



EFA
MORATALAZ

*1º CFGS Desarrollo de
Aplicaciones
Multiplataforma/Web*

BASES DE DATOS

JESÚS SANTIAGO RICO

UT5 – PROGRAMACIÓN DE BBDD





EFA
MORATALAZ

*1º CFGS Desarrollo de Aplicaciones
Multiplataforma/Web*

BASES DE DATOS

UT5 – PROGRAMACIÓN DE BBDD

- 1. INTRODUCCIÓN AL LENGUAJE DE PROGRAMACIÓN**
- 2. DECLARACIÓN DE VARIABLES**
- 3. ESTRUCTURAS DE CONTROL DE FLUJO**
- 4. PROCEDIMIENTOS Y FUNCIONES**
- 5. CONTROL DE ERRORES**
- 6. CURSORES**
- 7. DISPARADORES (TRIGGERS)**

INTRODUCCIÓN AL LENGUAJE DE PROGRAMACIÓN

1

- En MYSQL igual que ocurre en otros lenguajes de programación, se permite la declaración de **bloques compuestos**.
- Una declaración compuesta es un bloque que puede contener otros bloques, declaraciones para variables, manejadores de condiciones y cursores; y construcciones de control de flujo tales como bucles y pruebas condicionales.
- El principio y final de un bloque compuesto se define con la sintaxis: **BEGIN ... END**
- Los bloques compuestos, también se utilizan en el cuerpo de **definición de procedimientos y funciones almacenados y triggers**.
- Estos objetos se definen en términos de código SQL que se almacena en el servidor para su posterior invocación.

DECLARACIÓN DE VARIABLES

A large, stylized number 2 in a light green color, centered within a dark green rounded square. The number has a slight shadow effect.

- Para la declaración de variables se utiliza la palabra reservada **DECLARE**.
- DECLARE está permitido solo dentro de una declaración compuesta **BEGIN ... END** y debe estar al inicio, antes de cualquier otra declaración.
- Las declaraciones deben seguir un cierto orden:
 - Las declaraciones de **variables y condiciones** deben aparecer antes de las declaraciones de cursor o controlador.
 - Las declaraciones de cursores deben aparecer antes que las declaraciones de manejador.

```
DECLARE nombre_variable [, nombre_variable] tipo_variable [DEFAULT valor];
```

- Asignación de valor a una variable directamente:

```
SET nombre_variable = valor_variable;
```

- Asignación de valor a una o más variables como resultado de una consulta:

```
SELECT campo1, campo2, . . . INTO variable1, variable2, . . .  
  
FROM nombre_tabla WHERE . . .
```

ESTRUCTURAS DE CONTROL DE FLUJO



- Sentencia IF

```
IF condicion THEN
    sentencias;
[ELSEIF condicion2 THEN
    sentencias;] . . .
[ELSE
    sentencias;]
END IF
```

- **Sentencia CASE**

```
CASE variable
  WHEN valor1 THEN
    sentencias;
  [WHEN valor2 THEN
    sentencias;] . . .
  [ELSE
    sentencias;]
END CASE
```

```
CASE
  WHEN condicion THEN
    sentencias;
  [WHEN condicion2 THEN
    sentencias;] . . .
  [ELSE
    sentencias;]
END CASE
```

- **Sentencia LOOP:** Los bucles LOOP no incorporan condición de salida, sino que debe ser implementada utilizando la instrucción LEAVE.

```
[etiqueta_inicio:] LOOP  
    sentencias;  
END LOOP [etiqueta_fin]
```

- **Sentencia LEAVE:** Se utiliza para romper la ejecución de cualquier instrucción de control de flujo que se haya etiquetado, normalmente bucles LOOP.

```
LEAVE etiqueta;
```

- **Sentencia REPEAT:**

```
[etiqueta_inicio:] REPEAT  
  
    sentencias;  
  
UNTIL condicion  
  
END REPEAT [etiqueta_fin]
```

- **Sentencia WHILE:**

```
[etiqueta_inicio:] WHILE condicion DO  
  
    sentencias;  
  
END WHILE [etiqueta_fin]
```

PROCEDIMIENTOS Y FUNCIONES



- Los procedimientos y funciones almacenadas son conjuntos de comandos SQL que pueden ser almacenados en el servidor.
- Se asocia un nombre a un conjunto determinado de instrucciones para, posteriormente, ejecutarlo tantas veces como se desee sin necesidad de volver a escribirlas.

```
CREATE PROCEDURE nombre_procedimiento ([[IN | OUT | INOUT] param1 tipo,  
                                         [IN | OUT | INOUT] param2 tipo, . . .])  
BEGIN  
    sentencias;  
END  
  
CREATE FUNCTION nombre_funcion([param1 tipo, param2 tipo, . . .])  
RETURNS tipo  
BEGIN  
    sentencias;  
END
```

- Los parámetros en un procedimiento pueden ser de entrada (**IN**), salida (**OUT**) o de entrada y salida (**INOUT**), por defecto si no se define otra cosa son siempre de entrada.
- En las funciones, los parámetros son siempre de entrada.

```
-- Muestra toda la información sobre los usuarios  
CREATE PROCEDURE lista_usuarios()  
BEGIN  
    SELECT * FROM usuarios;  
END
```

- Cuando el procedimiento almacenado va a realizar varias operaciones y estas están relacionadas, es muy conveniente tratarlas como una **transacción**:

```
-- Procedimiento para dar de alta una nueva pista en un polideportivo
-- determinado. Se pasan como parámetros todos los datos necesarios
-- para dar de alta la nueva pista asumiendo que se trata de una
-- pista abierta al público

CREATE PROCEDURE nueva_pista (p_codigo VARCHAR(10), p_tipo VARCHAR(255),
                             p_precio DECIMAL(10,3), p_id_polideportivo INT)
BEGIN
    DECLARE existe_polideportivo INT;

    SET existe_polideportivo = (SELECT COUNT(*) FROM polideportivo
                                WHERE id_polideportivo = p_id_polideportivo);
    IF existe_polideportivo != 0 THEN
        START TRANSACTION;

        INSERT INTO pista (codigo, tipo, precio, id_polideportivo)
            VALUES (p_codigo, p_tipo, p_precio, p_id_polideportivo);

        INSERT INTO pistas_abiertas (id_pista, operativa)
            VALUES (LAST_INSERT_ID(), TRUE);

        COMMIT;
    END IF;
END;
```



```
-- Función que devuelva el número de reservas que ha realizado un usuario
-- determinado
CREATE FUNCTION get_numero_reservas(p_id_usuario INT)
RETURNS INT NOT DETERMINISTIC3
BEGIN
    DECLARE cantidad INT;
    DECLARE existe_usuario INT;

    SET existe_usuario = (SELECT COUNT(*) FROM usuario
                        WHERE id_usuario = p_id_usuario);

    IF existe_usuario = 0 THEN
        -- Si el usuario no existe se devuelve valor de error
        RETURN -1;
    END IF;

    -- Si todo va bien, se calcula la cantidad y se devuelve
    SET cantidad = (SELECT COUNT(*) FROM reserva R, usuario_reserva UR
                  WHERE R.id_reserva = UR.id_reserva
                      AND UR.id_usuario = p_id_usuario);

    RETURN cantidad;
END;
```

- A la hora de implementar nuevos procedimientos y funciones hay que tener en cuenta algunas cuestiones.
- Puesto que el delimitador ';' se utiliza para finalizar cualquier orden sobre el motor MySQL, éste debe ser modificado mientras se implementa cualquier procedimiento o función, puesto que cualquier instrucción SQL que forme parte del código, sería interpretada de forma independiente.
- Así, la forma habitual de escribir procedimientos o funciones es a través de la creación de scripts SQL utilizando la orden **DELIMITER** que permite modificar el delimitador de fin de orden en MySQL.

```
DELIMITER &&  
CREATE PROCEDURE ver_pistas()  
BEGIN  
    SELECT * FROM pista;  
END &&  
DELIMITER ;
```

- Una función es **DETERMINISTIC** (determinista), si siempre devuelve el mismo resultado para una misma entrada dada, y **NOT DETERMINISTIC** (no determinista) cuando no. Por defecto si no se indica es no determinista.

- Una vez creada una función, desde un script o directamente en la consola, la forma de invocarla es a través de la instrucción **SELECT** seguido del nombre de función.

```
SELECT get_numero_reservas(97);
```

- En el caso de procedimientos, la forma de invocarlo es a través de la instrucción **CALL** seguido del nombre del procedimiento.

```
CALL ver_pistas();
```

- Para **eliminar** procedimientos o funciones usamos la siguiente sintaxis:

```
DROP {PROCEDURE | FUNCTION} [IF EXISTS] sp_name
```

- Para **consultar** procedimientos o funciones:

```
SHOW CREATE {PROCEDURE | FUNCTION} sp_name
```

```
SHOW {PROCEDURE | FUNCTION} STATUS [LIKE 'pattern']
```

- Se debe tener en cuenta que dentro de funciones y procedimientos se pueden utilizar todo tipo de funciones, como por ejemplo las funciones matemáticas vistas en anteriores temas.

CONTROL DE ERRORES

5

- Como ocurre con muchos lenguajes de programación, MySQL también es capaz de **gestionar, mediante excepciones, los errores** que se puedan producir durante la ejecución de un fragmento de código en entornos transaccionales.
- En estos, si durante la ejecución de una transacción se produce algún fallo, es posible deshacer toda la operación para evitar inconsistencias en los datos.

```
DECLARE handler_action HANDLER
    FOR condition_value [, condition_value] ...
statement

handler_action: {
    CONTINUE: La ejecución continúa
    | EXIT: La ejecución finaliza
}

condition_value: {
    mysql_error_code
    | SQLSTATE [VALUE] sqlstate_value
    | condition_name
    | SQLWARNING
    | NOT FOUND
    | SQLEXCEPTION
}
```

- Si una de las condiciones se cumple, se ejecuta el statement (o declaración) definida.
- Veamos un ejemplo, basándonos en un ejemplo anterior, en el que, en el caso de producirse una SQLEXCEPTION, se producirá ROLLBACK deshaciéndose los cambios realizados en la transacción y tras esto, la ejecución termina.


```
CREATE PROCEDURE nueva_pista(p_codigo VARCHAR(10), p_tipo VARCHAR(255),  
    p_precio DECIMAL(10,3), p_id_polideportivo INT)  
BEGIN  
    DECLARE existe_polideportivo INT;  
    DECLARE EXIT HANDLER FOR SQLEXCEPTION  
    BEGIN  
        ROLLBACK;  
    END;  
  
    SET existe_polideportivo = (SELECT COUNT(*) FROM polideportivos  
                                WHERE id = p_id_polideportivo);  
    IF existe_polideportivo != 0 THEN  
        START TRANSACTION;  
  
        INSERT INTO pistas (codigo, tipo, precio, id_polideportivo)  
            VALUES (p_codigo, p_tipo, p_precio, p_id_polideportivo);  
        INSERT INTO pistas_abiertas (id_pista, operativa)  
            VALUES (LAST_INSERT_ID(), TRUE);  
  
        COMMIT;  
    END IF;  
END;
```

- Con el control de las excepciones hemos conseguido controlar cualquier posible fallo que puedan generar las instrucciones de la transacción y, en su lugar, lanzar una orden **ROLLBACK** que deshará todos los pasos intermedios de dicha transacción que ya se hubieran ejecutado.
- Hay que tener en cuenta que el control de excepciones hace que el fallo nunca se propague hacia quién hubiera ejecutado este código, por lo que el programa que invoque a este código no se quedará en ningún estado inconsistente.

CURSORES



- Un **cursor**, es un objeto que hace referencia a un conjunto de datos obtenidos de una consulta.
- A través del cursor se pueden recorrer los datos obtenidos a través de la consulta, uno por línea.
- **DECLARE** sirve para definir un cursor.
- Los cursores deben abrirse (**OPEN**) antes de leerse los datos de ellos y se deben cerrar (**CLOSE**) una vez se ha terminado.
- **FETCH** extrae la siguiente fila de un cursor.
- **CLOSE** se cierra el cursor.

```
DECLARE cursor_name CURSOR FOR SELECT_statement;  
  
OPEN cursor_name  
  
FETCH cursor_name INTO variable list;  
  
CLOSE cursor_name;
```

- Ejemplo:

```
CREATE PROCEDURE cursor_demo()  
BEGIN  
  DECLARE p_titulo VARCHAR(200);  
  DECLARE done INT DEFAULT 0;  
  DECLARE cur_titulos CURSOR FOR SELECT titulo FROM noticias;  
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done=1;  
  
  OPEN cur_titulos;  
  REPEAT  
    FETCH cur_titulos INTO p_titulo;  
    IF NOT done  
    THEN  
      INSERT INTO titulos (TITULO) VALUES (p_titulo);  
    END IF;  
  UNTIL done  
  END REPEAT;  
  
  CLOSE cur_titulos;  
END;
```

DISPARADORES (TRIGGERS)



- Los **disparadores o triggers** son procedimientos de la Base de Datos que se ejecutan o activan cada vez que ocurre un evento determinado sobre una tabla determinada, según se haya indicado en el momento de su implementación.
- Los eventos que se pueden asociar a la ejecución de un TRIGGER son: **INSERT, UPDATE y DELETE.**
- También puede decidirse que se activen antes o después del evento en cuestión, utilizando las palabras reservadas **BEFORE** (antes) y **AFTER** (después) para indicar que el disparador se ejecute antes o después que la sentencia que lo activa.

- Las palabras **NEW** y **OLD** se emplean para referirse a las filas afectadas por el disparador, es decir, **a las filas de la tabla sobre la que se activa**, para referirse al estado de esa fila, **antes (OLD)** o **después (NEW)** de haber actuado el disparador.
- En un disparador **INSERT** sólo se podrá utilizar la palabra **NEW** puesto que no hay versión anterior de esa fila, en un disparador **UPDATE** se podrá acceder a cada fila **antes (OLD) y después (NEW)** de haber sido actualizada; y en un disparador **DELETE** sólo se puede utilizar la palabra **OLD** puesto que no hay nueva versión de la fila.
- Hay que tener en cuenta que toda columna **precedida por OLD es de sólo lectura**, por lo que se podrá hacer referencia a ella, pero sólo para leerla. En el caso de las columnas precedidas por **NEW**, se **podrá leer y también modificar su valor** con la instrucción SET.

```
-- Calcula automáticamente la edad de los usuarios en el mismo momento en
-- el que se dan de alta, a partir de la fecha de nacimiento que
-- introduzca el usuario
CREATE TRIGGER nuevo_usuario BEFORE INSERT ON usuarios
FOR EACH ROW
BEGIN
    IF NEW.fecha_nacimiento IS NOT NULL THEN
        SET NEW.edad = YEAR(CURRENT_DATE()) - YEAR(NEW.fecha_nacimiento);
    END IF;
END;
```

- Por ejemplo, el disparador **nuevo_usuario**, se activará una vez por cada fila (**FOR EACH ROW**) insertada (**INSERT**) en la tabla usuarios. Y concretamente antes de que éstas se inserten (**BEFORE**).

7. Disparadores (Triggers)

```
-- Actualiza la fecha de última reserva de una pista
-- cada vez que ésta se reserva
CREATE TRIGGER anota_ultima_reserva AFTER INSERT ON reservas
FOR EACH ROW
BEGIN
    UPDATE pistas_abiertas
        SET fecha_ultima_reserva = CURRENT_TIMESTAMP()
        WHERE id_pista = NEW.id_pista;
END;
```

```
-- Registra una pista como pista clausurada al público cuando
-- ésta se elimina de la Base de Datos
CREATE TRIGGER retira_pista AFTER DELETE ON pistas_abiertas
FOR EACH ROW
BEGIN
    INSERT INTO pistas_cerradas (id_pista, fecha_clausura, motivo)
        VALUES (OLD.id_pista, CURRENT_TIMESTAMP(), 'Eliminada');
END;
```

- Para eliminar disparadores:

```
DROP TRIGGER [IF EXISTS] [schema_name.]trigger_name
```

- Para obtener información:

```
SHOW TRIGGERS [[FROM | IN] db_name] [LIKE 'pattern' | WHERE expr]
```

- **Usos y limitaciones de los disparadores**
- Antes de comenzar a utilizar los disparadores, conviene conocer cuándo deben ser utilizados, y cuáles son sus limitaciones.
- Uno de los usos más comunes de los disparadores es el utilizarlos para realizar validaciones y para mantener actualizados los campos calculados, de manera que cuando ocurra algún cambio en los datos se pueda actualizar automáticamente dicho campo calculado, si tuviera que verse afectado.
- Además, permiten realizar tareas de auditoría, puesto que es posible registrar la actividad que ocurre en una o varias tablas en otra tabla, con el fin de registrar las operaciones que se realizan sobre ella, cuando se hacen, quién las hace, etc.

- **Usos y limitaciones de los disparadores**
- En cuanto a las limitaciones, no puede haber dos disparadores en una misma tabla que correspondan al mismo momento y sentencia.
- Por ejemplo, no se pueden tener dos disparadores BEFORE UPDATE. Pero sí es posible tener los disparadores BEFORE UPDATE y BEFORE INSERT o BEFORE UPDATE y AFTER UPDATE.
- También cabe destacar que desde un disparador no es posible invocar a una consulta. Se puede llamar a procedimientos almacenados que devuelvan información a través de sus parámetros de salida siempre y cuando éste no realice dentro ninguna consulta.