CHAPTER 1

Relational Database Design Using PL/SQL

Contents...

Objectives...

- To learn Basics for PL/SQL and PL/pgSQL
- To know the Language Structure of PL/pgSQL
- To learn Programming Control Flow Statements like Conditional Statements, Loops etc.
- To define Stored Procedures, Stored Functions, Cursors and Triggers
- To handle Errors and Exceptions in PL/pgSQL

1.0 INTRODUCTION

- A relational database stores data in a set of simple relations, which perceives as tables. Each relation is composed of records or tuples and attributes or fields. A relational database is created using the relational model defined by E. F. Codd.
- A Relational DataBase Management System (RDBMS) is a software program use to create, maintain, modify and manipulate a relational database.
- Relational database is defined as, "the collection of normalized or structured relations with distinct relation names". The standard user and Application Programming Interface (API) of a relational database is the (SQL).
- SQL is the now - ubiquitous language for both querying and updating - never mind the name - of relational databases. Oracle Corporation introduced PL/SQL to overcome some limitations in SQL and to provide a more complete programming solution for those who sought to build mission-critical applications to run against the Oracle database.
- The PL/SQL (Procedural Language/Structured Query Language) programming language was developed by Oracle Corporation in the late 1980s as procedural extension language for SQL and the Oracle relational database.
- A language extension is a set of features that somehow enhance an existing language. The purpose of PL/SQL is to combine database language and procedural programming language. PL/SQL is a completely portable, high-performance transaction-processing language.
- The wide use of object oriented techniques/technologies, web applications, transitional databases etc. there is a need of developing a relational database processing with object oriented features.
- For this reason PostgreSQL starts from the University of California at Berkeley, in the late 1970s with the aim/goal/objective of developing a relational database possessing with object-oriented features. They named it Ingres. Later on, around the mid 1980s, a team of core developers led by Michael Stonebraker from the University of California started work on Ingres. The team added core object-oriented features in Ingres and named the new version PostgreSQL.

- PostgreSQL (Procedural Language/PostgreSQL) is a general-purpose Object Relational Database Management System (ORDBMS). It allows adding custom functions developed using different programming languages such as C/C++, Java, etc.
- PL/pgSQL is a loadable procedural language like PL/SQL for the PostgreSQL database system. The design of PL/pgSQL aimed to allow PostgreSQL users to perform more complex operations and computations than SQL, while providing ease of use.
- PL/pgSQL is a powerful extension for SQL that combines expressive power of SQL the more typical features of programming languages.
- PL/pgSQL is procedural language similar to Oracle's PL/SQL. Like PL/SQL, PL/pgSQL is a block structured language that support variable declaration, loops, logical constructs and advanced error handling.
- In this chapter, we study PL/SQL and PL/pgSQL. PL/pgSQL adds control structures such as conditionals, loops and exception handling to the SQL language. When we write a PL/pgSQL function, we can include any and all SQL commands, as well as the procedural statements added by PL/pgSQL.
- PL/SQL stands for 'Procedural Language extensions to the Structured Query Language (SQL)', used for designing relational databases.
- Relational database design is the most important task performed by application developers because the resulting database and all applications that access this database are based on this design.
- The goal/objective of a relational database design is to generate set of relation schemes that allows us to store information without unnecessary redundancy and also to retrieve the information easily.
- PL/pgSQL, as a fully featured programming language, allows much more procedural control than SQL, including the ability to use loops and other control structures. SQL statements and triggers can call functions created in the PL/pgSQL language.

1.1 INTRODUCTION TO PL/SQL

- PL/SQL is an extension of Structured Query Language (SQL) that is used in Oracle. Unlike SQL, PL/SQL allows the programmer to write code in a procedural language format. Full form of PL/SQL is "Procedural Language extensions to SQL".
- PL/SQL combines the data manipulation power of SQL with the processing power of procedural language to create super powerful SQL queries.
- PL/SQL means instructing the compiler 'what to do' through SQL and 'how to do' through its procedural way.
- Similar to other database languages, PL/SQL gives more control to the programmers by the use of loops, conditions and object-oriented concepts.
- PL/SQL is a block structured language that enables developers to combine the power of SQL with procedural statements. All the statements of a block are passed to SQL engine all at once which increases processing speed and decreases the traffic.

Disadvantages of SQL:

1. SQL does not provide procedural language capabilities/techniques such as condition checking, looping and branching.
2. SQL statements are passed to Oracle engine one at a time which increases traffic and decreases speed.
3. SQL has no facility of error checking during manipulation of data.

Features of PL/SQL:

1. PL/SQL is basically a procedural language, which provides the functionality of decision making, iteration and many more features of procedural programming languages.
2. PL/SQL can execute a number of queries in one block using single command.
3. One can create a PL/SQL unit such as procedures, functions, packages, triggers and types, which are stored in the database for reuse by applications.
4. PL/SQL provides a feature to handle the exception which occurs in PL/SQL block known as exception handling block.
5. Applications written in PL/SQL are portable to computer hardware or operating system where Oracle is operational.
6. PL/SQL offers extensive error checking.

Differences between SQL and PL/SQL:

Sr. No.	SQL	PL/SQL
1.	SQL is a single query that is used to perform DML and DDL operations.	PL/SQL is a block of codes that used to write the entire program blocks/procedure/ function, etc.
2.	It is declarative, that defines what needs to be done, rather than how things need to be done.	PL/SQL is procedural that defines how the things needs to be done.
3.	Execute as a single statement.	Execute as a whole block.
4.	Mainly used to manipulate data.	Mainly used to create an application.
5.	Cannot contain PL/SQL code in it.	It is an extension of SQL, so it can contain SQL inside it.

Structure of PL/SQL Block:

- PL/SQL extends SQL by adding constructs found in procedural languages, resulting in a structural language that is more powerful than SQL.
- The basic unit in PL/SQL is a block. All PL/SQL programs are made up of blocks, which can be nested within each other.
- Typically, each block performs a logical action in the program. A block in PL/SQL has the following structure:

```
DECLARE
    declaration statements;
BEGIN
    executable statements
EXCEPTION
    exception handling statements
END;
```

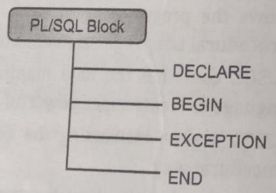

Fig. 1.1: Sections of a PL/SQL Block

- PL/SQL block contains following sections:

1. **Declare section** starts with DECLARE keyword in which variables, constants, records as cursors can be declared which stores data temporarily. It basically consists definition of PL/SQL identifiers. This part of the code is optional.

2. **Execution section** starts with BEGIN and ends with END keyword. This is a mandatory section and here the program logic is written to perform any task like loops and conditional statements. It supports all DML commands, DDL commands and SQL*PLUS built-in functions as well.
3. **Exception section** starts with EXCEPTION keyword. This section is optional which contains statements that are executed when a run-time error occurs. Any exceptions can be handled in this section.

'Hello World' Example using PL/SQL:

```
DECLARE
   message  varchar2(20):= 'welcome';
BEGIN
   dbms_output.put_line(message);
END;
/
```

- The END; line signals the end of the PL/SQL block. To run the code from the SQL command line, we may need to type '/' at the beginning of the first blank line after the last line of the code. When the above code is executed at the SQL prompt, it produces the following result/output:

```
Welcome
```

1.2 INTRODUCTION TO PL/pgSQL

- PL/pgSQL (Procedural Language/PostgreSQL) is a loadable procedural language for the PostgreSQL database system.
- A procedural language is a programming language which specifies order or sequence of steps that are followed to produce a desired result or output.
- We can use PL/pgSQL to group sequences of SQL and programming statements together within a database server. It allows us to develop complex functions in PostgreSQL that may not be possible using plain SQL statements.
- PL/pgSQL is a block-structured language. The complete text of a function definition must be a block.
- Currently procedural languages available in the standard PostgreSQL distribution are PL/pgSQL, PL/Tcl, PL/Perl and PL/Python.

1.2.1 Features of PL/pgSQL

- The features of PL/pgSQL are listed below:
  1. PL/pgSQL is easy to use.
  2. In PostgreSQL 9.0 and later, PL/pgSQL is installed by default.
  3. PL/pgSQL adds control structures to the SQL language.
  4. PL/pgSQL can be used to create functions and trigger procedures.
  5. It can perform complex computations.
  6. PL/pgSQL inherits all user-defined types, functions and operators.
  7. It can be defined to be trusted by the server.

1.2.2 Advantages of using PL/pgSQL

- Advantages of using PL/pgSQL are listed below:

1. **Portability:**

- One of important aspect of using PL/pgSQL is its portability as its functions are compatible with all platforms that can operate the PostgreSQL database system.

2. **SQL Support:**

- The use of SQL within PL/pgSQL code can increase the power, flexibility and performance of the programs. One can access to all data types, operators, and functions within PL/pgSQL code.
- If multiple SQL statements are executed from a PL/pgSQL code block, the statements are processed at one time, instead of the normal behavior of processing a single statement at a time.

3. **Better Performance:**

- PostgreSQL databases use SQL as query language. But every SQL statement must be executed individually by the database server. So client application must send each query to the database server, wait for it to be processed, receive and process the results, and do some computation, then send further queries to the server.
- All this incurs lot of inter-process communication and will also incur network overhead if the client is on a different machine than the database server.
- But with PL/pgSQL one can group a block of computation and a series of queries inside the database server, thus having the power of a procedural language, the ease of use of SQL and considerable savings of client/server communication overhead.
- It eliminates extra round trips between client and server, also avoids multiple times query parsing. This can result in a considerable performance increase as compared to an application that does not use PL/pgSQL.

1.2.3 Developing in PL/pgSQL

- Developing in PL/pgSQL is very easy and simple. There are two ways to develop in PL/pgSQL:
  1. We can use the text editor to create our functions and use psql to load and test those functions.
  2. Another good way to develop in PL/pgSQL is with a GUI database access tool that facilitates development in a procedural language such as PgAccess.

1.2.4 To Add Language PL/pgSQL to Database

- A procedural language must be "installed" into each database where it is to be used. But procedural languages installed in the default database are automatically available in all subsequently created databases.
- To add PL/pgSQL to the PostgreSQL database, we can either use the createlang application from the command line, or the CREATE LANGUAGE SQL command from within a database client such as psql.

1. **Using psql to add PL/pgSQL:**

- The use of the CREATE LANGUAGE command first requires the creation of the PL/pgSQL call handler, which is the function that actually processes and interprets the PL/pgSQL code.
- CREATE LANGUAGE is the SQL command which adds procedural languages to the currently connected database. Before it can be used, however, the CREATE FUNCTION command must first be used to create the procedural call handler.
- Here, is the **syntax** to create a PL/pgSQL call handler with CREATE FUNCTION:

```
CREATE FUNCTION plpgsql_call_handler()
        RETURNS OPAQUE AS '/postgres_library_path/plpgsql.so' LANGUAGE 'C'
```

- In above syntax, postgres_library_path is the absolute system path to the installed PostgreSQL library files. This path, by default, is /usr/local/pgsql/lib.
- Following example uses the CREATE FUNCTION command to create the PL/pgSQL call handler, assuming the plpgsql.so file is in the default location.

Example: Creating the PL/pgSQL call handler.

```
postgres=#   CREATE FUNCTION plpgsql_call_handler ()
postgres-#   RETURNS OPAQUE
postgres-#   AS '/usr/local/pgsql/lib/plpgsql.so'
postgres-#   LANGUAGE 'C';
CREATE
```

- Above example only creates the function handler; the language itself must also be added with the CREATE LANGUAGE command. Here, is the **syntax** to add PL/pgSQL to a database with following command:

```
CREATE LANGUAGE 'plpgsql' HANDLER plpgsql_call_handler
LANCOMPILER 'PL/pgSQL'
```

- In above syntax, plpgsql is the name of the language to be created, the plpgsql_call_handler is the name of the call handler function (e.g., the one created in previous Example, and the PL/pgSQL string constant following the LANCOMPILER keyword is an arbitrary descriptive note.
- Following example adds PL/pgSQL to the postgres database with the CREATE LANGUAGE command.

Example: Adding PL/pgSQL with CREATE LANGUAGE.

```
postgres=# CREATE LANGUAGE 'plpgsql' HANDLER plpgsql_call_handler
postgres-# LANCOMPILER 'PL/pgSQL';
CREATE
```

2. **Using createlang to add PL/pgSQL:**

- The createlang utility is simpler to use, as it abstracts the creation of the call handler and the language away from the user.
- To execute createlang we will first need to be at the command prompt. If the operating system username we are currently logged into is the same as that of a database superuser account on the target database, we can call createlang with the command shown in following in example.

Example: Using createlang as a Database Superuser.

```
$ cd /usr/local/pgsql/bin
postgres=# createlang plpgsql postgres
```

- The createlang program will return you to a shell prompt upon successful execution.

1.3 PL/pgSQL: LANGUAGE STRUCTURE

- PL/pgSQL's structure is similar to other programming languages such as 'C' language, in which each portion of code acts (and is created) as a function, all variables must be declared before being used, and code segments accept arguments when called and return arguments at their end.
- In this section, we will discuss the block organization of PL/pgSQL code, how to use comments, how PL/pgSQL expressions are organized, and the usage of statements.
- PL/pgSQL is a block-structured and case-insensitive language. A block comprises statements inside the same set of the DECLARE/BEGIN and END statements.
- **A block in PL/pgSQL** can be defined as follows:

```
[ <<label>> ]
[ DECLARE
```

```
    declarations ]
BEGIN
    statements
END [ label ];
```

- Each declaration and each statement within a block is terminated by a semicolon (;). A block that appears within another block must have a semicolon after END, as shown above; however the final END that concludes a function body does not require a semicolon.
- A 'label' is only needed if you want to identify the block for use in an EXIT statement, or to qualify the names of the variables declared in the block. If a label is given after END, it must match the label at the block's beginning.

1.3.1 Structure of PL/pgSQL Code Block

- PL/pgSQL code is organized as block structured code where each portion of code is designed to exist as a function.
- Code blocks are entered within a SQL CREATE FUNCTION call that creates the PL/pgSQL function in the PostgreSQL database.
- All variables must be declared before being used, and code segments accept arguments when called and return arguments at their end.
- The following example shows the structure of PL/pgSQL.

Example: Structure of a PL/pgSQL code block.

```
CREATE FUNCTION identifier (arguments) RETURNS type AS'
  DECLARE
    declaration;
    [...]
  BEGIN
    statement;
    [...]
  END;
'LANGUAGE 'plpgsql';
```

- The declaration section is denoted by the DECLARE keyword. In the declaration section of a code block, all variables are declared, and optionally initialized to a default value.
- A variable declaration specifies the variable's name and type. Each variable declaration is ended with a semicolon (;).
- After declaring variables, the main body of the code block is started with the BEGIN keyword. The code block's statements should appear after the BEGIN keyword.
- The END keyword designates the end of the code block. The main block of a PL/pgSQL function should return a value of its specified return type.
- A block of PL/pgSQL code can contain sub-blocks, which are code blocks nested within other code blocks. Sub-blocks can be useful for the organization of code within a large PL/pgSQL function.
- All sub-blocks must follow normal block structure, meaning they must start with the DECLARE keyword, followed by the BEGIN keyword and a body of statements, then end with the END keyword.

1.3.2 Data Types in PL/pgSQL (April 18)

- Variables in PL/pgSQL can be represented by any of SQL's standard data types, like an integer or char.
- Following is a brief list of commonly used data types in PL/pgSQL:

Sr. No.	Category	Data Type	Description
1.	Boolean	Boolean, Bool	A single true or false value.
2.	Binary	bit(n) bit varying(n) varbit(n)	An n-length bit string (exactly n binary bits). A variable n-length bit string (up to n binary bits).
3.	Character	character(n) char(n) character varying(n) varchar(n) text	A fixed length character string. A variable length character string of up to n characters. A variable length character string of unlimited length.
4.	Numeric	smallint, int2 integer, int, int4 bigint, int8 real, float4 double precision, float8, float numeric (p, s) decimal (p, s) money serial	A signed 2-byte integer. A signed 4-byte integer. A signed 8-byte integer, up to 18 digits in length. A 4-byte floating-point number. An 8-byte floating-point number. An exact numeric type with arbitrary precision p and scale s. A fixed precision, U.S. style currency. An auto-incrementing 4-byte integer.
5.	Date and Time	date time time with time zone timestamp (includes time zone) interval	The calendar date (day, month and year). The time of day. The time of day, including time zone information. Both the date and time. An arbitrarily specified length of time.
6.	Network	cidr (7 or 19 byes) inet (7 or 19 byes) macaddr (6 bytes)	An IP network specification. A network IP address, with optional subnet bits. A MAC address (e.g., an Ethernet cards hardware address).
7.	System	oid xid	An object (row) identifier. A transaction identifier.
8.	Geometric	line (32 bytes) lseg (32 bytes) box (32 bytes) circle (24 bytes) path (16+16n bytes) point (16 bytes) polygon (40+16n bytes)	An infinite line in a plane. A finite line segment in a plane. A rectangular box in a plane. A circle with center and radius. Open and closed geometric paths in a two-dimensional plane. Point on a plane. Polygon (similar to closed path).

- In addition to SQL data types, PL pgSQL also provides the additional RECORD data type, which is designed to allow to store row information without specifying the columns that will be supplied when data is inserted into the variable.

Comments in PL/pgSQL:

- There are two methods of commenting in PL/pgSQL. First single line comments, which represent a single line comment, and another is block comments/multiple line comment, which represent multiple line comments.

1. **Single Line Comment:**

- Single line comments begin with two dashes (--) and have no end-character, because the parser interprets any characters after the two dashes as a comment, until the end of the line.
- Following example demonstrates the use of single line comments.

Example: Using single line comments.

```
-- This will be interpreted as a single line comment.
```

2. **Multi-Line Comment or Block Comment:**

- The second type of comment is same as comment in C programming languages. Block comments begin with the forward slash and asterisk characters (/*) and end with the asterisk and forward slash characters (*/).
- Block comments can span multiple lines. While single line comments can be nested within block comments, block comments cannot be nested within other block comments.
- Following example shows the usage of a block comment.

Example: Using block comments.

```
/*
 * This is a
 * block
 * comment in PL/pgSQL.
 */
```

1.3.3 Statements and Expressions

- Like most programming languages, PL/pgSQL code is composed of statements and expressions. Most of code will be made of statements and expressions which are essential to certain types of data manipulation.
- PL/pgSQL code is composed of statements and expressions. PL/pgSQL adds a set of procedural constructs looping, exception and error handling, and conditional execution (that is, IF/THEN/ELSE).

1.3.3.1 Statements

- A statement performs an action within PL/pgSQL code, such as assignment of a value to a variable or the execution of a query.
- The organization of statements within a PL/pgSQL code block controls the order in which operations are executed within that code block.
- Declarative statements should appear in the declaration section after the DECLARE keyword, but these should only declare and/or initialize the variables that will be referenced within the code block.
- The most of statements will be placed between BEGIN keyword and before the END keyword. Every statement should end with a semicolon character (;).

- Basic types of statements in PL/pgSQL are as follows:

1. **Assignment Statement:** It assigns a value to a PL/pgSQL variable and written as:

Syntax: `variable := expression;`

Example: For assignment statement.

```
sum := sum + 100;
```

2. **Perform Statement:** This statement executes query and discards the result. Write the query in the same way as SQL SELECT command, but replace the initial keyword SELECT with PERFORM. A PERFORM statement sets FOUND true if it produces (and discards) one or more rows, false if no row is produced. This statement also used to call function and ignores its return data.

Syntax: `PERFORM query;` **OR** `PERFORM function_name(arguments);`

Example: For perform statement.

```
PERFORM * from Emp where eno := 1;
```

Example: Here, add_product is function name with two arguments pno and pname.

```
PERFORM add_product(pno,panme);
```

3. **SELECT INTO Statement:** The result of a SQL command giving a single row can be assigned to a record variable, row-type variable, or list of scalar variables. This is done by writing the base SQL command and adding an INTO clause. (Oct. 17)

Syntax: `SELECT select_expressions INTO [STRICT] target FROM ...;`

where, target can be a record variable, a row variable, or a comma-separated list of simple variables and record/row fields.

If a row or a variable list is used as target, the query's result columns must exactly match the structure of the target as to number and data types, or else a run-time error occurs. When a record variable is the target, it automatically configures itself to the row type of the query result columns. The following example shows usage of SELECT INTO statement.

Example: For SELECT INTO statement.

```
SELECT * INTO myrec FROM product WHERE pname = prod_name;
IF NOT FOUND THEN
    RAISE EXCEPTION ' PRODUCT % not found ', prod_name;
END IF;
```

4. **EXECUTE Statement:** Sometimes, we want to generate dynamic commands inside the PL/pgSQL function which involve different tables or different data types each time they are executed. There is no plan caching for commands executed via EXECUTE.

Instead, the command is prepared each time the statement is run. Thus the query string can be dynamically created within the function to perform actions on different tables and columns.

Syntax:

```
EXECUTE query-string [INTO [STRICT] target][USING expression [,...]];
```

where, query string is normally SELECT SQL statement. The optional target is a record variable, a row variable, or a comma-separated list of simple variables and record/row fields, into which the results of the command will be stored. The optional USING expressions supply values to be inserted into the command.

Example: For execute statement.

```
EXECUTE 'SELECT * FROM product  WHERE pno = ' || a INTO r;
```

5. **GET DIAGNOSTICS Statement:** There are several ways to determine the effect of a command. The first method is to use the GET DIAGNOSTICS command.

   Syntax:

   ```
   GET DIAGNOSTICS variable = item [ , ... ];
   ```

   This command allows retrieval of system status indicators. Each item is a key word identifying a state value to be assigned to the specified variable .The currently available status items are ROW_COUNT, the number of rows processed by the last SQL command sent to the SQL engine, and RESULT_OID, the OID of the last row inserted by the most recent SQL command.

Example: Example for get diagnostics statement.

```
GET DIAGNOSTICS integer_var = ROW_COUNT;
```

6. **Compound Statements:** Compound statements can contain loops such as FOR, WHILE etc. Looping structures/constructs contain statements that are executed on the evaluation of conditions.
7. **Conditional Statements:** Conditional statements perform a specified operation only when a certain defined condition is met. Hence, the logical resolution of the condition decides the flow of the program. PL/pgSQL supports IF, IF...THEN, IF...THEN...ELSE, CASE etc., conditional statements.
8. **Exception Handling Statements:** PL/pgSQL provides an exception handling mechanism to handle errors and reports for solution analysis. PL/pgSQL contains the RAISE statements to raise error or exception.
9. **RETURN Statement:** Every PL/pgSQL function must terminate with a return statement.

1.3.3.2 Expressions

- Expressions are calculations or operations that return their results as one of PostgreSQL's base data types.
- For example, an expression is x := a + b, which adds the variables a and b, then assigns the result to the variable x.
- Following example shows a simple PL/pgSQL function that assigns result of a multiplication expression to the variable x.

Example: Using expressions.

```
CREATE FUNCTION mult () RETURNS int4 AS '
DECLARE
   x int 4;
BEGIN
   x := 10 * 10;
   return x;
END;
' LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=#' SELECT mult() AS output;
```

Output:

```
------------
    100
(1 row)
```


1.3.4 Declaration

- Variables are used within PL/pgSQL code to store data of specified type. Variables in PL/pgSQL can be represented by any of SQL's standard data types, such as an INTEGER or CHAR.
- In addition to SQL data types, PL/pgSQL also provides the additional RECORD data type, which is designed to allow us to store row information without specifying the columns that will be supplied when data is inserted into the variable.
- All variables that we will be using within a code block must be declared under the DECLARE keyword. If a variable is not initialized to a default value when it is declared, its value will default to the SQL NULL type.
- A PL/pgSQL variable is a meaningful name for a memory location. A variable holds a value that can be changed through the block or function.
- A variable is always associated with a particular data type. A list of supported data types includes Boolean, char, integer, double precision, date, time and so on.

  Syntax for **Variable Declaration**:

  ```
  variable_name [CONSTANT] type [NOT NULL] [{DEFAULT|:=} expression];
  ```

- If DEFAULT clause given, it initializes the variable when the block is entered. If the DEFAULT clause is absent, then the variable is initialized to the SQL null value.
- The CONSTANT option prevents the variable from being modified to after initialization, so that its value will remain constant for the duration of the block. All variables declared as NOT NULL must have a non null default value specified otherwise error exists.

Example: For variable declaration.

```
uid  integer;
qty  numeric(5);
str  varchar(20);
emp_row   tablename%ROWTYPE;   (emp_row  emp%ROWTYPE ;)
emp_var   tablename.columnname%TYPE; (emp_var  emp.salary%TYPE;)
emp_rec   RECORD;
```

1.3.4.1 Declaring Function Parameters

- Functions can accept and return values called function parameters or arguments. Parameters passed to functions are named with the identifiers $1, $2, etc.
- Optionally, aliases can be declared for $n parameter names for increased readability. Either the alias or the numeric identifier can then be used to refer to the parameter value.
- There are two ways to create an alias. The preferred way is to give a name to the parameter in the CREATE FUNCTION command.

Example:

```
CREATE FUNCTION IT_tax(subtotal  real) RETURNS real AS'
BEGIN
    RETURN subtotal * 0.10;
END;
'LANGUAGE 'plpgsql';
```

- The another way is available before PostgreSQL 8.0, to explicitly declare an alias, using the declaration **syntax:**

```
name ALIAS FOR $n;
```

Example:

```
CREATE FUNCTION IT_tax(real) RETURNS real AS'
DECLARE
    subtotal ALIAS FOR $1;
BEGIN
    RETURN subtotal * 0.10;
END;
'LANGUAGE 'plpgsql';
```

- We can declare an alias for any variable, not just function parameters. The different name can be assigned for variables with predetermined names, such as NEW or OLD within a trigger procedure.

Example:

```
DECLARE
  prior ALIAS FOR old;
  updated ALIAS FOR new;
```

1.3.4.2 Attributes

- PL/pgSQL provides variable attributes to assist in working with database objects.
- Use attributes to assign a variable either the type of a database object, with the %TYPE attribute, or the row structure of a row with the %ROWTYPE attribute.

1. **The %TYPE Attribute:**

- The %TYPE is used to declare a variable with the type of a referenced database object.

Syntax:

```
variable_name table_name.column_name%TYPE
```

Example: Using the %TYPE attribute.

```
CREATE FUNCTION get_emp (text) RETURNS text AS '
DECLARE
  f_name ALIAS FOR $1;
  l_name emp.last_name%TYPE;
BEGIN
   SELECT INTO l_name last_name FROM emp WHERE first_name = f_name;
    return f_name || '' '' || l_name;
END;
' LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# SELECT get_emp('Kiran');
```

Output:

```
  get_emp
-------------
Kiran Gandhi
(1 row)
```


2. **The %ROWTYPE Attribute:**

- The %ROWTYPE is used to declare a PL/pgSQL row with the same structure as the row specified during declaration.
- It is similar to the RECORD data type, but a variable declared with %ROWTYPE will have the exact structure of that table row, whereas a RECORD variable is not structured and will accept a row from any table.

Example: Using the %ROWTYPE attribute.

```
CREATE FUNCTION get_emp (text) RETURNS text AS '
  DECLARE
    f_name ALIAS FOR $1;
   found_emp   emp%ROWTYPE;
  BEGIN
    SELECT INTO found_emp  * FROM emp  WHERE first_name = f_name;
      RETURN found_emp.first_name || '' '' || found_emp.last_name;
  END;
' LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# SELECT get_emp('Kiran');
```

Output:

```
  get_emp
-------------
Kiran Gandhi
(1 row)
```

1.3.4.3 Record

- Record variables are similar to ROW variables, but they have no predefined structure. Their structure depends on the structure of the actual row assigned to them during SELECT or FOR command.
- RECORD is only placeholder, not true data type.

Syntax:

```
variable_name  RECORD;
```

Example: Example for record in PL/pgSQL.

```
DECLARE
emp_rec RECORD;
```

1.3.4.4 Rename

- Variable identifiers can be renamed using the RENAME keyword.
- Renaming a variable's identifier only alters the way the variable is referenced; we will not modify the value of a variable by renaming it.
- Variables declared with %TYPE and %ROWTYPE can also renamed with the RENAME keyword.

Syntax:

```
RENAME old_identifier TO new_identifier;
```

Example:

```
RENAME uid to USER_ID;
```

1.4 CONTROLLING PROGRAM FLOW (CONDITIONAL STATEMENTS AND LOOPS)

- Like the most programming languages, PL/pgSQL also provides ways controlling flow of program execution by using conditional statements and loops.
- Control structures are probably the most important and useful part of PL/pgSQL. With PL/pgSQL's control structures, you can manipulate data in a very flexible and powerful way.

1.4.1 Conditional Statements

- A conditional statement specifies an action or set of actions that should be executed based on the result of logical condition specified within the statement. If a given condition is true, the specified action should be taken.
- PL/pgSQL has three forms of IF i.e., IF ... THEN, IF ... THEN ... ELSE, and IF ... THEN ... ELSIF ... THEN ... ELSE. Let us see in detail.

1. **IF....THEN Statement:**

- In IF....THEN statement a statement or block of statements is executed if a given condition evaluates true.

Syntax:

```
IF condition THEN
  statement;
END IF;
```

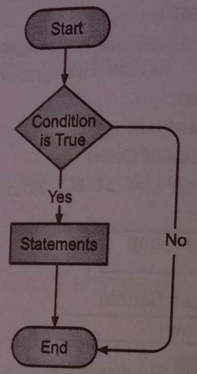

Fig. 1.2

- Fig. 1.2 shows flowchart of the simple IF statement.
- In the following example book-id and edition number is passed as input parameter and no. of books in stock is fetched for the same.

Example: Using the IF-THEN statement.

```
CREATE FUNCTION stk_amount (integer, integer) RETURNS integer AS '
  DECLARE
    id ALIAS FOR $1;
    edtn ALIAS FOR $2;
    bn TEXT;
    amount INTEGER;
  BEGIN
     SELECT INTO bn isbn FROM editions WHERE
     book_id = id AND edition = edtn;
   IF bn IS NULL THEN
     RETURN -1;
   END IF;
   SELECT INTO amount stock FROM stock WHERE isbn = bn;
   RETURN amount;
 END;
'LANGUAGE 'plpgsql';
```


Execution of Code:

```
postgres=# SELECT stk_amount(7808,1);
```

Output:

```
 stock_amount
--------------
           22
(1 row)
```

2. **IF...THEN...ELSE Statement:**

- The IF...THEN...ELSE statement allows you to execute a block of statements if a condition evaluates to true, otherwise a block of statements in else part is executed.

Syntax:

```
IF condition THEN
  statements
ELSE
   statements
END IF;
```

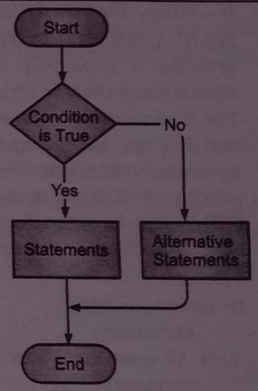

Fig. 1.3

- Fig. 1.3 shows flowchart for the IF...ELSE statement.
- In the following example first the ISBN number is retrieved, store it, and then use it to retrieve the stock number of the book in question.

Example: Using the IF...THEN..ELSE statement.

```
CREATE FUNCTION Chk_stock (integer,integer) RETURNS boolean AS '
  DECLARE
    b_id ALIAS FOR $1;
    b_edition ALIAS FOR $2;
    _isbn TEXT;
    stock_amount INTEGER;
  BEGIN
    SELECT INTO b_isbn isbn FROM editions WHERE
      book_id = b_id AND edition = b_edition;
    IF b_isbn IS NULL THEN
      RETURN FALSE;
    END IF;
     SELECT INTO stock_amount stock FROM stock WHERE isbn = b_isbn;
    IF stock_amount <= 0 THEN
      RETURN FALSE;
    ELSE
      RETURN TRUE;
    END IF;
  END;
' LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# SELECT Chk_stock(4513,2);
```

Output:

chk_stock
t

(1 row)

3. IF...THEN...ELSEIF...THEN...ELSE Statement:

- The IF...THEN...ELSEIF...THEN...ELSE statement provides a convenient method of checking several alternatives in turn.
- The IF conditions are tested successively until the first one that is true is found ,then the associated statement(s) are executed, after which control passes to the next statement after END IF. If none of the IF conditions is true, then the ELSE block (if any) is executed.

Syntax:

```
IF condition THEN
    statements
ELSE IF condition THEN
    statements
ELSE IF condition THEN
    statements
ELSE
    statements
END IF;
```

- Fig. 1.4 shows flowchart for the IF...ELSIF...ELSE statement.

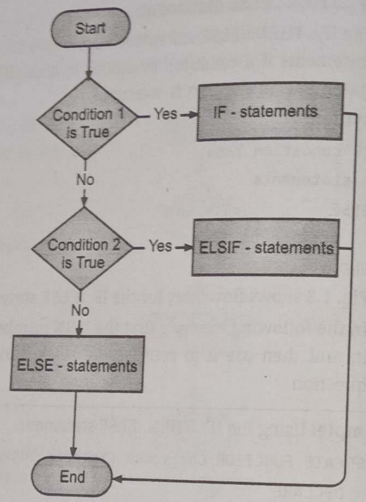

Fig. 1.4

- The following example accepts number and check whether it is positive, negative, zero or null.

Example: Using the IF...THEN...ELSEIF...THEN...ELSE statement.

```
CREATE FUNCTION chk_no(interger) RETURNS text AS'
DECLARE
    number ALIAS FOR $1;
    result text;
BEGIN
    IF number = 0 THEN
    result := 'zero';
    ELSIF number>0 THEN
    result := 'positive';
    ELSIF number<0 THEN
    result := 'negative';
ELSE
    result := 'NULL';
END IF;
    END;
'LANGUAGE 'plpgsql';
```


Execution of Code:

```
postgres=# SELECT chk_no(0);
```

Output:

chk_no
zero

4. CASE Statement:

- The CASE statement executes a block of code conditionally. The two forms of CASE are simple CASE (provides conditional execution based on equality of operands) and searched CASE (provides conditional execution based on the truth to Boolean expression.

(i) Simple CASE Statement:

- The simple form of CASE provides conditional execution based on equality of operands. The search-expression is evaluated (once) and successively compared to each expression in the WHEN clauses.
- If a match is found, then the corresponding statements are executed, and then control passes to the next statement after END CASE. If no match is found, the ELSE statements are executed; but if ELSE is not present, then a CASE_NOT_FOUND exception is raised.

Syntax:

```
CASE search-expression
WHEN expression [,expression [ ... ]] THEN
 statements
[ WHEN expression [,expression [ ... ]] THEN
statements
... ]
[ ELSE
statements ]
END CASE;
```

- The following example shows different prices depending on rate information.

Example: Using CASE statement.

```
CREATE FUNCTION get_price(p_film_id integer)RETURNS VARCHAR(50)AS'
DECLARE
    rate NUMERIC;
    price_segment VARCHAR(50);
BEGIN
    SELECT INTO rate rental_rate
    FROM film
    WHERE film_id = p_film_id;
    CASE rate
WHEN 0.99 THEN
    price_segment = 'average';
WHEN 2.99 THEN
    price_segment = 'normal';
WHEN 4.99 THEN
    price_segment = 'High';
ELSE
    price_segment = 'Unspecified';
```

```
    END CASE;
      RETURN price_segment;
  END;
  ' LANGUAGE 'plpgsql';
```

(ii) Searched CASE Statement:

- The searched CASE statement executes a condition based on the result of the Boolean expression. This is quite similar to IF...THEN...ELSIF statement.
- The evaluation of the expression continues until it finds a match and then subsequent statements are executed. Control is then transferred to the next statement after END CASE.
- **Syntax** for the searched CASE statement:

```
CASE
  WHEN boolean-expression 1 THEN
    statements
  [ WHEN boolean-expression 2 THEN
    statements
    ... ]
  [ ELSE
    statements ]
END CASE;
```

Example:

```
CREATE FUNCTION serach_case(marks integer)RETURNS varchar(50) AS'
DECLARE
   grade text;
BEGIN
   CASE
      WHEN marks >= 40 THEN
      grade := 'PASS';
      RETURN grade;
WHEN marks <= 39 AND marks > 0 THEN
      grade :. 'FAIL';
      RETURN grade;
  ELSE
     grade := 'Did not appear in exam
RETURN grade;
END CASE;
END;
  'LANGUAGE 'plpgsql';
```

1.4.2 Loops

- Loops conditions are used to perform some task repeatedly for fixed number of times or until some is valid. PL/pgSQL provides three iterative loops i.e., the basic loop, WHILE loop and the FOR loop.
- PL/pgSQL looping mechanism also known as iterative control structures, which performs repetitive tasks on logical results of certain conditions.

1.4.2.1 Simple/Basic Loop

- This is the basic unconditional loop which starts with keyword LOOP and executes the statements within its body until terminated by an EXIT or RETURN statement.

  Syntax:

```
[<<label>>]
LOOP
  statements
END LOOP;
```

- The optional label can be used by EXIT statements in nested loops to specify which level of nesting should be terminated.

```
EXIT [ label ] [ WHEN expression ];
```

- If no label is given, the innermost loop is terminated and the statement following END LOOP is executed next.
- If label is given, it must be the label of the current or some outer level of nested loop or block. Then the named loop or block is terminated and control continues with the statement after the loop's/block's corresponding END.
- If WHEN present, loop exit is occurs only if the specified condition is true, otherwise control passes to the statement after EXIT. EXIT can be used to cause early exit from all types of loops; it is not limited to use with unconditional loops.
- The following example shows usage of loop statement.

Example: Using simple/basic loop statement.

```
CREATE FUNCTION square_loop (integer) RETURNS integer AS '
  DECLARE
    num1 ALIAS FOR $1;
    result integer;
  BEGIN
result := num1;
LOOP
    result := result * result;
    EXIT WHEN result >= 50;
    END LOOP;
    RETURN result;
  END;
' LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# SELECT square_loop(3);
```

Output:

square_loop
81

(1 row)

1.4.2.2 WHILE Loop

- The WHILE statement repeats a sequence of statements till the condition evaluates to true.
- It is top tested or entry controlled loop where the condition is checked just before each entry to the loop body.

Syntax:

```
[<<label>>]
WHILE condition LOOP
   Statements
END LOOP [label];
```

- Fig. 1.5 shows flowchart for the WHILE loop statement.

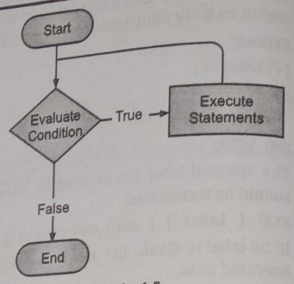

Fig. 1.5

Example: Using while loop.

```
CREATE FUNCTION Example_while_loop (integer, integer) RETURNS integer AS'
DECLARE
    low ALIAS FOR $1;
    high ALIAS FOR $2;
    result INTEGER = 0;
BEGIN
    WHILE result != high LOOP
    result := result + 1;
    END LOOP;
    RETURN result;
END;
' LANGUAGE 'plpgsql';
```

1.4.2.3 FOR Loop

- This form of FOR creates a loop that iterates over a range of integer values. The variable name is automatically defined as type integer and exists only inside the loop.
- The two expressions giving the lower and upper bound of the range are evaluated once when entering the loop. If the BY clause isn't specified the iteration step is 1, otherwise it's the value specified in the BY clause, which again is evaluated once on loop entry. If REVERSE is specified then the step value is subtracted, rather than added, after each iteration.

Syntax:

```
[<<label>>]
FOR name IN [ REVERSE ]expression ..expression [ BYexpression ] LOOP
   Statements
END LOOP [label ];
```

- Some examples of integer FOR loops:

```
FOR i IN 1..10 LOOP
-- i will take on the values 1,2,3,4,5,6,7,8,9,10 within the loop
END LOOP;
FOR i IN REVERSE 10..1 LOOP
```



```
-- i will take on the values 10,9,8,7,6,5,4,3,2,1 within the loop
END LOOP;
FOR i IN REVERSE 10..1 BY 2 LOOP
-- i will take on the values 10,8,6,4,2 within the loop
END LOOP;
```

- If the lower limit is greater than the upper limit (or less than, in the REVERSE case), then loop body will not execute at all.
- No error is raised.
- If a label is attached to the FOR loop, then the integer loop variable can be referenced with a qualified name, using that label.
- Fig. 1.6 shows flowchart for the FOR loop statement.
- The following example shows odd number generation.

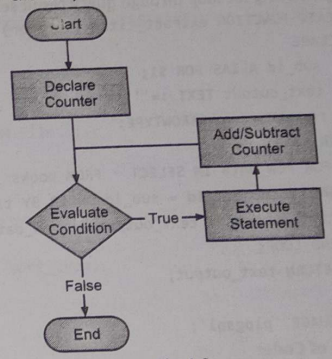

Fig. 1.6

Example: Using for loop.

```
CREATE FUNCTION Example_for_loop () RETURNS integer AS '
DECLARE
   counter interger;
BEGIN
  FOR counter IN 1..6 BY 2 LOOP
  RAISE NOTICE 'Counter: %', counter;
  END LOOP;
END;
  return counter;
'LANGUAGE 'plpgsql';
```

Output:

```
NOTICE:  Counter 1
NOTICE:  Counter 3
NOTICE:  Counter 5
```

1.4.2.4 Looping through Query Results

- Using a different type of FOR loop, we can iterate through the results of a query and manipulate that data accordingly.

Syntax:

```
[<<label>>]
FOR rec_1 IN query LOOP
   Statements
END LOOP [ label ];
```

- The rec_1 is a record variable, row variable, or comma-separated list of scalar variables. The rec_1 is successively assigned each row resulting from the query and the loop body is executed for each row. Here, is an example which fetches book title for given subject id.

Example: Using for loop through query result set.

```
CREATE FUNCTION extract_title (integer) RETURNS text AS '
DECLARE
    sub_id ALIAS FOR $1;
    text_output TEXT :=''\n'';
    row_data books%ROWTYPE;
BEGIN
    FOR row_data IN SELECT * FROM books
    WHERE subject_id = sub_id ORDER BY title  LOOP
    text_output := text_output || row_data.title || ''\n'';
    END LOOP;
    RETURN text_output;
END;
' LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# SELECT extract_title(2);
```

Output:

extract_title
Wings of Fire
Two states

(1 row)

- If the loop is terminated by an EXIT statement, the last assigned row value is still accessible after the loop.
- The query used in this type of FOR statement can be any SQL command that returns rows to the caller. SELECT is the most common case, but we can also use INSERT, UPDATE, or DELETE with a RETURNING clause.

1.4.2.5 FOR-IN-EXECUTE Statement

- The FOR-IN-EXECUTE statement is another way to iterate over rows of record set.

Syntax:

```
[<<label>>]
FOR rec_1  IN EXECUTE text_expression [ USING expression [, ... ] ] LOOP
    Statements
END LOOP [label];
```

Here, the source query is specified as a string expression, which is evaluated and re-planned on each entry to the FOR loop. This allows the programmer to choose the speed of a preplanned query or the flexibility of a dynamic query.

- As with EXECUTE, parameter values can be inserted into the dynamic command via USING. The following example demonstrates how to use the FOR loop statement to loop through a dynamic query.

- It accepts two parameters. The first parameter sort_type: 1 means sort the query result by title, 2 means sort the result by release year. The other parameter n is the number of rows to query from the film table. Notice that it will be used in the USING clause.

Example: Using FOR-IN-EXECUTE statement.

```
CREATE FUNCTION for_loop_dyn_query (sort_type INTEGER, n INTEGER)RETURNS void AS'
DECLARE
    rec RECORD;
    query text;
BEGIN
    query := 'SELECT title, release_year FROM film ';
    IF sort_type = 1 THEN
    query := query || 'ORDER BY title';
    ELSIF sort_type = 2 THEN
    query := query || 'ORDER BY release_year';
    ELSE
    RAISE EXCEPTION 'Invalid sort type %s', sort_type;
END IF;
    query := query || ' LIMIT $1';
    FOR rec IN EXECUTE query USING n
    LOOP
    RAISE NOTICE '% - %', rec.release_year, rec.title;
    END LOOP;
END;
'LANGUAGE ' plpgsql';
```

1.5 STORED PROCEDURES (April 18)

- PostgreSQL allows to extend the database functionality with user-defined functions by using various procedural languages, which are often referred to as stored procedures.
- With stored procedures we can create our own custom functions and reuse them in applications or as part of other database's workflow.
- A stored procedure is nothing but a series of declarative SQL statements which can be stored in the database catalogue.
- A procedure can be thought of as a function or a method. They can be invoked through triggers, other procedures, or applications on Java, PHP etc.
- All the statements of a block are passed to SQL engine all at once which increases processing speed and decreases the traffic.
- A stored procedure provides an important layer of security between the user interface and the database. It supports security through data access controls because end users may enter or change data, but do not write procedures.
- A stored procedure preserves data integrity because information is entered in a consistent manner. It improves productivity because statements in a stored procedure only must be written once.

Advantages of Stored Procedures:

1. They result in performance improvement of the application. If a procedure is being called frequently in an application in a single connection, then the compiled version of the procedure is delivered.

2. They reduce the traffic between the database and the application, since the lengthy statements are already fed into the database and need not be sent again and again via the application.
3. They add to code reusability, similar to how functions and methods work in other languages such as C/C++ and Java.
4. It supports security through data access controls and preserves data integrity.

Disadvantages of Stored Procedures:

1. Stored procedures can cause a lot of memory usage. The database administrator should decide an upper bound as to how many stored procedures are feasible for a particular application.
2. MySQL does not provide the functionality of debugging the stored procedures.

- PostgreSQL old versions does not support stored procedures in the sense that a database such as Oracle does, but it does support stored functions which will be discussed in next Section 1.6.
- But PostgreSQL version 11 will allow to write procedure just like other databases. Procedure is almost the same as FUNCTION without a return value.
- Procedure is created with the CREATE PROCEDURE statement in PostgreSQL 11. Unlike the CREATE FUNCTION statement, there are no RETURNS clause, ROWS clause etc.

Syntax:

```
CREATE [ OR REPLACE ] PROCEDURE
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...] ] )
  { LANGUAGE lang_name
    | TRANSFORM { FOR TYPE type_name } [, ... ]
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
  } ...
```

Example:

```
CREATE PROCEDURE procedure1(INOUT p1 TEXT)
    AS $$
BEGIN
    RAISE NOTICE 'Procedure Parameter: %', p1;
END;
$$
'LANGUAGE 'plpgsql';
```

- To execute PROCEDURE in PostgreSQL, use the CALL statement instead of SELECT statement.

```
postgres=# CALL procedure1 (' CREATE PROCEDURE functionality supported in PostgreSQL 11! ');
NOTICE: Procedure Parameter: CREATE PROCEDURE functionality supported in PostgreSQL 11!
```

p1
CREATE PROCEDURE functionality supported in PostgreSQL 11!

(1 row)

- We can also specify parameter name in the CALL statement. This is another way to execute the PROCEDURE.

```
postgres=# CALL procedure1 (p1=>'CREATE PROCEDURE functionality supported in PostgreSQL 11!');
NOTICE: Procedure Parameter: CREATE PROCEDURE functionality supported in PostgreSQL 11!
```

p1
CREATE PROCEDURE functionality supported in PostgreSQL 11!

(1 row)

- We can check the definition of created PROCEDURE from psql command i.e '\df'. The psql command '\df' is also used to display the definition of created FUNCTION.
- The PROCEDURE shows the Type column as "proc" and if it is FUNCTION then the Type column changed to "func".
- In the below list of functions, we have created one PROCEDURE so the Type column changed to "proc".

```
postgres=# \df
                    List of functions
```

Schema	Name	Result data type	Argument data types	Type
public	procedure1		INOUT p1 text	proc

(1 row)

1.6 STORED FUNCTIONS (April 15, 18; Oct. 17)

- Functions are the heart of most programming languages. PostgreSQL functions also known as stored procedures.
- Function allow to carry out operations that would normally take several queries and round trips in a single function within the database.
- Functions allow database reuse as other applications can interact directly with the stored procedures instead of a middle-tier or duplicating code.
- In PostgreSQL, functions are stored directly in the database and are called by the backend. When you insert a PL/pgSQL function into the database, the source code is converted into bytecode that can be executed efficiently by the PL/pgSQL bytecode interpreter.
- Functions can be created in a language PL/pgSQL using following **syntax:**

```
CREATE [OR REPLACE] FUNCTION function_name (arguments)
RETURNS return_datatype AS'
DECLARE
   declaration;
   [...]
BEGIN
   < function_body >
   [...]
   RETURN { variable_name | value }
END;
'LANGUAGE 'plpgsql';
```

where,

- function-name specifies the name of the function.
- [OR REPLACE] option allows modifying an existing function.
- Argument can be of types like IN, OUT, INOUT and VARIADIC as explained below:
  - **(i) IN:** By default, the parameter's type of any parameter in PostgreSQL is IN parameter. We can pass the IN parameters to the function but you cannot get them back as a part of result.
  - **(ii) OUT:** The OUT parameter is a part of the function arguments list and you can get the result back as the part of the result. To define OUT parameters, we use OUT keyword.
  - **(iii) INOUT:** The INOUT parameter is the combination IN and OUT parameters. It means that the caller can pass the value to the function. The function then changes the argument and passes the value back as a part of the result.

(iv) **VARIADIC:** A PostgreSQL function can accept a variable numbers of arguments with one condition that all arguments have the same data type. The arguments are passed to the function as an array.

- The function must contain a RETURN statement.
- RETURN clause specifies that data type we 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. The return value of function cannot left undefined.

Syntax:

```
RETURN expression;
```

- Function-body contains the executable part.
- The AS keyword is used for creating a standalone function.
- PL/pgSQL is the name of the language that the function is implemented in. For backward compatibility, the name can be enclosed by single quotes.

```
CREATE OR REPLACE FUNCTION hi_lo(IN a NUMERIC, IN b NUMERIC, IN  c NUMERIC, OUT hi
                                                        NUMERIC, OUT lo NUMERIC) AS'
BEGIN
  hi := GREATEST(a,b,c);
  lo := LEAST(a,b,c);
END;
'LANGUAGE 'plpgsql';
```

1.6.1 Calling a Function

(April 16)

- The normal syntax to call another PL/pgSQL function from within PL/pgSQL is to either reference the function in a SQL SELECT statement, or during the assignment of a variable.
- For examples:

```
1. SELECT function_name(arguments);
2. variable_identifier := function_name(arguments);
```

- The use of assignments and SELECT statements to execute functions is standard in PL/pgSQL because all functions in a PostgreSQL database must return a value of some type.
- We can also use the PERFORM keyword to call a function and ignore its return data.

```
PERFORM function_identifier(arguments);
```

Example: Calling a function.

```
postgres=#SELECT hi_lo(10,20,30);
```

Output:

```
 hi_lo
-------
 30,10
```

1.6.2 Dropping a Function

- Functions in PL/pgSQL dropped from a database with DROP FUNCTION.
- We can remove a function by using following **syntax**:

```
DROP FUNCTION NAME ( [ type [, ...] ] ) [ CASCADE | RESTRICT ]
```


where,

Type: The data type(s) of the function's arguments.

CASCADE: Automatically drop objects that depend on the function (such as operators or triggers).

RESTRICT: Refuse to drop the function if any objects depend on it.

Example: Dropping function.

```
postgres=#  DROP function hi_lo();
```

Difference between Stored Function and Stored Procedure:

Sr. No.	Stored Function	Stored Procedure
1.	Functions are not used for any kind of updating because functions are not allowed to change anything in the database, is only used for plain queries.	Generally, the purpose to use procedures is to modify the database data where return value is not required like delete, update, drop etc.
2.	Functions should contain at least one parameter and should return a value.	Stored procedures neither contain any parameter nor return any value.
3.	Functions are created with the CREATE FUNCTION command.	Procedures are created with the CREATE PROCEDURE command.
4.	We cannot call stored procedures from functions.	We can call functions from stored procedures.
5.	Functions can have only input parameters for it.	Procedures can have input or output parameters.
6.	Transactions are not allowed with functions.	Transactions can be executed from stored procedures; also it can use COMMIT and ROLLBACK inside procedures.

1.7 HANDLING ERRORS AND EXCEPTIONS

(April 17, 18)

- Error and exception handling is important thing in any programming language. Error is an illegal operation performed by the user which results in abnormal working of the program.
- An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. The RAISE statement is used to raise errors and exceptions during a PL/pgSQL function's operation.
- A RAISE statement sends specified information to the PostgreSQL elog mechanism which is the standard PostgreSQL error logging utility, which typically logs data.

(April 17)

Syntax:

```
RAISE level  ' 'format string' ' [, identifier [...] ];
```

- Followed the RAISE statement is the level option that specifies the error severity. There are following levels in PostgreSQL:

```
DEBUG
LOG
NOTICE
INFO
WARNING
EXCEPTION
```

1. **DEBUG:** DEBUG level statements send the specified text as a DEBUG: message to the PostgreSQL log and the client program if the client is connected to a database cluster running in debug mode. DEBUG level RAISE statements will be ignored by a database running in production mode.
2. **NOTICE:** This level statement sends the specified text as a NOTICE: message to the PostgreSQL log and the client program in any PostgreSQL operation mode.
3. **EXCEPTION:** This statement sends the specified text as an ERROR: message to the client program and the PostgreSQL database log. The EXCEPTION level also causes the current transaction to be aborted. If we don't specify the level, by default, the RAISE statement will use EXCEPTION level that raises an error and stops the current transaction.

- The format is a string that specifies the message. The format uses percentage (%) placeholders that will be substituted by the next arguments. The number of placeholders must match the number of arguments; otherwise PostgreSQL will report the error message.
- In the following example the first RAISE statement raises a debug level message. The second and third RAISE statements send a notice to the user. Notice the use of the percent sign (%) in the third RAISE statement to mark the location in the string at which the value of an integer is to be inserted.
- Finally, the fourth RAISE statement displays an error and throws an exception, causing the function to end and the transaction to be aborted.

Example: Using the RAISE statement.

```
CREATE FUNCTION raise_test_example () RETURNS integer AS'
DECLARE
    cnt INTEGER = 1;
BEGIN
    RAISE DEBUG ''The raise_test() function began.'';
    cnt = cnt + 1;
    RAISE NOTICE ''Variable cnt was changed.'';
    RAISE NOTICE ''Variable cnt's value is now %.'', cnt;
    RAISE EXCEPTION ''Variable % changed.  Transaction aborted.'', cnt;
    RETURN 1;
END;
'LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# SELECT raise_test_example();
```

Output:

```
NOTICE: Variable cnt was changed.
NOTICE: Variable cnt's value is now 2.
ERROR:  Variable 2 changed. Transaction aborted.
```

- We can attach additional information to the error report by writing USING followed by option = expression items. Each expression can be any string-valued expression. The allowed option key words are:

(i) MESSAGE: Sets the error message text. This option can't be used in the form of RAISE that includes a format string before USING.

(ii) DETAIL: Supplies an error detail message.

(iii) HINT: Supplies a hint message.

(iv) ERRCODE: Specifies the error code (SQLSTATE) to report, either by condition name or directly as a five-character SQLSTATE code. All messages emitted by the PostgreSQL server are assigned five-character error codes that follow the SQL standard's conventions for "SQLSTATE" codes. Applications that need to know which error condition has occurred should usually test the error code, rather than looking at the textual error message.

- The following example will abort the transaction with the given error message and hint:

```
RAISE EXCEPTION 'Nonexistent ID --> %', user_id USING HINT='Please check your user ID';
```

- These two examples show equivalent ways of setting the SQLSTATE:

```
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = 'unique_violation';
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = '23505';
```

- The following RAISE syntax in which the main argument is the condition name or SQLSTATE to be reported:

```
RAISE division_by_zero;
RAISE SQLSTATE '22012';
```

Trapping Errors:

- By default, any error occurring in a PL/pgSQL function aborts execution of the function, and indeed of the surrounding transaction as well.
- We can trap errors and recover from them by using a BEGIN block with an EXCEPTION clause.
- The syntax is an extension of the normal **syntax** for a BEGIN block is given below:

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
EXCEPTION
    WHEN condition [ OR condition ... ] THEN
        handler_statements
    [ WHEN condition [ OR condition ... ] THEN
        handler_statements
    ... ]
END;
```

- If no error occurs, this form of block simply executes all the statements, and then control passes to the next statement after END. But if an error occurs within the statements, further processing of the statements is abandoned, and control passes to the EXCEPTION list.
- The list is searched for the first condition matching the error that occurred. If a match is found, the corresponding handler_statements are executed, and then control passes to the next statement after END. If no match is found, it aborts processing of the function. The condition names can be like for example:

```
WHEN division_by_zero THEN ...
WHEN no_data_found THEN..
WHEN SQLSTATE '22012' THEN ...
```

- When an error is caught by an EXCEPTION clause, the local variables of the PL/pgSQL function remain as they were when the error occurred, but all changes to persistent database state within the block are rolled back.

1.8 CURSORS

(April 16, 17, 18)

- A PL/pgSQL cursor allows to encapsulate a query and process each individual row at a time. We use cursors when we want to divide a large result set into parts and process each part individually. If we process it at once, we may have a memory overflow error.
- In addition, we can develop a function that returns a reference to a cursor. This is an efficient way to return a large result set from a function. The caller of the function can process the result set based on the cursor reference.
- Cursors can be of two types implicit cursors and explicit cursors.
  1. **Implicit cursors** are declared and managed by PL/pgSQL for all DML and PL/pgSQL SELECT statements.
  2. **Explicit cursors** are declared and managed by the programmer. We will see how to manage explicit cursor operations.
- Explicit cursor operations are as follows:
  1. First, declare a cursor.
  2. Next, open the cursor.
  3. Then, fetch rows from the result set into a target.
  4. After that, check if there are more rows left to fetch. If yes, go to step 3, otherwise go to step 5.
  5. Finally, close the cursor.

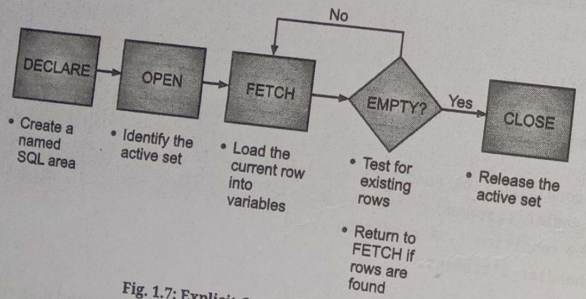

Fig. 1.7: Explicit Cursor Operations

- We will examine each step in more detail in the following sections.

1.8.1 Declaring Cursor Variables

(April 16)

- To access cursors in PL/pgSQL, we have to go through cursor variables. Cursor variables are always of the special data type REFCURSOR.
- There are two ways to create a cursor variable:
  1. First we need to declare a cursor variable at the declaration section of a block. PostgreSQL provides us with a special type called REFCURSOR to declare a cursor variable.

Syntax:

```
DECLARE
    my_cursor REFCURSOR;
```


  2. Another way to declare a cursor that bounds to a query is using the following **syntax**:

```
cursor_name [ [NO] SCROLL ] CURSOR [( name datatype, name data type, ...)] FOR query;
```

where, cursor_name specify a variable name for the cursor.

- Next, we specify whether the cursor can be scrolled backward using the SCROLL. If wer use NO SCROLL, the cursor cannot be scrolled backward.
- Next the CURSOR keyword followed by a list of comma-separated arguments (name datatype) that defines parameters for the query. These arguments will be substituted by values when the cursor is opened.
- After that, we specify a query following the FOR keyword. We can use any valid SELECT statement here.

Example: Using declarations for cursors.

```
DECLARE
    curs1 refcursor;
    curs2 CURSOR FOR SELECT * FROM tenk1;
curs3 CURSOR (key integer) IS SELECT * FROM tenk1 WHERE unique1 = key;
```

- In above example all three of these variables have the data type REFCURSOR.
  - The first declaration may be used with any query. The variable curs1 is said to be unbound since it is not bound to any particular query.
  - The second cursor is said to be bound cursor as specified query already bound to it.
  - The last has a parameterized query bound to it. (key will be replaced by an integer parameter value when the cursor is opened.) so it is called parameterized cursor.

1.8.2 Opening Cursor

- Cursors must be opened before they can be used to fetch query rows. PostgreSQL provides syntax for opening unbound and bound cursors.

1.8.2.1 Opening Unbound Cursor

- To open an unbound cursor there are two forms.
1. The first **form/syntax** is as follows:

```
OPEN unbound_cursor_variable [ [ NO ] SCROLL ] FOR query;
```

- Because unbound cursor variable is not bounded to any query when we declared it, we have to specify the query when we open it.
- For example,

```
OPEN my_cursor FOR SELECT * FROM city WHERE counter = p_country
```

2. PostgreSQL allows us to open a cursor and bound it to a dynamic query.

Here, is the **syntax:**

```
OPEN unbound_cursor_variable[ [ NO ] SCROLL ]
FOR EXECUTE query_string [USING expression [, ... ] ];
```

- In the following example, a dynamic query can be built that sorts rows based on a year parameter, and open the cursor that executes the dynamic query.

```
query := 'SELECT * FROM city ORDER BY $1';
OPEN cur_city FOR EXECUTE query USING year;
```

1.8.2.2 Opening Bound Cursor

- A bound cursor already bounds to a query when we declared it, so when we open it, we just need to pass the arguments to the query if necessary.

```
OPEN cursor_variable[(name:=value,name:=value,...)];
```

- In the following example, we open bound cursors cur2 and cur3 that we declared above:

```
OPEN curs2;
OPEN curs3(42);
```

1.8.3 Fetching Rows

- After opening a cursor, we can manipulate it using FETCH, MOVE, UPDATE, or DELETE statement. These manipulations need not occur in the same function that opened the cursor to begin with. We can return a refcursor value out of a function and let the caller operate on the cursor.
- Internally, a refcursor value is simply the string name of a so-called portal containing the active query for the cursor. This name can be passed around, assigned to other refcursor variables and so on, without disturbing the portal.
- All portals are implicitly closed at transaction end. Therefore, a refcursor value is usable to reference an open cursor only until the end of the transaction.

1. **Fetching the Next Row:**

- The FETCH statement retrives the next row from the cursor and assign it a target_variable, which could be a record, a row variable, or a comma-separated list of variables. If no more rows found, the target_variable is set to NULL.

Syntax:

```
FETCH [direction {FROM|IN}] cursor_variable INTO target_variable;
```

- By default, a cursor gets the next row if you don't specify the direction explicitly. The following is the valid directions for the cursor:

```
NEXT
LAST
PRIOR
FIRST
ABSOLUTE count
RELATIVE count
FORWARD
BACKWARD
```

- Note that FORWARD and BACKWARD directions are only for cursors declared with SCROLLoption.

For example,

```
FETCH cur2 INTO row1;
FETCH LAST FROM cur3 INTO title, release_year;
```

2. **Moving the Cursor:**

- If we want to move the cursor only without retrieving any row, we can use the MOVE statement.

```
MOVE [ direction { FROM | IN } ] cursor_variable;
```

- The direction accepts the same value as the FETCH statement.

For example:

```
MOVE cur2;
MOVE LAST FROM cur3;
MOVE RELATIVE -1 FROM cur2;
MOVE FORWARD 3 FROM cur3;
```

3. **Deleting or Updating Row:**

- Once, a cursor is positioned, we can delete or update row identifying by the cursor using DELETE WHERE CURRENT OF or UPDATE WHERE CURRENT OF statement as follows.

Syntax:

```
UPDATE table_name
SET column = value, ...
WHERE CURRENT OF cursor_variable;
DELETE FROM table_name
WHERE CURRENT OF cursor_variable;
```

For example,

```
UPDATE book SET release_year = p_year
WHERE CURRENT OF cur_book;
```

1.8.4 Closing Cursors

- To close an opening cursor, we use CLOSE statement. The **syntax** is as follows:

```
CLOSE cursor_variable;
```

For example,

```
CLOSE cur2;
```

- The CLOSE statement releases resources or frees up cursor variable.

Example: The following example shows steps of cursor.

```
CREATE OR REPLACE FUNCTION print_data() RETURNS integer AS '
DECLARE
-- declaring cursor
   prod_cursor CURSOR FOR SELECT * FROM product;
   rec product%ROWTYPE;
BEGIN
-- opening cursor
   OPEN prod_cursor;
   loop
--fetch the table row inside the loop
FETCH prod_cursor INTO rec;
--check if there is no record
 --exit from loop when record not found
   if not found then
   exit ;
   end if;
raise notice ''%'',rec.pname;
end loop;
CLOSE prod_cursor;
RETURN 1;
```

```
END;
'LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# \i c1.sql
CREATE FUNCTION
postgres=# SELECT print_data();
```

Output:

```
NOTICE:  pen
NOTICE:  box
 print_data
------------
          1
(1 row)
```

1.8.5 Returning Cursors

- PL/pgSQL functions can return cursors to the caller. This is useful to return multiple rows or columns, especially with very large result sets.
- To do this, the function opens the cursor and returns the cursor name to the caller. The caller can then fetch rows from the cursor.
- The cursor can be closed by the caller, or it will be closed automatically when the transaction closes.

Example: Example shows one way a cursor name can be supplied by the caller.

```
CREATE TABLE testdata (col text);
   INSERT INTO testdata VALUES ('123');
   CREATE FUNCTION reffunc1(refcursor) RETURNS refcursor AS'
BEGIN
   OPEN $1 FOR SELECT col FROM testdata;
   RETURN $1;
END;
'LANGUAGE plpgsql;
BEGIN;
   SELECT reffunc1('Cur1');
   FETCH ALL IN Cur1;
COMMIT;
```

1.8.6 Looping through a Cursor's Result

- There is a variant of the FOR statement that allows iterating through the rows returned by a cursor.
- The **syntax** is:

```
[<<label>>]
FOR recordvar IN bound_cursorvar [(argument_values)] LOOP
   Statements
END LOOP [label];
```

- The cursor variable must have been bound to some query when it was declared, and it cannot be open. The FOR statement automatically opens the cursor.

- When loop exits, it closes the cursor again. A list of actual argument value expressions must appear if and only if the cursor was declared to take arguments.
- These values will be substituted in the query, in just the same way as during an OPEN. The variable recordvar is automatically defined as type record and exists only inside the loop.
- Each row returned by the cursor is successively assigned to this record variable and the loop body is executed.

Example:

```
CREATE OR REPLACE FUNCTION print_data1() RETURNS integer AS '
DECLARE
   prod_cursor CURSOR FOR SELECT * FROM product;
   REC product%ROWTYPE;
 BEGIN
   FOR rec in prod_cursor loop
   raise notice ''%'',rec.pname;
END LOOP;
RETURN 1;
END;
'LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# \i c2.sql
CREATE FUNCTION
postgres=# SELECT print_data1();
```

Output:

```
NOTICE:  pen
NOTICE:  box
 print_data1
-------------
           1
(1 row)
```

Advantages using Cursors:

1. Cursors are a useful tool if you don't want to always execute the query and wait for the full result set before returning from a function.
2. Currently, cursors are also the only way to return multiple result sets out of a user-defined function.

Disadvantages using Cursors:

1. Cursors mainly work to pass data between functions on the server, and you are still limited to one record set per call returned to the database client.
2. Cursors are sometimes confusing to use, and bound and unbound cursors are not always interchangeable.

1.9 TRIGGER (April 15, 16, 17, 18; Oct. 17)

- A trigger is a set of SQL statements stored in the database catalog. Triggers define operations that are performed when a specific event occurs within the database.
- A PL/pgSQL trigger function can be referenced by a trigger as the operation to be performed when the trigger's event occurs.

- A trigger is a set of actions that are run automatically when a specified change operation (SQL INSERT, UPDATE, DELETE or TRUNCATE statement) is performed on a specified table or view.
- **Definition:** A trigger is, "a statement that is executed automatically by the system as a side effect of a modification to the database". (April 16)
- To design a trigger mechanism, we must:
  1. Specify the conditions under which the trigger is to be executed.
  2. Specify the actions to be taken when the trigger executes.
- A trigger is a special user-defined function that binds to a table. To create a new trigger, we must define a trigger function first, and then bind this trigger function to a table.
- The difference between a trigger and a user-defined function is that a trigger is automatically invoked when an event occurs whereas a stored procedure must be called explicitly.
- Triggering events can be insert, delete or update. Trigger can be set to fire BEFORE an event occur or AFTER an event occur or even we can bypass the event by using the INSTEAD OF command.
- If we configured a trigger to fire BEFORE an INSERT or UPDATE, we will have the additional benefit of modifying the new data to be inserted or updated and even skipping the operation itself.
- There are two types of trigger i.e., Row Level Trigger and Statement Level trigger. A **row level trigger** is fired for each affected row. A **statement level trigger** is fired only once for a statement.
- For example, consider the following statement:

```
UPDATE account_current SET balance = balance + 100 WHERE balance > 100000;
```

- Suppose executing this statement may affect 20 rows. If a row level trigger is defined for the table, the trigger will be fired for each 20 updated rows. But if it was a statement level trigger, it would have fired only once.

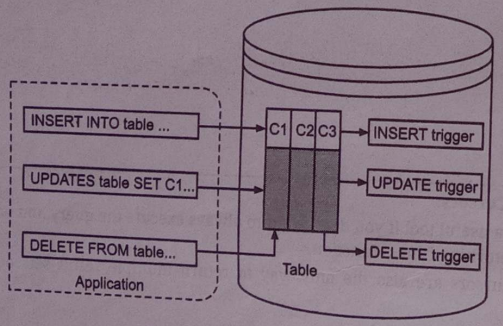

Fig. 1.8: Concept of Trigger

- It is important to understand trigger's advantages and disadvantages so that you can use it appropriately.

Advantages of using Triggers:

1. Triggers provide an alternative way to check the integrity of data.
2. Triggers can catch errors in business logic in the database layer.
3. SQL triggers provide an alternative way to run scheduled tasks. By using SQL triggers, you don't have to wait to run the scheduled tasks because the triggers are invoked automatically before or after a change is made to the data in the tables.
4. Triggers are very useful to audit the changes of data in tables.

Disadvantages of using Triggers:

1. SQL triggers only can provide an extended validation and they cannot replace all the validations. Some simple validations have to be done in the application layer. For example, we can validate user's inputs in the client side by using JavaScript or in the server side using server-side scripting languages such as JSP, PHP, ASP.NET, Perl, etc.
2. SQL triggers are invoked and executed invisible from the client applications; therefore, it is difficult to figure out what happen in the database layer.
3. Triggers may increase the overhead of the database server.

1.9.1 Creating Trigger (April 16, 18)

- To create a new trigger in PostgreSQL, we need to:
  1. Create a trigger function using CREATE FUNCTION statement.
  2. Bind this trigger function to a table using CREATE TRIGGER statement.
- A trigger procedure is created with the CREATE FUNCTION command, declaring it as a function with no arguments and a return type of trigger.
- The function must be declared with no arguments even if it expects to receive arguments specified in CREATE TRIGGER trigger arguments are passed via TG_ARGV, as described below.
- When a PL/pgSQL function is called as a trigger, several special variables are created automatically in the top-level block. They are:
  1. **NEW:** Data type RECORD; variable holding the new database row for INSERT/UPDATE operations in row-level triggers. This variable is NULL in statement-level triggers and for DELETE operations.
  2. **OLD:** Data type RECORD; variable holding the old database row for UPDATE/DELETE operations in row-level triggers. This variable is NULL in statement-level triggers and for INSERT operations.
  3. **TG_NAME:** Data type name; variable that contains the name of the trigger actually fired.
  4. **TG_WHEN:** Data type text; a string of BEFORE, AFTER, or INSTEAD OF, depending on the trigger's definition.
  5. **TG_LEVEL:** Data type text; a string of either ROW or STATEMENT depending on the trigger's definition.
  6. **TG_OP:** Data type text; a string of INSERT, UPDATE, DELETE, or TRUNCATE telling for which operation the trigger was fired.
  7. **TG_RELID:** Data type oid; the object ID of the table that caused the trigger invocation.
  8. **TG_TABLE_NAME:** Data type name; the name of the table that caused the trigger invocation.
  9. **TG_TABLE_SCHEMA:** Data type name; the name of the schema of the table that caused the trigger invocation.
  10. **TG_NARGS:** Data type integer; the number of arguments given to the trigger procedure in the CREATE TRIGGER statement.
  11. **TG_ARGV[]:** Data type array of text; the arguments from the CREATE TRIGGER statement. The index counts from 0. Invalid indexes (less than 0 or greater than or equal to tg_nargs) result in a null value.

Creating the Trigger Function:

- A trigger function is similar to an ordinary function, except that it does not take any arguments and has return value type trigger as follows:

```
CREATE FUNCTION trigger_function() RETURN trigger AS'
```

Creating the Trigger:

- To create a new trigger, we use the CREATE TRIGGER statement.

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

Where,

CREATE TRIGGER trigger_name clause creates a trigger with the given name.

{BEFORE | AFTER | INSTEAD OF} clause indicates at what time the trigger should get fired. i.e. for example, before or after updating a table. INSTEAD OF is used to create a trigger on a view. Before and after cannot be used to create a trigger on a view.

Event clause determines the triggering event. Event can be {INSERT [OR] | UPDATE [OR] | DELETE}. More than one triggering events can be used together separated by OR keyword. The trigger gets fired at all the specified triggering event.

[ON table_name] clause specifies table name on which event occurs.

[FOR EACH ROW] clause is used to determine whether a trigger must fire when each row gets affected (i.e. a Row Level Trigger) or just once when the entire sql statement is executed (i.e. statement level Trigger).

EXECUTE PROCEDURE specifies trigger function.

Example: A trigger to ensure that the qty entered for a product is never < 0 .

```
postgres=# SELECT * FROM product;
```

pid	pname	qty	rate
1	pen	100	10
2	box	20	100

(2 rows)

```
CREATE OR REPLACE FUNCTION chk_product_data()  RETURNS trigger AS '
DECLARE
BEGIN
IF  NEW.qty < 0  THEN
   RAISE EXCEPTION '' QTY is less than zero '';
END IF;
END;
'LANGUAGE  'plpgsql';
```

Execution of Code:

```
postgres=# \i f1.sql
CREATE FUNCTION
CREATE TRIGGER check_qty
```



```
   BEFORE INSERT ON product
   FOR EACH ROW
   EXECUTE PROCEDURE chk_product_data();
```

Execution of Code:

```
postgres=# \i t1.sql
CREATE TRIGGER
```

Output:

```
postgres=# insert into product values(3,'paper',-20,100);
ERROR: QTY is less than zero
postgres=#
```

1.9.2 Listing Triggers

- We can list down all the triggers in the current database from pg_trigger table as follows:

```
testtown=# SELECT * FROM pg_trigger;
```

- The above given PostgreSQL statement will list down all triggers.

1.9.3 Dropping Triggers

- To delete or destroy or drop a trigger, use a DROP TRIGGER statement.
- To execute this command, the current user must be the owner of the table for which the trigger is defined.

Syntax:

```
DROP TRIGGER [ IF EXISTS ] name ON table_name [ CASCADE | RESTRICT ]
```

Parameters:

Name	Description
IF EXISTS	Do not throw an error if the trigger does not exist. A notice is issued in this case.
Name	The name of the trigger to remove.
table_name	The name of the table for which the trigger is defined.
CASCADE	Automatically drop objects that depend on the trigger.
RESTRICT	Refuse to drop the trigger if any objects depend on it. This is the default.

- For example,

```
postgres=# DROP trigger chk_qty on product;
```

PRACTICE QUESTIONS

Q.I Multiple Choice Questions:

1. Which language is a loadable procedural language for the PostgreSQL?
   (a) SQL (b) MySQL
   (c) PL/pgSQL (d) All of These
2. Which database is the collection of normalized or structured relations with distinct relation names?
   (a) Relational (b) Network
   (c) Object-relational (d) None of these

3. A _____ is a software program you use to create, maintain, modify, and manipulate a relational database.
   (a) DBMS (b) RDBMS
   (c) ORDBMS (d) None of these
4. Which language is a general-purpose ORDBMS?
   (a) PostgreSQL (b) SQL
   (c) IBM DB2 (d) None of these
5. PL/SQL is Oracle's procedural language extension to which relational database language?
   (a) PostgreSQL (b) SQL
   (c) IBM DB2 (d) None of these
6. PostgreSQL functions also known as _____.
   (a) Trigger (b) Cursor
   (c) Stored procedure (d) None of these
7. Which procedural languages available in the PostgreSQL distribution?
   (a) PL/Perl (b) PL/Python
   (c) PL/pgSQL (d) All of these
8. _____ is the SQL command which adds procedural languages to the currently connected database.
   (a) CREATE LANGUAGE (b) CREATE FUNCTION
   (c) CREATE CURSOR (d) All of these
9. Which attribute is used to declare a PL/pgSQL row with the same structure as the row specified during declaration?
   (a) %TYPE (b) %TYPE and %ROWTYPE
   (c) %ROWTYPE (d) None of these
10. Loops in PL/pgSQL includes _____.
    (a) BASIC LOOP (b) FOR
    (c) WHILE (d) All of these
11. Which is a special user-defined function that binds to a table?
    (a) Cursor (b) Trigger
    (c) Function (d) None of these
12. Which statement used by PL/pgSQL for handling error and exceptions?
    (a) Try (b) RAISE
    (c) Throw (d) Catch
13. Types of Cursor includes _____.
    (a) Implicit (b) Explicit
    (c) Both (a) and (b) (d) None of these
14. Which is the most frequently used language for writing stored procedures?
    (a) SQL (b) PL/pgSQL
    (c) Both (a) and (b) (d) None of these

Answers

1. (c)	2. (a)	3. (b)	4. (a)	5. (b)	6. (c)	7. (d)	8. (a)	9. (c)	10. (d)
11. (b)	12. (b)	13. (c)	14. (b)						

Q.II Fill in the Blanks:

1. PL/pgSQL is a Block structured language.
2. Error is an illegal operation performed by the user which results in abnormal working of the program.
3. Expression are calculations or operations that return their results as one of PostgreSQL's base data types.
4. PL/pgSQL is a powerful extension for SQL.
5. PL/pgSQL code is organized as block structured code where each portion of code is designed to exist as a Function
6. PL/SQL programs are divided and written in logical blocks of code.
7. A PL/pgSQL cursor allows us to encapsulate a query and process each individual row at a time.
8. A PL/pgSQL variable is a meaningful name for a memory location which holds a value that can be changed through the block or function.
9. To create a new stored procedure, use the CREATE PROCEDURE statement.
10. Procedural Language/PostgreSQL stand for Postgresql.
11. The For-In statement is another way to iterate over rows of record set.
12. PL/pgSQL is a programming language developed specially for programming stored Proredur PostgreSQL.

Answers

1. block	2. Error	3. Expressions	4. SQL	5. function
6. PL/SQL	7. cursor	8. variable	9. CREATE PROCEDURE	10. PostgreSQL
11. FOR-IN-EXECUTE	12. procedures			

Q.III State True or False:

1. PostgreSQL allows adding custom functions developed using different programming languages like C/C++, Java, etc. True.
2. An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. True
3. SQL is a powerful extension for PL/pgSQL. False
4. The values of constants cannot be changed once they are initialized.
5. PL/SQL and PL/pgSQL are the block-structured languages.
6. The collection of structured relations with distinct relation names called DBMS.
7. A PostgreSQL cursor is a function invoked automatically whenever an event associated with a table occurs.
8. A statement performs an action within PL/pgSQL code, such as assignment of a value to a variable or the execution of a query.
9. RAISE statement is used to raise errors and exceptions during a PL/pgSQL function's operation.
10. To delete a trigger from a table, you use the DROP TRIGGER statement.
11. In PL/pgSQL the IF...THEN, IF...THEN....ELSE, CASE statement are looping statement.

Answers

1. (T)	2. (T)	3. (F)	4. (T)	5. (T)	6. (F)	7. (F)	8. (T)	9. (T)	10. (T)
11. (F)									

Q.IV Answer the following Questions:

(A) Short Answer Questions:

1. What is SQL?
2. What is PL/SQL?
3. What is PL/pgSQL?
4. Enlist features of PL/SQL.
5. Define the term stored procedure.
6. What is trigger?
7. What is function? Define it.
8. What is cursor? Enlist its types.
9. What is error?
10. How to create trigger in PL/pgSQL?
11. State two advantages of cursor.
12. Which looping statement used by PL/pgSQL?
13. What is procedure? How it differ from function?
14. Define exception.
15. Which statement used by PL/pgSQL to handle errors and exceptions.
16. Compare trigger, function and procedures (ant two points).

(B) Long Answer Questions:

1. What are the data types used by PL/pgSQL?
2. Explain features of PL/pgSQL.
3. Explain advantages and disadvantages of PL/pgSQL.
4. Explain language structure of PL/pgSQL.
5. How comments are specified in PL/pgSQL?
6. What are different types of basic statements available in PL/pgSQL?
7. Explain conditional statements in PL/pgSQL with example.
8. What are different types of loops available in PL/pgSQL?
9. What are different types of attributes in PL/pgSQL?
10. Explain concept of view in detail with example.
11. How errors and exception handled in PL/pgSQL?
12. What are different levels of RAISE statement?
13. How to create and call functions in PL/pgSQL?
14. Explain concept of cursor in detail with example.
15. Explain explicit cursor operations.
16. Define implicit cursor, explicit cursor, bound cursor, and unbound cursor.
17. Explain concept of trigger in detail with example.
18. State the events on which trigger can be fired.
19. What is row-level and statement-level trigger?
20. What are advantages and disadvantages of using triggers?
21. Explain special variables created when trigger is called automatically.

22. Consider the following Student-Marks database:
    Student (rollno integer, name varchar(30), address varchar(50), class varchar(10))
    Subject (scode varchar(10), subject_name varchar(20))
    Student-Subject are related with M-M relationship with attributes marks_scored.
    Create the above database in PostgreSQL.

Query:

1. Execute the following queries in PostgreSQL
   (i) Display the names of students scoring the maximum total marks.
   (ii) List the distinct names of all the subjects.
   (iii) Display class wise and subject wise student list.

Cursors and Triggers:

1. Write a stored function using cursors, to accept a address from the user and display the name, subject and the marks of the students staying at that address.
2. Write a stored function using cursors which will calculate total and percentage of each student.
3. Write a trigger before deleting a student record from the student table. Raise a notice and display the message "student record is being deleted".
4. Write a trigger to ensure that the marks entered for a student, with respect to a subject is never < 10 and greater than 100.

Stored Function:

1. Write a stored function using cursors to accept an address from the user and display the name, subject and the marks of the student staying at that address.

UNIVERSITY QUESTIONS AND ANSWERS

April 2015

1. What is trigger? **(5 M)**

Ans. Refer to Section 1.9.

2. Consider the following tables:
Student (sno, sname)
Teacher (tno, tname, qualification)
Student and Teacher are related with many-many relationship.
Write a plpgsql function using cursor to list details of students who have not taken 'Data structure' as a subject. **(5 M)**

Ans. Refer to Section 1.6.

April 2016

1. State the different ways to call a PL/PgSQL function. **(1 M)**

Ans. Refer to Section 1.6.1.

2. Define and explain cursor. **(5 M)**

Ans. Refer to Section 1.8.

3. Consider the following relation schema:
emp(eid, ename, dob, desig, doj, dname, sal)
Define a trigger which does not accept doj(date of joining) greater than the current date. **(5 M)**

Ans. Refer to Section 1.9.

April 2017

1. What do you mean by trigger? **(1 M)**

Ans. Refer to Section 1.9.

2. Write purpose and syntax of raise statement. (1 M)

Ans. Refer to Section 1.7.

3. Consider the following relation schema:
   Student (sno, sname)
   Teacher (tno, tname, qualification)
   Student and teacher are related with many-many relationship.
   Write a cursor to list details of students who have take RDBMS as a subject. (5 M)

Ans. Refer to Section 1.8.

October 2017

1. Differentiate between row-level trigger and statement level trigger. (1 M)

Ans. Refer to Section 1.9.

→ Fired for each affected row
→ Fired only once a stmt

2. List the purpose of SELECT INTO statement. (1 M)

Ans. Refer to Page 1.11.

3. Consider the following relation-schema:
   student(roll-no, name, address, class)
   subject(code, subject-name)
   stud-sub(roll-no, code, marks)
   Write a function which will accept rollno from the user and will display name, subjectname, and marks for the subject. (5 M)

Ans. Refer to 1.6.

April 2018

1. State difference between varchar and text data type of postgresQL. (1 M)

Ans. Refer to Section 1.3.2.

2. Define term trigger. (1 M)

Ans. Refer to Section 1.9.

3. Define term cursor. (1 M)

Ans. Refer to Section 1.8.

4. What is stored procedure? Give syntax to create stored procedure. (1 M)

Ans. Refer to Section 1.5.

5. What is trigger? Explain with syntax. (2 M)

Ans. Refer to Sections 1.9 and 1.9.1.

6. Consider the following entities and relationships. (2 M)
   Student (rollno, name, address, class)
   Stub-sub (rollno, code, marks)
   Define a trigger before insert for every row as a student, subject table, whenever marks entered is <0 or > 100, Raise an application error and display corresponding message. (3 M)

Ans. Refer to Section 1.7.

7. Consider the following relational database:
   Doctor (dno, dname, dcity)
   Hospital (hno, hname, hcity)
   doc-hosp (dno, hno)
   Write a function to return count of number of hospitals located in 'Ahmednagar' City. (3 M)

Ans. Refer to Section 1.6.

❖❖❖

Syllabus ...

1. Relational Database Design Using PLSQL **(8 Hrs.)**

1.1 Introduction to PLSQL.
1.2 PL/pgSqL: Datatypes, Language Structure.
1.3 Controlling the Program Flow, Conditional Statements, Loops.
1.4 Stored Procedures.
1.5 Stored Functions.
1.6 Handling Errors and Exceptions.
1.7 Cursors.
1.8 Triggers.

2. Transaction Concepts and Concurrency Control **(10 Hrs.)**

2.1 Describe a Transaction, Properties of Transaction, State of the Transaction.
2.2 Executing Transactions Concurrently Associated Problem in Concurrent Execution.
2.3 Schedules, Types of Schedules, Concept of Serializability, Precedence Graph for Serializability.
2.4 Ensuring Serializability by Locks, Different Lock Modes, 2PL and its Variations.
2.5 Basic Timestamp Method for Concurrency, Thomas Write Rule.
2.6 Locks with Multiple Granularity, Dynamic Database Concurrency (Phantom Problem).
2.7 Timestamps versus Locking.
2.8 Deadlock and Deadlock Handling - Deadlock Avoidance (Wait-Die, Wound-Wait), Deadlock Detection and Recovery (Wait for Graph).

3. Database Integrity and Security Concepts **(6 Hrs.)**

3.1 Domain Constraints.
3.2 Referential Integrity.
3.3 Introduction to Database Security Concepts.
3.4 Methods for Database Security.
  3.4.1 Discretionary Access Control Method.
  3.4.2 Mandatory Access Control.
  3.4.3 Role Base Access Control for Multilevel Security.
3.5 Use of Views in Security Enforcement.
3.6 Overview of Encryption Technique for Security.
3.7 Statistical Database Security.

4. Crash Recovery **(4 Hrs.)**

4.1 Failure Classification.
4.2 Recovery Concepts.
4.3 Log base recovery Techniques (Deferred and Immediate Update).
4.4 Checkpoints, Relationship between Database Manager and Buffer Cache. ARIES Recovery Algorithm.
4.5 Recovery with Concurrent Transactions (Rollback, Checkpoints, Commit).
4.6 Database Backup and Recovery from Catastrophic Failure.

5. Other Databases **(2 Hrs.)**

5.1 Introduction to Parallel and Distributed Databases.
5.2 Introduction to Object Based Databases.
5.3 XML Databases.
5.4 NoSQL Database.
5.5 Multimedia Databases.
5.6 Big Data Databases.