# SQL

# MOST IMPORTANT CONCEPTS

# PLACEMENT PREPARATION

## [EXCLUSIVE NOTES]

## [SAVE AND SHARE]

Curated By- HIMANSHU KUMAR(LINKEDIN)

## TOPICS COVERED-

## PART-4 :-

- ALL and ANY
- BETWEEN & IN Operator
- Arithmetic Operators
- DDL, DML, TCL and DCL
- Creating Roles
- Difference between functions and stored procedures in PL/SQL
- Trigger | Book Management Database

HIMANSHU KUMAR(LINKEDIN)

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

# ALL and ANY-

**ALL & ANY** are logical operators in SQL. They return boolean value as a result.

## ALL

ALL operator is used to select all tuples of SELECT STATEMENT. It is also used to compare a value to every value in another value set or result from a subquery.

- The ALL operator returns TRUE iff all of the subqueries values meet the condition. The ALL must be preceded by comparison operators and evaluates true if all of the subqueries values meet the condition.
- ALL is used with SELECT, WHERE, HAVING statement.

### ALL with SELECT Statement:

```
Syntax:
SELECT ALL field_name
FROM table_name
WHERE condition(s);
```

### ALL with WHERE or HAVING Statement:

```
Syntax:
SELECT column_name(s)
FROM table_name
WHERE column_name comparison_operator AL
(SELECT column_name
FROM table_name
WHERE condition(s));
```

**Example:**

Consider the following Products Table and OrderDetails Table,

## Products Table

| ProductID | ProductName | SupplierID | CategoryID | Price |
|---|---|---|---|---|
| 1 | Chais | 1 | 1 | 18 |
| 2 | Chang | 1 | 1 | 19 |
| 3 | Aniseed Syrup | 1 | 2 | 10 |
| 4 | Chef Anton's Cajun Seasoning | 2 | 2 | 22 |
| 5 | Chef Anton's Gumbo Mix | 2 | 2 | 21 |
| 6 | Boysenberry Spread | 3 | 2 | 25 |
| 7 | Organic Dried Pears | 3 | 7 | 30 |
| 8 | Northwoods Cranberry Sauce | 3 | 2 | 40 |
| 9 | Mishi Kobe Niku | 4 | 6 | 97 |

## OrderDetails Table

| OrderDetailsID | OrderID | ProductID | Quantity |
|---|---|---|---|
| 1 | 10248 | 1 | 12 |
| 2 | 10248 | 2 | 10 |
| 3 | 10248 | 3 | 15 |
| 4 | 10249 | 1 | 8 |
| 5 | 10249 | 4 | 4 |
| 6 | 10249 | 5 | 6 |
| 7 | 10250 | 3 | 5 |
| 8 | 10250 | 4 | 18 |
| 9 | 10251 | 5 | 2 |
| 10 | 10251 | 6 | 8 |
| 11 | 10252 | 7 | 9 |
| 12 | 10252 | 8 | 9 |
| 13 | 10250 | 9 | 20 |
| 14 | 10249 | 9 | 4 |

## Queries

**Find the name of the all the product.**

```
SELECT ALL ProductName
FROM Products
WHERE TRUE;
```

**Output:**

| ProductName |
| --- |
| Chais |
| Chang |
| Aniseed Syrup |
| Chef Anton's Cajun Seasoning |
| Chef Anton's Gumbo Mix |
| Boysenberry Spread |
| Organic Dried Pears |
| Northwoods Cranberry Sauce |
| Mishi Kobe Niku |

**Find the name of the product if all the records in the OrderDetails has Quantity either equal to 6 or 2.**

```
SELECT ProductName
FROM Products
WHERE ProductID = ALL (SELECT ProductId
                       FROM OrderDetails
                       WHERE Quantity = 6 OR Quanti
ty = 2);
```

**Output:**

| ProductName |
| --- |
| Chef Anton's Gumbo Mix |

**Find the OrderID whose maximum Quantity among all product of that OrderID is greater than average quantity of all OrderID.**

```
SELECT OrderID
FROM OrderDetails
GROUP BY OrderID
HAVING max(Quantity) > ALL (SELECT avg(Quantity)
                            FROM OrderDetails
                            GROUP BY OrderID);
```

**Output:**

| OrderID |
|---|
| 10248 |
| 10250 |

**ANY**

ANY compares a value to each value in a list or results from a query and evaluates to true if the result of an inner query contains at least one row.

- ANY return true if any of the subqueries values meet the condition.
- ANY must be preceded by comparison operators.
  **Syntax:**

```
SELECT column_name(s)
FROM table_name
WHERE column_name comparison_operator ANY
(SELECT column_name
FROM table_name
WHERE condition(s));
```

## Queries

Find the Distinct CategoryID of the products which have any record in OrderDetails Table.

```
SELECT DISTINCT CategoryID
FROM Products
WHERE ProductID = ANY (SELECT ProductID
                       FROM OrderDetails);
```

**Output:**

| CategoryID |
|---|
| 1 |
| 2 |
| 7 |
| 6 |

Finds any records in the OrderDetails table that Quantity = 9.

```
SELECT ProductName
FROM Products
WHERE ProductID = ANY (SELECT ProductID
                       FROM OrderDetails
                       WHERE Quantity = 9);
```

| ProductName |
|---|
| Organic Dried Pears |
| Northwoods Cranberry Sauce |

# BETWEEN & IN Operator-

## BETWEEN

The SQL BETWEEN condition allows you to easily test if an expression is within a range of values (inclusive). The values can be text, date, or numbers. It can be used in a SELECT, INSERT, UPDATE, or DELETE statement. The SQL BETWEEN Condition will return the records where expression is within the range of value1 and value2.

**Syntax:**

```
SELECT column_name(s)
FROM table_name
WHERE column_name BETWEEN value1 AND value2;
```

**Examples:**

Consider the following Employee Table,

| Fname | Lname | SSN | Salary | DOB |
|---|---|---|---|---|
| John | Smith | 123456789 | 30000 | 1988-05-02 |
| Franklin | Wong | 333445555 | 40000 | 1986-01-02 |
| Joyce | English | 453453453 | 80000 | 1977-12-08 |
| Ramesh | Narayan | 666884444 | 38000 | 1987-03-05 |
| James | Borg | 888665555 | 55000 | 1982-10-10 |
| Jennifer | Wallace | 987654321 | 43000 | 1985-08-07 |
| Ahmad | Jabbar | 987987987 | 25000 | 1990-06-28 |
| Alicia | Zeala | 999887777 | 25000 | 1980-09-14 |

## Queries

- **Using BETWEEN with Numeric Values:**
  List all the Employee Fname, Lname who is having salary between 30000 and 45000.

```
SELECT Fname, Lname
FROM Employee
WHERE Salary
BETWEEN 30000 AND 45000;
```

**Output:**

| Fname | Lname |
|---|---|
| John | Smith |
| Franklin | Wong |
| Ramesh | Narayan |
| Jennifer | Wallace |

- **Using BETWEEN with Date Values:**
  Find all the Employee having Date of Birth Between 01-01-1985 and 12-12-1990.

```
SELECT Fname, Lname
FROM Employee
where DOB
BETWEEN '1985-01-01' AND '1990-12-30';
```

Output:

| Fname | Lname |
|---|---|
| John | Smith |
| Franklin | Wong |
| Ramesh | Narayan |
| Jennifer | Wallace |
| Ahmad | Jabbar |

- **Using NOT operator with BETWEEN**
  Find all the Employee name whose salary is not in the range of 30000 and 45000.

```
SELECT Fname, Lname
FROM Employee
WHERE Salary
NOT BETWEEN 30000 AND 45000;
```

**Output:**

| Fname | Lname |
|---|---|
| Joyce | English |
| James | Borg |
| Ahmad | Jabbar |
| Alicia | Zeala |

## IN

IN operator allows you to easily test if the expression matches any value in the list of values. It is used to remove the need of multiple OR condition in SELECT, INSERT, UPDATE or DELETE. You can also use NOT IN to exclude the rows in your list. We should note that any kind of duplicate entry will be retained.

**Syntax:**

```
SELECT column_name(s)

FROM table_name

WHERE column_name IN (list_of_values);
```

## Queries

Find the Fname, Lname of the Employees who have Salary equal to 30000, 40000 or 25000.

```
SELECT Fname, Lname
FROM Employe
WHERE Salary IN (30000, 40000, 25000);
```

**Output:**

| Fname | Lname |
|---|---|
| John | Smith |
| Franklin | Wong |
| Ahmad | Jabbar |
| Alicia | Zeala |

Find the Fname, Lname of all the Employee who have Salary not equal to 25000 or 30000.

```
SELECT Fname, Lname
FROM Employee
WHERE Salary NOT IN (25000, 30000);
```

**Output:**

| Fname | Lname |
|---|---|
| Franklin | Wong |
| Joyce | English |
| Ramesh | Narayan |
| James | Borg |
| Jennifer | Wallace |

## Arithmetic Operators-

We can use various Arithmetic Operators on the data stored in the tables.

**Arithmetic Operators are:**

```
+           [Addition]
-           [Subtraction]
/           [Division]
*           [Multiplication]
%           [Modulus]
```

## Addition **(+) :**

It is used to perform **addition operation** on the data items, items include either single column or multiple columns.

**Implementation**:

```
SELECT employee_id, employee_name, salary, salary +
100
   AS "salary + 100" FROM addition;
```

**Output:**

| employee_id | employee_name | salary | salary+100 |
|---|---|---|---|
| 1 | alex | 25000 | 25100 |
| 2 | rr | 55000 | 55100 |
| 3 | jpm | 52000 | 52100 |
| 4 | ggshmr | 12312 | 12412 |

Here we have done addition of 100 to each Employee's salary i.e, addition operation on single column.

Let's perform **addition of 2 columns**:

```
SELECT employee_id, employee_name, salary, salary +
employee_id
   AS "salary + employee_id" FROM addition;
```

Output:

| employee_id | employee_name | salary | salary+employee_id |
|---|---|---|---|
| 1 | alex | 25000 | 25001 |
| 2 | rr | 55000 | 55002 |
| 3 | jpm | 52000 | 52003 |
| 4 | ggshmr | 12312 | 12316 |

Here we have done addition of 2 columns with each other i.e, each employee's employee_id is added with its salary.

**Subtraction** (-) :

It is use to perform **subtraction operation** on the data items, items include either single column or multiple columns.

**Implementation**:

```
SELECT employee_id, employee_name, salary, salary -
100
    AS "salary - 100" FROM subtraction;
```

**Output:**

| employee_id | employee_name | salary | salary-100 |
|---|---|---|---|
| 12 | Finch | 15000 | 14900 |
| 22 | Peter | 25000 | 24900 |
| 32 | Warner | 5600 | 5500 |
| 42 | Watson | 90000 | 89900 |

Here we have done subtraction of 100 to each Employee's salary i.e, subtraction operation on single column.
Let's perform **subtraction of 2 columns**:

```
SELECT employee_id, employee_name, salary, salary -
employee_id
    AS "salary - employee_id" FROM subtraction;
```

**Output:**

| employee_id | employee_name | salary | salary - employee_id |
|---|---|---|---|
| 12 | Finch | 15000 | 14988 |
| 22 | Peter | 25000 | 24978 |
| 32 | Warner | 5600 | 5568 |

| | | | |
|---|---|---|---|
| 42 | Watson | 90000 | 89958 |

Here we have done subtraction of 2 columns with each other i.e, each employee's employee_id is subtracted from its salary.

**Division** (/) : For **Division** refer this link-  Division in SQL

**Multiplication** (*) :

It is use to perform **multiplication** of data items.

**Implementation**:

```
SELECT employee_id, employee_name, salary, salary *
100
    AS "salary * 100" FROM addition;
```

**Output:**

| employee_id | employee_name | salary | salary * 100 |
|---|---|---|---|
| 1 | Finch | 25000 | 2500000 |
| 2 | Peter | 55000 | 5500000 |
| 3 | Warner | 52000 | 5200000 |
| 4 | Watson | 12312 | 1231200 |

Here we have done multiplication of 100 to each Employee's

salary i.e, multiplication operation on single column.

Let's perform **multiplication of 2 columns**:

```
SELECT employee_id, employee_name, salary, salary *
employee_id
     AS "salary * employee_id" FROM addition;
```

**Output:**

| employee_id | employee_name | salary | salary * employee_id |
|---|---|---|---|
| 1 | Finch | 25000 | 25000 |
| 2 | Peter | 55000 | 110000 |
| 3 | Warner | 52000 | 156000 |
| 4 | Watson | 12312 | 49248 |

Here we have done multiplication of 2 columns with each other i.e, each employee's employee_id is multiplied with its salary.

## Modulus ( % ) :

It is use to get **remainder** when one data is divided by another.

**Implementation**:

```
SELECT employee_id, employee_name, salary, salary %
25000
    AS "salary % 25000" FROM addition;
```

**Output:**

| employee_id | employee_name | salary | salary % 25000 |
|---|---|---|---|
| 1 | Finch | 25000 | 0 |
| 2 | Peter | 55000 | 5000 |
| 3 | Warner | 52000 | 2000 |
| 4 | Watson | 12312 | 12312 |

Here we have done modulus of 100 to each Employee's salary i.e, modulus operation on single column.

Let's perform **modulus operation between 2 columns**:

```
SELECT employee_id, employee_name, salary, salary %
employee_id
    AS "salary % employee_id" FROM addition;
```

**Output:**

| employee_id | employee_name | salary | salary % employee_id |
|---|---|---|---|

| | | | |
|---|---|---|---|
| 1 | Finch | 25000 | 0 |
| 2 | Peter | 55000 | 0 |
| 3 | Warner | 52000 | 1 |
| 4 | Watson | 12312 | 0 |

Here we have done modulus of 2 columns with each other i.e, each employee's salary is divided with its id and corresponding remainder is shown.
Basically, **modulus** is use to check whether a number is **Even** or **Odd**. Suppose a given number if divided by 2 and gives 1 as remainder, then it is an *odd number* or if on dividing by 2 and gives 0 as remainder, then it is an *even number*.

**Concept of NULL :**

If we perform any arithmetic operation on **NULL**, then answer is *always* null.
**Implementation**:

```
SELECT employee_id, employee_name, salary, type, ty
pe + 100
    AS "type+100" FROM addition;
```

**Output:**

| employee_id | employee_name | salary | type | type + 100 |
|---|---|---|---|---|

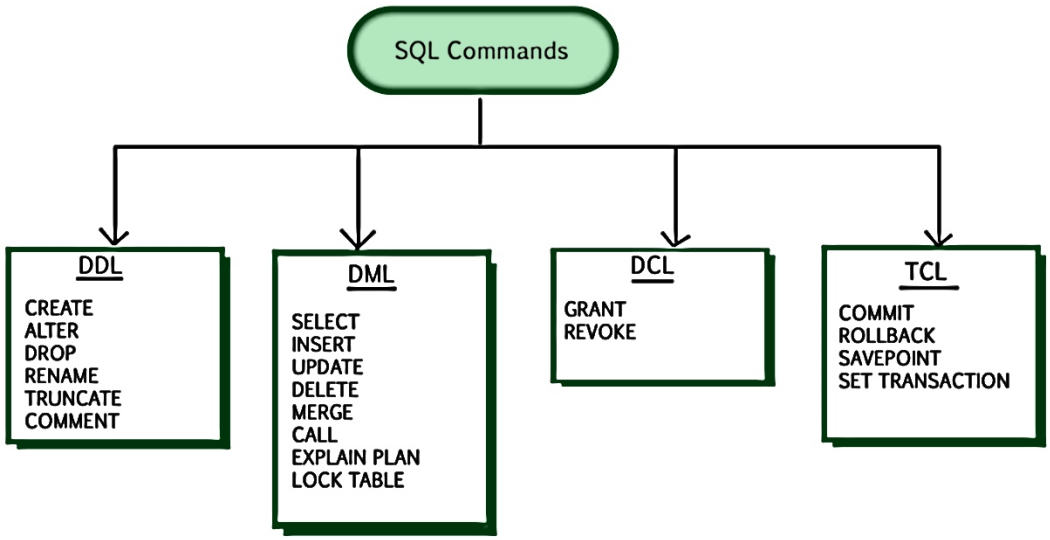
| | | | | |
|---|---|---|---|---|
| 1 | Finch | 25000 | NULL | NULL |
| 2 | Peter | 55000 | NULL | NULL |
| 3 | Warner | 52000 | NULL | NULL |
| 4 | Watson | 12312 | NULL | NULL |

Here output always came null, since performing any operation on null will always result in a *null value*.

**Note**: Make sure that NULL is **unavailable**, **unassigned**, **unknown**. Null is **not** same as *blank space* or *zero*.To get in depth understanding of NULL

## DDL, DML, TCL and DCL-

In this article, we'll be discussing Data Definition Language, Data Manipulation Language, Transaction Control Language, and Data Control Language.



## DDL (Data Definition Language) :

Data Definition Language is used to define the database structure or schema. DDL is also used to specify additional properties of the data. The storage structure and access methods used by the database system by a set of statements in a special type of DDL called a data storage and definition language. These statements define the implementation details of the database schema, which are usually hidden from the users. The data values stored in the database must satisfy certain consistency constraints.
For example, suppose the university requires that the account balance of a department must never be negative. The DDL provides facilities to specify such constraints. The database system checks these constraints every time the database is updated. In general, a constraint can be an arbitrary predicate pertaining to the database. However, arbitrary predicates may be costly to the test. Thus, the database system implements integrity constraints that can be tested with minimal overhead.

1. **Domain Constraints :** A domain of possible values must be associated with every attribute (for example, integer types, character types, date/time types). Declaring an attribute to be of a particular domain acts as the constraints on the values that it can take.
2. **Referential Integrity :** There are cases where we wish to ensure that a value appears in one relation for a given set of attributes also appear in a certain set of attributes in another relation i.e. Referential Integrity. For example, the department listed for each course must be one that actually exists.
3. **Assertions :** An assertion is any condition that the database must always satisfy. Domain constraints and Integrity constraints are special form of assertions.
4. **Authorization :** We may want to differentiate among the users as far as the type of access they are permitted on various data values in database. These differentiation are expressed in terms of Authorization. The most common being :

*read authorization* - which allows reading but not modification of data ;
*insert authorization* - which allow insertion of new data but not modification of existing data
*update authorization* - which allows modification, but not deletion.

**Some Commands:**

```
CREATE : to create objects in database

ALTER : alters the structure of database

DROP : delete objects from database

RENAME : rename an objects
```

Following SQL DDL-statement defines the department table :

```
create table department

(dept_name  char(20),

  building   char(15),

  budget     numeric(12,2));
```

Execution of the above DDL statement creates the department table with three columns - dept_name, building, and budget; each of which has a specific datatype associated with it.

## DML (Data Manipulation Language) :

DML statements are used for managing data with in schema objects. DML are of two types -

1. **Procedural DMLs** : require a user to specify what data are needed and how to get those data.
2. **Declarative DMLs** (also referred as **Non-procedural DMLs**) : require a user to specify what data are needed without specifying how to get those data.

Declarative DMLs are usually easier to learn and use than procedural DMLs. However, since a user does not have to specify how to get the data, the database system has to figure out an efficient means of accessing data.

## Some Commands :

```
SELECT: retrieve data from the database

INSERT: insert data into a table

UPDATE: update existing data within a table

DELETE: deletes all records from a table, space for the re
cords remain
```

**Example of SQL query that finds the names of all instructors in the History department :**

```
select instructor.name
 from instructor
 where instructor.dept_name = 'History';
```

The query specifies that those rows from the table instructor where the dept_name is History must be retrieved and the name attributes of these rows must be displayed.

## TCL (Transaction Control Language) :

Transaction Control Language commands are used to manage transactions in the database. These are used to manage the changes made by DML-statements. It also allows statements to be grouped together into logical transactions.

## Examples of TCL commands -

```
COMMIT: Commit command is used to permanently save any tra
nsaction into the database.

ROLLBACK: This command restores the database to last commi
tted state.


It is also used with savepoint command to jump to a savepo
int in a transaction.

SAVEPOINT: Savepoint command is used to temporarily save a
transaction so that you can rollback to that point wheneve
r necessary.
```

## DCL (Data Control Language) :

A Data Control Language is a syntax similar to a computer programming language used to control access to data stored in a database (Authorization). In particular, it is a component of Structured Query Language (SQL).

## Examples of DCL commands :

```
GRANT: allow specified users to perform specified tasks.

REVOKE: cancel previously granted or denied permissions.
```

The operations for which privileges may be granted to or revoked from a user or role apply to both the Data definition language (DDL) and the Data manipulation language (DML), and may include CONNECT,

SELECT, INSERT, UPDATE, DELETE, EXECUTE and USAGE.
In the Oracle database, executing a DCL command issues an implicit commit. Hence, you cannot roll back the command.

## Creating Roles-

A role is created to ease setup and maintenance of the security model. It is a named group of related privileges that can be granted to the user. When there are many users in a database it becomes difficult to grant or revoke privileges to users. Therefore, if you define roles:

- You can grant or revoke privileges to users, thereby automatically granting or revoking privileges.
- You can either create Roles or use the system roles pre-defined.

Some of the privileges granted to the system roles are as given below:

| System Roles | Privileges granted to the Role |
|---|---|
| Connect | Create table, Create view, Create synonym, Create sequence, Create session etc. |
| Resource | Create Procedure, Create Sequence, Create Table, Create Trigger etc. The primary usage of the Resource role is to restrict access to database objects. |
| DBA | All system privileges |

## Creating and Assigning a Role -

First, the (Database Administrator)DBA must create the role. Then the DBA can assign privileges to the role and users to the role.

**Syntax -**

```
CREATE ROLE manager;
Role created.
```

In the syntax:
'manager' is the name of the role to be created.

- Now that the role is created, the DBA can use the GRANT statement to assign users to the role as well as assign privileges to the role.
- It's easier to GRANT or REVOKE privileges to the users through a role rather than assigning a privilege directly to every user.
- If a role is identified by a password, then GRANT or REVOKE privileges have to be identified by the password.

### Grant privileges to a role -

```
GRANT create table, create view
TO manager;
Grant succeeded.
```

### Grant a role to users

```
GRANT manager TO SAM, STARK;
Grant succeeded.
```

### Revoke privilege from a Role :

```
REVOKE create table FROM manager;
```

**Drop a Role :**

```
DROP ROLE manager;
```

**Explanation -** Firstly it creates a manager role and then allows managers to create tables and views. It then grants Sam and Stark the role of managers. Now Sam and Stark can create tables and views. If users have multiple roles granted to them, they receive all of the privileges associated with all of the roles. Then create table privilege is removed from role 'manager' using Revoke.The role is dropped from the database using drop.

## Difference between functions and stored procedures in PL/SQL-

**Prerequisite:**

- Procedures in PL/SQL
- Functions in PL/SQL.

## Difference between functions and stored procedures in PL/SQL

Differences between Stored procedures(SP) and Functions(User-defined functions (UDF)):

**1. SP** *may or may not* return a value but **UDF must** return a value. The return statement of the function returns control to the calling program and returns the result of the function.
**Example:**

```
SP ->
```

```
create or replace procedure GEEKS(x int)
is y int;
begin
.....
UDF->
FUNCTION GEEKS(x int)
/*return statement so we must return value in funct
ion */
RETURN int IS
 y int;
BEGIN
.....
```

**2. SP** can have *input/output* parameters but **UDF only** has input parameters.

**Example:**

```
 SP ->
 CREATE OR REPLACE PROCEDURE Factorial(x IN NUMBER,
result OUT NUMBER)
 is
 begin
 ....
UDF ->
FUNCTION Factorial(x IN NUMBER) /* only input param
eter */
return  NUMBER
is
result NUMBER;
```

```
begin
.....
```

**3.** We can call **UDF** from **SP** but **cannot** call **SP** from a function.

**Example:**

```
Calling UDF cal() inside SP square() but reverse is
not possible.
set serveroutput on;
declare
a int;
c int;
function cal(temp  int)
return int
as
ans int;
begin
ans:=temp* temp;
return ans;
end;
procedure square(x in int, ans out int)
is
begin
dbms_output.put_line('calling function in procedure
');
ans:= cal(x);
end;
begin
```

```
a:=6;
square(a, c);
dbms_output.put_line('the answer is '|| c);
end;
OUTPUT:
calling function in the procedure
the answer is 36
```

4. **We cannot use SP in SQL statements like SELECT, INSERT, UPDATE, DELETE, MERGE, etc. but we can use them with UDF.**

**5. SP** is not allowed to be used in SELECT statements, but **UDF** can be used.

**6. SP** cannot be used anywhere in the WHERE/HAVING/SELECT part of SQL queries, but **UDF** can be used.

**7.** The execution plan can be re-used in **SP** but in **UDF** it will be compiled every time.

**8.** We can use try-catch exception handling in SP but we **cannot** do that in **UDF**.

**9.** We can use transactions in **SP** but it is not possible in **UDF**.

**10.** We can consider **UDF** as an expression but it is not possible in **SP.**

**11. SP** cannot be used in the join clause but it is possible in **UDF** as a resultset.

**12. SP** can have both table variables and temporary tables but **UDF** can have only table variables as it does not permit the use of temporary tables.

**Advantages of using UDF-**

- Enables faster execution
- Modular Programming
- Can reduce network traffic (data)

**Advantages of using** SP**-**

- Improved security measures
- Client/server traffic is reduced.
- Programming abstraction and efficient code reuse
- Pre-compiled Execution

## Trigger | Book Management Database-

Prerequisite - **SQL Trigger | Student Database** For example, given Library Book Management database schema with Student database schema. In these databases, if any student borrows a book from library then the count of that specified book should be decremented. To do so,*Suppose the schema with some data*,

```
mysql> select * from book_det;
+-----+-------------+--------+
| bid | btitle      | copies |
+-----+-------------+--------+
|   1 | Java        |     10 |
|   2 | C++         |      5 |
|   3 | MySql       |     10 |
|   4 | Oracle DBMS |      5 |
+-----+-------------+--------+
4 rows in set (0.00 sec)
mysql> select * from book_issue;
+------+------+--------+
| bid  | sid  | btitle |
+------+------+--------+
1 row in set (0.00 sec)
```

To implement such procedure, in which if the system inserts the data into the book_issue database a trigger should automatically invoke and decrements the copies attribute by 1 so that a proper track of book can be maintained.

**Trigger for the system -**

```
create trigger book_copies_deducts
after INSERT
on book_issue
for each row
update book_det set copies = copies - 1 where bid = new.bid;
```

Above trigger, will be activated whenever an insertion operation performed in a book_issue database, it will update the book_det schema setting copies decrements by 1 of current book id(bid).

**Results -**

```
mysql> insert into book_issue values(1, 100, "Java");
Query OK, 1 row affected (0.09 sec)
mysql> select * from book_det;
+-----+-------------+--------+
| bid | btitle      | copies |
+-----+-------------+--------+
|   1 | Java        |      9 |
|   2 | C++         |      5 |
|   3 | MySql       |     10 |
|   4 | Oracle DBMS |      5 |
+-----+-------------+--------+
4 rows in set (0.00 sec)
mysql> select * from book_issue;
```

```
+------+------+--------+
| bid  | sid  | btitle |
+------+------+--------+
|    1 |  100 | Java   |
+------+------+--------+
1 row in set (0.00 sec)
```

As above results show that as soon as data is inserted, copies of the book deducts from the book schema in the system.

**HIMANSHU KUMAR(LINKEDIN)**

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

**CREDITS- INTERNET**

DISCLOSURE- THE DATA AND IMAGES ARE TAKEN FROM GOOGLE AND INTERNET.

***CHECKOUT AND DOWNLOAD MY ALL NOTES***

***LINK*- https://linktr.ee/exclusive_notes**