PL/SQL

Autor:

Kevin Cárdenas.

2023

Índice

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:

```sql
CREATE TABLE empleados (
empleado_id INT PRIMARY KEY,
nombre VARCHAR(50),
apellido VARCHAR(50),
...
);
```

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:

```sql
CREATE TABLE ventas (
venta_id INT PRIMARY KEY,
empleado_id INT,
...
FOREIGN KEY (empleado_id) REFERENCES empleados(empleado_id)
);
```

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:

```sql
CREATE TABLE estudiantes (
estudiante_id INT,
curso_id INT,
nombre VARCHAR(50),
PRIMARY KEY (estudiante_id, curso_id)
);
```

Foreign Key

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

```sql
CREATE TABLE ordenes (
orden_id INT PRIMARY KEY,
cliente_id INT,
fecha DATE,
FOREIGN KEY (cliente_id) REFERENCES clientes(cliente_id)
);
```

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

```sql
CREATE TABLE empleados (
empleado_id INT PRIMARY KEY,
nombre VARCHAR(50),
jefe_id INT,
FOREIGN KEY (jefe_id) REFERENCES empleados(empleado_id)
);
```

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:

```
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:

```sql
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:

```sql
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:

```sql
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:

```sql
SELECT empleados.nombre, departamentos.nombre
FROM empleados
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:

```
SELECT categoria, COUNT(*) as total_productos
FROM productos
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:

```
SELECT nombre, precio
FROM productos
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';
```

```
    edad NUMBER := 30;
BEGIN
    -- Logic of program
    IF edad >= 18 THEN
        DBMS_OUTPUT.PUT_LINE(nombre || ' es mayor de edad');
    ELSE
        DBMS_OUTPUT.PUT_LINE(nombre || ' es menor de edad');
    END IF;
END;

-- Sentencias SQL
INSERT INTO empleados (nombre, salario)
    VALUES ('Ana', 5000);

COMMIT;
END;
```

En este ejemplo, el bloque de PL/SQL anónimo comienza con la palabra clave "BEGIN" y finaliza con "END;" 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:

```
CREATE OR REPLACE PROCEDURE calcular_salario(p_empleado_id NUMBER) AS
v_salario NUMBER;
BEGIN
    -- Logica para calcular el salario del empleado
    SELECT salario INTO v_salario
        FROM empleados
        WHERE empleado_id = p_empleado_id;

    DBMS_OUTPUT.PUT_LINE('El salario del empleado ' || p_empleado_id || ' es: ' || v_salario);
END;
```

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":

```
CREATE OR REPLACE TRIGGER insertar_empleado_trigger
AFTER INSERT ON empleados
FOR EACH ROW
BEGIN
    DBMS_OUTPUT.PUT_LINE('Nuevo empleado insertado: ' || :NEW.nombre);
END;
```

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.

```
DECLARE
    nombre VARCHAR2(50) := 'Juan';
BEGIN
    DBMS_OUTPUT.PUT_LINE('Nombre: ' || nombre);
END;
```

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

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

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

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

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

```
DECLARE
    es_mayor BOOLEAN := TRUE;
BEGIN
    IF es_mayor THEN
        DBMS_OUTPUT.PUT_LINE('Es mayor de edad');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Es menor de edad');
    END IF;
END;
```

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

```
DECLARE
    inicial CHAR(1) := 'A';
BEGIN
    DBMS_OUTPUT.PUT_LINE('Inicial: ' || inicial);
END;
```

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

```
DECLARE
    descripcion LONG := 'Esta es una descripcion larga';
BEGIN
    DBMS_OUTPUT.PUT_LINE('Descripcion: ' || descripcion);
END;
```

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

```
DECLARE
    datos RAW(100) := UTL_RAW.CAST_TO_RAW('010101');
BEGIN
    DBMS_OUTPUT.PUT_LINE('Datos: ' || UTL_RAW.CAST_TO_VARCHAR2(datos));
END;
```

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

```
DECLARE
    es_mayor BOOLEAN := FALSE;
    edad INTEGER := 18;
BEGIN
    es_mayor := (edad >= 18);
    IF es_mayor THEN
        DBMS_OUTPUT.PUT_LINE('Es mayor de edad');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Es menor de edad');
    END IF;
END;
```

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

```
DECLARE
    descripcion CLOB := 'Esta es una descripcion larga';
BEGIN
    DBMS_OUTPUT.PUT_LINE('Descripcion: ' || descripcion);
END;
```

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

```
DECLARE
    fecha_hora TIMESTAMP := SYSTIMESTAMP;
BEGIN
    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.

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

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

```
DECLARE
    TYPE tipo_empleado IS RECORD (
        nombre VARCHAR2(50),
        edad NUMBER,
        sueldo NUMBER(10, 2)
    );

    empleado tipo_empleado;
BEGIN
    empleado.nombre := 'Juan';
    empleado.edad := 30;
    empleado.sueldo := 5000.50;

    DBMS_OUTPUT.PUT_LINE('Nombre: ' || empleado.nombre);
    DBMS_OUTPUT.PUT_LINE('Edad: ' || empleado.edad);
    DBMS_OUTPUT.PUT_LINE('Sueldo: ' || empleado.sueldo);
END;
```

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

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

```
DECLARE
```

```
    nombre VARCHAR2(50);
    direccion VARCHAR2(100);
BEGIN
    nombre := 'Juan';
    direccion := 'Calle Principal';
    -- Resto del codigo...
END;
```

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

```
DECLARE
    nombre VARCHAR2(50) := 'Juan';
    apellido VARCHAR2(50) := 'Perez';
    nombre_completo VARCHAR2(100);
BEGIN
    nombre_completo := nombre || ' ' || apellido;
    dbms_output.put_line('Nombre completo: ' || nombre_completo);
    -- Resto del codigo...
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:

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

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

```
DECLARE
    cadena VARCHAR2(50) := 'Hello, World!';
    longitud NUMBER;
BEGIN
```

```
    longitud := LENGTH(cadena);
    dbms_output.put_line('Longitud: ' || longitud);
    -- Resto del codigo...
END;
```

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

```
DECLARE
    cadena VARCHAR2(50) := 'Hello, World!';
    cadena_mayusculas VARCHAR2(50);
BEGIN
    cadena_mayusculas := UPPER(cadena);
    dbms_output.put_line('Cadena en mayusculas: ' || cadena_mayusculas);
    -- Resto del codigo...
END;
```

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

```
DECLARE
    cadena VARCHAR2(50) := 'Hello, World!';
    cadena_minusculas VARCHAR2(50);
BEGIN
    cadena_minusculas := LOWER(cadena);
    dbms_output.put_line('Cadena en minusculas: ' || cadena_minusculas);
    -- Resto del codigo...
END;
```

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

```
DECLARE
    cadena VARCHAR2(50) := 'Hello, World!';
    posicion NUMBER;
BEGIN
    posicion := INSTR(cadena, 'World');
    dbms_output.put_line('Posicion de "World": ' || posicion);
    -- Resto del codigo...
END;
```

- `REPLACE(cadena, subcadena, nueva_subcadena)`: Reemplaza todas las ocurrencias de una subcadena por otra en una cadena. Ejemplo:

```
DECLARE
    cadena VARCHAR2(50) := 'Hello, World!';
```

```
    nueva_cadena VARCHAR2(50);
BEGIN
    nueva_cadena := REPLACE(cadena, 'World', 'Mundo');
    dbms_output.put_line('Cadena modificada: ' || nueva_cadena);
    -- Resto del codigo...
END;
```

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

```
DECLARE
    cadena VARCHAR2(50) := '   Hola, Mundo!   ';
    cadena_sin_espacios VARCHAR2(50);
BEGIN
    cadena_sin_espacios := TRIM(cadena);
    dbms_output.put_line('Cadena sin espacios: ' || cadena_sin_espacios);
    -- Resto del codigo...
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:

```
DECLARE
   edad NUMBER := 18;
BEGIN
   IF edad >= 18 THEN
      dbms_output.put_line('Eres mayor de edad');
   ELSE
      dbms_output.put_line('Eres menor de edad');
   END IF;
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:

```
CASE expresion
   WHEN valor1 THEN
      -- Codigo a ejecutar cuando la expresion es igual a valor1
   WHEN valor2 THEN
      -- Codigo a ejecutar cuando la expresion es igual a valor2
   ...
   ELSE
      -- Codigo a ejecutar cuando la expresion no coincide con ninguno de los valores anteriores
END CASE;
```

En este caso, `expresion` 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:

```
DECLARE
   nota NUMBER := 80;
BEGIN
   CASE nota
      WHEN 90 THEN
         DBMS_OUTPUT.PUT_LINE('Excelente');
      WHEN 80 THEN
         DBMS_OUTPUT.PUT_LINE('Bueno');
      WHEN 70 THEN
         DBMS_OUTPUT.PUT_LINE('Aceptable');
      ELSE
         DBMS_OUTPUT.PUT_LINE('Reprobado');
   END CASE;
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`:

```
DECLARE
   nota NUMBER := 75;
BEGIN
   CASE
      WHEN nota >= 90 THEN
         DBMS_OUTPUT.PUT_LINE('Excelente');
      WHEN nota >= 80 THEN
         DBMS_OUTPUT.PUT_LINE('Bueno');
      WHEN nota >= 70 THEN
         DBMS_OUTPUT.PUT_LINE('Aceptable');
      ELSE
         DBMS_OUTPUT.PUT_LINE('Reprobado');
   END CASE;
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:

```
DECLARE
   contador NUMBER := 1;
BEGIN
   LOOP
      dbms_output.put_line('Contador: ' || contador);
      contador := contador + 1;

      IF contador > 5 THEN
         EXIT; -- Sale del bucle
      END IF;
   END LOOP;
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:

```
DECLARE
   contador NUMBER := 1;
BEGIN
   WHILE contador <= 5 LOOP
      dbms_output.put_line('Contador: ' || contador);
      contador := contador + 1;
   END LOOP;
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:

```
DECLARE
   total NUMBER := 0;
BEGIN
   FOR i IN 1..5 LOOP
      total := total + i;
   END LOOP;

   dbms_output.put_line('Total: ' || total);
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:

```
DECLARE
   nota NUMBER := 85;
BEGIN
   IF nota >= 90 THEN
      dbms_output.put_line('Calificacion: A');
   ELSIF nota >= 80 THEN
      dbms_output.put_line('Calificacion: B');
   ELSIF nota >= 70 THEN
      dbms_output.put_line('Calificacion: C');
   ELSE
      dbms_output.put_line('Calificacion: D');
   END IF;
   -- Resto del codigo...
END;
```

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:

```
DECLARE
   limite_filas NUMBER := 5;
   limite_columnas NUMBER := 5;
BEGIN
   FOR i IN 1..limite_filas LOOP
      FOR j IN 1..limite_columnas LOOP
         dbms_output.put(i*j || ' ');
      END LOOP;
      dbms_output.new_line;
   END LOOP;
   -- Resto del codigo...
END;
```

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

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:

```
DECLARE
   TYPE matriz_t IS VARRAY(n) OF tipo_dato;
   matriz matriz_t := matriz_t(elemento1, elemento2, ..., elementon);
BEGIN
   -- Resto del codigo
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:

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

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

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

```
    -- Resto del codigo
END;
```

- **Obtención del tamaño de una matriz**:

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

- **Iteración sobre los elementos de una matriz**:

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

- **Concatenación de matrices**:

```
DECLARE
    TYPE matriz_numeros IS VARRAY(5) OF NUMBER;
    numeros1 matriz_numeros := matriz_numeros(1, 2, 3);
    numeros2 matriz_numeros := matriz_numeros(4, 5);
    numeros_concat matriz_numeros;
BEGIN
    numeros_concat := numeros1 || numeros2; -- Concatenacion de las matrices numeros1 y numeros2
    -- Resto del codigo
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:

```
CREATE OR REPLACE FUNCTION calcular_area_circulo(radio IN NUMBER) RETURN NUMBER IS
  area NUMBER;
BEGIN
  area := 3.14159 * radio * radio;
  RETURN area;
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:

```
CREATE OR REPLACE FUNCTION obtener_saludo(nombre IN VARCHAR2) RETURN VARCHAR2 IS
  saludo VARCHAR2(50);
BEGIN
  saludo := 'Hola, ' || nombre || '!';
  RETURN saludo;
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:

```
CREATE OR REPLACE FUNCTION calcular_promedio(numeros IN matriz_numeros) RETURN NUMBER IS
   suma NUMBER := 0;
   promedio NUMBER;
BEGIN
   FOR i IN 1..numeros.COUNT LOOP
      suma := suma + numeros(i);
   END LOOP;

   promedio := suma / numeros.COUNT;
   RETURN promedio;
END;

CREATE OR REPLACE FUNCTION obtener_persona(id IN NUMBER) RETURN persona IS
   p persona;
BEGIN
   SELECT * INTO p FROM personas WHERE persona_id = id;
   RETURN p;
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

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:

```
CREATE OR REPLACE PROCEDURE nombre_procedimiento (parametros)
```

```
AS
BEGIN
    -- Cuerpo del procedimiento (instrucciones SQL)
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.

```
CREATE OR REPLACE PROCEDURE insertar_empleado(
    p_ID INT,
    p_Nombre VARCHAR(100),
    p_Salario DECIMAL(10,2))
AS
BEGIN
    INSERT INTO Empleados (ID, Nombre, Salario)
    VALUES (p_ID, p_Nombre, p_Salario);
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:

```
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.

```
CREATE OR REPLACE PROCEDURE actualizar_salario_empleado(
    p_ID INT,
    p_NuevoSalario DECIMAL(10,2))
```

```
AS
BEGIN
    UPDATE Empleados
    SET Salario = p_NuevoSalario
    WHERE ID = p_ID;
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:

```
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.

```
CREATE OR REPLACE PROCEDURE eliminar_empleado(
p_ID INT)
AS
BEGIN
    DELETE FROM Empleados
    WHERE ID = p_ID;
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:

```
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:

```
CREATE OR REPLACE TRIGGER nombre_trigger
{BEFORE | AFTER} {INSERT | UPDATE | DELETE} ON nombre_tabla
[FOR EACH ROW]
BEGIN
    -- Cuerpo del trigger (instrucciones PL/SQL)
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`:

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

```
CREATE TABLE ClienteInformacion (
Identificacion VARCHAR(20) PRIMARY KEY,
NombreCompleto VARCHAR(100),
FechaRegistro DATE,
```

```
FechaUltimaActualizacion DATE
);

CREATE OR REPLACE PROCEDURE insert_cliente_control(
Identificacion IN VARCHAR,
NombreCompleto IN VARCHAR)
AS
BEGIN
    INSERT INTO ClienteInformacion VALUES (Identificacion, NombreCompleto, SYSDATE, SYSDATE);
END;

CREATE OR REPLACE TRIGGER tr_insert_cliente
AFTER INSERT ON Cliente
FOR EACH ROW
BEGIN
    insert_cliente_control(:new.strNumeroIdentificacion, :new.strNombre || ' ' || :new.strApellido);
END;

INSERT INTO Cliente (strTipoIdentificacion, strNumeroIdentificacion, strNombre, strApellido, strEmail)
VALUES ('cedula', '1111', 'El kakas', 'kakasNet', 'kakasNet@gmail.com');

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 almacenado 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 PL/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.

Eventos en Triggers

En el contexto de los triggers, un evento se refiere a una acción específica que ocurre en una tabla y que dispara la ejecución del trigger. Los eventos comunes en los que se pueden definir triggers son **INSERT**

(inserción de datos), **UPDATE** (actualización de datos) y **DELETE** (eliminación de datos). Veamos más detalles sobre cada uno de estos eventos:

1. **INSERT**: Ejemplo de trigger para el evento INSERT:

```
CREATE OR REPLACE TRIGGER tr_insert_empleado
AFTER INSERT ON Empleados
FOR EACH ROW
BEGIN
    INSERT INTO RegistroEventos (Evento, Fecha) VALUES ('Nuevo empleado insertado', SYSDATE);
END;
```

   En este ejemplo, se crea un trigger llamado "tr_insert_empleado" que se ejecutará después de cada inserción en la tabla "Empleados". El trigger inserta una nueva fila en la tabla "RegistroEventos" con información sobre el evento ocurrido (en este caso, la inserción de un nuevo empleado) y la fecha actual.

2. **UPDATE**: Ejemplo de trigger para el evento UPDATE:

```
CREATE OR REPLACE TRIGGER tr_update_empleado
BEFORE UPDATE ON Empleados
FOR EACH ROW
BEGIN
    :new.FechaActualizacion := SYSDATE;
END;
```

   En este ejemplo, se crea un trigger llamado "tr_update_empleado" que se ejecutará antes de cada actualización en la tabla "Empleados". El trigger asigna la fecha actual a la columna "FechaActualizacion" de la fila que se está actualizando.

3. **DELETE**: Ejemplo de trigger para el evento DELETE:

```
CREATE OR REPLACE TRIGGER tr_delete_empleado
AFTER DELETE ON Empleados
FOR EACH ROW
BEGIN
    INSERT INTO RegistroEventos (Evento, Fecha) VALUES ('Empleado eliminado', SYSDATE);
END;
```

   En este ejemplo, se crea un trigger llamado "tr_delete_empleado" que se ejecutará después de cada eliminación en la tabla "Empleados". El trigger inserta una nueva fila en la tabla "RegistroEventos" con información sobre el evento ocurrido (en este caso, la eliminación de un empleado) y la fecha actual.

Además de estos eventos básicos, algunos sistemas de gestión de bases de datos también admiten otros eventos específicos, como eventos de cambio de esquema (DDL triggers) que se disparan cuando se realizan cambios en la estructura de la base de datos, eventos de inicio y apagado del servidor, eventos de inicio y finalización de una transacción, entre otros.

Es importante tener en cuenta que los triggers se pueden definir para ejecutarse antes (**BEFORE**) o después (**AFTER**) de los eventos. Los triggers **BEFORE** se ejecutan antes de que se realice la acción en la tabla, lo que les permite influir en los datos o validarlos antes de que se realice el cambio. Los triggers **AFTER** se ejecutan después de que se haya completado la acción en la tabla y se pueden utilizar para realizar tareas adicionales o registrar información relacionada con el evento.

Es fundamental comprender los eventos que desencadenarán la ejecución del trigger y definir las acciones adecuadas que se deben realizar para garantizar la integridad de los datos y cumplir con los requisitos del sistema.

Eventos Múltiples en Triggers

Los eventos múltiples en los triggers permiten combinar varios eventos en un único trigger, lo que significa que el trigger se activará y ejecutará su lógica cuando ocurra cualquiera de los eventos especificados. Esto brinda flexibilidad para definir comportamientos específicos en función de diferentes eventos que pueden ocurrir en una tabla.

La sintaxis básica para crear un trigger con eventos múltiples es la siguiente:

```
CREATE OR REPLACE TRIGGER nombre_trigger
{BEFORE | AFTER} {eventos}
ON nombre_tabla
[FOR EACH ROW]
BEGIN
-- Cuerpo del trigger (instrucciones SQL)
END;
```

Donde eventos es una lista de eventos separados por comas (,) que se desea combinar en el trigger. Estos eventos pueden ser INSERT, UPDATE o DELETE, y se especifica si el trigger se debe activar antes (BEFORE) o después (AFTER) de los eventos.

A continuación, se muestra un ejemplo de un trigger que se activa tanto después de una inserción como después de una actualización en la tabla “Clientes”:

```
CREATE OR REPLACE TRIGGER tr_eventos_multiples
AFTER INSERT OR UPDATE ON Clientes
FOR EACH ROW
```

```
BEGIN
-- Logica del trigger
-- ...
END;
```

En este ejemplo, el trigger "tr_eventos_multiples" se activará tanto después de una inserción como después de una actualización en la tabla "Clientes". Puedes agregar la lógica personalizada dentro del bloque BEGIN-END para manejar los eventos según tus necesidades.

Los eventos múltiples en los triggers permiten tener un mayor control sobre las acciones realizadas en la base de datos, ya que cada evento puede tener su propia lógica de manejo dentro del trigger. Esto facilita la implementación de reglas de negocio y la automatización de tareas relacionadas con diferentes eventos en una tabla.

Pseudo-Variables :NEW y :OLD en un Trigger

En los triggers de bases de datos, las pseudo-variables `:NEW` y `:OLD` se utilizan para acceder a los valores de las filas afectadas por un evento en una tabla. Estas pseudo-variables son especialmente útiles en los triggers `AFTER INSERT`, `AFTER UPDATE` y `AFTER DELETE`, ya que permiten acceder a los valores antiguos y nuevos de una fila.

- `:NEW`: Esta pseudo-variable representa los nuevos valores de la fila después de una operación de inserción o actualización. Por ejemplo, si se inserta una nueva fila en una tabla, `:NEW` contendrá los valores de esa nueva fila. Si se actualiza una fila existente, `:NEW` contendrá los nuevos valores después de la actualización. Ejemplo de uso en un trigger AFTER INSERT:

```
CREATE OR REPLACE TRIGGER tr_insert_example
AFTER INSERT ON mi_tabla
FOR EACH ROW
BEGIN
    -- Acceder a los nuevos valores de la fila insertada
    -- Utilizar :NEW.columna para acceder a una columna especifica
    -- ...
END;
```

- `:OLD`: Esta pseudo-variable representa los valores antiguos de la fila antes de una operación de actualización o eliminación. Por ejemplo, si se actualiza una fila existente, `:OLD` contendrá los valores originales antes de la actualización. Si se elimina una fila, `:OLD` contendrá los valores de la fila eliminada.

  Ejemplo de uso en un trigger AFTER UPDATE:

```
CREATE OR REPLACE TRIGGER tr_update_example
AFTER UPDATE ON mi_tabla
FOR EACH ROW
BEGIN
    -- Acceder a los valores antiguos y nuevos de la fila actualizada
    -- Utilizar :OLD.columna y :NEW.columna para acceder a columnas especificas
    -- ...
END;
```

Estas pseudo-variables se pueden utilizar dentro del cuerpo de un trigger para realizar acciones basadas en los valores antiguos y nuevos de una fila. Algunos ejemplos comunes de uso incluyen:

- Comparar los valores antiguos y nuevos para aplicar ciertas reglas de negocio:

```
IF :OLD.estado = 'Pendiente' AND :NEW.estado = 'Completado' THEN
-- Realizar alguna accion si el estado ha cambiado de "Pendiente" a "Completado"
-- ...
END IF;
```

- Auditar los cambios realizados en la tabla:

```
INSERT INTO tabla_auditoria (fecha, accion, columna_antigua, columna_nueva)
VALUES (SYSDATE, 'UPDATE', :OLD.columna1, :NEW.columna1);
```

- Realizar actualizaciones adicionales en otras tablas basadas en los cambios en la fila actualizada:

```
UPDATE otra_tabla
SET columna = :NEW.columna
WHERE id = :OLD.id;
```

El uso de `:NEW` y `:OLD` en los triggers proporciona una manera conveniente de acceder a los datos de las filas afectadas y realizar lógica adicional basada en esos valores. Esto permite la implementación de reglas de negocio más avanzadas, auditoría de cambios y acciones personalizadas en la base de datos.

2.9. Schedulers

En Oracle, los Schedulers son componentes importantes que permiten la programación y automatización de tareas en la base de datos. Estos Schedulers son responsables de ejecutar trabajos de manera programada y periódica, lo que facilita la administración y gestión de la base de datos.

Oracle ofrece diferentes tipos de Schedulers que se pueden utilizar según los requerimientos y necesidades del sistema. Algunos de los Schedulers más utilizados son:

- **DBMS_JOB**: Este Scheduler permite programar trabajos en forma de bloques PL/SQL. Los trabajos se pueden ejecutar en un momento específico o de manera recurrente según una programación establecida.

- **DBMS_SCHEDULER**: Este Scheduler ofrece una funcionalidad más avanzada y flexible en comparación con DBMS_JOB. Permite programar trabajos utilizando diferentes tipos de acciones, como bloques PL/SQL, procedimientos almacenados, scripts externos, entre otros. Además, brinda opciones de configuración detalladas para controlar la programación, repetición y parámetros de los trabajos.

En nuestro ejemplo anterior, utilizamos el Scheduler DBMS_SCHEDULER para programar un trabajo diario que se ejecute a las 8 AM. A continuación se muestra el código utilizado:

```
BEGIN
    DBMS_SCHEDULER.CREATE_JOB(
        job_name => 'my_job',
        job_type => 'PLSQL_BLOCK',
        job_action => 'BEGIN my_procedure; END;',
        start_date => SYSTIMESTAMP,
        repeat_interval => 'FREQ=DAILY; BYHOUR=8; BYMINUTE=0;',
        enabled => TRUE
    );
END;
```

En este ejemplo, creamos un trabajo llamado `my\_job` que ejecuta un bloque PL/SQL utilizando el procedimiento `my\_procedure`. El trabajo está programado para ejecutarse diariamente a las 8 AM.

Los Schedulers brindan una forma poderosa de automatizar tareas y mejorar la eficiencia en la administración de la base de datos. Permiten realizar acciones programadas de manera automática, lo que reduce la intervención manual y asegura la ejecución oportuna de las tareas planificadas.

Ejemplo de uso de Schedulers en Oracle

A continuación, se presenta el código necesario para crear las tablas “Clientes” y “Productos” en Oracle:

```
CREATE TABLE "Clientes" (
    "NumeroIdentificacion" NVARCHAR2(450) NOT NULL,
    "TipoIdentificacion" NVARCHAR2(2000) NOT NULL,
    "Nombre" NVARCHAR2(2000) NOT NULL,
    "Apellido" NVARCHAR2(2000) NOT NULL,
    "FechaCreacion" NVARCHAR2(2000) NOT NULL,
```

```
    "FechaModificacion" NVARCHAR2(2000) NOT NULL,
    "CorreoElectronico" NVARCHAR2(2000) NOT NULL,
    "FechaNacimiento" NVARCHAR2(2000) NOT NULL,
    CONSTRAINT "PK_Clientes" PRIMARY KEY ("NumeroIdentificacion")
);
CREATE TABLE "Productos" (
    "NumeroDeCuenta" NVARCHAR2(450) NOT NULL,
    "ClienteId" NVARCHAR2(2000) NOT NULL,
    "Tipo" NVARCHAR2(2000) NOT NULL,
    "estado" NVARCHAR2(2000) NOT NULL,
    "GMF" NVARCHAR2(2000) NOT NULL,
    "saldo" NUMBER(18,2) NOT NULL,
    CONSTRAINT "PK_Productos" PRIMARY KEY ("NumeroDeCuenta")
);
```

A continuación, se muestra el procedimiento almacenado que se encarga de revisar y eliminar los clientes que no tienen productos asociados:

```
create or replace PROCEDURE ELIMINAR_CLIENTES_SIN_PRODUCTOS AS
BEGIN
    DELETE FROM "C##USER"."Clientes"
    WHERE "NumeroIdentificacion" NOT IN (SELECT "ClienteId" FROM "C##USER"."Productos");

    COMMIT;
END;
```

Finalmente, utilizaremos el Scheduler `DBMS_SCHEDULER` para programar el trabajo que ejecutará el procedimiento cada 30 minutos:

```
BEGIN
DBMS_SCHEDULER.CREATE_JOB (
    job_name        => 'ELIMINAR_CLIENTES',
    job_type        => 'PLSQL_BLOCK',
    job_action      => 'BEGIN ELIMINAR_CLIENTES_SIN_PRODUCTOS; END;',
    start_date      => SYSTIMESTAMP,
    repeat_interval => 'FREQ=MINUTELY; INTERVAL=30',
    end_date        => NULL,
    enabled         => TRUE,
    comments        => 'Eliminar clientes sin productos asociados cada 30 minutos'
);

COMMIT;
```

```
14 END;
```

En este ejemplo, creamos un trabajo llamado `ELIMINAR_CLIENTES` que ejecuta el procedimiento almacenado `ELIMINAR_CLIENTES_SIN_PRODUCTOS`. El trabajo está programado para ejecutarse cada 30 minutos a partir del momento de su creación.