# CSE 4308
# DBMS LAB
# Documentations

Prepared by,
Sabbir Ahmed
Lecturer,
Department of Computer Science and Engineering (CSE)
Islamic University of Technology (IUT)
Board Bazar, Gazipur-1704
Bangladesh.
Email: iamsabbirahmed12345@gmail.com , sabbirahmed@iut-dhaka.edu
Contact: +8801754221481, +8801630210656

# Table of Contents

# 1 LAB1

## 1.1 SQL BASICS

- How to open SQL Command Prompt?

  Just write 'sqlplus' in the command prompt!
  Or,
  Search 'run SQL command line' in the program search bar.

- How to connect to a database?

  Open SQL, write

  ```
  CONNECT USERNAME/PASSWORD;
  ```
  Or
  Type

  ```
  CONNECT
  ```

  Then there will be prompt for username. Input the username. Then it will ask for password. But now if u type the password, it will not show the characters as oracle keeps it hidden. After inputting the password hit enter and see the system will be connected.

- How to see existing users?

  First u have to connect to the database.

  ```
  SELECT USERNAME FROM DBA_USERS;
  ```

  To see the existing usernames sorted:

  ```
  SELECT USERNAME FROM DBA_USERS ORDER BY USERNAME;
  ```

  To see in which date the user was created:

  ```
  SELECT USERNAME, CREATED FROM DBA_USERS ORDER BY CREATED;
  ```

  The default order is ascending. Do make it appear in descending order you have to use the following query:

  ```
  SELECT USERNAME, CREATED FROM DBA_USERS ORDER BY CREATED DESC;
  ```
  (Here DESC stands for Descending).

When the new user is created, if you try to login with it, Login will be denied as the super user (system) has not given him the privilege to login. There are some privileges in oracle like:

Create session,
Connect,
Resource, (to be able to create table and add data)
DBA, (database admin privilege)
Etc....

- How to grant session privilege?

  GRANT CREATE SESSION TO USERNAME;
  Or
  GRANT CREATE SESSION TO USERNAME WITH ADMIN OPTION.

  To give him both create session and resource privilege:

  GRANT CREATE SESSION, RESOURCE TO USERNAME;

  To give all possible privilege at a single go, you can write:

  GRANT ALL PRIVILEGES TO USERNAME;

  (This is strictly NOT RECOMMENDED as the new user might get some unwanted privileges!)

- How to alter the system password in case of u forgotten it?

  CONN / AS SYSDBA
  ALTER USER USERNAME IDENTIFIED BY NEWPASSWORD;

  Or

  CONN SYSTEM / PASSWORD AS SYSDBA
  ALTER USER USERNAME IDENTIFIED BY NEWPASSWORD;

- How to create a user in oracle?

  CREATE USER username IDENTIFIED BY password

- How to change password?

  ALTER USER username IDENTIFIED BY password;

  But for that you have to login as SUPERUSER! (System/sys/new user who has such privilege!)

- To drop a user:

  DROP USER USERNAME CASCADE;

Here cascade means all the objects (for example: tables) should also be deleted with the removal of the user.

- We can format time in different ways using sql.Some examples are showed below:

```
SQL> select to_char(created, 'DD-MON-YYYY') from dba_users;

TO_CHAR(CRE
-----------
23-JAN-2018
23-JAN-2018
07-FEB-2006
07-FEB-2006
07-FEB-2006
07-FEB-2006
07-FEB-2006
07-FEB-2006
07-FEB-2006
07-FEB-2006

TO_CHAR(CRE
-----------
07-FEB-2006
07-FEB-2006
07-FEB-2006
07-FEB-2006

15 rows selected.

SQL> select to_char(created, 'DD-MONTH-YYYY') from dba_users;

TO_CHAR(CREATED,'
-----------------
23-JANUARY  -2018
23-JANUARY  -2018
07-FEBRUARY -2006
07-FEBRUARY -2006
07-FEBRUARY -2006
07-FEBRUARY -2006
07-FEBRUARY -2006
07-FEBRUARY -2006
07-FEBRUARY -2006
07-FEBRUARY -2006
07-FEBRUARY -2006

TO_CHAR(CREATED,'
-----------------
07-FEBRUARY -2006
07-FEBRUARY -2006
07-FEBRUARY -2006
07-FEBRUARY -2006

15 rows selected.

SQL>
```

```
SQL> select to_char(created, 'DD-MON-YYYY HH24:MI:SS') from dba_users;

TO_CHAR(CREATED,'DD-
--------------------
23-JAN-2018 00:52:15
23-JAN-2018 00:42:39
07-FEB-2006 22:10:13
07-FEB-2006 22:10:13
07-FEB-2006 22:40:15
07-FEB-2006 22:44:47
07-FEB-2006 22:10:24
07-FEB-2006 22:17:03
07-FEB-2006 22:27:15
07-FEB-2006 22:52:43
07-FEB-2006 22:38:38

TO_CHAR(CREATED,'DD-
--------------------
07-FEB-2006 22:35:21
07-FEB-2006 22:52:43
07-FEB-2006 22:40:14
07-FEB-2006 22:51:21

15 rows selected.
```

# 2 LAB2:

## 2.1 Creating table:

If u want to connect SQL from CMD, just type sqlplus!

- How to create a table?

```
CREATE TABLE DEPT (
DEPTID NUMBER,
DEPTNAME VARCHAR2 (20),
CONSTRAINTS PK_DEPT PRIMARY KEY (DEPTID)
);
```

Remember that SQL isn't case sensitive! But the values are

- To see the column names and their types in SQL:

```
DESC TABLE_NAME;
```

- How to show all the tables that are under a particular user?

DESC DBA_TABLES; [this DBA_TABLES is a table containing all the tables under each user.]

```
SELECT TABLE_NAME
FROM DBA_TABLES
WHERE OWNER='USERNAME';
```

- Suppose you want to create a new table 'DEPT' but you are not sure whether the table is already created or not. If it is already existing, oracle will not let u create a new table with the same name.

  How to be sure that whether a table called 'DEPT' is existing or not?

  One solution might be deleting the existing table.

```
DROP TABLE TABLE_NAME;
```

  But this is bad practice as the information might be necessary. So we can search

```
SELECT TABLE_NAME
FROM DBA_TABLES;
```

  But it will return a long list of the existing tables. It is difficult to search my desired table_name from the list.
  Another way is:

```
SELECT TABLE_NAME
FROM DBA_TABLES
WHERE TABLE_NAME='DEPT';
```

IF the DEPT table is existing, some rows will be selected from the table.

- How to disconnect a user?
  Just write DISC
- Syntax of primary key:

  CONSTRAINT *constraint_name* PRIMARY KEY *(column1, column2, ... column_n)*

- How to add primary key to a table which is already created, but suppose u have forgotten to add the primary key?

  Creating a table without primary key:

```
CREATE TABLE STD (
SID NUMBER,
SNAME VARCHAR (20),
);
```

```
ALTER TABLE TABLE_NAME ADD CONSTRAINT CONSTRAINT_NAME PRIMARY KEY (COL_NAME);
```

- How to assign a composite primary key?

```
CONSTRAINT CONSTARINT_NAME PRIMARY KEY (COL1, COL2...);
```

- How to insert values to a table?

  INSERT INTO *table_name* VALUES (*value1*, *value2*, *value3*, ...);

- You can also specify the column names:

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

Remember that SQL is not case sensitive. But it is case sensitive in cases like saved values like passwords.

For inserting string/characters you have to put single quotes 'value'.

- How to update values of a table?

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

```
UPDATE Customers
SET CONTACTNAME = 'Alfred Schmidt', City= 'Frankfurt'
WHERE CUSTOMERID = 1;
```

- HOW TO DELETE VALUES FROM A TABLE?

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

```
DELETE FROM Customers
WHERE CUSTOMERNAME='Alfreds Futterkiste';
```

## 2.2 How to add foreign key to a table?

Suppose we have a table called 'country' which has the columns as C_NAME, C_ID and C_ID is the primary key.

Now If I want to create a table called 'citizen' with the following columns like CITIZENID, CITIZENNAME and COUNTRYID we can define CITIZENID as the primary key of the citizen table. But here COUNTRYID should be referencing the 'country' table. Thus here COUNTRYID in the 'citizen' table is acting as the foreign key.

SQL:

```
CREATE TABLE COUNTRY (
        C_NAME VARCHAR2 (30),
        C_ID NUMBER,
        CONSTRAINTS PK_COUNTRY PRIMARY KEY (C_ID)
);

CREATE TABLE CITIZEN (
        CITIZENID NUMBER,
        CITIZENNAME VARCHAR2 (20),
        COUNTRYID NUMBER,
        CONSTRAINTS PK_CITIZEN PRIMARY KEY (COUNTRYID),
        CONSTRAINTS FK_CITIZEN_COUNTRY FOREIGN KEY (COUNTRYID)
        REFERNECES COUNTRY (C_ID)
);
```

REMEMBER THAT THE PARENT TABLE SHOULD ALWAYS BE CREATED FIRST!

And both of the columns (referencing and referenced) should be of the same types.

Now insert some values to both of the table. Here you'll see that if u want to add a data to citizen table with the C_ID which is not present in the country table, it will show error. This means the foreign key is working properly.

- What will be the constraint if a table contains two foreign keys?

  There should be two foreign key statements referencing each to the tables!

  NOTE:
- Why do we give constraint names?

  Here *pk_dept* is the rule/constraint name. It has many uses. You can write anything in the place of *pk_dept*. but this is the widely practiced naming convention. For example if table named 'student' has the DEPTID as the foreign key, the name of that foreign key constraint might be given as '*fk_student_dept*'.

  One of the use of this constraint name is, suppose you have inserted a value to the student table with student_id=1 where student_id is the primary key. Now if you add another student with the student_id=1, the command prompt will show that, 'pk_dept' rule is violated. In another case, suppose we want to add a row to student

table where the DEPT name is 'abc' which is not a valid department it is not entered in the DEPT table. So in this case command prompt will show that the 'fk_student_dept' constraint is violated.

## 2.3 Alter table

- How to add column to an existing table?

  ALTER TABLE *table_name*
  ADD *column_name datatype*;

- To delete column from an existing table:

  ALTER TABLE *table_name*
  DROP COLUMN *column_name*;

- How to change the datatype of an existing column?

  ALTER TABLE *table_name*
  MODIFY *column_name datatype*;

# 3 LAB4 & LAB5

## 3.1 Select query

- How to show all the values?

  Select * from table_name;

- How to select specific columns from a table?

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

- To select distinct values from a table:

  SELECT DISTINCT C_HOME FROM CITIZEN;

  See the difference with

  SELECT C_HOME FROM CITIZEN;

- Renaming Output Column:

  In SQL, when you write the above query, output will be a column with the name 'C_HOME' containing all the districts. We can rename the output column name for better visualization!
  [Renaming a column in the select query doesn't have any effect on the main table definition. It is only used for visualization]

  SYNTAX:

  SELECT COL_NAME AS NEW_COL_NAME FROM TABLE_NAME;

  EXAMPLE:

  SELECT C_HOME AS HOME_LIST
  FROM CITIZEN;

  SELECT DISTINCT (C_HOME) AS UNIQUE_DISTRICTS
  FROM CITIZEN;

- How can I count the number of unique districts?

  SELECT COUNT (DISTINCT (C_HOME))
  FROM CITIZEN;

  [In this case the output column name will be 'COUNT (DISTINCT (C_HOME) )' which looks odd! So we can rename the output as

  SELECT COUNT (DISTINCT (C_HOME) ) AS TOTAL_UNIQUE_DIST FROM CITIZEN;

Note:
Here count () is an aggregate function which counts the total number of columns presented by the query.

Note:

```
SELECT COUNT (DISTINCT (C_HOME)) FROM CITIZEN;
```

This query can also be written as

```
SELECT COUNT (*) AS TOTAL_UNIQUE_HOME
FROM (SELECT DISTINCT (C_HOME) FROM CITIZEN);
```

[EXPLAINATION: this sort of query is called NESTED query. Here first the inner query in the FROM section is being calculated. Inner query first returns the names of the distinct district names which are 'Dhaka', 'Ctg', 'Comilla', 'Khulna', 'Gazipur'. Then the outer query count the number of rows supplied by the inner query. So output will be 5]

## 3.2 Adding condition to query

If u want to select some values based on a specific condition, u have to use 'where' clause as well!

```
SELECT COL1, COL2...
FROM TABLE_NAME
WHERE CONTIDION;
```

In the where clause you can use operators like, =, >, <, <>, >=, <=, BETWEEN, LIKE, IN etc.

i. Find the citizen names and id who are more than 50 years old

```
SELECT C_ID, C_NAME
FROM CITIZEN
WHERE AGE>50;
```

ii. Find the citizens living in Dhaka

```
SELECT C_NAME AS NAME
FROM CITIZEN
WHERE C_HOME='Dhaka';
```

iii. Find the persons not living in Dhaka

```
SELECT C_NAME, C_HOME FROM CITIZEN WHERE C_HOME<>'Dhaka';
```

< > means! =
[NOTE: here Dhaka is CASE SENSETIVE!]

iv. Find the citizens having salary more than 20,000 and less than 50000

```
SELECT C_NAME, SALARY
FROM CITIZEN
WHERE SALARY BETWEEN 20000 AND 50000;
```

Suppose we want to see the values which are greater than 50000 and less than 20000, then we can use the 'NOT BETWEEN' clause.

```
SELECT C_NAME, SALARY
FROM CITIZEN
WHERE SALARY NOT BETWEEN 20000 AND 50000;
```

If you want to sort the output based on salary:

```
SELECT C_NAME, SALARY
FROM CITIZEN
WHERE SALARY NOT BETWEEN 20000 AND 50000 ORDER BY SALARY;
```

Default order of sorting is ascending. For Descending:

```
SELECT C_NAME, SALARY
FROM CITIZEN
```

```
WHERE SALARY NOT BETWEEN 20000 AND 50000
ORDER BY SALARY DESC;
```

[Note: this query should not necessarily be written in 3 lines, writing like 'SELECT C_ID, C_NAME FROM CITIZEN WHERE AGE>50;' will work also!]

The select clause may also contain arithmetic expressions involving operations +,-,*, /.

For example:

- Give the teachers 5% increment and show their new salary?

```
SELECT C_ID, C_NAME, SALARY*1.05
FROM CITIZEN
WHERE OCCUPATION='Teacher';
```

- How much money should I add to my budget if I want to give the teachers 5% bonus for EID?

```
SELECT SUM (SALARY*0.05) AS TOTAL_ADDITIONAL_BUDGET
FROM CITIZEN
WHERE OCCUPATION='Teacher';
```

## 3.3 SQL USING TWO/MORE TABLES [Natural join]

For better understanding, let's create the following two tables.

```
CREATE TABLE STD (
  STD_ID NUMBER (3),
  STD_NAME VARCHAR (2),
  STD_DEPT NUMBER (3),
  STD_CG NUMBER (3, 2),
  CONSTRAINTS PK_STD PRIMARY KEY (STD_ID),
  CONSTRAINTS FK_STD_DEPT FOREIGN KEY (STD_DEPT) REFERENCES DEPT (DEPT_ID)
);

CREATE TABLE DEPT (
  DEPT_ID NUMBER (3),
  DEPT_NAME VARCHAR (3),
  DEPT_BUILDING VARCHAR (3),
  DEPT_ESTD NUMBER (4),
  CONSTRAINTS PK_DEPT PRIMARY KEY (DEPT_ID)
);
```

Now suppose you need a list of students along with their DEPT_NAME.

SQL:

```
SELECT STD.STD_ID, DEPT.DEPT_NAME
FROM STD, DEPT
WHERE STD.STD_DEPT= DEPT.DEPT_ID;
```

In SQL you can rename the tables as well!
The above query could be written also as:

```
SELECT S.STD_ID, D.DEPT_NAME
FROM STD S, DEPT D
WHERE S.STD_DEPT= D.DEPT_ID;
```

It makes writing queries easier!

## 3.4 USING AND, OR, NOT IS SQL

### 3.4.1 AND:

Syntax:

```
SELECT column1, column2, ...
FROM table_name
WHERE condition1 AND condition2 AND condition3 ...;
```

Example:
Show the list who are from 'DHAKA' AND salary is a Teacher:

```
SELECT *
FROM CITIZEN
WHERE C_HOME='Dhaka' AND OCCUPATION='Teacher';
```

### 3.4.2 OR

Syntax:

```
SELECT column1, column2, ...
FROM table_name
WHERE condition1 OR condition2 OR condition3 ...;
```

Example:

Show the people who are either a teacher or a doctor

```
SELECT *
FROM CITIZEN
WHERE OCCUPATION='Teacher' OR OCCUPATION='Doctor';
```

### 3.4.3 NOT

```
SELECT column1, column2, ...
FROM table_name
WHERE NOT condition;
```

EXAMPLE:

Show the person not from Dhaka:

```
SELECT *
FROM CITIZEN
WHERE NOT C_HOME='Dhaka';
```

LET'S MIX UP THE CLAUSES!

Find the teachers from Dhaka or Ctg:

```
SELECT *
FROM CITIZEN
WHERE OCCUPATION='Teacher' AND (C_HOME='Dhaka' OR C_HOME='Ctg');
```

Show the teachers who are not from Dhaka:

```
SELECT *
FROM CITIZEN
WHERE OCCUPATION='Teacher' AND NOT C_HOME='Dhaka';
```

## 3.5 ORDER BY

Used to sort the output into ascending (default) or descending order.

SYNTAX:

```
SELECT COL1, COL2 . . .
FROM  TABLE
WHERE CONDITION
ORDER BY COL1, COL2... ASC|DESC;
```

Here the where clause is not necessary. In that case:

```
SELECT COL1, COL2. . .
FROM TABLE
ORDER BY COL1,COL2... ASC|DESC
```

# 3.6 Aggregate Functions

## 3.6.1 MIN ():

SYNTAX:

```
SELECT MIN (COL_NAME)
FROM TABLE_NAME;
```

EXAMPLE:

To find the person with the minimum salary:

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

## 3.6.2 MAX ():

SYNTAX:

```
SELECT MAX (COL_NAME)
FROM TABLE_NAME;
```

EXAMPLE:

To find the person with maximum salary:

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

We can always rename the column from MAX (SALARY) to 'desired name'

EXAMPLE:

```
SELECT MAX (SALARY) AS MAXIMUM_SALARY
FROM CITIZEN;
```

For selecting the maximum salary along with the salary holders name is quite a bit tricky!

```
SELECT C_NAME, MAX (SALARY)
FROM CITIZEN;
```

Will not work!

Because the C_NAME is returning all the C_NAME of citizen table and the max (salary) is returning only a single value. Thus oracle can't relate the two things!

What is the solution?

If we could find the max (salary) first and then match that salary with the table then return the name, the problem will be solved!

For this we need nested query!

```
SELECT C_NAME, SALARY
FROM CITIZEN
WHERE SALARY= (SELECT MAX (SALARY) FROM CITIZEN);
```

Here the (SELECT MAX (SALARY) FROM CITIZEN); executes first. It returns the maximum salary. Then the outer query matches this salary with all the salaries. Then the name is found! Tricky, Right?!

### 3.6.3 COUNT ():

COUNT () functions returns the number of rows matching a specific criteria.

EXAMPLE:

Find the number of teachers:

```
SELECT COUNT (C_ID)
FROM CITIZEN
WHERE OCCUPATION='Teacher';
```

To find the number of rows present in a table:

```
SELECT COUNT (*)
FROM CITIZEN;
```

### 3.6.4 AVG ():

Find the average salary of the citizens:

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

### 3.6.5 SUM ():

```
SELECT SUM (SALARY)
FROM CITIZEN;
```

Will return the addition of all the salaries of all citizens.

## 3.7 Aggregating and Grouping

There are circumstances where we would like to apply the aggregate function not only to a single set of tuples, but also to a group of sets of tuples; we specify this wish in SQL using the **group by** clause. The attribute or attributes given in the **group by** clause are used to form groups. Tuples with the same value on all attributes in the **group by** clause are placed in one group.

Suppose you want the average salary based on occupations!

```
SELECT OCCUPATION, AVG (SALARY)
FROM CITIZEN
GROUP BY OCCUPATION;
```

Reference: https://www.w3schools.com/sql/sql_groupby.asp

## 3.8 Having Clause

Sometimes we might work with conditions which apply on a group of tuples but not individual ones. In that case we use the '**Having'** clause.
It only works for aggregate functions.

Group the citizens based on living place and show the number of citizens living in each district where at least 2 people belong to that district.

```
SELECT COUNT (C_HOME) AS NUMBER_OF_CITIZEN, C_HOME
FROM CITIZEN
GROUP BY C_HOME
HAVING COUNT (C_HOME)>2;
```

Here HAVING clause works similar to where clause. But the difference is 'where' clause works on each tuples, but 'having' clause works on a group of tuples.

Reference: https://www.w3schools.com/sql/sql_having.asp

# 3.9 USING ROWNUM

**Syntax:**

```
SELECT column_name(s)
FROM table_name
WHERE ROWNUM <= number;
```

**Example**:

**Ques**: Find out the top 5 salary holders from the citizens table:
**Answer**:

```
SELECT *
FROM CITIZEN
WHERE ROWNUM<=5;
```

Wrong!

This query actually doesn't shows out desired outcome!
What to do?

Let's understand why this is happening.

Can we see the ROWNUMBERS??
Yes.
The query should be:

```
SELECT ROWNUM, C_ID, SALARY
FROM CITIZEN;
```

This is the answer to the query. We can see here the salary is not sorted.

| ROWNUM | C_ID | SALARY |
|---|---|---|
| 1 | 1 | 50000 |
| 2 | 2 | 60000 |
| 3 | 3 | 10000 |
| 4 | 4 | 500 |
| 5 | 5 | 40000 |
| 6 | 6 | 55000 |
| 7 | 7 | 5000 |
| 8 | 8 | 60000 |
| 9 | 9 | 1000 |
| 10 | 10 | 45000 |
| 11 | 11 | 20000 |
| 12 | 12 | 1500 |
| 13 | 13 | 100000 |
| 14 | 14 | 70000 |
| 15 | 15 | 50000 |
| 16 | 16 | 50000 |
| 17 | 17 | 50000 |
| 18 | 18 | 45000 |
| 19 | 19 | 500 |

So we have to sort based on the salary and then find out the TOP five rows.
So the following query should work.

```
SELECT ROWNUM, C_ID, SALARY
FROM CITIZEN
WHERE ROWNUM<=5
ORDER BY SALARY;
```

Wrong again!

| ROWNUM | C_ID | SALARY |
|---|---|---|
| 4 | 4 | 500 |
| 3 | 3 | 10000 |
| 5 | 5 | 40000 |
| 1 | 1 | 50000 |
| 2 | 2 | 60000 |

But it doesn't work!
Why???

Here what's happening is,

First the query is taking the first 5 rows from the table (regardless sorted or not) and then sorting them.

So we first need to sort the table based on the salary and then extract the top 5 rows, isn't it?

So the ultimate answer is:

```
SELECT ROWNUM, C_ID, SALARY
FROM (
        SELECT *
        FROM CITIZEN
        ORDER BY SALARY
        )
WHERE ROWNUM<=5;
```

| ROWNUM | C_ID | SALARY |
|---|---|---|
| 1 | 4 | 500 |
| 2 | 19 | 500 |
| 3 | 9 | 1000 |
| 4 | 12 | 1500 |
| 5 | 7 | 5000 |

**Ques**: Can you find out the top two salary holders?

```
SELECT C_ID, SALARY
FROM (SELECT * FROM CITIZEN ORDER BY SALARY DESC)
WHERE ROWNUM<=2;
```

**Ques**: Find out the second highest salary holder:

If we use the following query it'll not work!

```
SELECT C_ID, SALARY
FROM (
        SELECT *
        FROM CITIZEN
        ORDER BY SALARY DESC
        )
WHERE ROWNUM=2;
```

No Rows Selected!

What to do?

We need to use subquery to solve this problem!

Step 1: Sort the table based on salary.

```
SELECT C_ID, SALARY
FROM CITIZEN
ORDER BY SALARY DESC;
```

| C_ID | SALARY |
|---|---|
| 13 | 100000 |
| 14 | 70000 |
| 8 | 60000 |
| 2 | 60000 |
| 6 | 55000 |
| 16 | 50000 |
| 17 | 50000 |
| 1 | 50000 |
| 15 | 50000 |
| 10 | 45000 |
| 18 | 45000 |
| 5 | 40000 |
| 11 | 20000 |
| 3 | 10000 |
| 7 | 5000 |
| 12 | 1500 |
| 9 | 1000 |
| 4 | 500 |
| 19 | 500 |

Step 2: Showing the associated ROWNUMBERS with the answer.

```
SELECT ROWNUM, C_ID, SALARY
FROM (
        SELECT C_ID, SALARY
        FROM CITIZEN
        ORDER BY SALARY DESC
);
```

| ROWNUM | C_ID | SALARY |
|---|---|---|
| 1 | 13 | 100000 |
| 2 | 14 | 70000 |
| 3 | 8 | 60000 |
| 4 | 2 | 60000 |
| 5 | 6 | 55000 |
| 6 | 16 | 50000 |
| 7 | 17 | 50000 |
| 8 | 1 | 50000 |
| 9 | 15 | 50000 |

| | | |
|---|---|---|
| 10 | 10 | 45000 |
| 11 | 18 | 45000 |
| 12 | 5 | 40000 |
| 13 | 11 | 20000 |
| 14 | 3 | 10000 |
| 15 | 7 | 5000 |
| 16 | 12 | 1500 |
| 17 | 9 | 1000 |
| 18 | 4 | 500 |
| 19 | 19 | 500 |

Step 3:

Finally now we have to extract the 2nd row from this virtual table!

```
SELECT *
FROM (
        SELECT ROWNUM AS RN, C_ID, SALARY
        FROM (
                SELECT C_ID, SALARY
                FROM CITIZEN
                ORDER BY SALARY DESC
                )
        )

WHERE RN=2;
```

Look very carefully!

Here the renaming is necessary. If u write "ROWNUM=2" instead of RN=2, no rows will be returned as the meaning of ROWNUM is changed after every virtual table is generated. As we want to access the ROW Numbers of the previously generated tables, we must change the name!

The use of ROWNUM turns out to be tricky! But it is really handy at these sort of cases.

Reference: https://stackoverflow.com/questions/9240192/selecting-the-second-row-of-a-table-using-rownum?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa

# 3.10 SQL LIKE OPERATOR:

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

% - The percent sign represents zero, one, or multiple characters
_ - The underscore represents a single character

Syntax:

```
SELECT column1, column2, ...
FROM table_name
WHERE column LIKE pattern;
```

USE:

| | |
|---|---|
| WHERE name LIKE 'a%' | Finds any values that starts with 'a' |
| WHERE name LIKE '%a' | Finds any values that ends with 'a' |
| WHERE name LIKE '%gg%' | finds any values that has 'gg' as a substring |
| WHERE name LIKE '_r%' | finds any values that have 'r' at the second position. |
| WHERE name LIKE 'a_%_%; | finds any values that start with 'a' and have at least 3 characters in length. |
| WHERE name like 'a%o' | finds any value that start with 'a' and ends with 'e'. |

Example:

```
SELECT * FROM Customers
WHERE CUSTOMERNAME LIKE '%or%';
```

**Ques**: Find out the C_HOME which start with 'D' or 'C'

```
SELECT *
FROM CITIZEN
WHERE C_HOME LIKE 'C%' OR C_HOME LIKE 'D%';
```

Suppose you want all the city names which doesn't start with 'C'. For that you have to use 'NOT LIKE'.

```
SELECT *
FROM CITIZEN
WHERE C_HOME NOT LIKE 'C%';
```

Reference: https://www.w3schools.com/sql/sql_wildcards.asp

# 3.11 SQL ALIAS:

SQL aliases are used to give a table, or a column in a table, a temporary name.
Aliases are often used to make column names more readable.
An alias only exists for the duration of the query.

**Syntax:**

Column Syntax:

```
SELECT column_name AS alias_name
FROM table_name;
```

Table Syntax:

```
SELECT column_name(s)
FROM table_name AS alias_name;
```

**Example**:

```
SELECT C_NAME AS NAME, C_HOME AS ADDRESS
FROM CITIZEN;
```

Note: Although we have changed the name of C_HOME column as Name, it doesn't have any effect on the main table description. This name will only exist till the duration of the query.

**Ques**: Show the citizen names and a statement of their current occupation with their salary in the format – 'Occupation (salary)';

```
SELECT C_ID, OCCUPATION || '(' || SALARY || ')' AS STATEMENT
FROM CITIZEN;
```

```
   C_ID STATEMENT
------- -----------------
      1 Teacher (50000)
      2 Service (60000)
      3 Retired (10000)
      4 Student (500)
      5 Service (40000)
      6 Doctor (55000)
      7 Musician (5000)
      8 Engineer (60000)
      9 Student (1000)
     10 Teacher (45000)
     11 Farmer (20000)
     12 Student (1500)
     13 Business (100000)
     14 Doctor (70000)
     15 Teacher (50000)
     16 Musician (50000)
     17 Service (50000)
     18 Service (45000)
     19 Student (500)
```

Aliases can be useful when:

- There are more than one table involved in a query
- Functions are used in the query
- Column names are big or not very readable
- Two or more columns are combined together

Reference:

https://www.techonthenet.com/oracle/alias.php
https://www.w3schools.com/sql/sql_alias.asp

# 3.12 SQL UNION:

The UNION operator is used to combine the result-set of two or more SELECT statements.
Each SELECT statement within UNION must have the same number of columns
The columns must also have similar data types
The columns in each SELECT statement must also be in the same order

**Syntax**:

```
SELECT column_name(s) FROM table1
UNION
SELECT column_name(s) FROM table2;
```

The UNION operator selects only distinct values by default. To allow duplicate values, use **UNION ALL:**

```
SELECT column_name(s) FROM table1
UNION ALL
SELECT column_name(s) FROM table2;
```

**Examples**:

Suppose we have the following two tables:

*STUDENT (id, name, city, gap);*
*TEACHER (id, name, city, salary);*

Now suppose we want a list of all the available city in the two tables:

For this we need to find the CITY column from both of the tables and the somehow show them together! In these cases we use UNION.

```
SELECT CITY FROM STUDENT
UNION
SELECT CITY FROM TEACHER
ORDER BY CITY;
```

Here if we want to see the CGPA and SALARY also using this query, it's not possible as the first rule of UNION is that both of the individual queries should return the same column.

(If we needed the duplicate CITY names as well, we should've used 'UNION ALL' instead of 'UNION').

**Example**:

Show the record of the Highest and the lowest salary holders from the CITIZEN table

We can't do this using a single query. We can solve this using ROWNUM

```
SELECT ROWNUM, C_ID, OCCUPATION, SALARY
FROM (
        SELECT ROWNUM AS RN, C_ID, OCCUPATION, SALARY
        FROM (
                SELECT *
                FROM CITIZEN
                ORDER BY SALARY
        )
)
WHERE RN =1 OR RN= (SELECT COUNT (C_ID) FROM CITIZEN);
```

| ROWNUM | C_ID | OCCUPATION | SALARY |
|---|---|---|---|
| 1 | 4 | Student | 500 |
| 2 | 13 | Business | 100000 |

3 NESTED QUERIES!

This task can be solved easily using UNION operation.

```
SELECT *
FROM CITIZEN
WHERE SALARY = (SELECT MAX (SALARY) FROM CITIZEN)
UNION
SELECT *
FROM CITIZEN
WHERE SALARY = (SELECT MIN (SALARY) FROM CITIZEN);
```

| C_ID | C_NAME | C_HOME | AGE | OCCUPATION | GENDER | SALARY |
|---|---|---|---|---|---|---|
| 4 | D | Ctg | 13 | Student | Female | 500 |
| 13 | M | Ctg | 25 | Business | Male | 100000 |
| 19 | S | Ctg | 16 | Student | Male | 500 |

If you look at the results, you'll see the UNION operation is giving better output and it's easy to interpret as well.

In case of ROWNUM, we extracted the top and bottom rows after sorting. But as we see in this case, there are two person with the minimum salary. So both of the results should be shown. So here UNION given us better output.

Reference: https://www.w3schools.com/sql/sql_union.asp

# 3.13 SQL 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.

**Syntax**:

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (value1, value2, ...);
```

Or

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (SELECT STATEMENT);
```

**Example**:

**Ques**: Find the Citizen records who live in 'Dhaka' or 'Ctg' or 'Khulna'.

```
SELECT *
FROM CITIZEN
WHERE C_HOME IN ('Dhaka', 'Ctg', 'Khulna');
```

[NOTE: If you directly copy this query into the SQL command prompt, you might face some error as the interpretation of single quote is different in CMD than in MS WORD. Try to write the quoted string by your own in the CMD and see the difference. ]

**Ques**: Show the citizen records who don't live in the above mentioned cities!

```
SELECT *
FROM CITIZEN
WHERE C_HOME NOT IN ('Dhaka', 'Ctg', 'Khulna');
```

Well,
We can do the previous two queries using OR statement!

**Ques 1:**

```
SELECT *
FROM CITIZEN
WHERE C_HOME LIKE 'Dhaka' OR C_HOME LIKE 'Ctg' OR C_HOME LIKE 'Khulna';
```

Also in the following way:

```
SELECT *
FROM CITIZEN
WHERE C_HOME = 'Dhaka' OR C_HOME = 'Ctg' OR C_HOME = 'Khulna';
```

**Ques 2**:

```
SELECT *
FROM CITIZEN
WHERE C_HOME NOT LIKE 'Dhaka' AND C_HOME NOT LIKE 'Ctg' AND C_HOME NOT
LIKE 'Khulna';
```

'IN 'operator is also very useful if we are working with subqueries.

For example, if you want the ID and Address of the citizens who are 'Teacher' it can be done using simple queries or subqueries.

If we use condition in the where clause:

```
SELECT C_ID, C_HOME
FROM CITIZEN
WHERE OCCUPATION LIKE 'Teacher';
```

If you use subquery:

```
SELECT C_ID, C_HOME
FROM CITIZEN
WHERE C_ID = (SELECT C_ID FROM CITIZEN WHERE OCCUPATION = 'Teacher');
```

But it will not work!!

It is not working because the inner query is returning more than one row!

We can solve the problem using 'IN' clause.

So the proper answer should be:

```
SELECT C_ID, C_HOME
FROM CITIZEN
WHERE C_ID IN (SELECT C_ID FROM CITIZEN WHERE OCCUPATION = 'Teacher');
```

# 3.14 SQL NULL FUNCTIONS:

To understand this properly, let's consider the following scenario:

```
CREATE TABLE PRODUCTS (
        ID NUMBER PRIMARY KEY,
        NAME VARCHAR2 (10),
        INSTOCK NUMBER,
        TOTALORDER NUMBER,
        UNITPRICE NUMBER
);

INSERT INTO PRODUCTS VALUES (1, 'PEN', 10, 2, 5);
INSERT INTO PRODUCTS VALUES (2, 'ERASER', 10, 1, and 5);
INSERT INTO PRODUCTS VALUES (3, 'NOTEBOOK', 10, 6, 30);
INSERT INTO PRODUCTS VALUES (4, `PENCIL', 10, NULL, 5);
```

Here we have no order for ID = 4. Now if we want to execute a query like:

```
SELECT ID, UNITPRICE* (TOTALORDER+INSTOCK)
FROM PRODUCTS;
```

The value for the 4th row will be NULL although it has 10 pencils INSTOCK!

Actually, in this query the addition of INSTOCK+TOTALORDER is working like 10+NULL which is resulting into NULL and the Multiplication is giving NULL as well.

```
  ID UNITPRICE*(TOTALORDER+INSTOCK)
----- ------------------------------
   1                             60
   2                             55
   3                            480
   4
```

To solve this problem we have to somehow convert the NULL value of INSTOCK into 0. We can do this using the 'NVL ()' function is ORACLE (there are different solutions for this problem in the other systems.)

So the correct query should be:

```
SELECT ID, UNITPRICE* (NVL (TOTALORDER, 0) +NVL (INSTOCK, 0))
FROM PRODUCTS;
```

```
  ID UNITPRICE*(NVL (TOTALORDER, 0) +NVL (INSTOCK, 0))
---- ------------------------------------------------
   1                                              60
   2                                              55
   3                                             480
   4                                              50
```

Carefully notice the difference in the 4th column. This function is handy if NULL value is allowed in some columns of numeric type.

[Note: Here the 'UNITPRICE*(NVL (TOTALORDER, 0) +NVL (INSTOCK, 0))' column name in the output column looks UGLY! We can always rename it using ALIAS! ]

```
SELECT ID, UNITPRICE* (NVL (TOTALORDER, 0) +NVL (INSTOCK, 0)) AS TOTAL
FROM PRODUCTS;
```

| ID | TOTAL |
|---|---|
| 1 | 60 |
| 2 | 55 |
| 3 | 480 |
| 4 | 50 |

# 3.15 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.

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

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** - Uniquely identifies a row/record in another table
- **CHECK** - Ensures that all values in a column satisfies a specific condition
- **DEFAULT** - Sets a default value for a column when no value is specified
- **INDEX** - Used to create and retrieve data from the database very quickly

### 3.15.1 SQL 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, or update a record without adding a value to this field.

The following SQL ensures that the "ID", "LASTNAME", and "FIRSTNAME" columns will NOT accept NULL values:

```
CREATE TABLE Persons (
    ID INT NOT NULL,
    LASTNAME VARCHAR (20) NOT NULL,
    FIRSTNAME VARCHAR (255) NOT NULL,
    Age NUMBER
);
```

## 3.15.2 SQL ALTER TABLE Statement:

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

**ALTER TABLE - ADD Column:**

```
ALTER TABLE table_name
ADD column_name datatype;
```

**ALTER TABLE – DROP COLUMN:**

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

**ALTER TABLE – MODIFY COLUMN:**

```
ALTER TABLE table_name
MODIFY column_name datatype;
```

**Examples:**

*PERSONS (ID, LASTNAME, FIRSTNAME, Address, City);*

Now if we want to add a new column 'DateOfBirth' to the table, the statement will be:

```
ALTER TABLE PERSONS
ADD DATEOFBIRTH DATE;
```

Suppose now we want to store the birth year only in the DATEOFBIRTH column instead of the full date. So we have to use ALTER TABLE MODIFY COLUMN statement.

```
ALTER TABLE PERSONS
MODIFY COLUMN DATEOFBIRTH YEAR;
```

## 3.15.3 SQL UNIQUE CONSTRAINT:

The UNIQUE 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.

A PRIMARY KEY constraint automatically has a UNIQUE constraint.

However, you can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table.

**Syntax:**

The following SQL creates a UNIQUE constraint on the "ID" column when the "Persons" table is created:

```
CREATE TABLE Persons (
    ID INT NOT NULL UNIQUE,
    LASTNAME VARCHAR2 (20) NOT NULL,
    FIRSTNAME VARCHAR2 (20),
    Age INT
);
```

To name a UNIQUE constraint, and to define a UNIQUE constraint on multiple columns, use the following SQL syntax:

```
CREATE TABLE Persons (
    ID INT NOT NULL,
    LASTNAME VARCHAR2 (20) NOT NULL,
    FIRSTNAME VARCHAR2 (20),
    Age INT,
    CONSTRAINT UNIQUE_PERSON UNIQUE (ID,LASTNAME)
);
```

Here the pair of ID and LASTNAME should be unique.

**SQL UNIQUE Constraint on ALTER TABLE**

To create a UNIQUE constraint on the "ID" column when the table is already created, use the following SQL:

```
ALTER TABLE Persons
ADD UNIQUE (ID);
```

To name a UNIQUE constraint, and to define a UNIQUE constraint on multiple columns, use the following SQL syntax:

```
ALTER TABLE Persons
ADD CONSTRAINT UNIQUE_PERSON UNIQUE (ID, LASTNAME);
```

**DROP a UNIQUE Constraint:**

```
ALTER TABLE Persons
DROP CONSTRAINT UNIQUE_PERSON;
```

What if you didn't mention any constraint name for UNIQUE in the table statement?

```
ALTER TABLE PERSONS
DROP UNIQUE(ID);
```

Reference: https://stackoverflow.com/questions/5499574/how-to-drop-a-unique-constraint-from-table-column?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa

https://www.w3schools.com/sql/sql_unique.asp

## 3.15.4 SQL CHECK Constraint:

**SQL CHECK on CREATE TABLE:**

The following SQL creates a CHECK constraint on the "Age" column when the "Persons" table is created. The CHECK constraint ensures that you cannot have any person below 18 years:

```
CREATE TABLE Persons (
   ID INT NOT NULL,
   LASTNAME VARCHAR2 (20) NOT NULL,
   FIRSTNAME VARCHAR2 (20),
   AGE INT CHECK (AGE>=18)
);
```

To allow naming of a CHECK constraint, and for defining a CHECK constraint on multiple columns, use the following SQL syntax:

```
CREATE TABLE Persons (
   ID INT NOT NULL UNIQUE,
   LASTNAME VARCHAR2 (20) NOT NULL,
   FIRSTNAME VARCHAR2 (20),
   Age INT,
   CITY VARCHAR2 (20),
   CONSTRAINT CHECK_PERSON CHECK (AGE>=18 AND CITY='Sandnes')
);
```

**SQL CHECK on ALTER TABLE:**

To create a CHECK constraint on the "Age" column when the table is already created, use the following SQL:

```
ALTER TABLE Persons
ADD CHECK (Age>=18);

ALTER TABLE Persons
ADD CONSTRAINT CHECK_PERSON CHECK (Age>=18 AND City='Sandnes');
```

**DROP a CHECK Constraint:**

```
ALTER TABLE Persons
DROP CONSTRAINT CHECK_PERSON;
```

## 3.15.5 SQL DEFAULT constraint:

The DEFAULT constraint is used to provide a default value for a column.
The default value will be added to all new records IF no other value is specified.

The following SQL sets a DEFAULT value for the "City" column when the "Persons" table is created:

```
CREATE TABLE Persons (
   ID INT NOT NULL UNIQUE,
   LASTNAME VARCHAR2 (20) NOT NULL,
   FIRSTNAME VARCHAR2 (20),
   AGE INT,
   City VARCHAR2 (20) DEFAULT 'Sandnes'
);
```

The DEFAULT constraint can also be used to insert system values, by using functions like:

```
CREATE TABLE Orders (
   ID INT NOT NULL,
   ORDERNUMBER INT NOT NULL,
   ORDERDATE DATE DEFAULT SYSDATE,
   TOTALORDES NUMBER DEFAULT 0
);
```

The following line will show error.

```
INSERT INTO ORDERS VALUES (1, 12);

          *
ERROR at line 1:
ORA-00947: not enough values
```

To insert into a table with default value:

```
INSERT INTO ORDERS (ID, ORDERNUMBER) VALUES (1, 12);

> SELECT * FROM ORDERS;

    ID ORDERNUMBER ORDERDATR TOTALORDERS
------- ----------- --------- -----------
     1         12 02-MAY-18           0
```

Notice how the ORDERDATE and TOTALORDERS have come automatically!

Another way to insert into the table is:

```
INSERT INTO ORDERS VALUES (2, 10, DEFAULT, DEFAULT);
```

Note:

DEFAULT constraint automatically puts the default value to a particular column if the no value is given (NULL). Now what if you want to put the NULL value explicitly? For this case, you have to write NULL instead of DEFAULT in the previous query.

```
INSERT INTO ORDERS VALUES (6, 10, NULL, NULL);
```

```
> SELECT * FROM ORDERS;

     ID ORDERNUMBER ORDERDATR TOTALORDERS
------- ----------- --------- -----------
      1          12 02-MAY-18           0
      2          10 02-MAY-18           0
      6          10
```

**SQL DEFAULT on ALTER TABLE:**

```
ALTER TABLE Persons
MODIFY City DEFAULT 'Sandnes';
```

**DROP a DEFAULT Constraint:**

```
ALTER TABLE Persons
ALTER COLUMN City DROP DEFAULT;
```

Reference: http://www.zentut.com/sql-tutorial/sql-default-constraint/
https://www.w3schools.com/sql/sql_default.asp

## 3.15.6 SQL AUTO INCREMENT field:

Auto-increment allows a unique number to be generated automatically when a new record is inserted into a table.
Often this is the primary key field that we would like to be created automatically every time a new record is inserted.

**Syntax:**

```
CREATE SEQUENCE SEQ_PERSON
MINVALUE 1
START WITH 1
INCREMENT BY 1
CACHE 10;
```

The code above creates a sequence object called SEQ_PERSON that starts with 1 and will increment by 1. It will also cache up to 10 values for performance. The cache option specifies how many sequence values will be stored in memory for faster access.

To insert a new record into the "Persons" table, we will have to use the NEXTVAL function (this function retrieves the next value from SEQ_PERSON sequence):

```
CREATE TABLE Persons (
   ID NUMBER PRIMARY KEY,
   LASTNAME varchar (20) NOT NULL,
   FIRSTNAME varchar (20)
);

INSERT INTO Persons (ID, FIRSTNAME, LASTNAME)
VALUES (SEQ_PERSON.NEXTVAL,'Lars','Monsen');
```

# 3.16 SQL VIEW

In SQL, a view is a virtual table based on the result-set of an SQL statement.
A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.
You can add SQL functions, WHERE, and JOIN statements to a view and present the data as if the data were coming from one single table.

**Syntax:**

```
CREATE OR REPLACE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

A view always shows up-to-date data! The database engine recreates the data, using the view's SQL statement, every time a user queries a view.

Suppose we want to create a VIEW to get the sorted list of Teachers based on salary from the CITIZEN table.

```
CREATE OR REPLACE VIEW SORTED_CITIZEN_TEACHER AS
SELECT *
FROM CITIZEN
WHERE OCCUPATION='Teacher'
ORDER BY SALARY;
```

```
> SELECT * FROM SORTED_CITIZEN_TEACHER;

  C_ID C_NAME     C_HOME          AGE OCCUPATION      GENDER     SALARY
------- ---------- ---------- ---------- --------------- ------ ----------
    10 J         Comilla          32 Teacher        Male       45000
    15 O          Gazipur          53 Teacher         Male       50000
     1 A         Dhaka           25 Teacher        Male       50000
```

We can access the VIEW like a table.

```
SELECT *
FROM SORTED_CITIZEN_TEACHER;
```

```
  C_ID C_NAME     C_HOME          AGE OCCUPATION      GENDER     SALARY
------- ---------- ---------- ---------- --------------- ------ ----------
    10 J         Comilla          32 Teacher        Male       45000
    15 O          Gazipur          53 Teacher         Male       50000
     1 A         Dhaka           25 Teacher        Male       50000
```

```
SELECT C_NAME AS NAME, SALARY
FROM SORTED_CITIZEN_TEACHER
ORDER BY SALARY DESC;
```

```
NAME          SALARY
---------- ----------
A              50000
O              50000
J              45000
```

Notice how the salary is sorted into descending order here.

Creating a view using more than one tables.

```
CREATE TABLE PATIENT (
ID NUMBER PRIMARY KEY,
NAME VARCHAR2 (10),
AGE NUMBER
);

CREATE TABLE DISEASE (
DID NUMBER PRIMARY KEY,
D_NAME VARCHAR2 (10)
);

CREATE TABLE MEDICAL_DETAIL (
PID NUMBER,
DID NUMBER,
FOREIGN KEY (PID) REFERENCES PATIENT (ID),
FOREIGN KEY (DID) REFERENCES DISEASE (DID)
);

INSERT INTO PATIENT VALUES (1,'A', 23);
INSERT INTO PATIENT VALUES (2,'B', 23);
INSERT INTO PATIENT VALUES (3,'C', 23);

INSERT INTO DISEASE VALUES (1,'X');
INSERT INTO DISEASE VALUES (2,'Y');
INSERT INTO DISEASE VALUES (3,'Z');

INSERT INTO MEDICAL_DETAIL VALUES (1, 1);
INSERT INTO MEDICAL_DETAIL VALUES (2, 1);
INSERT INTO MEDICAL_DETAIL VALUES (3, 1);
INSERT INTO MEDICAL_DETAIL VALUES (1, 2);
INSERT INTO MEDICAL_DETAIL VALUES (1, 3);
INSERT INTO MEDICAL_DETAIL VALUES (2, 2);
INSERT INTO MEDICAL_DETAIL VALUES (3, 1);
```

Now suppose we want the medical detail of person 'A' and keep it in a view!

```
CREATE OR REPLACE VIEW MEDICAL_1 AS
SELECT P.ID, P.NAME, P.AGE, D.D_NAME
FROM PATIENT P, DISEASE D, MEDICAL_DETAIL MD
WHERE P.ID=MD.PID AND D.DID=MD.DID AND P.ID=1;
```

```
Select * from medical_1;

        ID NAME            AGE D_NAME
---------- ---------- ---------- ----------
         1 A                  23 X
         1 A                  23 Y
         1 A                  23 Z
```

**SQL DROP VIEW:**

```
DROP VIEW view_name;
```

**UPDATING A VIEW:**

First of all, VIEW is never stored in the hard drive. The view is created only when it is called by any query. So in first chance, you may think as VIEW might not be updatable. But VIEWs are updatable under certain circumstances.

**Example_1:**

```
CREATE TABLE PLAYER (
ID NUMBER PRIMARY KEY,
NAME VARCHAR2 (20),
SCORED_GOAL NUMBER
);

INSERT INTO PLAYER VALUES (1,'A', 25);
INSERT INTO PLAYER VALUES (2,'B', 35);
INSERT INTO PLAYER VALUES (3,'C', 45);
```

Now let's create a VIEW with the name and scored goals of different players.

```
CREATE VIEW PLAYER_GOAL AS
SELECT NAME, SCORED_GOAL
FROM PLAYER;
```

```
SQL> SELECT * FROM PLAYER_GOAL;

NAME                 SCORED_GOAL
-------------------- -----------
A                             25
B                             35
C                             45
```

Let's insert some values to the view.

```
INSERT INTO PLAYER_GOAL VALUES ('D', 33);
```

The system is not letting us inserting the values!

```
SQL>          INSERT INTO PLAYER_GOAL VALUES ('D', 33);
              INSERT INTO PLAYER_GOAL VALUES ('D', 33)
*
ERROR at line 1:
ORA-01400: cannot insert NULL into ("SYSTEM"."PLAYER"."ID")
```

Why?

To understand this clearly lets create another view with the player id and goal.

```
CREATE VIEW PLAYER_GOAL_ID AS
SELECT ID, SCORED_GOAL
FROM PLAYER;
```

```
SQL> select * from player_goal_id;

        ID SCORED_GOAL
---------- -----------
         1          25
         2          35
         3          45
```

Now let's try to insert into this VIEW.

```
INSERT INTO PLAYER_GOAL_ID VALUES (4, 45);
```

```
SQL>    INSERT INTO PLAYER_GOAL_ID VALUES (4, 45);

1 row created.
```

NO ERRORS!

ANOTHER THING, THE MAIN TABLE IS CHANGED ALSO!

```
SQL> SELECT * FROM PLAYER;

        ID NAME                 SCORED_GOAL
---------- -------------------- -----------
         1 A                             25
         2 B                             35
         3 C                             45
         4                               45
```

Actually what happened is we tried to insert data into the previous VIEW without ID, but this didn't take the value, because wherever some value is inserted to the VIEW, it actually tries to put the values into the physical table (as VIEW is never stored). ID is Primary KEY that means, it can't be null. So when we created the VIEW without ID, it didn't accept the value.

So an important observation is, INSERT INTO VIEW IS ONLY POSSIBLE IF IT CONTAINTS ALL THE NOT NULL COLUMNS OF THE TABLE.

**Examples_2:**

Let's think about our MEDICAL_1 VIEW.

```
Select * from medical_1;

  ID NAME            AGE D_NAME
----- ---------- ---------- --------
   1 A              23 X
   1 A              23 Y
   1 A              23 Z
```

```
SQL> INSERT INTO MEDICAL_1 VALUES (1,'A', 25,'X');
INSERT INTO MEDICAL_1 VALUES (1,'A', 25,'X')
*
ERROR at line 1:
ORA-01779: cannot modify a column which maps to a non-key-preserved table
```

WHY IS THIS ERROR COMING?
What do you think?
Ok, some may think like this is happening because DISEASE_ID is not added to the view. So let's change the definition of the VIEW!

```
CREATE OR REPLACE VIEW MEDICAL_1 AS
SELECT P.ID, P.NAME, P.AGE, D.DID, D.D_NAME
FROM PATIENT P, DISEASE D, MEDICAL_DETAIL MD
WHERE P.ID=MD.PID AND D.DID=MD.DID AND P.ID=1;
```

```
Select * from medical_1;

  ID NAME            AGE       DID D_NAME
------ ---------- ---------- ---------- ---------
   1 A              23         1 X
   1 A              23         2 Y
   1 A              23         3 Z
```

```
SQL> INSERT INTO MEDICAL_1 VALUES (4,'A', 25, 4,'X');
        INSERT INTO MEDICAL_1 VALUES (4,'A', 25, 4,'X')
        *
        ERROR at line 1:
        ORA-01779: cannot modify a column which maps to a non-key-preserved table
```

VIEW CAN'T BE INSERTED IF IN CONTAINS INFORMATIONS MORE THAN ONE TABLE.

Besides, INSERT INTO some VIEW isn't possible isn't possible if it contains some aggregate functions as well.

If the table contains any CHECK constraint, the newly inserted values must fulfill those criteria also.

References: https://docs.oracle.com/cd/B28359_01/server.111/b28310/views001.htm#ADMIN11775
https://www.w3schools.com/sql/sql_view.asp

# 4 Referenced Tables

## 4.1 Citizen

```
CREATE TABLE CITIZEN (
        C_ID NUMBER (3),
        C_NAME VARCHAR2 (10),
        C_HOME VARCHAR2 (10),
        AGE NUMBER (2),
        OCCUPATION VARCHAR2 (15),
        GENDER VARCHAR2 (6),
        SALARY NUMBER,
        CONSTRAINTS PK_CITIZEN PRIMARY KEY (C_ID)
);

INSERT INTO CITIZEN VALUES (1, 'A', 'Dhaka', 25, 'Teacher', 'Male', 50000);
INSERT INTO CITIZEN VALUES (2, 'B', 'Dhaka', 56, 'Service', 'Male', 60000);
INSERT INTO CITIZEN VALUES (3, 'C', 'Ctg', 71, 'Retired', 'Male', 10000);
INSERT INTO CITIZEN VALUES (4, 'D', 'Ctg', 13, 'Student', 'Female', 500);
INSERT INTO CITIZEN VALUES (5, 'E', 'Dhaka', 45, 'Service', 'Male', 40000);
INSERT INTO CITIZEN VALUES (6, 'F', 'Gazipur', 54, 'Doctor', 'Female', 55000);
INSERT INTO CITIZEN VALUES (7, 'G', 'Gazipur', 65, 'Musician', 'Female', 5000);
INSERT INTO CITIZEN VALUES (8, 'H', 'Dhaka', 56, 'Engineer', 'Male', 60000);
INSERT INTO CITIZEN VALUES (9, 'I', 'Ctg', 23, 'Student', 'Male', 1000);
INSERT INTO CITIZEN VALUES (10, 'J', 'Comilla', 32, 'Teacher', 'Male', 45000);
INSERT INTO CITIZEN VALUES (11, 'K', 'Comilla', 51, 'Farmer', 'Male', 20000);
INSERT INTO CITIZEN VALUES (12, 'L', 'Khulna', 15, 'Student', 'Female', 1500);
INSERT INTO CITIZEN VALUES (13, 'M', 'Ctg', 25, 'Business', 'Male', 100000);
INSERT INTO CITIZEN VALUES (14, 'N', 'Comilla', 52, 'Doctor', 'Male', 70000);
INSERT INTO CITIZEN VALUES (15, 'O', 'Gazipur', 53, 'Teacher', 'Male', 50000);
INSERT INTO CITIZEN VALUES (16, 'P', 'Dhaka', 35, 'Musician', 'Female', 50000);
INSERT INTO CITIZEN VALUES (17, 'Q', 'Khulna', 43, 'Service', 'Male', 50000);
INSERT INTO CITIZEN VALUES (18, 'R', 'Khulna', 34, 'Service', 'Female', 45000);
INSERT INTO CITIZEN VALUES (19, 'S', 'Ctg', 16, 'Student', 'Male', 500);
```

# 4.2 STD

```
CREATE TABLE STD (
        STD_ID NUMBER (3),
        STD_NAME VARCHAR (2),
        STD_DEPT NUMBER (3),
        STD_CG NUMBER (3, 2),
        CONSTRAINTS PK_STD PRIMARY KEY (STD_ID),
        CONSTRAINTS FK_STD_DEPT FOREIGN KEY (STD_DEPT) REFERENCES DEPT (DEPT_ID)
);

INSERT INTO STD VALUES (41,'A', 101, 3.5);
INSERT INTO STD VALUES (42,'B', 102, 3.6);
INSERT INTO STD VALUES (43,'C', 103, 3.7);
INSERT INTO STD VALUES (44,'D', 101, 3.8);
```

# 4.3 Student

```
CREATE TABLE STUDENT (
        S_ID NUMBER (2),
        S_NAME VARCHAR (3),
        S_HOME VARCHAR (10),
        S_DEPT NUMBER (3),
        S_SUP_ID NUMBER (4),
        S_CG NUMBER (3, 2),
        CONSTRAINTS PK_STUDENT PRIMARY KEY (S_ID),
        CONSTRAINTS FK_STUDENT_DEPT FOREIGN KEY (S_DEPT) REFERENCES DEPT
        (DEPT_ID),
        CONSTRAINTS FK_STUDENT_SUPERVISOR FOREIGN KEY (S_SUP_ID) REFERENCES
        SUPERVISOR (SUP_ID)
);


INSERT INTO STUDENT VALUES (1,'SA','DHAKA', 101, 1001, 3.5);
INSERT INTO STUDENT VALUES (2,'SB','CTG', 102, 1002, 3.6);
INSERT INTO STUDENT VALUES (3,'SC','DHAKA', 103, 1003, 3.7);
INSERT INTO STUDENT VALUES (4,'SD','COMILLA', 104, 1004, 3.8);
INSERT INTO STUDENT VALUES (5,'SE','SYLHET', 105, 1005, 3.5);
INSERT INTO STUDENT VALUES (6,'SF','DHAKA', 101, 1006, 3.9);
INSERT INTO STUDENT VALUES (7,'SG','RAJSHAHI', 101, 1001, 3.6);
```

# 4.4 DEPT

```
CREATE TABLE DEPT (
        DEPT_ID NUMBER (3),
        DEPT_NAME VARCHAR (3),
        DEPT_BUILDING VARCHAR (3),
        DEPT_ESTD NUMBER (4),
        CONSTRAINTS PK_DEPT PRIMARY KEY (DEPT_ID)
);


INSERT INTO DEPT VALUES (101,'CSE','NEW', 1998);
INSERT INTO DEPT VALUES (102,'SWE','NEW', 2018);
INSERT INTO DEPT VALUES (103,'MCE','OLD', 1998);
INSERT INTO DEPT VALUES (104,'EEE','OLD', 1998);
INSERT INTO DEPT VALUES (105,'CEE','OLD', 2009);
INSERT INTO DEPT VALUES (106,'BTM','OLD', 2018);
```

## 4.5 Supervisor

```
CREATE TABLE SUPERVISOR (
        SUP_ID NUMBER (4),
        SUP_STAFF_ID NUMBER (5),
        SUP_DEPT NUMBER (3),
        SUP_DPET_ID NUMBER (3),
        CONSTRAINTS PK_SUP PRIMARY KEY (SUP_ID),
        CONSTRAINTS FK_SUP_DEPT FOREIGN KEY (SUP_DEPT) REFERENCES DEPT (DEPT_ID),
        CONSTRAINTS FK_SUP_STAFF FOREIGN KEY (SUP_STAFF_ID) REFERENCE STAFF
        (STAFF_ID)
);
```

# 4.6 Manufacturers

```
CREATE TABLE Manufacturers (
        Code INTEGER,
        Name VARCHAR (255) NOT NULL,
        PRIMARY KEY (Code)
);


INSERT INTO Manufacturers (Codename) VALUES (1,'Sony');
INSERT INTO Manufacturers (Codename) VALUES (2,'Creative Labs');
INSERT INTO Manufacturers (Codename) VALUES (3,'Hewlett-Packard');
INSERT INTO Manufacturers (Codename) VALUES (4,'Iomega');
INSERT INTO Manufacturers (Codename) VALUES (5,'Fujitsu');
INSERT INTO Manufacturers (Codename) VALUES (6,'Winchester');
```

# 4.7 Products

```
CREATE TABLE Products (
        Code INTEGER,
        Name VARCHAR (255) NOT NULL,
        Price DECIMAL NOT NULL,
        Manufacturer INTEGER NOT NULL,
        PRIMARY KEY (Code),
        FOREIGN KEY (Manufacturer) REFERENCES Manufacturers (Code)
);


INSERT INTO Products (Code, Name, Price, and Manufacturer) VALUES (1,'Hard drive', 240,
5);
INSERT INTO Products (Code, Name, Price, and Manufacturer) VALUES (2,'Memory', 120, 6);
INSERT INTO Products(Code,Name,Price,Manufacturer) VALUES(3,'ZIP drive',150,4);
INSERT INTO Products (Code, Name, Price, and Manufacturer) VALUES (4,'Floppy disk', 5, 6);
INSERT INTO Products (Code, Name, Price, and Manufacturer) VALUES (5,'Monitor', 240, 1);
INSERT INTO Products (Code, Name, Price, Manufacturer) VALUES (6,'DVD drive', 180, 2);
INSERT INTO Products (Code, Name, Price, Manufacturer) VALUES (7,'CD drive', 90, 2);
INSERT INTO Products (Code, Name, Price, and Manufacturer) VALUES (8,'Printer', 270, 3);
INSERT INTO Products (Code, Name, Price, and Manufacturer) VALUES (9,'Toner cartridge',
66, 3);
INSERT INTO Products (Code, Name, Price, and Manufacturer) VALUES (10,'DVD burner', 180,
2);
INSERT INTO Products (Code, Name, Price, and Manufacturer) VALUES (11,'Card Reader',
180, 2);
```