SGBD : Programmation et administration des bases de données [M2106]

Hocine ABIR

17 février 2014

IUT Villetaneuse
E-mail: abir@iutv.univ-paris13.fr

TABLE DES MATIÈRES

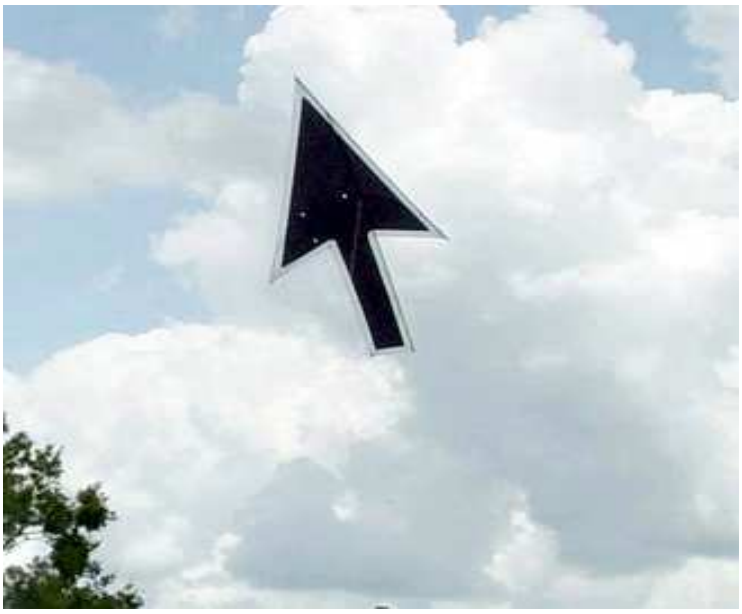
CHAPITRE

TROIS

Procédures Stockées (3-PL/pgSQL Curseurs/Triggers)

Première partie

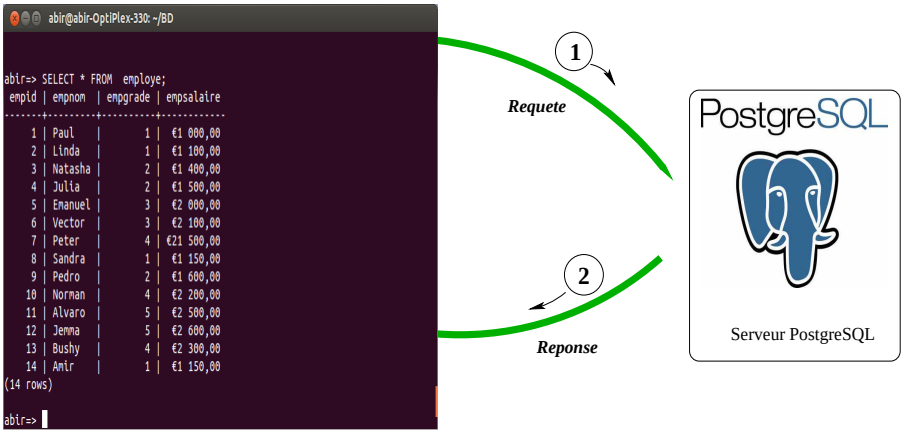
Curseurs



3.1 Curseurs en SQL

Deux modes de fonctionnement :

3.1.1 Cycle simple



Terminal

1. Interrogation : Envoi de la requête.
2. Réponse : Reception du résultat.

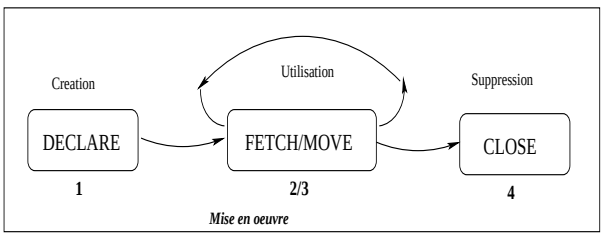
```
=> SELECT * FROM  employe;
 empid | empnom   | empgrade | empsalaire
-------+----------+----------+------------
     1 | Paul     |        1 |  €1 000,00
     2 | Linda    |        1 |  €1 100,00
     3 | Natasha  |        2 |  €1 400,00
     4 | Julia    |        2 |  €1 500,00
     5 | Emanuel  |        3 |  €2 000,00
     6 | Vector   |        3 |  €2 100,00
     7 | Peter    |        4 | €21 500,00
     8 | Sandra   |        1 |  €1 150,00
     9 | Pedro    |        2 |  €1 600,00
    10 | Norman   |        4 |  €2 200,00
    11 | Alvaro   |        5 |  €2 500,00
    12 | Jemma    |        5 |  €2 600,00
    13 | Bushy    |        4 |  €2 300,00
    14 | Amir     |        1 |  €1 150,00
(14 rows)
```

3.1.2 Cycle étendu

Un curseur est un mécanisme qui permet de décrire les résultats (ensemble de tuples) d'une requête SQL `SELECT`. Cette description permet en particulier de parcourir :

- itérativement l'ensemble des tuples ou partiellement,
- dans un ordre arbitraire l'ensemble des tuples.

La mise en oeuvre du mécanisme de curseur nécessite 3 étapes.



1. Création du curseur.

```
=> DECLARE mon_curseur CURSOR WITH HOLD FOR
->    SELECT * FROM EMPLOYE;
DECLARE CURSOR

=> select name,statement from pg_cursors;
    name     |                statement
-------------+-------------------------------------------
 mon_curseur | DECLARE mon_curseur CURSOR WITH HOLD FOR +
             |    SELECT * FROM EMPLOYE;
(1 row)
```

2. Utilisation du curseur.

```
=> FETCH  FROM mon_curseur;
 empid | empnom | empgrade | empsalaire
-------+--------+----------+------------
     1 | Paul   |        1 |  €1 000,00
(1 row)

=> MOVE 2 IN  mon_curseur;
MOVE 2

=> FETCH   FROM mon_curseur;
 empid | empnom | empgrade | empsalaire
-------+--------+----------+------------
     4 | Julia  |        2 |  €1 500,00
(1 row)

=> FETCH 5  FROM mon_curseur;
 empid | empnom  | empgrade | empsalaire
-------+---------+----------+------------
     5 | Emanuel |        3 |  €2 000,00
     6 | Vector  |        3 |  €2 100,00
     7 | Peter   |        4 | €21 500,00
     8 | Sandra  |        1 |  €1 150,00
     9 | Pedro   |        2 |  €1 600,00
(5 rows)
```

3. Suppression du curseur.

```
=> close  mon_curseur;
CLOSE CURSOR
```

3.1.3 Exemple d'Illustration

```
CREATE TABLE departements (
    dept_num    CHAR(4) ,
    dept_nom  VARCHAR(40) NOT NULL,
    PRIMARY KEY (dept_num),
    UNIQUE      (dept_nom)
);
```

```
=> select * from departements;
 dept_num |      dept_nom
----------+--------------------
 d001     | Marketing
 d002     | Finances
 d003     | Ressources Humaines
 d004     | Production
 d005     | Developement
 d006     | Gestion qualite
 d007     | Ventes
 d008     | Recherche
 d009     | Service Client
(9 rows)
```

```
=> BEGIN;
BEGIN
=> DECLARE dept SCROLL CURSOR FOR
 SELECT * FROM departements;
DECLARE CURSOR

=> FETCH FORWARD 4 FROM dept;
 dept_num |      dept_nom
----------+--------------------
 d001     | Marketing
 d002     | Finances
 d003     | Ressources Humaines
 d004     | Production
(4 rows)

=> FETCH BACKWARD 2 FROM dept;
 dept_num |      dept_nom
----------+--------------------
 d003     | Ressources Humaines
 d002     | Finances
(2 rows)

=> FETCH NEXT FROM dept;
 dept_num |      dept_nom
----------+--------------------
 d003     | Ressources Humaines
(1 row)

=> FETCH PRIOR FROM dept;
 dept_num | dept_nom
----------+----------
 d002     | Finances
(1 row)

=> FETCH FIRST FROM dept;
 dept_num | dept_nom
----------+-----------
 d001     | Marketing
(1 row)

=> FETCH LAST FROM dept;
 dept_num |    dept_nom
----------+----------------
 d009     | Service Client
(1 row)

=> FETCH NEXT FROM dept;
 dept_num | dept_nom
----------+----------
(0 rows)
```

```
=> FETCH FIRST FROM dept;
 dept_num | dept_nom
----------+-----------
 d001     | Marketing
(1 row)

=> FETCH PRIOR FROM dept;
 dept_num | dept_nom
----------+----------
(0 rows)

=> FETCH ALL FROM dept;
 dept_num |      dept_nom
----------+--------------------
 d001     | Marketing
 d002     | Finances
 d003     | Ressources Humaines
 d004     | Production
 d005     | Developement
 d006     | Gestion qualite
 d007     | Ventes
 d008     | Recherche
 d009     | Service Client
(9 rows)

=> FETCH FIRST FROM dept;
 dept_num | dept_nom
----------+-----------
 d001     | Marketing
(1 row)

=> update departements set dept_nom='Commercial'
 > WHERE CURRENT OF dept;
UPDATE 1
=> FETCH FIRST FROM dept;
 dept_num | dept_nom
----------+-----------
 d001     | Marketing
(1 row)

=> FETCH ALL FROM dept;
 dept_num |      dept_nom
----------+--------------------
 d002     | Finances
 d003     | Ressources Humaines
 d004     | Production
 d005     | Developement
 d006     | Gestion qualite
 d007     | Ventes
 d008     | Recherche
 d009     | Service Client
(8 rows)
```

```
=> MOVE LAST FROM dept;
MOVE 1
=> DELETE FROM departements
 > WHERE CURRENT OF dept;
DELETE 1
=> close dept;
CLOSE CURSOR

=> select * from departements;
 dept_num |      dept_nom
----------+--------------------
 d002     | Finances
 d003     | Ressources Humaines
 d004     | Production
 d005     | Developement
 d006     | Gestion qualite
 d007     | Ventes
 d008     | Recherche
 d001     | Commercial
(8 rows)

=> rollback;
ROLLBACK
```

3.1.4 DECLARE / CLOSE

3.1.5 Lecture FETCH

La commande `FETCH` permet de lire des tuples d'un curseur.

```
FETCH [ direction ] [ count ] { IN | FROM } cursor_variable_name
```

où :
- `direction` définie le sens du parcours du curseur, il peu être :
  - `FORWARD` : en avant (option par défaut).
  - `BACKWARD` : en arrière. `FORWARD -1` est équivalent à `BACKWARD 1`.
- `count` définie le nombre de tuples à lire, il peu être :
  - `#` : entier signé désignant le nombre de tuple à lire à partir de la position courante.
  - `NEXT` : équivalent à `+1` (option par défaut)
  - `PRIOR` : équivalent à `-1`
  - `ALL` : tout les tuples à partir de la position courante.

3.1.6 Déplacement MOVE

La commande `MOVE` permet de positionner (ou déplacer) le curseur sans lecture.

```
MOVE [ direction ] [ count ] { IN | FROM } cursor_variable_name
```

3.1.7 Exemple

```
-- Set up and use a cursor:
BEGIN WORK;
DECLARE agent CURSOR FOR SELECT * FROM employe;

-- Fetch first 5 rows in the cursor agent:
FETCH FORWARD 5 IN agent;
 em_nom | em_salaire
--------+------------
 Paul   |    3000.00
 Martin |    2800.00
 Joseph |    2500.00
 Sylvia |    2300.00
 Marc   |    2200.00
(5 lignes)

-- Fetch previous row:
FETCH BACKWARD 1 IN agent;
 em_nom | em_salaire
--------+------------
 Sylvia |    2300.00
(1 ligne)

CLOSE agent;

-- Skip first 5 rows:
MOVE FORWARD 5 IN agent;
MOVE 5
-- Fetch 6th row in the cursor agent:
FETCH  IN agent;
 em_nom  | em_salaire
---------+------------
 Josette |    2100.00
(1 ligne)
CLOSE agent;
COMMIT WORK;
```

3.2 Curseurs en PL/pgSQL

3.2.1 Introduction

En PL/pgSQL un curseur est une variable de type spécifique `refcursor`. Un curseur (variable curseur) peut être déclarer de trois facons :

1. non lié : définie pour une requête arbitraire c'est à dire non spécifiée lors de la déclaration du curseur :

```
1 nom_variable_curseur REFCURSOR;
```

2. totalement lié : définie pour une requête particulière qui peut être :

```
1 nom_variable_curseur CURSOR FOR requete_select;
```

3. partiellement lié (ou paramétré) :

```
1 nom_variable_curseur (arguments ) CURSOR FOR requete_select_parametre;
```

où `arguments` est une suite de couples `parameter_name datatype` séparés par des virgules.

3.2.2 Curseur non lié

```
1  CREATE function unbounded
2        (in grade int,out count int) AS
3  $$
4   DECLARE
5      emp_cursor REFCURSOR;
6      nom employe.empnom%TYPE;
7       salaire  employe.empsalaire%TYPE;
8
9   BEGIN
10     count :=0;
11     OPEN emp_cursor FOR
12     SELECT empnom, empsalaire
13           FROM employe
14          WHERE empgrade = grade;
15
16     LOOP
17       FETCH emp_cursor INTO nom, salaire;
18       EXIT WHEN NOT FOUND;
19       count:=count+1;
20       raise Notice E'%\t%',nom,salaire;
21     END LOOP;
22     CLOSE emp_cursor;
23 END;
24 $$ language plpgsql;
```

```
abir=> select * from unbounded(2);
NOTICE:  Natasha    €1 400,00
NOTICE:  Julia      €1 500,00
NOTICE:  Pedro      €1 600,00
 count
-------
     3
(1 row)

abir=>
abir=>
abir=> select * from unbounded(1);
NOTICE:  Paul     €1 000,00
NOTICE:  Linda    €1 100,00
NOTICE:  Sandra   €1 150,00
NOTICE:  Amir     €1 150,00
 count
-------
     4
(1 row)
```

3.2.3 Curseur totalement lié

```
CREATE function bounded
      (in grade int,out count int) AS
$$
 DECLARE
    emp_cursor CURSOR FOR
    SELECT empnom, empsalaire
          FROM employe
        WHERE empgrade = grade;
    nom employe.empnom%TYPE;
     salaire  employe.empsalaire%TYPE;

 BEGIN
    count :=0;
    OPEN emp_cursor;
    LOOP
      FETCH emp_cursor INTO nom, salaire;
      EXIT WHEN NOT FOUND;
      count:=count+1;
      raise Notice E'%\t%',nom,salaire;
    END LOOP;
    CLOSE emp_cursor;
END;
$$ language plpgsql;
```

```
=> select * from bounded(1);
NOTICE:  Paul   €1 000,00
NOTICE:  Linda  €1 100,00
NOTICE:  Sandra €1 150,00
NOTICE:  Amir   €1 150,00
 count
-------
     4
(1 row)

r=> select * from bounded(2);
NOTICE:  Natasha  €1 400,00
NOTICE:  Julia    €1 500,00
NOTICE:  Pedro    €1 600,00
 count
-------
     3
(1 row)
```

3.2.4 Curseur partiellement lié

```
CREATE function parbounded
      (in grade int,out count int) AS
$$
 DECLARE
    emp_cursor CURSOR (class int) IS
    SELECT empnom, empsalaire
          FROM employe
        WHERE empgrade = class;
    nom employe.empnom%TYPE;
     salaire  employe.empsalaire%TYPE;

 BEGIN
    count :=0;
    OPEN emp_cursor (grade);
    LOOP
      FETCH emp_cursor INTO nom, salaire;
      EXIT WHEN NOT FOUND;
      count:=count+1;
      raise Notice E'%\t%',nom,salaire;
    END LOOP;
    CLOSE emp_cursor;
END;
$$ language plpgsql;
```

```
=> select * from parbounded(1);
NOTICE:  Paul   €1 000,00
NOTICE:  Linda  €1 100,00
NOTICE:  Sandra €1 150,00
NOTICE:  Amir   €1 150,00
 count
-------
     4
(1 row)

=> select * from parbounded(4);
NOTICE:  Peter  €21 500,00
NOTICE:  Norman €2 200,00
NOTICE:  Bushy  €2 300,00
 count
-------
     3
(1 row)
```

3.3 Curseur Implicite

```
1  CREATE function implicit
2        (in grade int,out count int) AS
3  $$
4   DECLARE
5      nom employe.empnom%TYPE;
6       salaire  employe.empsalaire%TYPE;
7      cur text;
8      qry text;
9
10  BEGIN
11     count :=0;
12     FOR nom,salaire IN
13      SELECT empnom, empsalaire
14           FROM employe
15         WHERE empgrade = grade
16     LOOP
17       if count=0 Then
18             select name,statement
19              into cur,qry from pg_cursors;
20       END IF;
21       count:=count+1;
22       raise Notice E'%\t%',nom,salaire;
23     END LOOP;
24
25     raise INFO E'Nom : % \n\tRequete :\n\t %\n',cur,qry;
26 END;
27 $$ language plpgsql;
```

```
=> select * from implicit(1);
NOTICE:  Paul €1 000,00
NOTICE:  Linda €1 100,00
NOTICE:  Sandra €1 150,00
NOTICE:  Amir €1 150,00
INFO:  Nom : <unnamed portal 9>
Requete :
 SELECT empnom, empsalaire
           FROM employe
          WHERE empgrade = grade

 count
-------
     4
(1 row)
```

3.4 Curseur et Requête Dynamique

```
1  CREATE function dynamic
2        (in grade int,out cout int) AS
3  $$
4   DECLARE
5      emp_cursor REFCURSOR;
6      nom employe.empnom%TYPE;
7       salaire  employe.empsalaire%TYPE;
8      qry text;
9
10  BEGIN
11     cout :=0;
12     qry:='SELECT empnom, empsalaire
13            FROM employe
14            WHERE empgrade=$1';
15     raise Info '%',qry;
16     OPEN emp_cursor FOR EXECUTE qry USING grade;
17     LOOP
18       FETCH emp_cursor INTO nom, salaire;
19       EXIT WHEN NOT FOUND;
20       cout:=cout+1;
21       raise Notice E'%\t%',nom,salaire;
22     END LOOP;
23 END;
24 $$ language plpgsql;
```

```
=> select * from dynamic(1);
INFO:  SELECT empnom, empsalaire
            FROM employe
            WHERE empgrade=$1
NOTICE:  Paul €1 000,00
NOTICE:  Linda €1 100,00
NOTICE:  Sandra €1 150,00
NOTICE:  Amir €1 150,00
 cout
------
    4
(1 row)

=> select * from dynamic(2);
INFO:  SELECT empnom, empsalaire
            FROM employe
            WHERE empgrade=$1
NOTICE:  Natasha €1 400,00
NOTICE:  Julia €1 500,00
NOTICE:  Pedro €1 600,00
 cout
------
    3
(1 row)
```

3.5 Curseur en Argument

Une fonction PL/pgSQL peut retourner des curseurs. La valeur de retour de type `refcursor`. Le nom du curseur retourné par la fonction peut être transmis en argument ou généré automatiquement.

3.5.1 Curseur nommé

```
1  CREATE FUNCTION refspec(INOUT refcursor, grade int)
2  AS $$
3   BEGIN
4        OPEN $1 FOR
5          SELECT empnom, empsalaire
6            FROM employe
7          WHERE empgrade = grade;
8   END;
9  $$ LANGUAGE 'plpgsql';
10
11
12 CREATE function use_refsec
13        (in grade int,out count int) AS
14 $$
15  DECLARE
16     emp_cursor REFCURSOR;
17     nom employe.empnom%TYPE;
18      salaire  employe.empsalaire%TYPE;
19
20  BEGIN
21     count :=0;
22     SELECT refspec('emp_cursor',grade)
23            INTO emp_cursor;
24     LOOP
25       FETCH emp_cursor INTO nom, salaire;
26       EXIT WHEN NOT FOUND;
27       count:=count+1;
28       raise Notice E'%\t%',nom,salaire;
29     END LOOP;
30     CLOSE emp_cursor;
31 END;
32 $$ language plpgsql;
```

```
=> select use_refspec(3);
NOTICE:  Emanuel   €2 000,00
NOTICE:  Vector    €2 100,00
 use_cursor
------------
          2
(1 row)

=> select use_refsepc(4);
NOTICE:  Peter  €21 500,00
NOTICE:  Norman €2 200,00
NOTICE:  Bushy  €2 300,00
 use_cursor
------------
          3
(1 row)
```

3.5.2 Curseur non-nommé

```
1  CREATE FUNCTION refauto(OUT refcursor, grade int)
2  AS $$
3   BEGIN
4        OPEN $1 FOR
5          SELECT empnom, empsalaire
6            FROM employe
7          WHERE empgrade = grade;
8   END;
9  $$ LANGUAGE 'plpgsql';
10
11
12 CREATE function use_refauto
13        (in grade int,out count int) AS
14 $$
15  DECLARE
16     emp_cursor REFCURSOR;
17     nom employe.empnom%TYPE;
18      salaire  employe.empsalaire%TYPE;
19
20  BEGIN
21     count :=0;
22     SELECT refauto(grade)
23            INTO emp_cursor;
24     raise  info 'Curseur -> % ',emp_cursor;
25     LOOP
26       FETCH emp_cursor INTO nom, salaire;
27       EXIT WHEN NOT FOUND;
28       count:=count+1;
29       raise Notice E'%\t%',nom,salaire;
30     END LOOP;
31     CLOSE emp_cursor;
32 END;
33 $$ language plpgsql;
```

```
abir=> select use_refauto(3);
INFO:  Curseur -> <unnamed portal 2>
NOTICE:  Emanuel €2 000,00
NOTICE:  Vector €2 100,00
 use_refauto
-------------
           2
(1 row)

abir=> select use_refauto(5);
INFO:  Curseur -> <unnamed portal 3>
NOTICE:  Alvaro €2 500,00
NOTICE:  Jemma €2 600,00
 use_refauto
-------------
           2
(1 row)
```

Deuxième partie

Triggers

3.6 Procédure trigger

Une procédure trigger est une procédure stockée qui est activée par un événement c'est à dire que la procédure est exécutée (automatiquement) à chaque fois qu'un événement qui lui est associé se produit.

Formellement, une procédure trigger est une règle de la forme :

$$Evenement \longrightarrow Action$$

où

- *Action* :
  l'*Action* est définie par une fonction ayant une signature spécifique :
  ```
  function_name () RETURNS TRIGGER ...
  ```
  - la fonction associée à un trigger n'a pas d'arguments
  - et sa valeur de retour est de type particulier : `TRIGGER`.
- *Evenement* :
  l'*Evenement* est spécifié par un déclencheur (trigger). Un trigger est un mécanisme qui permet d'associer une action à un évenement. L'événement peut être soit `INSERT`, `UPDATE` ou `DELETE`.

3.6.1 Création d'une procédure trigger

```
CREATE FUNCTION function_identifier ()
  RETURNS TRIGGER AS
$$
  DECLARE
    -- declarations;
  BEGIN
    -- statements;
  END;
$$ LANGUAGE plpgsql;
```

Quand une procédure trigger est appelé par le "gestionnaire de triggers", aucun argument normal ne lui est transmis.

Les arguments d'une fonction trigger sont transmis dans une structure particulière comportant :

Trigger		
Champ	Type	Description
Evenement		
TG_NAME	name	le nom du trigger activé.
TG_OP	text	Indique quel évenement a activé le trigger. vaut soit INSERT, UPDATE ou DELETE
TG_RELNAME	name	le nom de la table indiquant l'objet de l'evénement.
TG_RELID	oid	l'identifiant de la table.
Comportement		
TG_LEVEL	text	Donne le Niveau (tuple ou commande) du trigger vaut soit ROW soit STATEMENT.
TG_WHEN	text	Donne le Moment d'action (avant ou après) du trigger vaut soit BEFORE soit AFTER.
Tuples		
NEW	record	Variable tuple contenant le nouveau tuple pour les opérations INSERT et UPDATE dans les trigger de niveau tuple .
OLD	record	Variable tuple contenant le nouveau tuple pour les opérations DELETE et UPDATE dans les trigger de niveau tuple .
Ligne Commande		
TG_NARGS	integer	le nombre d'arguments donnés à la fonction trigger dans la commande CREATE TRIGGER.
TG_ARGV[]	text	les arguments de la commande CREATE TRIGGER Les indices invalides (inférieurs à 0 ou supérieurs ou égaux à TG_NARGS) auront une valeur nulle.

Dans les procédures trigger PLpgSQL, le tuple pour lequel la trigger est déclenché est accessible par des variables SQL.
- NEW : est l'image du tuple à insérer ou après mis à jour.
- OLD : est l'image du tuple à supprimer ou avant mise à jour.

Chaque attribut de l'image du tuple est accessible en utilisant la notation pointé de

la forme `NEW.champ` ou `OLD.champ` où `champ` est le nom d'un attribut de la table. La valeur d'un attribut peut être modifié par une expression d'affection.

3.7 Création d'un trigger

Un trigger est créé par la commande :

```
1 CREATE TRIGGER trigger
2   [ BEFORE | AFTER ]                    -- Quand
3   [ INSERT | DELETE | UPDATE [ OR ... ] ] -- Evenement
4   ON relation                           -- Objet
5   FOR EACH [ ROW | STATEMENT ]          -- Mode
6     EXECUTE PROCEDURE procedure (argv);
```

le trigger peut agir :

1. Avant l'événement (`BEFORE`) :
   Il peut alors :
   - modifier le tuple inséré ou mis à jour,
   - annuler l'opération.
2. Après l'événement (`AFTER`) :
   Il **ne peut pas** :
   - modifier le tuple inserré ou mis à jour
   - annuler l'opération.
3. Au niveau tuple : `ROW`
4. Au niveau requête : `STATEMENT`

3.7.1 Valeur de retour d'une procédure trigger

La valeur de retour d'une fonction trigger consiste à la fois en
- un tuple (données)
- une donnée de contrôle

Les valeurs de retour possibles sont `OLD`,`NEW` ou `NULL`.

Evénement	variable tuple
`INSERT`	`NEW`
`DELETE`	`OLD`
`UPDATE`	`OLD` ou `NEW`

Une procédure trigger (comme toute autre procédure) est toujours éxécutée dans une transaction : celle de la Commande qui l'a déclenchée.

		Valeur de Retour	
Type	Evenement	tuple	Contrôle
BEFORE	INSERT	NULL	Ignorer le tuple sans avorter la Transaction
		NEW	Insérer le tuple
	UPDATE	NULL	Ignorer le tuple sans avorter la Transaction
		OLD	Ignorer le tuple sans avorter la Transaction
		NEW	Mettre à jour du tuple
	DELETE	NULL	Ignorer le tuple sans avorter la Transaction
		OLD	Supprimer le tuple
AFTER	INSERT	NULL	Valeur de retour ignorée
	UPDATE	NULL	Valeur de retour ignorée
	DELETE	NULL	Valeur de retour ignorée

3.7.2 Exemple

```
1  CREATE TABLE note (
2    num_etudiant    int,
3    num_controle    int,
4    note            decimal(4,2)
5  );
6
7  CREATE FUNCTION fmaj_note()
8    returns TRIGGER as
9  $$
10   BEGIN
11     IF NEW.note < OLD.note
12         THEN
13             RETURN NULL;
14     END IF;
15     RETURN NEW;
16     END;
17 $$ language plpgsql ;
18
19 CREATE TRIGGER tmaj_note
20     BEFORE
21     UPDATE on note
22        FOR EACH ROW
23        EXECUTE PROCEDURE fmaj_note();
```

```
SELECT * FROM note;
 num_etudiant | num_controle | note
--------------+--------------+-------
        12345 |            1 | 13.50
(1 row)

UPDATE note set note=note+2;
UPDATE 1
SELECT * FROM note;
 num_etudiant | num_controle | note
--------------+--------------+-------
        12345 |            1 | 15.50
(1 row)

UPDATE note set note=note-2;
UPDATE 0
SELECT * FROM note;
 num_etudiant | num_controle | note
--------------+--------------+-------
        12345 |            1 | 15.50
(1 row)
```

3.8 Intérêts des triggers

- Exemple 1 : Gestion des données dérivées.
- Exemple 2 : Règles de Gestion et Contraintes d'Intégrité.
- Exemple 3 : Administration de la Base de Données
- etc ..

On considère le schéma simplifié ci-dessous :

```
CREATE TABLE Article(
    id_article   int primary key,
   nom          varchar,
    prix         money
);

CREATE TABLE Facture(
   id_article    int  references  Article ,
  Quantite       int,
  Prix_total     money
 );
```

et l'instance suivante :

```
 id_article |         nom          | prix
------------+----------------------+-------
       1231 | Petit pain fromage   | €0,35
       1232 | Petit pain graines   | €0,35
       1233 | Petit pain céréales  | €0,35
       1234 | XXL Papier toilette  | €4,99
       1235 | Batonnets quate      | €0,32
       1236 | Mouchoirs blancs     | €0,79
(6 rows)
```

3.8.1 Exemple 1 (Compléter une ligne de facture)

Petit pain fromage	0,35 A
Petit pain graines e	0,70 A
2 x 0,35	
Petit pain céréales	1,05 A
3 x 0,35	
XXL Papier toilette	4,99 B
Batonnets ouates	0,32 B
Mouchoirs blancs	2,37 B
3 x 0,79	
A payer	55,95

```
CREATE or replace FUNCTION Exemple1()
  returns TRIGGER as
$$
  DECLARE
     aprix  money;
  BEGIN
   IF TG_OP='UPDATE' or TG_OP='INSERT' THEN
     SELECT prix into aprix FROM Article
       WHERE id_article=NEW.id_article;
     NEW.Prix_total =aprix*NEW.Quantite;
   END IF;
   IF TG_OP='DELETE' THEN
     SELECT prix into aprix FROM Article
       WHERE id_article=OLD.id_article;
   END IF;
   IF (FOUND) THEN
      IF TG_OP='UPDATE' or TG_OP='DELETE' THEN
       raise notice ' OLD Article % : % * % = % ',
        OLD.id_article,aPrix,OLD.quantite,OLD.prix_total ;
      END IF;
      IF TG_OP='UPDATE' or TG_OP='INSERT' THEN
       raise notice ' NEW Article % : % * % = % ',
        NEW.id_article,aPrix,NEW.quantite,NEW.prix_total;
      END IF;
      IF TG_OP='UPDATE' or TG_OP='INSERT' THEN
         RETURN NEW;
      ELSE
         RETURN OLD;
      END IF;
    END IF;
     RETURN NULL;
  END;
$$ language plpgsql ;

CREATE TRIGGER exemple1
     BEFORE
     UPDATE or INSERT or DELETE ON facture
        FOR EACH ROW
        EXECUTE PROCEDURE exemple1();
```

```
INSERT INTO facture values
         (1231,1),(1232,2),(1233,3),(1234,1),(1235,1),(1236,3);
NOTICE:   NEW Article 1231 : €0,35 * 1 = €0,35
NOTICE:   NEW Article 1232 : €0,35 * 2 = €0,70
NOTICE:   NEW Article 1233 : €0,35 * 3 = €1,05
NOTICE:   NEW Article 1234 : €4,99 * 1 = €4,99
NOTICE:   NEW Article 1235 : €0,32 * 1 = €0,32
NOTICE:   NEW Article 1236 : €0,79 * 3 = €2,37
INSERT 0 6

SELECT a.nom Article,a.prix "Prix U",
       f.quantite "Quantite",f.prix_total "Total Ligne"
    FROM article a, facture f
    WHERE a.id_article=f.id_article;
       article       | Prix U | Quantite | Total Ligne
---------------------+--------+----------+-------------
 Petit pain fromage  |  €0,35 |        1 |       €0,35
 Petit pain graines  |  €0,35 |        2 |       €0,70
 Petit pain céréales |  €0,35 |        3 |       €1,05
 XXL Papier toilette |  €4,99 |        1 |       €4,99
 Batonnets quate     |  €0,32 |        1 |       €0,32
 Mouchoirs blancs    |  €0,79 |        3 |       €2,37
(6 rows)

SELECT sum(prix_total) "A payer" FROM facture;
 A payer
---------
   €9,78
(1 row)
```

```
UPDATE Facture SET Quantite=4
   WHERE id_article=1231;
NOTICE:   OLD Article 1231 : €0,35 * 1 = €0,35
NOTICE:   NEW Article 1231 : €0,35 * 4 = €1,40
UPDATE 1

DELETE FROM Facture
    WHERE id_article=1234;
NOTICE:   OLD Article 1234 : €4,99 * 1 = €4,99
DELETE 1

UPDATE Facture SET prix_total='1'
    WHERE id_article=1231;
NOTICE:   OLD Article 1231 : €0,35 * 4 = €1,40
NOTICE:   NEW Article 1231 : €0,35 * 4 = €1,40
UPDATE 1

SELECT a.nom Article,a.prix "Prix U",
       f.quantite "Quantite",f.prix_total "Total Ligne"
    FROM article a, facture f
    WHERE a.id_article=f.id_article;
       article       | Prix U | Quantite | Total Ligne
---------------------+--------+----------+-------------
 Petit pain graines  |  €0,35 |        2 |       €0,70
 Petit pain céréales |  €0,35 |        3 |       €1,05
 Batonnets quate     |  €0,32 |        1 |       €0,32
 Mouchoirs blancs    |  €0,79 |        3 |       €2,37
 Petit pain fromage  |  €0,35 |        4 |       €1,40
(5 rows)
```

3.8.2 Exemple 2 (Ne pas autoriser la modification d'un Article)

```
1  CREATE or replace FUNCTION Exemple2()
2    returns TRIGGER as
3  $$
4    BEGIN
5      IF TG_OP='UPDATE' or TG_OP='DELETE' THEN
6       raise notice ' OLD Article % : % % ',
7          OLD.id_article,OLD.nom,OLD.Prix ;
8      END IF;
9      IF TG_OP='UPDATE' or TG_OP='INSERT' THEN
10      raise notice ' NEW Article % : % % ',
11         NEW.id_article,NEW.nom,NEW.Prix ;
12     END IF;
13      RETURN NULL;
14   END;
15 $$ language plpgsql ;
16
17 CREATE TRIGGER exemple2
18      BEFORE
19      UPDATE or INSERT or DELETE ON Article
20         FOR EACH ROW
21         EXECUTE PROCEDURE exemple2();
```

```
INSERT INTO article VALUES(44444,'Vin','35');
NOTICE:   NEW Article 44444 : Vin €35,00
INSERT 0 0

DELETE FROM Article WHERE id_article=44443;
NOTICE:   OLD Article 44443 : Voiture €9 980,00
DELETE 0

UPDATE Article SET prix='2000' WHERE id_article=44443;
NOTICE:   OLD Article 44443 : Voiture €9 980,00
NOTICE:   NEW Article 44443 : Voiture €2 000,00
UPDATE 0

select * from article;
 id_article |      nom      |   prix
------------+---------------+-----------
      44441 | Cote de Boeuf |    €45,50
      44442 | Foie Gras     |   €120,15
      44443 | Voiture       | €9 980,00
(3 rows)
```

3.8.3 Exemple 3 (Journal des Factures)

```
1  CREATE TYPE event AS
2     ENUM ('INSERT', 'DELETE', 'UPDATE');
3
4  CREATE TABLE journal_facture(
5   action        event,
6   estampille    timestamp
7                 default current_timestamp,
8   old_tuple     facture ,
9   new_tuple     facture
10 );
11
12 CREATE or replace FUNCTION Exemple3()
13   returns TRIGGER as
14 $$
15   BEGIN
16     IF TG_OP='INSERT' THEN
17        INSERT INTO journal_facture(action,new_tuple)
18            values(TG_OP::event,NEW);
19     ELSIF TG_OP='UPDATE' THEN
20        INSERT INTO journal_facture(action,old_tuple,new_tuple)
21            values(TG_OP::event,OLD,NEW);
22     ELSE  -- DELETE
23        INSERT INTO journal_facture(action,old_tuple)
24            values(TG_OP::event,OLD);
25     END IF;
26     RETURN NULL;
27   END;
28 $$ language plpgsql ;
29
30
31 CREATE TRIGGER exemple3
32      AFTER
33      UPDATE or INSERT or DELETE ON Facture
```

```
34      FOR EACH ROW
35      EXECUTE PROCEDURE exemple2();
```

```
select * from journal_facture;
 action | estampille | old_tuple | new_tuple
--------+------------+-----------+-----------
(0 rows)

select * from facture;
 id_article | quantite | prix_total
------------+----------+------------
       1232 |        2 |      €0,70
       1233 |        3 |      €1,05
       1235 |        1 |      €0,32
       1236 |        3 |      €2,37
       1231 |        4 |      €1,40
(5 rows)

 DELETE FROM facture Where quatite>2;
NOTICE:   OLD Article 1233 : €0,35 * 3 = €1,05
NOTICE:   OLD Article 1236 : €0,79 * 3 = €2,37
NOTICE:   OLD Article 1231 : €0,35 * 4 = €1,40
DELETE 3

INSERT INTO facture values
          (1231,10),(1233,2);
NOTICE:   NEW Article 1231 : €0,35 * 10 = €3,50
NOTICE:   NEW Article 1233 : €0,35 * 2 = €0,70
INSERT 0 2

UPDATE Facture SET Quantite=4
   WHERE id_article=1231;
NOTICE:   OLD Article 1231 : €0,35 * 10 = €3,50
NOTICE:   NEW Article 1231 : €0,35 * 4 = €1,40
UPDATE 1

SELECT * FROM facture;
 id_article | quantite | prix_total
------------+----------+------------
       1232 |        2 |      €0,70
       1235 |        1 |      €0,32
       1233 |        2 |      €0,70
       1231 |        4 |      €1,40
(4 rows)

SELECT * FROM Journal_facture;
 action |         estampille         |     old_tuple     |     new_tuple
--------+----------------------------+-------------------+-------------------
 DELETE | 2013-12-11 10:48:42.544715 | (1233,3,"€1,05")  | NULL
 DELETE | 2013-12-11 10:48:42.544715 | (1236,3,"€2,37")  | NULL
 DELETE | 2013-12-11 10:48:42.544715 | (1231,4,"€1,40")  | NULL
 INSERT | 2013-12-11 10:50:06.224707 | NULL              | (1231,10,"€3,50")
 INSERT | 2013-12-11 10:50:06.224707 | NULL              | (1233,2,"€0,70")
 UPDATE | 2013-12-11 10:50:57.632728 | (1231,10,"€3,50") | (1231,4,"€1,40")
(6 rows)
```

3.9 Visibilité des données

3.9.1 Règles de base

Niveau	Quand	Visibilité
ROW	BEFORE	ne voit pas les majs (tuple) de la requête qui l'a déclenché Par contre, il voit les majs des autres tuples.
	AFTER	voit les majs (tuple) de la requête qui l'a déclenché.
STATEMENT	AFTER	voit tous les tuples majs.
	BEFORE	ne voit aucune maj.

3.9.2 Exemple

```
1  CREATE TABLE test_table (
2    test_data     int
3  );
4
5  CREATE or replace FUNCTION trig_proc()
6   RETURNS TRIGGER AS
7  $$
8   DECLARE
9     nbtuples     int;
10    Table        name;
11    Quand        text;
12    r            test_table%ROWTYPE;
13  BEGIN
14     Quand := TG_WHEN;
15     Table := TG_RELNAME;
16     IF Table != 'test_table' THEN  -- mauvais usage
17       raise exception '% fired on %',
18             TG_NAME,Table;
19     END IF;
20
21    IF TG_OP='DELETE' THEN
22    raise notice  -- tuple en cours
23     '% [declenche %] % : ( % )',
24      TG_NAME, Quand ,TG_OP , OLD.test_data;
25    ELSE
26    raise notice  -- tuple en cours
27     '% [declenche %] % : ( % )',
28      TG_NAME, Quand ,TG_OP , NEW.test_data;
29    END IF;
30
31
32    -- Determiner les tuples de 'test_table' vus
```

```
33    -- par la procedure trigger !!!
34    FOR r IN SELECT * FROM test_table LOOP
35        raise notice
36     '       %   : ',r.test_data;
37    END LOOP;
38
39    -- Pas de valeur NULL (insert, update)
40    IF TG_OP != 'DELETE' AND Quand = 'BEFORE' THEN
41          IF NEW.test_data is NULL THEN RETURN NULL; END IF;
42    END IF;
43
44    -- Valeur de retour
45    if TG_OP= 'DELETE' then
46       RETURN OLD;
47    else -- UPDATE INSERT
48       RETURN NEW;
49    end if;
50  END;
51 $$ LANGUAGE plpgsql;
```

```
1  CREATE TRIGGER trig_before BEFORE INSERT OR UPDATE OR DELETE
2     ON test_table
3      FOR EACH ROW EXECUTE PROCEDURE trig_proc();
4
5  CREATE TRIGGER trig_after AFTER INSERT OR UPDATE OR DELETE
6     ON test_table
7      FOR EACH ROW EXECUTE PROCEDURE trig_proc();
```

3.9.3 Insertion

```
SELECT * FROM test_table;
 test_data
-----------
(0 rows)

INSERT INTO test_table VALUES (NULL);
NOTICE:  trig_before [declenche  BEFORE] INSERT : ( <NULL> )
INSERT 0 0
   -- Insertion ignorée => trig_after non déclenché

SELECT * FROM test_table;
 test_data
-----------
(0 rows)
```

```
INSERT INTO test_table VALUES (1);
NOTICE:  trig_before [declenche  BEFORE] INSERT : ( 1 )
NOTICE:  trig_after [declenche  AFTER] INSERT : ( 1 )
NOTICE:          1   :
INSERT 0 1
   -- trigger_before ne voit le tuple inséré

SELECT * FROM test_table;
 test_data
-----------
         1
(1 row)


INSERT INTO test_table SELECT test_data * 2 FROM test_table;
NOTICE:  trig_before [declenche  BEFORE] INSERT : ( 2 )
NOTICE:          1   :
NOTICE:  trig_after [declenche  AFTER] INSERT : ( 2 )
NOTICE:          1   :
NOTICE:          2   :
INSERT 0 1

SELECT * FROM test_table;
 test_data
-----------
         1
         2
(2 rows)
```

```
INSERT INTO test_table SELECT test_data + 2 FROM test_table;
NOTICE:  trig_before [declenche  BEFORE] INSERT : ( 3 )
NOTICE:          1   :
NOTICE:          2   :
NOTICE:  trig_before [declenche  BEFORE] INSERT : ( 4 )
NOTICE:          1   :
NOTICE:          2   :
NOTICE:          3   :
NOTICE:  trig_after [declenche  AFTER] INSERT : ( 3 )
NOTICE:          1   :
NOTICE:          2   :
NOTICE:          3   :
NOTICE:          4   :
NOTICE:  trig_after [declenche  AFTER] INSERT : ( 4 )
NOTICE:          1   :
NOTICE:          2   :
NOTICE:          3   :
NOTICE:          4   :
INSERT 0 2


SELECT * FROM test_table;
 test_data
-----------
         1
         2
         3
         4
(4 rows)
```

3.9.4 Suppression

```
delete from test_table where test_data>2;
NOTICE:  trig_before [declenche  BEFORE] DELETE : ( 3 )
NOTICE:          1   :
NOTICE:          2   :
NOTICE:          3   :
NOTICE:          4   :
NOTICE:  trig_before [declenche  BEFORE] DELETE : ( 4 )
NOTICE:          1   :
NOTICE:          2   :
NOTICE:          4   :
NOTICE:  trig_after [declenche  AFTER] DELETE : ( 3 )
NOTICE:          1   :
NOTICE:          2   :
NOTICE:  trig_after [declenche  AFTER] DELETE : ( 4 )
NOTICE:          1   :
NOTICE:          2   :
DELETE 2

SELECT * FROM test_table;
 test_data
-----------
         1
         2
(2 rows)
```

3.9.5 Mise à Jour

```
UPDATE test_table SET test_data=NULL where test_data=2;
NOTICE:  trig_before [declenche  BEFORE] UPDATE : ( <NULL> )
NOTICE:          1   :
NOTICE:          2   :
UPDATE 0

SELECT * FROM test_table;
 test_data
-----------
         1
         2
(2 rows)

UPDATE test_table SET test_data=test_data+2;
NOTICE:  trig_before [declenche  BEFORE] UPDATE : ( 3 )
NOTICE:          1   :
NOTICE:          2   :
NOTICE:  trig_before [declenche  BEFORE] UPDATE : ( 4 )
NOTICE:          2   :
NOTICE:          3   :
NOTICE:  trig_after [declenche  AFTER] UPDATE : ( 3 )
NOTICE:          3   :
NOTICE:          4   :
NOTICE:  trig_after [declenche  AFTER] UPDATE : ( 4 )
NOTICE:          3   :
NOTICE:          4   :
UPDATE 2

SELECT * FROM test_table;
 test_data
-----------
         3
         4
(2 rows)
```

3.10 Contrainte trigger

3.10.1 Définition

Une contrainte trigger est une contrainte implémentée par un trigger.

```
1 CREATE CONSTRAINT TRIGGER nom_trigger
2     AFTER [ INSERT | DELETE | UPDATE [ OR ...]
3     ON relation constraint attributes
4     FOR EACH ROW
5     EXECUTE PROCEDURE fonction_trigger ( args )
```

où :

- `nom_trigger` : nom de la contrainte trigger,
- `relation` : nom de la table associé aux évenements,
- `attributes` : attributs de la contrainte : SET CONSTRAINTS
- `func(args)` : procédure trigger à appeler

3.10.2 Exemple

On considère le schéma de relation `salle` suivante :

```
create table salle (
        salle  varchar(30) primary key,
        enseignant    varchar(30) not null
);
```

et l'instance :

```
SELECT  * FROM salle;
 salle | enseignant
-------+------------
 R100  | Jeremy
 R200  | Joan
 R205  | John
(3 rows)
```

La contrainte trigger `check_ens` suivante permet d'implémenter la règle de gestion suivante :

Un enseignant ne peut être affecté qu'à une seule salle

```
create or replace function check_ens()
returns trigger as
$$
DECLARE
        cpt  int;
BEGIN
        SELECT count(*) into cpt
           FROM salle
           WHERE enseignant = NEW.enseignant;
        IF  cpt > 1 THEN
            RAISE EXCEPTION 'Chevauchement enseignant %',
                            NEW.enseignant;
        END IF;
        return NULL;
END;
$$ language 'plpgsql';
```

```
create constraint trigger check_ens
 after UPDATE OR INSERT on salle
 initially deferred
 for each row execute procedure check_ens();
```

Supposons que l'on veut permuter les enseignants : 'John' et 'Joan' :

```
update salle
  set enseignant='John'
  where salle='R200';
ERROR:  Chevauchement enseignant John
```

```
begin;
BEGIN

update salle
   set enseignant='John'
   where salle='R200';
UPDATE 1

update salle
   set enseignant='Joan'
   where salle='R205';
UPDATE 1

commit;
COMMIT


 SELECT  * FROM salle;
 salle | enseignant
-------+------------
 R100  | Jeremy
 R200  | John
 R205  | Joan
(3 rows)
```