1. ***Discuss the four methods of file organization.***

**1. Sequential File Organization**

* **Description**: Records are stored one after another in a specific order, typically based on a key field.
* **Key Features**:
  + Access is sequential; the system reads from the start until the desired record is found.
  + Records are ordered logically, such as alphabetically or numerically.

**2. Direct (or Random) File Organization**

* **Description**: Records are stored at specific locations determined by a hashing algorithm applied to a key field.
* **Key Features**:
  + Direct access to records without the need for sequential searching.
  + Hashing function maps keys to storage locations.

**3. Indexed File Organization**

* **Description**: An index is created to keep track of the locations of records. The index acts as a lookup table that maps key fields to the addresses of records in the file.
* **Key Features**:
  + Allows both sequential and random access.
  + Multiple indexes can be maintained for different key fields.

**4. Clustered File Organization**

* **Description**: Records with similar characteristics or frequently accessed together are stored physically close to each other.
* **Key Features**:
  + Optimizes performance for queries involving related records.
  + Grouping is based on one or more clustering fields.

**Summary Table:**

| **File Organization Method** | **Access Type** | **Best For** | **Drawbacks** |
| --- | --- | --- | --- |
| Sequential | Sequential only | Batch processing | Inefficient random access |
| Direct | Random only | Real-time systems | Collision handling complexity |
| Indexed | Sequential/Random | Databases, complex queries | Storage and maintenance overhead |
| Clustered | Mixed | Grouped or related data access | Complexity in management |

1. ***Write and explain few advantages of Indexing.***

Indexing is a method of improving the efficiency of data retrieval in databases and file systems by maintaining a separate data structure (index) that holds pointers to the locations of the actual records. Here are a few key advantages:

**1. Faster Data Retrieval**

* **Explanation**: Indexing allows for quicker searches by reducing the number of records that need to be scanned. Instead of performing a full scan of the file, the system can use the index to directly locate the desired data.
* **Benefit**: Enhances performance, especially for large datasets, by enabling faster query execution.

**2. Efficient Random Access**

* **Explanation**: Indexing provides direct access to specific records by using key fields. It eliminates the need for sequentially searching through the entire dataset.
* **Benefit**: Suitable for applications where quick lookups or frequent random access is required, such as customer databases.

**3. Improved Query Performance**

* **Explanation**: Indexing optimizes the execution of complex queries by allowing the database to quickly find relevant records without scanning the whole table.
* **Benefit**: Speeds up operations like filtering, sorting, and joining data.

**4. Facilitates Multi-Key Searches**

* **Explanation**: Composite indexes can be created on multiple fields, enabling searches based on combinations of attributes.
* **Benefit**: Useful for applications requiring multi-dimensional queries, such as filtering data by date and customer location.

**5. Supports Sorting**

* **Explanation**: Indexing helps maintain records in a sorted order based on key fields, enabling faster sorting of results.
* **Benefit**: Reduces the time required for operations that involve ordering records, such as generating reports.

**6. Reduces Disk I/O**

* **Explanation**: By accessing only the necessary portions of the file using the index, the number of disk read/write operations is minimized.
* **Benefit**: Improves overall system efficiency and reduces latency.

**7. Facilitates Constraints and Uniqueness**

* **Explanation**: Indexes help enforce constraints like primary keys and unique keys by ensuring no duplicate entries exist for indexed fields.
* **Benefit**: Ensures data integrity and reliability.

**8. Supports Full-Text Search**

* **Explanation**: Indexing enables full-text search capabilities, where documents or text fields can be searched for specific words or phrases.
* **Benefit**: Makes searching through large text fields faster and more effective

1. ***State the causes of database failures.***

Database failures can occur due to a variety of reasons, ranging from hardware malfunctions to human errors. These failures disrupt the database's ability to function correctly and may lead to data loss or system downtime. The common causes of database failures include:

**1. Hardware Failures**

* **Description**: Malfunctions in physical components like servers, storage devices, or network hardware.
* **Examples**:
  + Hard disk crashes.
  + Power supply failures.
  + Memory or CPU errors.
* **Impact**: Data corruption, loss of accessibility, or complete database downtime.

**2. Software Failures**

* **Description**: Bugs or errors in the database management system (DBMS) software or related applications.
* **Examples**:
  + Faulty updates or patches.
  + Misconfigurations in the DBMS.
  + Operating system crashes.
* **Impact**: Database malfunctions, inability to process queries, or incorrect outputs.

**3. Power Failures**

* **Description**: Unexpected loss of power during database operations.
* **Examples**:
  + Sudden blackouts.
  + Unstable power supply.
* **Impact**: Abrupt termination of transactions, leading to data inconsistency or corruption.

**4. Human Errors**

* **Description**: Mistakes made by users, administrators, or developers while managing the database.
* **Examples**:
  + Accidental deletion or modification of records.
  + Misconfigured backups or restoration processes.
  + Incorrect SQL queries.
* **Impact**: Data loss, inconsistency, or system downtime.

**5. Transaction Failures**

* **Description**: Failures during the execution of database transactions.
* **Examples**:
  + Deadlocks.
  + Violations of integrity constraints.
  + Insufficient resources to complete a transaction.
* **Impact**: Partial updates to the database, leading to inconsistencies.

**6. Concurrency Issues**

* **Description**: Problems arising when multiple transactions attempt to access or modify the same data simultaneously.
* **Examples**:
  + Lost updates.
  + Dirty reads.
  + Non-repeatable reads.
* **Impact**: Inconsistent or incorrect data.

**7. Security Breaches**

* **Description**: Unauthorized access, hacking, or malicious attacks on the database.
* **Examples**:
  + SQL injection attacks.
  + Data breaches by insiders or external entities.
  + Ransomware affecting database files.
* **Impact**: Data theft, corruption, or unavailability.

**8. Natural Disasters**

* **Description**: Environmental events causing physical damage to database infrastructure.
* **Examples**:
  + Floods, earthquakes, or fires.
  + Lightning strikes causing power surges.
* **Impact**: Complete loss of database infrastructure or severe downtime.

**9. Network Failures**

* **Description**: Problems with the communication between the database server and its clients.
* **Examples**:
  + Network outages.
  + High latency or packet loss.
* **Impact**: Disrupted access to the database or incomplete transactions.

**10. Lack of Proper Maintenance**

* **Description**: Failure to perform regular database upkeep.
* **Examples**:
  + Not updating the DBMS.
  + Ignoring error logs or performance warnings.
* **Impact**: Gradual performance degradation, increased susceptibility to failures.

**11. Overloading**

* **Description**: Excessive load on the database server due to high user traffic or resource-intensive queries.
* **Examples**:
  + Too many concurrent users.
  + Complex analytics or reporting tasks.
* **Impact**: Server crashes, slow performance, or timeouts.

1. ***Write short on Index and explain structure of Index in Database.***

An **index** in a database is a data structure that improves the speed of data retrieval operations on a table. It works like a book's index, where the system can locate data without scanning the entire table. Indexes are created on one or more columns of a table and provide pointers to the actual data stored in the database.

**Structure of an Index**

Indexes are typically organized using specific data structures to enable efficient data access. Common index structures include:

**1. B-Tree Index**

* **Description**: The most widely used structure in relational databases.
* **Structure**:
  + A balanced tree with nodes containing keys and pointers.
  + The root node points to child nodes, which further divide into leaf nodes containing actual data pointers.
  + Ensures balanced height for consistent access time.
* **Advantages**:
  + Efficient for both range queries (e.g., BETWEEN, >=, <=) and exact match queries.
  + Automatically rebalances itself when data is inserted or deleted.
* **Use Case**: General-purpose indexing for most applications.

**2. Hash Index**

* **Description**: Uses a hash function to map keys to specific locations.
* **Structure**:
  + Key-value pairs, where the hash function calculates the location of the value for a given key.
  + Direct mapping enables fast lookups.
* **Advantages**:
  + Extremely fast for exact match queries.
* **Disadvantages**:
  + Inefficient for range queries.
  + May require additional handling for hash collisions.
* **Use Case**: Applications requiring exact lookups, such as primary key searches.

**3. Bitmap Index**

* **Description**: Represents indexed columns as a series of bitmaps.
* **Structure**:
  + Each unique value in the column has a bitmap vector, with bits representing the presence (1) or absence (0) of the value in rows.
* **Advantages**:
  + Compact storage for columns with low cardinality (few unique values).
  + Ideal for queries combining multiple conditions (AND, OR).
* **Disadvantages**:
  + Inefficient for high-cardinality columns.
* **Use Case**: Data warehouses and analytics systems.

**4. Clustered Index**

* **Description**: The data rows are stored physically in the order of the indexed column.
* **Structure**:
  + The table itself is the index; no separate index structure is maintained.
  + Only one clustered index per table is allowed.
* **Advantages**:
  + Faster access to data that is frequently sorted or searched in sequence.
* **Disadvantages**:
  + Updates can be costly as they may require reordering.
* **Use Case**: Tables where data access often involves range queries or sorting.

**5. Non-Clustered Index**

* **Description**: Maintains a separate structure from the actual data table.
* **Structure**:
  + Contains key values and pointers to the data rows in the table.
  + Allows multiple non-clustered indexes on a single table.
* **Advantages**:
  + Flexible and supports multiple columns for indexing.
* **Disadvantages**:
  + Slower than clustered indexes for sequential data access.
* **Use Case**: Exact lookups or queries involving non-primary key columns.

1. ***List and explain different attributes of Indexing.***

Indexing in databases involves various attributes that define how indexes function and impact database performance. These attributes help determine the efficiency, scope, and limitations of an index. Here are the key attributes of indexing:

**1. Key Field (Indexed Column)**

* **Description**: The column(s) on which the index is created.
* **Purpose**: Defines the data the index is built upon.
* **Example**: In a customers table, an index on the customer\_id column allows quick lookup of customer details using their IDs.
* **Impact**: Selecting the right key field is crucial for query performance.

**2. Uniqueness**

* **Description**: Determines whether the index ensures unique values in the indexed column(s).
* **Types**:
  + **Unique Index**: Guarantees no duplicate values (e.g., primary keys).
  + **Non-Unique Index**: Allows duplicate values (e.g., columns with repeated data).
* **Example**: A unique index on email ensures no two users can register with the same email address.
* **Impact**: Enhances data integrity when uniqueness is required.

**3. Clustered vs. Non-Clustered**

* **Description**: Determines whether the index affects the physical storage of data.
  + **Clustered Index**: Data rows are stored in the order of the indexed column.
  + **Non-Clustered Index**: A separate structure points to the actual data rows.
* **Example**: A clustered index on order\_date stores orders sorted by date.
* **Impact**: Clustered indexes improve range queries, while non-clustered indexes are versatile for lookups.

**4. Cardinality**

* **Description**: Refers to the uniqueness of values in the indexed column.
* **Types**:
  + **High Cardinality**: Many unique values (e.g., SSN or ProductID).
  + **Low Cardinality**: Few unique values (e.g., Gender or Status).
* **Example**: An index on city in a national database has lower cardinality than one on customer\_id.
* **Impact**: High-cardinality indexes are more effective for unique lookups, while low-cardinality indexes are useful for filtering and grouping.

**5. Multi-Column Index**

* **Description**: An index built on two or more columns.
* **Purpose**: Facilitates queries involving combinations of columns.
* **Example**: An index on (first\_name, last\_name) speeds up searches for a full name.
* **Impact**: Efficient for compound queries but may have limitations if queried columns don't follow the defined order.

**6. Covering Index**

* **Description**: An index that contains all the columns needed to satisfy a query.
* **Purpose**: Reduces the need to access the actual table.
* **Example**: An index on product\_id and price helps fetch both columns directly for pricing queries.
* **Impact**: Enhances performance by avoiding additional table lookups.

**7. Storage Overhead**

* **Description**: Refers to the additional disk space required to maintain the index.
* **Impact**:
  + Larger indexes consume more storage.
  + Frequent updates or inserts may increase maintenance costs.
* **Example**: A large table with multiple indexes may significantly increase disk usage.

**8. Order**

* **Description**: Defines whether the index stores data in ascending or descending order.
* **Purpose**: Supports queries requiring specific sort orders.
* **Example**: An index on price in ascending order supports queries like ORDER BY price ASC.
* **Impact**: Pre-sorted indexes reduce sorting overhead during query execution.

**9. Index Visibility**

* **Description**: Determines whether the index is visible to the query optimizer.
* **Types**:
  + **Visible Index**: Actively used by the optimizer for queries.
  + **Invisible Index**: Not used by default but can be manually referenced for testing or special cases.
* **Impact**: Useful for maintaining or testing index impact without affecting regular queries.

**10. Index Type**

* **Description**: Specifies the structure of the index.
* **Examples**:
  + **B-Tree**: Balanced tree for range and exact match queries.
  + **Hash**: Hash mapping for exact match queries.
  + **Bitmap**: Bitmaps for low-cardinality columns.
* **Impact**: The choice of index type depends on the nature of queries and data.

**11. Index Maintenance**

* **Description**: Refers to the effort required to update indexes during insert, update, or delete operations.
* **Impact**:
  + Frequent changes to indexed columns increase overhead.
  + Properly maintained indexes ensure consistent performance.

1. ***Draw and explain PL/SQL block structure.***

A PL/SQL block is the basic unit of a PL/SQL program and consists of three main sections: **Declarative**, **Executable**, and **Exception Handling**. These sections are logically organized to define variables, write program logic, and handle errors.

**Structure of a PL/SQL Block**

Here is a visual representation of the PL/SQL block structure:

DECLARE

-- Declaration Section

-- Variables, constants, cursors, etc. are declared here.

BEGIN

-- Executable Section

-- Main program logic: SQL statements, PL/SQL code, loops, conditions.

EXCEPTION

-- Exception Handling Section

-- Handles runtime errors or exceptions.

END;

/

**Explanation of Each Section**

**1. Declarative Section**

* **Purpose**: To declare variables, constants, and other objects that will be used in the block.
* **Key Points**:
  + Optional; you can skip this section if no declarations are required.
  + Variables declared here are local to the block and cannot be accessed outside.
* **Syntax Example**:

DECLARE

employee\_name VARCHAR2(50);

bonus NUMBER := 1000;

**2. Executable Section**

* **Purpose**: Contains the core logic of the block, where SQL statements, control structures, and PL/SQL code are executed.
* **Key Points**:
  + Mandatory section; every PL/SQL block must have at least one statement here.
  + Includes statements like SELECT, INSERT, UPDATE, DELETE, loops, and conditional checks.
* **Syntax Example**:

BEGIN

SELECT first\_name INTO employee\_name FROM employees WHERE employee\_id = 101;

DBMS\_OUTPUT.PUT\_LINE('Employee Name: ' || employee\_name);

**3. Exception Handling Section**

* **Purpose**: Handles runtime errors or exceptions that occur during the execution of the block.
* **Key Points**:
  + Optional; if not provided, the block will terminate with an unhandled exception if an error occurs.
  + Includes pre-defined exceptions (NO\_DATA\_FOUND, TOO\_MANY\_ROWS) or user-defined exceptions.
* **Syntax Example**:

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('No employee found with the given ID.');

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('An unexpected error occurred.');

**Complete Example of a PL/SQL Block**

DECLARE

employee\_name VARCHAR2(50);

salary NUMBER;

BEGIN

-- Fetch employee details

SELECT first\_name, salary INTO employee\_name, salary FROM employees WHERE employee\_id = 101;

-- Display employee details

DBMS\_OUTPUT.PUT\_LINE('Employee Name: ' || employee\_name);

DBMS\_OUTPUT.PUT\_LINE('Salary: ' || salary);

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Employee not found.');

WHEN OTHERS THEN

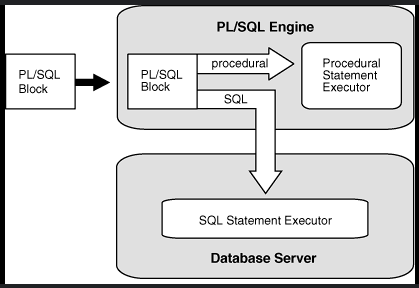
DBMS\_OUTPUT.PUT\_LINE('An unexpected error occurred.');

END;

/

**Key Characteristics**

1. **Modularity**: PL/SQL blocks can be nested, making it easy to build modular programs.
2. **Error Handling**: Built-in mechanism to handle exceptions ensures robust programs.
3. **Reusability**: A block can be reused by embedding it in procedures, functions, or packages.



1. ***State the uses and advantages of PL/SQL.***

**Uses of PL/SQL**

PL/SQL (Procedural Language/Structured Query Language) is Oracle's procedural extension to SQL. It combines SQL's data manipulation power with procedural constructs, enabling the creation of robust, reusable, and efficient database applications.

**1. Creating Stored Procedures and Functions**

* PL/SQL allows the creation of reusable **procedures** and **functions** that encapsulate business logic and can be called from applications or other database operations.

**2. Building Triggers**

* **Triggers** in PL/SQL automatically execute predefined actions in response to specific events (e.g., INSERT, UPDATE, or DELETE operations).

**3. Implementing Complex Business Logic**

* PL/SQL is used to write logic that involves loops, conditionals, and multi-step operations, which cannot be performed using plain SQL.

**4. Transaction Management**

* It allows fine-grained control over database transactions, including **COMMIT**, **ROLLBACK**, and **SAVEPOINT**.

**5. Error Handling**

* PL/SQL provides robust error-handling capabilities through its **EXCEPTION** block, enabling developers to gracefully handle runtime errors.

**6. Batch Processing**

* PL/SQL is used to write **scripts** for bulk operations, such as data migrations, batch updates, or generating reports.

**7. Integration with Applications**

* PL/SQL integrates seamlessly with client applications, enabling database-centric application development with minimal effort.

**8. Improved Performance for SQL Operations**

* Complex SQL operations embedded in PL/SQL are executed on the server side, reducing network traffic and improving efficiency.

**Advantages of PL/SQL**

**1. Procedural Programming Capabilities**

* PL/SQL extends SQL by adding programming constructs like loops, conditionals, and variables, allowing developers to create more sophisticated logic.

**2. Enhanced Performance**

* By bundling SQL statements in a PL/SQL block, fewer round trips are required between the client and the server, reducing network overhead and improving performance.

**3. Portability**

* PL/SQL code can run on any Oracle database without modification, ensuring compatibility across different environments.

**4. Robust Error Handling**

* The **EXCEPTION** handling mechanism in PL/SQL allows developers to anticipate and gracefully manage runtime errors, improving application stability.

**5. High Security**

* PL/SQL allows developers to hide business logic in the database using stored procedures, functions, and packages, ensuring better security and data integrity.

**6. Modular Design**

* PL/SQL encourages modular programming by allowing developers to create reusable code components like procedures, functions, and packages, reducing duplication and simplifying maintenance.

**7. Transaction Control**

* PL/SQL provides control over transactions with commands like **COMMIT**, **ROLLBACK**, and **SAVEPOINT**, ensuring data consistency.

**8. Integration with SQL**

* PL/SQL seamlessly integrates SQL with procedural constructs, making it easy to embed SQL queries in application logic.

**9. Reduced Network Traffic**

* PL/SQL executes entire blocks of code on the database server, reducing the need for frequent communication between the client and the server.

**10. Scalability**

* PL/SQL can handle large volumes of data and complex business logic, making it suitable for enterprise-grade applications.

1. ***Explain the term stored procedure, and give examples why stored procedures are useful.***

**What is a Stored Procedure?**

A **stored procedure** is a precompiled block of SQL and procedural code (written in PL/SQL or another database-supported language) stored within the database. It performs a specific task and can be executed multiple times by applications or users. Stored procedures encapsulate logic and operations in a modular format, reducing redundancy and improving maintainability.

**Key Features of Stored Procedures**

1. **Precompiled**: Stored procedures are compiled and stored in the database, making them faster to execute.
2. **Reusable**: They can be reused by multiple applications, ensuring consistency across the system.
3. **Secure**: Access to the underlying database objects can be controlled through procedures.
4. **Support for Parameters**: Procedures can accept **input**, **output**, or **input-output** parameters for flexibility.

**Syntax Example**

A simple stored procedure example in PL/SQL:

CREATE OR REPLACE PROCEDURE GetEmployeeDetails (emp\_id IN NUMBER, emp\_name OUT VARCHAR2) IS

BEGIN

SELECT first\_name INTO emp\_name FROM employees WHERE employee\_id = emp\_id;

END;

/

* **emp\_id**: Input parameter to pass the employee ID.
* **emp\_name**: Output parameter to return the employee's name.

**Why Stored Procedures are Useful**

**1. Performance Optimization**

* **Example**: Instead of sending multiple SQL queries from an application to the database, a stored procedure encapsulates the logic and executes it on the server. This reduces network traffic and improves performance.
* **Benefit**: Reduces latency and improves the speed of complex operations.

**2. Code Reusability**

* **Example**: A procedure to calculate sales tax can be reused across different applications.
* **Benefit**: Saves development time and ensures consistent results.

**3. Centralized Logic**

* **Example**: A stored procedure to manage user authentication ensures all applications follow the same logic.
* **Benefit**: Simplifies maintenance and reduces errors caused by inconsistent logic in different applications.

**4. Enhanced Security**

* **Example**: Users can be granted permission to execute a stored procedure without having direct access to the underlying tables.
* **Benefit**: Protects sensitive data and enforces better security policies.

**5. Error Handling**

* **Example**: A stored procedure can handle runtime exceptions like division by zero or missing data and return meaningful error messages.
* **Benefit**: Improves application stability and user experience.

**6. Simplification of Complex Operations**

* **Example**: A procedure can automate daily tasks like calculating payroll or generating reports based on multiple tables.
* **Benefit**: Reduces manual effort and minimizes the risk of human errors.

**7. Scalability**

* **Example**: A procedure to handle bulk data processing for analytics.
* **Benefit**: Ensures the system can handle increasing workloads efficiently.

**8. Dependency Management**

* **Example**: Changes to the logic can be made in one stored procedure without altering multiple application layers.
* **Benefit**: Simplifies updates and reduces the risk of breaking dependencies.

**Real-World Use Cases**

1. **Data Validation**: A stored procedure can validate data before inserting it into the database.
2. **Automated Backups**: Automating database backups via stored procedures scheduled in a database job.
3. **Business Logic Enforcement**: Enforcing rules like "only managers can approve expense reports."
4. **Reporting**: Aggregating sales data by region or time period for analytics.
5. ***How do you write comments in a PL/SQL code? Explain user input and output statement of PL/SQL.***

**Writing Comments in PL/SQL Code**

Comments in PL/SQL help improve code readability by explaining the purpose or logic of the code. There are two types of comments:

**1. Single-Line Comments**

* **Syntax**: Use -- to start a comment. The text following -- on the same line is considered a comment.
* **Example**:

-- This is a single-line comment

SELECT \* FROM employees WHERE department\_id = 10;

**2. Multi-Line Comments**

* **Syntax**: Enclosed between /\* and \*/. Can span multiple lines.
* **Example**:
* /\*

This is a multi-line comment.

It is useful for providing detailed descriptions

or temporarily disabling large blocks of code.

\*/

BEGIN

DBMS\_OUTPUT.PUT\_LINE('Hello, PL/SQL!');

END;

**User Input and Output Statements in PL/SQL**

PL/SQL provides mechanisms to interact with users through **input** and **output** operations. These are commonly used in scripts or applications integrated with PL/SQL code.

**1. Output Statement: DBMS\_OUTPUT.PUT\_LINE**

* **Purpose**: Displays messages or data to the user, typically for debugging or informational purposes.
* **Syntax**:

DBMS\_OUTPUT.PUT\_LINE(message);

* **Example**:

BEGIN

DBMS\_OUTPUT.PUT\_LINE('Welcome to PL/SQL Programming!');

END;

* **Note**: To see the output, the server output must be enabled:

**2. Input Statements**

PL/SQL itself does not provide built-in input functionality directly. Input is typically handled:

* **Externally**: Through application frontends, SQL\*Plus, or tools like Oracle Forms.
* **Via Bind Variables**: Used in interactive environments like SQL\*Plus to accept user input.

**Example Using Bind Variables in SQL\*Plus**

-- Declare a variable to hold user input

VARIABLE emp\_id NUMBER;

-- Accept input from the user

EXEC :emp\_id := &Enter\_Employee\_ID;

-- Use the input in a PL/SQL block

BEGIN

DECLARE

emp\_name VARCHAR2(50);

BEGIN

SELECT first\_name INTO emp\_name FROM employees WHERE employee\_id = :emp\_id;

DBMS\_OUTPUT.PUT\_LINE('Employee Name: ' || emp\_name);

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('No employee found with the given ID.');

END;

END;

/

**Explanation**

1. **Output (DBMS\_OUTPUT.PUT\_LINE)**:
   * Outputs messages to the console.
   * Often used for debugging and user notifications.
2. **Input**:
   * Direct input in PL/SQL is not possible.
   * Interactive tools like SQL\*Plus use bind variables (:variable) to accept user input.
3. ***Write a PL/SQL code to print largest number from three numbers (accept three numbers from user)***

DECLARE

num1 NUMBER;

num2 NUMBER;

num3 NUMBER;

largest NUMBER;

BEGIN

-- Accept input from the user

num1 := &Enter\_First\_Number;

num2 := &Enter\_Second\_Number;

num3 := &Enter\_Third\_Number;

-- Determine the largest number

IF num1 > num2 AND num1 > num3 THEN

largest := num1;

ELSIF num2 > num1 AND num2 > num3 THEN

largest := num2;

ELSE

largest := num3;

END IF;

-- Print the largest number

DBMS\_OUTPUT.PUT\_LINE('The largest number is: ' || largest);

END;

/

1. ***Explain any one control structure in PL/SQL with example.***

The **IF-THEN-ELSE** control structure is used in PL/SQL to execute different blocks of code based on certain conditions. It allows conditional branching, enabling the program to perform specific actions depending on whether a condition is true or false.

IF condition THEN

-- Statements to execute if the condition is true

ELSIF another\_condition THEN

-- Statements to execute if the second condition is true

ELSE

-- Statements to execute if none of the above conditions are true

END IF;

**Example: Determine Grade Based on Marks**

DECLARE

marks NUMBER := &Enter\_Marks;

grade VARCHAR2(10);

BEGIN

IF marks >= 90 THEN

grade := 'A+';

ELSIF marks >= 80 THEN

grade := 'A';

ELSIF marks >= 70 THEN

grade := 'B';

ELSIF marks >= 60 THEN

grade := 'C';

ELSE

grade := 'F';

END IF;

DBMS\_OUTPUT.PUT\_LINE('The grade is: ' || grade);

END;

/

1. ***Show how functions and procedures are called in a PL SQL block with simple example.***

### ****How to Call Functions and Procedures in a PL/SQL Block****

In PL/SQL, **functions** and **procedures** are subprograms that encapsulate reusable logic. Procedures perform actions and do not return values, while functions return a value.

### ****Calling a Procedure in a PL/SQL Block****

#### **Procedure Example**

CREATE OR REPLACE PROCEDURE DisplayGreeting (username IN VARCHAR2) IS

BEGIN

DBMS\_OUTPUT.PUT\_LINE('Hello, ' || username || '!');

END;

/

#### **Calling the Procedure**

You can call a procedure within a PL/SQL block directly.

plsql

Copy code

BEGIN

DisplayGreeting('Alice');

END;

/

#### **Output**:

Copy code

Hello, Alice!

1. ***Create a PL/SQL script to display a series of numbers.***

DECLARE

N NUMBER; -- Variable to store the upper limit of the series

BEGIN

-- Prompt user to enter the upper limit

N := &Enter\_Upper\_Limit;

-- Loop to display numbers from 1 to N

FOR i IN 1..N LOOP

DBMS\_OUTPUT.PUT\_LINE('Number: ' || i);

END LOOP;

END;

/

1. ***Write and explain syntax for creating procedure. Give example***

CREATE [OR REPLACE] PROCEDURE procedure\_name

(

parameter\_name [IN | OUT | IN OUT] datatype,

...

)

IS

-- Declaration section

BEGIN

-- Executable section

-- Code to perform the task

EXCEPTION

-- Exception handling section (optional)

END procedure\_name;

/

**Explanation**

1. **CREATE OR REPLACE**:
   * Creates a new procedure or replaces an existing one with the same name.
2. **procedure\_name**:
   * The name of the procedure.
3. **Parameters**:
   * **IN**: Used to pass input values to the procedure (default type).
   * **OUT**: Used to return values to the caller.
   * **IN OUT**: Used to pass values to the procedure and return updated values to the caller.
4. **Declaration Section**:
   * Declares variables, constants, and cursors required within the procedure.
5. **BEGIN-END**:
   * Contains the executable statements.
6. **EXCEPTION**:
   * Handles errors or exceptions that occur during execution.
7. ***Differentiate between SQL and PL/SQL.***

**Definition**

* **SQL**:
  + SQL is a **query language** used for interacting with relational databases. It is a declarative language used to perform operations like querying, inserting, updating, and deleting data.
  + SQL is primarily used for defining and manipulating the data in a database.
* **PL/SQL**:
  + PL/SQL is an **extension of SQL** provided by Oracle. It is a **procedural language** that allows you to write more complex logic using SQL queries combined with variables, loops, conditions, and exception handling.
  + PL/SQL is used for creating stored procedures, functions, triggers, and anonymous blocks, allowing for more advanced programming logic.

**2. Type of Language**

* **SQL**:
  + **Declarative** language.
  + Focuses on **what** to do (describing the operations on the data).
* **PL/SQL**:
  + **Procedural** language.
  + Focuses on **how** to do something (allowing for programming constructs like loops, conditionals, and variables).

**3. Functionality**

* **SQL**:
  + SQL is used to perform **single data operations** like querying (SELECT), inserting (INSERT), updating (UPDATE), and deleting (DELETE) records.
  + SQL operates **on a single statement** at a time.
* **PL/SQL**:
  + PL/SQL allows you to write **complex logic** that can include loops, conditional branches (IF-THEN-ELSE), and exceptions handling (EXCEPTION).
  + PL/SQL can execute **multiple SQL statements** as part of a single block of code.

**4. Execution**

* **SQL**:
  + SQL statements are executed **one at a time**.
  + SQL queries return data sets directly to the user or application.
* **PL/SQL**:
  + PL/SQL is executed as a **block** of code, which may contain multiple SQL statements.
  + PL/SQL can execute SQL queries and return the results, but it is more often used to perform logic and operations that involve multiple SQL statements.

**5. Usage**

* **SQL**:
  + Primarily used for **data manipulation** and **data definition**.
  + Example uses: SELECT queries, creating tables, altering table structure, or defining relationships between tables.
* **PL/SQL**:
  + Used for **procedural programming** within the database.
  + Example uses: Writing stored procedures, triggers, functions, and anonymous blocks of code.

**6. Error Handling**

* **SQL**:
  + SQL has **no built-in error handling**. If a query fails, it returns an error and stops.
* **PL/SQL**:
  + PL/SQL provides **error handling** capabilities with the EXCEPTION block, where you can catch and handle errors gracefully.

**Comparison Table**

| **Feature** | **SQL** | **PL/SQL** |
| --- | --- | --- |
| **Type** | Declarative | Procedural |
| **Purpose** | Data manipulation and querying | Complex logic, stored procedures, functions, triggers |
| **Execution** | Single query statement at a time | Block of statements (multiple SQL + logic) |
| **Error Handling** | No built-in error handling | Supports exception handling |
| **Control Structures** | No loops, conditions, or variables | Supports loops, conditionals, and variables |
| **Usage** | Data retrieval, insertion, updates | Writing procedures, functions, and triggers |
| **Examples** | SELECT, INSERT, UPDATE, DELETE | BEGIN...END, FOR, IF...ELSE |
| **Performance** | Optimized for set-based operations | Suitable for procedural logic, but may incur slight overhead |
| **Complexity** | Simple queries and operations | More complex logic and database programming |
| **Output** | Direct result set or status | Can return values, modify data, or perform actions |

1. ***Write and explain syntax for creating function with parameters. Give example***

In PL/SQL, a **function** is a subprogram that performs a specific task and returns a value. Functions can have parameters that allow you to pass input values into the function for processing. Below is the syntax for creating a function with parameters:

**General Syntax for Creating a Function with Parameters**

CREATE [OR REPLACE] FUNCTION function\_name

(

parameter1\_name parameter1\_type [IN | OUT | IN OUT],

parameter2\_name parameter2\_type [IN | OUT | IN OUT],

...

)

RETURN return\_type

IS

-- Declaration section (optional)

BEGIN

-- Executable section (contains logic to calculate and return result)

RETURN result\_value;

EXCEPTION

-- Exception handling section (optional)

END function\_name;

/

**Explanation of the Syntax**

1. **CREATE [OR REPLACE]**:
   * CREATE: Creates a new function.
   * OR REPLACE: If a function with the same name exists, it is replaced with the new definition.
2. **function\_name**:
   * The name you give to the function. This is how the function will be referenced later.
3. **parameter\_name parameter\_type**:
   * The parameters for the function. Each parameter must have a name and a type (e.g., salary IN NUMBER).
4. **Parameter Modes:**
   * IN: The parameter is **input only** (you provide the value when calling the function).
   * OUT: The parameter is **output only** (it is used to return a value).
   * IN OUT: The parameter can **accept input** and also **return output**.
5. **RETURN return\_type**:
   * Specifies the data type of the value the function will return (e.g., NUMBER, VARCHAR2).
6. **IS**:
   * Indicates the start of the function body where declarations and logic will follow.
7. **Declaration Section** (optional):
   * Variables, constants, and cursors can be declared in this section.
8. **Executable Section**:
   * Contains the logic of the function. This is where you write the code that will calculate or process the result.
9. **RETURN**:
   * This keyword is used to return the result value from the function.
10. **Exception Handling** (optional):

* The EXCEPTION section is used to handle any errors that occur during the execution of the function.

**Example of a Function with Parameters**

Let’s create a simple function that calculates the **area of a rectangle** based on its length and width.

**Function Definition**

CREATE OR REPLACE FUNCTION CalculateRectangleArea (

length IN NUMBER, -- Input parameter: length of the rectangle

width IN NUMBER -- Input parameter: width of the rectangle

)

RETURN NUMBER -- Return type: The calculated area of the rectangle

IS

area NUMBER; -- Local variable to store the calculated area

BEGIN

-- Calculate the area of the rectangle

area := length \* width;

-- Return the calculated area

RETURN area;

EXCEPTION

WHEN OTHERS THEN

-- If any error occurs, return 0 as a default value

RETURN 0;

END CalculateRectangleArea;

/

1. ***In what cursor attributes the outcomes of DML statement execution are saved? Explain***

**Cursor Attributes in PL/SQL**

In PL/SQL, cursor attributes are used to retrieve and store information about the outcome of Data Manipulation Language (DML) statements, such as INSERT, UPDATE, DELETE, and SELECT. These attributes provide useful information about the execution status of a DML operation, such as the number of rows affected.

The following **cursor attributes** are used to capture outcomes of DML statements:

**1. %ROWCOUNT**

* **Definition**: This attribute returns the number of rows affected by the DML operation (e.g., how many rows were inserted, updated, or deleted).
* **Usage**: It is used to check the count of rows that were processed by the cursor.

**Example:**

DECLARE

CURSOR emp\_cursor IS

SELECT employee\_id FROM employees WHERE department\_id = 10;

emp\_id employees.employee\_id%TYPE;

BEGIN

OPEN emp\_cursor;

FETCH emp\_cursor INTO emp\_id;

CLOSE emp\_cursor;

DBMS\_OUTPUT.PUT\_LINE('Number of rows processed: ' || emp\_cursor%ROWCOUNT);

END;

/

**Explanation**: In the example above, %ROWCOUNT would return the number of rows that were fetched from the emp\_cursor.

**2. %FOUND**

* **Definition**: This attribute returns a Boolean value:
  + TRUE if the last DML operation affected at least one row (for INSERT, UPDATE, or DELETE) or if the FETCH operation retrieves at least one row.
  + FALSE if no rows were affected or fetched.
* **Usage**: It helps check whether any rows were processed in the DML statement or retrieved by the FETCH statement.

**Example:**

DECLARE

CURSOR emp\_cursor IS

SELECT employee\_id FROM employees WHERE department\_id = 10;

emp\_id employees.employee\_id%TYPE;

BEGIN

OPEN emp\_cursor;

FETCH emp\_cursor INTO emp\_id;

IF emp\_cursor%FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('A row has been fetched.');

ELSE

DBMS\_OUTPUT.PUT\_LINE('No rows found.');

END IF;

CLOSE emp\_cursor;

END;

/

**Explanation**: In the above example, if the cursor fetches any rows, %FOUND will return TRUE; otherwise, it will return FALSE.

**3. %NOTFOUND**

* **Definition**: This is the opposite of %FOUND. It returns a Boolean value:
  + TRUE if no rows were affected by the DML statement (for INSERT, UPDATE, DELETE) or if the FETCH operation did not retrieve any rows.
  + FALSE if at least one row was affected or retrieved.
* **Usage**: It can be used to check if a DML statement or FETCH operation was unsuccessful.

**Example:**

DECLARE

CURSOR emp\_cursor IS

SELECT employee\_id FROM employees WHERE department\_id = 20;

emp\_id employees.employee\_id%TYPE;

BEGIN

OPEN emp\_cursor;

FETCH emp\_cursor INTO emp\_id;

IF emp\_cursor%NOTFOUND THEN

DBMS\_OUTPUT.PUT\_LINE('No rows fetched.');

ELSE

DBMS\_OUTPUT.PUT\_LINE('A row has been fetched.');

END IF;

CLOSE emp\_cursor;

END;

/

**Explanation**: If no rows are fetched from the cursor, %NOTFOUND will return TRUE; otherwise, it will return FALSE.

**4. %ISOPEN**

* **Definition**: This attribute returns a Boolean value:
  + TRUE if the cursor is currently open.
  + FALSE if the cursor is closed.
* **Usage**: It is often used to check whether a cursor is open before performing operations like FETCH or CLOSE.

**Example:**

DECLARE

CURSOR emp\_cursor IS

SELECT employee\_id FROM employees WHERE department\_id = 10;

emp\_id employees.employee\_id%TYPE;

BEGIN

OPEN emp\_cursor;

IF emp\_cursor%ISOPEN THEN

DBMS\_OUTPUT.PUT\_LINE('Cursor is open.');

ELSE

DBMS\_OUTPUT.PUT\_LINE('Cursor is closed.');

END IF;

CLOSE emp\_cursor;

END;

/

**Explanation**: In the example, %ISOPEN checks if the cursor is open before proceeding with further operations.

1. ***Define cursor and state its types.***

**Summary of Cursor Types**

| **Cursor Type** | **Definition** | **When to Use** | **Example Usage** |
| --- | --- | --- | --- |
| **Implicit Cursor** | Automatically created for single-row SELECT INTO or DML operations. | Used for simple SELECT INTO, INSERT, UPDATE, or DELETE operations. | Handling a single-row query or DML statement without manual cursor management. |
| **Explicit Cursor** | Manually declared and controlled by the programmer to handle multiple rows. | Used when you need to process multiple rows of data with complex logic. | Fetching and processing data row-by-row using loops. |

A **cursor** in PL/SQL is a pointer or a mechanism that allows you to retrieve rows from a database query and process each row individually. It is used to manage the context of a query when dealing with multiple rows of data. Cursors are essential when you need to process result sets row-by-row, especially when the result of a query exceeds a single row.

A cursor holds the SQL statement that retrieves the data from the database, as well as the context for navigating through the result set. In PL/SQL, cursors are typically used in loops to fetch rows and perform actions on each row.

**Types of Cursors**

There are **two main types of cursors** in PL/SQL:

1. **Implicit Cursor**
2. **Explicit Cursor**

Let's look at each type in more detail:

**1. Implicit Cursor**

* **Definition**: An **implicit cursor** is automatically created by Oracle whenever a SELECT INTO statement or any DML operation (such as INSERT, UPDATE, DELETE) is executed.
* **Usage**: You don't need to explicitly define or declare an implicit cursor. Oracle automatically handles it when the SQL statement is executed. These cursors are used for SQL statements that return a single row or when DML operations are performed.
* **Attributes**: Implicit cursors can be accessed using specific attributes:
  + %FOUND: Returns TRUE if the DML statement affected at least one row.
  + %NOTFOUND: Returns TRUE if no rows were affected.
  + %ROWCOUNT: Returns the number of rows affected by the DML statement.
  + %ISOPEN: Always returns FALSE for an implicit cursor (since it is automatically closed after the SQL statement executes).
* **Example**:

DECLARE

emp\_name employees.employee\_name%TYPE;

BEGIN

SELECT employee\_name INTO emp\_name

FROM employees

WHERE employee\_id = 101;

DBMS\_OUTPUT.PUT\_LINE('Employee Name: ' || emp\_name);

END;

/

In this example, the SELECT INTO statement uses an implicit cursor to fetch data.

**2. Explicit Cursor**

* **Definition**: An **explicit cursor** is explicitly declared by the programmer to handle a query that returns multiple rows. Explicit cursors are used when you need to process a result set row by row.
* **Usage**: You must explicitly declare, open, fetch, and close an explicit cursor. This allows greater control over the query and provides more flexibility in handling complex result sets.
* **Lifecycle**:
  + **Declare**: The cursor is declared with the CURSOR keyword.
  + **Open**: The cursor is opened, which executes the SQL query associated with it.
  + **Fetch**: You fetch the rows one by one into variables.
  + **Close**: The cursor is closed after processing the rows.
* **Attributes**: Explicit cursors can use the following attributes:
  + %FOUND: Returns TRUE if the last FETCH retrieved a row.
  + %NOTFOUND: Returns TRUE if the last FETCH did not retrieve a row.
  + %ROWCOUNT: Returns the number of rows fetched so far.
  + %ISOPEN: Returns TRUE if the cursor is open, otherwise FALSE.
* **Example**:

DECLARE

CURSOR emp\_cursor IS

SELECT employee\_id, employee\_name

FROM employees

WHERE department\_id = 10;

emp\_id employees.employee\_id%TYPE;

emp\_name employees.employee\_name%TYPE;

BEGIN

OPEN emp\_cursor;

LOOP

FETCH emp\_cursor INTO emp\_id, emp\_name;

EXIT WHEN emp\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Employee ID: ' || emp\_id || ' Name: ' || emp\_name);

END LOOP;

CLOSE emp\_cursor;

END;

/

In this example:

* The **explicit cursor** emp\_cursor is declared to fetch employees in department 10.
* The cursor is **opened** to execute the SQL query.
* The **FETCH** statement is used to retrieve the rows one by one and print them using DBMS\_OUTPUT.PUT\_LINE.
* The cursor is **closed** after processing all rows.

1. ***Explain implicit and explicit cursors. Give example***

**2. Explicit Cursor**

* **Definition**: An **explicit cursor** is explicitly declared by the programmer to handle a query that returns multiple rows. Explicit cursors are used when you need to process a result set row by row.
* **Usage**: You must explicitly declare, open, fetch, and close an explicit cursor. This allows greater control over the query and provides more flexibility in handling complex result sets.
* **Lifecycle**:
  + **Declare**: The cursor is declared with the CURSOR keyword.
  + **Open**: The cursor is opened, which executes the SQL query associated with it.
  + **Fetch**: You fetch the rows one by one into variables.
  + **Close**: The cursor is closed after processing the rows.
* **Attributes**: Explicit cursors can use the following attributes:
  + %FOUND: Returns TRUE if the last FETCH retrieved a row.
  + %NOTFOUND: Returns TRUE if the last FETCH did not retrieve a row.
  + %ROWCOUNT: Returns the number of rows fetched so far.
  + %ISOPEN: Returns TRUE if the cursor is open, otherwise FALSE.
* **Example**:

DECLARE

CURSOR emp\_cursor IS

SELECT employee\_id, employee\_name

FROM employees

WHERE department\_id = 10;

emp\_id employees.employee\_id%TYPE;

emp\_name employees.employee\_name%TYPE;

BEGIN

OPEN emp\_cursor;

LOOP

FETCH emp\_cursor INTO emp\_id, emp\_name;

EXIT WHEN emp\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Employee ID: ' || emp\_id || ' Name: ' || emp\_name);

END LOOP;

CLOSE emp\_cursor;

END;

/

**1. Implicit Cursor**

* **Definition**: An **implicit cursor** is automatically created by Oracle whenever a SELECT INTO statement or any DML operation (such as INSERT, UPDATE, DELETE) is executed.
* **Usage**: You don't need to explicitly define or declare an implicit cursor. Oracle automatically handles it when the SQL statement is executed. These cursors are used for SQL statements that return a single row or when DML operations are performed.
* **Attributes**: Implicit cursors can be accessed using specific attributes:
  + %FOUND: Returns TRUE if the DML statement affected at least one row.
  + %NOTFOUND: Returns TRUE if no rows were affected.
  + %ROWCOUNT: Returns the number of rows affected by the DML statement.
  + %ISOPEN: Always returns FALSE for an implicit cursor (since it is automatically closed after the SQL statement executes).
* **Example**:

DECLARE

emp\_name employees.employee\_name%TYPE;

BEGIN

SELECT employee\_name INTO emp\_name

FROM employees

WHERE employee\_id = 101;

DBMS\_OUTPUT.PUT\_LINE('Employee Name: ' || emp\_name);

END;

/

In this example, the SELECT INTO statement uses an implicit cursor to fetch data.

1. ***Explain steps of cursor implementation with syntax and example.***

**Steps of Cursor Implementation in PL/SQL**

In PL/SQL, a **cursor** allows you to retrieve data from a query, row by row, and process it in a structured manner. There are two types of cursors in PL/SQL: **implicit cursors** and **explicit cursors**. Here, we will explain the implementation of an **explicit cursor**, which provides more control over the query execution and processing of results.

**Steps to Implement an Explicit Cursor**

To implement a cursor in PL/SQL, you follow these basic steps:

1. **Declare the Cursor**
2. **Open the Cursor**
3. **Fetch Data from the Cursor**
4. **Process the Fetched Data**
5. **Close the Cursor**

**1. Declare the Cursor**

First, you need to declare the cursor to define the SQL query that will retrieve the data. This is done using the CURSOR keyword.

**Syntax:**

CURSOR cursor\_name IS

SELECT\_statement;

* cursor\_name: Name of the cursor.
* SELECT\_statement: The SQL query to retrieve data.

**Example:**

CURSOR emp\_cursor IS

SELECT employee\_id, employee\_name

FROM employees

WHERE department\_id = 10;

**2. Open the Cursor**

After declaring the cursor, you need to open it using the OPEN statement. This causes the query associated with the cursor to be executed and the result set to be created.

**Syntax:**

OPEN cursor\_name;

**Example:**

OPEN emp\_cursor;

**3. Fetch Data from the Cursor**

Once the cursor is opened, you can use the FETCH statement to retrieve rows from the cursor one at a time. You fetch the result of the query into variables that are defined in the PL/SQL block.

**Syntax:**

FETCH cursor\_name INTO variable1, variable2, ...;

* cursor\_name: The name of the cursor.
* variable1, variable2, ...: The variables that will hold the fetched values.

**Example:**

FETCH emp\_cursor INTO emp\_id, emp\_name;

**4. Process the Fetched Data**

After fetching the data, you can process it in any way you like (e.g., print it, update other records, or perform calculations). The fetched data is stored in the variables you defined earlier.

You typically use a **loop** to fetch and process all rows from the cursor. The loop continues until the cursor reaches the end of the result set.

**Syntax (Loop Example):**

LOOP

FETCH cursor\_name INTO variable1, variable2, ...;

EXIT WHEN cursor\_name%NOTFOUND;

-- Process the data here

END LOOP;

* %NOTFOUND: A cursor attribute that returns TRUE if no rows were fetched.

**Example:**

LOOP

FETCH emp\_cursor INTO emp\_id, emp\_name;

EXIT WHEN emp\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Employee ID: ' || emp\_id || ' Name: ' || emp\_name);

END LOOP;

**5. Close the Cursor**

Once all rows are fetched and processed, you should **close** the cursor to release the resources associated with it.

**Syntax:**

CLOSE cursor\_name;

**Example:**

CLOSE emp\_cursor;

**Complete Example of Cursor Implementation**

Now, let's look at a complete example that demonstrates how to declare, open, fetch, process, and close a cursor.

DECLARE

-- Declare cursor

CURSOR emp\_cursor IS

SELECT employee\_id, employee\_name

FROM employees

WHERE department\_id = 10;

-- Declare variables to store the fetched data

emp\_id employees.employee\_id%TYPE;

emp\_name employees.employee\_name%TYPE;

BEGIN

-- Open the cursor

OPEN emp\_cursor;

-- Loop to fetch and process rows

LOOP

FETCH emp\_cursor INTO emp\_id, emp\_name;

EXIT WHEN emp\_cursor%NOTFOUND; -- Exit the loop when no rows are left

-- Process the data

DBMS\_OUTPUT.PUT\_LINE('Employee ID: ' || emp\_id || ' Name: ' || emp\_name);

END LOOP;

-- Close the cursor after processing all rows

CLOSE emp\_cursor;

END;

/

1. ***What do you mean by trigger ? Explain it by a suitable example.***

A **trigger** in PL/SQL is a **named PL/SQL block** that is automatically executed (or fired) in response to a **specific event** that occurs within a database. These events typically involve DML (Data Manipulation Language) operations such as **INSERT**, **UPDATE**, or **DELETE** on a table or view. Triggers allow automatic execution of certain actions (such as data validation, auditing, or logging) whenever the specified event occurs.

**1. Create the employees Table:**

CREATE TABLE employees (

employee\_id NUMBER PRIMARY KEY,

employee\_name VARCHAR2(100),

salary NUMBER

);

**2. Create the audit\_log Table:**

CREATE TABLE audit\_log (

audit\_id NUMBER PRIMARY KEY,

employee\_id NUMBER,

old\_salary NUMBER,

new\_salary NUMBER,

action\_date TIMESTAMP

);

**3. Create the Trigger:**

Now, let’s create a **BEFORE UPDATE trigger** to log the old and new salary values before an update is made to the employees table.

CREATE OR REPLACE TRIGGER salary\_update\_audit

BEFORE UPDATE OF salary ON employees

FOR EACH ROW

BEGIN

-- Insert a record into the audit\_log table

INSERT INTO audit\_log (audit\_id, employee\_id, old\_salary, new\_salary, action\_date)

VALUES (audit\_log\_seq.NEXTVAL, :OLD.employee\_id, :OLD.salary, :NEW.salary, SYSDATE);

END;

/

**Explanation**:

* **BEFORE UPDATE**: The trigger fires **before** the salary column is updated in the employees table.
* **OF salary**: Specifies that the trigger should only fire when the salary column is updated.
* **FOR EACH ROW**: This is a row-level trigger, so it will execute for each row being updated.
* **:OLD**: Refers to the value of the salary before it is updated (i.e., the old salary).
* **:NEW**: Refers to the value of the salary after it is updated (i.e., the new salary).
* The trigger inserts an entry into the audit\_log table, recording the employee\_id, old\_salary, new\_salary, and the action\_date.