KARNAVATI UNIVERSITY | UIT

Unitedworld Institute Of Technology

B.Tech. Computer Science & Engineering

Semester : 3rd

Introduction To Database Management System

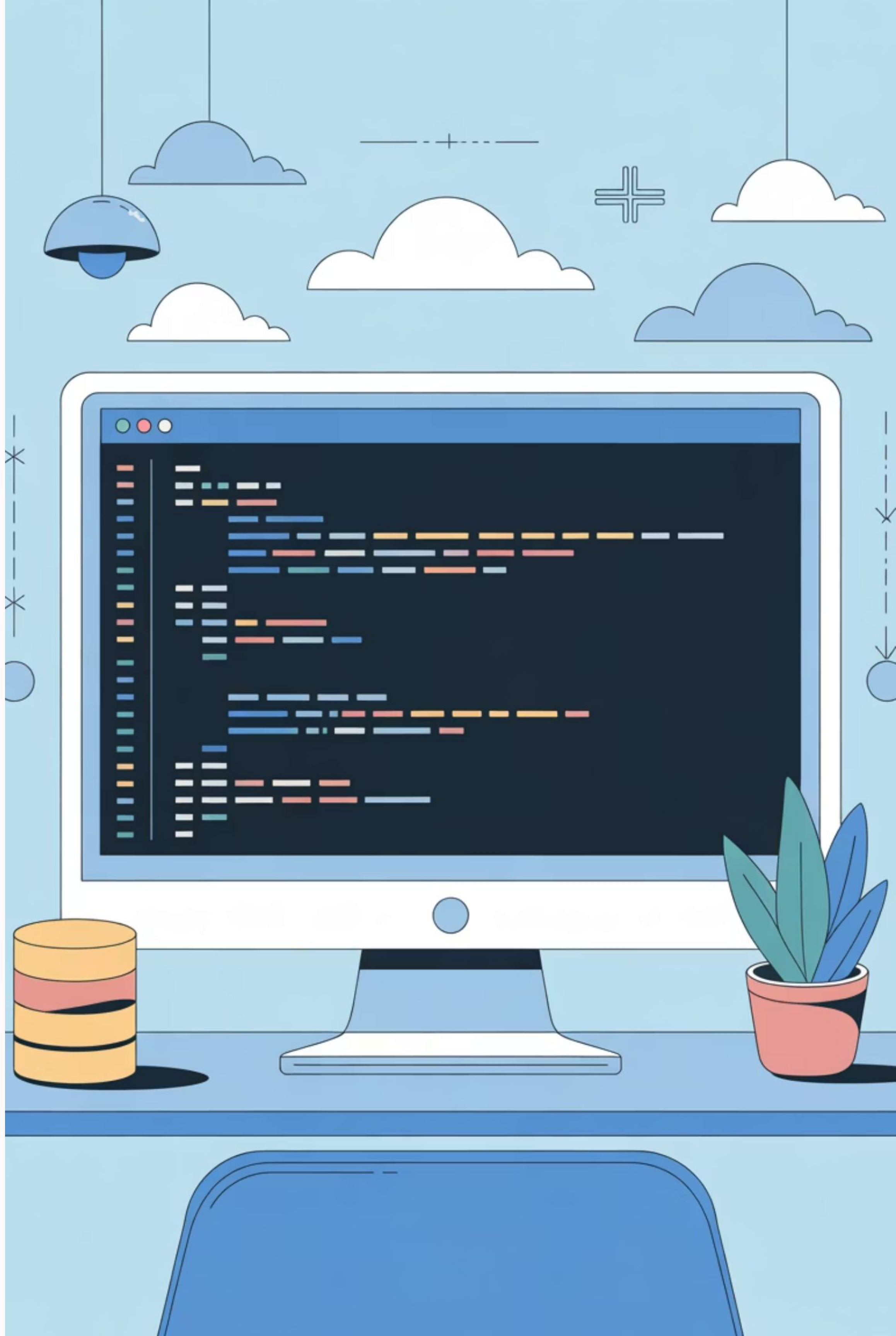
Course Code: 71203002003

Unit – 3 : STRUCTURED QUERY LANGUAGE - SQL AND PL/SQL

Prepared by:
Mr. Utsav Kapadiya
Assistant Professor (UIT)

PL/SQL: Stored Procedures, Functions, and Triggers

Master the essential building blocks of database programming and learn to write reusable, efficient, and automated database logic



What is PL/SQL?

PL/SQL (Procedural Language/Structured Query Language) extends SQL by allowing you to group multiple SQL statements together into a single, cohesive unit called a program block. These blocks combine the data manipulation power of SQL with the procedural capabilities of programming languages.

Instead of executing queries one at a time, you can bundle logic, add control structures, handle errors, and create modular, maintainable database code. This makes your database operations more powerful, secure, and efficient.

Three Core Building Blocks

Stored Procedures

Performs a specific task or series of operations. Think of it as a set of instructions you save and execute whenever needed.

Use case: Adding records, updating data, complex business logic

Functions

Similar to procedures but with a key difference: functions must return a value. They can be used directly in SQL queries.

Use case: Calculations, grade assignments, data transformations

Triggers

Special blocks that execute automatically when specific database events occur, like INSERT, UPDATE, or DELETE operations.

Use case: Audit logging, validation, enforcing business rules

Understanding Stored Procedures

A Stored Procedure is a named PL/SQL block that performs one or more operations and is stored permanently in the database. Once created, you can call it anytime—just like reusing a recipe card in a kitchen.

Key Concept: Think of a stored procedure like a recipe card. You write down the steps once and reuse them every time you need to make that dish. No need to rewrite the instructions each time!

Stored Procedure Syntax

Basic Structure

```
CREATE [OR REPLACE] PROCEDURE procedure_name(

parameter_name datatype [, ...])ISBEGIN  -- SQL or PL/SQL

statementsEND;/
```

Syntax Breakdown

CREATE OR REPLACE: Creates new or updates existing procedure

parameter_name: Input/output variables

IS/AS: Marks beginning of declaration section

BEGIN...END: Contains executable statements

/ Executes the creation statement

Stored Procedure Example

Let's create a practical stored procedure that adds a student record to the database. This procedure accepts three parameters: student ID, name, and marks.

```
CREATE OR REPLACE PROCEDURE add_student(  p_id IN NUMBER,  p_name IN VARCHAR2,  p_marks IN NUMBER)ISBEGIN  INSERT INTO student(id, name, marks)
VALUES (p_id, p_name, p_marks);    DBMS_OUTPUT.PUT_LINE('Student record added successfully.');END;/
```

Executing the Procedure

```
BEGIN  add_student(101, 'Aarav', 90);END;/
```

Expected Output

```
Student record added successfully.
```

The student record with ID 101 is now inserted into the database.

Why Use Stored Procedures?

Code Reusability

Write once, use everywhere. Instead of repeating the same SQL logic across multiple applications, call one stored procedure from anywhere.

Improved Performance

Procedures are precompiled and stored in the database. This means faster execution compared to sending SQL statements each time.

Easier Maintenance

When business logic changes, update the procedure once instead of modifying code in multiple places across your applications.

Enhanced Security

Grant execute permissions on procedures without exposing underlying tables. Users can perform operations without direct table access.

Understanding Functions

A Function is similar to a stored procedure, but with one critical difference: it must return a value. Functions can be used directly in SQL statements, making them incredibly versatile for calculations and transformations.

Analogy: If a procedure is like a machine that performs an action, a function is like a calculator—it takes input, processes it, and gives back a result you can use immediately.

Function Syntax and Example

General Syntax

```
CREATE [OR REPLACE] FUNCTION function_name(
parameter_name datatype [, ...])RETURN
return_datatypeISBEGIN  -- statements  RETURN value;END;/
```

The RETURN clause is mandatory and specifies the data type of the value the function will return.

Practical Example: Grade Calculator

```
CREATE OR REPLACE FUNCTION get_grade( p_marks NUMBER)RETURN
VARCHAR2IS v_grade VARCHAR2(2);BEGIN IF p_marks >= 90 THEN
v_grade := 'A'; ELSIF p_marks >= 75 THEN v_grade := 'B'; ELSE
v_grade := 'C'; END IF;  RETURN v_grade;END;/
```

Calling Functions

Method 1: PL/SQL Block

```
DECLARE  v_result VARCHAR2(2);BEGIN  v_result :=
get_grade(82);  DBMS_OUTPUT.PUT_LINE('Grade: ' ||
v_result);END;/
```

Output: `Grade: B`

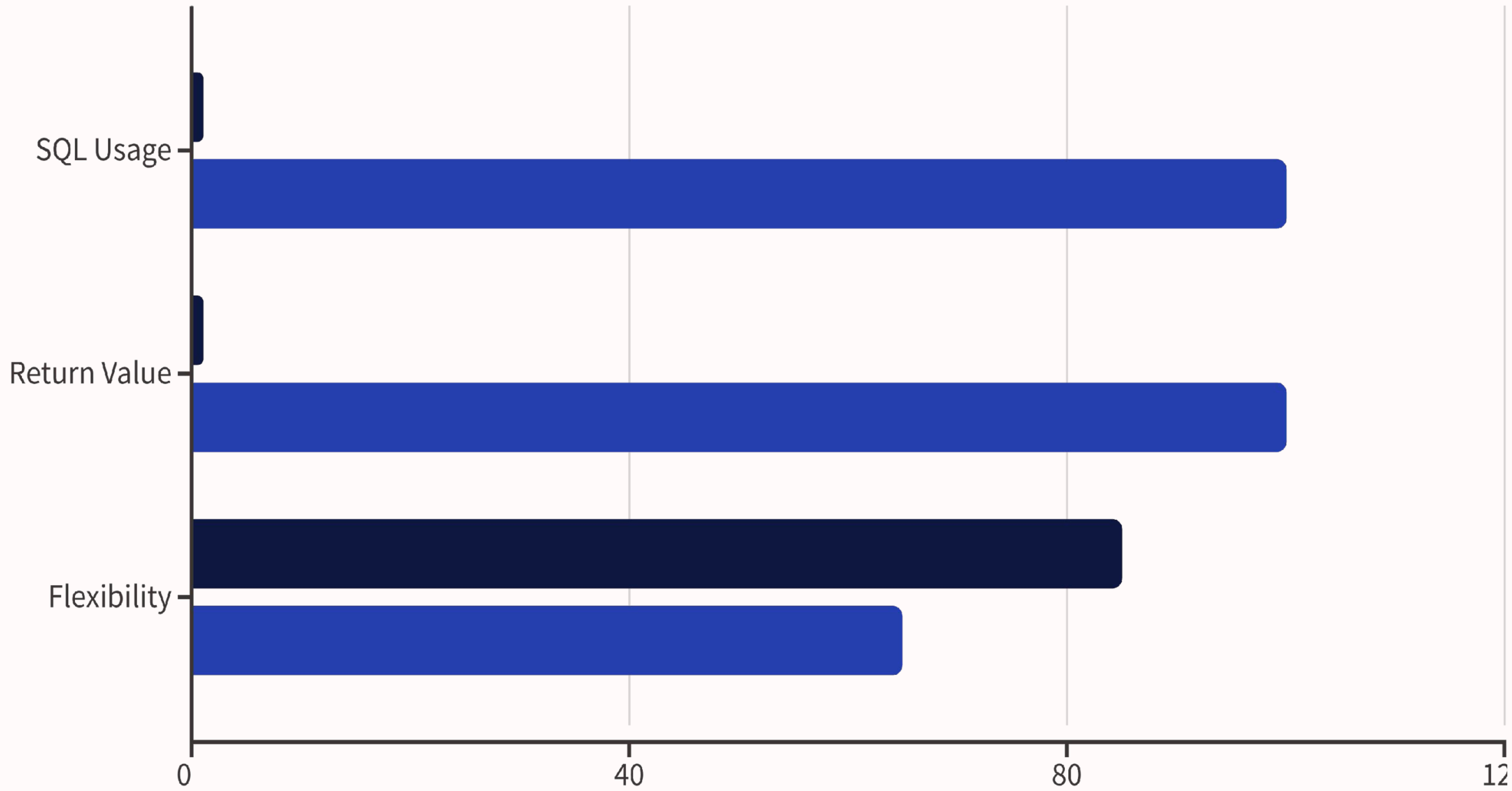
Method 2: Direct SQL Query

```
SELECT get_grade(82) FROM dual;
```

Output: Returns `B`

Functions can be used anywhere you'd use a SQL expression: SELECT clauses, WHERE conditions, ORDER BY statements, and more.

Procedures vs Functions



Understanding Triggers

A Trigger is a special type of stored procedure that executes automatically when a specific database event occurs. Unlike procedures and functions that you call manually, triggers respond to events like INSERT, UPDATE, or DELETE operations on tables.

Perfect Analogy: A trigger is like an automatic alarm system. When someone opens the door (INSERT event), the alarm automatically rings (trigger fires). You don't need to press any button—it just happens!

Trigger Syntax

```
CREATE [OR REPLACE] TRIGGER trigger_nameBEFORE | AFTERINSERT | UPDATE | DELETEON table_name[FOR EACH ROW]BEGIN  -- Actions to perform automaticallyEND;/
```

1

Timing

Choose BEFORE or AFTER to specify when the trigger fires relative to the triggering event

2

Event

Specify which DML operation activates the trigger: INSERT, UPDATE, DELETE, or combinations

3

Target Table

Define which table the trigger monitors for events

4

Row Level

Add FOR EACH ROW to execute once per affected row (optional)

Trigger Example: Audit Logging

Let's create a practical trigger that automatically logs every time a new student record is inserted. This is useful for tracking database changes and maintaining audit trails.

Step 1: Create Log Table

```
CREATE TABLE student_log (  user_name VARCHAR2(30),
action_time DATE,  action VARCHAR2(20));
```

Step 2: Create the Trigger

```
CREATE OR REPLACE TRIGGER student_insert_triggerAFTER
INSERT ON studentFOR EACH ROWBEGIN  INSERT INTO
student_log  VALUES (USER, SYSDATE, 'Record
Inserted');END;/
```

Now, whenever someone inserts a student record, the trigger automatically creates a log entry capturing who did it and when.

Trigger in Action

1

User Action

```
INSERT INTO studentVALUES (201,
'Isha', 95);
```

A developer or application inserts a new student record

2

Trigger Fires

The `student_insert_trigger` automatically executes without any manual call

3

Log Created

```
SELECT * FROM student_log;
```

Result:USER: UTSAVACTION_TIME: 05-NOV-2025ACTION: Record Inserted

This happens seamlessly in the background, creating a complete audit trail of all database modifications.

Types of Triggers

BEFORE Trigger

Fires before the DML statement executes. Perfect for validation, data modification, or preventing invalid operations before they occur.

AFTER Trigger

Fires after the DML statement completes. Ideal for logging, creating audit trails, or cascading changes to related tables.

INSTEAD OF Trigger

Used on views (not tables) to perform custom operations. Allows you to make non-updatable views modifiable through custom logic.

Row-Level Trigger

Uses `FOR EACH ROW` clause. Executes once for every row affected by the DML statement. Access to `:OLD` and `:NEW` values.

Statement-Level Trigger

Executes once for the entire DML statement, regardless of how many rows are affected. More efficient for bulk operations.

Advantages of Triggers

Automatic Business Rules

Enforce complex business logic automatically without relying on application code. Rules execute consistently regardless of which application accesses the database.

Audit Trails

Maintain detailed logs of who changed what and when. Essential for compliance, security analysis, and debugging data issues.

Data Integrity

Validate data before it enters the database. Prevent orphaned records, enforce referential integrity, and maintain data consistency automatically.

Automated Calculations

Automatically compute derived values, update related tables, or maintain summary data without manual intervention.

Trigger Limitations

Difficult to Debug

Because triggers execute automatically and silently, tracking down problems can be challenging. Unexpected trigger behavior may cause mysterious errors in applications.

Performance Impact

Overuse of triggers can significantly slow down DML operations. Each insert, update, or delete must wait for all related triggers to complete before proceeding.

Hidden Complexity

Business logic buried in triggers is invisible to applications. Developers may not realize triggers exist, leading to confusion about why data changes unexpectedly.

Best Practice: Use triggers judiciously for critical database-level operations like auditing and validation. Avoid implementing complex business logic that would be better suited for application code.

Quick Reference Comparison

Feature	Stored Procedure	Function	Trigger
Returns Value?	No	Yes (required)	No
Call Method	Manual (EXECUTE/BEGIN)	Manual (assignment/query)	Automatic (on event)
Use in SQL?	No	Yes	No
Primary Purpose	Perform tasks	Calculate and return	Respond to events
Execution Trigger	User/program call	User/program call	DML event (INSERT/UPDATE/DELETE)

Procedure: Like a recipe—follow the steps when you need to cook the dish

Function: Like a calculator—input numbers, get instant result

Trigger: Like an alarm—automatically activates when someone opens the door

Practice Examples Summary

Here's a complete overview of the commands we've learned and how to execute each type of PL/SQL block:

Stored Procedure

Command:

```
BEGIN  add_student(1, 'Riya',
89);END;/
```

Result: Adds a new student record to the database

Function

Command:

```
SELECT get_grade(82)FROM dual;
```

Result: Returns `'B'` based on the marks provided

Trigger

Command:

```
INSERT INTO studentVALUES (201,
'Isha', 95);
```

Result: Automatically logs the insert event to the audit table

---

Congratulations! You now understand the three fundamental building blocks of PL/SQL. Practice creating your own procedures, functions, and triggers to solidify these concepts. Start with simple examples and gradually build more complex database logic as you grow more comfortable with the syntax and patterns.

Thank You!