BASES DE DONNÉES
COURS 3
FONCTIONS ET CONTRAINTES DYNAMIQUES

Mickaël Coustaty

Jean-Loup Guillaume

Laboratoire Informatique Image Interaction (L3I)
Université de La Rochelle - Pôle Sciences et Technologie - Avenue Michel Crépeau - 17042 LA ROCHELLE CEDEX 1 France
Tél : +33 (0)5 46 45 82 62 – Fax : 05.46.45.82.42 – Site internet : http://l3i.univ-larochelle.fr/

OBJECTIFS DU COURS 3

Maîtriser l'écriture de fonctions en plpgsql

- Fonctions simples
- Fonctions sur les tables
- Curseurs, règles et triggers pour automatiser les traitements

PROGRAMMATION EN SQL

Objectif : écrire des fonctions exécutables dans une requête SQL

- Une fonction est un ensemble de calculs que l'on veut exécuter
- Les calculs sont des choses que l'on pourrait faire en SQL

Exemple :

```
SELECT 1 + 2;
```

- Requête SQL qui calcule la somme de 1 et 2 et retourne 3
- Ou plutôt retourne un enregistrement avec un attribut de type integer

```
CREATE FUNCTION ajoute12() RETURNS integer
AS 'SELECT 1 + 2;' LANGUAGE SQL;
```

- Fonction écrite en SQL qui retourne un entier

```
SELECT ajoute12();
```

- Exécute la fonction dans un requête SQL et retourne l'enregistrement

PROGRAMMATION EN SQL

Les langages procéduraux permettent d'effectuer des traitements plus complexes :

- Instructions conditionnelles (if then else)
- Instructions répétitives (boucles)

En pratique les nouvelles versions de SQL permettent de faire plus :

- Définitions de vues locales (Common Table Expression)
- Appels récursifs (norme 1999)
- SQL est depuis aussi puissant que n'importe quel langage

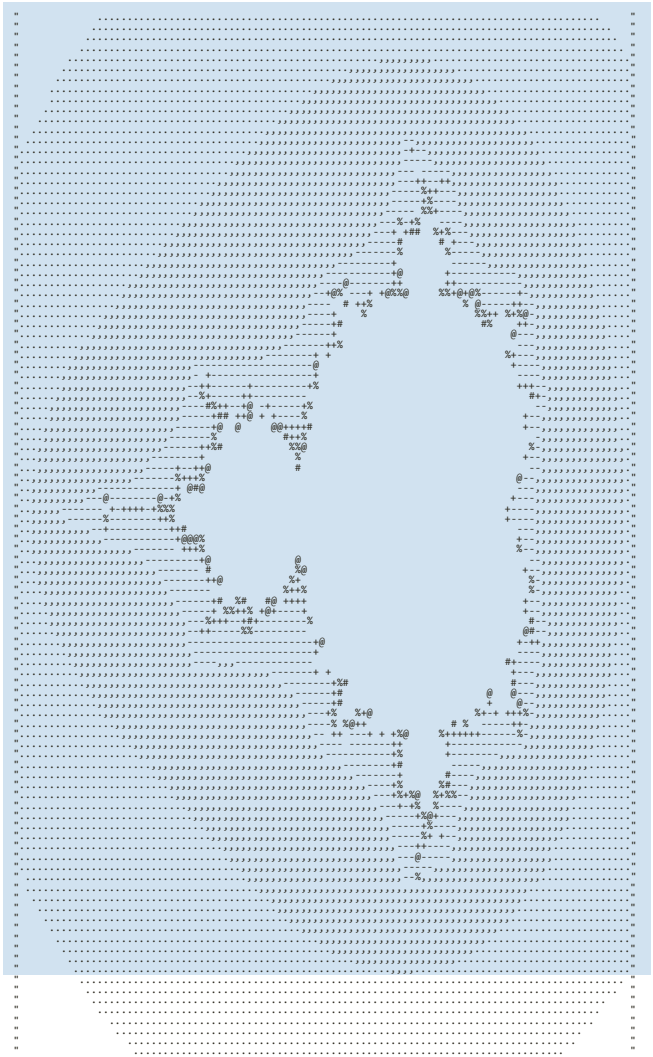
Mais on pratique on préfère utiliser des langages traditionnels

- Langages traditionnels : **SQL**, C
- Langages procéduraux : **PL/pgSQL**, PL/Python, PL/Perl

EXEMPLE DE CALCUL RÉCURSIF

```
WITH RECURSIVE
x(i)
AS (
    VALUES(0)
UNION ALL
    SELECT i + 1 FROM x WHERE i < 101
),
Z(Ix, Iy, Cx, Cy, X, Y, I)
AS (
    SELECT Ix, Iy, X::FLOAT, Y::FLOAT, X::FLOAT, Y::FLOAT, 0
    FROM
        (SELECT -2.2 + 0.031 * i, i FROM x) AS xgen(x,ix)
    CROSS JOIN
        (SELECT -1.5 + 0.031 * i, i FROM x) AS ygen(y,iy)
    UNION ALL
    SELECT Ix, Iy, Cx, Cy, X * X - Y * Y + Cx AS X, Y * X * 2 + Cy, I + 1
    FROM Z
    WHERE X * X + Y * Y < 16.0
    AND I < 27
),
Zt (Ix, Iy, I) AS (
    SELECT Ix, Iy, MAX(I) AS I
    FROM Z
    GROUP BY Iy, Ix
    ORDER BY Iy, Ix
)
SELECT array_to_string(
    array_agg(
        SUBSTRING(
            ' .,,,-----++++%%%%@@@@#### ',
            GREATEST(I,1),
            1
        )
    ),''
)
FROM Zt
GROUP BY Iy
ORDER BY Iy;
```

ET LE RÉSULTAT…



PROCEDURAL LANGUAGE/POSTGRESQL

Objectifs de PL/pgSQL

- Effectuer des opérations et calculs plus complexes qu'en SQL de base
- Facilité d'utilisation
- Exécuté sur le serveur donc pas d'AR entre client et serveur
- Création de fonctions et triggers même sans être root (TRUSTED mode)

Une fonction PLpgSQL peut être appelée

- Par une commande SQL ou depuis une autre fonction
  - `SELECT ajoute12();`
- Par un déclencheur (trigger)
  - De manière automatique en cas de modification de la base

SYNTAXE GÉNÉRALE

Informations nécessaires

- Nom de la fonctions
- Type et arguments en entrée et en sortie
- Des variables locales
- Le code de la fonction (entre ‘)
- Le langage utilisé pour programmer (plpgsql dans ce cours)

```
CREATE FUNCTION ma_fonction (arguments)
RETURNS typederetour AS '
[DECLARE
  declarations de variables]
BEGIN
  code de la fonction
END; '
LANGUAGE plpgsql;
```

EXEMPLES

```
CREATE FUNCTION ajouteUn (x integer)
RETURNS integer AS '
BEGIN
  return x+1;
END; ' LANGUAGE plpgsql;

SELECT ajouteUn(3);
-- 4


CREATE FUNCTION concatene (ch1 text, ch2 text)
RETURNS text as '
BEGIN
  -- pour concaténer des chaines : ||
  return ch1 || ch2;
END; ' LANGUAGE plpgsql;

SELECT concatene(nom, prenom) FROM utilisateurs;
```

EXEMPLES - HELLO WORLD

```
-- version qui retourne le résultat (fenêtre sortie de données)
CREATE FUNCTION hello_world(nom text)
RETURNS TEXT
AS '
BEGIN
  RETURN ''Bonjour '' || nom;
END; ' LANGUAGE plpgsql;

SELECT hello_world('JL');


-- version qui affiche le résultat (fenêtre messages)
CREATE FUNCTION hello_world(nom text)
RETURNS VOID
AS '
BEGIN
  RAISE NOTICE ''Bonjour %'', nom;
END; ' LANGUAGE plpgsql;

SELECT hello_world('JL');
```

EXEMPLES - FACTORIELLE

```
CREATE FUNCTION factorielle(a integer)
RETURNS INTEGER AS '
DECLARE
BEGIN
  IF a<2 THEN
    RETURN a;
  ELSE
    RETURN a*factorielle (a-1);
  END IF;
END; ' LANGUAGE plpgsql;


SELECT * FROM test WHERE id = factorielle (5);
```

CHAÎNES DE CARACTÈRES

Syntaxe pour les chaînes de caractères :

- Entourer avec des '
  - Si ' dans une chaine il faut la dupliquer pour ne pas confondre avec la fin de la chaîne
- Entourer avec des tag, en remplaçant tag par ce qu'on veut
  - On peut juste mettre $$
  - Utiliser différents tag permet d'imbriquer des chaînes simplement

```
'Attention à l''utilisation d''apostrophes'

$$Pas attention à l'utilisation d'apostrophes$$

$tag$Pas attention à l'utilisation d'apostrophes$tag$
```

CHAÎNES DE CARACTÈRES - FONCTIONS

```
-- Concaténation
SELECT 'x' || 'y';                          -- xy

-- Longueur en bits ou en caractères
SELECT char_length('tést');                 -- 4
SELECT bit_length('tést');                  -- 40 (le é prend deux octets)

-- Minuscules/majuscules
SELECT upper('Tést');                       -- TÉST
SELECT lower('Test');                       -- test

-- Recherche
SELECT position('jour' in 'Bonjour');       -- 4

-- Sous-chaîne
SELECT substring('Bonjour' from 3 for 2); -- nj

-- Remplacer une partie de la chaîne
SELECT replace('Bonjour', 'o', 'a');        -- Baujaur

-- Nettoyer une chaîne
SELECT char_length(trim('Bonjour  '));      -- 7

-- Découper une chaîne
SELECT regexp_split_to_array('abc', 'b')  -- {a,c}
```

GESTION DES PARAMÈTRES - ENTRÉE

Indiqués au moment de la déclaration de la fonction

- Nom de la variable et type
- Type uniquement
  - Avec utilisation de $i pour récupérer l'argument i (commence à 1)
  - Avec création d'alias (utilisé dans pg<8.0)

```
CREATE FUNCTION taxes(montant_ht real) RETURNS real AS $$
BEGIN return montant_ht*0.196;
END $$ LANGUAGE PLPGSQL;

CREATE FUNCTION taxes(real) RETURNS real AS $$
BEGIN return $1*0.196;
END $$ LANGUAGE PLPGSQL;

CREATE FUNCTION taxes(real) RETURNS real AS $$
DECLARE montant_total ALIAS FOR $1;
BEGIN return montant_total*0.196;
END $$ LANGUAGE PLPGSQL;
```

GESTION DES PARAMÈTRES - SORTIE

De manière classique : RETURNS type + RETURN

On peut également définir les paramètres dans la définition

- Utilisation du mot clé OUT (IN est par défaut)
- La variable vaut NULL au début => affectation dans la fonction
- Permet d'avoir plusieurs valeurs de retour simplement

```
CREATE FUNCTION tva (total_ht real)
RETURNS real AS $$
BEGIN
  RETURN total_ht * 1.196;
END; $$ LANGUAGE plpgsql;

CREATE FUNCTION tva (total_ht real, OUT total_ttc real) AS $$
BEGIN
  total_ttc := total_ht * 1.196;
END; $$ LANGUAGE plpgsql;
```

VARIABLES

Définition dans la zone DECLARE d'un bloc

Affectation

- Avec :=
- Avec SELECT INTO

```
CREATE OR REPLACE FUNCTION cherche (id_in integer)
RETURNS text AS $$
DECLARE
  res text;
BEGIN
  tax := subtotal * 0.06;

  -- met le résultat (select val ...) dans res
  SELECT INTO res val FROM test WHERE test.id=id_in;
  return res;
END; $$ LANGUAGE plpgsql;
```

DÉCLARATION DE VARIABLES

Uniquement dans la section DECLARE d'un bloc

- `name [CONSTANT] type [NOT NULL] [{ DEFAULT | := } expression];`
- CONSTANT : la variable ne peut pas être modifiée
- type : tous les types de PostgreSQL + TYPE, ROWTYPE et RECORD
- NOT NULL : tentative d'assigner NULL => erreur
- DEFAULT : valeur par défaut à l'entrée dans le bloc

```
-- création d'une constante
user_id CONSTANT integer := 10;

-- création d'un tableau de chaînes de caractères
tab text[];

-- création d'un entier valeur 32 par défaut
qtte integer DEFAULT 32;

-- création d'une variable de même type que user_id dans la table users
uid users.user_id%TYPE;
```

CHANGEMENT DE TYPE

On peut changer le type des variables :

- `CAST(val as TYPE)`

Les types doivent être compatibles

- Par exemple `CAST('test' AS INT)` produit une erreur

```
CREATE OR REPLACE FUNCTION ajoute_int(v1 int, v2 int)
RETURNS numeric(5,2) AS $$
BEGIN
    RETURN CAST (v1 + v2 AS numeric(5,2));
END; $$ LANGUAGE plpgsql;

SELECT ajoute_int(1,2);
-- 3.00

SELECT ajoute_int(CAST(1.5 AS INT), CAST(2.0 AS INT));
-- 4.00
```

IF THEN ELSE

Syntaxe générale :

- `IF … THEN … [ELSEIF … ] [ELSE …] END IF`
- ELSEIF optionnels et plusieurs possibles
- ELSE optionnel

On peut imbriquer des IF THEN ELSE END IF;

```
IF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements ]
[ ELSE
    statements ]
END IF;
```

CASE

Deux versions :

- Simple : une expression est comparée (=) aux différents cas possibles
- Equivalente à un IF THEN ELSE

```
CASE search-expression
  WHEN expression [, expression [ ... ]] THEN
    statements
  [ ELSE
    statements ]
END CASE;

CASE
  WHEN boolean-expression THEN
    statements
  [ ELSE
    statements ]
END CASE;
```

BOUCLES SIMPLES

Boucle simple :

- `LOOP instructions END LOOP ;`
- EXIT : sort de la boucle
- RETURN : sort de la fonction
- CONTINUE : revient au début de la boucle sans la terminer

```
LOOP
  ...
  IF count > 0 THEN
    EXIT;
  END IF;
END LOOP;

LOOP
  ...
  EXIT WHEN count > 0;
END LOOP;
```

BOUCLES WHILE

While

- `WHILE condition LOOP ... END LOOP;`
- Code exécuté tant que la condition est vérifiée

```
WHILE amount_owed > 0 LOOP
  ...
END LOOP;

a := 100;
WHILE (a <=200) LOOP
  a:=a+1;
END LOOP;
```

EXCEPTIONS

Gestion d'erreurs dans un bloc

- `BEGIN ...`
- `EXCEPTION WHEN condition THEN ... END`
- En cas d'exception tout le bloc est annulé

Liste des erreurs

- http://www.postgresql.org/docs/9.3/static/errcodes-appendix.html
- OTHERS pour les autres cas

```
INSERT INTO mytab(firstname, lastname) VALUES('Tom', 'Jones');
BEGIN
  UPDATE mytab SET firstname = 'Joe' WHERE lastname = 'Jones';
  y := 0 / 0;
EXCEPTION
  WHEN division_by_zero THEN RAISE NOTICE 'division par zéro';
END;
-- insert ok, update annulé
```

INFORMATIONS ET ERREURS

Indiquer un problème : RAISE

- RAISE NOTICE 'message informatif'
- RAISE EXCEPTION : lève une exception

```
CREATE OR REPLACE FUNCTION cherche (my_id int)
RETURNS text AS $$
DECLARE
  resultat text;
BEGIN
  SELECT * INTO resultat FROM test WHERE id = my_id;
  IF NOT FOUND THEN
    RAISE EXCEPTION no_data_found;
  END IF;
  return resultat;
EXCEPTION
  WHEN no_data_found THEN RAISE NOTICE 'pas trouvé %', my_id; return -1;
END;$$ LANGUAGE PLPGSQL;

-- si erreur : affiche "NOTICE:  pas trouvé" et retourne -1
-- si ok : retourne le résultat
```

BOUCLES FOR

For :

- `FOR name IN [ REVERSE ] expression .. expression [ BY expression ] LOOP ... END LOOP`
- Code exécuté pour un ensemble de valeurs

```
FOR i IN 1..10 LOOP ... END LOOP;
-- toutes les valeurs de 1 à 10 inclus

FOR i IN REVERSE 10..1 LOOP ... END LOOP;
-- toutes les valeurs de 10 à 1 inclus

FOR i IN REVERSE 10..1 BY 2 LOOP ... END LOOP;
-- toutes les valeurs de 10 à 1 par pas de 2
```

LES BLOCS

Un bloc permet de déclarer des variables et d'exécuter du code

- `[DECLARE ...] BEGIN ... END;`
- Le code principal de la fonction est donc un bloc

On peut déclarer des blocs dans des blocs

- Meilleure lisibilité du code
- Définition de variables de portée locale au sous-bloc

```
CREATE FUNCTION ma_fonction(parametres)
RETURNS type_de_retour AS '
DECLARE
  ...
BEGIN
  ...
  DECLARE
    ...
  BEGIN
    ...
  END;
  ...
END; ' LANGUAGE plpgsql;
```

LES BLOCS

```
CREATE OR REPLACE FUNCTION f() RETURNS VOID AS $$
<<a>> -- nom du bloc
DECLARE
  x integer := 4;
BEGIN
  RAISE NOTICE '% %', x, a.x;
  <<b>> -- nom du bloc
  DECLARE
    x integer := 6;
  BEGIN
    RAISE NOTICE '% % %', x, a.x, b.x;
  END;
  RAISE NOTICE '% %', x, a.x;
END; $$ LANGUAGE plpgsql;

SELECT f();
-- NOTICE:  4 4
-- NOTICE:  6 4 6
-- NOTICE:  4 4
```

CONTRAINTES DE DOMAINES

On peut vérifier une contrainte par l'appel à une fonction

- CHECK directement dans la table ou utilisation de domaine
- Fonction booléenne : retourne VRAI (si la contrainte est vérifiée) ou FAUX

```
-- vérifie qu'un nombre est multiple de 10
CREATE FUNCTION multiple10 (a integer)
RETURNS boolean AS $$
BEGIN
  RETURN a%10 = 0;
END;
$$ LANGUAGE plpgsql;

CREATE TABLE test (
  t integer primary key CHECK (multiple10(t))
);

insert into test values (5);  -- erreur
insert into test values (10); -- ok
```

MODIFICATION / SUPPRESSION

Remplacer une fonction sans changer sa définition

- `CREATE OR REPLACE …`
- Ne fonctionne pas si on change le nombre/type de attributs

Supprimer une fonction :

- `DROP FUNCTION IF EXISTS nom(arguments);`

TYPES POLYMORPHES

```
CREATE FUNCTION ajoute_int(v1 int, v2 int, v3 int)
RETURNS int AS $$
BEGIN
    RETURN v1 + v2 + v3;
END; $$ LANGUAGE plpgsql;

SELECT ajoute_int(1,2,3);       -- 6
SELECT ajoute_int(1.0,2.0,3.0); -- erreur


CREATE FUNCTION ajoute_any(v1 anyelement, v2 anyelement, v3 anyelement)
RETURNS anyelement AS $$
BEGIN
    RETURN v1 + v2 + v3;
END; $$ LANGUAGE plpgsql;

SELECT ajoute_any(1,2,3);       -- 6
SELECT ajoute_any(1.0,2.0,3.0); -- 6.0
SELECT ajoute_any(interval '1 day', interval '1 hour', interval '1 minute');
```

CURSEURS

Mickaël Coustaty

Jean-Loup Guillaume

Laboratoire Informatique Image Interaction (L3I)
Université de La Rochelle - Pôle Sciences et Technologie - Avenue Michel Crépeau - 17042 LA ROCHELLE CEDEX 1 France
Tél : +33 (0)5 46 45 82 62 – Fax : 05.46.45.82.42 – Site internet : http://l3i.univ-larochelle.fr/

TYPES COMPOSITES

Type ROWTYPE
- Contient des enregistrements de schéma défini

Type RECORD
- Comme ROWTYPE mais sans type prédéfini (cf plus loin)

Type TABLE
- Un ensemble d'enregistrements. Syntaxe particulière

```
-- déclaration de my_rows dont le schéma est celui de la table2
DECLARE
  my_rows table2%ROWTYPE;

-- type de retour de type table
CREATE OR REPLACE FUNCTION test(int)
RETURNS TABLE(id_ret int, val_ret text) AS $$
BEGIN
  RETURN QUERY SELECT * FROM test WHERE test.id<$1;
END;
$$ LANGUAGE PLPGSQL;
```

SELECT INTO

Récupérer le résultat d'une requête SQL en PLPGSQL ?

- Si on veut récupérer un seul enregistrement : SELECT INTO
  - Retourne NULL si rien à récupérer
  - Retourne le premier enregistrement si plusieurs résultats

```
DECLARE
  d varchar;
  -- d users.nom%TYPE
BEGIN
  SELECT INTO d nom FROM users WHERE id=$1;
END


DECLARE
  u users%rowtype;
BEGIN
  SELECT INTO u * FROM users WHERE id=$1;
  RAISE NOTICE '%', u.nom;
END
```

BOUCLES FOR SUR UN ENSEMBLE

Boucler sur un tableau

- `FOREACH element IN ARRAY tableau LOOP ... END LOOP`

Boucler sur un ensemble d'enregistrements

- `FOR ligne IN query LOOP ... END LOOP`

```
CREATE OR REPLACE FUNCTION display()
RETURNS VOID
AS $$
DECLARE
  x RECORD;
BEGIN
  FOR x IN SELECT * FROM users LOOP
    RAISE NOTICE '%', x.id;
  END LOOP;

END; $$ LANGUAGE plpgsql;
```

RETURN NEXT

Si le type de retour est SETOF typetable

- RETURN NEXT;
  - Ne sort pas de la fonction mais ajoute l'enregistrement au résultat
- RETURN QUERY q
  - Idem mais avec une requête
  - Peut donc ajouter plusieurs ligne d'un coup

```
CREATE OR REPLACE FUNCTION tousA()
RETURNS SETOF users AS $$
DECLARE
    r users%rowtype;
BEGIN
    FOR r IN SELECT * FROM users WHERE nom LIKE 'A%'
    LOOP
        ...
        RETURN NEXT r; -- ajoute ligne courante au résultat
    END LOOP;
    RETURN;
END $$ LANGUAGE 'plpgsql' ;

SELECT * FROM tousA();
```

CURSEURS

Récupérer le résultat d'une requête SQL en PLPGSQL ?

- Si on veut récupérer plusieurs enregistrements dans l'ordre : for
- Sinon : curseurs
- Les deux sont gérées avec des curseurs en interne

Un curseur est comme une table :

- Contient un ensemble d'enregistrements
- Peut être parcouru de ligne en ligne

```
-- version simple avec parcours dans l'ordre
FOR t IN SELECT * FROM users WHERE id < 11 LOOP
  RAISE NOTICE '%', t.nom;
END LOOP;
```

CURSEURS - OPÉRATIONS

Déclaration du curseur qui recevra les résultats de la requête SQL

- type CURSOR pour le curseur
- type RECORD pour un enregistrement

Ouverture du curseur (OPEN)

- Exécution de la requête
- Le résultat de la requête est placé dans le curseur

Parcours du curseur pour réaliser les traitements

- Parcours et récupération des lignes une à une (FETCH)
- Se déplacer sans récupérer (MOVE)

Fermeture du curseur (CLOSE)

CURSEURS - EXEMPLE

```
CREATE OR REPLACE FUNCTION affiche ()
RETURNS VOID AS $$
DECLARE
  curs CURSOR FOR select * from users;
  t RECORD ;
BEGIN
  OPEN curs;
  LOOP
    FETCH curs INTO t;
    IF NOT FOUND THEN EXIT; END IF;
    RAISE NOTICE '%', t.nom;
  END LOOP;
  CLOSE curs;
END
$$ LANGUAGE plpgsql;

SELECT affiche();
```

DECLARE

```
-- version simple
DECLARE
  curs1 CURSOR FOR SELECT * FROM users WHERE id=11;
BEGIN
  OPEN curs1

-- version avec paramètre
DECLARE
  curs2 CURSOR (key integer) FOR SELECT * FROM users WHERE id = key;
BEGIN
  OPEN curs2(11) ;

-- version générique
DECLARE
  curs3 REFCURSOR;
BEGIN
  OPEN curs3 FOR select * from users WHERE id=11;
```

FETCH / MOVE

```
LOOP
  FETCH ts INTO t;
  IF NOT FOUND THEN EXIT; END IF;
  RAISE NOTICE '%', t.val;
END LOOP;

RAISE NOTICE 'fin';

LOOP
  FETCH BACKWARD FROM ts INTO t;
  IF NOT FOUND THEN EXIT; END IF;
  RAISE NOTICE '%', t.val;
END LOOP;

MOVE RELATIVE +1 FROM ts;

FETCH ts INTO t;
RAISE NOTICE '%', t.val;
```

CURSEURS

Pour parcourir un curseur à l'envers il faut ajouter SCROLL

- `curs SCROLL CURSOR FOR ...`
- Ca peut fonctionner sans, ou pas…

Le curseur est fermé par défaut à la fin d'une transaction :

- Libération des ressources
- CLOSE est donc « inutile » en théorie

Une fonction peut retourner un curseur

CONTRAINTES DYNAMIQUES
RÈGLES ET TRIGGERS

Mickaël Coustaty

Jean-Loup Guillaume

Laboratoire Informatique Image Interaction (L3I)
Université de La Rochelle - Pôle Sciences et Technologie - Avenue Michel Crépeau - 17042 LA ROCHELLE CEDEX 1 France
Tél : +33 (0)5 46 45 82 62 – Fax : 05.46.45.82.42 – Site internet : http://l3i.univ-larochelle.fr/

LES CONTRAINTES DYNAMIQUES

Contrainte dynamique

- Liée à un événement déclencheur
  - Modification portant sur une table (INSERT, UPDATE ou DELETE)
- Exécution de code en réponse à l'événement déclencheur

2 techniques pour implémenter une contrainte dynamique

- Règle : contrôles et des traitements simples
- Trigger : contrôles et traitements plus élaborés

Règles et triggers sont liés à une table pour un événement donné

VARIABLES D'ENVIRONNEMENT

Durant le traitement d'une contrainte dynamique, on dispose :

- de(s) ANCIEN(S) enregistrement(s) : variable OLD
  - DELETE : enregistrements qui vont être supprimés
  - UPDATE : enregistrements modifiés (valeur avant modification)
  - INSERT : pas d'enregistrement avant…

- de(s) NOUVEAU(X) enregistrement(s) : variable NEW
  - INSERT : enregistrements insérés
  - UPDATE : enregistrements modifiés (valeur après modification)
  - DELETE : pas d'enregistrement après…

RÈGLES - CRÉATION

```
CREATE RULE <nomregle>
AS ON <evenement_declencheur>
TO <nomtable>
DO ALSO | INSTEAD <traitement>
```

```
CREATE OR REPLACE RULE double_insert
AS ON INSERT
TO test
WHERE id = 11
DO ALSO INSERT INTO archive VALUES (NEW.id, NEW.val);

CREATE RULE refus_insert
AS ON INSERT
TO test
WHERE id = 12
DO INSTEAD NOTHING;
```

LES DÉCLENCHEURS

Définition : un déclencheur ou trigger est

- Associé à une table et s'exécute automatiquement à chaque fois que l'événement déclencheur associé se produit
  - Soit AVANT la réalisation de l'événement déclencheur (souvent pour l'empêcher ou le modifier)
  - Soit APRES la réalisation de l'événement déclencheur (souvent pour afficher une notice ou déclencher d'autres évènements)

Ils permettent de traiter des cas plus complexes que les règles

LES DÉCLENCHEURS

Un déclencheur est constitué :

- D'une déclaration : type d'événement, table associée, etc.
- D'une fonction à exécuter si l'événement est déclenché
  - Cette fonction est le corps du déclencheur
  - Elle ne prend aucun argument et retourne un type indéterminé

Syntaxe :

```
CREATE TRIGGER nomtrigger
{ BEFORE | AFTER} { event [ OR event…]}
ON nomtable
{ FOR EACH ROW | STATEMENT}
EXECUTE PROCEDURE fonctionDeclencheur ();
```

DÉCLENCHEUR - EXEMPLE

```
-- table pour les archives
CREATE TABLE archives (
  id INTEGER,
  val CHAR(20),
  date_supression DATE);

-- fonction d'archivage
CREATE OR REPLACE FUNCTION Archivage()
RETURNS TRIGGER AS $$
BEGIN
  INSERT INTO archives VALUES (OLD.id, OLD.val, CURRENT_DATE);
  RETURN NEW;
END; $$ LANGUAGE 'plpgsql';

-- trigger exécuté avant chaque suppression ou modification de produit
CREATE TRIGGER archiveArticles
BEFORE DELETE OR UPDATE ON produits
FOR EACH ROW
  EXECUTE PROCEDURE Archivage();
```

DÉCLENCHEURS - REMARQUES

La fonction doit être définie avant le déclencheur

Ne prend aucun argument d'entrée

- Mais dispose de NEW et OLD selon l'événement

Est déclarée comme retournant un type TRIGGER

Une même fonction peut être utilisée par plusieurs déclencheurs

Si une fonction exécute des commandes SQL, alors ces commandes peuvent lancer des déclencheurs en cascade

DÉCLENCHEUR - EXEMPLE

```
CREATE TABLE test (
  id INTEGER,
  val CHAR(20)
);

CREATE OR REPLACE FUNCTION Duplication()
RETURNS TRIGGER AS $$
BEGIN
  INSERT INTO test VALUES (NEW.id, NEW.val);
  RETURN NEW;
END; $$ LANGUAGE 'plpgsql';

CREATE TRIGGER archiveArticles
BEFORE INSERT ON test
FOR EACH ROW
  EXECUTE PROCEDURE Duplication();

INSERT INTO test VALUES (1,'x');
```

MODE FOR EACH ROW / STATEMENT

```
CREATE TRIGGER nomtrigger
{ BEFORE | AFTER} { event [ OR event…]}
ON nomtable
{ FOR EACH ROW | STATEMENT}
EXECUTE PROCEDURE fonctionDeclencheur ();
```

Mode FOR EACH ROW :

- Exécuté avant ou après chaque ligne affectée par la modification

Mode STATEMENT :

- Exécuté avant ou après la requête générale de modification
- même si aucun tuple n'est concerné par l'événement déclencheur

RETOUR DES FONCTIONS

Les fonctions doivent toujours retourner un type TRIGGER

Trigger STATEMENT :

- return NULL

Trigger FOR EACH ROW / AFTER

- le type de retour n'a aucune importance, donc return NULL

Trigger FOR EACH ROW / BEFORE

- Si INSERT ou UPDATE : return NEW
- Si DELETE : return OLD
- Si return NULL, alors l'événement déclencheur n'est pas réalisé
  - Permet d'empêcher le traitement prévu

EVENT TRIGGERS LDD

Détection d'événements LDD = sur la base et pas sur une table

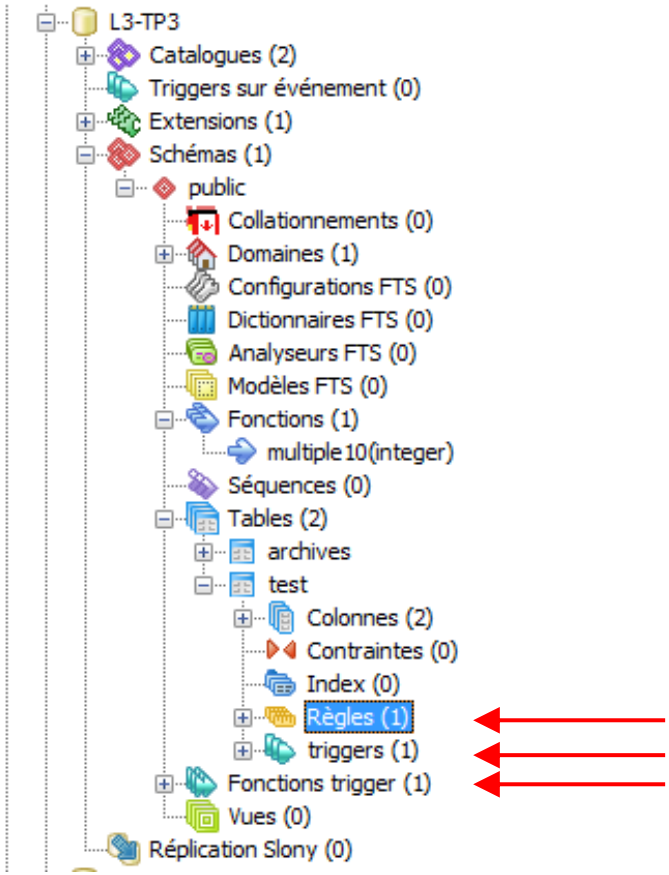
- ddl_command_start et ddl_command_end : CREATE, DROP, ALTER, etc.
- table_rewrite = ALTER TABLE
- sql_drop = toute opération DROP

```
CREATE OR REPLACE FUNCTION surveillance ()
RETURNS event_trigger AS $$
BEGIN
  RAISE NOTICE 'evenement : % %', tg_event, tg_tag;
END;
$$ LANGUAGE plpgsql;

CREATE EVENT TRIGGER surveillance
ON sql_drop
EXECUTE PROCEDURE surveillance();

DROP TABLE users;
-- NOTICE:  evenement : sql_drop DROP TABLE
```

PGADMIN


L3-TP3
Catalogues (2)
Triggers sur événement (0)
Extensions (1)
Schémas (1)
public
Collationnements (0)
Domaines (1)
Configurations FTS (0)
Dictionnaires FTS (0)
Analyseurs FTS (0)
Modèles FTS (0)
Fonctions (1)
multiple10(integer)
Séquences (0)
Tables (2)
archives
test
Colonnes (2)
Contraintes (0)
Index (0)
Règles (1)
triggers (1)
Fonctions trigger (1)
Vues (0)
Réplication Slony (0)