

# **TEMA 5:**

## **PROCEDIMENTS EMMAGATZEMATS**

### **EN MySQL**

1. Introducció
2. Avantatges i inconvenients
3. Tipus: procediments i funcions
4. Paràmetres i variables
5. Constructors de control de flux
6. Handlers. Gestió d'errors
7. Cursors
8. Triggers. Disparadors
9. Permisos

## 1. Introducció

Un procediment emmagatzemat (PE) és una rutina (procediment o funció) formada per un conjunt d'ordres SQL que es guarda en el servidor, com un objecte més de la BD.

En MySQL funcionen a partir de la versió 5. Per a comprovar la versió:

```
show variables like '%version%';
```

Els PE es poden executar:

- De forma directa (per exemple: entorn textual de Mysql o des del Workbench)
- Des d'una aplicació que accedisca a la BD (per exemple: des d'un programa Java)
- Des d'un altre PE

Alguns IDEs (aplicacions d'ajuda al programador) lliures per a la gestió de PE són:

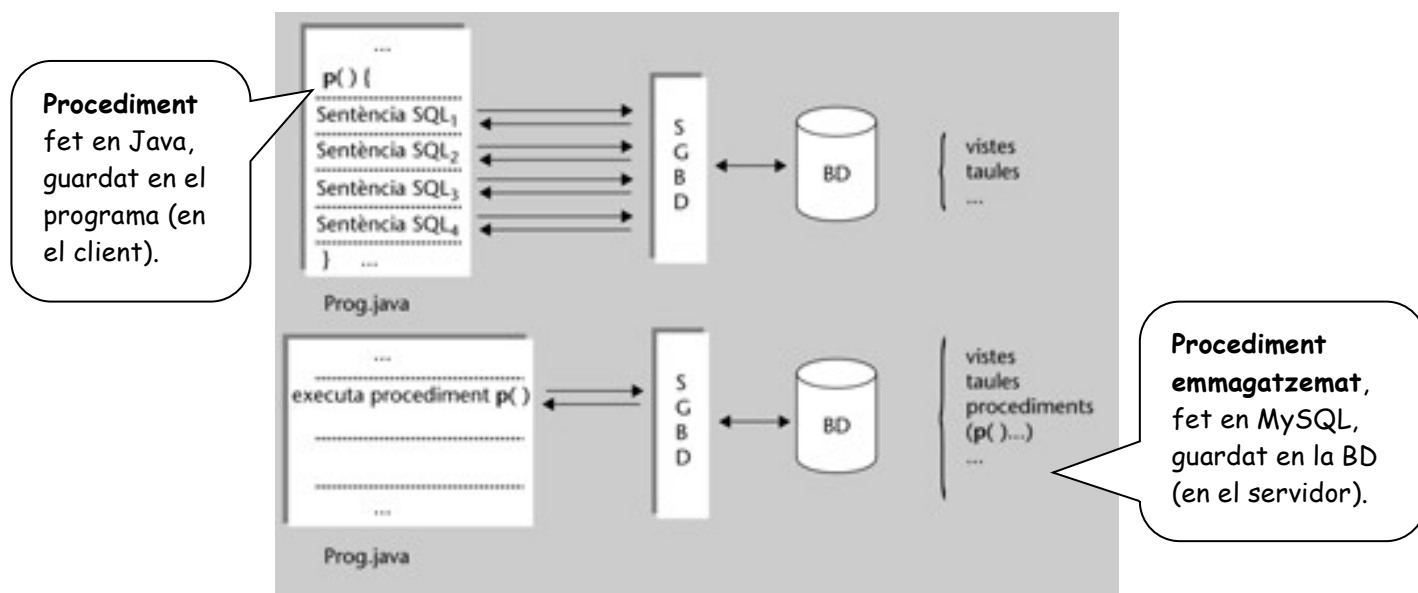
IDE	PROPIETARI
MySQL Workbench	Oracle
TOAD	Dell

En la taula mysql.proc hi ha informació dels procediments creats: nom, propietari, base de dades on està, etc.

No està permès l'ús de la ordre "USE nom\_base\_dades" dins d'un procediment per a canviar de BD però sí que es pot accedir a objectes d'altres BD amb el nom qualificat: nom\_bd.nom\_objecte.

## 2. Avantatges i inconvenients

A la figura que es mostra, apareixen dues arquitectures de desenvolupament d'aplicacions possibles, una fa ús dels PE, i l'altra no:



### 2.1. Avantatges

1. Millora el rendiment ja que no cal enviar tanta informació entre client i servidor. Es nota més eixa millora si client i servidor no estan en la mateixa màquina.
2. El PE pot ser utilitzat per diferents aplicacions, independentment del llenguatge i la plataforma.
3. Facilita la feina al programador, ja que ell sap que existeix un PE que li proporciona un determinat servei, no necessita saber les sentències SQL que incorpora el procediment, ni els elements de l'esquema de la BD.
4. Seguretat. Podem llevar permisos als usuaris sobre la BD i només donar permisos sobre els procediments. Així controlem de quina forma volem que siguin accedides les dades.

## 2.2. Inconvenients

1. Existeix una dependència amb el SGBD utilitzat. És a dir, cada sistema gestor de BD té el seu propi subllenguatge de PE:

SGBD	Subllenguatge SQL per a PE
MySQL	SQL 2003
PostgreSQL	PLPGSQL
Oracle	PL/SQL + Java
DB2 (IBM)	PL/SQL + Java
Informix (IBM)	SPL

Per tant si fem un PE amb un subllenguatge d'un SGBD i més endavant canviem de sistema gestor, haurem de tornar a refer els procediments.

2. No hi ha eines de depuració de procediments emmagatzemats.

### 3. Tipus: procediments i funcions

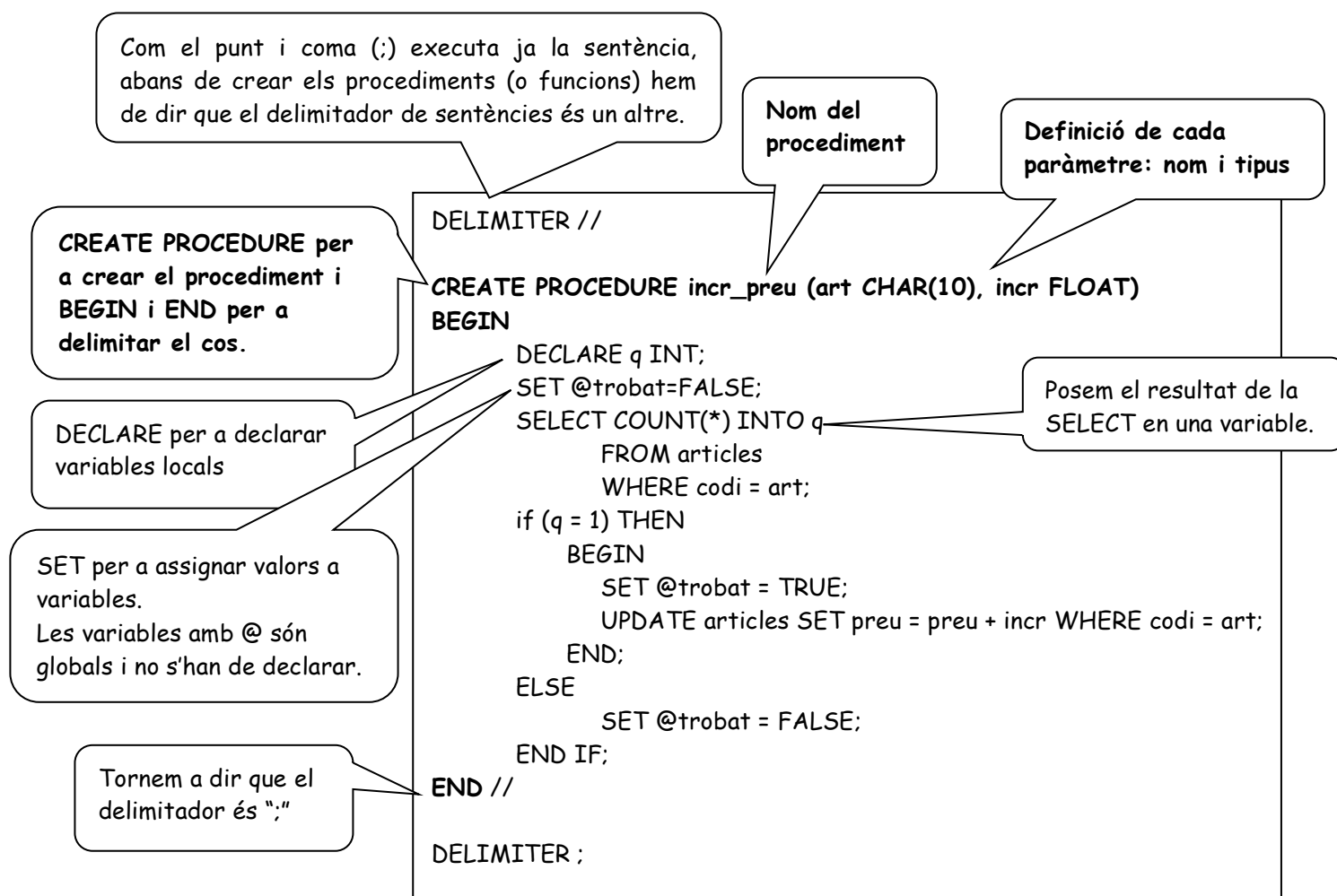
Hi ha dos tipus de rutines:

- Procediments (procedures)
- Funcions (functions)

Un procediment és un conjunt de tasques o accions que no retornen res a l'usuari, en canvi, les funcions retornen un valor. Igual que en qualsevol llenguatge de programació, la crida també serà diferent ja que les funcions han d'arreglar eixe valor.

#### 3.1. Procediments

Veiem les parts més comunes amb un exemple. Suposem que tenim una base de dades amb una taula d'articles i volem un procediment per a que incremente el preu d'un article.



Per a fer la crida a este procediment:

CALL és la sentència per a invocar l'execució del procediment.

```
CALL incr_preu ("pomes", 0.25);
```

```
SELECT @trobat;
```

Per si volem consultar el valor de la variable global modificada pel procediment.

És a dir: el client de la BD només ha de saber el nom del procediment i quins valors ha de passar com a paràmetre. No té per què saber ni el nom de la taula ni els noms dels camps.

Ara bé, com hem vist, si volem consultar si el procediment ha trobat o no l'article, també haurà de conèixer el nom de la variable global que modifica el procediment. Per això serà convenient que, en compte d'un procediment, usar una funció que retorne eixe valor.

Per a esborrar un procediment: `DROP PROCEDURE [IF EXISTS] nomProcediment`

### 3.2. Funcions

Veiem les parts principals amb un altre exemple. En una BD de futbol, on tenim en la taula de partits els gols que ha marcat cada equip en cada partit, volem una funció que ens retorne quants partits ha guanyat cada equip.

CREATE FUNCTION per a crear la funció.

El BEGIN-END no caldria si el cos de la funció només tinguera 1 instrucció: el RETURN

Amb RETURN retornarem al client el valor calculat.

```
DELIMITER //
```

```
CREATE FUNCTION guanyats(equip VARCHAR(3)) RETURNS INT
BEGIN
```

```
    DECLARE q INT;
```

```
    SELECT COUNT(*) INTO q
    FROM partits
    WHERE (equipc = equip AND gols< golsc)
    OR (equipf = equip AND golsf> golsc);
```

```
    RETURN q;
```

```
END;
//
```

```
DELIMITER ;
```

El RETURNS és per a indicar el tipus del valor que retorna la funció.

Per a fer la crida a una funció no es fa amb el CALL. En compte d'això, com la funció retorna un valor, la crida ha d'estar dins d'una expressió que arreplegue eixe valor.

Per tant, la crida es pot donar en diferents contextos (sempre formant part d'una expressió). Exemples de crides:

a) Mostrar els partits guanyats pel Barça:

```
SELECT guanyats("BAR");
```

b) Guardar en una variable el valor retornat per a després usar-la com siga:

```
SET @x = guanyats("BAR");  
SELECT CONCAT("El Barça ha guanyat ", @x, " partits");
```

c) Mostrar (entre altres coses) els partits guanyats per l'equip de cada jugador i/o posar-los en una condició :

```
select *, guanyats(jugadors.equip)  
from jugadors  
where guanyats(jugadors.equip) > 10;
```

e) Incrementar 1000 euros a cada jugador per cada partit que ha guanyat el seu equip:

```
update jugadors  
set sou = sou + 1000 * guanyats(jugadors.equip);
```

Per a esborrar una funció: DROP FUNCTION [IF EXISTS] nomFunció

## 4. Paràmetres i variables

### 4.1. Declaració de paràmetres

Si no s'indica res, cada paràmetre és d'entrada (el valor es proporciona en el moment de la crida). Però també podem passar al PE variables com a paràmetre per a que el PE retorne algun valor en estes variables.

Per a indicar si és un paràmetre és d'entrada i/o d'eixida posarem davant de cadascun:

- IN: Paràmetre d'entrada. En el moment de cridar la funció o el procediment, s'ha de donar este valor. És l'opció per defecte.
- OUT: Paràmetre de sortida. El valor es retorna en este paràmetre després de l'execució del procediment (per tant, no podem passar una constant a la rutina en eixe paràmetre).
- INOUT: Paràmetre d'entrada i de sortida.

Nota: el nom de les variables en MySQL no són *case sensitive*. És a dir: es considera la mateixa variable: preu, Preu, PREU...

#### Exercici resolt:

Fes un procediment que retorne en un paràmetre la quantitat d'usuaris que hi ha al nostre servidor MySQL. Nota: eixa informació està a la taula "user" de la bd "mysql".

```
CREATE PROCEDURE quantUsuaris (OUT param1 INT)
BEGIN
    select count(*) into param1 from mysql.user;
END
//
CALL quantUsuaris (@a)//
SELECT @a//
```



## 4.2. Declaració de variables locals

L'àmbit de visibilitat estarà només entre el BEGIN i END on està declarada la variable.

Sintaxi:

```
DECLARE nom_var[, ...] tipus [DEFAULT valor];
```

Sempre es declaren just a continuació del BEGIN d'un PE.

En cada sentència DECLARE es poden declarar diverses variables però sempre del mateix tipus i mateix valor inicial (DEFAULT). Si necessitàrem diferents tipus de variable o diferents valors inicials, haurem de posar diferents sentències DECLARE.

Exemples:

```
DECLARE edat INT; -- Si no posem valor per defecte serà nul.  
DECLARE preu, import INT DEFAULT 0;  
SET edat = 18; -- Assignem un valor a la variable;
```

## 4.3. Variables globals (variables de sessió)

L'àmbit de visibilitat és en qualsevol lloc de la connexió. El valor de les variables globals es guarda mentre està activa la connexió a la BD.

És l'única forma d'usar variables des de fora d'un PE.

No s'aconsella el seu ús dins dels PE.

No es declaren, simplement s'indica el nom de la variable que volem usar amb el signe @ davant: @nom\_var

Exemples:

```
SET @x = 10;  
CALL quantUsuaris(@quantitat);
```

#### 4.4. Assignar valor a les variables

- Amb SET (ja ho hem vist abans):

```
SET nom = "Pep";  
SET naix = 1970;  
SET cog = "Garcia", edat = naix + 45;  
SET @comptador = 0;  
SET @comptador = @comptador + 1
```

- Amb SELECT... INTO:

```
SELECT nom_alu, cog_alu INTO nom, cog LIMIT 1  
FROM alumnes  
WHERE codi = 5;
```

Si la Select retornara més d'una fila de resultats, donaria error.

## 5. Constructors de control de flux

### 5.1 Sentència IF

Sintaxi:

```
IF condició THEN sentència
[ ELSEIF condició THEN sentència ]
[ ELSE sentència ]
END IF;
```

Exemple:

```
IF a<b and a<c THEN SET min = a;
ELSEIF b<c THEN SET min = b;
ELSE SET min = c;
END IF
```

### 5.2. Sentència CASE

Sintaxi:

```
CASE case_value
    WHEN when_value THEN statement_list
    [WHEN when_value THEN statement_list] ...
    [ELSE statement_list]
END CASE
```

O bé:

```
CASE
    WHEN search_condition THEN statement_list
    [WHEN search_condition THEN statement_list] ...
    [ELSE statement_list]
END CASE
```

Exemples:

```
CASE mes
    WHEN 1 THEN set nom='gener';
    WHEN 2 THEN set nom='febrer';
    .....
    ELSE set nom='error';
END CASE
```

```
CASE
    WHEN nota>=0 and nota <5 THEN set text='insuficient';
    WHEN nota>=5 and nota <6 THEN set text='aprovat';
    .....
END CASE;
```

### 5.3. Sentència WHILE - DO

Sintaxi:

```
WHILE condició DO
    sentències
END WHILE;
```

Mentre es compleix la condició, s'executen les sentències.

Exemple:

```
CREATE PROCEDURE exemple_while_do()
BEGIN
    DECLARE n INT DEFAULT 5;
    WHILE n > 0 DO
        ...
        SET n = n -1;
    END WHILE;
END
```

### 5.4. Sentència REPEAT - UNTIL

Sintaxi:

```
REPEAT
    sentències
UNTIL condició END REPEAT;
```

Es repetixen les sentències fins que la condició es complisca. És com un do-while però la condició ha de ser la contrària per a que s'executen les sentències.

Exemple:

```
CREATE PROCEDURE exemple_repeat_until()
BEGIN
    DECLARE n INT DEFAULT 5;
    REPEAT
        ...
        SET n = n -1;
    UNTIL n<=0 END REPEAT;
END
```

Pregunta: fan el mateix els dos exemples anteriors de do-while i repeat-until?

Nota: l'operador condicional d'igualtat es representa per un signe igual: =

## 5.5. Sentència LOOP, LEAVE i ITERATE

Exemple:

```
CREATE PROCEDURE exemple_loop(n INT)
BEGIN
    label1: LOOP                -- Creació del bucle (loop) etiquetat com 'label1'
        SET n = n + 1;
        IF n < 10 THEN
            ITERATE label1;    -- Anem a l'inici del bucle (com el continue de Java)
        END IF;
        LEAVE label1;         -- Eixim del bucle (com el break de Java)
    END LOOP label1;
    SET @x = n;
END
```

Explicació:

Amb LOOP creem un bucle infinit: no posem cap condició ni al principi (com en el while-do) ni al final (com en el repeat-until), sinó dins del cos del bucle: amb el LEAVE eixirem del bucle i amb el ITERATE tornarem directament a l'inici del bucle. Estes dos sentències també es poden posar dins dels altres bucles que hem vist.

Estes sentències (LOOP, LEAVE i ITERATE) no són recomanables perquè trenquen la idea de la programació estructurada.

### **EXERCICIS DE PROCEDIMENTS I FUNCIONS (BD futbol)**

1. Crea el procediment *inserirGolejador* al qual li passes com a paràmetre el dorsal, el codi de l'equip, el nom del jugador, la quantitat de partits jugats i la quantitat de gols marcats. El procediment caldrà guardar eixa informació (en les taules de jugadors i golejadors) però caldrà tindre en compte primer si eixe jugador ja existia o no en la base de dades (ja que haurà de fer *inserts* o *updates* en cada cas).

A continuació fes 3 crides al procediment per comprovar que funciona correctament:

- a) Una crida amb un jugador que no existia
- b) Una crida amb un jugador que existia com a jugador però no com a golejador
- c) Una crida amb un jugador que existia com a jugador i golejador

2. Crea el procediment *inserirJornades* a la quan se li passa una quantitat de jornades i ha d'inserir tantes jornades com diga eixe número. La data no es posarà i el número de les jornades serà correlatiu a l'última jornada que hi ha guardada en la taula de jornades. A continuació, crida al procediment per a inserir 10 jornades. Comprova que s'han creat bé.

3. Funció *tipusJugador*, que li passes un jugador (dorsal i equip) i retorna una paraula depenent dels gols marcats. Si 0 gols, "cono"; entre 1 i 10 "normal"; entre 11 i 20 "bo"; entre 20 i 30 "crack"; i més de 30 "megacrack". Fes-ho amb l'estructura del CASE. A continuació (usant una crida a la funció):

- a) Mostra de cada jugador del Barça mostra el nom i el tipus de jugador que és.
- b) Mostra els megacracks de la lliga: nom del jugador i nom curt del seu equip.
- c) Mostra quants megacracks té cada equip.

4. Funció *quantsJugadors* que li passes com a paràmetre el codi d'un equip i retorna la quantitat de jugadors d'eixe equip. Després, mitjançant crides a eixa funció, fes els següents problemes:

- a) Mostra la quantitat de jugadors del Barça
- b) Mostra de cada equip de més de 25 jugadors: el codi, el nom curt i la quantitat de jugadors que té.
- c) Modifica la taula d'equips. Cal incrementar un 10% el pressupost dels equips que tenen més de 25 jugadors.
- d) Modifica la taula d'equips. Cal incrementar 1000 euros de pressupost per cada jugador que tinguen.
- e) Esborra els equips que no tinguen jugadors.

5. Després d'un partit volem posar els gols que ha fet un jugador.

- a) Fes el procediment *posarGolsAJugador* que li passes el jugador (dorsal i codi d'equip), la quantitat de gols que ha fet en el partit i quants d'estos han sigut de penal. El procediment ha de sumar eixes quantitats en la taula de golejadors (si no existia com a golejador caldrà inserir-lo). El procediment ha de calcular el total acumulat de gols que tindrà ara i els de penal i ho tornarà amb els mateixos paràmetres (és a dir: seran paràmetres d'entrada i eixida). També ha d'incrementar en 1 la quantitat de partits jugats. La quantitat total de partits jugats serà tornada en altre paràmetre (que serà d'eixida).
- b) Fes una crida al procediment per a dir que el dorsal 10 del Barça acaba de jugar un partit on ha marcat 3 gols (1 d'ells de penal).
- c) Després d'eixa crida, mostra per pantalla el valor d'eixes variables que ha retornat el procediment: total de gols d'eixe jugador, els de penal i la quantitat total de partits jugats.

## 6. Handlers. Gestió d'errors

Amb SQL podem gestionar els errors generats per les ordres SQL. Si per exemple fem un SELECT a una taula que no existeix, ens dona l'error 1146. Podem fer que si alguna vegada ocorre eixe error, que faça tal cosa. Per a aconseguir això cal definir un HANDLER.

Veiem primer un exemple i després explicarem la sintaxi.

Exemple: PE de prova per a saber si ha hagut error de clau duplicada

```
CREATE TABLE prova (codi int, primary key (codi));

DELIMITER //
CREATE PROCEDURE demo_handler()
BEGIN
    DECLARE CONTINUE HANDLER FOR
        SQLSTATE '23000'
        SET @err = 1;
    SET @x = 1;
    INSERT INTO prova VALUES (1); -- Ok
    SET @x = 2;
    INSERT INTO prova VALUES (1); -- Error (*)
    SET @x = 3;
END;
//

CALL demo_handler();//
SELECT @x//
```

Estem dient que si més endavant ocorre l'error 23000 (error per clau duplicada), que la variable @err tinga el valor 1 i que "CONTINUE" executant-se el tros de codi del bloc BEGIN-END.

Si executem això vorem que no ha aparegut cap error i, a més, el valor de @x és 3. Per què?

(\*) L'error provocat no apareix ja que ha estat "agafat" pel handler que hem creat. A més, com que eixe handler estava definit com a CONTINUE, la rutina ha continuat executant-se.

Si en compte de CONTINUE, el handler haguera estat definit com EXIT, en executar el 2n insert hauria acabat el procediment. En eixe cas @x valdria 2.



Vegem ara la sintaxi del HANDLER:

```
DECLARE tipus_handler HANDLER FOR  
  
codi_error [, ...]  
  
sentència;
```

On tenim:

- **sentència**: pot ser una ordre SQL simple o composta (bloc BEGIN - END)

- **tipus\_handler**: indica si continuen executant-se les següents instruccions a l'error o no:

CONTINUE    → continuar la rutina.

EXIT        → Se n'ix del bloc BEGIN ... END on està

- **codi\_error**: identificador de l'error (o llista d'errors) que es vol controlar.

Es pot posar de distintes formes:

```
codi_error_mysql  
| SQLSTATE codi_error_sqlstate  
| nom_error  
| SQLWARNING  
| NOT FOUND  
| SQLEXCEPTION
```

Vegem estes opcions:

a) `codi_error_mysql`

És un codi d'error numèric.

Exemple: **1022**      -- Clau duplicada

Problema: estos codis només són vàlids per a MySQL. Per tant, no és portable a altres SGBD.

b) `SQLSTATE codi_error_sqlstate`

És la paraula reservada `SQLSTATE` més un codi alfanumèric, vàlid per a SQL estàndard.

Exemple: **SQLSTATE '23000'**      -- Clau duplicada

Un codi `SQLSTATE` equival a molts codis d'error de MySQL. Exemples:

DESCRIPCIÓ ERROR	CODI MySQL	CODI SQLSTATE
Clau duplicada en taula	1022	'23000'
Columna no pot ser nula	1048	
Columna ambigua	1052	
Violació de clau aliena	1216	
...	...	
Taula desconeguda	1051	'42S02'
Taula inexistent	1146	
...	...	
Accés denegat a la BD	1044	'42000'
BD desconeguda	1049	
...	...	
...	...	...

Podem consultar els codis d'error de MySQL i `SQLSTATE` ací:

<https://dev.mysql.com/doc/refman/5.5/en/error-messages-server.html>

A més, `SQLSTATE 'HY000'` es pot usar com a codi d'error general.

c) `nom_error`

És un nom que li posem a l'error, el qual haurem declarat prèviament. A eixe d'error se li coneix com una "condició".

Exemple de condició: `taula_desconeguda`

La forma de declarar una condició és:

```
DECLARE taula_desconeguda CONDITION FOR 1051;
```

O bé:

```
DECLARE error_de_taula CONDITION FOR SQLSTATE '42S02';
```

d) `SQLWARNING`

És una abreviació per a tots els codis `SQLSTATE` que comencen per 01.

e) `NOT FOUND`

És una abreviació per a tots els `SQLSTATE` que comencen per 02.

Sol usar-se per a establir la condició de recórrer un cursor (ja ho vorem).

f) `SQLEXCEPTION`

És una abreviació per a tots els codis `SQLSTATE` que no són `SQLWARNING` ni `NOT FOUND`.

Nota: els handlers s'han de definir després de les variables i condicions.

## 7. Cursors

Els cursors s'utilitzen per a recórrer un a un els registres del resultat d'una SELECT per a fer alguna operació amb eixos valors. El mode d'operació és:

1. Declarar un cursor amb un nom i una SELECT associada:

```
DECLARE CURSOR c_alumnes FOR
  SELECT camp1, camp2, camp3
  FROM alumnes
  WHERE curs = "1DAM";
```

Cal respectar l'ordre de les declaracions:

- 1r: variables i condicions
- 2n: cursors
- 3r: handlers

La SELECT no pot contindre la instrucció INTO.

També declararem un HANDLER per a saber quan eixim del cursor.

```
DECLARE CONTINUE HANDLER FOR
  NOT FOUND
  SET @acabat = TRUE;
```

No caldria el HANDLER si sabem que la SELECT del cursor només va a obtenir una fila de resultats.

2. Recórrer els elements del cursor:

- a. Obrir el cursor:

```
OPEN c_alumnes;
```

Executa la SELECT del cursor (sense mostrar res) i deixa preparades les dades obtingudes en la consulta per a ser processades posteriorment.

- b. Bucle on, en cada iteració, obtindrem els valors dels camps de la SELECT en cada registre.

```
-- INICI BUCLE
```

```
FETCH c_alumnes INTO var1, var2, var3
```

```
-- Ací faríem operacions amb eixes variables
```

```
-- FI BUCLE
```

Si el cursor pot retornar més d'una fila, hem de recórrer-ho amb un While-do, repeat-until, etc on la condició d'acabar serà la que haurà activat el handler quan ja no queden files per llegir en el cursor.

Obté el següent registre disponible. Si no existeix donarà l'error de "sense dades", amb el valor SQLSTATE 02000 (zero files seleccionades o processades).

- c. Tancar el cursor:

```
CLOSE c_alumnes;
```

Serveix per a alliberar recursos.

Si no el tanquem nosaltres es tanca automàticament quan s'arriba al final del bloc BEGIN...END on està declarat. Però és convenient tancar-lo.

Els cursors de MySQL són només de lectura (no podem modificar els valors de la taula ni esborrar-los) i només es mouen cap endavant (cap al registre següent seguint l'ORDER BY de la SELECT del cursor).

Exercici resol't:

A partir de les dades de la següent taula:

PARTITS = equipc + equipf + jornada + golsc + golsf

Crea un procediment per a inserir en la taula *quinieles* els resultats dels partits de la jornada que es passa com a paràmetre.

QUINIELES = equipc + equipf + jornada + resultat

En el camp "resultat" cal posar un 1, una X o un 2, depenent del resultat del partit.

```
CREATE PROCEDURE fer_quiniela(j INT)
BEGIN
```

**-- 1r) DECLARACIÓ DE VARIABLES -----**

```
DECLARE ec, ef CHAR(3);
DECLARE gf, gc INT DEFAULT 0;
DECLARE resul CHAR(1);
DECLARE fi_cursor INT DEFAULT 0;
```

**-- 2n) DECLARACIÓ DEL CURSOR -----**

```
DECLARE c_partits CURSOR FOR
    SELECT equipc, equipf, golsc, golsf
    FROM partits
    WHERE jornada = j;
```

**-- 3r) DECLARACIÓ DEL HANDLER PER A EIXIR DEL CURSOR**

```
DECLARE CONTINUE HANDLER FOR
    SQLSTATE '02000' -- Codi error de 0 dades seleccionades
    SET fi_cursor = 1;
```

```
-- 4+) RECORREGUT DEL CURSOR -----  
OPEN c_partits;  -- Obrim cursor  
REPEAT  
    FETCH c_partits INTO ec, ef, gc, gf;  -- Recuperem dades  
  
    IF NOT fi_cursor THEN  -- Comprovem que no s'han acabat les files  
        CASE  
            WHEN gc > gf THEN SET resul = '1';  
            WHEN gc = gf THEN SET resul = 'x';  
            ELSE                SET resul = '2';  
        END CASE;  
  
        INSERT INTO quinieles (equipc, equipf, jornada, resultat)  
            VALUES (ec, ef, j, resul);  
    END IF;  
UNTIL fi_cursor END REPEAT;  
CLOSE c_partits;  -- Tanquem cursor  
  
END //
```

## 8. Disparadors (Triggers)

### 8.1. Introducció

Un disparador és un objecte amb nom dins d'una base de dades associat a una taula (no temporal ni vista) i fa que s'executen unes sentències **abans o després** de que passe en la taula un esdeveniment en concret: **inserció, esborrat o modificat**.

Per exemple, donada la següent taula:

```
CREATE TABLE comptes (num INT, saldo DECIMAL (10,2));
```

Volem fer que cada vegada que s'inserisca un nou compte, que s'acumule en una variable el saldo dels comptes. Per a fer això crearem el següent disparador:

```
CREATE TRIGGER abans_ins_compte  
BEFORE INSERT ON comptes  
FOR EACH ROW SET @suma = @suma + NEW.saldo;
```

Hem definit el disparador de nom `abans_ins_compte` així: "**Abans d'inserir** en la taula **comptes**, per cada fila a inserir, **acumula el nou saldo** en la variable `@suma`".

Com que `@sum` fa d'acumulador, abans d'usar-lo caldrà inicialitzar-lo a 0:

```
SET @suma = 0;
```

```
INSERT INTO comptes VALUES (1, 100), (2, 150), (3, 200);
```

```
INSERT INTO comptes VALUES (4, 250);
```

```
SELECT @suma;      // Mostrarà la suma total: 700
```

## 8.2. Sintaxi de CREATE TRIGGER

```
CREATE TRIGGER [nom_bd.]nom_disparador  
{BEFORE | AFTER} {INSERT | UPDATE | DELETE} ON nom_taula  
FOR EACH ROW sentència
```

I per a esborrar-lo: `DROP TRIGGER [nom_bd.] nom_disparador`

La sentència pot ser composta (bloc BEGIN-END). En eixe cas:

- ✓ caldrà indicar que el delimitador de sentències no siga el ";", amb DELIMITER, igual com fèiem en els cursors.
- ✓ es podrà fer ús de sentències condicionals i bucles

En les sentències es pot fer referència als valors que hi ha a la taula en el registre afectat abans o després de l'operació:

- Si es fa un UPDATE:

**OLD.nom\_camp** és el valor del camp abans de ser modificat

**NEW.nom\_camp** és el valor del camp després de ser modificat

- Si es fa un INSERT:

(OLD.nom\_camp no té sentit)

**NEW.nom\_camp** és el valor que s'insereix en eixe camp

- Si es fa un DELETE:

OLD.nom\_camp és el valor del camp del registre que s'esborra

**NEW.nom\_camp** no té sentit)



## 8.3. Utilització de disparadors

### 8.3.1. Limitacions

- Limitacions en els noms:
  - Una taula no pot tindre dos disparadors amb el mateix nom. I és convenient que en una mateixa BD no es repetisquen els noms dels disparadors.
- Limitacions en els tipus:
  - En una taula no poden haver dos disparadors del mateix tipus. Per exemple, no pot tindre dos disparadors BEFORE UPDATE.
- Limitacions en la sentència:
  - No es pot usar el CALL (invocar procediments emmagatzemats), però sí invocar a funcions.
  - No es poden iniciar o acabar transaccions (START TRANSACTION, COMMIT, o ROLLBACK).
  - En BEFORE INSERT, el valor NEW per a una columna AUTO\_INCREMENT és 0
  - Permisos:
    - Per a fer SET NEW.nom\_camp = valor cal tindre permís d'UPDATE sobre el camp (i fer-ho en triggers de tipus BEFORE)
    - Per a fer SET nom\_var = NEW.nom\_camp cal tindre permís de SELECT sobre el camp

### 8.3.2. Alguns usos

- Actualitzar camps de taules que depenen del valor d'altres camps: import en una línia de factura, el total d'una factura...
- Modificar els valors a introduir en la taula (amb BEFORE INSERT o BEFORE UPDATE):

Exemple: controlar que una nota estiga entre 0 i 10

```
delimiter //  
CREATE TRIGGER nota_ok BEFORE UPDATE ON notes  
FOR EACH ROW  
BEGIN  
    IF NEW.nota < 0 THEN  
        SET NEW.nota = 0;  
    ELSEIF NEW.nota > 10 THEN  
        SET NEW.nota = 10;  
    END IF;  
END; //  
delimiter;
```

Nota: Caldria repetir el mateix per a BEFORE INSERT. I estaria bé poder posar les sentències associades en un procediment emmagatzemat, però MySQL no permet fer ús del CALL en els disparadors.

#### Gestió d'errors:

L'ordre d'execució de les sentències és:

1r) Les del BEFORE

2n) Les pròpies accions INSERT, UPDATE, DELETE

3r) Les de l'AFTER

Per tant, quan falle alguna instrucció, no s'executaran les següents.

## 9. Permisos

- Per a donar permís de crear PE:

```
GRANT CREATE ROUTINE
```

```
ON nom_bd.*          -- o bé, sense el nom_bd.
```

```
TO nom_usuari
```

A partir d'eixe moment, l'usuari `nom_usuari` ja podrà crear PE i tindrà tots els permisos sobre eixos PE (execució, modificació i esborrat).

- Per a donar permís per a modificar o esborrar un PE:

```
GRANT ALTER ROUTINE
```

```
ON { PROCEDURE | FUNCTION } nom_proc
```

```
TO nom_usuari;
```

- Per a donar permís d'execució d'un PE:

```
GRANT EXECUTE
```

```
ON { PROCEDURE | FUNCTION } nom_proc
```

```
TO nom_usuari;
```

- Per a llevar permisos estan els revoke corresponents:

```
REVOKE { CREATE ROUTINE | ALTER ROUTINE | EXECUTE }
```

```
ON nom_proc
```

```
FROM nom_usuari
```

Exemples:

```
REVOKE CREATE ROUTINE ON * FROM pep;
REVOKE ALTER ROUTINE ON PROCEDURE llistar FROM pep;
REVOKE ALTER ROUTINE ON FUNCTION quants FROM pep;
REVOKE EXECUTE ON llistar FROM pep;
```

- Taules del sistema associades:

En la taula `mysql.proc` hi ha informació dels procediments creats (nom, propietari, etc).

En la taula `mysql.procs_priv` hi ha informació dels permisos dels processos de cada usuari.