PLSQL

Álvaro González Sotillo

19 de abril de 2024

Índice

1. Introducción

- PLSQL es un lenguaje de programación ejecutado en los servidores Oracle
  - Con acceso a todas las sentencias SQL
  - Incluye variables, funciones, control de flujo...
- PLSQL es un lenguaje imperativo
  - Completa a SQL, que es un lenguaje declarativo

1.1. Palabras reservadas

- Vista `V$RESERVED_WORDS`
- Definen estructuras de programa
- No pueden ser usados como identificadores

1.2. Identificadores

- Nombres definidos por el programador
  - No puede ser una palabra reservada
  - Constante, variable, excepción, paquete, función, procedimiento, tabla, cursor...
  - Hasta 30 caracteres
  - Comienza por una letra.
  - Puede contener `$`, `#`, pero no puede contener operadores `+ %=/*`

2. Bloques anónimos

```
select * from pepe where nombre='a';

SET SERVEROUTPUT ON;
begin
  dbms_output.put_line('Hola');
END;
/
```

Listado 1: Bloque anónimo

3. Variables

- Valores referenciados por un identificador
- Deben declararse al principio de los bloques

```
SET SERVEROUTPUT ON;

DECLARE
  msg varchar(255);
BEGIN
  msg := 'Hola';
  dbms_output.put_line(msg);
END;
/
```

3.1. Tipos de variable

- Se pueden utilizar todos los tipos SQL
  - `char`, `varchar`
  - `number`, `integer`, `float`
  - `date`, `timestamp`
  - `blob`, `clob`
- Tipos propios de PLSQL
  - `bool`
  - `pls_integer`

3.2. Tipos referidos

- `%type` : Tipo de un campo de una tabla
- `%rowtype` : Tipo compuesto, referido a una fila de una tabla

```
create table cliente( id integer, nombre varchar(255) );

DECLARE
  filacliente cliente%rowtype;
BEGIN
  filacliente.id := 1;
  filacliente.nombre := 'María';
  insert into cliente values filacliente;
END;
/
```

4. Control de flujo

4.1. Condicional

```
DECLARE
  numero integer := 1;
BEGIN
  if( numero < 0 ) then
    dbms_output.put_line( 'Menor que cero');
  elsif( numero > 0 ) then
    dbms_output.put_line( 'Mayor que cero');
  else
    dbms_output.put_line( 'Igual que cero');
  end if;
END;
/
```

4.2. Condicional múltiple (I)

```
case
  when vsalario<0 then
    dbms_output.put_line('Incorrecto');
  when vsalario=0 THEN
    dbms_output.put_line('Gratis!');
  when vsalario<10000 then
    dbms_output.put_line('Salado!');
  when vsalario<90000 then
    dbms_output.put_line('Mas o menos');
  else
    dbms_output.put_line('Correcto');
end case;
```

4.3. Condicional múltiple (II)

```
case v_job_grade
  when 1 THEN
    dbms_output.put_line('Jefe!');
  when 2 then
    dbms_output.put_line('Jefecito');
  when 3 then
    dbms_output.put_line('Empleado regular');
  ELSE
    dbms_output.put_line('CEO');
end case;
```

4.4. Ejemplos de `case`

Queremos implementar un servicio de traducción de español a inglés. El servicio no está disponible los lunes

4.4.1. `case` como sentencia con un valor

```
declare
    v varchar(100) := 'Hasta luego';
begin
    if to_char(sysdate,'D')=1 then
        dbms_output.put_line('es mi día libre' );
    else

        case v
            when 'Hola' then
                dbms_output.put_line('Hello' );
            when 'Adiós' then
                dbms_output.put_line('Bye' );
            else
                dbms_output.put_line('No traduction' );
        end case;
    end if;
end;
/
```

4.4.2. `case` como sentencia con múltiples comparaciones

```
declare
    v varchar(100) := 'Hola';
begin
    case
        when to_char(sysdate,'D')=1 then
            dbms_output.put_line('es mi día libre' );
        when v='Hola' then
            dbms_output.put_line('Hello' );
        when v='Adiós' then
            dbms_output.put_line('Bye' );
        else
            dbms_output.put_line('No traduction' );
    end case;
end;
/
```

4.4.3. `case` como expresión con múltiples comparaciones

```
declare
    v varchar(100) := 'Hola';
    traduccion varchar(100);
begin
    traduccion := case
        when to_char(sysdate,'D')=1 then
            'es mi día libre'
        when v='Hola' then
            'Hello'
        when v='Adiós' then
            'Bye'
        else
            'No traduction'
    end;
    dbms_output.put_line(traduccion);
end;
/
```

4.4.4. `case` como expresión con un valor

```
declare
    v varchar(100) := 'Hola';
    traduccion varchar(100);
begin
    if to_char(sysdate,'D')=1  then
        traduccion := 'es mi día libre';
    else
        traduccion := case v
            when 'Hola' then
                'Hello'
            when 'Adiós' then
                'Bye'
            else
                'No traduction'
        end;
    end if;
    dbms_output.put_line(traduccion);
end;
/
```

4.4.5. Case usado en sentencias `sql`

```
select nombre, precioventa, case
  when precioventa >= 100 then 'carísimo'
  when precioventa >= 10 then 'caro'
  else 'barato' end as rango
from productos
order by 3;
```

```
select nombre,precioventa, 'caro'
from productos
where precioventa >= 10
union
select nombre,precioventa, 'barato'
from productos
where precioventa < 10;
```

4.5. Bucle `loop`

```
LOOP
    -- Instrucciones
    IF (expresion) THEN
      -- Instrucciones
      EXIT;
    END IF;
END LOOP;
```

4.6. Bucle `while`

```
WHILE (expresion) LOOP
    -- Instrucciones
END LOOP;
```

4.7. Bucle `for`

```
DECLARE
  c PLS_INTEGER DEFAULT 0;
BEGIN
  FOR c IN REVERSE 1..10 LOOP
    dbms_output.put_line ('Contador = '||c);
  END LOOP;
END;
```

4.8. Ejercicios

- Imprime los números del 1 al 100
- Imprime la suma de los números del 1 al 100
- Imprime los números pares del 1 al 100
- Imprime los números primos del 1 al 100
- Imprime la suma de los números primos del 1 al 100
- Encuentra un número primo mayor de 1000000
- Calcula el máximo común divisor entre 1234516 y 77636519368
- Calcula el mínimo común múltiplo entre 1234516 y 77636519368

4.9. Ejercicios

- Imprime un rectángulo de tamaño 8x6

```
########
########
########
########
########
########
```

- Imprime un tablero de ajedrez de tamaño NxM (con la función `MOD`)

```
# # # #
 # # # #
# # # #
 # # # #
# # # #
 # # # #
```

5. Funciones y procedimientos

- Son bloques de código identificados con un nombre
- Pueden invocarse desde otros bloques de código
- En la invocación, se utilizan parámetros
  - De entrada
  - De salida

(youtube)

5.1. Funciones

- Las funciones devuelven **siempre** un valor
- Pueden recibir parámetros
- Por convenio:
  - El resultado de una función solo depende de sus parámetros
  - Una función no cambia la base de datos

5.1.1. Ya hemos usado funciones

```
select to_date('01-01-01','dd-MM-yyyy') from dual;
select * from student where mod(studentid,2)=0;

sqrt, upper, substr, len, nvl...
```

5.1.2. Funciones propias

- Se pueden crear nuevas funciones, usando las que ya existen

```
CREATE OR REPLACE FUNCTION es_par(numero IN number)
RETURN boolean
IS
  resto number;
BEGIN
  resto := mod(numero,2);
  if( resto = 0 ) then
    return true;
  else
    return false;
  end if;
END;
/
```

5.1.3. Invocar una función

Las funciones pueden invocarse:

- Desde otra función o procedimiento
- Desde un bloque anónimo
- Desde SQL (ver más adelante)

```
CREATE OR REPLACE FUNCTION es_par_varchar(numero IN number)
RETURN varchar
IS
BEGIN
  -- AQUÍ SE INVOCA LA FUNCIÓN DEL APARTADO ANTERIOR
  if( es_par(numero) ) then
    return 'Sí es par';
  else
```

```
    return 'No es par';
  end if;
END;
/

DECLARE
  n number := 32;
BEGIN
  dbms_output.put_line( 'El número ' || n || ' ' || es_par_varchar(n) );
END;
/
```

5.1.4. Funciones en SQL

- Una función puede utilizarse en SQL

```
select empno, es_par_varchar(empno) from empleados;
select es_par_varchar(89) from dual;
```

- Una función que devuelve un tipo que no existe en SQL no se puede usar

```
select es_par(89) from dual;
ORA-00902: tipo de dato no válido
00902. 00000 -  "invalid datatype"
```

5.1.5. Funciones predefinidas

replace	sysdate	lpad	instr
substr	nvl	trim	trunc
upper	to_date	mod	length
lower	to_char	decode	
rpad	to_number		

Formatos de `to_number`
Formatos de `to_date`

5.1.6. Ejemplos de `to_date` y `to_char`

```
declare
  fechaentexto varchar(255);
  fecha date;
begin
  fechaentexto := '11/may/18';
  fecha := to_date(fechaentexto,'DD/MON/YY');
  dbms_output.put_line( to_char(fecha,'DD "de" MONTH "de" YYYY'));
end;
/


declare
    pi number(20,10) := 3.141597265;
begin
    dbms_output.put_line( to_char(pi, 'B9999' ) );
end;
/
```

5.1.7. Ejercicios

- Imprime un listado con la inicial de los empleados y sus apellidos

  ```
  A. Pérez
  F. González
  M. Ruiz
  ```

- Convierte la cadena `11/MAY/20` a fecha, e imprímela como `11 de Mayo de 2020`
- Imprime el número PI con 0,3 y 4 decimales.

5.1.8. Ejercicios

- Haz una función que devuelva `1` si un número es primo o `0` si es compuesto
  - Tiene que *devolver* un valor, no *imprimir* un valor
- Haz una función que devuelva capitalizada la palabra recibida
  - Si recibe `paLABRa` debe devolver `Palabra`
- Haz una función que devuelva capitalizada una frase, capitalizando cada palabra

5.1.9. Ejercicio resuelto: `capitalizar`

```
create or replace function capitalizar( palabra varchar ) return varchar as
  inicial char(1);
  resto varchar(1024);
begin
  inicial := substr(palabra,1,1);
  resto := substr(palabra,2,length(palabra)-1);
  return upper(inicial) || lower(resto);
end;
```

5.2. Procedimientos

- Los procedimentos no devuelven un valor
  - Pero pueden tener parámetros `out`

```
CREATE OR REPLACE PROCEDURE aumenta_salario(vempno IN number)
IS
BEGIN
  update empleados
    set salario=salario+100
    where empno = vempno;
END;
/
```

5.3. Parámetros `in`

- Es el tipo de parámetros por defecto
- Un parámetro `in` se pasa *por valor*
- Se copia el valor introducido en el parámetro
- Un cambio del parámetro no afecta al bloque llamante

```
create or replace procedure suma_uno(n in numeric) is
begin
  n := n +1;
end;
/

declare
  numero numeric(10,0);
begin
  numero := 3;
  sumauno(numero);
  dbms_output.put_line(numero);
end;
/
```

5.4. Parámetros `out`

- Un parámetro `out` se pasa *por referencia*
- Un cambio del parámetro afecta al bloque llamante

```
create or replace procedure suma_uno(n in out numeric) is
begin
  n := n +1;
end;
/

declare
  numero numeric(10,0);
begin
  numero := 3;
  sumauno(numero);
  dbms_output.put_line(numero);
end;
/
```

6. Sentencias SQL en PLSQL

- Desde PLSQL pueden utilizarse las sentencias del DML
  - `select`
  - `update`
  - `insert`
  - `delete`

(youtube)

6.1. Variables en `select`

- Se puede leer el valor de un campo y guardarlo en una variable con `select ... into ... from...`
- Hay que asegurarse que la *query* devuelve solo una fila
- El número de columnas debe coincidir con el número de variables

```
create table empleados( empno number(20), salario number(8,2), nombre varchar(255));
insert into empleados(empno,salario,nombre) values (1,2000,'María');
insert into empleados(empno,salario,nombre) values (2,1000,'Juan');

DECLARE
  vempno NUMBER := 2;
  vsalario NUMBER;
BEGIN
  SELECT salario INTO vsalario FROM empleados WHERE empno=vempno;
  dbms_output.put_line('El empleado ' || vempno || ' tiene un sueldo de '||vsalario||' ');
end;
/
```

- Es fácil confundir variables con nombres de columna.
- Convenio: comenzar todas las variables con `v`

6.2. Ventaja de los tipos `%type`

- Una variable puede copiar su tipo de una columna de una tabla
- Así, si cambia la definición de la tabla (con un `alter table`), el PLSQL sigue siendo válido

```
DECLARE
  vempno empleados.empno%TYPE := 2;
  vsalario empleados.salario%TYPE;
BEGIN
  SELECT salario INTO vsalario FROM empleados WHERE empno=vempno;
  dbms_output.put_line('El empleado ' || vempno || ' tiene un sueldo de '||vsalario||' ');
end;
/
```

6.3. Ventaja de los tipos `%rowtype`

- Una variable puede contener todas las columnas de una fila
- Si cambia la definición de la tabla (con un `alter table`), el PLSQL sigue siendo válido

```
DECLARE
  vempno empleados.empno%TYPE := 2;
  vempleado empleados%ROWTYPE;
BEGIN
  SELECT * INTO vempleado FROM empleados WHERE empno=vempno;
  dbms_output.put_line('El empleado ' || vempno || ' tiene un sueldo de '|| vempleado.salario ||' ');
  dbms_output.put_line('El empleado ' || vempno || ' se llama '|| vempleado.nombre ||' ');
end;
/
```

```
create or replace function empleado_con_id(
  p_empno empleados.empno%TYPE
)
return empleados%rowtype
as
    empleado empleados%rowtype;
begin

    select *
    into empleado
    from empleados
    where empno=p_empno;

    return empleado;
end;
/
```

6.4. Variables en `insert`, `update`, `delete`

- Se utilizan como un valor inmediato

```
declare
  vempno number;
begin
  vempno := 100;
  insert into empleados(empno, salario, nombre)
         values( vempno, 1000, 'Manolo');
  update empleados
         set salario = salario + 100
         where empno = vempno;
  delete from empleados where empno = vempno;
end;
/
```

6.5. Ventaja de los tipos `%rowtype` en `insert`

- Una variable `%ROWTYPE` se puede usar en un `insert`
- El resultado puede ser más limpio

```
DECLARE
  vempleado empleados%ROWTYPE;
BEGIN
  vempleado.empno := 4;
  vempleado.salario := 3000;
  vempleado.nombre := 'Susana';
  insert into empleados values vempleado;
end;
/
```

6.6. Recorrer consultas

- `for` puede recorrer las filas de una consulta
- En cada vuelta, la variable del `for` tiene el `%ROWTYPE` de la consulta

```
DECLARE
  salariototal number := 0;
  numeroempleados number := 0;
  mediasalario number := 0;
begin
  for empleado in (select * from empleados) loop
    dbms_output.put_line(empleado.nombre || ' con salario ' || empleado.salario );
    numeroempleados := numeroempleados + 1;
    salariototal := salariototal +empleado.salario;
  end loop;
  dbms_output.put_line( 'Hay ' || numeroempleados || ' empleados ');
  dbms_output.put_line( 'Sus sueldos suman ' || salariototal );

  mediasalario := salariototal / numeroempleados;
  dbms_output.put_line( 'La media de los sueldos es ' || mediasalario);
end;
/
```

7. Control de errores

- Si se produce un error, se lanza una **excepción**
  - Se interrumpe el flujo de programa
  - Hasta que se **atrapa**
  - Puede atraparse en cada bloque/función/procedimiento

```
DECLARE
  -- Declaraciones
BEGIN
  -- Ejecucion
EXCEPTION
  -- Excepcion
END;
```

(youtube)

7.1. Sección *exception*

- Se especifican varios tipos de excepción que se esperan

```
DECLARE
  -- Declaraciones
BEGIN
  -- Ejecucion
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    -- Se ejecuta cuando ocurre una excepcion de tipo NO_DATA_FOUND
  WHEN ZERO_DIVIDE THEN
    -- Se ejecuta cuando ocurre una excepcion de tipo ZERO_DIVIDE
  WHEN OTHERS THEN
    -- Se ejecuta cuando ocurre una excepcion de un tipo no tratado
    -- en los bloques anteriores
END;
```

7.2. Ejemplo

```
create table empleados( empno number(20), salario number(8,2), nombre varchar(255));
insert into empleados(empno,salario,nombre) values (1,2000,'María');
insert into empleados(empno,salario,nombre) values (2,1000,'Juan');

DECLARE
  unavariable varchar(255);
BEGIN
  select nombre into unavariable from empleados;
```

```
EXCEPTION
  WHEN TOO_MANY_ROWS THEN
    dbms_output.put_line('select...into ha devuelto más de una fila!');
  WHEN NO_DATA_FOUND THEN
    dbms_output.put_line('select...into no ha devuelto ninguna fila!');
  WHEN OTHERS THEN
    dbms_output.put_line('Error no previsto:' || sqlcode || ': ' || sqlerrm );
END;
```

7.3. Excepciones predefinidas

- Estas son algunas (hay muchas)

```
NO_DATA_FOUND      TOO_MANY_ROWS   ACCESS_INTO_NULL
INVALID_NUMBER     NO_DATA_FOUND   VALUE_ERROR
ROWTYPE_MISMATCH   ZERO_DIVIDE
```

https://www.techonthenet.com/oracle/exceptions/named_system.php

7.4. SQLCODE y SQLERRM

- Funciones predefinidas
- `SQLCODE`: Número de error (independiente del idioma)
- `SQLERRM`:
  - Sin parámetros: Mensaje de error en el idioma de la base de datos
  - Con un parámetro: mensaje de ese `sqlcode`

```
DECLARE
  result NUMBER;
BEGIN
  SELECT 1/0 INTO result FROM DUAL;
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.put_line('Error:'||TO_CHAR(SQLCODE));
    DBMS_OUTPUT.put_line(SQLERRM);
END;
```

7.5. ¿Qué excepciones hay que atrapar?

- Una excepción debe ser *excepcional*
- El flujo normal de programa debe evitarlas
  - Un `select into` debe estar hecho para devolver como mucho una fila
  - Antes se debería hacer `select count()` con un `if` para comprobar que devuelve una fila
- Solo se atrapa si se puede solucionar el error, si no se deja pasar
  - No tiene sentido que imprimamos *Ha habido un error*, pero que la función/procedimiento que nos llamó no lo sepa
  - Si nadie lo atrapa, llega hasta el usuario, que es el que tiene que enterarse y el que lo podrá arreglar

7.6. Excepciones de usuario

- En ocasiones queremos enviar un mensaje de error personalizado
- Están disponibles los números de error entre `-20001` y `20999`
- Se pueden atrapar con `when others` y comprobarse con `SQLCODE`

```
DECLARE
  n number;
BEGIN
  SELECT count(*) into n from empleados
  if( n < 10 ) then
    RAISE_APPLICATION_ERROR(-20001,'La empresa necesita al menos 10 empleados');
  end if;
EXCEPTION
  WHEN OTHERS THEN
    if( sqlcode = -20001) then
      dbms_output.put_line('Pocos empleados');
    end if;
END;
```

8. Disparadores (*triggers*)

- Las funciones y procedimientos se invocan desde *fuera* de la base de datos
- Los disparadores los lanza la propia base de datos en respuesta a eventos
- Cada tabla tiene sus propios eventos
- Los disparadores se pueden lanzar *antes* o *después* del evento
- Los disparadores se pueden lanzar una vez por cada fila afectada, o una vez para toda la sentencia SQL

(youtube 1) (youtube 2) (youtube 3)

8.1. Sintaxis (casi) completa

```
CREATE [OR REPLACE] TRIGGER <nombre_trigger>
  {BEFORE|AFTER}
  {DELETE|INSERT|UPDATE [OF col1, col2, ..., colN]
  [OR {DELETE|INSERT|UPDATE [OF col1, col2, ..., colN]...]}
  ON <nombre_tabla>
  [FOR EACH ROW [WHEN (<condicion>)]]
DECLARE
  -- variables locales
BEGIN
  -- Sentencias
[EXCEPTION]
  -- Sentencias control de excepcion
END;
```

8.2. Ejemplo: Al menos 10 empleados

```
CREATE or replace TRIGGER personal_minimo
BEFORE DELETE ON empleados
declare
  n number;
begin
  SELECT count(*) into n from empleados;
  if( n < 10 ) then
    RAISE_APPLICATION_ERROR(-20001,'La empresa necesita al menos 10 empleados');
  end if;
end;
/
```

8.3. Eventos DML

Evento DML	
`delete`	Borrado de una fila
`insert`	Inserción de una fila
`update`	Modificación de fila
`update of`	Modificación de un campo de una fila
`instead of`	Anula la orden, pero no provoca un error

- Se pueden combinar para un mismo *trigger*
  - Las funciones `INSERTING`, `UPDATING` y `DELETING` sirven para diferenciar por qué se ha lanzado

8.3.1. Ejemplo de *trigger* en varios eventos DML

```
CREATE or replace TRIGGER ejemplo_or
BEFORE DELETE OR UPDATE OR INSERT ON empleados
begin
  case
    when inserting THEN
      dbms_output.put_line('Insertando empleados');
    when updating then
      dbms_output.put_line('Actualizando empleados');
    when deleting then
      dbms_output.put_line('Borrando empleados');
    else
      dbms_output.put_line('Inesperado');
  end case;
end;
/
```

8.4. Eventos DDL

Evento DDL	
ALTER	Modificación de objetos
COMMENT	
CREATE	Creación de objetos
DDL	
DROP	Borrado de Objetos
GRANT	Otorgar privilegios
RENAME	
REVOKE	Quitar privilegios
TRUNCATE	

8.5. Eventos de sistema

Evento DDL	
ANALYSE	
ASSOCIATE STATISTICS	
AUDIT	
DISASSOCIATE STATISTICS	
LOGON	Entrada de usuario
LOGOFF	Salida de usuario
NOAUDIT	
RENAME	
SERVERERROR	
STARTUP	Servidor arrancado
SHUTDOWN	Servidor parado
SUSPEND	

8.6. `for each row`

- Por defecto, un *trigger* se lanza una vez por cada sentencia SQL que provoque cambios
- Si se especifica `for each row`, se lanza una vez por cada fila cambiada

8.7. Momentos del evento

- Se puede lanzar
  - `before`
  - `after`
  - `instead of`: No se ejecuta el SQL, sino otro alternativo. Útil para vistas modificables.
- Las variables `:old` y `:new` existen en los *triggers* tipo `for each row`
  - `:old`: Variable tipo `%rowtype` con los datos antiguos de la fila
  - `:new`: Datos nuevos de la fila

8.7.1. Resumen de momentos y variables

Momento	Evento	:old	:new
before	delete	Lectura	
before	insert		Lectura/escritura
before	update	Lectura	Lectura/escritura
after	delete	Lectura	
after	insert		Lectura
after	update	Lectura	Lectura

8.8. Ejemplo típico: Autonuméricos

```
create sequence empleado_empno_seq;
CREATE or replace TRIGGER asignar_empleado_empno
BEFORE INSERT ON empleados
for each row
begin
  if :new.empno is null then
    :new.empno = empleado_empno_seq.nextval;
  end if;
end;
/
```

8.9. Ejemplo típico: auditoría

- Se necesitan los datos del usuario que creó y modificó un empleado por última vez

```
create table empleados( empno number(20), salario number(8,2), nombre varchar(255));
insert into empleados(empno,salario,nombre) values (1,2000,'María');
insert into empleados(empno,salario,nombre) values (2,1000,'Juan');


alter table empleados add (
  createdby varchar(255),
  createddate timestamp,
  modifiedby varchar(255),
  modifieddate timestamp
);
```

8.9.1. Actualizar campos `createdXXXX`

```
create or replace trigger audit_creacion_empleados
  before insert
  on empleados
  for each row
begin
  dbms_output.put_line('Empleado ' || :new.nombre || ' creado por:' || user );
  :new.createdby := user;
  :new.createddate := systimestamp;
end;
/
```

8.9.2. Actualizar campos `modifiedXXXX`

```
create or replace trigger audit__modificacion_empleados
  before update
  on empleados
  for each row
begin
  dbms_output.put_line('Empleado ' || :new.nombre || ' modificado por:' || user );
  :new.modifiedby := user;
  :new.modifieddate := systimestamp;
end;
/
```

8.10. Ejemplo típico: desnormalización

- Imaginemos que necesitamos saber la masa salarial total

```
create or replace function MASA_SALARIAL return number
as
  total number;
begin
  select sum(salario) into total from empleados;
  return total;
end;
/
```

- Ventajas: simple, rendimiento es escritura
- Desventajas: rendimiento en lectura, se consulta cada vez

8.10.1. Con triggers

```
create table cantidadesprecalculadas(nombre varchar(255), valor number(20,2));

create or replace trigger mantener_masa_salarial
after insert or update or delete
on empleados
for each row
begin
  if inserting then
    update cantidadesprecalculadas set valor=valor + :new.salario where nombre = 'masasalarial';
  elsif deleting then
    update cantidadesprecalculadas set valor=valor - :old.salario where nombre = 'masasalarial';
  elsif updating then
    update cantidadesprecalculadas set valor=valor - :old.salario + :new.salario where nombre = 'masasalarial';
  end if;
end;
/

create or replace function MASA_SALARIAL_PRECALCULADA return number
as
  total number;
begin
  select valor into total from cantidadesprecalculadas where nombre='masasalarial';
  return total;
end;
/
```

8.10.2. Ventajas de la desnormalización

- La tabla `masasalarial` tiene un atributo calculado, que depende de otros datos
  - No está normalizada
  - Es más complicado que la función `MASA_SALARIAL`
  - Es más lento en escritura
- Pero es mucho más eficiente en lectura, porque no se calcula cada vez que se consulta

8.10.3. Prueba de rendimiento

- Estas consultas simples suelen estar ya optimizadas por el gestor de base de datos

```
set serveroutput off;
begin
  for i in 1 .. 10000 loop
     insert into empleados(empno,salario,nombre)
     values(100+i, i, 'Nombre Inventado');
  end loop;
end;
/
set serveroutput on;
```

```
declare
  numero number;
begin
  for i in 1 .. 1000 loop
    select /*+ NO_RESULT_CACHE */ masa_salarial_precalculada into numero from dual;
  end loop;
end;
/
```

9. Referencias

- Ejercicios
- Formatos:
  - Transparencias
  - PDF
  - Página web
  - EPUB
- Creado con:
  - Emacs
  - org-re-reveal
  - Latex
- Alojado en Github