INSIA
Bases de données
PostgreSQL - 3 – PL-SQL
Régles et triggers

Bertrand LIAUDET

SOMMAIRE

Première édition : avril 2009

PL-SQL

PL / PGSQL

Présentation

Les PL-SQL de PostgreSQL

PostgreSQL permet d'utiliser plusieurs langages de programmation qui permettent d'écrire des procédures, des fonctions et des triggers : les PL-SQL.

Langage PL/SQL

« sql » est un PL-SQL assez standard livré avec PostgreSQL.

Langage PL/PgSQL

« plpgsql » est un PL-SQL assez standard livré avec PostgreSQL.

Langage PL/Perl

« plperl » est un PL-SQL qu'il faut installer en plus de PostgreSQL.

Il faut installer la librairie : plperl.dll

Langage PL/Php

« plphp » un PL-SQL qu'il faut installer en plus de PostgreSQL.

http://www.commandprompt.com/community/plphp

Il existe deux versions du langage : plphp et plphpu. Le plphpu (u pour unstrusted) permet des accès aux systèmes de fichiers de PostgreSQL

Langage PL/Java

« pljava » un PL-SQL qu'il faut installer en plus de PostgreSQL

Activation d'un PL-SQL de PostgreSQL

Create language

La commande SQL « create language » permet d'activer un PL-SQL

Postgres-#create language plpgsql ;

Createlang

La commande SE « createlang » permet d'activer un PL-SQL

$createlang plpgsql base

Create language

La commande SQL « create language » permet d'activer un PL-SQL

Postgres-#create language plpgsql ;

Droits nécessaires pour utiliser le PL – SQL

Pour créer des fonctions PL-SQL, il faut avoir un droit sur le language :

GRANT … ON LANGUAGE …

Langages PL / SQL et PL / PGSQL

Documentation

http://docs.postgresqlfr.org/8.1/plpgsql.html

Fonction

Principe

Les langages PL / SQL et PL / PGSQL permettent d'écrire des fonctions. Les procédures sont des fonctions dont le type de retour sera « void ».

Création et structure d'une fonction

```
CREATE OR REPLACE
FUNCTION  nomFonction (paramètres) RETURNS type AS $$
[DECLARE]
    -- bloc de déclaration des variables locales
[BEGIN]
    -- bloc des instructions de la fonction
[END] $$ LANGUAGE nomLangage ;
```

Suppression d'une fonction

```
DROP FUNCTION nomFonction();
```

Quand on remplace une fonction, on doit garder le même type de retour. Pour changer le type de retour, il faut donc commencer par supprimer la fonction.

Exemple 1

- ***plpgsql***

```
CREATE OR REPLACE
FUNCTION  prixTTC (prixHT numeric) RETURNS numeric AS $$
BEGIN
    Return prixHT *1.196;
END $$ LANGUAGE 'plpgsql' ;
```

- ***sql***

```
CREATE OR REPLACE
FUNCTION  prixTTC (numeric) RETURNS numeric AS $$
    select $1 *1.196;
$$ LANGUAGE SQL;
```

Pas de BEGIN et de END en PL / SQL

Usages

- ***1ere forme : retour d'un attribut de type structuré***

```
Select prixTTC(10);
```

- ***2eme forme : retour d'une table***

```
Select * from prixTTC(10) ;
```

Exemple 2

- ***plpgsql***

```
CREATE OR REPLACE
FUNCTION  nbLivresEmpruntes (v_numAdherent int) RETURNS int AS $$
DECLARE
    v_nbLivres int;
    v_nomAdherent text;
BEGIN
    select nom into v_nomAdherent from adherents
    where na=v_numAdherent;
    RAISE NOTICE 'Adherent %, n° %', v_nomAdherent, v_numAdherent;
    select count(*) into v_nbLivres from emprunter
    where na=v_numAdherent and dateret is null;
    RAISE NOTICE 'Nombre d''emprunts : %',v_nbLivres ;
    Return v_nbLivres;
END $$ LANGUAGE 'plpgsql' ;
```

Déclaration des variables : bloc DECLARE

Les variables sont déclarées une à la fois avec leurs types.

Affectation des variables

« = » ou « := » au choix !

Retour

Pas de retour

RETURNS void

Retour simple

Tous les types simples : int, varchar, numeric, etc.

Retour d'enregistrement

RETURNS record // pour tout enregistrement : enregistrement générique. Attention à l'usage !

RETURNS nomTable // enregistrement correspondant à une table

- ***sql***

```
DROP FUNCTION testRecord(integer);
CREATE OR REPLACE
FUNCTION  testRecord(integer) RETURNS record
AS $$
    select * from oeuvres where no=$1;
$$ LANGUAGE SQL RETURNS NULL ON NULL INPUT;
```

- ***plpgsql***

```
DROP FUNCTION testRecordTable(integer);
CREATE OR REPLACE
FUNCTION  testRecordTable(v_no integer) RETURNS oeuvres AS $$
DECLARE
    res oeuvres;
BEGIN
    select * into res from oeuvres where no=v_no;
    return res;
END $$ LANGUAGE 'plpgsql' ;
```

```
DROP FUNCTION testRecord(integer);
CREATE OR REPLACE
FUNCTION  testRecord(v_nl integer) RETURNS record AS $$
DECLARE
    res record;
BEGIN
    select l.nl, l.editeur, o.* into res from livres l, oeuvres o
    where l.no=o.no and nl=v_nl;
    return res;
END $$ LANGUAGE 'plpgsql' ;
```

- ***Usage avec retour d'un attribut de type structuré***

```
Select testRecord(1);
```

Cet usage renvoie un type structuré (champs entre parenthèses).

- ***Usage avec retour d'une table***

```
Select * from testRecordTable(1) as (num integer, titre varchar,
    auteur varchar);
```

Avec les « record » (enregistrement générique non typé) et les « SETOF record », il faut préciser le nom et le type du record de sortie dans l'usage avec un AS.

Retour d'ensemble

RETURNS SETOF typeSimple

RETURNS SETOF record

RETURNS SETOF nomTable

- ***sql***

```
DROP FUNCTION test(integer);
CREATE OR REPLACE
FUNCTION  test(integer) RETURNS SETOF oeuvres AS
$$
    select * from oeuvres where no<$1;
$$ LANGUAGE SQL ;
```

- ***plpgsql***

```
DROP FUNCTION test(integer);
CREATE OR REPLACE
FUNCTION  test(num_oeuvre integer) RETURNS SETOF oeuvres AS $$
DECLARE
    resultat oeuvres%ROWTYPE ;
BEGIN
    For resultat in select * from oeuvres where no < num_oeuvre
    loop
         Return next resultat;
    End loop;
    RETURN;
END $$ LANGUAGE 'plpgsql' ;
```

- ***Usage avec retour d'un attribut de type structuré***

```
Select * from test(5);
```

- ***Usage avec retour d'un attribut de type structuré***

```
Select * from test(5);
```

Avec les types « table » les « SETOF table », on peut faire un « select * from function » simple.

Avec les « record » (enregistrement générique non typé) et les « SETOF record », il faut préciser le nom et le type du record de sortie dans l'usage avec un AS.

Paramètres en sortie : IN

IN : on peut préciser IN devant les paramètres en entrée. Remarque : les paramètres en entrée ne peuvent pas être modifés.

Paramètres en sortie : OUT

OUT : la sortie standard (le retour) peut être passée en paramètre : le nom de la variable est précédé d'un OUT et mis entre guillemets : ce sera le nom de la colonne résultat.

- ***plpgsql***

```
drop function if exists equa1();
create or replace function equa1(a float, b float, out x float, out
    nbsol integer) returns record as $$
-- remarque: les OUT imposent le returns record
begin
    if a = 0 then
         if b = 0 then
             nbsol=-1;
         else
             nbsol=0;
         end if;
    else
         nbsol=1;
         x=-b/a;
    end if;
    -- RAISE INFO 'equa1 : nbsol %',nbsol;
    return;
end $$ language 'plpgsql';
```

- ***Usage avec retour d'un attribut de type structuré***

```
select equa1(1.0, 2.0);
```

- ***Usage avec retour d'une table***

```
select * from equa1(1.0, 2.0);
```

Affichage

Une fonction est faite pour renvoyer des résultats qui sont considérés somme une table.

Les fonctions ne sont pas faites pour produire de l'affichage.

Toutefois le RAISE permet d'envoyer des messages d'informations qui sont indépendants, du point de vue de l'affichage, des résultats de la fonction.

Le RAISE

```
RAISE typeRaise 'chaine', variables ;
```

TypeRaise : NOTICE, INFO, EXCEPTION

Mais aussi : WARNING, DEBUG, LOG

NOTICE : affichage simple

INFO : affiche le contexte en plus (qui a appelé la fonction).

EXCEPTIONS : pour gérer les exceptions.

Exemple

```
CREATE OR REPLACE
FUNCTION  plus(v_a integer,v_b integer) RETURNS integer AS $$
BEGIN
    RAISE INFO 'fonction plus : addition de % et %', v_a, v_b;
--  RAISE WARNING 'fonction plus : addition de % et %', v_a, v_b;
--  RAISE NOTICE 'fonction plus : addition de % et %', v_a, v_b;
    Return v_a + v_b;
END $$ LANGUAGE 'plpgsql' ;
```

```
CREATE OR REPLACE
FUNCTION  main(v_a integer,v_b integer) RETURNS void AS $$
DECLARE
    res integer;
BEGIN
    RAISE INFO 'main : appel à la fonction plus';
    res = plus(v_a, v_b) ;
    RAISE NOTICE 'résultat = %', res ;
END $$ LANGUAGE 'plpgsql' ;
```

Concaténation de chaînes

- ***sql***

```
CREATE OR REPLACE
FUNCTION  prixTTC (numeric) RETURNS varchar
AS $$
    select 'prix ht : ' || $1 || ' – prix ttc : ' || $1*1.196;
$$ LANGUAGE SQL;
```

- ***pspgsql***

```
DROP FUNCTION IF EXISTS prixTTC(numeric);
CREATE OR REPLACE
FUNCTION  prixTTC (prixHT numeric) RETURNS varchar AS $$
BEGIN
    return 'prix ht : ' || $1 || ' – prix ttc : ' || $1*1.196;
END $$ LANGUAGE 'plpgsql' ;
```

- ***Usages***

```
select prixTTC(10);
select * from prixTTC(10);
```

Transtypage et CAST

Transtyper une variable

- ***Opérateur « :: »***

  L'expression « maVariable ::type » change le type de la variable.

  Un entier peut être casté en chaîne.

  Un ensemble de variable peut être casté en enregistrement

```
SELECT ROW (attributs) ::nomTypeEnregistrement FROM …
SELECT ROW (attributs) ::nomTable FROM …
```

- ***Fonction CAST***

  CAST(expression) AS type

  Une chaîne peut être castée en entier.

- ***Transtypage automatique***

  Les opérandes autour d'un opérateur (+, ||, etc.) sont transtypés automatiquement.

  Les opérandes passés en paramètre à des fonctions doivent être transtypés explicitement.

Structures de contrôle

Test

```
IF (condition) THEN
    Instructions;
ELSEIF (condition) THEN
    Instructions;
ELSE
    Instructions;
END IF ;
```

Boucle for

```
FOR i IN 1..N loop
        Instructions;
END LOOP ;
```

Il n'est pas nécessaire de déclarer i.

Boucle for avec select

```
RETURNS SETOF nomTable
...
FOR resultat in SELECT * FROM nomTable ... LOOP
    ...
    RETURN NEXT resultat;
    ...
END LOOP;
```

Boucle while

```
WHILE condition LOOP
        Instructions;
END LOOP ;
```

Il n'est pas nécessaire de déclarer i.

Boucle sans fin

```
LOOP
    ...
END LOOP;
```

Débranchements

EXIT : fait sortir de la boucle

RETURN : fait sortir de la fonction

Exemple

- ***pspgsql***

```
\!cls
drop function if exists pgcd(integer);
create or replace function pgcd(a integer, b integer) returns
    integer as $$
declare
    va integer; // on duplique les paramètres en entrée
    vb integer; // car ils ne peuvent pas être modifiés.
begin
    va=a;
```

```
    vb:=b; // = ou := pour l'affectation
    while va != vb loop
         if va > vb then
            va=va-vb;
         else
            vb=vb-va;
         end if;
    end loop;
    return va;
end $$ language 'plpgsql';
```

- ***Usages***

```
select pgcd(57, 135);
select * from pgcd(57, 135) ;
```

Curseur, Fetch

Principe

La gestion des curseurs s'apparente à la gestion de fichiers en lecture.

Un curseur est un pointeur sur une ligne d'un select.

Un fetch permet de récupérer la ligne pointée par un curseur et de déplacer le curseur.

Déclaration d'un curseur : type refcurseur

```
curseur refcurseur ;
```

Ouverture du curseur

```
OPEN curseur SCROLL FOR SELECT … ;
```

Le SCROLL permet de revenir en arrière.

A l'ouverture le curseur ne pointe sur aucune ligne.

Déplacement du curseur et lecture d'une ligne

```
FETCH NEXT FROM curseur INTO maVariable;
```

Le NEXT permet de spécifier le déplacement : NEXT : suivant, PRIOR ou BACKWARD : précédent, LAST : à la fin, FIRST : au début, etc.

Le INTO permet de lire la ligne pointée par le curseur.

Exemple 1

```
CREATE OR REPLACE
FUNCTION  test() RETURNS SETOF oeuvres
AS $$ DECLARE
    nb_lignes integer:=0;
    curseur refcursor;
    une_oeuvre oeuvres%ROWTYPE;
BEGIN
    OPEN curseur SCROLL FOR SELECT * FROM oeuvres order by no;
    FETCH NEXT FROM curseur INTO une_oeuvre;
    RETURN NEXT une_oeuvre;

    FETCH LAST FROM curseur INTO une_oeuvre;
    RETURN NEXT une_oeuvre;

    FETCH PRIOR FROM curseur INTO une_oeuvre;
    RETURN NEXT une_oeuvre;

    FETCH FIRST FROM curseur INTO une_oeuvre;
    RETURN NEXT une_oeuvre;

    CLOSE curseur;
END $$ LANGUAGE 'plpgsql' ;
```

Exemple 2

```
DROP FUNCTION test(integer);
CREATE OR REPLACE
FUNCTION  test(num_oeuvre integer) RETURNS SETOF oeuvres
AS $$ DECLARE
    nb_lignes integer:=0;
    curseur refcursor;
    une_oeuvre oeuvres%ROWTYPE;
BEGIN
    OPEN curseur SCROLL FOR
         SELECT * FROM oeuvres WHERE no < num_oeuvre ORDER BY no;
    LOOP
         FETCH NEXT FROM curseur INTO une_oeuvre;
         IF(une_oeuvre IS NULL) THEN
             EXIT; -- sortie de boucle
         END IF;
         RETURN NEXT une_oeuvre;
    END LOOP;
    RAISE NOTICE 'Sortie de la boucle';
END $$ LANGUAGE 'plpgsql' ;
```

- ***Usage***

```
select * from test(5) ;
```

L'usage avec retour d'un attribut de type structuré n'est pas autorisé.

Exceptions

La gestion des exceptions s'apparente à celle des langages orientés objet.

```
CREATE OR REPLACE
FUNCTION  nblTab(maTable varchar) RETURNS integer
AS $$ DECLARE
    nb_lignes integer;
BEGIN
    IF (maTable is null or maTable='NULL') THEN
         RAISE EXCEPTION 'table indefinie';
    END IF;
    EXECUTE 'select count(*) from ' || maTable INTO nb_lignes;
    RETURN nb_lignes;
EXCEPTION
    WHEN UNDEFINED_TABLE THEN
         RAISE NOTICE 'la table n''existe pas';
         RETURN 0 ;
    WHEN RAISE_EXCEPTION THEN
         RAISE NOTICE 'la table est indefinie';
         RETURN 0 ;
END $$ LANGUAGE 'plpgsql' ;
```

Dans le bloc EXCEPTION, des sous blocs WHEN conditions THEN.

La condition correspond à des nombreux codes d'erreur prédéfinis.

Le RAISE_EXCEPTION correspond à la ou les exceptions spécifiques de la fonction. Il peut y avoir plusieurs « RAISE EXCEPTION » dans la fonction, mais il ramène à un seul « RAISE_EXCEPTION » et il n'y a pas de moyen de les distinguer.

http://www.postgresql.org/docs/8.1/static/plpgsql-errors-and-messages.html

Exécute

A noter dans l'exemple précédent le « EXECUTE » qui permet d'exécuter une chaîne de caractère contenant un « SELECT » ou n'importe quel ordre SQL.

Exemple 1

```
DROP FUNCTION test(varchar);
CREATE OR REPLACE
FUNCTION  test(varchar) RETURNS varchar
AS $$
    if (! Defined $_[0]){
         Return undef;
    }
    return uc($_[0]); -- uc: function de mise en majuscule
$$ LANGUAGE 'plperl' ;
```

$_[0] contient le premier paramètre, $_[1] le deuxième, etc.

Exemple 2

```
DROP FUNCTION test();
CREATE OR REPLACE
FUNCTION  test() RETURNS SETOF varchar
AS $$
    # la requête
    my $requete = spi_exec_query('SELECT * from auteur');

    # nombre de lignes
    my $nbLignes = $requete->{processed};

    # affichage d'un message
    elog(INFO, "$nbLignes lignes dans requête") ;

    # parcours de la table
    foreach my $i (0 .. $nbLignes –1){
         my $ma_ligne = $requete->{rows}[$i];
         return_next("$ma_ligne->{auteur}."  écrit ".$ma_ligne-
    >{titre}."."") ;
    }
    return ;
$$ LANGUAGE 'plperl' ;
```

my $nomVar : declaration de variable locale.

REGLES ET TRIGGERS

Les règles

Présentation et création

Les règles sont des actions déclenchées par une action du DML select inclus.

Ces actions peuvent remplacer l'action du déclencheur ou s'y ajouter (comme pour un trigger after).

```
CREATE [OR REPLACE] RULE nomRègle AS ON événement
TO nomTable [WHERE condition]
DO  [ALSO | INSTEAD]
{NOTHING | commande | (commande; commande…) };
```

L'événement est un ordre du DML: INSERT, UPDATE, DELETE, ou SELECT ;

La condition permet de filtrer l'événement en fonction des valeurs de la ligne sélectionnée. Cette ligne est désignée par OLD ou NEW. OLD désigne les anciennes valeurs du tuple en cours (valable pour les UPDATE et les DELETE). NEW désigne les nouvelles valeurs du tuple en cours (valable pour les INSERT et les UPDATE).

ALSO précise que le déclencheur est aussi réalisé tandis que INSTEAD précise que seule la règle est réalisée.

L'action à réaliser peut être constituée de plusieurs commandes. S'il n'y a pas d'action, on écrit NOTHING (utile pour ne pas exécuter le déclencheur).

Exemple

Dans la BD bibliothèque, on décide de ne plus supprimer les adhérents mais de préciser pour chaque adhérent s'il est actif ou pas.

Les applications qui utilisent la BD envoyaient des ordres DELETE.

On va créer une règle qui remplacera les DELETE dans la table des adhérents par une mise à jour du champ « actif » de la table des adhérents.

```
CREATE RULE supprime_adherent AS ON DELETE
TO adherent
DO INSTEAD
UPDATE adherent SET actif = false WHERE NA=OLD.NA;
```

Suppression d'une règle

```
DROP RULE nomRègle ON nomTable;
```

Les triggers (déclencheurs)

Présentation

- Les triggers sont des actions déclenchés par une action du DML select exclu.
- Les triggers peuvent être codés en PL/PgSQL, PL/Perl, PL/Python, langage C, etc
- Les triggers se composent de deux éléments : le déclencheur et la fonction exécutée par le déclencheur. Cette fonction est sans argument et renvoie un résultat de type trigger.
- Plusieurs déclencheurs peuvent appeler la même fonction.
- La fonction déclenchée peut interagir avec le SE si elle est écrite en C, PL/PerlU (U pour untrusted).
- OLD et NEW désignent la ligne à l'origine du déclenchement : OLD désigne les anciennes valeurs de cette ligne. OLD n'existe que pour un UPDATE ou un DELETE. NEW désigne les nouvelles valeurs de cette ligne. NEW n'existe que pour un INSERT ou un UPDATE.
- Les déclencheurs peuvent se déclencher en cascade.

Syntaxe et création

```
CREATE TRIGGER nomTrigger
{ BEFORE | AFTER } {INSERT | UPDATE | DELETE | conjonction }
ON nomTable
{ FOR EACH ROW | FOR EACH STATEMENT }
EXECUTE PROCEDURE nomFonction();
```

BEFORE ou AFTER :

Le trigger BEFORE permet de faire des vérifications avant la modification pour éventuellement arrêter la modification (RETURN NULL). Il permet aussi de faire de modifications sur le tuple en cours de traitement avant la modification (RETURN NEW).

Le trigger AFTER permet de faire des modifications sur d'autres tuples que le tuple en cours de traitement, après la modification de ce dernier (RETURN NULL).

INSERT, UPDATE ou DELETE :

Ce sont les trois instructions qui déclenchent le déclencheur.

On peut aussi mettre une conjonction de ces instructions : INSERT OR UPDATE, etc.

FOR EACH ROW ou FOR EACH STATEMENT

FOR EACH ROW précise que la fonction sera déclenchée pour chaque tuple de l'instruction INSERT, UPDATE ou DELETE.

FOR EACH STATEMENT précise que la fonction sera déclenchée une seule fois au début (BEFORE) ou à la fin (AFTER) de l'instruction INSERT, UPDATE ou DELETE, quel que soit le nombre de tuples concernés par l'instruction, et même s'il y en a 0.

Les triggers se déclenchent dans cet ordre :

BEFORE FOR EACH STATEMENT

BEFORE FOR EACH ROW

AFTER FOR EACH STATEMENT

AFTER FOR EACH ROW

Remarque

Il peut y avoir plusieurs déclencheurs de même type sur une même table. L'ordre de déclenchement sera l'ordre alphabétique des triggers. En cas de trigger BEFORE, le tuple en cours passe successivement d'un trigger à l'autre, le NEW du précédent devenant le OLD du suivant.

Boucle infinie !

Attention, un système de triggers complexe peut aisément produire des boucles infinies ! C'est à la charge du programmeur que d'éviter cela !

La fonction déclenchée par un trigger peut porter sur la table déclenchant le trigger. Toutefois, il faut faire attention aux boucles infinies.

Suppression

```
DROP TRIGGER nomTrigger ON nomTable;
```

Fonction du trigger

La fonction du trigger est une fonction PL/PgSQL avec quelques spécificités :

Spécificités d'en-tête

- Elle n'a pas de paramètres formels
- Elle renvoie un type « trigger » : RETURNS trigger
- En général, on donne le même nom au trigger et à la fonction, par simplification, mais ce n'est pas obligé.

Spécificité du corps

- La fonction a accès à deux variables : OLD et NEW qui correspondent au tuple en cours de modification avec ses anciennes valeurs ou ses nouvelles.
- Elle renvoie soit NULL, soit NEW. Dans le cas d'un trigger AFTER, elle renvoie toujours NULL. Dans le cas d'un trigger BEFORE, elle renvoie NULL si on souhaite éviter la modification (l'INSERT ou l'UPDATE). Elle renvoie NEW si on souhaite valider la modification.
- La fonction a accès à des variables qui lui permettent de connaître les caractéristiques du déclencheur. Par exemple TG_OP indique le type d'événement qui a provoqué l'exécution de la fonction. CF. http://www.postgresql.org/docs/7.4/interactive/plpgsql-trigger.html

Exemple

Trigger qui affiche le nombre de livres actuellement emprunté par un adhérent

```
CREATE OR REPLACE FUNCTION NbLivresEmpruntes()
RETURNS trigger AS $$
DECLARE
    NbLivres integer;
BEGIN
    SELECT count(*) INTO nbLivres
    FROM emprunter
    WHERE NA=NEW.NA
    AND dateRet is NULL;

    RAISE INFO 'Déclencheur provoqué par %', TG_OP ;
    RAISE INFO '% livres actuellement emprunté par l'adhérent %',
         nbLivres, NEW.NA ;
    RETURN NULL ;
END ; $$ LANGUAGE PLPGSQL ;

CREATE TRIGGER NbLivresEmpruntes
AFTER INSERT ON emprunter
FOR EACH ROW
EXECUTE PROCEDURE NbLivresEmpruntes();
```

TP

Le TP est fait en PLPGSQL

Programmation sans tables

1) Ecrire une fonction de résolution d'une équation du second degré dans R. On traitera tous les cas possibles et on utilisera une fonction de résolution d'une équation du premier degré.
2) Ecrire une fonction qui l'utilise et qui gère l'affichage.

BD biblio

1) Charger la BD biblio.
2) Dans la BD biblio, écrire une fonction qui permet d'afficher les livres sortis avec le numéro et le nom des personnes, le numéro du livre et son titre ainsi que la date d'emprunt.
3) Dans la même fonction, faire une variante qui permette d'afficher les livres sortis pour un adhérent.
4) Ecrire une fonction qui permet d'afficher les livres en retard avec le numéro et le nom des personnes, le numéro du livre et son titre ainsi que la date d'emprunt, la durée maximum d'emprunt, le nombre de jours de retard. Dans la mémé fonction, on aura la possibilité de traiter un adhérent particulier.
5) Réécrire la fonction 3 en utilisant des curseurs.
6) Dans la BD biblio, créer la procédure stockée qui gère un emprunt. On considère que la durée max est à 14 jours par défaut.
7) Dans la BD biblio, créer la procédure stockée qui gère un retour.
8) Dans la BD biblio, mettre en place le système qui permet d'éviter la suppression des adhérents, des livres et des œuvres.
9) Dans la BD biblio, écrire une règle qui, à chaque insertion de livres, affiche le livre et l'œuvre associée.
10) Dans la BD biblio, écrire une règle qui, à chaque emprunt, affiche : *prénom nom* (*numéro*) a emprunté *titre* de *auteur* aux éditions *édition* (*numéro livre*) le *date*.
11) Dans la BD biblio, écrire un trigger qui affiche le nombre de livres identiques restant après un emprunt et après un retour.
12) Dans la BD biblio, écrire un trigger qui empêche l'emprunt d'un livre déjà sorti.

BD Ecoling

1) Charger la BD Ecoling.
2) Dans la BD Ecoling, écrire une fonction qui permet d'afficher le nom du meilleur élève et du moins bon pour un examen donné avec leurs notes.
3) Dans la BD Ecoling, écrire un trigger qui vérifie, quand on donne une note à un étudiant, que l'évaluation concerne le groupe auquel l'étudiant appartient.
4) Quelle serait la solution de modélisation qui permettrait d'éviter l'écriture du trigger tout en vérifiant la contrainte ? Pouvez-vous envisager de reconfigurer toute la base en conséquence ?

BD Commandes

1. Charger la BD des commandes.
2. Faire le graphe des tables.
3. Regardez les index.
4. On souhaite gérer les livraisons dans cette BD. Chaque commande donne lieu à une ou plusieurs livraisons. Une livraison concerne un ou plusieurs produits de la commande dans la quantité commandée ou dans une quantité inférieure. Créer les tables correspondant à la situation.
5. Créer un trigger qui vérifie que chaque produit livré est bien un produit commandé.
6. Créer un trigger qui met à jour le nombre de produits livrés à chaque livraison : on ajoutera une quantité livrée dans la table détails_commande et on gérera cet attribut calculé via un trigger.
7. Créer un trigger qui met à jour l'état de la commande : on ajoutera un attribut booléen « livré » dans la table des commandes et on gérera cet attribut calculé via un trigger.
8. Créer un trigger qui empêche de livrer un produit qui n'est pas commandé ou de livrer un produit dans une quantité supérieure à la quantité commandée.