420-C42

Langages d'exploitation des bases de données

Partie 20

PL/pgSQL IV

Procédures, fonctions et déclencheurs

PL/pgSQL IV

Procédures et fonctions

- Il est possible de créer des procédures et des fonctions qui utilisent le langage PL/pgSQL au lieu du SQL.
- On retrouve ainsi la flexibilité des procédures et fonctions avec la puissance ajoutée du langage procédurale.
- La syntaxe correspond à un amalgame de la déclaration des procédures et fonctions jointe à la déclaration d'un bloc PLPGSQL.
- Pour les fonctions, la valeur de retour doit être spécifiée par l'instruction RETURN.

PL/pgSQL IV

Procédures

- Exemple 1 – Statistiques départementales
- Ce premier exemple crée trois procédures qui facilitent et standardisent les opérations suivantes :
  - On crée une nouvelle table permettant de faire des statistiques départementales regroupées par genre. Ces statistiques sont identifiées dans le temps et par catégorie (*mode*) de genres. Il existe ainsi 5 modes de genre : les femmes (0), les hommes (1), les genres non-binaires (2), les femmes & les hommes (3) et finalement, tous les genres (4).
  - On crée les trois procédures suivantes :
    - Création de la table (incluant un premier *snapshot*) — create_stats(mode)
    - Faire un *snapshot* pour un *mode* particulier. — snapshot_stats(mode)
    - Faire tous un *snapshot* pour tous les *modes*. — snapshot_all_stats()

PL/pgSQL IV

Procédures

```
CREATE OR REPLACE PROCEDURE create_stats(mode INT DEFAULT 4)
LANGUAGE PLPGSQL
AS $$
BEGIN
    DROP TABLE IF EXISTS stats_employe;
    CREATE TABLE stats_employe(
        id SERIAL PRIMARY KEY,
        quand TIMESTAMP NOT NULL,
        qui TEXT NOT NULL,
        nombre INT NOT NULL,
        salaire_min NUMERIC(7,2) NULL,
        salaire_moy NUMERIC(7,2) NULL,
        salaire_max NUMERIC(7,2) NULL
    );
    CALL snapshot_stats(mode); -- cette procédure sera créée par la suite
END$$;
```

PL/pgSQL IV

Procédures

```
CREATE OR REPLACE PROCEDURE snapshot_stats(mode INT DEFAULT 4)
LANGUAGE PLPGSQL
AS $$
DECLARE
	pattern TEXT := 'fhx';
BEGIN
	CASE mode				-- mode =>
		WHEN 0 THEN			-- 0 => femmes
			pattern := 'f';
		WHEN 1 THEN			-- 1 => hommes
			pattern := 'h';
		WHEN 2 THEN			-- 2 => non binaires
			pattern := 'x’;
		WHEN 3 THEN			-- 3 => femmes et hommes
			pattern := 'fh';
		WHEN 4 THEN			-- 4 => tous les genres
			pattern := 'fhx';
	END CASE;
	INSERT INTO stats_employe(quand, qui, nombre, salaire_min, salaire_moy, salaire_max)
		SELECT CURRENT_TIMESTAMP, pattern, COUNT(*), MIN(salaire), AVG(salaire), MAX(salaire)
			FROM employe
			WHERE '%' || genre || '%' LIKE pattern;
END$$;
```

PL/pgSQL IV

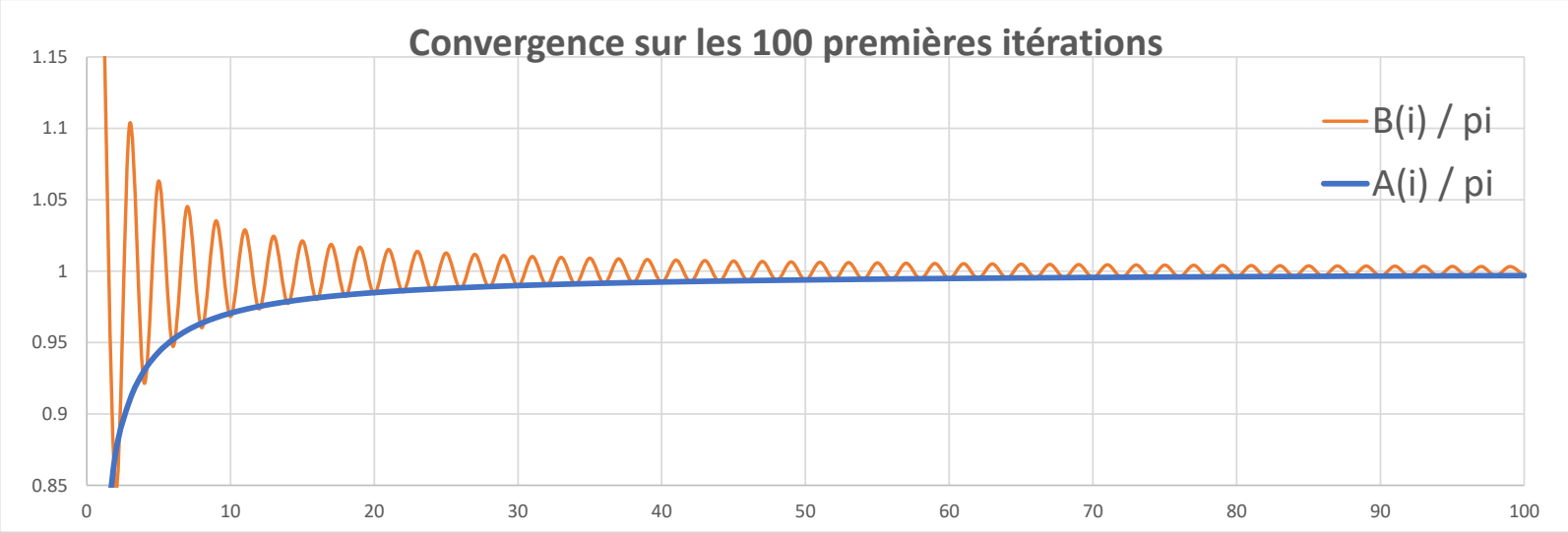
Procédures

```
CREATE OR REPLACE PROCEDURE snapshot_all_stats()
LANGUAGE PLPGSQL
AS $$
BEGIN
    FOR i IN 0..4 BY 1 LOOP
        CALL snapshot_stats(i);
    END LOOP;
END$$;

CALL create_stats();
CALL snapshot_stats(0);
CALL snapshot_all_stats();

SELECT * FROM stats_employe;
```

PL/pgSQL IV

Fonctions

- Exemple 2 - Le nombre π peut être exprimé de plusieurs façons dont ces deux séries mathématiques :
  - $A(i) = \pi = \sqrt{6 \sum_{i=1}^{\infty} \frac{1}{i^2}} \approx \sqrt{6 \left(\frac{1}{1} + \frac{1}{4} + \frac{1}{9} + \frac{1}{16} + \frac{1}{25} + \frac{1}{36} + \cdots \right)}$
  - $B(i) = \pi = 4 \sum_{i=1}^{\infty} \frac{(-1)^{i+1}}{2i-1} \approx 4 \left(\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \cdots \right)$



PL/pgSQL IV

Fonctions

- Avec PostgreSQL, comment calculer laquelle de ces deux séries converge le plus rapidement?
- Le taux de convergence μ d'une série s dont la valeur de convergence est connu $s*$ peut être approximé par :
  - $\mu \approx \frac{|s_{i+1} - s^*|}{|s_i - s^*|}$
  - $\mu_a(i) \approx \frac{|A(i+1) - \pi|}{|A(i) - \pi|}$
  - $\mu_b(i) \approx \frac{|B(i+1) - \pi|}{|B(i) - \pi|}$
- L'exemple suivant crée 4 fonctions implémentant ces séries, le taux de convergence et une fonction utilitaire effectuant la comparaison.

PL/pgSQL IV

Fonctions

```
CREATE OR REPLACE FUNCTION pi_A(max_i INT DEFAULT 1000)
    RETURNS DOUBLE PRECISION
LANGUAGE PLPGSQL
AS $$
DECLARE
    pi_value DOUBLE PRECISION := 0.0;
BEGIN
     IF max_i <= 0 THEN
        max_i := 1000;
    END IF;
    FOR i IN 1..max_i LOOP
        pi_value := pi_value + 1.0 / i^2;
    END LOOP;
    RETURN sqrt(6.0 * pi_value);
END$$;
```

PL/pgSQL IV

Fonctions

```
CREATE OR REPLACE FUNCTION pi_B(max_i INT DEFAULT 1000)
    RETURNS DOUBLE PRECISION
LANGUAGE PLPGSQL
AS $$
DECLARE
    pi_value DOUBLE PRECISION := 0.0;
BEGIN
     IF max_i <= 0 THEN
        max_i := 1000;
    END IF;
    FOR i IN 1..max_i LOOP
        pi_value := pi_value + (-1.0)^(i + 1) / (2.0 * i - 1.0);
    END LOOP;
    RETURN 4.0 * pi_value;
END$$;
```

PL/pgSQL IV

Fonctions

```
CREATE OR REPLACE FUNCTION taux_convergence(
        s_i DOUBLE PRECISION,
        s_ip1 DOUBLE PRECISION,
        s_star DOUBLE PRECISION)
    RETURNS DOUBLE PRECISION
LANGUAGE PLPGSQL
AS $$
BEGIN
    RETURN abs(s_ip1 - s_star) / abs(s_i - s_star);
END$$;
```

PL/pgSQL IV

Fonctions

```
DROP FUNCTION IF EXISTS compare_taux;
CREATE OR REPLACE FUNCTION compare_taux(
        depth INT DEFAULT 100,
        format TEXT DEFAULT '0.999999990')
    RETURNS TABLE (i INT, pi_reel TEXT, pi_a TEXT, pi_b TEXT, taux_conv_a TEXT, taux_conv_b TEXT)
LANGUAGE SQL
AS $$
    SELECT     depth AS i,
        to_char(pi(), format) AS "Pi réel",
        to_char(pi_A(depth), format) AS "Pi série A",
        to_char(pi_B(depth), format) AS "Pi série B",
        to_char(taux_convergence(pi_A(depth + 1), pi_A(depth), pi()), format) AS "Taux conv. A",
        to_char(taux_convergence(pi_B(depth + 1), pi_B(depth), pi()), format) AS "Taux conv. B";
$$;
SELECT compare_taux(100);   -- voir note 1
SELECT * FROM compare_taux(100);   -- voir note 2
```

- **Attention, cette fonction retourne une table!** Ainsi, la requête **1** retourne la table et non pas une requête sur la table. Ainsi, le résultat correspond à une seule colonne contenant une liste de valeurs pour chaque ligne. La requête **2** retourne toutes les colonnes tel qu'attendu.

PL/pgSQL IV

Fonctions

- Exemple 3
- On désire produire une fonction qui fait la transformation d'un nombre entier décimale en une représentation texte de n'importe quelle base entre 2 et 36.
- On sait que pour une base donnée, il est nécessaire d'avoir un symbole représentant chacune des valeurs possibles. On choisit ici d'utiliser les chiffres de 0 à 9 et ensuite les lettres de A à Z. Ainsi, ayant 36 symboles disponibles, il est possible de représenter des nombres jusqu'en base 36. Par exemple :
  - base 2 : 0 – 1
  - base 7 : 0 à 6
  - base 21 : 0 à 9 + A à K

PL/pgSQL IV

Fonctions

- On crée une seule fonction réalisant cette tâche.
- Il aurait été probablement avantageux de découper la fonction en deux parties (logique et représentation). Toutefois, dans ce cas-ci, une seule fonction est justifiable.
- À titre pédagogique, la fonction est récursive. Une version itérative serait toutefois favorable.
- La fonction possède un 3^{e} paramètre permettant d'ajouter un préfixe représentant la base utilisée. Les préfixes standards sont utilisés pour les bases 2 (0b…), 8 (0…), 10(*rien*) et 16 (0x…). Toutefois, pour tous les autres cas, on ajoute le préfixe suivant : b*x*_ où x représente la base.

PL/pgSQL IV

Fonctions

```
CREATE OR REPLACE FUNCTION dec_to_base(
        number BIGINT,
        base INT DEFAULT 2,
        include_base_info BOOLEAN DEFAULT TRUE)
    RETURNS TEXT
LANGUAGE PLPGSQL
AS $$
DECLARE
    digit_value INT;
    digit_char CHAR;
BEGIN
    IF base NOT BETWEEN 2 AND 36 THEN
        RAISE EXCEPTION 'La base doit être dans l''interval [2, 36]';
    END IF;
    -- la suite sur l'autre page
```

PL/pgSQL IV

Fonctions

```
    IF number = 0 THEN
        IF include_base_info THEN
            CASE base
                WHEN 2 THEN RETURN '0b';
                WHEN 8 THEN RETURN '0';
                WHEN 10 THEN RETURN '';
                WHEN 16 THEN RETURN '0x';
                ELSE RETURN 'b' || TO_CHAR(base, 'FM9') || '_';
            END CASE;
        ELSE
            RETURN '';
        END IF;
    ELSE
        digit_value := number % base;
        IF digit_value < 10 THEN
            digit_char := CHR(digit_value + ASCII('0'));
        ELSE
            digit_char := CHR(digit_value - 10 + ASCII('A'));
        END IF;
        RETURN dec_to_base(number / base, base, include_base_info) || digit_char;
    END IF;
END$$;
```

PL/pgSQL IV

Fonctions

- Plusieurs exercices intéressants peuvent être envisagés avec cet exemple :
  - réaliser cette fonction en version itérative
  - réaliser la fonction inverse base_to_dec, convertir un nombre de n'importe quelle base (2 à 36) vers la base 10 (version récursive et itérative)
  - réaliser la fonction base_to_base, convertir un nombre de n'importe quelle base vers n'importe quelle autre base (version récursive et itérative)
  - convertir une chaîne de caractères en une série de nombres en base 10 (la chaîne de caractères étant considérée un nombre en base 36):
    - il faut retirer les accents
    - tous les caractères de ponctuation sont considérés comme des séparateurs de mot
- HexSpeak : que valent?
  - SELECT dec_to_base(195948557, 16, FALSE);
  - SELECT dec_to_base(3405691582, 16, FALSE);
  - SELECT dec_to_base(3735929054, 16, FALSE);

PL/pgSQL IV

Déclencheurs

- Les déclencheurs sont des outils puissants qui permettent l'automatisation d'appels de procédures dans divers contextes liés au DML.
- Par exemple, un déclencheur pourra être utiliser pour automatiser ces types d'actions lors des opérations en cours :
  - arrêter une insertion ou une mise à jour invalide selon des conditions plus complexes
  - calculer automatiquement certains champs
  - faire des insertions, des mises à jour ou des suppressions automatiques dans d'autres tables
- Les déclencheurs sont si puissants qu'il est possible de garantir une intégrité totale de la BD en tout temps seulement du côté serveur.
- Il faut comprendre que même si les 6 contraintes fondamentales sont puissantes, elles ne permettent pas de valider tous les cas. Principalement lorsqu'une donnée est liée indirectement aux données d'une autre table.

PL/pgSQL IV

Déclencheurs

- Avantages :
  - Automatisation :
    - validations plus avancées
      valider des conditions dans d'autres tables, …
    - génération de données en temps opportun
      générer une valeur selon un patron donné, …
    - opérations spécifiques
      mettre à jour une table d'historique, …
  - Augmente l'intégrité des données.
  - Facilite les opérations liées à la base de données.
- Désavantages :
  - plus complexes à programmer et à maintenir
  - ce sont des objets qui nécessitent des ressources significatives, un usage abusif entraîne une réduction de performance de la base de données.

PL/pgSQL IV

Déclencheurs

- Avec PostgreSQL, un déclencheur est déclaré en 2 parties :
  - création de la fonction opérationnelle (la tâche à faire lors de l'évènement)
  - création du *trigger* lui-même en considérant ses conditions et la fonction à lier
- Il existe plusieurs types d'évènement DML pouvant générer un déclenchement. Voici les évènements de base :
  - BEFORE [INSERT | UPDATE | DELETE]
    le déclenchement a lieu avant l'opération
  - AFTER [INSERT | UPDATE | DELETE]
    le déclenchement a lieu après l'opération

PL/pgSQL IV

Déclencheurs

- Le déclencheur doit être lié à une fonction spéciale de type « *trigger function* » :
  - qui ne prend aucun argument en entrée
  - qui retourne un objet TRIGGER.
- Les « *trigger function* » possède des variables spéciales permettant de manipuler des informations contextuelles à l'évènement courant. On présente ici l'essentiel :
  - NEW : de type RECORD — INSERT – UPDATE
    donne accès à la nouvelle valeur (exemple : NEW.id, NEW.name, ...)
  - OLD : de type RECORD — UPDATE – DELETE
    donne accès à l'ancienne valeur (exemple : OLD.id, OLD.name, ...)
  - TG_OP : de type TEXT — INSERT – UPDATE – DELETE
    chaîne de caractères correspondant à l'opération ('INSERT', 'UPDATE', 'DELETE')
  - TG_TABLE_NAME : de type TEXT — INSERT – UPDATE – DELETE
    chaîne de caractères représentant la table concernée

PL/pgSQL IV

Déclencheurs

- La « *trigger function* » doit retourner
  - un RECORD correspondant à la table associée
    l'évènement déclencheur continue avec la valeur retournée pour la ligne en cours
  - NULL
    l'évènement déclencheur est annulé pour la ligne en cours
  - il est toujours possible de lever une exception pour arrêter toute l'opération en cours et retourner un message en conséquence
- Quelques situations typiques :
  - pour INSERT et UPDATE, si tout va bien, on retourne NEW
  - pour INSERT, si on veut modifier une valeur spécifiquement, on la modifie par NEW.value := valeur;, et on retourne NEW
  - pour DELETE, on retourne NULL pour annuler la suppression ou OLD pour poursuivre la suppression (le contenu du RECORD de retour n'a aucune importance en soit mais doit être retourné pour autoriser la poursuite de la suppression en cours)
  - on lève une exception si on détecte une logique violant certaines règles d'intégrité

PL/pgSQL IV

Déclencheurs

- Le DDL simplifié associé au déclencheur est :

CREATE TRIGGER *nom_trigger*
{ BEFORE | AFTER } { INSERT | UPDATE | DELETE } [OR …]
ON *nom_table*
FOR EACH ROW
EXECUTE PROCEDURE *nom_procedure*();
DROP TRIGGER [IF EXISTS] *nom_trigger* ON *nom_table*;

- Les diapositives suivantes présentent 3 exemples :
  - Exemple 1 – Mettre à jour une table d'historique automatiquement
  - Exemple 2 – Générer un numéro de série automatiquement
  - Exemple 3 – Gestion d'une contrainte d'intégrité

PL/pgSQL IV

Déclencheurs – Exemple 1

- Exemple 1 – Mettre à jour une table d'historique automatiquement
- Prenons l'exemple où on désire connaître l'historique des superviseurs de département. Sachant que les départements auront différents superviseurs au fil du temps, on pourrait réaliser cette stratégie :
  - créer une table d'historique pour les superviseurs de département
  - on insère une entrée dans la table d'historique pour chacun de ces évènements :
    - à chaque INSERT dans la table département
    - à chaque UPDATE dans la table département (si le superviseur change)
    - à chaque DELETE dans la table département
- On nomme ce genre de table : table d'audit

PL/pgSQL IV

Déclencheurs – Exemple 1

- Création de la table

```
CREATE TABLE histo_departement(
    id              SERIAL      PRIMARY KEY,
    type            TEXT        NOT NULL,
    event           DATE        NOT NULL,
    department      INT         NOT NULL REFERENCES departement(id),
    superviseur     INT         NOT NULL REFERENCES employe(nas)
);
-- attention, le fait d'ajouter des contraintes de clés étrangères dans
-- cette table fait en sorte qu'ils sera impossible d'effacer un
-- département!
```

PL/pgSQL IV

Déclencheurs – Exemple 1

- Création de la fonction activée par le déclencheur

```
CREATE OR REPLACE FUNCTION histo_dep_update() RETURNS TRIGGER
LANGUAGE PLPGSQL AS $$
DECLARE
    do_hist_insert BOOLEAN;
BEGIN
    CASE TG_OP -- TG_OP => l'operation du trigger
        WHEN 'INSERT', 'DELETE' THEN
            do_hist_insert := TRUE;
        WHEN 'UPDATE' THEN
            do_hist_insert := (NEW.superviseur <> OLD.superviseur);
    END CASE;
    IF do_hist_insert THEN
        INSERT INTO histo_department
            VALUES (DEFAULT, TG_OP, CURRENT_TIMESTAMP, NEW.id, NEW.superviseur);
    END IF;
    RETURN NEW;
END$$;
```

PL/pgSQL IV

Déclencheurs – Exemple 1

- Création du déclencheur

```
CREATE TRIGGER histo_dep_trig
    AFTER INSERT OR UPDATE ON departement
    FOR EACH ROW
    EXECUTE PROCEDURE histo_dep_update();

INSERT INTO departement VALUES(DEFAULT, 'Lettres', 'Laval', 222);
INSERT INTO departement VALUES(DEFAULT, 'Arts', 'Laval', 123);
-- l'instruction DELETE est impossible à cause des clés étrangères
UPDATE departement SET superviseur = 123 WHERE nom = 'Arts';
UPDATE departement SET superviseur = 222 WHERE nom = 'Arts';
```

PL/pgSQL IV

Déclencheurs – Exemple 2

- Exemple 2 – Générer un numéro de série automatiquement
- Supposons une table de produits dont les numéros de série doivent être généré et unique pour chaque nouveau produit.
- Le patron est le suivant : DX-%%-$$$$-######
  - %% : les deux premières lettres du modèle en majuscule
  - $$$$: année et mois de production avec deux chiffres chacun
  - ###### : un numéro séquentiel unique depuis le début de la production convertit en base 32 (voir l'exemple précédent) – le no débute à 1000.
- La stratégie est la suivante, à chaque insertion d'un nouveau produit, on détermine son numéro de série automatiquement à l'aide d'un déclencheur. De plus, une séquence sera utilisée pour le no. séq.

PL/pgSQL IV

Déclencheurs – Exemple 2

- Création des tables et de la séquence

```
CREATE TABLE modele (
    id          SERIAL          PRIMARY KEY,
    nom         VARCHAR(64)     NOT NULL CHECK(length(nom) >= 2)
);

CREATE TABLE produit(
    id              SERIAL      PRIMARY KEY,
    modele          INT         NOT NULL REFERENCES modele(id),
    no_serie        CHAR(17)    NOT NULL UNIQUE
);

CREATE SEQUENCE modele_no_seq
    START WITH 1000
    INCREMENT BY 1;
```

PL/pgSQL IV

Déclencheurs – Exemple 2

- Création de la fonction liée au déclencheur

```
CREATE OR REPLACE FUNCTION genere_no_serie() RETURNS TRIGGER
LANGUAGE PLPGSQL AS $$
BEGIN
    SELECT 'DX-' ||
                (SELECT SUBSTRING(UPPER(nom), 1, 2)
                        FROM modele WHERE id = NEW.modele) || '-' ||
                TO_CHAR(CURRENT_DATE, 'YYDD') || '-' ||
                (SELECT LPAD(dec_to_base(
                                        nextval('modele_no_seq'),
                                        32, FALSE),
                                    6, '0'))
        INTO NEW.no_serie; -- on change le no. de série ici!!!
    RETURN NEW;
END$$;
```

PL/pgSQL IV

Déclencheurs – Exemple 2

- Création du déclencheur

```
CREATE TRIGGER genere_no_serie_trig
    BEFORE INSERT ON produit
    FOR EACH ROW
    EXECUTE PROCEDURE genere_no_serie();

INSERT INTO modele VALUES
    (DEFAULT, 'Oblivion'), (DEFAULT, 'Dystopia'), (DEFAULT, 'Absolute’);

DO LANGUAGE PLPGSQL $$
BEGIN
    FOR i IN 1..10000 LOOP
        INSERT INTO produit(modele)
            VALUES ((SELECT id FROM modele WHERE nom = 'Dystopia’));
    END LOOP;
END$$;
```

PL/pgSQL IV

Déclencheurs – Exemple 3

- Exemple 3 – Gestion d'une contrainte d'intégrité
- Pour ce dernier exemple, on désire valider les contraintes suivantes :
  - un employé ne peut avoir lui-même comme superviseur
  - un employé ne peut être le superviseur de son superviseur
- Une simple contrainte CHECK peut répondre à la 1re validation alors qu'il est impossible de le faire pour la 2e.
- Il suffit d'ajouter un déclencheur effectuant cette validation lors d'un INSERT ou d'un UPDATE dans la table employe.

PL/pgSQL IV

Déclencheurs – Exemple 3

- Création de la fonction liée au déclencheur

```
CREATE OR REPLACE FUNCTION valide_supsup() RETURNS TRIGGER
LANGUAGE PLPGSQL AS $$
DECLARE
    nouveau_sup employe.superviseur%TYPE;
BEGIN
    SELECT emp.superviseur INTO nouveau_sup
        FROM employe AS emp
        WHERE nas = NEW.superviseur;
    IF NEW.superviseur = NEW.nas THEN
        RAISE EXCEPTION 'L''employé ne peut être son propre superviseur! NAS employe : %', NEW.nas;
    END IF;
    IF nouveau_sup = NEW.nas THEN
        RAISE EXCEPTION
            'L''employé ne peut être le superviseur de son superviseur! NAS employe : %', NEW.nas;
    END IF;
    RETURN NEW;
END$$;
```

PL/pgSQL IV

Déclencheurs – Exemple 3

- Création de la fonction liée au déclencheur

```
CREATE TRIGGER valide_supsup_trig
    BEFORE INSERT OR UPDATE ON employe
    FOR EACH ROW
    EXECUTE PROCEDURE valide_supsup();
```