6.SQL PROGRAMATIC
PROCEDURI STOCATE,FUNCȚII CURSOARE,TRIGGERE,EVENIMENTE

I. OBIECTIVE

1. Studiu , implementare și utilizare proceduri stocate ,cursoare, funcții
2. Studiu , implementare și utilizare triggere și evenimente

II. FUNDAMENTE TEORETICE

2.1.PROCEDURI STOCATE , FUNCȚII ȘI CURSOARE

În dezvoltarea aplicațiilor cu baze de date sunt posibile două abordări pentru stocarea și execuția programelor și anume: programele pot fi memorate *local*, în cadrul aplicațiilor ce trimit comenzi către serverul de baze de date și prelucreaza rezultatele trimise, respectiv posibilitatea de a dezvolta programele și a le înregistra ca *proceduri stocate în server* și de a crea aplicații care apelează aceste proceduri și doar prelucrează rezultatele returnate de acestea. Astfel, *o procedură stocată este un set de instrucțiuni SQL stocat pe serverul de baze de date ca un obiect distinct*.

Procedurile stocate sunt similare procedurilor din alte limbaje de programare, în sensul în care acceptă parametrii de intrare și returnează valori prin parametrii de ieșire către un program apelant, conțin instrucțiuni de programare care efectuează operații în baza de date, respectiv returnează către apelant o valoare ce indică succesul sau eșecul execuței procedurii. Procedurile stocate pot să *returneze valori* și *să modifice valori* de variabile , *să execute operații* de comparare cu valorile de variabile folosite de sistem. Procedurile stocate pot să primească și să returneze valori care nu provin neapărat dintr-o tabelă, ci sunt calculate prin execuția procedurii.

Utilizarea procedurilor stocate prezintă o serie de avantaje , cele mai importante fiind **programarea modulară** (procedura este creată o singură data și aplelata de mai multe ori) , **imbunatățirea performanțelor** (compilarea si optimizarea lor se realizeaza o singură dată, la crearea procedurii, fiind memorate într-o formă direct executabilă), **reducerea traficului în rețea** (o prelucrare poate presupune execuția a sute de linii de cod ce poate fi realizată printr-un simplu apel de procedură stocată) și support pentru **mecanisme suplimentare de securitate** , astfel anumiți utilizatori nu au acces direct la codul procedurilor stocate, iar dreptul de execuție poate fi acordat distinct , doar utilizatorilor ce au nevoie.

Se impun câteva observații legate de crearea și utilizarea procedurilor stocate și anume:

1. Pot fi create proceduri stocate pentru operațiile de adaugare/ modificare/ stergere/ citire controland astfel in mod programat fiecare din aceste tipuri de operații
2. Fiind stocata pe server, procedura beneficiază de acces *direct, rapid și imediat la baza de date*
3. Simplifică *interogările parametrizate* facilitând *rularea repetată* a aceleiasi interogări cu seturi diferite de date
4. Oferă facilități pentru dezvoltarea aplicațiilor cu baze de date în maniera Client /Server prin *modularizarea aplicației* separând astfel modulele pentru client și server.
5. Componentele de pe server (procedurile stocate) pot fi *dezvoltate separat* de componentele client ,fapt ce face posibilă *refolosirea* lor între aplicațiile client
6. Impun pe partea de server reguli orientate spre date, implementează reguli și relații logice care sporesc controlul informațiilor in sistem;

Sintaxa de creare a unei proceduri stocate /funcții este următoarea :

```
CREATE
    [DEFINER = user]
    PROCEDURE sp_name ([proc_parameter[,...]])
    [characteristic ...] routine_body

CREATE
    [DEFINER = user]
    FUNCTION sp_name ([func_parameter[,...]])
    RETURNS type
    [characteristic ...] routine_body

proc_parameter:
    [ IN | OUT | INOUT ] param_name type

func_parameter:
    param_name type

type:
    Any valid MySQL data type
 characteristic:
COMMENT 'string'
  | LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }

routine_body:
    Valid SQL routine statement
```

Se observă faptul că sintaxa este comună pe anumite aspecte, totuși vor fi menționate diferențele existente. În definirea procedurii pot fi utilizate clauze diverse. Astfel,clauza DEFINER asociază rutinei stocate un utilizator ale cărui privilegii vor fi verificate la momentul execuţiei sale.
Dacă valoarea SQL SECURITY e DEFINER se verifică privilegiile utilizatorului care a creat procedura stocată, iar dacă este INVOKER se verifică privilegiile utilizatorului care apelează procedura stocată. Clauzele DETERMINISTIC, respectiv NON-DETERMINISTIC (implicit) descriu comportamentul procedurii stocate. O rutină(procedură) este deterministică dacă produce întotdeauna aceleaşi rezultate pentru aceiaşi parametri de intrare, respectiv este nedeterministică dacă la execuţii diferite va avea execuții diferite, imprevizibile. Spre exemplu, apelul unor funcţii precum NOW() sau RAND() poate determina generarea unui comportament nedeterministic. MySQL nu verifică corectitudinea comportamentului declarat, însă declararea sa greşită poate determina modulul de optimizare să realizeze *planuri de execuţie neperformante*, deci timp de răspuns la interogarea datelor mult prea mare.

Pentru o procedură stocată pot fi precizate clauze care indică modul în care sunt utilizate datele în cadrul acesteia, fără a limita operaţiile permise:

- CONTAINS SQL – specifică faptul că rutina stocată nu conţine instrucţiuni care citeşte sau scrie date; NO SQL – arată că rutina stocată nu conţine comenzi SQL
- READS SQL DATA – indică faptul că rutina stocată conţine instrucţiuni care citesc date (cum ar fi SELECT), dar nu şi instrucţiuni care le scriu;
- MODIFIES SQL DATA – precizează faptul că rutina stocată poate conţine instrucţiuni care scriu date (precum INSERT, UPDATE sau DELETE).

Procedurile stocate trebuie să întoarcă un scalar. De asemenea, în timp ce procedurile pot apela instrucţiuni ce întorc mai multe valori, acest lucru nu este permis în cadrul funcţiilor.

În mod similarInstructiunea **DROP PROCEDURE** <*proc_nume*> și respectiv **ALTER PROCEDURE** <*proc_nume*>, permit ștergerea sau modificarea procedurii.

Procedura are un *nume, o listă de parametrii* (de intrare IN, de ieșire OUT sau de intrare/iesire INOUT) și *corpul procedurii* ce conține instrucțiunile ce trebuie executate separae prin caracterul ; acestea fiind încadrate de cuvintele cheie BEGIN..END . În corpul procedurii se pemite specificarea de variabile locale (notate cu @x) sau utilizarea de variabile sistem, utilizarea de structuri de control și mecanisme pentru gestionarea erorilor. Parametrii de tip IN sunt *parametrii impliciți.*Există și proceduri fără parametrii.

Întrucât pentru clientul MySQL caracterul ';' reprezintă un *delimitator* care marchează încheierea unei instrucţiuni, acest tip de delimitator trebuie să fie redefinit astfel încât întreaga definiţie a rutinei stocate să fie transmisă serverului. Acest lucru se realizează prin intermediul cuvântului-cheie DELIMITER. Un delimitator poate constă dintr-unul sau mai multe caractere, cele mai uzuale fiind // sau */.

Resurse suplimentare:http://www.mysqltutorial.org/mysql-stored-procedure-tutorial.aspx
http://www.mysqltutorial.org/mysql-stored-function/

Crearea unei proceduri fără parametrii,care conține mai multe fraze SQL:

```
CREATE PROCEDURE P1 ()
BEGIN
        SET @a = 5;
        SET @b = 5;
        INSERT INTO t VALUES (@a);
        SELECT s1 * @a FROM t WHERE s1 >= @b;
END//
```

Crearea unei proceduri cu declarații de variabile:

```
CREATE PROCEDURE P2()
BEGIN
        DECLARE a INT /* no DEFAULT clause */;
        DECLARE b INT /* no DEFAULT clause */;
        SET a = 5; /* SET statement */
        SET b = 5; /* SET statement */
        INSERT INTO t VALUES (a);
        SELECT s1*a  FROM t WHERE s1 >= b;
END; //
```

Crearea unei proceduri care conține clauza DEFAULT:

```
CREATE PROCEDURE P3()
BEGIN
        DECLARE a,b INT DEFAULT 5
        SELECT s1 * a FROM t WHERE s1 >= b;
        INSERT INTO t VALUES (a);
END;//
```

Funcțiile în MySQL sunt construcții ce returnează valori ce pot fi utilizate în fraze SQL valide. Funcțiile sunt utile pentru diverse implementări auxiliare necesare procesării în aplicații cu baze de date,

însă nu sunt permise manipulările de date din tabele, modificarea privilegiilor de acces la date, sau controlul tranzacțiilor. Un exemplu de creare a unei funcții :

```
CREATE FUNCTION factorial (n DECIMAL(3,0))
  RETURNS DECIMAL(20,0)
  DETERMINISTIC
 BEGIN
       DECLARE factorial DECIMAL(20,0) DEFAULT 1;
       DECLARE counter DECIMAL(3,0); SET counter = n; factorial_loop:
REPEAT
       SET factorial = factorial * counter;
       SET counter = counter - 1;
UNTIL counter = 1
END REPEAT;
RETURN factorial;
END //
```

Resurse suplimentare: http://www.mysqltutorial.org/mysql-functions.aspx

Diferenţa dintre o procedură şi o funcţie SQL constă în modul prin care rezultatul este întors. Astfel, o procedură poate realiza modificări asupra tabelelor fără să producă o valoare în mod necesar, iar în cazul când aceasta trebuie să fie vizibilă către programul care a apelat-o, transferul său se poate realiza prin intermediul parametrilor. O funcţie întoarce rezultat în mod necesar

Variabile și structuri de control permise în corpul unei proceduri vor fi descrise succint prin intermediul exemplelor.

Pot fi utilizate variabile definite de utilizator, a căror durată de viaţă nu depăşeşte însă sesiunea curentă. Sintaxa d ecreare a unei variabile este :

```
SET variable = expr [, variable = expr] ...
 variable: {
user_var_name
| param_name
  | local_var_name
  | {GLOBAL | @@GLOBAL.} system_var_name
  | {PERSIST | @@PERSIST.} system_var_name
  | {PERSIST_ONLY | @@PERSIST_ONLY.} system_var_name
  | [SESSION | @@SESSION. | @@] system_var_name}
```

Numele de variabile folosesc caractere alfa-numerice precum şi caracterele ., _ şi $, acesta fiind case in-sensitive. Atribuirea se poate face fie folosind =, fie folosind :=.

Variabilele pot lua doar valori de tip numeric sau şir de caractere (binar sau non-binar), însă în cazul numerelor reale poate avea loc o pierdere a preciziei. Atribuirile care folosesc tipuri nepermise vor fi convertite în mod automat către un tip permis. De asemenea, o variabilă poate avea valoarea NULL.O variabilă ce nu a fost iniţializată are valoarea NULL iar tipul de date asociat este şir de caractere.

Construcții de control a fluxului

IF-THEN-ELSE Sintaxa

```
IF search_condition THEN statement_list
    [ELSEIF search_condition THEN statement_list] ...
    [ELSE statement_list]
END IF
```

Exemplu

```
CREATE PROCEDURE P4(IN parameter1 INT)
BEGIN
        DECLARE variable1 INT;
        SET variable1 = parameter1 + 1;
        IF variable1 = 0 THEN
                INSERT INTO t VALUES (17);
        END IF;
        IF parameter1 = 0 THEN
                UPDATE t SET s1
                = s1 + 1;  ELSE
                UPDATE t SET s1 = s1 + 2;
        END IF;
END; //
```

Sintaxă CASE

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

Exemplu

```
CREATE PROCEDURE P5 (IN parameter1 INT)
BEGIN
        DECLARE variable1 INT;
        SET variable1 = parameter1 + 1;
        CASE variable1
        WHEN 0 THEN INSERT INTO t VALUES (17);
        WHEN 1 THEN INSERT INTO t VALUES (18);
        ELSE INSERT INTO t VALUES (19);   END
        CASE;
END; //
```

WHILE..END WHILE Sintaxă

```
[begin_label:] WHILE search_condition DO

statement_list
END WHILE [end_label]
```

Exemplu

```
CREATE PROCEDURE P6 ()
BEGIN
        DECLARE v INT;
        SET v = 0;
        WHILE v < 5 DO
        INSERT INTO t VALUES (v);
        SET v = v + 1;
        END WHILE;
END; //
```

REPEAT ... END REPEAT Sintaxă

```
[begin_label:] REPEAT
statement_list UNTIL
search_condition END
REPEAT [end_label]
```

Exemplu

```
CREATE PROCEDURE P7 ()
        BEGIN
                DECLARE v INT;
                SET v = 0;
                REPEAT
                        INSERT INTO t VALUES (v);
                        SET v = v + 1;
                UNTIL v >= 5
                END REPEAT;
        END; //
```

LOOP ... END LOOP Sintaxă

```
[begin_label:] LOOP
statement_list
END LOOP [end_label]
```

Exemplu

```
CREATE PROCEDURE P8 ()
BEGIN
        DECLARE v INT; SET v
        = 0;   loop_label:
        LOOP   INSERT INTO
        t VALUES
        (v);
        SET v = v + 1;
        IF v >= 5 THEN
        LEAVE loop_label;
        END IF;
        END LOOP;
END; //
```

GOTO

```
CREATE PROCEDURE P9 (IN parameter_1 INT, OUT parameter_2 INT)
LANGUAGE SQL DETERMINISTIC SQL SECURITY INVOKER
        BEGIN
        DECLARE v INT;   label
goto_label;
start_label: LOOP IF v = v THEN LEAVE start_label;
ELSE ITERATE start_label;
END IF;
END LOOP start_label;
REPEAT
WHILE 1 = 0 DO BEGIN END;
END WHILE;
UNTIL v = v END REPEAT;
GOTO goto_label;
END;/
```

CURSOARE

Specific bazelor de date relaționale și limbajului SQL este modul de operare asupra relațiilor ca un tot unitar, astfel orice operatie se realizează asupra setului complet de tuple care satisface anumite condiții. Cursorul este un *mod complementar* de lucru, reprezentând un mecanism care permite alocarea unei zone de memorie și accesarea informației provenită dintr-o instrucțiune SQL. Un cursor este similar unui pointer la zona de context care în cazul bazelor d edate este secvența de tuple.

Prin cursor, un program SQL poate să controleze zona de context și ce se întâmplă cu aceasta atunci când instrucțiunea SQL este prelucrată. Această informație include numărul de linii care au fost prelucrate de instrucțiune, un pointer la reprezentarea instrucțiunii analizate (analiza unei instrucțiuni SQL este procesul prin care informația este transferată la server, timp în care instrucțiunea SQL este evaluata ca validă). Intr-un query, setul activ se referă la liniile care vor fi returnate. Pentru ca MySQL să proceseze o instrucțiune SQL, este nevoie ca el să creeze o zonă de memorie cunoscută ca zona/buffer de context; aceasta va conține informația necesară prelucrării instrucțiunilor.

Caracteristicile unui cursor MySQL sunt:

1. ASENSITIVE - serverul poate realiza sau nu copii ale tabelului care conţine rezultatele care sunt parcurse;
2. Read-ONLY - proprietatea de a nu putea fi suprascrise;
3. NONSCROLLABLE - proprietatea de a putea fi parcurse într-o singură direcţie şi în ordine.

Definirea și operații de manipulare a cursoarelor

Operaţia de tip SELECT care este asociată cursorului nu poate avea clauza INTO. O rutină stocată poate defini mai multe cursoare însă fiecare trebuie identificat printr-o denumire unică.
Declararea unui cursor de realizează astfel :

```
DECLARE cursor_name CURSOR FOR select_statement
```

```
OPEN cursor_name;
```

Deschidere cursorului necesară pentru ca acesta să poată fi folosit.;

Obținerea valorilor. În momentul în care este apelată, se trece la următoarea înregistrare (dacă aceasta există), iar valorile sunt transferate în cele ale variabilelor definite, numărul lor trebuind să corespundă cu al celor din instrucţiunea SELECT asociată declarării cursorului.

```
FETCH [[NEXT] FROM] cursor_name INTO var_name [, var_name] ...
```

Închiderea cursorului.În situaţia în care cursorul nu este dechis, operația va genera o eroare. Dacă nu este închis explicit, acest lucru se va realiza automat la sfârşitul blocului BEGIN … END în care a fost definit.

```
CLOSE cursor_name;
```

Exemplu:

```
BEGIN
        DECLARE a,b INT;
       DECLARE cur_1 CURSOR FOR SELECT s1 FROM t;
        DECLARE CONTINUE HANDLER FOR NOT FOUND
        SET b = 1;
        OPEN cur_1;
        REPEAT FETCH cur_1 INTO a;
        UNTIL b = 1
        END REPEAT;
               CLOSE cur_1;
        SET return_val = a;
END;//
```

Detalii suplimentare: http://www.mysqltutorial.org/mysql-cursor/

2.2.TRIGGERE ȘI EVENIMENTE

Un trigger este o secvență de cod care este executată în mod implicit (automat fără a necesita un apel ca și procedura stocată) atunci când asupra tabelului asociat se execută o operație de acces la date , altfel spus, ori de câte ori este executată o instrucțiune INSERT, UPDATE sau DELETE, pentru a verifica respectarea unor constângeri sau pentru a realiza anumite calcule asupra unor valori.

Triggerele sunt utile deoarece permit implementarea regulilor de bussines specifice aplicației, în mod direct atașat bazei de date,astfel în afara codului aplicației . Ele reprezintă pre-filtre sau postfiltre care pot să împiedice sau să execute anumite activități asupra datelor.Triggerele se referă la acele reguli, în general constrângeri care țin în mod inerent de structura bazei de date și sunt induse de semantica colecției de date și relativ independente de fiecare aplicație în parte. Triggerele sunt mecanisme prin care SQL ofera programatorilor de aplicații și proiectanților de baze de date posibilitatea să completeze mecanismele de verificare a integrității bazei de date (constrângeri, valori implicite) într-un mod flexibil și complex.

Triggerele permit execuția unei secvențe de cod SQL, fiind considerate proceduri speciale deasemenea salvate în baza de date care generează automat anumite valori ce derivă din valorile coloanelor, determină respectarea restricțiilor și privilegiilor, face posibilă jurnalizarea transparentă a evenimentelor din baza de date, colectează în mod informal statistici în legătură cu modul de accesare a tabelelor.Totuși , instrumentele declarative (chei străine, valori implicite) și procedurile stocate vor fi preferate atunci când este posibil datorită simplității în implementare.

Sintaxa de creare a unui trigger:

```
CREATE
    [DEFINER = user]
TRIGGER trigger_name
trigger_time trigger_event
    ON tbl_name FOR EACH ROW
[trigger_order]
trigger_body

trigger_time: { BEFORE | AFTER }

trigger_event: { INSERT | UPDATE | DELETE }

trigger_order: { FOLLOWS | PRECEDES } other_trigger_name
```

Un trigger are trei componente:

- *o instructiune de declarare* - aceasta specifică instrucțiunile SQL care activează declanșatorul (evenimentul declanșator)
- *o restrictie de declansare* - specifica condiția care trebui să fie adevarată pentru ca declanșatorul să fie activat (conditia)
- *actiunea declansatorului* care specifica blocul de instrucțiuni care trebuie executate (acțiunea).

Proprietățile trigerelor :

- Triggerele sunt caracterizate de un **moment de timp pentru acțiune** - astfel ele pot fi activate înaintea unui eveniment sau după un eveniment.

- Triggerele pot defini un **eveniment,** astfel ele pot fi activate în timpul unei inserări, sau actualizări sau a unei operații de ștergere.
- Triggerele sunt caracterizate de **o anumită granularitate** (câte tuple intervin în operație), astfel FOR EACH ROW definește faptul ca activarea triggerului va apărea pentru anumite linii din tabela și nu pentru tabela ca un întreg.
- Triggerele au în corpul lor o **frază SQL validă** (cu anumite limitări in utilizarea frazelor/clauzelor SQL – vezi manual MySQL).

IMPORTANT.În corpul frazei SQL din cadrul unui trigger, poate fi referită orice coloană a tabelei, însă se va specifica explicit coloana nouă **(NEW),** respectiv coloana veche **(OLD),** acestea fiind numite **variabile de tranziție,** astfel pentru o operatie de tip Insert, doar variabila NEW este permisă, pentru o operație Delete, doar variabilă OLD este legală, iar pentru o operație Update, ambele variabile pot fi utilizate, similar exemplului :

```
CREATE TRIGGER t21_au   BEFORE
       UPDATE ON t22
              FOR EACH ROW
BEGIN
       SET @old = OLD . s1;
       SET @new = NEW.s1;
END;//
```

Exemplu de creare de tabelă și trigger pentru inserare

```
CREATE TABLE t22 (s1 INTEGER)//
CREATE TRIGGER t22_bi
       BEFORE INSERT ON t22
       FOR EACH ROW
       BEGIN
       SET @x = 'Trigger was activated!';
       SET NEW.s1 = 55;
END;//
```

Exemplu de creare a unei constrângeri de tip Check**.**

```
CREATE TABLE t25 (s1 INT, s2 CHAR(5), PRIMARY KEY (s1), CHECK (LEFT(s2,1)='A'))
```

Specificând că inserările și actualizările sunt ilegale pentru acele valori pentru care cel mai din stânga caracter al coloanei s2 nu este A. In MySQL nu e posibilă această declarare la momentul creării tabelei, motiv pentru care este necesară rezolvarea folosind triggere, așa cum rezultă din exemplul următor :

```
CREATE TRIGGER t25_bi
BEFORE INSERT ON t25
FOR EACH ROW
IF LEFT(NEW.s2,1)<>'A' THEN SET NEW.s1=0; END IF;//
CREATE TRIGGER t25_bu
BEFORE UPDATE ON t25
FOR EACH ROW
IF LEFT(NEW.s2,1)<>'A' THEN SET NEW.s1=0; END IF;//
```

În MySQL triggerele pot fi create doar ca *posesor al bazei de date (owner).* Atunci când se adaugă un trigger pentru o coloană, linie sau tabel se schimbă uneori și modul de accesare și modul cum interacționează alte obiecte cu acesta. Triggerele acționează asupra unei tabele, astfel nu pot fi create *două triggere pentru același eveniment și asupra aceleiași tabele***.**

Exemple tipice de utilizare a triggerelor sunt :

- Cascadarea operațiilor de *modificare de la tabela curentă către alte tabele din baza de date*
- Anularea modificărilor care ar duce la *violarea integrității referențiale* de cheie străină (mijloace alternative mai sofisticate decât declarațiile de tip cheie străină)
- Exprimarea unor constrângeri **de tip Check** în mod programatic (MySQL nu implementează acest tip de constrângere specifică BD Oracle și MS-SQL)

La crearea și utilizarea obiectelor de tip trigger se presupune respectarea condițiilor :

- triggerele nu pot fi create pentru tabele temporare,
- triggerele trebuie să fie create numai pentru tabele din baza de date curentă
- la eliminarea unui tabel, toate obiectele trigger asociate cu acest tabel sunt eliminate automat împreună cu tabelul.

Triggerul și operația care îl declanșează sunt considerate ca un tot unitar (tranzacție) ceea ce înseamnă că dacă execuția triggerului eșuează, se va anula și operația care a declanșat triggerul.

Orice constrângere referitoare la tabela căreia îi este atașat un trigger este verificată înainte de execuția triggerului , de acceea orice operație care ar viola o constrângere de cheie primară sau străină este respinsă și implicit triggerul asociat nu se va activa, din acest motiv dacă se dorește implementarea unei constrângeri de integritate referențială în varianta cascadată prin folosirea unui trigger, este necesar să se șteargă sau să se deacvtiveze declarația de cheie straină pe care o înlocuiește.

O serie de instrucțiuni SQL sunt interzise în corpul unui trigger ,de exemplu CREATE/ALTER/DROP pe diverse obiecte din baza de date cum ar fi tabela, vederea, triggerul, indexul, procedura). Un trigger nu poate folosi apeluri de rutine stocate care întorc valori utilizatorului (pot fi folosite doar proceduri ale căror rezultate pot fi transmise prin parametri de tip OUT sau INOUT) sau care folosesc SQL dinamic. Totodată nu pot fi folosite operaţii care încep sau termină o tranzacţie (COMMIT, ROLLBACK).

Instrucțiunile :
DROP TRIGGER <*trigger_nume*> și respectiv
ALTER TRIGGER <*trigger_nume*>, permit ștergerea , respectiv modificarea triggerului.

Resurse suplimentare: http://www.mysqltutorial.org/mysql-triggers.aspx
https://www.techonthenet.com/mysql/triggers/index.php

Evenimente

Evenimentele sunt sarcini planificate ce sunt executate la un moment dat sau periodic în cazul în care planificatorul de execuţii este pornit. Planificatorul execuţiilor poate avea starea: pornit, oprit sau dezactivat. De regulă, acestea sunt folosite pentru realizarea unor operaţii de întreţinere fie asupra anumitor tabele din baza de date, fie asupra sistemului de baze de date însuşi.

```
CREATE
    [DEFINER = user]
    EVENT
    [IF NOT EXISTS]
event_name
    ON SCHEDULE schedule
    [ON COMPLETION [NOT] PRESERVE]
    [ENABLE | DISABLE | DISABLE ON SLAVE]
    [COMMENT 'string']
    DO event_body;
```

```
schedule:
    AT          [+ INTERVAL
timestampinterval]                    ...
  | EVERY interval
    [STARTS timestamp [+ INTERVAL interval] ...]
    [ENDS                                    timestamp
 interval:          [+ INTERVAL interval] ...]  quantity {YEAR |
QUARTER | MONTH |                               DAY | HOUR | MINUTE |
WEEK | SECOND | YEAR_MONTH | DAY_HOUR | DAY_MINUTE |
DAY_SECOND | HOUR_MINUTE | HOUR_SECOND | MINUTE_SECOND}
```

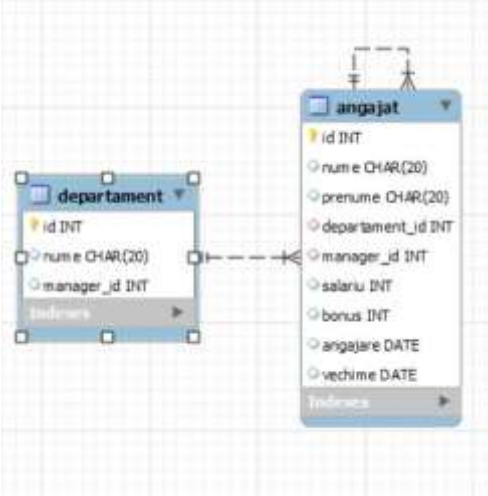
Mai multe detalii pot fi consultate in tutorial:
http://www.mysqltutorial.org/mysql-triggers/working-mysql-scheduled-event/

O serie de alte exemple de utilizare a mecanismelor prezentate pentru crearea de elemente programatice in MySQL pot fi studiate la :https://www.geeksengine.com/database/

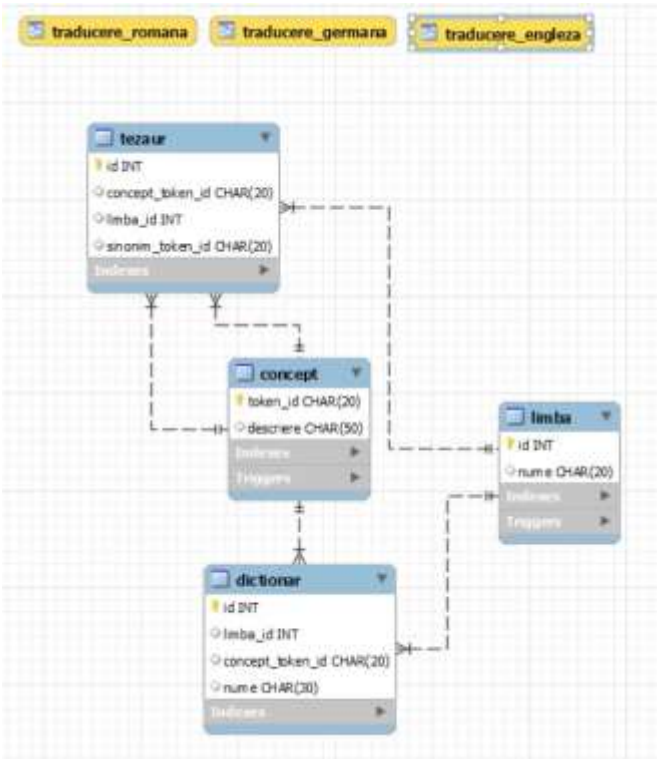
III. PROBLEME REZOLVATE

3.1. Pentru baza de date BD_personal al cărui model este reprezentat ,se vor studia și testa procedurile stocate implementate. Se vor crea situații de test adecvate astfel încât toate aceste elemente programate în baza de date să poată fi analizate

.



3.2. Aplicația gestionează un dicționar și tezaur Multilingv Entitățile cu care se operează sunt: limba, concept, tezaur și dictionar; Pentru fiecare entitate se creează un tabel. In tabela 'limba' sunt grupate limbile în care sunt disponibile traduceri; tabela 'concept' cuprinde toate conceptele (sau noțiunile) pentru care există traduceri în diferite limbi; în tabela 'tezaur' sunt grupate, în jurul unui concept dintro anume limbă, acele noțiuni care sunt, într-un grad mai mare sau mai mic, sinonime cu acesta; cum aria semantică a unui termen poate diferi de la o limbă la alta, sinonimele nu sunt universal valabile, ci numai pentru o anume limba.



Modelul bazei de date a fost realizat astfel încât ,în tabela ‘dictionar’ sunt furnizate traduceri pentru toate conceptele, în toate limbile disponibile. E nevoie să se adauge automat, la adăugarea unei limbi sau a unui nou concept, înregistrări în tabela dictionar’; acest proces va fi automatizat folosind două triggere și anume :

1. **Trigger AFTER INSERT pe tabela concept**. Acest trigger se utilizează pentru a adăuga înregistrări in tabela dictionar la adaugarea unui nou concept, cate o inregistrare pentru fiecare limba definită în tabela ‘limba’.Procedura implementează cursorul pentru tabela ‘limba’;se parcurg toate înregistrările tabelei, și pentru fiecare înregistrare se adaugă o nouă înregistrare în tabela ‘dictionar’. In campul ‘nume’ unde se adaugă traducerea propriu-zisă, se introduce un cod cu semnificația ‘traducere inexistenta’. Trigger-ul apelează procedura CURSOR_LIMBA.

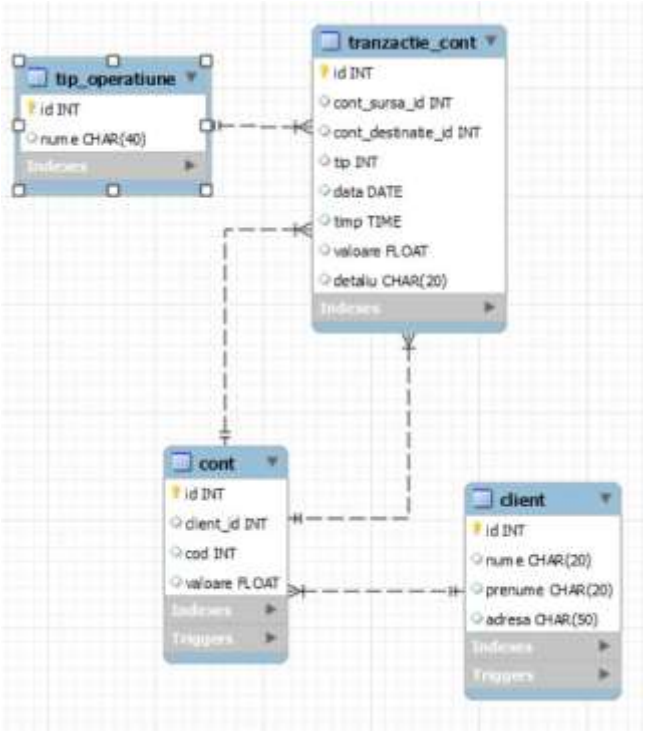
2. **Trigger AFTER INSERT pe tabela limba**. Acest trigger se utilizează pentru a adăuga înregistrări în tabela ‘dictionar’ la adăugarea unei noi limbi, câte o înregistrare pentru fiecare concept definit în tabela ‘concept’. Procedura implementează cursorul pentru tabela ‘concept’; se parcurg toate înregistrările tabelei, și pentru fiecare înregistrare se adaugă o nouă înregistrare în tabela ‘dictionar’. In câmpul ‘nume’ unde se adaugă traducerea propriu-zisă se introduce un cod cu semnificația ‘traducere inexistenta’. Trigger-ul apelează procedura CURSOR_CONCEPT.

3.3. Fie aplicația ce gestionează conturile clienților unei bănci. Pentru acestea, operațiunile permise sunt: Deschidere cont, Inchidere cont, Transfer intre conturi, Depunere ghiseu, Depunere ATM, Extragere ghiseu, Extragere ATM, Balanta cont, Istoric tranzactii.

Informațiile despre clienți sunt stocate în tabela **client**; informațiile despre conturile clientilor sunt stocate in tabela **cont**.Tipurile de operatiuni sunt stocate in tabela (nomenclator) **tip_operatiune**; tabela **tranzactie_cont** stocheaza informatiile despre operațiunile executate, inclusiv despre operatiuni de interogare precum ‘Istoric tranzactii’;

Operațiunile care implică transferuri intre conturi sunt executate in interiorul unor tranzactii, pentru corectitidine. O tranzacție este o unitate de procesare ce conține cod SQL și care se realizează în întregime sau deloc .

Logica operatiunilor este implementată folosind: triggere - pentru creare cont la crearea unui client, respectiv proceduri stocate ,pentru diferite operatiuni;



Se utilizează triggere pentru unele operațiuni care trebuie înregistrate automat, după cum urmează:

1. La creearea unui client, i se creeaza automat și un cont. Clienții sunt creati prin adăugarea unei noi inregistrari in tabela de client; trigger-ul este 'insert_cont_client', declanseaza dupa INSERT in tabela client si are ca efect inserarea unei noi inregistrari in tabela cont.

2. La adaugarea unei înregistrări în tabela cont (la creearea unui cont), se adaugă o înregistrare de tip 'Deschidere cont' in tabela 'tranzactie_cont', tabela care înregistrează toate tranzactiile/operațiunile care au loc. S-a preferat ca alte tipuri de modificări ale 'cont' si 'tranzactie_cont' să nu utilizeze triggere pentru modificări corespunzatoare în tabela complementară.

Procedurile implementate:

1. **Procedura alimentare cont** -procedura primește ca parametrii codul contului destinație și valoarea depunerii; tipul operațiunii este 'Depunere ghiseu'; tranzacția constă în două operațiuni complementare:
   înregistrarea unei tranzacții în tabela 'tranzactie_cont' și cresterea soldului contului accesat cu aceeași valoare (tabela 'cont'). Tranzacția este finalizată numai dacă se operează cu un cont destinație existent; altfel, se face rollback (tranzacția este anulata).
2. **Procedura extragere ATM** -procedura primește ca parametru codul contului debitat și valoarea extragerii; tranzacția este fie finalizată, cu transferul efectuat (contul există și suma există în cont), fie tranzacția este finalizată, iar transferul eșuat (cont existent dar suma insuficientă), fie tranzacția nu este finalizată (cont inexistent, se face rollback).

3. **Procedura istoric tranzactii** -procedura primește ca parametrii codul contului debitat, data inițială, data finală și opțiunea pentru a stoca operațiunea curentă (vizualizare istoric tranzactii) în tabela de tranzacții; opțiunea este necesară pentru că e posibil ca această operațiune să fie taxată în anumite condiții si atunci trebuie să fie inregistrată.

3.3. Se va parcurge tutorialul de la adresa
https://www.ntu.edu.sg/home/ehchua/programming/sql/SampleDatabases.html#zz-2.3
Bazele de date sunt masiv populate și conțin obiecte de tip proceduri stocate și triggere

IV. Exerciții propuse

4.1.Se consideră următoarele tabele:

Tabela **angajati** avand câmpurile: id tip INTEGER primary key, NUME de tip VARCHAR (NOT NULL), salar_neg de tip numar cu 3 cifre si 2 zecimale, data _angajarii de tip DATE,data _concedierii de tip DATE, nr_copii de tip INT (NOT NULL). Coloana salar_neg contine salariul brut negociat pe zi.

Tabela **salarii** avand câmpurile: ID intreg nenul, nr_zile intreg nenul, brut de tip numar cu 3 cifre si 2 zecimale, deducere de tip numar cu 3 cifre si 2 zecimale, deducere_copii de tip numar cu 3 cifre si 2 zecimale, impozit de tip numar cu 3 cifre si 2 zecimale, net de tip numar cu 3 cifre si 2 zecimale.

Tabela **deduceri** avand campurile: brut_min de tip numar cu 3 cifre si 2 zecimale, brut_max de tip numar cu 3 cifre si 2 zecimale, deducere de tip numar cu 3 cifre si 2 zecimale.

Tabela **deduceri_copii** avand campurile: nr_copii, deducere_copii de tip numar cu 3 cifre si 2 zecimale.

Se cunosc următoarele:

a. Deducerile se acordă în funcție de venitul brut lunar, conform următorului tabel:

Venit brut minim	Venit brut maxim	Deducere
0	300	100
300.01	600	200
600.01	800	300
800.01	999.99	400

Limitele tranșelor și respectiv cuantumurile acestor deduceri se citesc din tabela deduceri.

b. Angajații care au copii beneficiază de deduceri suplimentare pentru fiecare copil, conform următorului tabel:

Nr Copii	Deducere
1	100
2	170

3	220
4	260

Cuantumurile acestor deduceri se citesc din tabela deduceri_copii.

c. Impozitul datorat statului se calculează cu formula:
impozit = (venit brut - deducere – deducere pentru copii) * 0.16
d. Venitul net este obținut cu ajutorul formulei: venit brut ,daca impozitul <= 0
venit net =venit brut –impozit, daca impozitul > 0

Se cere:

1. Să se creeze cele 4 tabele și să se populeze cu informații tabelele angajati, deduceri și deduceri_copii.
2. Să se implementeze procedura/trigger care la orice adăugare de noi linii in tabela SALARII sau orice actualizare a coloanei nr_zile va actualiza (recalcula) informația din coloanele: brut, deducere, deducere_copii, impozit si respectiv net.
3. Să se implementeze o procedură/ trigger, care la orice modificare a tabelei DEDUCERI va actualiza (recalcula) informația din coloanele: deducere, impozit și respectiv net.
4. Să se implementeze o procedură/trigger, care la orice modificare a tabelei DEDUCERI_COPII va actualiza (recalcula) informația din coloanele: deducere_copii, impozit si respectiv net. Se presupune ca in tabela DEDUCERI_COPII se fac modificări doar asupra coloanei care cuprinde cuantumul deducerii nu și asupra celei care conține numărul de copii.
5. Să se implementeze o procedura/trigger, care la orice înregistrare a unei concedieri (introducerea datei concedierii in tabela ANGAJATI) va șterge toate liniile din tabela SALARII referitoare la angajații concediați.

4.2. Uneori, după proiectarea unei baze de date, este necesară actualizarea structurii acesteia; din păcate însă nu toate actualizările efectuate se dovedesc a fi corecte, si modificările trebuie să fie anulate. În acest context, se cere dezvoltarea unui mecanism de versionare care să faciliteze tranziţia de la o versiune a bazei de date la alta.
Scrieţi un script SQL care va:

- modifica tipul unei coloane;
- adauga o constrângere de “valoare implicită” pentru un câmp;
- creea/şterge o tabelă;
- adăuga un câmp nou;
- creea/şterge o constrângere de cheie străină.

Pentru fiecare dintre scripturile de mai sus scrieţi/generaţi un alt script care implementează inversul operaţiei. De asemenea, creaţi o nouă tabelă care să memoreze versiunea structurii bazei de date, presupunând că acestă versiune este pur şi simplu un număr întreg.
Creaţi pentru fiecare din scripturi câte o procedură stocată şi asiguraţi-vă ca numele acestora să fie simplu şi clar.
De asemenea, scrieţi o altă procedură stocată ce primeşte ca parametru un număr de versiune şi aduce baza de date la versiunea respectivă.