U.E BASE DE DONNÉES

Concepts Avancés de PostgreSQL

Elyes Lamine

école d'ingénieurs d'Informatique et de Systèmes d'Information pour la Santé
www.isis-ingenieur.fr

L3 INFO

Sommaire

Séquences

- Génération de clés primaires
  - Il est préférable d'avoir des identificateurs non significatifs, par exemple un nombre entier, dont la valeur n'est pas liée aux propriétés de l'objet
  - L'idée est de créer une suite de nombres entiers de telle sorte que jamais le même nombre n'est utilisé deux fois

```
SELECT MAX(ident) + 1 FROM MaTable;  --> Très lourd
```

- Tous les SGBD offrent désormais une facilité pour obtenir des identificateurs générés automatiquement par le SGBD sans avoir à accéder à une table.
- Cette facilité n'est malheureusement pas standardisée
  - MySQL permet d'ajouter la clause `AUTO_INCREMENT` à une colonne
  - DB2 et SQL Server ont une clause `IDENTITY` pour dire qu'une colonne est un identifiant
  - Oracle, DB2 et PostgreSQL utilisent des séquences.

Génération d'une séquence

- Une séquence est une sorte de table particulière qui permet de générer un nombre proprement.
- Les nombres générés proviennent d'une suite que l'on aura au préalable paramétrée dans la séquence.
- Il peut exister une infinité de séquences au sein d'une même base de données.
- Chaque séquence a ses propres caractéristiques et la génération d'un nombre n'affecte pas les autres séquence de la base.
- Une même table peut utiliser une ou plusieurs séquences différentes.

Création d'une séquence - 1

- La syntaxe Postgresql propose de nombreuses options

```
CREATE [TEMPORARY|TEMP] SEQUENCE nom
[INCREMENT [BY] incrément]
[ MINVALUE valeurmin | NO MINVALUE ]
[ MAXVALUE valeurmax | NO MAXVALUE ]
[ START [WITH] début] [[NO]CYCLE ]
[ OWNED BY {table.colonne | NONE} ]
```

- Exemple :

```
CREATE SEQUENCE ma_sequence MINVALUE 10 MAXVALUE 600 INCREMENT 7 CYCLE;
```

- TEMPORARY ou TEMP : l'objet séquence n'est créé que pour la session en cours et automatiquement supprimé lors de la sortie de session.
- INCREMENT BY incrément : précise la valeur à ajouter à la valeur courante de la séquence pour créer une nouvelle valeur.
  - Une valeur positive crée une séquence ascendante, une valeur négative une séquence descendante. 1 est la valeur par défaut.

Création d'une séquence - 2

```
CREATE [TEMPORARY|TEMP] SEQUENCE nom
[INCREMENT [BY] incrément]
[ MINVALUE valeurmin | NO MINVALUE ]
[ MAXVALUE valeurmax | NO MAXVALUE ]
[ START [WITH] début] [[NO]CYCLE ]
[ OWNED BY {table.colonne | NONE} ]
```

- `MINVALUE valeur min` (resp. `MAXVALUE valeurmax`) détermine la valeur minimale (resp. maximale) de la séquence.
- `START WITH début` permet à la séquence de démarrer n'importe où.
- `CYCLE` autorise la séquence à recommencer au début lorsque `valeurmax` ou `valeurmin` sont atteintes.
- `OWNED BY` permet d'associer la séquence à une colonne de table spécifique. De cette façon, la séquence sera automatiquement supprimée si la colonne (ou la table entière) est supprimée.

Utilisation d'une séquence

- Après la création d'une séquence, les fonctions NEXTVAL, CURRVAL et SETVAL sont utilisées pour agir sur la séquence.
  - SETVAL :Initialisation de la séquence

```
SELECT SETVAL('ma_sequence',456);-- on initialise ma_sequence à 45
```

  - NEXTVAL : Incrémentation de la séquence

```
SELECT NEXTVAL('ma_sequence'); -- on incrémente la prochaine valeu
```

  - CURRVAL : Lecture de la valeur courante de la séquence

```
SELECT CURRVAL('ma_sequence'); -- on demande la valeur courante
```

- Exemple

```
1 CREATE SEQUENCE ma_sequence MINVALUE 1;
2 INSERT INTO employe VALUES (NEXTVAL('ma_sequence'),'Fabien');
```

Création d'une séquence d'une manière implicite

- En Postgresql 9.xxx, une séquence est implicitement créée lorsque l'on déclare la création d'une table avec une colonne de type SERIAL.
  - La colonne se transformera en type INTEGER mais aura une valeur par défaut à NEXTVAL('Nom_table_NomClonne_seq');
- Exemple

```
CREATE TABLE ma_table (idauto SERIAL, nom VARCHAR(100));

NOTICE: CREATE TABLE will create implicit sequence
"ma_table_idauto_seq" for serial column "ma_table.idauto"
```

Modification, suppression d'une séquence

- Modifier une séquence

```
ALTER SEQUENCE nomDeLaSequence ... ;
```

- Exemples

```
ALTER SEQUENCE ma_sequence RESTART WITH 123;
ALTER SEQUENCE ma_sequence MINVALUE 10;
ALTER SEQUENCE ma_sequence INCREMENT 7;
```

- Suppression d'une séquence

```
DROP SEQUENCE nomSequence;
```

  - Mais si l'on tente de supprimer une séquence utilisée par une table, Postgresql renvoi le message d'erreur suivant :

```
ERROR: cannot drop sequence ma_table_idauto_seq because table
ma_table column idauto requires it

ASTUCE : You may drop table ma_table column idauto instead.
```

- Lorsque l'on supprime la colonne, la séquence qui y était liée est automatiquement supprimée.

```
ALTER TABLE ma_table DROP COLUMN idauto ;
```

Règles Postgresql- (1)

- Par défaut les vues ne sont pas modifiables dans Postgresql (contrairement à Access ou Oracle).
  - Les règles permettent de réécrire les commandes SELECT, INSERT, UPDATE, DELETE réalisées sur une table ou sur une vue.
- La syntaxe d'une règle est :

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

  - Événement : SELECT, INSERT, UPDATE ou DELETE.
  - Table : Le nom de la table ou de la vue sur laquelle s'applique la règle.
  - Condition : Toute expression SQL conditionnelle (renvoyant un type boolean).
    - L'expression de la condition ne peut pas faire référence à une table autre que NEW ou OLD ni contenir de fonction d'agrégat.

Règles Postgresql - (2)

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

- INSTEAD : Les commandes sont exécutées à la place de la commande originale.
- ALSO : Les commandes sont exécutées en plus de la commande originale. ALSO est utilisé par défaut
- Commande : l'action de la règle. Les commandes valides sont SELECT , INSERT , UPDATE , DELETE ou NOTIFY.
- À l'intérieur d'une condition ou d'une commande , les noms des tables spéciales NEW et OLD peuvent être utilisés pour faire référence aux valeurs de la table référencée.
  - NEW peut être utilisé dans les règles ON INSERT et ON UPDATE pour faire référence à la nouvelle ligne
  - OLD est utilisé dans les règles ON UPDATE et ON DELETE pour référencer la ligne existant avant modification ou suppression.

Règles Postgresql - (3)

- Exemple 1 :

```
1 CREATE OR REPLACE RULE "Regle1"
2 AS ON UPDATE TO Mavue_Pilote
3 DO INSTEAD UPDATE Pilote
4 SET salaire = new.salaire WHERE nom = new.nom;
```

- Exemple 2 :

```
1 CREATE OR REPLACE RULE "rule_InfoPersonne"
2 AS ON INSERT TO Info_Personnel
3 DO INSTEAD INSERT INTO Personne
4 VALUES (new.Personne_ID, 1, new.Nom,
5 new.Prenom,  NULL, NULL, NULL, new.Email);
```

Le langage procédural

- SQL est un langage déclaratif non procédural : Chaque expression SQL (Requête naturelle) doit être exécutée individuellement par le serveur de bases de données.
- Les traitements complexes exigent l'utilisation des **variables** et des **structures de programmation** comme les boucles et les alternatives
- Besoin d'un **langage procédural** pour lier plusieurs requêtes SQL avec des variables et dans les structures de programmation habituelles
  - Extension de SQL : des requêtes SQL cohabitent avec les structures de contrôle habituelles de la programmation structurée (blocs, alternatives, boucles)
  - Des variables permettent l'échange d'information entre les requêtes SQL et le reste du programme
- PL/SQL = PROCEDURAL LANGUAGE/SQL
  - peut être utilisé pour l'écriture des procédures stockées et des déclencheurs (Triggers)

PL/SQL

- PL/SQL est un langage de programmation propre à Oracle
- Existe dans d'autres SGBDR
  - MySQL : PL/SQL like
  - Sybase et Microsoft SQL server : Transact-SQL
  - PostgreSQL : PL/pgSQL
  - DB2 (IBM) : SQL Procedural Language
- Il permet de :
  - Créer des fonctions utilisateurs et triggers,
  - Ajouter des structures de contrôle au langage SQL,
  - Effectuer des traitements complexes,
  - Gérer les exceptions

PostgreSQL et langage procédural

- PostgreSql permet d'utiliser plusieurs 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 de PostgreSql .
  - http ://www.commandprompt.com/community/plphp
- Langage PL/Java
  - pljava un PL-SQL qu'il faut installer en plus de PostgreSql .

Activation d'un PL-SQL

- Activation d'un PL-SQL de PostgreSql
  - CREATE LANGUAGE

```
Postgres-# CREATE LANGUAGE plpgsql ;
```

  - Createlang

```
$createlang plpgsql base
```

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

```
GRANT ... ON LANGUAGE ...
```

Création et structure d'une fonction

- Déclaration 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.

Exemples

- Création d'une fonction avec Langage pgSql

```
1 CREATE OR REPLACE FUNCTION prixTTC (prixHT numeric)
2 RETURNS numeric AS $$
3 BEGIN
4   RETURN  prixHT * 1.196;
5 END; $$ LANGUAGE 'plpgsql' ;
```

- Création d'une fonction avec Langage PL / SQL

```
1 CREATE OR REPLACE FUNCTION prixTTC (numeric)
2 RETURNS numeric AS $$
3         SELECT $1 * 1.196;
4 $$ LANGUAGE 'SQL';
```

  - Pas de BEGIN et de END en PL / SQL

- Usages
  - 1ère forme : target-list → SELECT prixTTC(10);
  - 2ème forme : FROM clause → SELECT * FROM prixTTC(10);
  - 3ème forme : WHERE clause → SELECT * FROM vente WHERE prixTTC(10) = vente.total;

Section DECLARE : Déclaration des variables

- Toutes les variables utilisées dans un bloc doivent être déclarées dans la section déclaration du bloc ;
  - Sauf la variable d'une boucle `FOR` déclarée automatiquement comme variable de type `integer`.
- Syntaxe générale d'une déclaration de variable

```
nom [CONSTANT] type [NOT NULL][{DEFAULT | := } expression ];
```

  - `DEFAULT` : valeur initiale assignée à la variable. Si la clause `DEFAULT` n'est pas indiquée, la variable est initialisée à `NULL`.
  - `CONSTANT` empêche l'assignation de la variable, de sorte que sa valeur reste constante pour la durée du bloc
- Déclaration multiple interdite
- Exemples
  - `quantité integer DEFAULT 32;`
  - `url varchar := 'http://mysite.com';`
  - `id_utilisateur CONSTANT integer := 10;`

Le type de variables

- Les types simples sont les types standards SQL :`boolean, text, char, varchar, integer, float, date,time`.
- Ces types sont complétés par :
  - le type `RECORD`.
    - `nom RECORD;`
  - `variable%TYPE` : le copie de types (que le type de colonne référencée)
    - `%TYPE` fournit le type de données d'une variable ou d'une colonne de table.
    - En utilisant `%TYPE` on n'a pas besoin de connaître le type de données de la structure à laquelle on fait référence et, plus important, si le type de données de l'objet référencé change dans le futur.
    - `New_champ utilisateurs.id_utilisateur%TYPE;`
  - Types ligne : `nom nom_table%ROWTYPE` ⇒ Variable de type composite : appelée variable ligne (ou variable row-type).
    - `v_employe emp%ROWTYPE;` ⇒ déclare que la variable `v_employe` contiendra une ligne de la table `emp`

Opérateurs

- Arithmétique : `+, -, *, /`
- Concaténation de chaîne : `||`
- Comparaison : `=, <>, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN`
- Logique : `AND, OR, NOT`
- Affectation `:=`
  - Parenthèses `( )` pour contrôler les priorités des opérations
- Affectation d'une valeur à une variable :
  - `<nom_de_variable> := <expression>`
    - « = » ou « := » au choix !
- faire un **cast** et changer le type d'une expression
  - `expression::type`
  - `CAST(expression) AS type`

Affectation des variables

- Affectation par la directive INTO de la requête SELECT
  - SELECT INTO Variable Attribut FROM relation [WHERE ...] ;

```
SELECT INTO v_nom  nom FROM emp WHERE matr = 509;
```

- SELECT INTO V1, V2, ,Vn A1,A2, ,An from relation [where .....] ;
  - les variables V1,V2, ,Vn reçoivent les valeurs des attributs projetés A1,A2, ,An

```
SELECT INTO v_nom, v_sal nom, salaire FROM emp WHERE matr = 509;
```

- Le résultat d'une commande SELECT manipulant plusieurs colonnes (mais une seule ligne) peut être assignée à une variable de type RECORD ou ligne (%ROWTYPE ou une liste de valeurs scalaires.

```
SELECT INTO v_enreg * FROM emp WHERE matr = 509;
V_enreg.nom
```

Types de paramètres et de résultats

- Les fonctions PL/PGSQL peuvent avoir pour arguments tous types supportés par le serveur.
- Toute table, ou colonne est considéré comme un type. Il est possible d'utiliser des types composites à partir de ces objets grâce à %TYPE et %ROWTYPE.
- Le PL/PGSQL permet aussi d'utiliser des types polymorphes, déclarés comme **ANYELEMENT** et **ANYARRAY**.
- Le type RECORD indique un tuple dont la composition (colonnes) est précisé à l'appel de la fonction. Il est utilisé pour retourner les tuples.
  - Il ne peut être utilisé en arguments d'une fonction.

Alias de paramètres de fonctions - 1

- Les paramètres passés aux fonctions sont nommés par les identifiants `$1, $2, etc`.

```
1 CREATE FUNCTION taxe_ventes(real)
2 RETURNS real AS $$
3 BEGIN
4  RETURN $1* 0.06;
5 END $$ LANGUAGE 'plpgsql' ;
```

- Des alias peuvent être déclarés, éventuellement, pour les noms de paramètres de type `$n` afin d'améliorer la lisibilité.
  - L'alias ou l'identifiant numérique peuvent être utilisés indifféremment pour se référer à la valeur du paramètre.

Alias de paramètres de fonctions - 2

- Il existe deux façons de créer un alias :
  1. La façon préférée est de donner un nom au paramètre dans la commande

```
1 CREATE FUNCTION taxe_ventes(total real)
2 RETURNS real AS $$
3 BEGIN
4  RETURN total * 0.06;
5 END $$ LANGUAGE 'plpgsql' ;
```

  2. L'autre façon, la seule disponible pour les versions antérieures de PostgreSql 8.0, est de déclarer explicitement un alias en utilisant la syntaxe de déclaration
     - `nom ALIAS FOR $n;`

```
1 CREATE FUNCTION taxe_ventes(real)
2 RETURNS real AS $$
3 DECLARE
4 sous_total ALIAS FOR $1;
5 BEGIN
6  RETURN sous_total * 0.06;
7 END $$ LANGUAGE 'plpgsql' ;
```

Paramètres de fonctions

- Les paramètres peuvent être utilisés uniquement pour passer des données, jamais pour passer les noms de tables.
- Par exemple
  - `INSERT INTO mytable VALUES ($1);` → est correct
  - `INSERT INTO $1 VALUES (42);` → n'est pas correct
- Le mode d'un paramètre :
  - soit `IN` (entrée),
  - soit `OUT` (sortie),
  - soit `INOUT`(entrée/sortie).
    - En cas d'omission, la valeur par défaut est `IN`.

Instruction return

- `RETURN` accompagné d'une expression termine la fonction et renvoie le valeur de l'expression à l'appelant.
  - A utiliser également avec des fonctions `PL/pgSQL` qui ne renvoient pas d'ensemble de valeurs.
- Le résultat de l'expression sera automatiquement converti vers le type de retour de la fonction, comme décrit pour les affectations.
- Si on déclare la fonction avec des paramètres en sortie, écrire seulement `RETURN` sans expression. Les valeurs courantes des paramètres en sortie seront renvoyées
- Si on déclare que la fonction renvoie `VOID`, une instruction `RETURN` peut être utilisée pour quitter rapidement la fonction
  - mais ne pas écrire d'expression après `RETURN`.

Valeur de retour d'une fonction - 1

- Pas de retour → `RETURNS VOID`

```
1 CREATE FUNCTION logfunc(logtxt text)
2 RETURNS VOID AS $$
3 DECLARE
4  curtime timestamp := now();
5 BEGIN
6  INSERT INTO logtable VALUES (logtxt, curtime);
7 END; $$ LANGUAGE 'plpgsql';
```

- Tous les types simples : `int, text, numeric, ...`

```
1 CREATE OR REPLACE FUNCTION increment(i integer)
2 RETURNS integer AS $$
3 BEGIN
4  RETURN i + 1;
5 END; $$ LANGUAGE plpgsql;
```

Valeur de retour d'une fonction - 2

- Retour d'enregistrement
  - RETURNS RECORD→ pour tout enregistrement → enregistrement générique. Attention à l'usage
  - RETURNS nomTable→ enregistrement correspondant à une table
- Exemple

```
1 CREATE OR REPLACE FUNCTION testRecord(v_nl integer)
2 RETURNS RECORD AS $$
3 DECLARE
4 res RECORD;
5 BEGIN
6  SELECT INTO res l.nl, l.editeur,o.* FROM livres l, oeuvres o
7 WHERE l.no=o.no and nl=v_nl;
8 RETURN res;
9 END $$ LANGUAGE 'plpgsql'
```

Valeur de retour d'une fonction - 3

- Retour d'un ensemble de valeurs
  - `RETURNS SETOF typeSimple`
  - `RETURNS SETOF RECORD`
  - `RETURNS SETOF nomTable`
- Lorsqu'une fonction `PL/pgSQL` est déclarée renvoyer `SETOF type quelconque` les éléments individuels à renvoyer sont spécifiés dans les commandes `RETURN NEXT`, et ensuite une commande `RETURN` finale sans arguments est utilisée pour indiquer que la fonction a terminé son exécution.
- Les fonctions qui utilisent `RETURN NEXT` devraient être appelées d'après le modèle suivant :
  - `SELECT * FROM une_fonction();`

Valeur de retour d'une fonction - 4

- Exemple

```
1  CREATE TYPE numtype AS (num int, doublenum int);
2
3  CREATE OR REPLACE FUNCTION GetNum(int)
4  RETURNS SETOF numtype AS $$
5  DECLARE
6   r numtype%rowtype;
7   i int;
8  BEGIN
9   FOR i IN 1 .. $1 LOOP
10   r.num := i;
11   r.doublenum := i*2;
12  RETURN NEXT r;
13 END LOOP;
14  RETURN;
15  END $$ LANGUAGE 'plpgsql';
```

Instruction RAISE

- Utilisez l'instruction `RAISE` pour rapporter des messages et lever des erreurs.
- `RAISE TypeRaise 'chaine de formatage', variables;`
  - `TypeRaise` : `NOTICE, INFO, EXCEPTION, WARNING, DEBUG, LOG`
    - `NOTICE` → affichage simple
    - `INFO` → affiche le contexte en plus (qui a appelé la fonction).
    - `EXCEPTION` → pour gérer les exceptions.
  - Au sein de la chaîne de formatage, `%` est remplacé par la valeur de la variable optionnelle suivante.
- Exemple

```
RAISE NOTICE 'quantité vaut ici %', vente.quantite;
```

Structures conditionnelles

- Les instructions `IF` et `CASE` vous permettent d'exécuter des commandes basées sur certaines conditions.
- `PL/pgSQL` a cinq formes de `IF` :
  - `IF ...THEN`
  - `IF ...THEN ...ELSE`
  - `IF ...THEN ...ELSE IF`
  - `IF ...THEN ...ELSIF ...THEN ...ELSE`
  - `IF ...THEN ...ELSEIF ...THEN ...ELSE`
- et deux formes de `CASE` :
  - `CASE ...WHEN ...THEN ...ELSE ...END CASE`
  - `CASE WHEN ...THEN ...ELSE ...END CASE`

Structures conditionnelles

- Exemple 1

```
IF v_id_utilisateur <> 0 THEN
        UPDATE utilisateurs SET email = v_email
        WHERE id_utilisateur = v_id_utilisateur;
END IF;
```

- Exemple 2

```
IF var1 < 0 THEN
        var2 := -1;
ELSIF var1 < 10 THEN
 var2 := 1;
ELSE
 var2 := null;
END IF;
```

- Exemple 3

```
CASE
WHEN x BETWEEN 0 AND 10 THEN
 msg := 'valeur entre zéro et dix';
WHEN x BETWEEN 11 AND 20 THEN
 msg := 'valeur entre onze et vingt';
END CASE;
```

Boucles simples - 1

- Boucle LOOP

```
[<<label>>]
 LOOP
  instructions
         EXIT [label] [WHEN expression-booléenne ];
 END LOOP [label];
```

  - Cette instruction boucle jusqu'à rencontrer l'instruction EXIT.
  - Cette dernière peut être conditionnée par WHEN qui précise quand la sortie de boucle doit s'effectuer.

- Exemple

```
1 i := 1;
2 LOOP
3 i := i+1;
4 EXIT WHEN i=10;
5 END LOOP;
```

Boucles simples - 2

- Boucle WHILE

```
[<<label>>]
 WHILE expression-booléenne
  LOOP instructions
  END LOOP [ label ];
```

  - L'instruction WHILE répète une séquence d'instructions aussi longtemps que expression-booléenne est évaluée à vrai.
  - L'expression est vérifiée juste avant chaque entrée dans le corps de la boucle.

- Exemple

```
1 WHILE emp.salaire <1350 AND emp.job ='aide' LOOP
2 -- quelques traitements ici
3 END LOOP;
```

Boucles simples - 3

- Boucle FOR

```
[<<label>>]
FOR nom IN [REVERSE ] expression1 .. Expression2 [BY expression]
LOOP
instruction
END LOOP [ label ];
```

  - nom est une variable de type entier créée automatiquement en début de boucle puis détruite à la fin. Il n'est pas nécessaire de la déclarer explicitement (sous DECLARE).
  - expression1 et expression2sont des expressions entières donnant les limites inférieures et supérieures de la plage et qui sont évaluées une fois en entrant dans la boucle
  - Si la clause BY n'est pas spécifiée, l'étape d'itération est de 1, sinon elle est de la valeur spécifiée dans la clause BY.
  - Si REVERSE est indiquée, alors la valeur de l'étape est soustraite, plutôt qu'ajoutée, après chaque itération.

Boucles simples - 4

- Exemple

```
1 FOR i IN 1 .. n LOOP
2 total := total + tab[i];
3 END LOOP;
4 --
5 FOR j IN REVERSE n .. 1 LOOP
6 total := total + tab[j];
7 END LOOP;
```

Boucles simples - 4

- Instruction CONTINUE

```
CONTINUE [ label ] [ WHEN expression-booléenne ];
```

  - Si aucun label n'est donné, la prochaine itération de la boucle interne est commencée. C'est-à-dire que toutes les instructions restantes dans le corps de la boucle sont ignorées et le contrôle revient à l'expression de contrôle de la boucle pour déterminer si une autre itération de boucle est nécessaire. Si le label est présent, il spécifie le label de la boucle dont l'exécution va être continuée.
  - Si WHEN est spécifié, la prochaine itération de la boucle est commencée seulement si l'expression-booléenne est vraie.

- Exemple

```
1 LOOP
2 -- quelques traitements
3 EXIT WHEN nombre > 100;
4 CONTINUE WHEN nombre < 50;
5 -- quelques traitements pour nombre IN [50 .. 100]
6 END LOOP;
```

Boucler dans les résultats de requêtes

```
[<<label>>]
FOR cible IN requête LOOP
instructions
END LOOP [ label ];
```

- La `cible` est une variable de type `RECORD, ROWTYPE` ou une liste de variables scalaires séparées par une virgule.
- La `cible` est affectée successivement à chaque ligne résultant de la ***requête*** et le corps de la boucle est exécuté pour chaque ligne.

```
1  CREATE OR REPLACE FUNCTION verifierToutesLesLivraisons()
2  RETURNS void AS $$
3  DECLARE
4   e record;
5  BEGIN
6  FOR e IN select bl.id from bonDeCommande bc
7  inner join bonDeLivraison bl on bc.id=bl.idCommande
8  where bl.dateLivraison<=bc.dateCommande
9  LOOP
10  RAISE NOTICE 'la livraison n% est invalide',e.id;
11 END LOOP; RETURN; END $$ language 'plpgsql';
```

Boucler dans les résultats de requêtes - 2

- Exemple 1

```
DROP FUNCTION test(integer);
CREATE OR REPLACE FUNCTION test(prix_oeuvre integer)
RETURNS SETOF oeuvres AS $$
DECLARE
 resultat oeuvres%ROWTYPE ;
BEGIN
 FOR resultat IN SELECT * FROM oeuvres
 WHERE Prix >= prix_oeuvre
        LOOP
         RETURN NEXT resultat;
   END LOOP;
  RETURN;
 END $$ LANGUAGE 'plpgsql' ;
```

Boucler dans les résultats de requêtes - 2

- Exemple 2

```
CREATE OR REPLACE FUNCTION forin(a ingredients.unit%TYPE)
RETURNS DECIMAL(5,3) AS $$
DECLARE
toto RECORD; -- noter une variable de type RECORD
prix ingredients.unitprice%TYPE;
s REAL DEFAULT 0; -- initialisation avec DEFAULT
i INTEGER := 0 ; -- initialisation avec :=
BEGIN
 FOR toto IN SELECT name, unitprice FROM ingredients
 WHERE unit = a
  LOOP
   s := s + toto.unitprice;
   i := i + 1 ;
  END LOOP;
IF i = 0 THEN
 RETURN 0;
END IF;
RETURN (s/i)::DECIMAL(5,3) ; --noter un cast
END $$ LANGUAGE 'plpgsql';
```

Executer le code SQL à l'intérieur d'une fonction

- On peut mettre le code SQL directement dans les fonctions PL/pgSQL : CREATE TABLE, INSERT, UPDATE, DELETE, etc. sauf SELECT et sauf l'appel à une autre fonction si le résultat de SELECT ou de l'appel n'est pas récupéré.
- PERFORM peut être utilisé pour évaluer une expression SQL sans récupération de résultat
  - PERFORM requete-select ;

```
PERFORM * FROM ingredients WHERE unitprice < 5;
-- PERFORM remplace le mot SELECT.
PERFORM  prixTTC (50);
```

- L'instruction EXECUTE permet d'exécuter une requête contenue dans une chaîne de caractères. Cela permet de faire de requêtes dynamiques qui ne sont pas précompilées.
  - EXECUTE command-string [INTO[STRICT]target] [USING expression [, ...]];

```
1 EXECUTE 'SELECT count(*) FROM matable
2 WHERE insere_par = $1 AND insere <= $2' INTO c
3 USING utilisateur_verifie, date_verifiee;
```

Obtention du statut du résultat

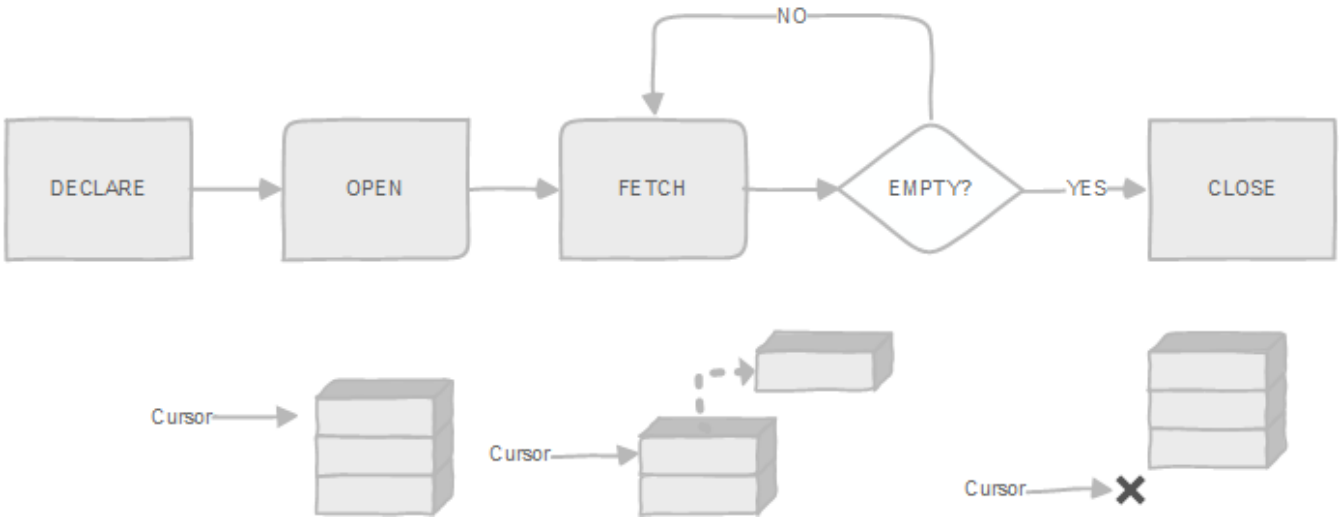
- La variable `FOUND` de type boolean est initialisée à false au début de chaque fonction PL/pgSQL
- Elle est positionnée par chacun des types d'instructions suivants :
  - Une instruction `SELECT INTO` positionne `FOUND` à true si elle renvoie une ligne, false si aucune ligne n'est renvoyée.
  - Une instruction `PERFORM` positionne `FOUND` à true si elle produit une ligne, false si aucune ligne n'est produite.
  - Les instructions `UPDATE, INSERT, et DELETE` positionnent `FOUND` à true si au moins une ligne est affectée, false si aucune ligne n'est affectée.
  - La commande `FOR` positionne `FOUND` à true si elle effectue une itération une ou plusieurs fois, sinon elle renvoie false.

Exemple d'utilisation de FOUND

```
1  CREATE OR REPLACE FUNCTION intot(ingredients.ingredientid%TYPE)
2  RETURNS ingredients.unit%TYPE AS $$
3  DECLARE
4  a ingredients%ROWTYPE; --Declaration d'une variable du type:
5  --ligne de la table ingredients
6  BEGIN
7  -- Maintenant on peut faire SELECT dans a:
8  SELECT * INTO a
9  FROM ingredients
10 WHERE $1 = ingredients.ingredientid ;
11 IF FOUND THEN --FOUND est TRUE si le select
12 --a trouve des lignes
13 RETURN a.unit;
14 ELSE
15 RETURN 'unknown' ;
16 END IF;
17 END;
18 $$ LANGUAGE 'plpgsql';
```

Les Curseurs - 1

- Un curseur est un pointeur sur une ligne d'un select qui permet de manipuler les résultats de requêtes ligne par ligne..
- Un `FETCH` permet de récupérer la ligne pointée par un curseur et de déplacer le curseur.
- La gestion des curseurs s'apparente à la gestion de fichiers en lecture.



Les Curseurs - 2

- Un des moyens de créer une variable curseur est de simplement la déclarer comme une variable de type `refcursor`.
- Un autre moyen est d'utiliser la syntaxe de déclaration de curseur qui est en général :

```
nom CURSOR [ ( arguments ) ] FOR requete ;
```

- Exemple

```
DECLARE
    curs1 refcursor;
    curs2 CURSOR FOR SELECT * FROM tenk1;
    curs3 CURSOR (cle integer) IS SELECT * FROM tenk1
    WHERE unique1 = cle;
```

- Les curseurs associés à une requête sont qualifiés de curseur lié (`bound_cursor`)

Les Curseurs - 3

- Ouvrir un curseur non lié :

```
1 OPEN unbound_cursor FOR query;
```

- Ouvrir un curseur lié :

```
1 OPEN bound_cursor [(arg)];
```

- Manipuler une ligne de résultat :

```
1 FETCH [ direction { FROM | IN } ] curseur INTO cible;
```

  - FETCH avance le curseur sur la ligne suivante, la place dans une cible, qui peut être une variable ligne, une variable record ou une liste de variables et indique via FOUND si elle existe.

- Fermer un curseur et libérer les ressources associées

```
1 CLOSE curseur;
```

Les Curseurs - 4

- Exemple

```
1  CREATE OR REPLACE FUNCTION fct() RETURNS void AS $$
2  DECLARE
3      rec RECORD;
4      curs refcursor;
5  BEGIN
6      EXECUTE 'CREATE TABLE temp (idliv integer, titre varchar(50))';
7      OPEN curs FOR SELECT liv_num, liv_titre FROM livre;
8      LOOP
9          FETCH curs INTO rec;
10         EXIT WHEN NOT FOUND;
11         INSERT INTO temp VALUES(rec.liv_num,rec.liv_titre);
12     END LOOP;
13     CLOSE curs;
14 END;
15 $$ LANGUAGE plpgsql;
```

Les Curseurs - 5

```
1  CREATE OR REPLACE FUNCTION get_film_titles(p_year INTEGER)
2     RETURNS text AS $$
3  DECLARE
4   titles TEXT DEFAULT '';
5   rec_film   RECORD;
6   cur_films CURSOR(p_year INTEGER)
7   FOR SELECT title, release_year
8   FROM film
9   WHERE release_year = p_year;
10 BEGIN
11    -- Open the cursor
12    OPEN cur_films(p_year);
13     LOOP
14     -- fetch row into the film
15       FETCH cur_films INTO rec_film;
16     -- exit when no more row to fetch
17       EXIT WHEN NOT FOUND;
18      -- build the output
19       IF rec_film.title LIKE '%ful%' THEN
20          titles := titles || ',' || rec_film.title ||
21       ':' || rec_film.release_year;
22       END IF;
23    END LOOP;
24      -- Close the cursor
25    CLOSE cur_films;
26     RETURN titles;
27 END; $$
```

Gestion des exceptions

- Par défaut, toute erreur survenant dans une fonction PL/pgSQL annule l'exécution de la fonction mais aussi de la transaction qui l'entoure.
- On peut récupérer les erreurs en utilisant un bloc avec une clause EXCEPTION.

```
1 BEGIN
2  instructions
3 EXCEPTION
4  WHEN condition [ OR condition ... ] THEN
5          instructions_gestion_erreurs
6 [ WHEN condition [ OR condition ... ] THEN
7          instructions_gestion_erreurs ... ]
8 END;
```

- Si une erreur survient à l'intérieur des instructions, le traitement en cours des instructions est abandonné et le contrôle est passé à la clause EXCEPTION.

Gestion des exceptions

- Large spectre de conditions d'erreurs
- Pas de possibilité de définir es propres exceptions
- Les conditions d'erreurs identifiées par le SGBD PGSQL sont répertoriées dans des catégories qui sont consultables dans l'appendix de la documentation de ce dernier
  - ex. `https://www.postgresql.org/docs/11/errcodes-appendix.html`
- Quelques conditions d'erreurs :

  `division_by_zero, privilege_not_granted, integrity_co`

```
1 BEGIN
2 i := j/k;
3 EXCEPTION
4   WHEN division_by_zero THEN
5     RAISE NOTICE ' What!!! i=% j=% k=%', i, j, k;
6 RETURN NULL;
7 -- sinon exception non levée
8 END;
```

Gestion des exceptions

```
1  -- Module(#nomModule: VARCHAR(20), nbHeures: INTEGER)
2  CREATE FUNCTION printNbHeuresModules(VARCHAR(20)) RETURNS void AS
3  $print HeuresMod$
4  DECLARE
5  Heures Module.nbHeures%TYPE;
6  BEGIN
7  -- NbHeure du module dont l'identifiant est passé en paramètre
8  SELECT nbHeures INTO Heures
9  FROM Module WHERE nomModule = $1;
10 RAISE NOTICE 'Nombre de heures de %: %', $1, Heures;
11 EXCEPTION
12 WHEN no_data_found THEN
13 RAISE no_data_found
14 USING MESSAGE = 'Module % inexistant dans la base:' || $1;
15 END; $print HeuresMod$ LANGUAGE plpgsql;
```

- Un bloc contenant une clause EXCEPTION est significativement plus coûteux en entrée et en sortie qu'un bloc sans. Du coup, ne pas utiliser EXCEPTION sans besoin.

Les déclencheurs - Triggers

- Un déclencheur (`trigger`) est un script qui se déclenche sur certains évènements de la base de données.
- Les triggers se définissent sur les objets de type `TABLE`.
- Ils se rapportent aux évènements suivants : `INSERT, UPDATE, DELETE`
- Ils peuvent être déclenchés avant (`BEFORE`) ou après (`AFTER`) un évènement donné.
- Un trigger est détruit avec la destruction d'une table et peut être désactivé.
- Lors de l'exécution d'un trigger, des variables spéciales sont créées automatiquement : `NEW, OLD, ...`
- Après la définition de la fonction, il faut encore définir le trigger lui-même avec `CREATE TRIGGER`

Les déclencheurs - Triggers - 1

- Un déclencheur (`trigger`) est un script qui se déclenche sur certains évènements de la base de données.
- Les triggers se définissent sur les objets de type `TABLE`.
- Ils se rapportent aux évènements suivants : `INSERT, UPDATE, DELETE`
- Ils peuvent être déclenchés avant (`BEFORE`) ou après (`AFTER`) un évènement donné.
- Un trigger est détruit avec la destruction d'une table et peut être désactivé.
- Lors de l'exécution d'un trigger, des variables spéciales sont créées automatiquement : `NEW, OLD, ...`
- Après la définition de la fonction, il faut encore définir le trigger lui-même avec `CREATE TRIGGER`

Les déclencheurs - Triggers - 2

- La mise place une procédure déclenchée se fait en deux temps :
  1. définition d'une fonction réalisant les opérations réalisées automatiquement lors de l'observation des événements déclencheurs ;

```
1 CREATE FUNCTION nom_fonc() RETURNS TRIGGER AS $$
2  ...
3  $$ LANGUAGE plpgsql;
```

     - Noter que la fonction doit être déclarée avec aucun argument même si elle s'attend à recevoir les arguments spécifiés dans CREATE TRIGGER.

  2. définition du déclencheur lui-même avec CREATE TRIGGER

```
1 CREATE TRIGGER nomTrigger
2   { BEFORE | AFTER }      -- moment du déclenchement
3   { event [ OR ... ] }    -- événements concernés
4   ON nomTable             -- table concernée
5   [ FOR [ EACH ] { ROW | STATEMENT } ]
6   -- modalité d'exécution des opérations automatiques
7   EXECUTE PROCEDURE nomFonction(arguments)
8   -- appel de la fonction gérant les opérations automatiques
```

Les déclencheurs - Triggers - 3

- Définition de la fonction trigger

```
1  CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
2  BEGIN
3  -- Verifie que nom_employe et salary sont donnés
4  IF NEW.nom_employe IS NULL THEN
5  RAISE EXCEPTION 'nom_employe ne peut pas être NULL';
6  END IF;
7  IF NEW.salaire IS NULL THEN
8  RAISE EXCEPTION '% ne peut pas avoir un salaire', NEW.nom_employe;
9  END IF;
10 -- Qui travaille pour nous si la personne doit payer pour cela ?
11 IF NEW.salaire < 0 THEN
12 RAISE EXCEPTION '% ne peut pas avoir un salaire négatif',
13 NEW.nom_employe;
14 END IF;
15 -- Rappelons-nous qui a changé le salaire et quand
16 NEW.date_dermodif := current_timestamp;
17 NEW.update_user  := current_user;
18 RETURN NEW;
19 END;
20 $emp_stamp$ LANGUAGE plpgsql;
```

- Création du trigger

```
1  CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
2  FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```

Les déclencheurs - Triggers - 3

- Quand une fonction PL/pgSQL est appelée en tant que trigger, plusieurs variables spéciales sont créées automatiquement dans le bloc de plus haut niveau.
  - `NEW` : un enregistrement correspondant à une ligne en insertion ou modification.
  - `OLD` : un enregistrement correspondant à une ligne en destruction ou modification.
  - `TG_NAME` : type de trigger en cours d'exécution.
  - `TG_WHEN` : BEFORE ou AFTER
  - `TG_LEVEL` : ROW ou STATEMENT
  - `TG_OP` : INSERT, UPDATE ou DELETE
  - `TG_RELID` : identifiant de l'objet ayant déclenché le trigger.
  - `TG_RELNAME` : nom de la table ayant provoqué le trigger.
  - `TG_NARGS` : nombre d'arguments
  - `TG_ARGV[]` : tableau d'arguments

Les déclencheurs - Triggers - 3

- Définition de la fonction trigger

```
1 CREATE FUNCTION trace() RETURNS TRIGGER AS $$
2 BEGIN
3 RAISE NOTICE '% % % on %',
4 TG_OP, TG_WHEN, TG_LEVEL, TG_RELNAME;
5 RETURN NEW;
6 END;
7 $$ LANGUAGE plpgsql;
```

- Création du trigger

```
1 CREATE TRIGGER trace_entiers BEFORE INSERT OR UPDATE
2 ON entiers FOR EACH STATEMENT EXECUTE PROCEDURE trace();
```

- Résultat

```
1 INSERT entiers(i) VALUES(1);
2 -- NOTICE: INSERT BEFORE STATEMENT on entiers
3 -- INSERT 0 1
```

Les déclencheurs - Triggers - 3

- Définition de la fonction trigger

```
1  CREATE OR REPLACE FUNCTION NbLivresEmpruntes()
2  RETURNS trigger AS $$
3  DECLARE
4  NbLivres integer;
5  BEGIN
6  SELECT count(*) INTO nbLivres
7  FROM emprunter
8  WHERE NA=NEW.NA
9  AND dateRet is NULL;
10 RAISE INFO '% livres actuellement emprunté par l''adhérent %',
11 nbLivres, NEW.NA ;
12 RETURN NULL ;
13 END ; $$ LANGUAGE PLPGSQL ;
```

- Création du trigger

```
1
2  CREATE TRIGGER NbLivresEmpruntes
3  AFTER INSERT ON emprunter
4  FOR EACH ROW
5  EXECUTE PROCEDURE NbLivresEmpruntes();
```

Les déclencheurs - Triggers - 3

```
1  CREATE OR REPLACE FUNCTION audit_proc()
2  RETURNS trigger AS $$
3  DECLARE
4      arg_num integer;
5      arg_color varchar;
6  BEGIN
7      arg_num   := TG_ARGV[0];
8      arg_color := TG_ARGV[1];
9
10     IF NEW.id > arg_num THEN
11        raise notice 'new record id % is greater than
12 control number %', NEW.id, arg_num;
13     ELSE
14        raise notice 'new record id % is less than or equal
15 to control number %', NEW.id, arg_num;
16     END IF;
17      IF NEW.color IS NOT NULL AND NEW.color != arg_color THEN
18      raise exception 'new color % is not equal
19 to control color %', NEW.color, arg_color;
20     END IF;
21     NEW.update_date := current_timestamp;
22     NEW.update_user := current_user;
23     return NEW;
24 END;$$ LANGUAGE plpgsql;
```

- Création du trigger

```
1  CREATE TRIGGER trig_audit
2     BEFORE INSERT OR UPDATE ON bar
3     FOR EACH ROW
4     EXECUTE PROCEDURE audit_proc(100, 'blue');
```