# Teórico 4

# Triggers y Procedimientos Almacenados (Server Programming)

# Procedimientos Almacenados (Store Procedure)

- Un procedimiento almacenado es un procedimiento (programa) que contienen comandos SQL que son almacenados en el servidor de base de datos. Estos procedimientos pueden ser invocados por un cliente.

- Poseen parámetros de entrada y salida.

- Están disponibles desde el estándar SQL:1999.

- Cada motor de base de datos tiene su propio lenguaje. Por ejemplo PL/SQL en Oracle, PL/pgSQL en Postgres. MySQL sigue la sintaxis SQL:2003.

- También algunos motores de Base de Datos (por ej. PostgreSQL) permiten la definición de procedimientos en lenguaje de programación, como Java, C, C++, Python, etc.

# Ventajas

- Rendimiento: al ser ejecutados por el motor de bases de datos ofrecen un excelente rendimiento, ya que no es necesario transportar datos a ninguna parte.

- Los procedimientos almacenados son analizados y optimizados en el momento de su creación. Al contrario cuando se envía una instrucción SQL desde un cliente tiene que ser compilada y optimizada cada vez que son enviadas por el cliente.

- Reducen el tráfico de RED.

# Ventajas (sigue)

- Centralización: Al formar parte de la Base de datos están siempre disponibles.

- Seguridad: Facilitan la administración de seguridad. Por ejemplo, se puede conceder permisos a un usuario para ejecutar un determinado procedimiento almacenado, aunque el usuario no disponga de los permisos necesarios sobre los objetos afectados por las acciones individuales de dicho procedimiento.

- Encapsulamiento: Por ejemplo, un usuario puede invocar un procedimiento para eliminar un cliente, pero no se entera de los detalles de la acción.

# Desventajas

- Debido a que cada motor de base de datos implementa sus propias definiciones de procedimientos almacenados, esto hace muy difícil la migración a otro motor de base de datos.

# Procedimiento Almacenados en MySQL

- MySQL incorpora triggers y Procedimientos Almacenados desde la versión 5.

- MySQL sigue la sintaxis SQL:2003 para procedimientos almacenados. Se escriben en el lenguaje SQL/PSM.

# Sintaxis

CREATE PROCEDURE *sp_name* ([*parameter*[,...]]) [*characteristic* [...]]
*routine_body*

CREATE FUNCTION *sp_name* ([*parameter*[,...]]) RETURNS *type* [*characteristic* [*characteristic*]] *routine_body*

*parameter*: [ IN | OUT | INOUT ] *param_name type.*

*type*: cualquier tipo de datos de MySQL.

*characteristic*: LANGUAGE SQL
|{ CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
| SQL SECURITY { DEFINER | INVOKER }
| COMMENT '*string*'

*routine_body*: procedimientos almacenados o comandos SQL válidos

# Sintaxis del Lenguaje

- Sentencia compuesta BEGIN ... END (bloques).
- Sentencia DECLARE (declaraciones).
- Variables.
- Condiciones y handlers (manejo de excepciones).
- Cursores (para recorrido de tablas).
- Constructores de control de flujo.

# Bloques

- Los bloques de código en los procedimientos almacenados están delimitados por las sentencias BEGIN ... END.

- El programa puede tener un bloque principal y subBloques anidados, cada subBloque puede tener sus manejadores de excepciones.

# DECLARE

- El comando DECLARE se usa para definir varios componentes locales de una rutina:
  - Variables locales
  - Condiciones y handlers.
  - Cursores.

# Variables

**Declaración**

DECLARE *var_name*[,...] *type* [DEFAULT *value*]

Ejemplo DECLARE x int;

**Asignación**

- Asignación clásica: SET *var_name* = *expr* [, *var_name* = *expr*] ...

  Ejemplo SET x = 10;

- Asignación con SELECT:
  SELECT *col_name*[,...] INTO *var_name*[,...] *table_expr*

  Ejemplo:SELECT max(precio) INTO x FROM producto;

# Declaración de Handler

DECLARE *handler_type* HANDLER FOR *condition_value*[,...] *sp_statement*

*handler_type*: CONTINUE | EXIT | UNDO

EXIT: detiene la ejecución.
CONTINUE: continúa la ejecución
UNDO: deshace lo que hizo.

*condition_value*: {SQLSTATE *sqlstate_value*
| *condition_name*
| SQLWARNING
| NOT FOUND
| SQLEXCEPTION
| *mysql_error_code* }

# Tipos de Condiciones

*mysql_error_code: s*on códigos de error específicos de MySQL.

**SQLSTATE:**
Es estándar, cualquier valor SQLSTATE coincide con cualquier motor de base de datos, ya sea Oracle, PostgreSQL, Firebird, etc.

**SQLWARNING, NOT FOUND o SQLEXCEPTION:**
Los SQLWARNING son todos los SQLSTATE que empiezan con 01. Los NOT FOUND son todos los SQLSTATE que empiezan con 02. Y los SQLEXCEPTION son todos los demás.

**Nota 1***:* Listado de códigos de errores de MySQL:
*http://dev.mysql.com/doc/refman/5.5/en/error-handling.html*

**Nota 2***:* Mysql desde la versión 5.5 permite crear y lanzar excepciones por parte del usuario.

# Cursores

- Declaración

  DECLARE *cursor_name* CURSOR FOR *select_statement.*

- Operaciones sobre cursores
  - Apertura del cursor: OPEN *cursor_name*.
  - Avance del cursor(FETCH): Este comando trata el siguiente registro (si existe) usando el cursor abierto ya especificado, y avanza el puntero del cursor, deja el valor del registro en variables. Su Sintaxis:

    FETCH *cursor_name* INTO *var_name* [, *var_name1*] ...
  - Cierre del Cursor: CLOSE *cursor_name .*

# Sentencias de Control de Flujo

Las sentencias de control de flujo de programa son:

- IF THEN ELSE
- CASE
- LOOP
- LEAVE
- ITERATE
- REPEAT
- WHILE

# Sentencia REPEAT

- Su sintaxis es:

REPEAT
*statement_list*
UNTIL *search_condition* END REPEAT

- Semántica: El/Los comando/s dentro de un comando REPEAT se repite hasta que la condición *search_condition* es verdadera.

# Modificación de Procedimientos

Se utiliza el comando ALTER, su sintaxis es:

ALTER {PROCEDURE | FUNCTION} *sp_name* [*characteristic* ...]

*characteristic:*

{ CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }

| SQL SECURITY { DEFINER | INVOKER }

| COMMENT '*string*'

Este comando puede usarse para cambiar las características de un procedimiento o función almacenada

# Excepciones

- Desde Mysql 5.5 se permite al usuario lanzar sus propias excepciones:

```
Signal sqlstate '45000' set message_text ='lanzo un excepción'
```

# Ejemplo

```
CREATE PROCEDURE cursordemo(IN parametro int )
BEGIN
    DECLARE done INT DEFAULT 0;
    DECLARE a1 CHAR(16);
    DECLARE b1,c1 INT;

    DECLARE cursor1 CURSOR FOR SELECT a,b,c FROM tabla1 where id<parametro;

    DECLARE CONTINUE HANDLER FOR SQLSTATE '02000'
                        BEGIN
                          SET done = 1;
                        END;

    OPEN cursor1;
    REPEAT
            FETCH cursor1 INTO a1, b1, c1;
            IF NOT done
              THEN
                IF b1 < c1
                  THEN INSERT INTO tabla2 VALUES (a1,b1);
                  ELSE INSERT INTO tabla3 VALUES (a1,c1);
                END IF;
             END IF;
    UNTIL done END REPEAT;
    CLOSE cursor1;
END
```

# Invocación de procedimientos

La invocación de procedimientos se realiza mediante la sentencia CALL, su sintaxis es:

CALL *nombreProcedimiento* ([*parametroActual*[, ...]])

Ejemplo:

CALL cursordemo(2);

# Nota

Para más detalle sobre el lenguaje de funciones y procedimientos almacenados de MySQL consultar el manual:

http://dev.mysql.com/doc/refman/5.5/en/create-procedure.html

# Funciones Almacenadas en PostgreSQL

Postgres implementa Funciones almacenadas, estas funciones almacenadas pueden ser escrita en diferentes lenguajes, como ser: PL/pgSQL, C, PL/Java, etc.

A continuación se presenta el lenguaje PL/pgSQL.

# Estructura de Pl/pgSQL

PL/pgSQL es un lenguaje orientado a bloques. La estructura básica del lenguaje se define de la siguiente manera:

**CREATE** [**OR REPLACE**] **FUNCTION** *nombreFuncion* ([*parametro*[,...]])
[**RETURNS** *tipoRetorno*] **AS** $$
[**DECLARE**
    *declaración de variables locales de la función*]
**BEGIN**
    *sentencias propias de la función*
[**EXCEPTION**
        **WHEN** *condición* [ **OR** *condición* ... ]
        **THEN** *sentencias para manejar la excepción*
        [**WHEN** *condición* [ **OR** *condición* ... ]
        **THEN** *sentencias para manejar la excepción* ... ]
]
**END;**
$$ **LANGUAGE** 'plpgsql';

# Parámetros

•Su sintaxis es:

[ IN | OUT | INOUT ] [*nombreParametro*] *tipo*

Si el nombre del parámetro no se especifica se accede por $n, de acuerdo al número del parámetro.

Ejemplo:

```
CREATE FUNCTION sum_n_product(int, y int, OUT sum int, OUT prod int) AS
$$
BEGIN
  sum := $1 + y;
  prod := $1 * y;
END;
$$ LANGUAGE plpgsql;
```

# Declaraciones de variables

- Su sintaxis para variables locales es:

*nombre* [ CONSTANT ] *tipo* [ NOT NULL ] [ { DEFAULT | := } *expresión* ];

- Su sintaxis para alias de parámetros es:

  *nombreAlias* ALIAS FOR *$n;*

  Donde %n es el número del parámetro

# Asignación de variables

*nombreVariable := expresión;*

# Cursores

Su sintaxis resumida es:

- DECLARE *nombre* CURSOR FOR *sentenciaSelect*

Sentencia para manipular cursores Fetch, Close, Move. La variable FOUND indica si se recupero o no una fila.

Sintaxis completa para cursores:

http://www.postgresql.org/docs/9.3/interactive/sql-declare.html

# Bloques

- Los bloques de código en los procedimientos almacenados están delimitados por las sentencias BEGIN ... END.

- El programa puede tener un bloque principal y subBloques anidados, cada subBloque puede tener sus manejadores de excepciones.

# Sentencias de Control de Flujo

Pl/pgSQL provee sentencias de control flujo, tales como:

- LOOP
- CASE
- EXIT
- IF THEN ELSE
- FOR
- WHILE

# Sentencia FOR

CASO 1 : Sentencia FOR de proceso cíclico numérico

FOR *variable* IN *inicio*.. *fin* LOOP
*cuerpo del for*
END LOOP;

Se puede usar la sentencia FOR de forma inversa

FOR *variable* REVERSE *fin* .. *inicio* LOOP
*cuerpo del for*
END LOOP;

Ejemplo:

```
FOR inc IN 1..10 LOOP
        factorial := factorial * inc;
END LOOP;
```

# Sentencia FOR (Sigue)

- CASO 2: Sentencia FOR aplicada a procesamiento de registros sql (cursores Implícitos):

  FOR *variableRegistro* IN *sentencia_select_sql* LOOP
  *cuerpo del for*
  END LOOP;

  Ejemplo :

```
DECLARE
   registro RECORD;
BEGIN
FOR registro IN SELECT * FROM productos LOOP
   stock := registro.sock_actual + 100;
END LOOP
```

- **Nota:** se pueden declarar registros genéricos como en el ejemplo o registro del tipo de una tabla, por ejemplo persona%rowtype.

# Excepciones

- Postgres permite que el usuario lance sus propias excepciones.

  raise exception 'el mensaje!';

- Mas detalle:

https://www.postgresql.org/docs/9.3/static/plpgsql-errors-and-messages.html

# Invocación de procedimientos

La invocación de procedimientos en Postgres se realiza mediante la sentencia SELECT, su sintaxis es:

SELECT *nombreProcedimiento* ([*parametroActual*[, ...]])

Ejemplo:

SELECT procedimiento(2);

# Mas Información sobre Funciones Almacenadas en Postgres

http://www.postgresql.org/docs/9.3/interactive/plpgsql.html

# Procedimientos Almacenados en Oracle

- Oracle tiene un potente lenguaje de programación para procedimientos almacenados, este lenguaje se denomina PL/SQL.

# Procedimientos Almacenados

CREATE [OR REPLACE] PROCEDURE *nombreProc* ([*parametro*[,...]])
AS
[ [DECLARE]
*declaraciones* ]
BEGIN
*programa*
[ EXCEPTION
*tratamiento de excepciones* ]
END; /

# Funciones Almacenadas

CREATE [OR REPLACE] FUNCTION *nombreFunc*
([*parametro*[,...]])
RETURN *tipo*
AS
[*declaraciones*]
BEGIN
*programa*
[ EXCEPTION
*tratamiento de excepciones* ]
END; /

# Parámetros

*nombreParametro* [ IN | OUT | IN OUT ] *tipo* [ **DEFAULT** *expr* ]

# Declaraciones de variables

*nombreVariable tipo* [NOT NULL] [:= *valor* | DEFAULT *valor*]

# Asignaciones

*nombreVariable := expresion;*

# Bloques

- Los bloques de código en los procedimientos almacenados están delimitados por las sentencias BEGIN ... END.

- El programa puede tener un bloque principal y subBloques anidados, cada subBloque puede tener sus manejadores de excepciones

# Tratamiento de Excepciones

**Begin**

**.....**

**EXCEPTION**

**WHEN** *condicion* [ **OR** *condicion* ... ]

**THEN** *sentencias para manejar la excepción*

[**WHEN** *condición* [ **OR** *condición* ... ]

**THEN** *sentencias para manejar la excepción* ... ]

**End;**

# Excepciones generadas por el usuario

Oracle permite lanzar excepciones definidas por el usuario, para esto se puede utilizar el siguiente procedimiento:

RAISE_APPLICATION_ERROR(*Error_number* INT, *texto TEXT*)

Donde *Error_number* puede estar en te -20000 y -20999.

Otra forma es declarar una excepción y luego lanzarla con RAISE.

# Ejemplo de Manejo de Excepciones

- **DECLARE**
  err_num **NUMBER**;
  err_msg **VARCHAR2**(255);
  result **NUMBER**;

  RAISE_APPLICATION_ERROR(-20011,’prueba de excepción ’) ;

  **EXCEPTION**
  -- captura cualquier excepción
  **WHEN OTHERS THEN**
  err_num := **SQLCODE**;
  err_msg := **SQLERRM**;
  DBMS_OUTPUT.put_line('Error:'||**TO_CHAR**(err_num));
  DBMS_OUTPUT.put_line(err_msg);
  **END**;

# Cursores

- Se utilizan de manera similar a MySQL.

- Declaración:
  DECLARE *cursor_name* CURSOR FOR *select_statement.*

- Operaciones sobre cursores
  - Apertura del cursor: OPEN *cursor_name*.

  - Avance del cursor(FECTH): Este comando trata el siguiente registro (si existe) usando el cursor abierto ya especificado, y avanza el puntero del cursor, deja el valor del registro en variables. Su Sintaxis:
    FETCH *cursor_name* INTO *var_name* [, *var_name1*] ...

  - Cierre del Cursor: CLOSE *cursor_name*

# Cursores (Atributos)

- Los cursores en PL/SQL tienen atributos, estos son:
  - %FOUND: devuelve *true* si la última operación fetch devolvió una fila
  - %NOTFOUND: devuelve *false* si la última operación fetch retorno una fila.
  - %ISOPEN: retorna *true* si el cursor está abierto.
  - %ROWCOUNT: retorna el número de filas del cursor.

# Ejemplo de manejo de cursores

```
CREATE PROCEDURE listar IS

CURSOR cproductos IS SELECT nProd, precio FROM
productos

regProducto cproductos%ROWTYPE;

BEGIN
  OPEN cproductos;
  FETCH cproductos INTO regProducto;
  WHILE cproductos%FOUND  LOOP

        dbms_output.put_line( regProducto.Precio);

  FETCH cproductos INTO regProducto;
  END LOOP;
  CLOSE cproductos;
END;
/
```

# Invocación de procedimientos

La invocación de procedimientos en Oracle se realiza mediante la sentencia BEGIN ... END, su sintaxis es:

[DECLARE
 *declaraciones*]
BEGIN
        *nombreProc* ([*parametroActual*[, ...]])
END

Ejemplo:

```
DECLARE
  nombre_parametro int;
BEGIN
  nombre_parametro  := 3;
  pruebaProc (3,nombre_parametro);
END;
/
```

# Reglas de reescritura

- Los sistemas de reglas de producción son conceptualmente simples, pero hay muchos puntos sutiles implicados en el uso actual de ellos.

# Sistema de Reglas de Postgres

- El sistema de reglas de reescritura de queries es totalmente diferente a los procedimientos almacenados y los triggers. El sistema modifica los queries tomando en consideración las reglas de reescritura definidas.

- Las vistas en postgres se definen con reglas.

# Ejemplo

CREATE VIEW vista AS SELECT * FROM tabla;

Es lo mismo que ejecutar la siguiente secuencia:

CREATE TABLE vista

(la misma lista de atributos de tabla);

CREATE RULE ReescribeVista AS ON SELECT TO vista DO INSTEAD SELECT * FROM tabla;

Nota: Postgres, cuando se define una vista, internamente ejecuta esa secuencia.

# Reglas de actualización

Las reglas de actualización se definen para las operaciones INSERT, UPDATE, or DELETE (evento). Su sintaxis es:

CREATE [ OR REPLACE ] RULE *nombre*
AS ON *evento* TO *tabla*
[ WHERE *condicion* ]
DO [ ALSO | INSTEAD ]
{ NOTHING | *comando* | ( *comando* ; *comando* ... ) }

# Ejemplo de regla update

```
CREATE RULE log_precio_productos
AS ON UPDATE TO productos
WHERE NEW.precio <> OLD.precio
DO ALSO
INSERT INTO productos_log VALUES (NEW.codigo,
    NEW.nombre, OLD.precio, NEW.precio,  current_user,
    current_timestamp);
```

# Disparadores(Trigger)

Un disparador es una orden que ejecuta el DBMS automáticamente ante alguna modificación en la base de datos. Uno de los usos mas importante es el de generar información de auditoria, en el práctico trabajaremos auditoria en MySQL.

Para diseñar un trigger es necesario :

1)Especificar la condición de disparo.

2)Especificar la acciones que se van a ejecutar cuando la condición de disparo de cumpla.

# Condiciones de disparo

Un trigger se puede disparar antes o después de, insertar, modificar o eliminar un registro en una tabla.

# Sintaxis

- En SQL92 no incluye a los trigger, la definición de un trigger no esta normalizado, por esto es que cada RDBMS tiene su forma no normalizada de definición de trigger.
- La Norma SQL1999 incorpora la sintaxis para los trigger, pero de todas formas los RDBMS mantienen su forma no normalizada de definición de trigger.
- Son muy utilizados para la generación de información de auditoria.
- El lenguaje utilizado en la programación de un trigger, es el mismo lenguaje que para la definición de procedimientos almacenados.

**Nota:** La variable contextual OLD almacena temporariamente los viejos valores del registro en la eliminación o actualización.
La variable contextual NEW almacena temporariamente los nuevos valores del registro en la inserción o actualización.

# Triggers en Mysql

- Están disponibles desde la versión 5.0.2
- Su sintaxis es:

  CREATE TRIGGER *nombre_trigger momento_disparo evento_disparo* ON *nombre_tabla*

  FOR EACH ROW

  *sentencia_trigger*

# Triggers en Postgres

Su sintaxis es:

CREATE TRIGGER *nombre* { BEFORE | AFTER }

{ *evento* [ OR ... ] }

ON *tabla* [ FOR [ EACH ] { ROW | STATEMENT } ]

EXECUTE PROCEDURE *nombreFunc* ( *argumentos* )

# Triggers en Oracle

**Su Sintaxis es:**

**CREATE** [ **OR REPLACE** ] **TRIGGER** [ *esquema*. ]*nombre*
{ **BEFORE** | **AFTER** | **INSTEAD OF** }
{ *dml_event_clause*
| *ddl_event* [ **OR** *ddl_eventN* ]...
| *database_event* [ **OR** *database_eventN* ]... }
**ON** {*table* | *view* | [ *esquema*.]**SCHEMA** | **DATABASE** }
**[FOR EACH ROW]**
  [ **WHEN** (*condicion*) ]
    { *pl/sql_block* | *call_procedure_statement* }

# Ejemplo

Se quiere definir un trigger que se dispare antes de la eliminación de un registro para la tabla articulos, y que realice la inserción de un nuevo registro en la tabla articulos_bajas con los valores de nart, descr y precio del articulo eliminado de la tabla articulo, además que inserte el usuario que realizó la operación y la fecha y hora de la eliminación.

# Ejemplo en Mysql

```
delimiter $$
CREATE TRIGGER trigger_baja_articulos
     AFTER  DELETE ON articulos
     FOR EACH ROW
      BEGIN
        INSERT INTO articulos_bajas VALUES (OLD.nart,
            OLD.descr, OLD.precio, CURRENT_USER(), NOW() );
      END;
$$
delimiter ;
```

# Ejemplo en PostgreSQL

```
// crear la función que se ejecutará en el trigger
create function funcion_auditoria () returns trigger as
   $$
   BEGIN
    insert into auditoria values(1,old.nart,now(),old.cant-new.cant);
   return new;
  END;$$
  LANGUAGE 'plpgsql';

// crear el trigger para generar información de auditoria
create trigger trigger_auditoria after update on articulos for each row
     execute procedure funcion_auditoria();
```

# Ejemplo en Oracle

```
CREATE TRIGGER trigger_baja_articulos
    AFTER  DELETE ON articulos
    FOR EACH ROW

     BEGIN
      INSERT INTO articulos_bajas VALUES (:OLD.nart,
          :OLD.descr, :OLD.precio, USER, SYSDATE );
     END;
```

# Ejemplo de Auditoria en Mysql

```
delimiter $$
CREATE TRIGGER trigger_baja_articulos
     AFTER  DELETE ON articulos
     FOR EACH ROW
      BEGIN
        INSERT INTO articulos_bajas VALUES (OLD.nart,
            OLD.descr, OLD.precio, CURRENT_USER(), NOW() );
      END;
$$
delimiter ;
```

Nota: El usuario que queda registrado (dado por CURRENT_USER()) es el usuario con el cual se estableció la conexión con la base de datos. Generalmente en una auditoria se pretende que quede registrado el usuario de la aplicación (que se conecta a la base de datos).

# Trigger MySQL(Cont.)

- Para poder tener acceso desde un trigger al usuario de la aplicación cliente de la base de datos es necesario registrar esta información en una variable de sesión
- Así de esta forma la información queda disponible en la sesión de la base de datos para que pueda ser accedida, por ejemplo por el código de un trigger.

- Registrar datos en variable de sesión.

```
set @id_usuario = 4
```

- Acceso a esta variable.

```
INSERT INTO AUDITORIA VALUES(0,@id_usuario,new.fecha);
```