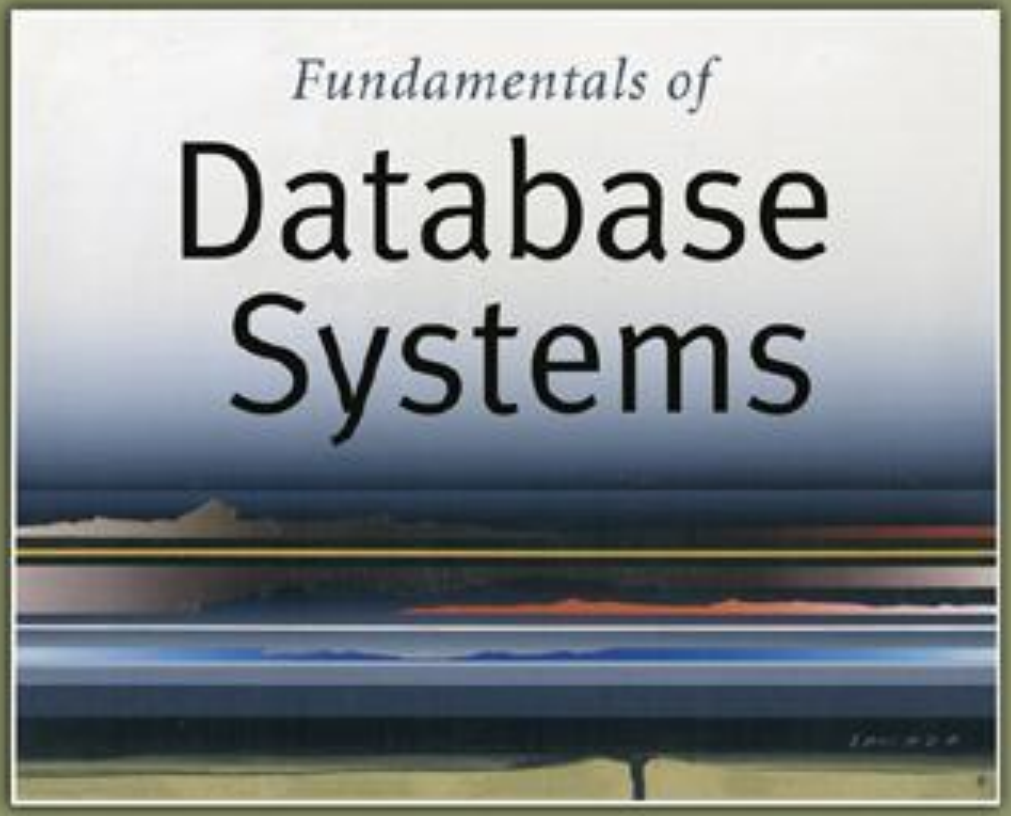


5th Edition

Elmasri / Navathe

DCL COMMANDS

DCL COMMANDS

- DCL includes commands such as GRANT and REVOKE which mainly deal with the rights, permissions, and other controls of the database system.
- Syntax:

  GRANT privileges_names ON object TO user;
- **Parameters Used**:
  - **privileges_name**: These are the access rights or privileges granted to the user.
  - **object:** It is the name of the database object to which permissions are being granted. In the case of granting privileges on a table, this would be the table name.
  - **user:** It is the name of the user to whom the privileges would be granted.

- **Privileges**:

Privilege	Description
SELECT	*select statement on tables*
INSERT	*insert statement on the table*
DELETE	*delete statement on the table*
INDEX	*Create an index on an existing table*
CREATE	*Create table statements*
ALTER	*Ability to perform ALTER TABLE to change the table definition*
DROP	*Drop table statements*
ALL	*Grant all permissions except GRANT OPTION*
UPDATE	*Update statements on the table*
GRANT	*Allows to grant the privilege that*

Different ways of granting privileges to the users:

- **Granting SELECT Privilege to a User in a Table:**
  - To grant Select Privilege to a table named “users” where User Name is Amit, the following GRANT statement should be executed.

    GRANT SELECT ON Users TO'Amit'@'localhost;

- **Granting more than one Privilege to a User in a Table**:
  - To grant multiple Privileges to a user named “Amit” in a table “users”, the following GRANT statement should be executed.

GRANT SELECT, INSERT, DELETE, UPDATE ON Users TO ‘Amit'@'localhost;

- **Granting All the Privilege to a User in a Table:**
  - To Grant all the privileges to a user named “Amit” in a table “users”, the following Grant statement should be executed.

    GRANT ALL ON Users TO 'Amit'@'localhost;

- **Granting a Privilege to all Users in a Table:**
  - To Grant a specific privilege to all the users in a table “users”, the following Grant statement should be executed.

    GRANT SELECT ON Users TO '*'@'localhost;

In the above example the “*” symbol is used to grant select permission to all the users of the table “users”.

Revoke command

- Revoke command withdraw user privileges on database objects if any granted.
- It does operations opposite to the Grant command.
- When a privilege is revoked from a particular user U, then the privileges granted to all other users by user U will be revoked.
- Syntax:
- REVOKE privileges ON object FROM user;

- **Example:**
- **grant insert**,
- select on accounts to Ram
- By the above command user ram has granted permissions on accounts database object like he can query or insert into accounts.

- **revoke insert**,
- select on accounts from Ram
- By the above command user ram's permissions like query or insert on accounts database object has been removed.

Aliases and DISTINCT

- **Aliases**
- In SQL, we can use the same name for two (or more) attributes as long as the attributes are in *different relations*

- Aliases are the temporary names given to tables or columns for the purpose of a particular SQL query.

- It is used when the name of a column or table is used other than its original name, but the modified name is only temporary.

Alias cntd…

- Aliases are created to make table or column names more readable.
- The renaming is just a temporary change and the table name does not change in the original database.
- Aliases are useful when table or column names are big or not very readable.
- These are preferred when there is more than one table involved in a query.

ALIASES

- Some queries need to refer to the same relation twice
  - In this case, *aliases* are given to the relation name
- Query 1: For each employee, retrieve the employee's name, and the name of his or her immediate supervisor.

```
Q1:  SELECT    E.FNAME, E.LNAME, S.FNAME, S.LNAME
     FROM      EMPLOYEE E S
     WHERE     E.SUPERSSN=S.SSN
```

  - In Q1, the alternate relation names E and S are called *aliases* or *tuple variables* for the EMPLOYEE relation
  - We can think of E and S as two different *copies* of EMPLOYEE; E represents employees in role of *supervisees(employee)* and S represents employees in role of *supervisors*

Syntax for Column Alias

- ***SELECT column as alias_name FROM table_name;***
  - ***column: fields in the table***

- **Column Alias**
- **Example:**
  - SELECT CustomerID AS SSN FROM Customer;

- **Table Alias**
  - Generally, table aliases are used to fetch the data from more than just a single table and connect them through field relations.
- **To fetch the CustomerName and Country of the customer with Age = 21.**
- **Example:**

```
SELECT s.CustomerName, d.Salary
FROM Customer AS s, Department
AS d WHERE s.Age=21 AND
s.DNO=d.DNO;
```

USE OF DISTINCT

- SQL does not treat a relation as a set; duplicate tuples can appear
- To eliminate duplicate tuples in a query result, the keyword **DISTINCT** is used
- For example, the result of Q11 may have duplicate SALARY values whereas Q11A does not have any duplicate values

```
Q11:    SELECT      SALARY
        FROM        EMPLOYEE

Q11A:   SELECT      DISTINCT SALARY
        FROM        EMPLOYEE
```

ANY Operator

- The ANY operator is used to verify if any single record of a query satisfies the required condition.
- This operator returns a TRUE, if the given condition is satisfied for any of the values in the range.
- If none of the values in the specified range satisfy the given condition, this operator returns false.
- You can also use another query (subquery) along with this operator.

Syntax

- Column_name operator ANY (subquery);
  - **column_name** is the name of a column in the main query.
  - **operator** is a comparison operator such as =, <, >, <=, >=, or <>.
  - **subquery** is a SELECT statement that returns a single column of values.

Example:

Consider the CUSTOMERS table which contains the personal details of customers including their name, age, address and salary etc.

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	Hyderabad	4500.00
7	Muffy	24	Indore	10000.00

- list out the details of all the CUSTOMERS whose SALARY is **greater than** the SALARY of any customer whose AGE is 32.
  - SELECT * FROM CUSTOMERS WHERE SALARY > ANY (SELECT SALARY FROM CUSTOMERS WHERE AGE = 32);

The result obtained is as follows –

ID	NAME	AGE	ADDRESS	SALARY
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	Hyderabad	4500.00
7	Muffy	24	Indore	10000.00

ALL Operator

- The SQL ALL operator returns all the records of the SELECT statement.
- It returns TRUE if the given condition is satisfied for ALL the values in the range.
- It always returns a Boolean value.
- It is used with SELECT, WHERE and HAVING statements in SQL queries.
- The data type of the values returned from a subquery must be the same as the outer query expression data type.

Syntax:

- Column_name operator ALL (subquery);
- Example
- The details of all the customers whose salary is **not equal to** the salary of any customer whose age is 25
  - SELECT * FROM CUSTOMERS WHERE SALARY <> ALL (SELECT SALARY FROM CUSTOMERS WHERE AGE = 25);

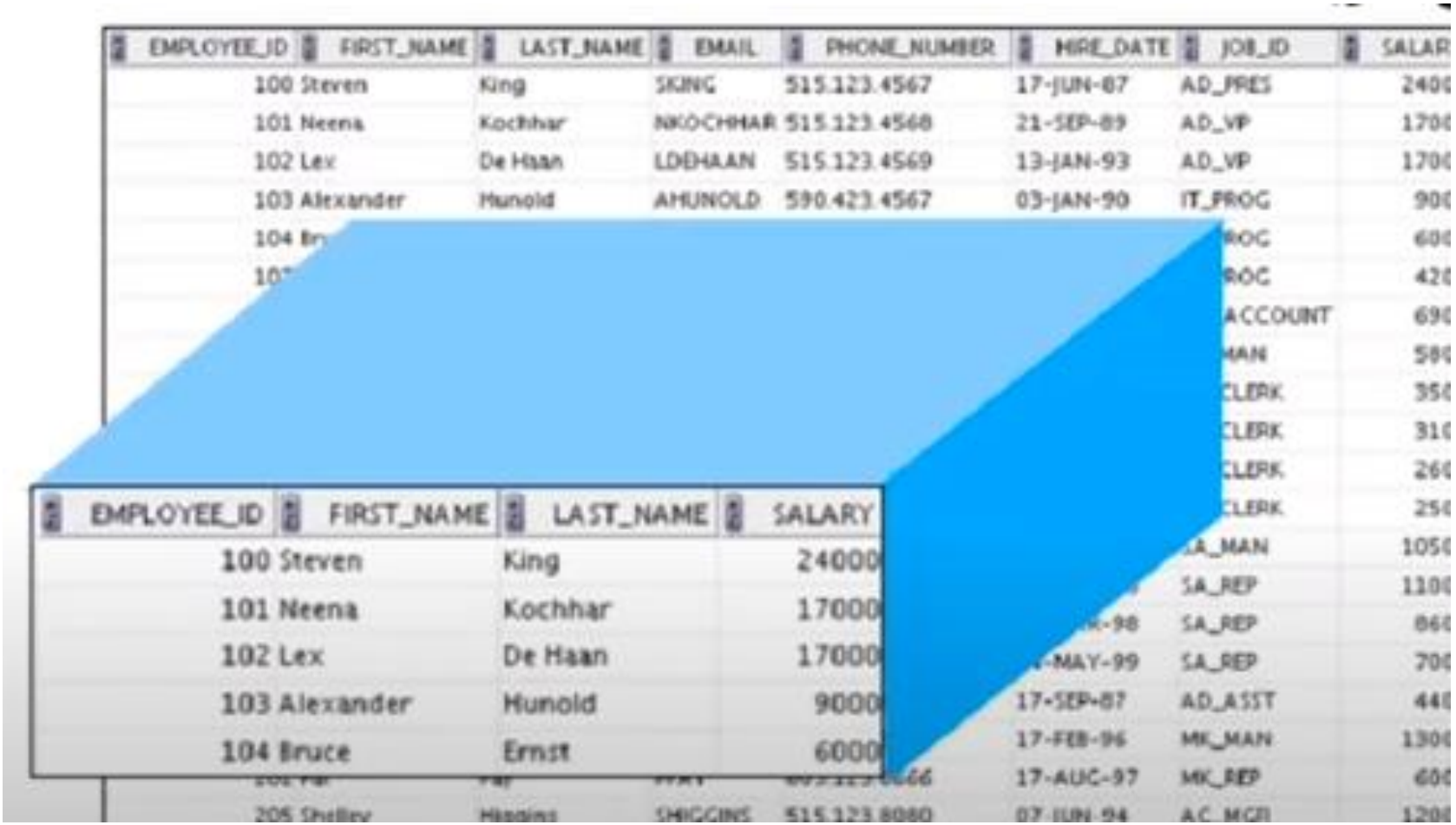
The output of the above query is as follows –

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
3	Kaushik	23	Kota	2000.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	Hyderabad	4500.00
7	Muffy	24	Indore	10000.00

Views

- Views is a virtual table.
- A view can represent a subset of a real table, selecting certain columns or certain rows from an ordinary table.
- A view can even represent joined tables.
- Because views are assigned separate permissions, you can use them to restrict table access so that the users see only specific rows or columns of a table.
- A view can be created from one or many tables

View



Creating view

- The PostgreSQL views are created using the **CREATE VIEW** statement.
- The PostgreSQL views can be created from a single table, multiple tables, or another view.

```
CREATE [TEMP|TEMPORARY] VIEW view_name
AS SELECT column1, column2.....
FROM table_name
WHERE [condition];
```

- If the optional TEMP or TEMPORARY keyword is present, the view will be created in the temporary space.
- Temporary views are automatically dropped at the end of the current session

Example

```
 id | name  | age | address    | salary
----+-------+-----+------------+--------
  1 | Paul  |  32 | California |  20000
  2 | Allen |  25 | Texas      |  15000
  3 | Teddy |  23 | Norway     |  20000
  4 | Mark  |  25 | Rich-Mond  |  65000
  5 | David |  27 | Texas      |  85000
  6 | Kim   |  22 | South-Hall |  45000
  7 | James |  24 | Houston    |  10000
```

```
CREATE VIEW COMPANY_VIEW
AS SELECT ID, NAME, AGE
FROM COMPANY;
```

- SELECT * FROM COMPANY_VIEW;

- This would produce the following result −

```
 id | name  | age
----+-------+-----
  1 | Paul  |  32
  2 | Allen |  25
  3 | Teddy |  23
  4 | Mark  |  25
  5 | David |  27
  6 | Kim   |  22
  7 | James |  24
(7 rows)
```

Dropping Views

- Obviously, where you have a view, you need a way to drop the view if it is no longer needed.
- The syntax is very simple and is given below −
  - DROP VIEW view_name;

- Following is an example to drop the CUSTOMERS_VIEW from the CUSTOMERS table.

- DROP VIEW CUSTOMERS_VIEW;

Advantages of PostgreSQL views

1) Simplifying complex queries
   - Views help simplify complex queries.
   - Typically, you first create views based on complex queries and store them in the database.
   - Then, you can use simple queries based on views instead of using complex queries.
2) Security and access control
   - You can create views that expose subsets of data in the base tables, hiding sensitive information.
   - This is particularly useful when you have applications that require access to distinct portions of the data.
3) Logical data independence
   - If your applications use views, you can freely modify the structure of the

Updating a View

- A view can be updated under certain conditions which are given below –
- The SELECT clause may not contain the keyword DISTINCT.
- The SELECT clause may not contain summary functions.
- The SELECT clause may not contain set functions.
- The SELECT clause may not contain set operators.
- The SELECT clause may not contain an ORDER BY clause.
- The FROM clause may not contain multiple tables.
- The WHERE clause may not contain subqueries.
- The query may not contain GROUP BY or HAVING.
- Calculated columns may not be updated.
- All NOT NULL columns from the base table must be included in the view in order for the
  INSERT query to function.

- **Inserting Rows into a View**
- Rows of data can be inserted into a view. The same rules that apply to the UPDATE
  command also apply to the INSERT command.
- Here, we cannot insert rows in the CUSTOMERS_VIEW because we have not included all the NOT NULL columns in this view, otherwise you can insert rows in a view in a similar way as you insert them in a table.

- **Deleting Rows into a View**
- Rows of data can be deleted from a view. The same rules that apply to the UPDATE and INSERT commands apply to the DELETE command.
- Following is an example to delete a record having AGE = 22.

```
SQL > DELETE FROM CUSTOMERS_VIEW  WHERE age = 22;
```

- This would ultimately delete a row from the base table CUSTOMERS and the same would reflect in the view itself

- Dropping Views
- Obviously, where you have a view, you need a way to drop the view if
  it is no longer needed. The syntax is very simple and is given below –

- DROP VIEW view_name;
- Following is an example to drop the CUSTOMERS_VIEW from the CUSTOMERS table.

Creating View from multiple tables

- View from multiple tables can be created by simply include multiple tables in the SELECT statement.
- In the given example, a view is created named MarksView from two tables
  Student_Detail and Student_Marks.
- Syntax:

```
CREATE VIEW view_name AS
SELECT table1.col_name1,
table1.col_name2,table2.col_nam
e1 FROM table1,table2
WHERE table1.col_name = Table2.col_name; (matching column)
```

- Example:

```
CREATE VIEW MarksView AS
SELECT Student_Detail.NAME,
Student_Detail.ADDRESS, Student_Marks.MARKS
FROM Student_Detail, Student_Mark
WHERE Student_Detail.NAME = Student_Marks.NAME;
```

Assertion

- An assertion is a constraint or condition that must always be true for the database to remain in a consistent state.
- Assertions are used to enforce business rules, integrity constraints, or any other conditions that the database administrator deems necessary for the data to be valid and meaningful.
- Example:
- For example, suppose you have a database for a library management system.
  - You might have an assertion that ensures that the total number of copies of a book available in the library cannot be negative.
  - This assertion would be enforced whenever a new copy is added or removed from the library.

- In SQL, assertions can be declared using the CREATE ASSERTION statement.
- However, not all database systems support assertions directly.
- Some databases provide alternative mechanisms for enforcing constraints, such as triggers, check constraints, or stored procedures.

Example

- Suppose we have a table "Orders" and we want to ensure that the total amount of all orders is always less than 10000.
  - CREATE ASSERTION orders_total CHECK ((SELECT SUM(amount) FROM Orders) < 10000); SQL
- This assertion ensures that the total amount of all orders will never exceed 10000.
- If an attempt is made to add an order that would cause the total to exceed 10000, the database system will reject it.

PL /SQL-

- Cursor
- Function
- Procedures
- Triggers

Cursor:

- In DBMS, a cursor is a database object used to retrieve or manipulate data row by row, especially when dealing with result sets that consist of multiple rows.
- Think of it like a pointer or a reference to a specific row within a result set.
- Cursors are particularly useful when you need to work with data sequentially, process one row at a time,
- or when you need to perform operations that can't be easily accomplished with set-based operations.

There are generally two types of cursors:

- **Implicit Cursor**:
  - Automatically created by the DBMS for SQL statements like SELECT, INSERT, UPDATE, or DELETE.
  - Implicit cursors are managed by the DBMS internally, and you don't need to explicitly declare or control them in your code.
- **Explicit Cursor**:
  - Created explicitly by the programmer using the DECLARE CURSOR statement.
  - Explicit cursors provide more control over the result set traversal, allowing the programmer to open, fetch, and close the cursor as needed.

Explicit cursors

- Explicit cursors in database management systems (DBMS) are programmatically created and managed by the developer to retrieve data row by row from a result set.

1. Declaration:

- First, you declare the cursor using the DECLARE CURSOR statement. This statement defines the SQL query that the cursor will use to fetch data.

```
DECLARE cursor_name CURSOR FOR
    SELECT column1, column2
    FROM table_name
    WHERE condition;
```

2. Opening the Cursor:

- After declaring the cursor, you need to open it using the OPEN statement. This step initializes the cursor and executes the SQL query, populating the result set.

```
OPEN cursor_name;
```

3. Fetching Rows:

- Once the cursor is opened, you can fetch rows from the result set one by one using the FETCH statement. This retrieves the next row of data from the result set and advances the cursor position.

```
FETCH cursor_name into variable1. variable2;
```

4. Processing Rows:

- After fetching a row, you can perform operations or calculations using the retrieved data. This step allows you to apply business logic or perform calculations on a row-by-row basis.

5. Closing the Cursor:

- Once you've finished processing all the rows, or when you no longer need the cursor, you should close it using the CLOSE statement. This releases the resources associated with the cursor.
  - CLOSE cursor_name;

Example

```sql
-- Create a table for demonstration
CREATE TABLE employees (
    employee_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50)
);

-- Insert some sample data
INSERT INTO employees (first_name, last_name)
VALUES ('John', 'Doe'),
       ('Jane', 'Smith'),
       ('Alice', 'Johnson');
```

- -- Declare the cursor
  - DECLARE ecsr CURSOR FOR SELECT eid FROM emp;

```
-- Declare variables to hold fetched data
DECLARE
    emp_id INT;
    emp_first_name VARCHAR(50);
    emp_last_name VARCHAR(50);

    -- Declare an explicit cursor
    CURSOR emp_cursor IS
        SELECT employee_id, first_name, last_name
        FROM employees;

-- Open the cursor
OPEN emp_cursor;

-- Fetch and process each row
LOOP
    FETCH emp_cursor INTO emp_id, emp_first_name, emp_last_name;
    EXIT WHEN NOT FOUND; -- Exit loop when no more rows
```

- -- Output the retrieved data
  - RAISE NOTICE 'Employee ID: %, Name: % Surname:%', emp_id, emp_first_name, emp_last_name;
  - END LOOP;

- -- Close the cursor
  - CLOSE emp_cursor;

Example:

```
company=# DO $$
company$# declare
company$# emp_id int;
company$# ecsr cursor for select eid from emp;
company$# begin
company$# open ecsr;
company$# LOOP
company$# fetch ecsr into emp_id;
company$# exit when not found;
company$# raise notice 'Employee Id:%',emp_id;
company$# end loop;
company$# close ecsr;
company$# end;
company$# $$;
NOTICE:  Employee Id:101
NOTICE:  Employee Id:102
NOTICE:  Employee Id:103
NOTICE:  Employee Id:104
DO
```

Stored Functions and Procedures

"A **procedures** or **function** is a group or set of SQL statements that perform a specific task.“

The major difference between a procedure and a function is, a function must always return a value, but a procedure may or may not return a value (in SQL).

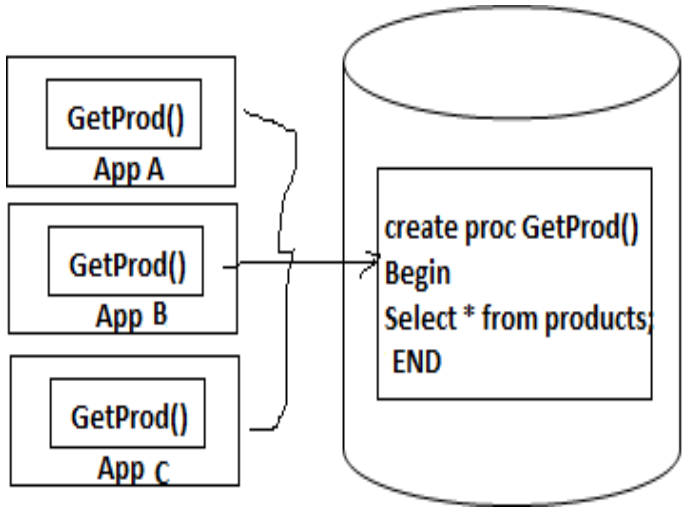
Differences between Function & Procedure.

- Following are the important differences between SQL Function and SQL Procedure.

Sr. No.	Key	Function	Procedure
1	Definition	A function is used to calculate result using given inputs.	A procedure is used to perform certain task in order.
2	Call	A function can be called by a procedure.	A procedure cannot be called by a function.
3	DML	DML statments cannot be executed within a function.	DML statements can be executed within a procedure.
4	SQL, Query	A function can be called within a query.	A procedure cannot be called within a query.
5	SQL, Call	Whenever a function is called, it is first compiled before being called.	A procedure is compiled once and can be called multiple times without being compiled.
6	SQL, Return	A function returns a value and control to calling function or code.	A procedure returns the control but not any value to calling function or code.

Procedures

- A procedure/stored procedure is a named PL/SQL block which performs one or more specific task.
- This is similar to a procedure in other programming languages.
- Stored procedures can increase productivity by writing once and using it many times.
- A stored procedure has a header and a body.
- The header consists of the name of the procedure and the parameters or variables passed to the procedure.
- The body consists or declaration section, execution section and exception section similar to a general PL/SQL Block.



Creating Procedure

```
CREATE [OR REPLACE] PROCEDURE procedure_name
  ([parameter datatype [,...] ])

LANGUAGE SQL

AS $$

  sql_body

$$;
```

OR REPLACE	Optional. CREATE PROCEDURE defines a new procedure. CREATE OR REPLACE PROCEDURE will either create a new procedure, or replace an existing procedure.
procedure_name	Required. Specify the name to assign to this procedure.
parameter	Optional. Specify one or more parameters passed into the procedure.
sql_body	Specify the block of SQL statements to be executed.

- To invoke a stored procedure, the CALL statement is used.
- Syntax:
  - CALL procedure_name(parameter(s));

Example

```
 empid |   name    |    dept
-------+-----------+------------
     1 | Leonardo  | Sales
     2 | Raphael   | Accounting
     3 | Donatello | Sales
(3 rows)
```

- Create a procedure to display record of employees belongs to ‘Sales’ dept.

```
CREATE OR REPLACE PROCEDURE get_employee()
LANGUAGE plpgSQL
AS $$
BEGIN
SELECT * FROM employees WHERE dept='Sales';
END;
$$;
```

Example 2

NEW | POSTGRESQL | RUN

```
14  select * from stored_Procedure_Prac;
15
16  create or replace procedure prac_transfer(
17     sender int,
18     receiver int,
19     amount dec
20  )
21  language plpgsql
22  as $$
23  begin
24      -- subtracting the amount from the sender's account
25      update stored_Procedure_Prac
26      set balance = balance - amount
27      where id = sender;
28
29      -- adding the amount to the receiver's account
30      update stored_Procedure_Prac
31      set balance = balance + amount
32      where id = receiver;
33
34      commit;
35  end;$$;
36
37  call prac_transfer(1, 2, 1000);
38
39  SELECT * FROM stored_Procedure_Prac;
40
```

STDIN

Input for the program (Optional)

```
CREATE TABLE
INSERT 0 1
INSERT 0 1
 id | name |  balance
----+------+-----------
  1 | Don  | 400000.00
  2 | Bob  | 400000.00
(2 rows)

CREATE PROCEDURE
CALL
 id | name |  balance
----+------+-----------
  1 | Don  | 399000.00
  2 | Bob  | 401000.00
(2 rows)
```

Create Procedure with Parameter

```
CREATE PROCEDURE get_employees(var1 integer)

LANGUAGE SQL
AS $$

  SELECT * FROM Employee LIMIT var1;
  SELECT COUNT(EmpID) AS TotalEmployee FROM Employee;

$$;
```

```
EmpID | Name   | City       | Age | Salary
------+--------+------------+-----+--------
1     | John   | London     | 25  | 3000
2     | Marry  | New York   | 24  | 2750
3     | Jo     | Paris      | 27  | 2800
4     | Kim    | Amsterdam  | 30  | 3100
5     | Ramesh | New Delhi  | 28  | 3000
6     | Huang  | Beijing    | 28  | 2800
```

After successful execution, the procedure can be called as follows:

```
CALL get_employees(3);
```

```
EmpID | Name  | City     | Age | Salary
------+-------+----------+-----+--------
1     | John  | London   | 25  | 3000
2     | Marry | New York | 24  | 2750
3     | Jo    | Paris    | 27  | 2800

 TotalEmployee
---------------
 6
```

Stored function

Stored function

- A stored function in MySQL is a set of SQL statements that perform some task/operation and return a single value.
- The stored function is almost similar to the procedure in MySQL, but it has some differences that are as follows:
  - The function parameter may contain only the **IN parameter** but can't allow specifying this parameter, while the procedure can allow **IN, OUT, INOUT parameters**.
  - The stored function can return only a single value definedin the function header.
  - The stored function may also be called within SQL statements.
  - It may not produce a result set.
- Thus, we will consider the stored function

Functions in postgresql

- The basic syntax to create a function is as follows −
- CREATE [OR REPLACE] FUNCTION function_name (arguments)
- RETURNS return_datatype
- language plpgsql
- AS
- $variable_name$
- declare
  - -- variable declaration
- begin
  - -- stored procedure body
- end; $$

- **function-name** specifies the name of the function.
- [OR REPLACE] option allows modifying an existing function.
  - The function must contain a **return** statement.
- **RETURN** clause specifies that data type you are going to return from the function.
  - The **return_datatype** can be a base, composite, or domain type, or can reference the type of a table column.
- **function-body** contains the executable part.
  - The AS keyword is used for creating a standalone function.
- **plpgsql** is the name of the language that the function is implemented in.
  - Here, we use this option for PostgreSQL, it Can be SQL, C, internal, or the name of a user-defined procedural language.
  - For backward compatibility, the name can be enclosed by single quotes.

Example

- Function totalRecords() is as follows −

	id [PK] integer	name character varying (100)	age integer	address character varying (255)	salary numeric (10,2)
1	1	Paul	32	California	20000.00
2	2	Allen	25	Texas	15000.00
3	3	Teddy	23	Norway	20000.00
4	4	Mark	25	Rich-Mond	65000.00
5	5	David	27	Texas	85000.00
6	6	Kim	22	South-Hall	45000.00

- Create a function

```
CREATE OR REPLACE FUNCTION get_employee_count()
RETURNS INTEGER AS $$
DECLARE
   total_records INTEGER;
BEGIN
   -- Execute a SQL query to count the number of records in the employee table
   SELECT COUNT(*) INTO total_records FROM employee;

   -- Return the total number of records
   RETURN total_records;
END;
```

- CallIng a function :
  - select totalRecords();

	totalrecords integer
1	7

```
CREATE FUNCTION deactivate_unpaid_accounts() RETURNS void
LANGUAGE SQL
AS $$
  UPDATE accounts SET active = false WHERE balance < 0;
$$;
```

```
SELECT deactivate_unpaid_accounts();
```

Triggers

Triggers

- A trigger is a special type of stored procedure that automatically runs when an event occurs in the database server.
- DML triggers run when a user tries to modify data through a data manipulation language (DML) event.
- DML events are INSERT, UPDATE, or DELETE statements on a table or view.
- These triggers fire when any valid event fires, whether table rows are affected or not.

```
create trigger [trigger_name]
[before | after]
{insert | update | delete}
on [table_name]
[for each row]
[trigger_body]
```

- Create Triger syntax:
- **Explanation of syntax:**
- create trigger [trigger_name]:
  - Creates or replaces an existing trigger with the trigger_name.
- [before | after]:
  - This specifies when the trigger will be executed.
- {insert | update | delete}:
  - This specifies the DML operation.
- on [table_name]:
  - This specifies the name of the table associated with the trigger.

- [for each row]:
  - This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected.
- [trigger_body]:
  - This provides the operation to be performed as trigger is fired

- **BEFORE and AFTER of Trigger:**
  - BEFORE triggers run the trigger action before the triggering statement is run.
  - AFTER triggers run the trigger action after the triggering statement is run.

- **For example,** before or after the triggering event the **OLD and NEW** signify the row's states in the table.
- How to Create a New Trigger
- We will follow the below process to generate a new trigger in PostgreSQL:
  - **Step1:** Firstly, we will create a trigger function with the help of the **CREATE FUNCTION** command.
  - **Step2:** Then, we will fix the trigger function to a table with the help of the **CREATE TRIGGER** command.

Trigger

- Step 1: Syntax of Create trigger function

```
CREATE FUNCTION trigger_function()
  RETURNS TRIGGER
  LANGUAGE PLPGSQL
AS $$
BEGIN
  -- trigger logic goes here?
END;
$$
```

- Step 2: Syntax of Creating trigger

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER} { event }
 ON table_name
 [FOR [EACH] { ROW | STATEMENT }]
 EXECUTE PROCEDURE trigger_function
```

Trigger _function example

```
CREATE OR REPLACE FUNCTION data_log()
  RETURNS TRIGGER
  LANGUAGE PLPGSQL
  AS
$$
BEGIN
INSERT INTO main(Backup_id,Backup_name)
         VALUES(OLD.ID,OLD.name);

RETURN NEW;
END;
$$
```

```
create trigger data_backup
after delete
on origin
for each row
execute procedure data_log();
```

Example 2

- write a code to implement trigger in postgresql. create table student(id,name,mark1,mark2,total) apply trigger on it, when new row will be inserted into student table total will be calculated automatically. (total=marks1+marks2)

```sql
-- Create the student table
CREATE TABLE student (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    mark1 INTEGER,
    mark2 INTEGER,
    total INTEGER
);
```

```sql
-- Create the trigger function
CREATE OR REPLACE FUNCTION calculate_total()
RETURNS TRIGGER AS $$
BEGIN
    NEW.total := NEW.mark1 + NEW.mark2;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

```sql
-- Create the trigger
CREATE TRIGGER update_total
BEFORE INSERT ON student
FOR EACH ROW
EXECUTE FUNCTION calculate_total();
```

```
 id | sname | maths | physics | tot
----+-------+-------+---------+-----
(0 rows)
```

```sql
-- Insert the row
INSERT INTO student VALUES (1, 'Amit', 60, 70, 0);
```

```
 id | name | mark1 | mark2 | total
----+------+-------+-------+-------
  1 | Amit |    60 |    70 |   130
(1 row)
```

TCL Commands

- ***COMMIT***
- ***ROLLBACK***
- ***SAVEPOINT***

Transaction Control Languages.

- These commands are used for maintaining consistency of the database and for the management of transactions made by the DML commands.
- A **Transaction** is a set of SQL statements that are executed on the data stored in DBMS.
- Whenever any transaction is made these transactions are temporarily happen in database.
- So to make the changes permanent, we

use **TCL** commands.

TCL

Control transactions in a table

Any operation performed on a table using DML

ID	Name	AGE	COURSE
1	Adil	21	Computer
2	Aziz	22	Mathematics
3	Kashif	24	Physics
4	Waqar	20	Chemistry

- insert data
- delete data
- update data

TCL Commands:

- **1. COMMIT :**
- This command is used to save the data permanently.
- Whenever we perform any of the DML command like -INSERT, DELETE or UPDATE, these can be rollback if the data is not stored permanently.
- So in order to be at the safer side COMMIT command is used.

  Syntax: commit;

- **2. ROLLBACK :**
- This command is used to get the data or restore the data to the last savepoint or last committed state.
- If due to some reasons the data inserted, deleted or updated is not correct, you can rollback the data to a particular savepoint or if savepoint is not done, then to the last committed state.

  Syntax: rollback;

Note :- Table can be rolled-back only if it is temporary.
If you committed your changes, it cannot be rolled-back.

- **3. SAVEPOINT :**
- This command is used to save the data at a particular point temporarily, so that whenever needed can be rollback to that particular point.
- **Syntax**: Savepoint A;

Example:

- Consider the following Table Student:

```
UPDATE STUDENT SET NAME = 'Sherlock' WHERE NAME = 'Jolly';
COMMIT;
ROLLBACK;
```

- By using this command you can update the record and save it permanently by using **COMMIT** command.
- Now after COMMIT :

Name	Marks
John	79
Jolly	65
Shuzan	70

Name	Marks
John	79
Sherlock	65
Shuzan	70

- If commit was not performed then the changes made by the update command can be rollback.

Example:

```
SET autocommit = 1;
use mydb;

create table student(sid int(3),sname varchar(15));

insert into student values(101,'abc');
insert into student values(102,'abc');
insert into student values(103,'abc');

commit;
```

```
SET autocommit = 1;
use mydb;

select * from student;

delete from student where sid=103;
```

```
delete from student where sid=103;
rollback;
```

```
delete from student where sid=103;
commit;
rollback;
```

Savepoint

- If on the given table savepoint is performed

Name	Marks
John	79
Jolly	65
Shuzan	70

```
Insert into student values ('Jack',95);
Update student set name= 'Rossie' where marks =70;
SAVEPOINT A
Insert into student values (zack',76);
SAVEPOINT B
Insert into student values ('Bruno',85);
SAVEPOINT C
```

Name	Marks
John	79
Jolly	65
Rossie	70
Jack	95
Zack	76
Bruno	85

- Now if we Rollback to Savepoint B:
- Rollback to B;
- The resulting Table will be-

Name	Marks
John	79
Jolly	65
Rossie	70
Jack	95
Zack	76

- Now if we Rollback to Savepoint A:
- Rollback to A;
- The resulting Table will be-

Name	Marks
John	79
Jolly	65
Rossie	70
Jack	95

Thank you