8. COD MEMORAT IN BAZA DE DATE

8.1. Concepte....2
8.2. Functii si proceduri memorate in baza de date....2
8.2.1. Concepte. Comparatie intre functii si proceduri....2
8.2.2. Definirea unei proceduri....3
8.2.2.1. Sintaxa generala....3
8.2.2.2. Problema delimitatorului....3
8.2.2.3. Parametrii unei proceduri....3
8.2.2.4. Corpul unei proceduri. Blocuri de instructiuni....4
8.2.2.5. Caracteristici suplimentare ale unei rutine....5
8.2.3. Apelarea unei proceduri....5
8.2.3.1. Modalitate de apelare si particularitati....5
8.2.3.2. Date de intrare si de iesire....6
8.2.4. Functii....7
8.2.5. Vizualizarea rutinelor memorate in baza de date si a parametrilor acestora....7
8.2.6. Variabile locale....8
8.2.7. Structuri de control al executiei....9
8.2.7.1. Tipuri de instructiuni suportate de MySQL....9
8.2.7.2. Instructiuni decizionale....10
8.2.7.3. Incheierea fortata a executiei unui bloc: instructiunea LEAVE....11
8.2.7.4. Instructiuni repetitive....12
8.2.8. Conditii si handlere: gestionarea erorilor din cadrul unei rutine....14
8.2.8.1. Concepte....14
8.2.8.2. Declararea conditiilor si a handlerelor....14
8.2.9. Cursoare: parcurgerea rand cu rand a inregistrarilor unui result set....15
8.2.9.1. Concepte si pasi de urmat....15
8.2.9.2. Definirea unui cursor....15
8.2.9.3. Utilizarea cursorului pentru accesarea datelor....16
8.3. Trigger-e....17
8.3.1. Concepte....17
8.3.2. Definirea unui trigger....17
8.3.3. Referirea la valorile coloanelor din cadrul unui trigger....18
8.3.4. Vizualizarea triggerelor....18
8.3.5. Stergerea triggerelor....19
8.4. BIBLIOGRAFIE....19
8.5. ANEXA....20

8.1. Concepte

Sistemele de baze de date moderne permit clientului sa includa in insasi baza de date portiuni de cod SQL refolosibil, pe care il poate mai apoi executa in mod repetat, fara a fi necesara retransmisia lui. Codul memorat in baza de date poate contine variabile, structuri de control etc., permitand implementarea unei logici complexe si deschizand astfel calea mutarii unei parti a logicii aplicatiilor de pe client pe server.

MySQL permite includerea de cod in baza de date sub urmatoarele forme:

- **functii si proceduri** – asemanatoare cu notiunile omonime din programare, acestea reprezinta seturi de instructiuni SQL definite de client si stocate pe server, accesibile prin intermediul unui nume si care pot primi date de intrare si furniza date de iesire
- **trigger-e** – sunt portiuni de cod asociate unei tabele si care se executa automat atunci cand asupra tabelei se aplica operatii de tip INSERT, UPDATE sau DELETE
- **event-uri** – reprezinta portiuni de cod executate in momente de tip prestabilite, conform unui program, ceea ce permite automatizarea anumitor operatii din baza de date (ex: rapoarte periodice)

A stoca cod pe serverul de baze poate oferi urmatoarele avantaje:

- mai putin trafic intre server si client. Clientul transmite serverului codul o singura data si il poate apoi executa in mod repetat fara a-l retransmite
- intermedierea accesului la date. Pentru siguranta maxima, toate operatiile de manipulare a datelor pot trece prin functii/proceduri intermediare care se asigura permanent de validitatea informatiei din baza de date, clientii nemaiavand acces direct la tabele (asemanator cu principiul incapsularii si al getterilor/setterilor din programarea obiectuala)
- validitatea datelor nu mai depinde de client, ci este impusa ca parte a bazei de date. In loc ca fiecare aplicatie client sa efectueze propria validare a datelor inainte de a le trimite serverului de baze de date, regulile pot fi memorate direct pe server si, in plus, apelate automat ori de cate ori se introduc/modifica date. Acesta este un lucru pozitiv mai ales in cazul unei structuri eterogene a clientilor – aplicatii scrise in diferite limbaje sau ruland pe diverse platforme hardware si software – deoarece in acest fel exista un singur set de reguli (logica de validare/prelucrare a datelor nu este multiplicata cu numarul de clienti) si clientii sunt degrevati de implementarea regulilor de validitate a datelor (cu toate particularitatile pe care acest lucru le presupune pe platforma pe care ei ruleaza)

Dezavantajul major al stocarii si executiei codului pe serverul SQL este cresterea incarcarii acestuia, ceea ce solicita un hardware mai performant mai ales in situatia accesului concurent din partea multor clienti. Pe de alta parte, acesta poate deveni un avantaj atunci cand clientii dispun de resurse reduse, deoarece complexitatea softului client scade.

8.2. Functii si proceduri memorate in baza de date

8.2.1. Concepte. Comparatie intre functii si proceduri

Atat functiile, cat si procedurile reprezinta seturi de instructiuni SQL definite de catre client, stocate pe serverul SQL si carora li se atribuie un nume. Clientul poate apoi executa respectivele portiuni de cod referindu-le cu simplul lor nume, asadar efectuand trafic minim cu serverul SQL.

Atat functiile, cat si procedurile pot primi date de intrare si pot furniza date de iesire, dar fac acest lucru in mod diferit.

Iata principalele diferente intre functii si proceduri:

- o functie returneaza date si deci poate fi folosita in cadrul unei expresii. O procedura, chiar daca produce date, le furnizeaza printr-un alt mecanism si nu poate fi folosita ca operand intr-o expresie
- o functie returneaza o singura valoare; o procedura poate produce fie un set de rezultate (“result set”), fie mai multe valori de iesire
- o functie se apeleaza folosind simplul sau nume; o procedura se apeleaza folosind instructiunea CALL urmata de numele procedurii

Date fiind aspectele comune ale functiilor si procedurilor, in continuarea acestui material vom folosi termenul generic de “rutine” acolo unde dorim sa ne referim atat la functii, cat si la proceduri.

Vom incepe prin a prezenta modul de definire si utilizare a procedurilor, cu mentiunea ca majoritatea elementelor prezentate sunt aplicabile si functiilor.

8.2.2. Definirea unei proceduri

8.2.2.1. Sintaxa generala

Sintaxa generala de definire a unei proceduri este urmatoarea:

```
CREATE PROCEDURE nume (definitie_parametru_1, definitie_parametru_2,...)
[...caracteristici suplimentare...]
corpul_procedurii
```

Din definitie pot lipsi parametrii si caracteristicile suplimentare.

O procedura este asociata unei baze de date, ceea ce are urmatoarele implicatii:

- procedura dispare automat la stergerea bazei de date de care apartine
- numele său nu trebuie sa fie identic cu al altor proceduri din aceeasi baza de date, insa pot exista proceduri cu nume identic in alte baze de date
- folosind o procedura cu numele ei scurt, se considera ca ea face parte din baza de date curenta. Pentru a referi o procedura din alta baza de date se va folosi sintaxa cunoscuta de la tabele: *bazadedate.procedura*

O procedura poate avea nume identic cu cel al unei tabele sau chiar al unei functii aflate in aceeasi baza de date.

8.2.2.2. Problema delimitatorului

Atunci cand o procedura contine mai multe instructiuni SQL (si in general asa se si intampla), fiecare instructiune trebuie incheiata cu caracterul ; (punct si virgula). Acelasi caracter este din pacate folosit si de catre unele programe client (ex: mysql, MySQL Query Browser etc.) pe post de delimitator de final de instructiune. Din acest motiv, astfel de programe vor afisa o eroare la incercarea de definire a unei proceduri, deoarece vor considera primul caracter ; ca fiind finalul intregii definitii, cand el reprezinta de fapt doar finalul primei instructiuni componente a procedurii.

Pentru rezolvarea problemei exista comanda locala DELIMITER (inteleasa de catre *mysql*, *MySQL Query Browser*, *MySQL Workbench* si destule alte softuri client) care schimba caracterul perceput ca indicator de final de instructiune. Putem astfel sa modificam temporar delimitatorul, pe durata definirii unei rutine, si sa-l refacem dupa incheierea definitiei:

```
DELIMITER $$
CREATE PROCEDURE salut(IN nume VARCHAR(200))
BEGIN
      SELECT CONCAT('Salut ', nume, '!');
END
$$
DELIMITER ;
```

8.2.2.3. Parametrii unei proceduri

Parametrii unei proceduri permit pasarea de valori de intrare la apelarea sa si returnarea de valori calculate in cadrul procedurii. Definitia unui parametru se prezinta astfel:

```
[IN|OUT|INOUT] nume_parametru tip_date
```

Tipul de date poate fi oricare dintre cele suportate de MySQL. Numele parametrului trebuie sa fie unic in lista de parametri ai procedurii si nu este case sensitive.

Parametrii unei proceduri pot fi de 3 feluri:

- parametri de tip **IN** – reprezinta date de intrare pentru procedura; valorile lor sunt specificate la apelarea procedurii, fie sub forma de valori constante, fie sub forma de variabile. In ultimul caz, orice modificare adusa variabilei in interiorul procedurii nu este vizibila pentru apelant (nu modifica variabila originala)
- parametri de tip **OUT** – reprezinta variabile in care procedura depune informatie pentru ca ea sa fie disponibila apelantului. Un astfel de parametru are valoarea initiala NULL (chiar daca a fost specificata o valoare pentru el la apelare!), urmand a-i fi modificata in cadrul procedurii
- parametri de tip **INOUT** – reprezinta o combinatie intre primele doua tipuri de parametri: valoarea pasata la apelare este disponibila in interiorul procedurii, iar in cazul modificarii noua valoare va fi disponibila apelantului

Exemplu:

```
DELIMITER $
CREATE PROCEDURE test(IN i INT, OUT j INT, INOUT k INT)
BEGIN
      SELECT i,j,k;
      SET i=1,j=2,k=3;
END$
DELIMITER ;
SET @a=10,@b=20,@c=30;
CALL test(@a,@b,@c);
+------+------+------+
| i    | j    | k    |
+------+------+------+
|   10 | NULL |   30 |
+------+------+------+
SELECT @a,@b,@c;
+------+------+------+
| @a   | @b   | @c   |
+------+------+------+
| 10   | 2    | 3    |
+------+------+------+
```

8.2.2.4. Corpul unei proceduri. Blocuri de instructiuni

Corpul unei proceduri contine o singura instructiune SQL, care poate fi simpla sau compusa. O instructiune compusa consta din mai multe instructiuni simple incadrate intre cuvintele cheie BEGIN si END si terminate fiecare cu caracterul ; (punct si virgula):

```
-- procedura cu o singura instructiune simpla
CREATE PROCEDURE random() SELECT RAND()*100;

-- procedura cu instructiune compusa, formata din mai multe instructiuni simple
DELIMITER $
CREATE PROCEDURE stats()
BEGIN
      SELECT Judet,COUNT(*) AS Populatie GROUP BY Judet;
      SELECT Oras,COUNT(*) AS Populatie GROUP BY Oras;
END$
DELIMITER ;
```

In cadrul corpului unei proceduri putem folosi:

- parametri prezenti in definitia procedurii (cu simplul lor nume, fara @ in fata!)

- variabile declarate local folosind DECLARE (vezi subcapitolul dedicat)
- variabile definite de client in sesiunea curenta (cele precedate de @) si care sunt vizibile automat in cadrul procedurii
- expresii SQL ce le implica si pe cele enumerate mai sus
- structuri de control al executiei (instructiuni decizionale, repetitive – vezi subcapitolele corespunzatoare)

Blocul BEGIN...END poate avea o eticheta atasata, ceea ce ajuta la identificarea sa - fie in scopul controlului executiei (vezi subcapitolul dedicat), fie pentru simpla crestere a lizibilitatii codului. Aceasta ajuta in special pentru blocuri BEGIN...END imbricate:

```
unu: BEGIN
       SELECT 'Inauntrul lui unu';
       doi: BEGIN
              LEAVE unu;
       END doi;
END;   -- label-ul pentru delimitatorul de final este optional
```

Detalierea diferitelor elemente ce pot participa in corpul unei proceduri va fi efectuata in capitolele urmatoare.

8.2.2.5. Caracteristici suplimentare ale unei rutine

Prezentam urmatoarele caracteristici de interes ale unei rutine:

- **COMMENT '...descriere...'** – permite atasarea unui sir de caractere la definitia rutinei. In acest fel fiecare definitie poate contine si o scurta descriere a metodei (scop, rezultat produs, parametri folositi etc)
- **[NOT] DETERMINISTIC** – ajuta MySQL sa optimizeze interogarile in care participa procedura. O procedura deterministica este una care, apelata de mai multe ori cu acelasi set de parametri de intrare, va produce de fiecare data acelasi rezultat. Una non-deterministica nu are un output predictibil pe baza parametrilor de intrare (ex: o procedura care produce rezultate calculate cu RAND()). Valoarea implicita este NOT DETERMINISTIC. Alegerea gresita a tipului de procedura poate avea rezultate nefaste (mai ales in cazul unei proceduri non-deterministice declarata ca DETERMINISTIC) deoarece MySQL nu face verificari, ci foloseste ca atare informatia oferita de creatorul procedurii

Atentie! Atunci cand pe serverul MySQL este activat binary logging-ul, nu vor putea fi create functii NON-DETERMINISTIC! Eroarea afisata este:

*ERROR 1418 (HY000): This function has none of DETERMINISTIC, NO SQL, or READS SQL DATA in its declaration and binary logging is enabled (you *might* want to use the less safe log_bin_trust_function_creators variable)*

8.2.3. Apelarea unei proceduri

8.2.3.1. Modalitate de apelare si particularitati

Apelarea unei proceduri se realizeaza folosind instructiunea **CALL**, urmata de numele procedurii si de un set de paranteze rotunde intre care se afla eventualele valori corespunzatoare parametrilor din definitia procedurii:

```
USE proc
CALL salut('Mihai');        -- procedura aflata in baza de date curenta
CALL test.random();         -- procedura aflata in alta baza de date
```

ATENTIE! O procedura este atasata unei anumite baze de date. La apelarea procedurii se schimba temporar baza de date curenta cu cea de care apartine procedura, urmand a fi restaurata la incheierea executiei. Aceasta inseamna ca a referi, din cadrul procedurii, o tabela cu numele ei scurt va determina cautarea tabelei in baza de date din care face parte procedura!

```
USE test
CREATE PROCEDURE testproc() SELECT * FROM t;
USE work
SHOW TABLES LIKE 't';

+--------------------+
| Tables_in_test (t) |
+--------------------+
| t                  |
+--------------------+
-- tabela t exista in baza de date curenta, deci SELECT * FROM t, rulat interactiv, ar functiona
CALL test.testproc();
ERROR 1146 (42S02): Table 'test.t' doesn't exist
```

8.2.3.2. Date de intrare si de iesire

O procedura poate folosi in operatiile pe care le efectueaza date de intrare provenite din doua surse:

- valori ale parametrilor de tip IN sau INOUT, folosind simplul nume al parametrului in cauza
- valori ale variabilelor de sesiune, definite de catre client. O variabila de sesiune definita inaintea apelarii procedurii va fi automat vizibila in interiorul acesteia si, in plus, orice modificari aduse variabilei sunt vizibile in cadrul sesiunii. Daca o procedura defineste o variabila de sesiune, aceasta va ramane disponibila in cadrul sesiunii si dupa incheierea executiei procedurii

Exemplul 1:

```
CREATE PROCEDURE apel(IN p1 INT) SET @a = p1, @b = 20, @c = 30;
SET @a = 1, @b = 2;
CALL apel(99);
SELECT @a,@b,@c;
+------+------+------+
| @a   | @b   | @c   |
+------+------+------+
| 99   | 20   | 30   |
+------+------+------+
```

In cazul parametrilor de tip IN, atunci cand se paseaza ca parametru la apelare o variabila de sesiune, valoarea acesteia nu poate fi modificata in interiorul procedurii:

```
CREATE PROCEDURE modificare(IN p1 INT) SET p1 = 100;
SET @a = 5;
CALL modificare(@a);
SELECT @a;
+------+
| @a   |
+------+
| 5    |
+------+
-- @a ar fi fost modificat daca p1 era de tip INOUT!
```

O procedura poate produce rezultate de iesire in 3 feluri:

- poate produce unul sau mai multe result set-uri, prin simpla utilizare repetata a instructiunii SELECT:

```
DELIMITER $
CREATE PROCEDURE listari() BEGIN
    SELECT * FROM t1 LIMIT 1;
    SELECT * FROM t2 LIMIT 2;
END$
CALL listari();
+------+
```

```
| i    |
+------+
|    1 |
+------+
1 row in set (0.00 sec)

+------+
| j    |
+------+
|    2 |
+------+
1 row in set (0.00 sec)
```

- poate memora valori in parametrii de tip OUT sau INOUT din definitia sa, ele fiind astfel disponibile apelantului (vezi exemplul 1)
- poate modifica variabile de sesiune (vezi exemplul 1)

Nota: *o procedura nu poate fi folosita in cadrul unei expresii, asa cum se intampla cu functiile:*

```
SELECT 1+random(); -- NU FUNCTIONEAZA!
```

8.2.4. Functii

Majoritatea elementelor discutate pana acum in cazul procedurilor se aplica in acelasi mod si pentru functii, cu urmatoarele exceptii/diferente:

- o functie produce o singura valoare de iesire si NU poate returna un result set
- definitia unei functii include o clauza RETURNS care specifica tipul de date al valorii returnate
- toti parametrii unei functii sunt automat de tip IN. Putem specifica explicit IN ca parte a definitiei parametrului (desi este inutil), dar nu si OUT sau INOUT
- o functie se apeleaza folosind simplul sau nume si eventualul set de parametri (fara CALL)
- o functie poate fi folosita in cadrul unei expresii

```
DELIMITER $
CREATE FUNCTION AflaProduse(Prag INT) RETURNS INT DETERMINISTIC
COMMENT 'Functia returneaza numarul de produse care au pretul sub pragul specificat'
BEGIN
      DECLARE nr INT;
      SELECT COUNT(*) FROM Produse WHERE Pret<Prag INTO nr;
      RETURN nr;
END$
DELIMITER ;
SET @p = 100;
SELECT CONCAT('Avem ', AflaProduse(@p), ' produse mai ieftine de ', @p, ' RON') AS Concluzie;

+--------------------------------------+
| Concluzie                            |
+--------------------------------------+
| Avem 10 produse mai ieftine de 100 RON |
+--------------------------------------+
```

Pentru a returna o valoare dintr-o functie se foloseste instructiunea RETURN, urmata de o expresie al carei tip de date trebuie sa corespunda cu tipul de date “promis” in clauza RETURNS. Expresia poate consta dintr-o simpla valoare (“literal”), o variabila locala sau de sesiune sau poate fi una complexa care implica operatori, functii predefinite etc.

8.2.5. Vizualizarea rutinelor memorate in baza de date si a parametrilor acestora

Avem la dispozitie urmatoarele instructiuni SQL pentru vizualizarea listei de functii/proceduri:

- SHOW PROCEDURE STATUS pentru vizualizarea listei de proceduri
- SHOW FUNCTION STATUS pentru vizualizarea listei de functii

```
SHOW PROCEDURE STATUS LIKE 'apel';
*************************** 1. row ***************************
           Db: test
         Name: apel
         Type: PROCEDURE
      Definer: root@localhost
     Modified: 2010-11-25 15:45:04
      Created: 2010-11-25 15:45:04
Security_type: DEFINER
      Comment:

SHOW FUNCTION STATUS LIKE 'AflaProduse';
*************************** 1. row ***************************
           Db: test
         Name: AflaProduse
         Type: FUNCTION
      Definer: root@localhost
     Modified: 2010-11-25 16:13:45
      Created: 2010-11-25 16:13:45
Security_type: DEFINER
      Comment: Functia returneaza numarul de produse care au pretul sub prag
```

Instructiunile de mai sus afiseaza numai caracteristicile rutinei, nu si definitia acesteia. In acest ultim scop exista alte doua instructiuni:

- SHOW CREATE PROCEDURE afiseaza instructiunea SQL care creeaza procedura
- SHOW CREATE FUNCTION indeplineste acelasi rol in cazul functiilor

```
SHOW CREATE PROCEDURE APEL\G
*************************** 1. row ***************************
       Procedure: apel
        sql_mode:
Create Procedure: CREATE DEFINER=`root`@`localhost` PROCEDURE `apel`(IN p1 INT)
SET @a = p1, @b = 20, @c = 30
```

Nota: *o modalitate alternativa de a obtine informatii despre rutinele memorate pe server este interogarea bazei de date cu meta-informatie INFORMATION_SCHEMA (mai exact tabela ROUTINES)*

8.2.6. Variabile locale

In afara de parametrii din definitia rutinei si de variabilele de sesiune, o rutina mai poate folosi in corpul sau variabile interne. Aceste variabile se declara la inceputul rutinei si sunt vizibile numai pe parcursul executiei ei, disparand automat la incheierea executiei rutinei. Ele pot fi folosite pentru a memora temporar informatii produse in cadrul rutinei, pot participa in expresii, iar in cazul functiilor pot fi folosite ca parametru pentru instructiunea RETURN.

Variabilele locale sunt create cu ajutorul instructiunii DECLARE, specificand numele variabilei, tipul ei de date si – optional - valoarea din oficiu.

```
DECLARE var1, var2, .....       tip_date     [DEFAULT valoare_implicita];
```

Instructiunile DECLARE pot fi folosite doar inauntrul unui bloc BEGIN...END si trebuie plasate inaintea oricaror alte instructiuni prezente in blocul respectiv. In cazul blocurilor imbricate, daca exista o variabila cu acelasi nume intr-un bloc interior si intr-unul exterior, in cadrul celui interior este disponibila doar variabila declarata in cadrul acelui bloc, nu si cea exterioara.

Nota: *instructiunea DECLARE este folosita si pentru declararea de conditii, handlere sau cursoare. Atunci cand in cadrul unui bloc BEGIN...END se declara mai multe astfel de entitati, variabilele trebuie sa fie primele.*

La declararea mai multor variabile cu o singura instructiune DECLARE, toate vor avea acelasi tip de date si aceeasi valoare default. Pentru a crea variabile cu tip de date diferit sau cu valoare de pornire diferita este necesara utilizarea mai multor instructiuni DECLARE separate.

Atunci cand lipseste valoarea default din definitia variabilei, implicit variabila va avea valoarea de pornire NULL.

O variabila locala este accesata cu simplul sau nume (fara @), de aceea este bine, pentru a evita ambiguitatile, ca numele variabilelor locale sa nu fie identice cu nume de tabele sau coloane existente.

O variabila locala poate primi o valoare in urmatoarele moduri:

- folosind instructiunea SET, in acelasi mod ca in cazul variabilelor de sesiune (dar fara @)
- folosind constructia SELECT...INTO variabila, detaliata mai jos
- folosind constructia FETCH...INTO variabila daca folosim cursoare (vezi subcapitolul despre utilizarea cursoarelor)

Constructia SELECT...INTO poate fi folosita pentru a atribui o valoare unei variabile de orice natura (variabila locala, variabila de sesiune, parametru al unei rutine). Sintaxa sa este:

```
SELECT expr1,expr2,.... INTO var1, var2,...
```

Este necesara respectarea urmatoarelor restrictii:

- numarul de variabile trebuie sa fie egal cu numarul de coloane furnizate de catre interogarea SELECT
- interogarea SELECT trebuie sa produca exact un rand. Daca produce mai multe rezulta o eroare, daca nu produce niciun rand rezulta un warning si variabilele raman nemodificate:

```
SELECT Pret,Nume FROM Produse LIMIT 1 INTO @p,@n;
SELECT @p,@n;
+------+-------+
| @p   | @n    |
+------+-------+
| 10   | Caise |
+------+-------+

SELECT Pret,Nume FROM Produse LIMIT 1 INTO @p;
ERROR 1222 (21000): The used SELECT statements have a different number of columns

SELECT Pret,Nume FROM Produse LIMIT 2 INTO @p,$n;
ERROR 1172 (42000): Result consisted of more than one row
```

8.2.7. Structuri de control al executiei

8.2.7.1. Tipuri de instructiuni suportate de MySQL

MySQL suporta un set de instructiuni de control al executiei ce permit programarea in baza de date in adevaratul sens al cuvantului. Ele se impart in urmatoarele categorii:

- instructiuni decizionale – sunt folosite pentru a executa instructiuni SQL in mod conditionat, in functie de valori ale unor expresii definite de catre programator. Instructiuni disponibile: IF si CASE
- instructiuni repetitive – permit executarea in mod repetat si controlat a unei succesiuni de instructiuni SQL (ex: parcurgerea unui set de rezultate si prelucrarea succesiva a inregistrarilor componente). Instructiuni disponibile: LOOP, ITERATE, REPEAT, WHILE
- instructiuni pentru alterarea logicii executiei in cadrul unui bloc – permit incheierea fortata a iterarii curente sau a unui intreg bloc de tip repetitiv sau BEGIN...END. Instructiuni disponibile: LEAVE si ITERATE

Nota: *exista si o functie predefinita IF(), care este insa diferita de instructiunea IF discutata aici.*

8.2.7.2. Instructiuni decizionale

8.2.7.2.1. Instructiunea decizionala simpla (IF)

Instructiunea IF permite executarea conditionata a uneia din mai multe portiuni de cod SQL in functie de valoarea unor conditii specificate. Sintaxa sa generala este:

```
IF conditie1 THEN instructiuni1
    [ELSEIF conditie2 THEN instructiuni2] ...
    [ELSE instructiuni_final]
END IF;
```

Daca *conditie1* se evalueaza ca true se va executa primul set de instructiuni; in caz contrar, daca *conditie2* este true, se executa al doilea set de instructiuni s.a.m.d. Daca niciuna dintre conditii nu este indeplinita se va executa ultimul set de instructiuni – cel de pe ramura cu else.

Exemplu:

```
DELIMITER $
CREATE FUNCTION StareCivila(cod BIGINT) RETURNS VARCHAR(50) DETERMINISTIC
BEGIN
      DECLARE st TINYINT(1);
      SELECT stare FROM oameni WHERE cnp=cod INTO st;
      IF st=1 THEN RETURN 'casatorit';
      ELSEIF st=0 THEN RETURN 'necasatorit';
      ELSE RETURN 'stare civila necunoscuta';
      END IF;
END$
DELIMITER ;
```

8.2.7.2.2. Instructiunea decizionala multipla

Instructiunea decizionala multipla este folosita pentru a implementa o logica conditionala complexa. Ea are doua forme posibile – iat-o pe prima:

```
CASE expresie_de_referinta
    WHEN expresie1 THEN instructiuni1
    [WHEN expresie2 THEN instructiuni2] ...
    [ELSE instructiuni_final]
END CASE
```

In aceasta forma, instructiunea CASE functioneaza astfel:

- este evaluata valoarea expresiei de referinta
- este comparata pe rand aceasta valoare cu valorile expresiilor 1, 2 etc.
- daca valoarea expresiei de referinta este egala cu valoarea uneia dintre expresiile secundare, se executa codul din dreptul respectivei expresii secundare
- daca valoarea expresiei de referinta nu este egala cu niciuna dintre valorile expresiilor secundare:
  - daca exista ramura ELSE, se executa codul corespunzator ei
  - daca nu exista ELSE se va produce o eroare cu mesajul *Case not found for CASE statement*

Exemplu:

```
DELIMITER $
CREATE FUNCTION anotimp(d DATE) RETURNS VARCHAR(50) DETERMINISTIC BEGIN
```

```
    CASE INTERVAL(MONTH(d),3,5,8,12) -- produce 0 pt luna<3, 1 pt luna <5 etc
        WHEN 0 THEN RETURN 'iarna';
        WHEN 1 THEN RETURN 'primavara';
        WHEN 2 THEN RETURN 'vara';
        WHEN 3 THEN RETURN 'toamna';
        ELSE RETURN 'iarna'; -- singurul caz ramas: luna = 12
    END CASE;
END$
DELIMITER ;
SELECT anotimp('2010-03-14'); -- primavara
```

Cea de-a doua forma a instructiunii CASE este:

```
CASE
  WHEN expresie1 THEN instructiuni1
  [WHEN expresie2 THEN instructiuni2] ...
  [ELSE instructiuni_final]
END CASE
```

Functionarea ei este urmatoarea:

- se evalueaza pe rand fiecare expresie
- cand se intalneste prima expresie cu valoarea true se executa setul de instructiuni corespunzator ei
- daca nicio expresie nu are valoarea true:
  - daca exista ELSE se executa setul de instructiuni corespunzator lui
  - daca nu exista ELSE se genereaza o eroare (*Case not found for CASE statement*)

Aceasta a doua forma este utila atunci cand ne intereseaza testarea succesiva a mai multor conditii care nu presupun neaparat egalitatea:

- atunci cand valoarea testata poate fi si NULL, nu putem folosi prima forma a lui CASE deoarece compararea cu NULL nu produce niciodata TRUE; este necesara folosirea operatorilor IS NULL si IS NOT NULL
- atunci cand dorim verificarea unei relatii de inegalitate (ex: o valoare mai mare sau mai mica decat un prag)

```
DELIMITER $
CREATE FUNCTION categoriebyvarsta(varsta INT) RETURNS VARCHAR(50) DETERMINISTIC
BEGIN
CASE
    WHEN varsta BETWEEN 6 AND 18 THEN RETURN 'elev';
    WHEN varsta < 23 THEN RETURN 'student';
    ELSE RETURN 'somer';
END CASE;
END$
DELIMITER ;
```

8.2.7.3. Incheierea fortata a executiei unui bloc: instructiunea LEAVE

Instructiunea LEAVE este folosita atunci cand se doreste parasirea unui bloc BEGIN...END sau a unei instructiuni repetitive inaintea executarii tuturor instructiunilor componente ale blocului. LEAVE primeste ca parametru eticheta blocului pe care il incheie:

```
DELIMITER $
CREATE PROCEDURE test() BEGIN
SELECT 'a';
unu: BEGIN
    SELECT 'b';
    LEAVE unu;
    SELECT 'c';
END unu;
SELECT 'd';
```

```
END$
DELIMITER ;
CALL test();
-- vor fi afisate a,b si d, dar nu si c (deoarece LEAVE determina "sarirea" instructiunii sale)
```

In cazul blocurilor imbricate, LEAVE poate fi folosit pentru a incheia executia unui bloc de nivel mai inalt sau chiar a procedurii (in acest ultim caz fiind asemanator cu RETURN din functii, insa fara a produce o valoare):

```
CREATE PROCEDURE test2()
proc: BEGIN
            BEGIN
                  SELECT 'a';
                  LEAVE proc;
            END;
            SELECT 'b';
END proc$
CALL test2();
-- se afiseaza doar a (SELECT 'b' nu se mai executa)
```

8.2.7.4. Instructiuni repetitive

8.2.7.4.1. Generalitati si tipuri

Instructiunile repetitive ne permit sa executam in mod repetat o secventa de cod SQL fara a fi nevoie sa o scriem de tot atatea ori. Exista doua categorii de instructiuni disponibile:

- instructiunea LOOP creeaza o bucla infinita, care se incheie numai la cererea programatorului folosind instructiunea LEAVE sau RETURN
- instructiunile REPEAT...UNTIL si WHILE creeaza bucle care includ si o conditie de incheiere (ceea ce nu exclude iesirea fortata cu LEAVE sau RETURN in caz de nevoie!)

8.2.7.4.2. Instructiunea LOOP

Instructiunea LOOP este folosita pentru a executa in mod repetat o secventa de instructiuni. Sintaxa sa generala este:

```
[eticheta:] LOOP
    instructiuni
END LOOP [eticheta]
```

Succesiunea de instructiuni din cadrul buclei se executa in mod repetat, fara nicio conditie de oprire. Iesirea din bucla se efectueaza fie folosind instructiunea LEAVE, fie – in cazul unei functii – RETURN. **Atentie! Daca uitati sa folositi LEAVE sau RETURN, bucla va itera la infinit, blocand executia in acel punct!**

```
CREATE PROCEDURE test3() BEGIN
      DECLARE i INT DEFAULT 1;
      bucla: LOOP
          SET i = i+1;
          IF i=3 THEN LEAVE bucla;
          END IF;
      END LOOP;
      SELECT i;
END$
```

In exemplul de mai sus, instructiunile din cadrul buclei vor fi repetate pana cand *i* atinge valoarea 3, moment in care se paraseste bucla si se executa instructiunea ce-i urmeaza (afisarea lui *i*).

8.2.7.4.3. Instructiunea WHILE

Instructiunea WHILE aduce in plus fata de LOOP specificarea conditiei de continuare a repetitiei. Sintaxa sa generala este:

```
[eticheta:] WHILE expresie DO
    instructiuni
END WHILE [eticheta]
```

Setul de instructiuni cuprins in interiorul buclei va fi executat in mod repetat atat timp cat *expresie* se evalueaza ca true. Valoarea expresiei este reevaluata inaintea fiecarei executii a instructiunilor, ceea ce inseamna ca, daca expresia are valoarea false de la bun inceput, nu se va executa nici macar o repetitie.

8.2.7.4.4. Instructiunea REPEAT...UNTIL

Sintaxa generala a instructiunii REPEAT...UNTIL este:

```
[eticheta:] REPEAT
    instructiuni
UNTIL expresie -- valoarea expresiei reprezinta conditia de OPRIRE!
END REPEAT [eticheta]
```

Instructiunea REPEAT...UNTIL functioneaza pe aceleasi principii ca WHILE – repeta setul de instructiuni continut pana la comutarea valorii unei expresii - dar cu doua diferente majore:

- valoarea expresiei reprezinta conditia de OPRIRE, nu de continuare. **Setul de instructiuni continut se va repeta cat timp expresia se evalueaza ca FALSE!**
- valoarea expresiei este re-evaluata DUPA fiecare executie, ceea ce inseamna ca instructiunile continute se vor executa intotdeauna cel putin o data, chiar daca conditia de oprire a executiei este indeplinita de la bun inceput.

Exemplu: se genereaza in mod aleator doua numere pana cand se intampla ca ele sa fie egale, moment in care se afiseaza numarul de incercari:

```
DELIMITER $
CREATE PROCEDURE coincidenta() BEGIN
    DECLARE n1,n2,tries INT DEFAULT 0;
    REPEAT
        SET n1 = FLOOR(RAND()*10),
            n2 = FLOOR(RAND()*10),
            tries = tries+1;
    UNTIL n1=n2
    END REPEAT;
    SELECT CONCAT('Am reusit din ',tries,' incercari!') AS '';
END$
DELIMITER ;
CALL coincidenta();
```

8.2.7.4.5. Incheierea fortata a iterarii curente: instructiunea ITERATE

Instructiunea ITERATE este folosita in cadrul corpului instructiunilor repetitive (LOOP, REPEAT si WHILE) pentru a determina saltul la finalul iterarii curente. Ca si LEAVE, ea este urmata obligatoriu de eticheta blocului caruia i se aplica:

```
DELIMITER $
CREATE PROCEDURE iterare() BEGIN
    DECLARE i INT DEFAULT -1;
    bucla: WHILE i<3 DO
      SET i = i+1;
```

```
      IF i=1 THEN ITERATE bucla;
      END IF;
      SELECT i;
    END WHILE;
END$
DELIMITER ;
CALL iterare; -- vor fi afisate 0, 2 si 3, deoarece pentru i=1 este sarita instructiunea SELECT i
```

8.2.8. Conditii si handlere: gestionarea erorilor din cadrul unei rutine

8.2.8.1. Concepte

Un handler reprezinta un set de instructiuni ce se executa ca reactie la o anumit tip de eroare aparuta in executia codului SQL al unei rutine. In acest fel, programatorul poate detecta problemele aparute in executia codului sau si le poate trata. In lipsa unui handler, executia codului se incheie cu o eroare sau un warning.

Fiecare handler este asociat uneia sau mai multor conditii. O conditie descrie o anumita situatie de eroare aparuta in executia codului MySQL; asadar un acelasi handler poate fi apelat pentru unul sau mai multe tipuri de eroare.

Erorile SQL sunt descrise prin doi parametri:

- un cod numeric de eroare, care este un identificator intern al erorii in cadrul MySQL (nu este portabil pe alte sisteme de baze de date). Este un numar de 4 cifre
- un cod SQLSTATE, care este un identificator de eroare standardizat (se regaseste in standardul ANSI SQL) sub forma unui sir de 5 caractere. Nu toate codurile MySQL au coduri SQLSTATE corespondente; pentru cele fara corespondent se foloseste HY000

```
CREATE TABLE oameni(i int);
ERROR 1050 (42S01): Table 'oameni' already exists
-- cod MySQL 1050, SQLSTATE 42S01
```

8.2.8.2. Declararea conditiilor si a handlerelor

Atat conditiile, cat si handlerele trebuie definite folosind instructiunea DECLARE la inceputul blocului pentru care se doreste tratarea particulara a unei erori. Atunci cand in acel context exista mai multe instructiuni DECLARE, ordinea corecta de declarare este: variabile, conditii, cursoare si abia apoi handlere.

Sintaxele posibile de declarare a unei conditii sunt:

```
DECLARE nume_conditie CONDITION FOR cod_numeric_mysql
DECLARE nume_conditie CONDITION FOR SQLSTATE valoare_sqlstate
```

Numele conditiei poate fi folosit mai apoi pentru a o referi din cadrul definitiei unui handler. Observam ca tipul de eroare poate fi specificat in oricare dintre cele doua moduri (cod MySQL sau SQLSTATE).

Declaratia de handler ataseaza o portiune de cod (handlerul) uneia sau mai multor conditii declarate anterior:

```
DECLARE tip_handler HANDLER
    FOR conditie1, [conditie2,  ...]
    instructiune
```

Ori de cate ori va aparea una dintre conditiile de eroare specificate, se va executa codul ce constituie corpul handlerului. Instructiunea executata poate fi una bloc, permitand astfel tratarea complexa a erorilor aparute.

Tipul de handler controleaza modul in care evolueaza executia codului dupa rularea handlerului; valori posibile:

- **CONTINUE** va determina continuarea rularii codului SQL de la linia imediat urmatoare celei care a generat eroarea
- **EXIT** va determina iesirea din blocul BEGIN...END **in care este definit handlerul** (**chiar daca eroarea a aparut intr-un sub-bloc aflat in adancime!**)

Atunci cand are loc o eroare pentru care nu exista handler definit, actiunea implicita este EXIT.

Nota: *standardul SQL defineste si tipul de handler UNDO, insa el nu este inca suportat de MySQL in momentul scrierii acestui material.*

Conditiile pentru care este apelat handlerul pot fi specificate in urmatoarele moduri:

- simplu cod numeric MySQL
- valoare SQLSTATE, sub forma *SQLSTATE valoare*
- nume de conditie definita anterior cu DECLARE
- clasa predefinita de erori:
  - SQLWARNING – se refera la toate erorile al caror SQLSTATE incepe cu 01
  - NOT FOUND – se refera la erorile al caror SQLSTATE incepe cu 02 (util mai ales in cazul cursoarelor, pentru a detecta finalul unui result set)
  - SQLEXCEPTION – cuprinde toate erorile al caror SQLSTATE nu incepe cu 01, 02 sau 03

Exemplu: la incercarea de introducere a unei persoane cu CNP duplicat consemnam incidentul intr-o tabela de logging:

```
DECLARE dublarepersoana CONDITION FOR 1062;
-- echivalent cu a scrie:
-- DECLARE dublarepersoana CONDITION FOR SQLSTATE 23000;
DECLARE CONTINUE HANDLER FOR dublarepersoana
BEGIN
	INSERT INTO Logs(Moment,Text) VALUES(NOW(),CONCAT('Persoana duplicat'));
END;
```

Nota: *pentru a ignora o eroare, i se poate atasa acesteia un handler de tip CONTINUE cu corp vid (BEGIN END;)*

8.2.9. Cursoare: parcurgerea rand cu rand a inregistrarilor unui result set

8.2.9.1. Concepte si pasi de urmat

Chiar daca instructiunile de control al executiei de pana acum ne-ar permite sa parcurgem seturi de inregistrari in scopul prelucrarii lor, acest lucru s-ar realiza ineficient (folosind instructiuni SELECT separate sau cu clauze LIMIT). Cursoarele ne ofera posibilitatea de a parcurge rand cu rand un result set, formuland interogarea o singura data si avand apoi acces la valorile componente ale fiecarei inregistrari.

Utilizarea unui cursor consta din urmatorii pasi:

- declararea cursorului folosind DECLARE si formularea interogarii SELECT ce produce result set-ul de parcurs
- “deschiderea” cursorului – executarea interogarii si memorarea temporara a result set-ului
- accesarea si prelucrarea datelor, inregistrare cu inregistrare
- inchiderea cursorului – eliberarea memoriei ocupate de result set

Un cursor poate fi deschis si inchis de mai multe ori pe parcursul unei rutine, oferind de fiecare data acces la datele corespunzatoare interogarii.

Cursoarele MySQL au urmatoarele limitari:

- ofera acces read-only la date (modificarea datelor obtinute prin intermediul cursorului nu produce modificari in tabela de provenienta)
- cursorul parcurge result set-ul numai inainte si cu cate o singura inregistrare (nu ne putem intoarce si nu putem sari inregistrari)

8.2.9.2. Definirea unui cursor

Crearea unui cursor se realizeaza folosind instructiunea DECLARE, care atribuie un nume cursorului si specifica interogarea ce-i produce datele. Sintaxa este:

```
DECLARE nume_cursor CURSOR FOR interogare_select
```

Restrictii:

- interogarea trebuie sa fie una de tip SELECT, dar nu SELECT...INTO
- numele cursorului nu trebuie sa fie identic cu al altui cursor din acelasi bloc de instructiuni.
- in cazul existentei mai multor instructiuni DECLARE in cadrul acelui bloc, locul cursoarelor este intre conditii si handlere

Exemplu:

```
DECLARE de_triat CURSOR FOR SELECT Nume,Prenume,Inaltime FROM Oameni WHERE Greutate>80
```

8.2.9.3. Utilizarea cursorului pentru accesarea datelor

Imediat dupa declarare, un cursor este un simplu nume pus in corespondenta cu o interogare. Pentru a-l putea folosi in scopul accesarii datelor este necesara executarea interogarii si memorarea temporara a datelor, folosind instructiunea OPEN:

```
OPEN nume_cursor
```

Odata deschis, cursorul dispune de un pointer intern care indica inregistrarea curenta. Obtinerea datelor acestei inregistrari se realizeaza cu instructiunea FETCH, care plaseaza valorile coloanelor in variabilele specificate si avanseaza cu o pozitie pointerul intern:

```
FETCH nume_cursor INTO variabila1 [,variabila2, ....]
```

Variabilele in care FETCH salveaza datele pot fi de orice natura (locale, de sesiune, parametri de tip OUT sau INOUT). Numarul de variabile trebuie sa corespunda cu numarul de coloane produse de interogarea ce defineste cursorul.

In cazul in care s-a ajuns la sfarsitul result set-ului si deci nu mai exista inregistrari este generata o eroare *No data*, cu codul MySQL 1329 si SQLSTATE 02000. Detectia epuizarii result set-ului se realizeaza definind un handler pentru aceasta eroare (vezi exemplul de mai jos).

Odata incheiata parcurgerea datelor, memoria ocupata de catre acestea trebuie eliberata folosind instructiunea CLOSE:

```
CLOSE nume_cursor
```

Avand in vedere ca fiecare apel catre FETCH produce o singura inregistrare, instructiunea FETCH lucreaza foarte bine in colaborare cu structurile de control al executiei (LOOP, WHILE, REPEAT):

```
CREATE PROCEDURE triere() BEGIN
    DECLARE n VARCHAR(100);
    DeCLARE h INT;
    DECLARE greii CURSOR FOR SELECT Nume,Inaltime FROM oameni WHERE Greutate>80;
    DECLARE EXIT HANDLER FOR 1329 BEGIN END;
    OPEN greii;
    bucla: LOOP
        FETCH greii INTO n,h;
```

```
        IF h < 180 THEN ITERATE bucla;  -- ii pastram doar pe cei mai inalti de 180cm
        END IF;
        INSERT INTO NumeAlesi(Name) VALUES(n);
    END LOOP;
    CLOSE greii;
END$
```

O implementare alternativa a procedurii de mai sus ar fi fost sa definim un handler de tip CONTINUE care seteaza un flag atunci cand parcurgerea result set-ului s-a incheiat. Avantajul in acest caz este ca la executarea handlerului nu se iese din intregul bloc BEGIN...END, ci doar din blocul LOOP, ceea ce permite executarea altor instructiuni pe finalul procedurii:

```
CREATE PROCEDURE triere2() BEGIN
    DECLARE n VARCHAR(100);
    DECLARE h,gata INT DEFAULT 0;
    DECLARE greii CURSOR FOR SELECT Nume,Inaltime FROM oameni WHERE Greutate>80;
    DECLARE CONTINUE HANDLER FOR 1329 BEGIN SET gata=1; END;
    OPEN greii;
    bucla: LOOP
        FETCH greii INTO n,h;
        IF gata=1 THEN LEAVE bucla;END IF;
        IF h < 180 THEN ITERATE bucla;
        END IF;
        INSERT INTO NumeAlesi(Name) VALUES(n);
    END LOOP;
    CLOSE greii;
    SELECT * FROM NumeAlesi;
END$
```

8.3. Trigger-e

8.3.1. Concepte

Un trigger reprezinta o secventa de cod SQL asociata unei tabele si care poate fi executata ca reactie la operatiile care modifica date din acea tabela - inainte sau dupa efectuarea operatiei - putand astfel modifica/valida valorile care patrund in tabela.

Un trigger poate fi folosit in numeroase moduri – iata cateva idei:

- un trigger poate valida sau chiar modifica datele care patrund intr-o tabela prin intermediul operatiilor de tip UPDATE sau INSERT. In acest fel putem impune anumite reguli de validitate a datelor din insasi baza de date (ex: o varsta de persoana sa fie intre 0 si 115, desi tipurile de date INT permit valori superioare; formatarea intr-un anume fel a numerelor de telefon – eliminarea punctelor si a caracterelor - (minus) etc)
- un trigger poate avea acces la datele unei inregistrari inainte ca aceasta sa fie introdusa/modificata in tabela. Putem astfel sa anulam cu totul operatia, sa modificam datele ce patrund in tabela sau sa consemnam operatia intr-o tabela cu loguri
- cu ajutorul unui trigger putem stabili valori default dinamice (calculate pe baza unei expresii) pentru coloane ale caror tipuri de date nu suporta acest lucru. Exemplul clasic este cel al coloanelor de tip DATE sau DATETIME, care nu pot primi ca valoare default CURRENT_DATE sau CURRENT_TIMESTAMP in definitia coloanei, dar care pot fi initializate cu acele valori printr-un trigger

8.3.2. Definirea unui trigger

Un trigger se defineste cu ajutorul instructiunii CREATE TRIGGER avand sintaxa urmatoare:

```
CREATE TRIGGER nume_trigger      moment_declansare   tip_operatie
      ON nume_tabela FOR EACH ROW instructiune
```

Parametrii instructiunii sunt urmatorii:

- tipul de operatie indica instructiunea SQL aplicata tabelei care declanseaza triggerul. Valori posibile:
  - INSERT – triggerul va fi activat pentru operatii care introduc inregistrari noi. Acestea includ nu numai instructiunea INSERT, ci si REPLACE, LOAD DATA INFILE etc.
  - UPDATE – triggerul va fi activat pentru operatii care modifica inregistrari existente (instructiuni UPDATE, REPLACE etc)
  - DELETE – triggerul va fi activat in cazul instructiunilor care produc stergerea inregistrarilor (DELETE, REPLACE). **Atentie! Triggerul nu este activat pentru DROP TABLE si TRUNCATE TABLE, deoarece acestea nu folosesc DELETE!**
- momentul declansarii stabileste daca triggerul va rula inaintea efectuarii operatiei (putand astfel opri executia operatiei) sau dupa (modificand datele deja introduse/actualizate). Valori posibile: BEFORE si AFTER
- instructiunea ce formeaza corpul trigger-ului poate fi una compusa si trebuie sa se supuna urmatoarelor restrictii:
  - corpul unui trigger nu poate contine instructiuni care, explicit sau implicit, demareaza sau incheie o tranzactie
  - un trigger nu poate produce un result set

Nota: *daca o instructiune SQL modifica/introduce mai multe inregistrari, eventualul trigger asociat va fi rulat cate o data pentru fiecare inregistrare.*

8.3.3. Referirea la valorile coloanelor din cadrul unui trigger

In functie de momentul in care actioneaza triggerul (BEFORE sau AFTER) este posibil sa avem nevoie de acces la vechea valoare a coloanei sau/si la noua valoare. Putem folosi urmatoarea sintaxa:

- **NEW.numecoloana** se refera la noua valoare a coloanei. *NEW.numecoloana* poate fi folosit numai in cazul operatiilor de tip UPDATE si INSERT, deoarece in cazul DELETE nu exista inregistrare/valoare noua. In cadrul unui trigger de tip BEFORE putem modifica noua valoare folosind *SET NEW.numecoloana=expresie*.
- **OLD.numecoloana** se refera la vechea valoare a coloanei, lucru posibil numai pentru operatiile UPDATE si DELETE (in cazul lui INSERT nu exista valoare veche). **Atentie! Vechea valoare a coloanei este read-only!**

Nota: *vechea valoare a coloanei este disponibila chiar daca triggerul este de tip AFTER UPDATE, si deci la apelarea sa inregistrarea a fost deja modificata!*

Exemplu: un trigger care aduce inaltimea unei persoane in intervalul 0...230 cm:

```
CREATE TRIGGER procust BEFORE INSERT ON oameni
FOR EACH ROW BEGIN
       IF NEW.inaltime<0 THEN SET NEW.inaltime=0;
       ELSEIF NEW.inaltime>230 THEN SET NEW.inaltime = 230;
       END IF;
END$
```

8.3.4. Vizualizarea triggerelor

Pentru a obtine lista de triggere definite pentru o tabela putem folosi instructiunea SHOW TRIGGERS cu sintaxa urmatoare:

```
SHOW TRIGGERS [FROM  bazadedate] [LIKE 'pattern_tabela']
```

Atentie! La utilizarea clauzei LIKE, pattern-ul nu este comparat cu numele triggerului, ci cu al tabelei de care apartine triggerul! Astfel putem determina lista de triggere pentru o anume tabela scriind:

```
SHOW TRIGGERS LIKE 'oameni'\G
*************************** 1. row ***************************
  Trigger: validare_inaltime
```

```
    Event: INSERT
    Table: oameni
Statement: BEGIN
      IF NEW.inaltime<0 THEN SET NEW.inaltime=0;
      ELSEIF NEW.inaltime>230 THEN SET NEW.inaltime = 230;
      END IF;
END
   Timing: BEFORE
  Created: NULL
 sql_mode:
  Definer: root@localhost
```

Incepand cu MySQL 5.1.21, definitia unui trigger poate fi vizualizata si cu:

```
SHOW CREATE TRIGGER numetrigger
```

8.3.5. Stergerea triggerelor

Instructiunea folosita pentru stergerea unui trigger este DROP TRIGGER, cu urmatoarea sintaxa:

```
DROP TRIGGER [IF EXISTS] [bazadedate.]nume_trigger
```

8.4. BIBLIOGRAFIE

- MySQL Reference Manual
  - Stored routines
    - http://dev.mysql.com/doc/refman/5.1/en/stored-programs-views.html
    - http://dev.mysql.com/doc/refman/5.1/en/sql-syntax-compound-statements.html
  - Mesaje si coduri de eroare:
    - http://dev.mysql.com/doc/refman/5.1/en/error-messages-server.html
- Carti:
  - MySQL 5 Certification Study Guide – cap. 18,19
  - MySQL (4th Edition) – cap. 4
  - MySQL Stored Procedure Programming(O'Reilly 2006)

8.5. ANEXA

Conventie

-referitoare la conditiile de desfasurare a examenului final-

1. **Caracteristici generale ale examenului final teoretic :**

   a. se numeste Final Exam
   b. este format din intrebari selectate din toate examenele de capitol
   c. este accesibil pe http://www.infoacademy.net → Login → Student home → Nume_Clasa → TakeAssesment
   d. este de tip test grila si are in medie 30-60 de intrebari cu raspuns unic sau multiplu
   e. poate fi sustinut de maxim 2 ori, la interval de 1-2 saptamani intre cele 2 incercari
   f. poate fi sustinut numai la sediul academiei InfoAcademy in prezenta unui reprezentant InfoAcademy
   g. se considera promovat daca numarul de raspunsuri corecte este cel putin egal cu 75% din numarul total al raspunsurilor (la a doua incercare pragul scade la 70%)
   h. Dureaza 60 de minute

2. **Conditii de participare la examenul final teoretic :**

   Poate participa la examenul final teoretic orice student Infoacademy care :

   a. A promovat toate examenele partiale cu cel putin 75% (optim 85%)
   b. Se afla in termenul contractual (maxim 4 saptamani de la data ultimei predari).
   c. A achitat la timp obligatiile financiare convenite la inscriere
   d. Se prezinta la examenul final la data si ora aleasa in momentul inscrierii on-line la examenul final (prin www.infoacademy.net)
   e. Are asupra sa un act cu fotografie care ii atesta identitatea
   f. Se obliga sa respecte conditiile de desfasurare ale examenului final incluse in aceasta Conventie

3. **Inscrierea la examenul final**

   a. Poate fi facuta prin www.infoacademy.net cu cel mult 4 saptamani inainte de data la care se doreste sustinerea examenului final
   b. Este recomandabil sa fie facuta din timp (chiar inainte de finalizarea modulului pentru care se face inscrierea) pentru acomodarea psihologica, pregatirea temeinica a examenului si accesul la data si ora dorite.
   c. Se poate renunta la inscrierea facuta, prin ANULAREA INSCRIERII (accesarea linkului de anulare trimis in e-mailul de confirmare a inscrierii, sau folosirea facilitatii de anulare din formularul de inscriere).

4. **In timpul desfasurarii examenului final , studentul se obliga :**

   a. sa nu comunice direct sau prin dispozitive electronice cu alte persoane
   b. sa nu salveze pe nici un fel de suport intrebarile/imaginile/paginile incluse in examenul sustinut
   c. sa nu incerce sa transmita prin Intranet/Extranet/Internet continutul intrebarilor, imagini sau pagini web continute in examen sau aflate pe calculatorul de pe care sustine examenul
   d. sa nu utilizeze materiale scrise/inregistrate pentru a afla raspunusuri la intrebarile incluse in examen
   e. sa nu utilizeze dispozitive de calcul (de tip calculatorul inclus in telefon,ceas, sistemul de operare, etc..) pentru a efectua operatii de orice tip.
   f. Sa nu aiba asupra sa dispozitive de inregistrare audio/video (de tip telefon celular,etc..)
   g. Sa nu afecteze prin tinuta sau comportament desfasurarea examenului final .

5. **Dupa terminarea examenului final, studentul :**

   a. Iese din cont (Logout), preda ciorna si paraseste sala de examen
   b. In maxim o saptamana completeaza InfoAcademy feedback. Acesta este disponibil pe www.infoacademy.net, in contul de student InfoAcademy, prin intermediul link-ului Course Feedback din cadrul clasei urmate
   c. In maxim 3 luni de la data examenului final depune cererea de diploma prin www.infoacademy.net
   d. Se prezinta la academie in cel mult o luna de la primirea e-mailului care ii confirma ca poate ridica diploma
   e. Pentru ridicarea diplomei studentul se identifica cu un act cu fotografie care ii atesta identitatea.

6. **Incalcarea CONVENTIEI**

   a. Incalcarea prevederilor articolului 4 a,b,c,d,e,f duce la eliminarea din examenul final si din academie cu pierderea oricaror drepturi asupra modulului al carui examen final se sustine.
   b. Incalcarea prevederilor articolului 4.g duce la eliminarea studentului din examenul final cu pierderea uneia din cele doua sanse de sustinere a acestuia

Prezentarea la examenul final echivaleaza cu acceptarea de catre participant a termenilor acestei conventii; examinatul este de acord sa o respecte intocmai si se supune sanctiunilor acestei conventii in vigoare in cazul nerespectarii ei.