELECT FROM table_name **WHERE** column **LIKE** '_XXXX%'

Continued….

Statement	Description
WHERE SALARY LIKE '200%'	Finds any values that start with 200
WHERE SALARY LIKE '%200%'	Finds any values that have 200 in any position
WHERE SALARY LIKE '_00%'	Finds any values that have

SQL **ORDER BY** clause

The SQL **ORDER BY** clause is used to sort the data in ascending or descending order, based on one or more columns. Some database sorts query results in ascending order by default.

Syntax:

The basic syntax of ORDER BY clause is as follows:

SELECT column-list **FROM** table_name **[WHERE condition] ORDER BY** [column1, column2, .. columnN] **[ASC | DESC];**

Continued....

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

```
SQL> SELECT * FROM CUSTOMERS
     ORDER BY NAME, SALARY;
```

This would produce the following result:

ID	NAME	AGE	ADDRESS	SALARY
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
3	kaushik	23	Kota	2000.00
2	Khilan	25	Delhi	1500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00
1	Ramesh	32	Ahmedabad	2000.00

Continued....

```
SQL> SELECT * FROM CUSTOMERS
     ORDER BY NAME DESC;
```

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
+----+----------+-----+-----------+----------+
```

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
7	Muffy	24	Indore	10000.00
6	Komal	22	MP	4500.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
5	Hardik	27	Bhopal	8500.00
4	Chaitali	25	Mumbai	6500.00

To fetch the rows with own preferred order, the SELECT query would as follows:

```
SQL> SELECT * FROM CUSTOMERS
    ORDER BY (CASE ADDRESS
    WHEN 'DELHI'   THEN 1
    WHEN 'BHOPAL'   THEN 2
    WHEN 'KOTA'   THEN 3
    WHEN 'AHMADABAD' THEN 4
    WHEN 'MP'  THEN 5
    ELSE 100 END) ASC, ADDRESS DESC;
```

SQL **GROUP BY** clause

- The SQL **GROUP BY** clause is used in collaboration with the **SELECT** statement to arrange identical data into groups.

- The **GROUP BY** clause follows the **WHERE** clause in a **SELECT** statement and precedes the **ORDER BY** clause.

Syntax:

SELECT column1, column2 **FROM** table_name **WHERE** [conditions]
GROUP BY column1, column2
ORDER BY column1, column2

GROUP_BY

SELECT `gender` FROM `members`
;

gender

Female

Female

Male

Female

Male

GROUP BY

```
SELECT `gender` FROM `members`
  GROUP BY `gender`;
```

gender

Female

Male

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

```
SQL> SELECT NAME, SUM(SALARY) FROM CUSTOMERS
     GROUP BY NAME;
```

NAME	SUM(SALARY)
Chaitali	6500.00
Hardik	8500.00
kaushik	2000.00
Khilan	1500.00
Komal	4500.00
Muffy	10000.00
Ramesh	2000.00

Now, let us have following table where CUSTOMERS table has the following records with duplicate names:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Ramesh   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | kaushik  |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

```
SQL> SELECT NAME, SUM(SALARY) FROM CUSTOMERS
     GROUP BY NAME;
```

```
+----------+--------------+
| NAME     | SUM(SALARY)  |
+----------+--------------+
| Hardik   |      8500.00 |
| kaushik  |      8500.00 |
| Komal    |      4500.00 |
| Muffy    |     10000.00 |
| Ramesh   |      3500.00 |
+----------+--------------+
```

SQL **DISTINCT** keyword

- The SQL **DISTINCT** keyword is used in conjunction with SELECT statement to eliminate all the duplicate records and fetching only unique records.

- There may be a situation when you have multiple duplicate records in a table. While fetching such records, it makes more sense to fetch only unique records instead of fetching duplicate records.

Syntax:

- **SELECT DISTINCT** column1, column2,.....columnN **FROM** table_name **WHERE [condition]**

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

```
SQL> SELECT SALARY FROM CUSTOMERS
     ORDER BY SALARY;
```

SALARY
1500.00
2000.00
2000.00
4500.00
6500.00
8500.00
10000.00

Now, let us use DISTINCT keyword with the above SELECT query and see the result:

```
SQL> SELECT DISTINCT SALARY FROM CUSTOMERS
     ORDER BY SALARY;
```

```
+----------+
| SALARY   |
+----------+
|  1500.00 |
|  2000.00 |
|  4500.00 |
|  6500.00 |
|  8500.00 |
| 10000.00 |
+----------+
```

SQL - Constraints

- Constraints are the rules enforced on data columns on table.
- These are used to limit the type of data that can go into a table.

- NOT NULL Constraint: Ensures that a column cannot have NULL value.
- DEFAULT Constraint: Provides a default value for a column when none is specified.
- UNIQUE Constraint: Ensures that all values in a column are different.
- PRIMARY Key: Uniquely identified each rows/records in a database table.

Continued……

- FOREIGN Key: Uniquely identified a rows/records in any another database table.
- CHECK Constraint: The CHECK constraint ensures that all values in a column satisfy certain conditions.

Null constraint

```
CREATE TABLE CUSTOMERS(
        ID    INT                  NOT NULL,
        NAME  VARCHAR (20)         NOT NULL,
        AGE   INT                  NOT NULL,
        ADDRESS   CHAR (25) ,
        SALARY    DECIMAL (18, 2),
        PRIMARY KEY (ID)
);
```

If CUSTOMERS table has already been created, then to add a NOT NULL constraint to SALARY column in Oracle and MySQL, you would write a statement similar to the following:

```
ALTER TABLE CUSTOMERS
   MODIFY SALARY  DECIMAL (18, 2) NOT NULL;
```

SQL - DEFAULT Constraint

- The **DEFAULT constraint** provides **a default value** to a column when the **INSERT INTO** statement **does not provide a specific value.**

```
CREATE TABLE CUSTOMERS(
        ID   INT              NOT NULL,
        NAME VARCHAR (20)     NOT NULL,
        AGE  INT              NOT NULL,
        ADDRESS  CHAR (25) ,
        SALARY   DECIMAL (18, 2) DEFAULT 5000.00,
        PRIMARY KEY (ID)
);
```

If CUSTOMERS table has already been created, then to add a DFAULT constraint to SALARY column, you would write a statement similar to the following:

```
ALTER TABLE CUSTOMERS
   MODIFY SALARY  DECIMAL (18, 2) DEFAULT 5000.00;
```

Drop Default Constraint:

To drop a DEFAULT constraint, use the following SQL:

```
ALTER TABLE CUSTOMERS
    ALTER COLUMN SALARY DROP DEFAULT;
```

UNIQUE Constraint

Example:

For example, the following SQL creates a new table called CUSTOMERS and adds five columns. Here, AGE column is set to UNIQUE, so that you can not have two records with same age:

```
CREATE TABLE CUSTOMERS(
       ID   INT              NOT NULL,
       NAME VARCHAR (20)     NOT NULL,
       AGE  INT              NOT NULL UNIQUE,
       ADDRESS  CHAR (25) ,
       SALARY   DECIMAL (18, 2),
       PRIMARY KEY (ID)
);
```

Continued....

If CUSTOMERS table has already been created, then to add a UNIQUE constraint to AGE column, you would write a statement similar to the following:

```
ALTER TABLE CUSTOMERS
   MODIFY AGE INT NOT NULL UNIQUE;
```

You can also use following syntax, which supports naming the constraint in multiple columns as well:

```
ALTER TABLE CUSTOMERS
   ADD CONSTRAINT myUniqueConstraint UNIQUE(AGE, SALARY);
```

DROP a UNIQUE Constraint:

To drop a UNIQUE constraint, use the following SQL:

```
ALTER TABLE CUSTOMERS
    DROP CONSTRAINT myUniqueConstraint;
```

If you are using MySQL, then you can use the following syntax:

```
ALTER TABLE CUSTOMERS
    DROP INDEX myUniqueConstraint;
```

Create Primary Key:

- Here is the syntax to define ID attribute as a primary key in a CUSTOMERS table.
- **CREATE TABLE** CUSTOMERS(ID INT NOT NULL, NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR (25) ,

SALARY DECIMAL (18, 2),

PRIMARY KEY

To create a PRIMARY KEY constraint on the "ID" column
when CUSTOMERS table already exists, use the following SQL syntax:

ALTER TABLE CUSTOMER **ADD PRIMARY KEY (ID);**

For defining a PRIMARY KEY constraint on multiple columns, use the following SQL syntax:

```
CREATE TABLE CUSTOMERS(
       ID   INT              NOT NULL,
       NAME VARCHAR (20)     NOT NULL,
       AGE  INT              NOT NULL,
       ADDRESS  CHAR (25) ,
       SALARY   DECIMAL (18, 2),
       PRIMARY KEY (ID, NAME)
);
```

To create a PRIMARY KEY constraint on the "ID" and "NAMES" columns when CUSTOMERS table already exists, use the following SQL syntax:

```
ALTER TABLE CUSTOMERS
   ADD CONSTRAINT PK_CUSTID PRIMARY KEY (ID, NAME);
```

Delete Primary Key:

- You can clear the primary key constraints from the table, Use Syntax:

ALTER TABLE CUSTOMERS **DROP PRIMARY KEY ;**

```
CREATE TABLE ORDERS (
       ID          INT        NOT NULL,
       DATE        DATETIME,
       CUSTOMER_ID INT references CUSTOMERS(ID),
       AMOUNT     double,
       PRIMARY KEY (ID)
);
```

If ORDERS table has already been created, and the foreign key has not yet been set, use the syntax for specifying a foreign key by altering a table.

```
ALTER TABLE ORDERS
   ADD FOREIGN KEY (Customer_ID) REFERENCES CUSTOMERS (ID);
```

DROP a FOREIGN KEY Constraint:

To drop a FOREIGN KEY constraint, use the following SQL:

```
ALTER TABLE ORDERS DROP
FOREIGN KEY
```

For example, the following SQL creates a new table called CUSTOMERS and adds five columns. Here, we add a CHECK with AGE column, so that you can not have any CUSTOMER below 18 years:

```
CREATE TABLE CUSTOMERS(
        ID   INT              NOT NULL,
        NAME VARCHAR (20)     NOT NULL,
        AGE  INT              NOT NULL CHECK (AGE >= 18),
        ADDRESS  CHAR (25) ,
        SALARY   DECIMAL (18, 2),
        PRIMARY KEY (ID)
);
```

If CUSTOMERS table has already been created, then to add a CHECK constraint to AGE column, you would write a statement similar to the following:

```
ALTER TABLE CUSTOMERS
   MODIFY AGE INT NOT NULL CHECK (AGE >= 18 );
```

You can also use following syntax, which supports naming the constraint in multiple columns as well:

```
ALTER TABLE CUSTOMERS
   ADD CONSTRAINT myCheckConstraint CHECK(AGE >= 18);
```

Using Arithmetic Operators

SELECT ename, sal, sal+300 **FROM**

ENAME	SAL	SAL+300
SMITH	800	1100
ALLEN	1600	1900
WARD	1250	1550
JONES	2975	3275
MARTIN	1250	1550

Operator Precedence

*	/	+	-

Using operator precedence

```
SELECT ename, sal, 12 * sal + 100
  FROM emp ;
```

ENAME	SAL	12*SAL+100
SMITH	800	9700
ALLEN	1600	19300
WARD	1250	15100
JONES	2975	35800
MARTIN	1250	15100

Using Paranthesis

SELECT ename, sal, 12 * (sal + 100)
FROM emp;

ENAME	SAL	12*(SAL+100)
SMITH	800	10800
ALLEN	1600	20400
WARD	1250	16200
JONES	2975	36900
MARTIN	1250	16200
BLAKE	2850	35400
CLARK	2450	30600

Null Values in Arithmetic Expressions

- **Arithmetic expressions containing a null value evaluate to null.**

```
SELECT ename, 12 * sal + comm
FROM emp
WHERE ename ='KING'
```

ENAME	JOB	COMM
SMITH	CLERK	
ALLEN	SALESMAN	300
WARD	SALESMAN	500
JONES	MANAGER	
MARTIN	SALESMAN	1400
BLAKE	MANAGER	
CLARK	MANAGER	
SCOTT	ANALYST	
KING	PRESIDENT	

```
SELECT ename,  12 * sal + comm
FROM emp
WHERE ename ='KING'
```

ENAME	12*SAL+COMM
KING	

Using Column Aliases

```
SELECT ename  AS Ad ,  sal Maaş
FROM emp;
```

AD	MAAŞ
SMITH	800
ALLEN	1600
WARD	1250
JONES	2975
MARTIN	1250

```
SELECT  ename  "Adı",   sal * 12   "Yıllık Ücret"
   FROM     emp ;
```

Adı	Yıllık Üc
SMITH	9600
ALLEN	19200
WARD	15000
JONES	35700

Using the Concatenation Operator

- **SELECT** CONCAT(ename, job) **AS** Employees **FROM** emp ;

Employees
SMITHCLERK
ALLENSALESMAN
WARDSALESMAN
JONESMANAGER
MARTINSALESMAN
BLAKEMANAGER
CLARKMANAGER

Using Literal Character Strings

SELECT CONCAT(ename **'is a' ' '** job) **AS** Employee Details **FROM** emp ;

Employee Details
SMITHis a CLERK
ALLENis a SALESMAN
WARDis a SALESMAN
JONESis a MANAGER
MARTINis a SALESMAN
BLAKEis a MANAGER
CLARKis a MANAGER
SCOTTis a ANALYST
KINGis a PRESIDENT

Using the Aggregate Functions

prod_code	prod_name	prod_desc	prod_price
1	WildTech 250Gb 1700	SATA Disk Drive	120
2	Moto Razr	Mobile Phone	200
3	Microsoft 10-20 Keyboard	Ergonmoc Keyboard	49
4	EasyTech Mouse 7632	Cordless Mouse	49
5	Dell XPS 400	Desktop PC	999
6	Buffalo AirStation Turbo G	Wireless Ethernet Bridge	60
7	Apple iPod Touch	Portable Music/Movie Player	199
8	Apple iPhone 8Gb	Smart Phone	399

Using AVG() Function

```
mysql> SELECT AVG(prod_price) AS price_ag FROM products;
+----------+
| price_ag |
+----------+
|  259.375 |
+----------+
1 row in set (0.00 sec)
```

Continued...

We can also be selective about the rows used in the average calculation by using the *WHERE* clause:

```
mysql> SELECT AVG(prod_price) AS price_avg FROM products WHERE prod_price BETWEEN 10 and 199;
+-----------+
| price_avg |
+-----------+
|     95.4  |
+-----------+
1 row in set (0.00 sec)
```

Using COUNT()

```
mysql> SELECT COUNT(*) FROM products;
+----------+
| price_ag |
+----------+
|        8 |
+----------+
1 row in set (0.00 sec)
```

Continued...

Similarly, we can restrict our criteria to list the number of products beneath a specific price threshold:

```
mysql> SELECT COUNT(prod_price) AS low_price_items FROM products WHERE prod_price < 200;
+-----------------+
| low_price_items |
+-----------------+
|               5 |
+-----------------+
1 row in set (0.00 sec)
```

Using MAX()

```
mysql> SELECT MAX(prod_price) AS max_price FROM products;
+-----------+
| max_price |
+-----------+
|       999 |
+-----------+
1 row in set (0.00 sec)
```

MIN()

```
mysql> SELECT MIN(prod_price) AS min_price FROM products;
+-----------+
| max_price |
+-----------+
|        49 |
+-----------+
1 row in set (0.00 sec)
```

SUM()

```
mysql> SELECT SUM(prod_price) AS total_price FROM products;
+-----------+
| max_price |
+-----------+
|      2075 |
+-----------+
1 row in set (0.00 sec)
```

Using Multiple Aggregate Functions

```
mysql> SELECT MAX(prod_price) AS max_price, MIN(prod_price) AS max_price FROM products;
+-----------+-----------+
| max_price | max_price |
+-----------+-----------+
|       999 |        49 |
+-----------+-----------+
1 row in set (0.00 sec)
```