

# BASES DE DATOS DISTRIBUIDAS

Sistemas informáticos I

## 3.4 PROCEDIMIENTOS ALMACENADOS Y TRIGGERS

Bases de datos distribuidas

## APLICACIONES DENTRO DE LA BD

- Vamos a considerar dos formas de definir funcionalidad de usuario **dentro** de una base de datos SQL:
  - Funciones y procedimientos almacenados: Se ejecutan a petición del usuario
  - Triggers: Se ejecutan cuando ocurre un evento asociado a una tabla (p.ej., una inserción o una actualización)
- Conjunto de sentencias SQL y lógica de programa compilado, verificado y almacenado en el SGBD
- Tratado por el servidor como cualquier otro objeto de la BBDD y almacenado en el catálogo de la misma.
- Al igual que cualquier otra metaestructura se gestionan con los comandos:
  - CREATE
  - ALTER (habitualmente se usa DROP + CREATE)
  - DROP

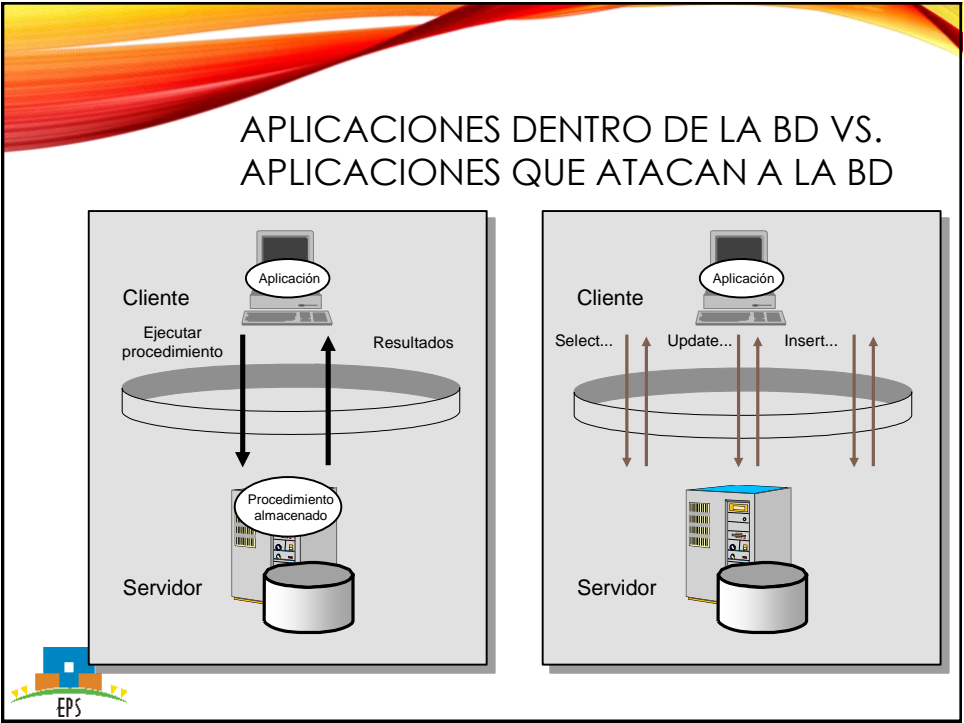


## APLICACIONES DENTRO DE LA BD

- No todos los SGBD los soportan:
  - No hay estándares: implementación propia de cada fabricante
  - PL/SQL (*Procedural Language for SQL*) definición de lenguaje incorporado en Oracle 6 que posteriormente ha sido asumida por otros fabricantes (p.ej., IBM DB2)
  - PostgreSQL soporta la incorporación de funcionalidad dentro de la base de datos desde 1997
    - El cuerpo de la función es una cadena de caracteres que puede que se pasa al SGBD indicándole el lenguaje en el que está escrito
      - Soportados por omisión:
        - SQL
        - PL/pgSQL
        - C, solo usuarios privilegiadas
      - A nivel administrativo se pueden cargar módulos con otros lenguajes:
        - PL/PERL
        - PL/TCL
        - PL/Python
        - ...



<https://www.oracle.com/database/technologies/appdev/plsql.html>



### FUNCIONES Y PROCEDIMIENTOS ALMACENADOS

- Mejoran de rendimiento frente al SQL interactivo
- Su acceso está controlado por los mecanismos de seguridad
- Aceptan parámetros de entrada
- Los procedimientos se invocan, las funciones se incluyen dentro de una sentencia SQL
- La sintaxis se valida en tiempo de ejecución, no durante la creación



## PRIMEROS EJEMPLOS SENCILLOS

```
CREATE PROCEDURE num_empleados_dpto (idDpto INT, OUT n INT)
BEGIN
    SELECT count(*) INTO n FROM Empleado WHERE idDepartamento = idDpto;
END //

# Invocación a un procedimiento
CALL num_empleados_dpto(1, @n);
SELECT @n;

CREATE FUNCTION f_num_empleados_dpto (idDpto INT)
RETURNS INT
BEGIN
    DECLARE n INT;
    SELECT count(*) INTO n FROM Empleado WHERE idDepartamento = idDpto;
    RETURN n;
END //

# Invocación de una función
SELECT f_num_empleados_dpto(1) as n;
```



## FUNCIONALIDAD EN POSTGRESQL

```
CREATE FUNCTION nombre_funcion (tipos-argumentos)
RETURNS integer AS $$
DECLARE
    -- declaraciones
BEGIN
    -- cuerpo
END;
$$ LANGUAGE lenguaje;
```

Puede ser ‘

```
CREATE FUNCTION clean_emp() RETURNS void AS '
DELETE FROM emp
WHERE salary < 0;
' LANGUAGE SQL;
```




## FUNCIONALIDAD EN POSTGRESQL

Paso de argumentos referenciados por nombre o por posición

```
CREATE FUNCTION add_em(integer, integer) RETURNS integer AS $$  
  SELECT $1 + $2;  
$$ LANGUAGE SQL;  
  
SELECT add_em(1, 2) AS answer;  
  
answer  
-----  
      3
```

```
CREATE FUNCTION add_em(x integer, y integer) RETURNS integer AS $$  
  SELECT x + y;  
$$ LANGUAGE SQL;
```



## FUNCIONALIDAD EN POSTGRESQL


Paso de tuplas: el tipo será el nombre de la tabla que las contiene

```
CREATE TABLE emp (  
  name      text,  
  salary    numeric,  
  age       integer,  
  cubicle   point  
);  
  
INSERT INTO emp VALUES ('Bill', 4200, 45, '(2,1)');
```

```
CREATE FUNCTION double_salary(emp) RETURNS numeric AS $$  
  SELECT $1.salary * 2 AS salary;  
$$ LANGUAGE SQL;
```

```
SELECT name, double_salary(emp.*) AS dream  
FROM emp  
WHERE emp.cubicle ~= point '(2,1)';
```

name	dream
Bill	8400



## FUNCIONALIDAD EN POSTGRESQL

Retorno de una función con SQL "puro":

- Si la última cláusula es un SELECT → primera fila devuelta por la consulta
- Si la última cláusula no es un SELECT → debe tener cláusula RETURNING

```
CREATE FUNCTION tfl (accountno integer, debit numeric) RETURNS integer AS $$  
    UPDATE bank  
        SET balance = balance - debit  
        WHERE accountno = tfl.accountno  
    RETURNING balance;  
$$ LANGUAGE SQL;
```



## FUNCIONALIDAD EN POSTGRESQL

Tipos de datos en PL/pgSQL:

- Todos los tipos de variable definidos para SQL son válidos en PL/pgSQL
- Sintaxis general para la declaración de variables:  
name [ CONSTANT ] type [ NOT NULL ] [ { DEFAULT | := } expression ];
- Se pueden definir tipos en base a atributos o tuplas:  
micampo mitabla.campo%TYPE;  
mitupla mitabla%ROWTYPE;

```
CREATE OR REPLACE FUNCTION trae_pelicula (integer) RETURNS text AS $$  
DECLARE  
    pelicula_id ALIAS FOR $1;  
    encontrada_pelicula pelicula%ROWTYPE;  
BEGIN  
    SELECT INTO encontrada_pelicula * FROM pelicula WHERE id = pelicula_id;  
    RETURN encontrada_pelicula.titulo || " (" || encontrada_pelicula.agno || ")";  
END;  
$$ LANGUAGE plpgsql;
```



# FUNCIONALIDAD EN POSTGRESQL

Argumentos de salida en PL/pgSQL: como tuplas

```
CREATE OR REPLACE FUNCTION hi_lo(  
    a NUMERIC,  
    b NUMERIC,  
    c NUMERIC,  
    OUT hi NUMERIC,  
    OUT lo NUMERIC)  
AS $$  
  
BEGIN  
    lo = LEAST(a, b, c);  
    hi = GREATEST(a, b,c);  
END; $$  
LANGUAGE plpgsql;
```

```
1 SELECT hi_lo(10,20,30);
```

hi_lo
(30,10)

```
1 SELECT * FROM hi_lo(10,20,30);
```

hi	lo
30	10



# FUNCIONALIDAD EN POSTGRESQL

Condicionales en PL/pgSQL: IF...THEN...ELSE IF...ELSE...END IF

```
CREATE OR REPLACE FUNCTION cadena_mas_larga(text, text) RETURNS int4 AS $$  
DECLARE  
    in_uno ALIAS FOR $1;  
    in_dos ALIAS FOR $2;  
    lon_uno int4;  
    lon_dos int4;  
    result int4;  
BEGIN  
    lon_uno := (SELECT LENGTH(in_uno));  
    lon_dos := (SELECT LENGTH(in_dos));  
    IF lon_uno > lon_dos THEN RETURN lon_uno;  
    ELSE RETURN lon_dos;  
    END IF;  
END;  
$$ LANGUAGE plpgsql;
```



## FUNCIONALIDAD EN POSTGRESQL

Bucles en PL/pgSQL: WHILE LOOP [WHILE|FOR]...END LOOP

```
CREATE FUNCTION cuentac(text,text) RETURNS INT4 AS $$
DECLARE
    intext ALIAS FOR $1; inchar ALIAS FOR $2;
    lon int4; resultado int4;
    i int4; tmp char;
BEGIN
    lon:= length(intext); i:=1;
    resultado:=0;
    WHILE i<= lon LOOP
        tmp := substr(intext,i,1);
        IF tmp = inchar THEN
            resultado := resultado +1;
        END IF;
        i:=i+1;
    END LOOP;
    RETURN resultado;
END
$$ LANGUAGE plpgsql;
```



## FUNCIONALIDAD EN POSTGRESQL

Excepciones: RAISE { NOTICE | EXCEPTION}

```
CREATE OR REPLACE FUNCTION suma(int4, int4) RETURNS int4 AS $$
DECLARE
    inicio ALIAS FOR $1; fin ALIAS FOR $2;
    resultado int;
BEGIN
    IF (inicio <1) THEN
        RAISE EXCEPTION "inicio debe ser mayor que 1";
    ELSE
        IF(inicio <= fin) THEN
            resultado := (fin+1)*fin/2 - (inicio-1)*inicio/2;
        ELSE
            RAISE EXCEPTION "El valor inicial % debe ser menor que el final
            %", inicio, fin;
        END IF;
    END IF;
    RETURN resultado;
END
$$ LANGUAGE plpgsql;
```





# FUNCIONALIDAD EN POSTGRESQL

## Cursores

```
CREATE OR REPLACE FUNCTION pelis_con_letra (text) RETURNS text AS $$
DECLARE
  character ALIAS FOR $1; temporal record;
  tmp_character text;
  total text;
BEGIN
  total := "";
  FOR temporal IN SELECT titulo FROM pelicula LOOP
    tmp_character := substr(temporal.titulo,1,1);
    IF tmp_character = character THEN
      total := total || temporal.titulo || " . ";
    END IF;
  END LOOP;
  RETURN total;
END;
$$ LANGUAGE plpgsql;
```



# FUNCIONALIDAD EN POSTGRESQL

## Retorno de tuplas:

- Un único elemento → RETURN NEXT
  - RETURNS RECORD
  - RETURNS nombreTabla%ROWTYPE
  - RETURNS TABLE
  - Argumentos de salida
- Múltiples elementos → RETURN QUERY
  - RETURNS TABLE(columnas)
  - RETURNS SET OF tipo
  - Argumentos de salida

```
CREATE FUNCTION listchildren(text) RETURNS SETOF text AS $$
  SELECT name FROM nodes WHERE parent = $1
$$ LANGUAGE SQL;
```

name	parent
Top	
Child1	Top
Child2	Top
Child3	Top
SubChild1	Child1
SubChild2	Child1

(6 rows)

```
SELECT listchildren('Top');
listchildren
```

name	listchildren
Child1	
Child2	
Child3	

(3 rows)

```
SELECT name, listchildren(name) FROM nodes;
```

name	listchildren
Top	Child1
Top	Child2
Top	Child3
Child1	SubChild1
Child1	SubChild2

(5 rows)



## FUNCIONALIDAD EN POSTGRESQL

Retorno de tuplas

```
CREATE OR REPLACE FUNCTION storeopeninghours_tostring(numeric)
  RETURNS SETOF RECORD AS $$
DECLARE
  open_id ALIAS FOR $1;
  result RECORD;
BEGIN
  RETURN QUERY SELECT '1', '2', '3';
  RETURN QUERY SELECT '3', '4', '5';
  RETURN QUERY SELECT '3', '4', '5';
END
$$;
```



## FUNCIONALIDAD EN POSTGRESQL

Retorno de tuplas

```
1 CREATE OR REPLACE FUNCTION get_film (p_pattern VARCHAR)
2 RETURNS TABLE (
3   film_title VARCHAR,
4   film_release_year INT
5 )
6 AS $$
7 BEGIN
8   RETURN QUERY SELECT
9     title,
10    cast( release_year as integer)
11 FROM
12   film
13 WHERE
14   title LIKE p_pattern ;
15 END; $$
16
17 LANGUAGE 'plpgsql';
```



# TRIGGERS

- Se pueden considerar un tipo especial de procedimiento almacenado
- La principal diferencia es que un trigger se invoca **de forma automática** en respuesta a una modificación de datos en una tabla
- Características:
  - Mecanismo alternativo para validar la integridad de los datos
  - Ofrecen una funcionalidad equivalente a un planificador de tareas dentro de la propia BBDD
  - Mecanismo sencillo para realizar una auditoría de datos independiente de la aplicación

El diagrama ilustra la arquitectura de triggers. En la parte superior, un 'Cliente' con una 'Aplicación' envía una 'Update...' a un servidor. El servidor contiene un 'Disparador' (trigger) que se activa al recibir la actualización. El disparador interactúa con una base de datos ('Disparadores') y puede generar acciones adicionales.

# TRIGGERS

- Se pueden considerar un tipo especial de procedimiento almacenado
- La principal diferencia es que un trigger se invoca de forma automática en respuesta a una modificación de datos en una tabla
- A la hora de crear un trigger se debe especificar el evento que los dispara

```
CREATE TRIGGER Proyecto_before_delete BEFORE DELETE ON Proyecto
FOR EACH ROW
BEGIN
    INSERT INTO HistProyecto (idProyecto, f_inicio, idResponsable, f_borrado)
    (SELECT idProyecto, f_inicio, idResponsable, now() FROM Proyecto
     WHERE idProyecto = OLD.idProyecto);
END //
```

- OLD vs. NEW

## EJEMPLO DE TRIGGER EN POSTGRESQL

```
CREATE OR REPLACE FUNCTION tr_function()
RETURNS TRIGGER
AS $$
BEGIN
    NEW.c3 = NEW.c1 + NEW.c2
    RETURN NEW
END;
$$
LANGUAGE 'plpgsql';

CREATE TRIGGER tr BEFORE INSERT ON tableName
FOR EACH ROW EXECUTE
PROCEDURE tr_function();
```

