

# PL/SQL

*Autor:*

Kevin Cárdenas.

2023

# Índice

|  |           |
|--|-----------|
| <b>1. SQL</b>  | <b>2</b>  |
| 1.1. Keys . . . . .                                      | 2         |
| 1.2. Tipo de Relaciones . . . . .                        | 4         |
| 1.3. Normalización de una Base de Datos . . . . .        | 6         |
| 1.4. Tipos de Operaciones en una Sentencia SQL . . . . . | 11        |
| <b>2. PL/SQL</b>   | <b>14</b> |
| 2.1. Bloques de PL/SQL . . . . .                         | 14        |
| 2.2. Variables y constantes . . . . .                    | 16        |
| 2.3. Strings . . . . .                                   | 19        |
| 2.4. Condicionales y bucles . . . . .                    | 22        |
| 2.5. Matrices (Arrays) . . . . .                         | 27        |
| 2.6. Funciones . . . . .                                 | 30        |
| 2.7. Procedimientos Almacenados en SQL . . . . .         | 32        |
| 2.8. Triggers . . . . .                                  | 34        |

# 1. SQL

SQL (Structured Query Language) es un lenguaje de programación utilizado para administrar y manipular bases de datos relacionales. Es el estándar de facto para interactuar con sistemas de gestión de bases de datos (DBMS) y permite realizar diversas operaciones, como consultas, inserciones, actualizaciones y eliminaciones de datos.

SQL proporciona un conjunto de comandos y sintaxis específicos que permiten interactuar con una base de datos relacional. Estos comandos se dividen en diferentes categorías, entre las que destacan:

- Consultas (Queries): Permite extraer información de la base de datos mediante la especificación de condiciones y criterios de búsqueda.
- Inserciones (Inserts): Permite agregar nuevos registros a una tabla de la base de datos.
- Actualizaciones (Updates): Permite modificar los valores de los registros existentes en una tabla.
- Eliminaciones (Deletes): Permite eliminar registros de una tabla.
- Uniones (Joins): Permite combinar datos de múltiples tablas basándose en una condición de unión.
- Agrupaciones (Group By): Permite agrupar los registros de una tabla según un criterio específico y realizar cálculos agregados.
- Ordenamiento (Order By): Permite ordenar los registros de una tabla según uno o más atributos.

Las consultas (queries) son una parte fundamental de SQL y se utilizan para recuperar información específica de una base de datos. Mediante las consultas, se pueden especificar condiciones de búsqueda, filtros y criterios de ordenamiento para obtener los datos deseados.

## 1.1. Keys

En el contexto de las bases de datos, una base de datos relacional es un conjunto de tablas que están interrelacionadas mediante keys (claves). Las keys desempeñan un papel fundamental en el diseño y la estructura de una base de datos relacional, ya que se utilizan para identificar y relacionar los registros almacenados en las tablas.

Una base de datos relacional se organiza en tablas que contienen filas (registros) y columnas (atributos). Cada registro en una tabla se identifica de forma única mediante una key, que puede ser un atributo o un conjunto de atributos. Las keys permiten establecer relaciones entre las tablas y garantizan la integridad de los datos.

En una base de datos relacional, las keys se utilizan para garantizar la unicidad de los registros y establecer relaciones entre las tablas.

## Primary Key

Una clave primaria (*Primary Key*) es un atributo o conjunto de atributos que identifica de forma única cada registro en una tabla. Es utilizada para garantizar la integridad y unicidad de los datos. Cada tabla en una base de datos relacional debe tener una clave primaria.

Ejemplo de declaración de una clave primaria en SQL:

```
1 CREATE TABLE empleados (  
2 empleado_id INT PRIMARY KEY,  
3 nombre VARCHAR(50),  
4 apellido VARCHAR(50),  
5 ...  
6 );
```

En este ejemplo, la columna `empleado_id` se declara como la clave primaria de la tabla `empleados`.

## Foreign Key

Una clave foránea (*Foreign Key*) es un atributo o conjunto de atributos que establece una relación entre dos tablas en una base de datos relacional. La clave foránea hace referencia a la clave primaria de otra tabla y se utiliza para mantener la integridad referencial entre las tablas.

Ejemplo de declaración de una clave foránea en SQL:

```
1 CREATE TABLE ventas (  
2 venta_id INT PRIMARY KEY,  
3 empleado_id INT,  
4 ...  
5 FOREIGN KEY (empleado_id) REFERENCES empleados(empleado_id)  
6 );
```

En este ejemplo, la columna `empleado_id` en la tabla `ventas` se declara como una clave foránea que hace referencia a la clave primaria `empleado_id` en la tabla `empleados`.

Demos más ejemplos:

## Primary Key

Ejemplo 1: Tabla de estudiantes con clave primaria compuesta:

```

1 CREATE TABLE estudiantes (
2     estudiante_id INT,
3     curso_id INT,
4     nombre VARCHAR(50),
5     PRIMARY KEY (estudiante_id, curso_id)
6 );

```

## Foreign Key

Ejemplo 1: Tabla de órdenes de compra con clave foránea:

```

1 CREATE TABLE ordenes (
2     orden_id INT PRIMARY KEY,
3     cliente_id INT,
4     fecha DATE,
5     FOREIGN KEY (cliente_id) REFERENCES clientes(cliente_id)
6 );

```

Ejemplo 2: Tabla de empleados con clave foránea a sí misma (jerarquía de empleados):

```

1 CREATE TABLE empleados (
2     empleado_id INT PRIMARY KEY,
3     nombre VARCHAR(50),
4     jefe_id INT,
5     FOREIGN KEY (jefe_id) REFERENCES empleados(empleado_id)
6 );

```

## 1.2. Tipo de Relaciones

En SQL, se pueden establecer diferentes tipos de relaciones entre tablas en una base de datos relacional. Algunos de los tipos de relaciones comunes son:

- Relación uno a uno (One-to-One): En esta relación, cada registro en la tabla A está asociado con un único registro en la tabla B, y viceversa. Esto significa que la clave primaria de una tabla se relaciona directamente con la clave primaria de otra tabla. Por ejemplo, en una base de datos de empleados, se puede tener una tabla “Empleados” y una tabla “DetallesEmpleados”, donde cada registro en la tabla “Empleados” tiene una correspondencia única en la tabla “DetallesEmpleados”.
- Relación uno a muchos (One-to-Many): En esta relación, cada registro en la tabla A puede estar asociado con varios registros en la tabla B, pero cada registro en la tabla B está asociado con un

único registro en la tabla A. Esto se logra mediante el uso de una clave primaria en la tabla A que se convierte en una clave foránea en la tabla B. Por ejemplo, en una base de datos de clientes y pedidos, se puede tener una tabla “Clientes” y una tabla “Pedidos”, donde cada cliente puede realizar varios pedidos, pero cada pedido está asociado a un único cliente.

- **Relación muchos a muchos (Many-to-Many):** En esta relación, varios registros en la tabla A pueden estar asociados con varios registros en la tabla B mediante una tabla intermedia. Esta tabla intermedia contiene las claves primarias de ambas tablas como claves foráneas. Por ejemplo, en una base de datos de estudiantes y cursos, se puede tener una tabla “Estudiantes”, una tabla “Cursos” y una tabla intermedia “Inscripciones”, donde varios estudiantes pueden estar inscritos en varios cursos.

Establecer y mantener estas relaciones es fundamental para mantener la integridad y coherencia de los datos en una base de datos relacional. Las claves primarias y foráneas se utilizan para establecer estas relaciones y asegurar que los registros estén correctamente relacionados entre sí.

A continuación, se presentan ejemplos de tablas que ilustran los diferentes tipos de relaciones:

### Relación uno a uno (One-to-One)

Tabla “Empleados”:

| ID | Nombre | Apellido |
|----|--------|----------|
| 1  | John   | Doe      |
| 2  | Jane   | Smith    |

Tabla “DetallesEmpleados”:

| ID | Salario |
|----|---------|
| 1  | 5000    |
| 2  | 6000    |

### Relación uno a muchos (One-to-Many)

Tabla “Clientes”:

| ID | Nombre | Apellido |
|----|--------|----------|
| 1  | John   | Doe      |
| 2  | Jane   | Smith    |

Tabla “Pedidos”:

| ID | ClienteID | Producto |
|----|-----------|----------|
| 1  | 1         | Laptop   |
| 2  | 1         | Teléfono |
| 3  | 2         | Tablet   |

### Relación muchos a muchos (Many-to-Many)

Tabla “Estudiantes”:

| ID | Nombre | Apellido |
|----|--------|----------|
| 1  | John   | Doe      |
| 2  | Jane   | Smith    |

Tabla “Cursos”:

| ID | Nombre      |
|----|-------------|
| 1  | Matemáticas |
| 2  | Historia    |

Tabla “Inscripciones”:

| EstudianteID | CursoID | FechaInscripción |
|--------------|---------|------------------|
| 1            | 1       | 2022-01-01       |
| 1            | 2       | 2022-02-01       |
| 2            | 1       | 2022-03-01       |

## 1.3. Normalización de una Base de Datos

La normalización es un proceso utilizado para organizar y diseñar estructuras de bases de datos relacionales de manera eficiente y sin redundancias innecesarias. Se basa en una serie de reglas (formas normales) que ayudan a eliminar la redundancia y mejorar la integridad de los datos.

Algunas de las formas normales comunes son:

### Primera Forma Normal (1NF)

Requiere que los valores de cada columna en una tabla sean atómicos (indivisibles) y no se repitan. Esto implica dividir una tabla en múltiples tablas más pequeñas y relacionadas, donde cada tabla representa una entidad única. Por ejemplo, consideremos una tabla de empleados donde cada empleado tiene un único ID, nombre y apellido. La 1NF se aseguraría de que cada columna contenga información atómica y que no haya duplicados.

Por ejemplo:

Consideremos una tabla de empleados que contiene información sobre los empleados de una empresa:

| ID | Nombre | Apellido | Departamento |
|----|--------|----------|--------------|
| 1  | John   | Doe      | Ventas       |
| 2  | Jane   | Smith    | Finanzas     |
| 3  | Mark   | Johnson  | Ventas       |

En este ejemplo, la tabla no cumple con la primera forma normal (1NF) porque la columna “Departamento” contiene información repetida. Para aplicar la 1NF, podemos dividir la tabla en dos:

#### **Tabla Empleados:**

| ID | Nombre | Apellido |
|----|--------|----------|
| 1  | John   | Doe      |
| 2  | Jane   | Smith    |
| 3  | Mark   | Johnson  |

#### **Tabla Departamentos:**

| ID | Departamento |
|----|--------------|
| 1  | Ventas       |
| 2  | Finanzas     |

Ahora, cada tabla contiene información única y no hay redundancia en los datos.

### **Segunda Forma Normal (2NF)**

Requiere que la tabla esté en 1NF y que todos los atributos no clave dependan completamente de la clave primaria. En otras palabras, no debe haber dependencias parciales. Esto implica dividir la tabla en múltiples tablas para evitar redundancias y asegurarse de que cada atributo dependa completamente de la clave primaria. Por ejemplo, si tenemos una tabla de ventas donde la clave primaria es una combinación de ID de venta y ID de producto, y tenemos atributos como nombre del producto y precio, la 2NF se aseguraría de que el nombre del producto y el precio dependan completamente de la clave primaria.

Por ejemplo:

Supongamos que tenemos una tabla de pedidos que registra los pedidos realizados por los clientes:

| ID Pedido | ID Cliente | Nombre Cliente | Dirección            |
|-----------|------------|----------------|----------------------|
| 1         | 1001       | John Doe       | Calle Principal 123  |
| 2         | 1002       | Jane Smith     | Calle Secundaria 456 |



En este ejemplo, la tabla no cumple con la segunda forma normal (2NF) porque el nombre del cliente y la dirección dependen de la clave primaria “ID Cliente” y no de la clave primaria “ID Pedido”. Para aplicar la 2NF, podemos dividir la tabla en dos:

**Tabla Pedidos:**

| ID Pedido | ID Cliente |
|-----------|------------|
| 1         | 1001       |
| 2         | 1002       |

**Tabla Clientes:**

| ID Cliente | Nombre Cliente | Dirección            |
|------------|----------------|----------------------|
| 1001       | John Doe       | Calle Principal 123  |
| 1002       | Jane Smith     | Calle Secundaria 456 |

Ahora, el nombre del cliente y la dirección dependen completamente del ID del cliente, evitando así la redundancia de datos.

### Tercera Forma Normal (3NF)

Requiere que la tabla esté en 2NF y que no haya dependencias transitivas, es decir, ningún atributo no clave depende de otro atributo no clave. Esto implica dividir la tabla en múltiples tablas para evitar dependencias transitivas y asegurarse de que cada atributo dependa únicamente de la clave primaria. Por ejemplo, si tenemos una tabla de pedidos donde la clave primaria es el ID del pedido y tenemos atributos como dirección de envío y ciudad, y además tenemos un atributo de código postal que depende de la ciudad, la 3NF se aseguraría de que el atributo de código postal se mueva a una tabla separada que solo contenga información relacionada con las ciudades.

Veamos un ejemplo para ilustrar la Tercera Forma Normal (3NF):

Supongamos que tenemos una tabla de pedidos donde la clave primaria es el ID del pedido. Además, tenemos atributos como dirección de envío, ciudad y código postal. Para cumplir con la 3NF, debemos asegurarnos de que ningún atributo no clave dependa de otro atributo no clave. En este caso, el atributo de código postal depende de la ciudad, lo cual genera una dependencia transitiva. Para resolver esto, podemos dividir la tabla en dos:

**Tabla Pedidos:**

| ID Pedido | Dirección de envío   | ID Ciudad |
|-----------|----------------------|-----------|
| 1         | Calle Principal 123  | 1         |
| 2         | Calle Secundaria 456 | 2         |

**Tabla Ciudades:**

| ID Ciudad | Ciudad   | Código Postal |
|-----------|----------|---------------|
| 1         | Ciudad A | 12345         |
| 2         | Ciudad B | 67890         |

En este ejemplo, hemos separado el atributo de código postal en una tabla separada llamada “Ciudades”. Ahora, la tabla de pedidos solo contiene información relacionada con los pedidos, mientras que la tabla de ciudades contiene información relacionada con las ciudades y sus códigos postales. Esto cumple con la Tercera Forma Normal (3NF) al eliminar la dependencia transitiva y asegurarnos de que cada atributo dependa únicamente de la clave primaria correspondiente.

**Cuarta Forma Normal (4NF)**

La Cuarta Forma Normal (4NF) se refiere a una forma de normalización más avanzada que aborda la redundancia causada por dependencias multivaluadas en una base de datos relacional.

En la 4NF, se busca eliminar las dependencias multivaluadas, es decir, las dependencias en las que un conjunto de atributos determina múltiples conjuntos de valores para otros atributos.

Un ejemplo sencillo de la 4NF es el siguiente:

Consideremos una tabla “Productos” con los siguientes atributos:

| ID Producto | Nombre Producto | Categorías               |
|-------------|-----------------|--------------------------|
| 1           | Laptop          | Electrónica, Informática |
| 2           | Televisor       | Electrónica, Hogar       |

En este ejemplo, la columna “Categorías” es multivaluada, lo que significa que un producto puede tener múltiples categorías asociadas. Para aplicar la 4NF, podemos crear una nueva tabla “Categorías” y utilizar una clave foránea para establecer la relación:

**Tabla Productos:**

| ID Producto | Nombre Producto |
|-------------|-----------------|
| 1           | Laptop          |
| 2           | Televisor       |

**Tabla Categorías:**

| ID Producto | Categoría   |
|-------------|-------------|
| 1           | Electrónica |
| 1           | Informática |
| 2           | Electrónica |
| 2           | Hogar       |

De esta manera, hemos eliminado la redundancia en la columna “Categorías” y hemos establecido una relación adecuada entre los productos y sus categorías.

### Quinta Forma Normal (5NF)

La Quinta Forma Normal (5NF) es una forma de normalización aún más avanzada que aborda las dependencias de unión, también conocidas como dependencias de proyección y unión.

En la 5NF, se busca eliminar las dependencias de unión al descomponer las relaciones en tablas más pequeñas y más especializadas.

Un ejemplo de la 5NF es el siguiente:

Consideremos una tabla “Pedidos” con los siguientes atributos:

| ID Pedido | Productos         |
|-----------|-------------------|
| 1         | Laptop, Televisor |
| 2         | Teléfono, Tablet  |

En este ejemplo, la columna “Productos” tiene una dependencia de unión, ya que contiene múltiples valores separados por comas. Para aplicar la 5NF, podemos crear una nueva tabla “ProductosPedidos” para representar las relaciones entre los productos y los pedidos:

**Tabla Pedidos:**

| ID Pedido |
|-----------|
| 1         |
| 2         |

**Tabla ProductosPedidos:**

| ID Pedido | Producto  |
|-----------|-----------|
| 1         | Laptop    |
| 1         | Televisor |
| 2         | Teléfono  |
| 2         | Tablet    |

De esta manera, hemos eliminado la dependencia de unión en la columna “Productos” y hemos descompuesto la relación en dos tablas más especializadas.

La Cuarta Forma Normal (4NF) y la Quinta Forma Normal (5NF) son formas normales más avanzadas que se utilizan para abordar situaciones más complejas y garantizar una mayor eficiencia y estructura en las bases de datos relacionales.

## 1.4. Tipos de Operaciones en una Sentencia SQL

SQL admite una amplia gama de operaciones para interactuar con una base de datos. Algunas de las operaciones más comunes en SQL son:

- **SELECT**: Utilizado para recuperar datos de una o varias tablas.
- **INSERT**: Utilizado para insertar nuevos registros en una tabla.
- **UPDATE**: Utilizado para actualizar registros existentes en una tabla.
- **DELETE**: Utilizado para eliminar registros de una tabla.
- **JOIN**: Utilizado para combinar datos de dos o más tablas basándose en una condición de unión.
- **GROUP BY**: Utilizado para agrupar filas según un criterio y realizar cálculos agregados en los grupos resultantes.
- **ORDER BY**: Utilizado para ordenar los resultados de una consulta según uno o más atributos.

### SELECT

La operación **SELECT** se utiliza para recuperar datos de una o varias tablas en una base de datos. Permite especificar qué columnas se desean seleccionar, así como aplicar condiciones y criterios de filtrado para obtener resultados más específicos.

Un ejemplo de uso de la operación **SELECT** sería:

```
1 SELECT nombre, apellido FROM empleados WHERE edad > 30;
```

Esta consulta selecciona los nombres y apellidos de los empleados cuya edad es mayor a 30.

### INSERT

La operación **INSERT** se utiliza para insertar nuevos registros en una tabla. Permite especificar los valores a ser insertados en cada columna de la tabla.

Un ejemplo de uso de la operación **INSERT** sería:

```
1 INSERT INTO clientes (nombre, apellido, correo) VALUES ('Juan', 'Perez', 'juan@example.com');
```

Esta consulta inserta un nuevo cliente en la tabla “clientes” con los valores especificados para nombre, apellido y correo.

## UPDATE

La operación UPDATE se utiliza para actualizar registros existentes en una tabla. Permite modificar los valores de una o varias columnas en base a una condición.

Un ejemplo de uso de la operación UPDATE sería:

```
1 UPDATE productos SET precio = 10.99 WHERE categoria = 'Electronica';
```

Esta consulta actualiza el precio de todos los productos de la categoría 'Electrónica' a 10.99.

## DELETE

La operación DELETE se utiliza para eliminar registros de una tabla. Permite especificar una condición para determinar qué registros deben ser eliminados.

Un ejemplo de uso de la operación DELETE sería:

```
1 DELETE FROM pedidos WHERE fecha < '2022-01-01';
```

Esta consulta elimina todos los pedidos cuya fecha sea anterior al 1 de enero de 2022.

## JOIN

La operación JOIN se utiliza para combinar datos de dos o más tablas en base a una condición de unión. Permite obtener resultados que relacionen registros de diferentes tablas.

Un ejemplo de uso de la operación JOIN sería:

```
1 SELECT empleados.nombre, departamentos.nombre
2 FROM empleados
3 JOIN departamentos ON empleados.departamento_id = departamentos.id;
```

Esta consulta combina los datos de las tablas “empleados” y “departamentos” utilizando la condición de unión “empleados.departamento\_id = departamentos.id”, y selecciona los nombres de los empleados junto con los nombres de los departamentos a los que pertenecen.

## GROUP BY

La operación GROUP BY se utiliza para agrupar filas según un criterio y realizar cálculos agregados en los grupos resultantes. Permite obtener resultados resumidos y estadísticas sobre los datos de una tabla.

Un ejemplo de uso de la operación GROUP BY sería:

```
1 SELECT categoria, COUNT(*) as total_productos
2 FROM productos
3 GROUP BY categoria;
```

Esta consulta agrupa los productos por categoría y cuenta cuántos productos hay en cada categoría, mostrando el resultado en la columna “total\_productos”.

## ORDER BY

La operación ORDER BY se utiliza para ordenar los resultados de una consulta según uno o más atributos. Permite especificar el orden ascendente (ASC) o descendente (DESC) de los resultados.

Un ejemplo de uso de la operación ORDER BY sería:

```
1 SELECT nombre, precio
2 FROM productos
3 ORDER BY precio DESC;
```

Esta consulta selecciona los nombres y precios de los productos y los ordena de forma descendente según el precio.

## 2. PL/SQL

PL/SQL (Procedural Language/Structured Query Language) es un lenguaje de programación procedural desarrollado por Oracle Corporation para su uso con el sistema de gestión de bases de datos Oracle. Fue creado en la década de 1990 como una extensión del lenguaje SQL estándar, con el objetivo de proporcionar capacidades avanzadas de programación y lógica empresarial en el contexto de una base de datos relacional.

PL/SQL combina elementos de lenguajes de programación procedurales tradicionales, como variables, estructuras de control de flujo y subrutinas, con la potencia del lenguaje SQL para interactuar con la base de datos. Esto permite a los desarrolladores crear aplicaciones más complejas y sofisticadas que pueden aprovechar todas las capacidades de una base de datos relacional.

Una de las principales ventajas de PL/SQL es su estrecha integración con Oracle Database. Los programas PL/SQL se ejecutan directamente en el servidor de la base de datos, lo que reduce la necesidad de enviar múltiples consultas desde una aplicación cliente y minimiza la cantidad de datos transferidos a través de la red. Esto mejora significativamente el rendimiento y la eficiencia de las aplicaciones, especialmente en entornos empresariales donde el acceso a la base de datos es fundamental.

PL/SQL se utiliza ampliamente para desarrollar funciones, procedimientos almacenados, desencadenadores (triggers) y paquetes, que encapsulan la lógica de negocio y proporcionan una capa de abstracción adicional sobre los datos almacenados en la base de datos. Estas construcciones permiten una mejor organización y modularidad del código, promoviendo la reutilización y el mantenimiento eficiente de la lógica empresarial.

### 2.1. Bloques de PL/SQL

Un bloque de PL/SQL es una unidad básica de código en PL/SQL. Puede contener declaraciones, sentencias SQL y lógica de programación. Los bloques de PL/SQL se utilizan para encapsular la lógica de negocio y pueden ser anónimos o nombrados.

Los bloques anónimos se ejecutan de forma inmediata, mientras que los bloques nombrados se almacenan en la base de datos y se pueden invocar desde otras partes del sistema.

A continuación, se muestra un ejemplo de un bloque de PL/SQL anónimo:

```
1 SET SERVEROUTPUT ON --Imprimir mensajes en consola
2
3 DECLARE
4     nombre VARCHAR2(50) := 'Juan';
```

```

5      edad NUMBER := 30;
6 BEGIN
7     -- Logic of program
8     IF edad >= 18 THEN
9         DBMS_OUTPUT.PUT_LINE(nombre || ' es mayor de edad');
10    ELSE
11        DBMS_OUTPUT.PUT_LINE(nombre || ' es menor de edad');
12    END IF;
13 END;
14
15 -- Sentencias SQL
16 INSERT INTO empleados (nombre, salario)
17     VALUES ('Ana', 5000);
18
19 COMMIT;
20 END;
21 /

```

En este ejemplo, el bloque de PL/SQL anónimo comienza con la palabra clave “BEGIN” y finaliza con “/” para indicar el final del bloque. Dentro del bloque, se pueden realizar declaraciones como la declaración de variables y constantes. Además, se puede incluir lógica de programación como condicionales y sentencias SQL para manipular la base de datos.

Por otro lado, los bloques de PL/SQL nombrados se almacenan en la base de datos y se pueden invocar desde otras partes del sistema. A continuación se muestra un ejemplo de un procedimiento almacenado, que es un tipo de bloque de PL/SQL nombrado:

```

1 CREATE OR REPLACE PROCEDURE calcular_salario(p_empleado_id NUMBER) AS
2     v_salario NUMBER;
3 BEGIN
4     -- Logica para calcular el salario del empleado
5     SELECT salario INTO v_salario
6     FROM empleados
7     WHERE empleado_id = p_empleado_id;
8
9     DBMS_OUTPUT.PUT_LINE('El salario del empleado ' || p_empleado_id || ' es: ' || v_salario);
10 END;
11 /

```

En este ejemplo, se crea un procedimiento almacenado llamado “calcular\_salario” que acepta un parámetro de empleado\_id. Dentro del procedimiento, se realiza una consulta para obtener el salario del



empleado correspondiente al empleado\_id proporcionado. Luego, se muestra el salario utilizando la función DBMS\_OUTPUT.PUT\_LINE.

Además de los bloques anónimos y los procedimientos almacenados, PL/SQL también ofrece la posibilidad de crear disparadores (triggers). Los disparadores son bloques de PL/SQL que se ejecutan automáticamente en respuesta a eventos específicos que ocurren en la base de datos, como la inserción, actualización o eliminación de datos en una tabla.

A continuación se muestra un ejemplo de un disparador (trigger) que se activa después de insertar una nueva fila en la tabla “empleados”:

```
1 CREATE OR REPLACE TRIGGER insertar_empleado_trigger
2 AFTER INSERT ON empleados
3 FOR EACH ROW
4 BEGIN
5     DBMS_OUTPUT.PUT_LINE('Nuevo empleado insertado: ' || :NEW.nombre);
6 END;
7 /
```

En este ejemplo, el disparador “insertar\_empleado\_trigger” se ejecuta después de cada inserción en la tabla “empleados”. El bloque de PL/SQL dentro del disparador muestra un mensaje que indica el nombre del nuevo empleado que se ha insertado.

Los disparadores son una poderosa herramienta en PL/SQL que permiten automatizar acciones y aplicar lógica adicional en la base de datos en respuesta a eventos específicos.

## 2.2. Variables y constantes

En PL/SQL, se pueden declarar variables y constantes para almacenar y manipular datos. Las variables se utilizan para almacenar valores temporales y pueden cambiar durante la ejecución del programa. Las constantes, por otro lado, son valores fijos que no pueden modificarse una vez que se les ha asignado un valor. Tanto las variables como las constantes pueden tener diferentes tipos de datos, como enteros, caracteres, fechas, etc.

Algunos de los tipos de datos más comunes en PL/SQL. Cada tipo de dato tiene características específicas y se debe elegir según el tipo de datos que se desea almacenar y manipular en la base de datos.

- **VARCHAR2**: se utiliza para almacenar cadenas de caracteres.

```
1 DECLARE
2     nombre VARCHAR2(50) := 'Juan';
3 BEGIN
```

```

4 DBMS_OUTPUT.PUT_LINE('Nombre: ' || nombre);
5 END;

```

- **NUMBER**: se utiliza para almacenar números enteros o decimales.

```

1 DECLARE
2     sueldo NUMBER(10, 2) := 5000.50;
3     extras NUMBER(10, 2) := 200.50;
4 BEGIN
5     DBMS_OUTPUT.PUT_LINE('Saldo: ' || TO_CHAR(sueldo + extras));
6 END;

```

- **DATE**: se utiliza para almacenar fechas y horas.

```

1 DECLARE
2     fecha_nacimiento DATE := TO_DATE('1990/01/01', 'YYYY/MM/DD');
3 BEGIN
4     DBMS_OUTPUT.PUT_LINE('Fecha de nacimiento: ' || TO_CHAR(fecha_nacimiento, 'DD/MM/YYYY'));
5 END;

```

- **BOOLEAN**: se utiliza para almacenar valores de verdadero o falso.

```

1 DECLARE
2     es_mayor BOOLEAN := TRUE;
3 BEGIN
4     IF es_mayor THEN
5         DBMS_OUTPUT.PUT_LINE('Es mayor de edad');
6     ELSE
7         DBMS_OUTPUT.PUT_LINE('Es menor de edad');
8     END IF;
9 END;

```

- **CHAR**: se utiliza para almacenar caracteres de longitud fija.

```

1 DECLARE
2     inicial CHAR(1) := 'A';
3 BEGIN
4     DBMS_OUTPUT.PUT_LINE('Inicial: ' || inicial);
5 END;

```

- **LONG**: se utiliza para almacenar cadenas de longitud variable.

```

1 DECLARE
2     descripcion LONG := 'Esta es una descripcion larga';
3 BEGIN
4     DBMS_OUTPUT.PUT_LINE('Descripcion: ' || descripcion);
5 END;

```

- **RAW**: se utiliza para almacenar datos binarios.

```

1 DECLARE
2     datos RAW(100) := UTL_RAW.CAST_TO_RAW('010101');
3 BEGIN
4     DBMS_OUTPUT.PUT_LINE('Datos: ' || UTL_RAW.CAST_TO_VARCHAR2(datos));
5 END;

```

- **BLOB**: se utiliza para almacenar datos binarios grandes.

```

1 DECLARE
2     es_mayor BOOLEAN := FALSE;
3     edad INTEGER := 18;
4 BEGIN
5     es_mayor := (edad >= 18);
6     IF es_mayor THEN
7         DBMS_OUTPUT.PUT_LINE('Es mayor de edad');
8     ELSE
9         DBMS_OUTPUT.PUT_LINE('Es menor de edad');
10    END IF;
11 END;

```

- **CLOB**: se utiliza para almacenar cadenas de caracteres grandes.

```

1 DECLARE
2     descripcion CLOB := 'Esta es una descripcion larga';
3 BEGIN
4     DBMS_OUTPUT.PUT_LINE('Descripcion: ' || descripcion);
5 END;

```

- **TIMESTAMP**: se utiliza para almacenar fechas y horas con precisión de fracciones de segundo.

```

1 DECLARE
2     fecha_hora TIMESTAMP := SYSTIMESTAMP;
3 BEGIN
4     DBMS_OUTPUT.PUT_LINE('Fecha y hora: ' || TO_CHAR(fecha_hora, 'DD/MM/YYYY HH24:MI:SS.FF'));

```

- **INTERVAL**: se utiliza para almacenar intervalos de tiempo.

```
1 DECLARE
2     duracion INTERVAL DAY TO SECOND := INTERVAL '3' HOUR;
3 BEGIN
4     DBMS_OUTPUT.PUT_LINE('Duracion: ' || duracion);
5 END;
```

- **RECORD**: se utiliza para almacenar un conjunto de valores relacionados.

```
1 DECLARE
2     TYPE tipo_empleado IS RECORD (
3         nombre VARCHAR2(50),
4         edad  NUMBER,
5         sueldo NUMBER(10, 2)
6     );
7
8     empleado tipo_empleado;
9 BEGIN
10    empleado.nombre := 'Juan';
11    empleado.edad := 30;
12    empleado.sueldo := 5000.50;
13
14    DBMS_OUTPUT.PUT_LINE('Nombre: ' || empleado.nombre);
15    DBMS_OUTPUT.PUT_LINE('Edad: ' || empleado.edad);
16    DBMS_OUTPUT.PUT_LINE('Sueldo: ' || empleado.sueldo);
17 END;
```

- **TABLE**: se utiliza para almacenar conjuntos de datos en forma de tabla.

```
1 DECLARE
2     TYPE tipo_empleados IS TABLE OF VARCHAR2(50);
3     empleados tipo_empleados := tipo_empleados('Juan', 'Maria', 'Pedro');
4 BEGIN
5     FOR i IN 1..empleados.COUNT LOOP
6         DBMS_OUTPUT.PUT_LINE('Empleado ' || i || ': ' || empleados(i));
7     END LOOP;
8 END;
```

## 2.3. Strings

En PL/SQL, los strings se representan utilizando el tipo de dato `VARCHAR2` o `CHAR`. Los strings son utilizados para almacenar y manipular datos textuales. A continuación se presentan algunos métodos y

operaciones comunes para trabajar con strings en PL/SQL:

**Declaración y asignación de strings** Se pueden declarar variables de tipo string utilizando el tipo de dato VARCHAR2. Luego, se pueden asignar valores utilizando el operador de asignación :=. Por ejemplo:

```
1 DECLARE
2     nombre VARCHAR2(50);
3     direccion VARCHAR2(100);
4 BEGIN
5     nombre := 'Juan';
6     direccion := 'Calle Principal';
7     -- Resto del codigo...
8 END;
```

**Concatenación de strings** La concatenación de strings se puede realizar utilizando el operador de concatenación ||. Por ejemplo:

```
1 DECLARE
2     nombre VARCHAR2(50) := 'Juan';
3     apellido VARCHAR2(50) := 'Perez';
4     nombre_completo VARCHAR2(100);
5 BEGIN
6     nombre_completo := nombre || ' ' || apellido;
7     dbms_output.put_line('Nombre completo: ' || nombre_completo);
8     -- Resto del codigo...
9 END;
```

**Funciones y operadores de manipulación de strings** PL/SQL proporciona varias funciones y operadores para manipular strings. Aquí se presentan algunos ejemplos:

- SUBSTR(cadena, inicio, longitud): Retorna una subcadena de una cadena dada. Ejemplo:

```
1 DECLARE
2     cadena VARCHAR2(50) := 'Hello, World!';
3     subcadena VARCHAR2(20);
4 BEGIN
5     subcadena := SUBSTR(cadena, 1, 5);
6     dbms_output.put_line('Subcadena: ' || subcadena);
7     -- Resto del codigo...
8 END;
```

- **LENGTH(cadena)**: Retorna la longitud de una cadena. Ejemplo:

```
1 DECLARE
2     cadena VARCHAR2(50) := 'Hello, World!';
3     longitud NUMBER;
4 BEGIN
5     longitud := LENGTH(cadena);
6     dbms_output.put_line('Longitud: ' || longitud);
7     -- Resto del codigo...
8 END;
```

- **UPPER(cadena)**: Convierte una cadena a mayúsculas. Ejemplo:

```
1 DECLARE
2     cadena VARCHAR2(50) := 'Hello, World!';
3     cadena_mayusculas VARCHAR2(50);
4 BEGIN
5     cadena_mayusculas := UPPER(cadena);
6     dbms_output.put_line('Cadena en mayusculas: ' || cadena_mayusculas);
7     -- Resto del codigo...
8 END;
```

- **LOWER(cadena)**: Convierte una cadena a minúsculas. Ejemplo:

```
1 DECLARE
2     cadena VARCHAR2(50) := 'Hello, World!';
3     cadena_minusculas VARCHAR2(50);
4 BEGIN
5     cadena_minusculas := LOWER(cadena);
6     dbms_output.put_line('Cadena en minusculas: ' || cadena_minusculas);
7     -- Resto del codigo...
8 END;
```

- **INSTR(cadena, subcadena)**: Encuentra la posición de una subcadena dentro de una cadena. Ejemplo:

```
1 DECLARE
2     cadena VARCHAR2(50) := 'Hello, World!';
3     posicion NUMBER;
4 BEGIN
5     posicion := INSTR(cadena, 'World');
6     dbms_output.put_line('Posicion de "World": ' || posicion);
7     -- Resto del codigo...
8 END;
```

- **REPLACE(cadena, subcadena, nueva\_subcadena)**: Reemplaza todas las ocurrencias de una subcadena por otra en una cadena. Ejemplo:

```

1 DECLARE
2     cadena VARCHAR2(50) := 'Hello, World!';
3     nueva_cadena VARCHAR2(50);
4 BEGIN
5     nueva_cadena := REPLACE(cadena, 'World', 'Mundo');
6     dbms_output.put_line('Cadena modificada: ' || nueva_cadena);
7     -- Resto del codigo...
8 END;
```

- **TRIM(cadena), LTRIM(cadena), RTRIM(cadena)**: Estas funciones se utilizan para eliminar caracteres de una cadena, a la izquierda o derecha respectivamente. Ejemplo:

```

1 DECLARE
2     cadena VARCHAR2(50) := '  Hola, Mundo!  ';
3     cadena_sin_espacios VARCHAR2(50);
4 BEGIN
5     cadena_sin_espacios := TRIM(cadena);
6     dbms_output.put_line('Cadena sin espacios: ' || cadena_sin_espacios);
7     -- Resto del codigo...
8 END;
```

## 2.4. Condicionales y bucles

En PL/SQL, los condicionales y los bucles son fundamentales para controlar el flujo de ejecución de un programa. Permiten tomar decisiones y repetir tareas de manera eficiente.

### Condicionales

En PL/SQL, se utiliza la estructura **IF-THEN-ELSE** para evaluar una condición y ejecutar diferentes bloques de código según el resultado. A continuación se muestra un ejemplo:

```

1 DECLARE
2     edad NUMBER := 18;
3 BEGIN
4     IF edad >= 18 THEN
5         dbms_output.put_line('Eres mayor de edad');
6     ELSE
7         dbms_output.put_line('Eres menor de edad');
```

```

8  END IF;
9  END;

```

En este ejemplo, se evalúa la variable `edad` y se muestra un mensaje según el resultado.

## Condicional CASE

El condicional **CASE** en PL/SQL es una estructura de control que permite evaluar una expresión y tomar decisiones basadas en su valor. Proporciona una alternativa a la estructura IF-THEN-ELSE y es especialmente útil cuando se tienen múltiples condiciones a evaluar.

La sintaxis básica del condicional **CASE** es la siguiente:

```

1  CASE expression
2      WHEN valor1 THEN
3          -- Codigo a ejecutar cuando la expresion es igual a valor1
4      WHEN valor2 THEN
5          -- Codigo a ejecutar cuando la expresion es igual a valor2
6          ...
7      ELSE
8          -- Codigo a ejecutar cuando la expresion no coincide con ninguno de los valores anteriores
9  END CASE;

```

En este caso, `expression` es la expresión que se evalúa y `valor1`, `valor2`, etc., son los valores posibles que se comparan con la expresión. El bloque de código correspondiente se ejecutará cuando la expresión coincida con uno de los valores especificados.

A continuación, se muestra un ejemplo de uso del condicional **CASE** en PL/SQL:

```

1  DECLARE
2      nota NUMBER := 80;
3  BEGIN
4      CASE nota
5          WHEN 90 THEN
6              DBMS_OUTPUT.PUT_LINE('Excelente');
7          WHEN 80 THEN
8              DBMS_OUTPUT.PUT_LINE('Bueno');
9          WHEN 70 THEN
10                 DBMS_OUTPUT.PUT_LINE('Aceptable');
11         ELSE
12             DBMS_OUTPUT.PUT_LINE('Reprobado');
13     END CASE;
14 END;

```



En este ejemplo, la variable `nota` se evalúa y se ejecuta el código correspondiente según el valor de la nota. Si la nota es 90, se mostrará “Excelente”; si es 80, se mostrará “Bueno”; si es 70, se mostrará “Aceptable”; y en cualquier otro caso, se mostrará “Reprobado”.

El condicional CASE también permite el uso de condiciones adicionales utilizando la cláusula `WHEN-THEN`:

```
1 DECLARE
2     nota NUMBER := 75;
3 BEGIN
4     CASE
5         WHEN nota >= 90 THEN
6             DBMS_OUTPUT.PUT_LINE('Excelente');
7         WHEN nota >= 80 THEN
8             DBMS_OUTPUT.PUT_LINE('Bueno');
9         WHEN nota >= 70 THEN
10            DBMS_OUTPUT.PUT_LINE('Aceptable');
11        ELSE
12            DBMS_OUTPUT.PUT_LINE('Reprobado');
13    END CASE;
14 END;
```

En este caso, no se especifica una expresión en el `CASE`, pero se utilizan condiciones para evaluar la variable `nota` y ejecutar el código correspondiente.

## Bucles

En PL/SQL, los bucles permiten repetir tareas y controlar el flujo de ejecución. Veamos algunos tipos de bucles comunes:

### Bucle LOOP

El bucle `LOOP` es un bucle infinito que se repite hasta que se encuentra una instrucción de salida. Puedes utilizar la instrucción `EXIT` para salir del bucle. Aquí tienes un ejemplo:

```
1 DECLARE
2     contador NUMBER := 1;
3 BEGIN
4     LOOP
5         dbms_output.put_line('Contador: ' || contador);
6         contador := contador + 1;
7
8         IF contador > 5 THEN
```

```

9      EXIT; -- Sale del bucle
10
11      END IF;
12  END LOOP;
13 END;
```

En este ejemplo, se utiliza un bucle LOOP para mostrar el valor del contador y se sale del bucle cuando el contador supera 5.

## Bucle WHILE

El bucle WHILE se repite mientras se cumpla una condición especificada. El bucle sigue ejecutándose siempre que la condición sea verdadera. Aquí tienes un ejemplo:

```

1 DECLARE
2     contador NUMBER := 1;
3 BEGIN
4     WHILE contador <= 5 LOOP
5         dbms_output.put_line('Contador: ' || contador);
6         contador := contador + 1;
7     END LOOP;
8 END;
```

En este ejemplo, se utiliza un bucle WHILE para mostrar el valor del contador y se repite mientras el contador sea menor o igual a 5.

## Bucle FOR

El bucle FOR se utiliza para recorrer un conjunto de valores en un rango o una lista. Puedes especificar un rango de valores utilizando INICIO..FIN o proporcionar una lista separada por comas de valores. Aquí tienes un ejemplo:

```

1 DECLARE
2     total NUMBER := 0;
3 BEGIN
4     FOR i IN 1..5 LOOP
5         total := total + i;
6     END LOOP;
7
8     dbms_output.put_line('Total: ' || total);
9 END;
```

En este ejemplo, se utiliza un bucle **FOR** para sumar los números del 1 al 5 y se muestra el resultado final.

La principal diferencia entre los bucles **LOOP** y **WHILE** radica en cómo se controla la condición de finalización: en el caso del bucle **LOOP**, se controla internamente dentro del bucle, mientras que en el bucle **WHILE**, se evalúa antes de cada iteración. El bucle **FOR** se utiliza específicamente para iterar sobre colecciones de elementos, como cursores o conjuntos de registros.

Es importante tener cuidado con los bucles que no paran. Si un bucle no tiene una instrucción de salida o una condición de finalización que se cumpla, se producirá lo que se conoce como un “bucle infinito”. En ese caso, el bucle se ejecutará continuamente sin detenerse, lo que puede llevar a problemas como un uso excesivo de recursos del sistema y un bloqueo del programa. Los bucles infinitos son un error común en la programación y deben evitarse. Si accidentalmente creas un bucle infinito, es posible que debas detener la ejecución del programa manualmente o reiniciar el entorno de ejecución

En PL/SQL, es posible anidar condicionales y bucles, lo que permite realizar estructuras de control más complejas y flexibles. A continuación, se presentan ejemplos prácticos de condicionales y bucles anidados:

## Condicionales anidados

Los condicionales anidados permiten evaluar múltiples condiciones y ejecutar diferentes bloques de código en función de los resultados. Aquí tienes un ejemplo que determina la calificación de un estudiante en base a su nota:

```
1 DECLARE
2     nota NUMBER := 85;
3 BEGIN
4     IF nota >= 90 THEN
5         dbms_output.put_line('Calificacion: A');
6     ELSIF nota >= 80 THEN
7         dbms_output.put_line('Calificacion: B');
8     ELSIF nota >= 70 THEN
9         dbms_output.put_line('Calificacion: C');
10    ELSE
11        dbms_output.put_line('Calificacion: D');
12    END IF;
13    -- Resto del codigo...
14 END;
15 /
```

En este ejemplo, se evalúa la nota del estudiante y se imprime la calificación correspondiente utilizando condicionales anidados.

## Bucles anidados

Los bucles anidados permiten iterar sobre múltiples conjuntos de datos. Aquí tienes un ejemplo de un bucle FOR anidado que genera una tabla de multiplicación:

```
1 DECLARE
2     limite_filas NUMBER := 5;
3     limite_columnas NUMBER := 5;
4 BEGIN
5     FOR i IN 1..limite_filas LOOP
6         FOR j IN 1..limite_columnas LOOP
7             dbms_output.put(i*j || ' ');
8         END LOOP;
9         dbms_output.new_line;
10    END LOOP;
11    -- Resto del codigo...
12 END;
13 /
```

En este ejemplo, se utiliza un bucle FOR anidado para generar una tabla de multiplicación con el límite de filas y columnas especificado.

Los condicionales y bucles anidados brindan flexibilidad para controlar el flujo de ejecución en situaciones más complejas. Puedes combinarlos y ajustarlos según tus necesidades para lograr la lógica deseada en tu programa.

Recuerda que estos son solo ejemplos básicos, y puedes adaptarlos y expandirlos según tus requerimientos específicos.

## 2.5. Matrices (Arrays)

En PL/SQL, las matrices se pueden representar utilizando tipos de datos de matriz, como **VARRAY** (Variable Array). Una matriz, también conocida como vector, es una estructura de datos unidimensional que puede contener un conjunto de elementos del mismo tipo.

### Introducción a las matrices

Para declarar una matriz en PL/SQL, se utiliza la siguiente sintaxis:

```

1 DECLARE
2     TYPE matriz_t IS VARRAY(n) OF tipo_dato;
3     matriz matriz_t := matriz_t(elemento1, elemento2, ..., elementon);
4 BEGIN
5     -- Resto del codigo
6 END;
```

Donde:

- **matriz\_t**: es el tipo de dato de la matriz.
- **n**: es el tamaño máximo de la matriz (número de elementos).
- **tipo\_dato**: es el tipo de dato de los elementos de la matriz.
- **matriz**: es el nombre de la matriz.
- **elemento1, elemento2, ..., elementon**: son los elementos que se asignarán a la matriz durante la inicialización.

A continuación, se muestra un ejemplo de declaración e inicialización de una matriz de números en PL/SQL:

```

1 DECLARE
2     TYPE matriz_numeros IS VARRAY(5) OF NUMBER;
3     numeros matriz_numeros := matriz_numeros(1, 2, 3, 4, 5);
4 BEGIN
5     -- Resto del codigo
6 END;
```

## Acceso a una matriz

Los elementos de una matriz se pueden acceder utilizando el nombre de la matriz seguido de un índice entre paréntesis. El índice indica la posición del elemento en la matriz, comenzando desde 1.

A continuación, se muestra un ejemplo de acceso a los elementos de una matriz en PL/SQL:

```

1 DECLARE
2     TYPE matriz_numeros IS VARRAY(5) OF NUMBER;
3     numeros matriz_numeros := matriz_numeros(1, 2, 3, 4, 5);
4     elemento NUMBER;
5 BEGIN
6     elemento := numeros(3); -- Acceso al elemento en la posicion 3
7     DBMS_OUTPUT.PUT_LINE('Elemento: ' || elemento);
8 END;
```

## Operaciones con matrices

En PL/SQL, se pueden realizar diversas operaciones con matrices, como actualización de elementos, obtención del tamaño de la matriz, iteración sobre los elementos y concatenación de matrices. A continuación, se presentan algunos ejemplos de estas operaciones:

### ■ Actualización de elementos de una matriz:

```
1 DECLARE
2     TYPE matriz_numeros IS VARRAY(5) OF NUMBER;
3     numeros matriz_numeros := matriz_numeros(1, 2, 3, 4, 5);
4 BEGIN
5     numeros(3) := 10; -- Actualizacion del elemento en la posicion 3
6     -- Resto del codigo
7 END;
```

### ■ Obtención del tamaño de una matriz:

```
1 DECLARE
2     TYPE matriz_numeros IS VARRAY(5) OF NUMBER;
3     numeros matriz_numeros := matriz_numeros(1, 2, 3, 4, 5);
4     total_elementos INTEGER;
5 BEGIN
6     total_elementos := numeros.COUNT; -- Obtener la cantidad de elementos en la matriz
7     DBMS_OUTPUT.PUT_LINE('Total de elementos: ' || total_elementos);
8 END;
```

### ■ Iteración sobre los elementos de una matriz:

```
1 DECLARE
2     TYPE matriz_numeros IS VARRAY(5) OF NUMBER;
3     numeros matriz_numeros := matriz_numeros(1, 2, 3, 4, 5);
4 BEGIN
5     DBMS_OUTPUT.PUT_LINE('Elementos de la matriz:');
6     FOR i IN 1..numeros.COUNT LOOP
7         DBMS_OUTPUT.PUT_LINE('Elemento ' || i || ': ' || numeros(i));
8     END LOOP;
9 END;
```

### ■ Concatenación de matrices:

```
1 DECLARE
2     TYPE matriz_numeros IS VARRAY(5) OF NUMBER;
```

```

3      numeros1 matriz_numeros := matriz_numeros(1, 2, 3);
4      numeros2 matriz_numeros := matriz_numeros(4, 5);
5      numeros_concat matriz_numeros;
6 BEGIN
7      numeros_concat := numeros1 || numeros2; -- Concatenacion de las matrices numeros1 y numeros2
8      -- Resto del codigo
9 END;
```

Estos ejemplos muestran algunas de las operaciones más comunes que se pueden realizar con matrices en PL/SQL. Puedes adaptar y combinar estas operaciones según tus necesidades específicas.

Recuerda que la sintaxis y las funciones específicas pueden variar dependiendo de la versión de Oracle y de las características soportadas.

## 2.6. Funciones

Las funciones en PL/SQL son subprogramas que realizan un cálculo o una operación y devuelven un valor. Pueden tener parámetros de entrada y/o de salida, lo que les permite aceptar valores proporcionados al llamar a la función y retornar un resultado al finalizar su ejecución.

### Parámetros de entrada

Los parámetros de entrada de una función son utilizados para pasar valores desde el código que llama a la función hacia el interior de la función. Estos valores son utilizados dentro de la función para realizar cálculos u operaciones. Los parámetros de entrada se definen en la declaración de la función y deben especificar su tipo de datos.

A continuación, se muestra un ejemplo de una función con parámetros de entrada:

```

1 CREATE OR REPLACE FUNCTION calcular_area_circulo(radio IN NUMBER) RETURN NUMBER IS
2     area NUMBER;
3 BEGIN
4     area := 3.14159 * radio * radio;
5     RETURN area;
6 END;
```

En este ejemplo, la función `calcular_area_circulo` acepta un parámetro de entrada llamado `radio` de tipo `NUMBER`. Dentro de la función, se utiliza este parámetro para calcular el área de un círculo y se devuelve el resultado.

## Parámetros de salida

Los parámetros de salida de una función son utilizados para devolver valores desde la función hacia el código que la llama. Estos valores son definidos y asignados dentro de la función y luego son retornados al finalizar su ejecución. Los parámetros de salida se definen en la declaración de la función utilizando la cláusula `RETURN`.

A continuación, se muestra un ejemplo de una función con parámetros de salida:

```
1 CREATE OR REPLACE FUNCTION obtener_saludo(nombre IN VARCHAR2) RETURN VARCHAR2 IS
2     saludo VARCHAR2(50);
3 BEGIN
4     saludo := 'Hola, ' || nombre || '!';
5     RETURN saludo;
6 END;
```

En este ejemplo, la función `obtener_saludo` acepta un parámetro de entrada llamado `nombre` de tipo `VARCHAR2` y devuelve un saludo personalizado utilizando ese nombre.

## Ejemplos

A continuación, se presentan ejemplos de funciones que combinan parámetros de entrada y salida:

```
1 CREATE OR REPLACE FUNCTION calcular_promedio(numeros IN matriz_numeros) RETURN NUMBER IS
2     suma NUMBER := 0;
3     promedio NUMBER;
4 BEGIN
5     FOR i IN 1..numeros.COUNT LOOP
6         suma := suma + numeros(i);
7     END LOOP;
8
9     promedio := suma / numeros.COUNT;
10    RETURN promedio;
11 END;
12
13 CREATE OR REPLACE FUNCTION obtener_persona(id IN NUMBER) RETURN persona IS
14     p persona;
15 BEGIN
16     SELECT * INTO p FROM personas WHERE persona_id = id;
17     RETURN p;
18 END;
```



En el primer ejemplo, la función `calcular_promedio` recibe una matriz de números como parámetro de entrada y calcula el promedio de esos números. Devuelve el promedio como resultado.

En el segundo ejemplo, la función `obtener_persona` recibe un `ID` como parámetro de entrada y busca en una tabla de personas el registro correspondiente a ese `ID`. Retorna el registro de persona encontrado.

## 2.7. Procedimientos Almacenados en SQL

Un procedimiento almacenado es una colección de instrucciones SQL que se guarda en el servidor de base de datos y se puede invocar para realizar operaciones específicas. Proporciona una forma conveniente de ejecutar tareas sin tener que volver a escribir el código SQL cada vez.

La sintaxis básica para crear un procedimiento almacenado es la siguiente:

```
1 CREATE OR REPLACE PROCEDURE nombre_procedimiento (parametros)
2 AS
3 BEGIN
4 -- Cuerpo del procedimiento (instrucciones SQL)
5 END;
```

Los parámetros son valores opcionales que se pueden pasar al procedimiento para su procesamiento. Pueden ser de entrada, salida o ambos.

A continuación, se presentan algunos ejemplos prácticos de procedimientos almacenados en SQL:

### Ejemplo 1: Procedimiento para insertar un nuevo registro

Supongamos que tenemos una tabla “Empleados” con las siguientes columnas: `ID`, `Nombre` y `Salario`. Queremos crear un procedimiento almacenado para insertar un nuevo empleado en la tabla.

```
1 CREATE OR REPLACE PROCEDURE insertar_empleado(
2 p_ID INT,
3 p_Nombre VARCHAR(100),
4 p_Salario DECIMAL(10,2))
5 AS
6 BEGIN
7 INSERT INTO Empleados (ID, Nombre, Salario)
8 VALUES (p_ID, p_Nombre, p_Salario);
9 END;
```

En este ejemplo, el procedimiento almacenado `insertar_empleado` toma tres parámetros de entrada: `p_ID`, `p_Nombre` y `p_Salario`. Luego, ejecuta una instrucción `INSERT` para agregar un nuevo empleado a la

tabla “Empleados” con los valores proporcionados.

Para llamar a este procedimiento almacenado, se puede utilizar la siguiente sentencia:

```
1 EXECUTE insertar_empleado(1, 'John Doe', 5000.00);
```

Esto insertará un nuevo registro en la tabla “Empleados” con el ID 1, el nombre “John Doe” y un salario de 5000.00.

## Ejemplo 2: Procedimiento para actualizar el salario de un empleado

Supongamos que queremos crear un procedimiento almacenado para actualizar el salario de un empleado en la tabla “Empleados” basado en su ID.

```
1 CREATE OR REPLACE PROCEDURE actualizar_salario_empleado(  
2 p_ID INT,  
3 p_NuevoSalario DECIMAL(10,2))  
4 AS  
5 BEGIN  
6 UPDATE Empleados  
7 SET Salario = p_NuevoSalario  
8 WHERE ID = p_ID;  
9 END;
```

En este ejemplo, el procedimiento almacenado `actualizar_salario_empleado` toma dos parámetros de entrada: `p_ID` y `p_NuevoSalario`. Luego, ejecuta una instrucción `UPDATE` para modificar el salario de un empleado en la tabla “Empleados” con el ID correspondiente.

Para llamar a este procedimiento almacenado, se puede utilizar la siguiente sentencia:

```
1 EXECUTE actualizar_salario_empleado(1, 6000.00);
```

Esto actualizará el salario del empleado con ID 1 a 6000.00 en la tabla “Empleados”.

## Ejemplo 3: Procedimiento para eliminar un empleado

Supongamos que queremos crear un procedimiento almacenado para eliminar un empleado de la tabla “Empleados” basado en su ID.

```
1 CREATE OR REPLACE PROCEDURE eliminar_empleado(  
2 p_ID INT)  
3 AS  
4 BEGIN  
5 DELETE FROM Empleados  
6 WHERE ID = p_ID;
```

```
7 END;
```

En este ejemplo, el procedimiento almacenado "eliminar\_empleado" toma un parámetro de entrada: p\_ID. Luego, ejecuta una instrucción DELETE para eliminar un empleado de la tabla "Empleados" con el ID correspondiente.

Para llamar a este procedimiento almacenado, se puede utilizar la siguiente sentencia:

```
1 EXECUTE eliminar_empleado(1);
```

Esto eliminará el empleado con ID 1 de la tabla "Empleados".

Los procedimientos almacenados son útiles para encapsular lógica compleja de base de datos, mejorar el rendimiento, reutilizar código y mantener la integridad de los datos. Se pueden llamar desde aplicaciones o incluso desde otros procedimientos almacenados.

## 2.8. Triggers

Un trigger es un objeto de base de datos que se asocia con una tabla y se dispara automáticamente cuando ocurre un evento en la tabla, como una operación de inserción, actualización o eliminación. Los triggers permiten ejecutar acciones personalizadas antes o después de que ocurra un evento en la tabla.

La sintaxis básica para crear un trigger es la siguiente:

```
1 CREATE OR REPLACE TRIGGER nombre_trigger
2 {BEFORE | AFTER} {INSERT | UPDATE | DELETE} ON nombre_tabla
3 [FOR EACH ROW]
4 BEGIN
5 -- Cuerpo del trigger (instrucciones PL/SQL)
6 END;
```

El disparador se puede configurar para ejecutarse antes o después de un evento (BEFORE o AFTER), y se especifica qué tipo de evento desencadenará el trigger (INSERT, UPDATE o DELETE). Además, se puede agregar la cláusula opcional FOR EACH ROW para indicar que el trigger se ejecutará una vez por cada fila afectada por el evento.

A continuación, se muestra un ejemplo de trigger que invoca el procedimiento almacenado insert\_cliente después de cada inserción en la tabla Cliente:

```
1 CREATE OR REPLACE TRIGGER tr_insert_cliente
2 AFTER INSERT ON Cliente
3 FOR EACH ROW
4 BEGIN
5 insert_cliente_control(:new.strNumeroIdentificacion, :new.strNombre || ' ' || :new.strApellido);
```

```
6 END;
```

En este ejemplo, el trigger `tr_insert_cliente` se ejecutará después de cada inserción en la tabla `Cliente`. Utiliza la pseudo-variable `:new` para acceder a los valores de la nueva fila insertada y los pasa como argumentos al procedimiento almacenado `insert_cliente_control`.

## Ejemplo de Uso

A continuación, se muestra un ejemplo completo que combina la creación de la tabla `ClienteInformacion`, el procedimiento almacenado `insert_cliente_control` y el trigger `tr_insert_cliente`:

```
1 CREATE TABLE ClienteInformacion (  
2 Identificacion VARCHAR(20) PRIMARY KEY,  
3 NombreCompleto VARCHAR(100),  
4 FechaRegistro DATE,  
5 FechaUltimaActualizacion DATE  
6 );  
7  
8 CREATE OR REPLACE PROCEDURE insert_cliente_control(  
9 Identificacion IN VARCHAR,  
10 NombreCompleto IN VARCHAR)  
11 AS  
12 BEGIN  
13 INSERT INTO ClienteInformacion VALUES (Identificacion, NombreCompleto, SYSDATE, SYSDATE);  
14 END;  
15  
16 CREATE OR REPLACE TRIGGER tr_insert_cliente  
17 AFTER INSERT ON Cliente  
18 FOR EACH ROW  
19 BEGIN  
20 insert_cliente_control(:new.strNumeroIdentificacion, :new.strNombre || ' ' || :new.strApellido);  
21 END;  
22  
23 INSERT INTO Cliente (strTipoIdentificacion, strNumeroIdentificacion, strNombre, strApellido, strEmail)  
24 VALUES ('cedula', '1111', 'El kakas', 'kakasNet', 'kakasNet@gmail.com');  
25  
26 SELECT * FROM ClienteInformacion;
```

En este ejemplo, se crea la tabla `ClienteInformacion` con las columnas requeridas. Luego, se define el procedimiento almacenado `insert_cliente_control`, que inserta un nuevo registro en la tabla `ClienteInformacion`. Después, se crea el trigger `tr_insert_cliente`, que invoca el procedimiento almace-

nado después de cada inserción en la tabla **Cliente**. Finalmente, se realiza una inserción en la tabla **Cliente** y se muestra el contenido de la tabla **ClienteInformacion** para verificar los resultados.

Los triggers son una herramienta poderosa en SQL que permiten automatizar tareas y ejecutar lógica personalizada en respuesta a eventos en las tablas. Pueden ser utilizados para mantener la integridad de los datos, auditar cambios, aplicar reglas de negocio y mucho más. Es importante tener en cuenta las implicaciones de rendimiento y la correcta gestión de los eventos que desencadenan los triggers para asegurar un funcionamiento óptimo de la base de datos.