Bases de données

Triggers

Nadime Francis

Université Gustave Eiffel
LIGM - 4B130 Copernic
nadime.francis@univ-eiffel.fr

ON DELETE : une forme implicite de trigger

```
CREATE TABLE stocke(
  idmag int REFERENCES magasin(idmag) ON DELETE CASCADE,
  idpro int REFERENCES produit(idpro) ON DELETE CASCADE,
  prixUnit numeric(5,2),
  quantite int,
  PRIMARY KEY(idmag,idpro)
);
```

Supprimer un produit ou un magasin entraine la suppression de la ligne correspondante dans stocke.

ON DELETE : une forme implicite de trigger

```
CREATE TABLE stocke(
  idmag int REFERENCES magasin(idmag) ON DELETE CASCADE,
  idpro int REFERENCES produit(idpro) ON DELETE CASCADE,
  prixUnit numeric(5,2),
  quantite int,
  PRIMARY KEY(idmag,idpro)
);
```

Supprimer un produit ou un magasin entraine la suppression de la ligne correspondante dans stocke.

```
CREATE TABLE facture(
  idfac int primary key,
  datefac date,
  numcli int REFERENCES client(numcli) ON DELETE SET NULL,
  idmag int REFERENCES magasin(idmag) ON DELETE CASCADE
);
```

Lorsqu'un client est supprimé, il n'est plus renseigné sur les factures qu'il avait contractées.

Triggers (déclencheurs)

- Trigger
  - Opération déclenchée automatiquement lors de certains événements

- Structure ECA (Événement – Condition – Action)
  - Événement : une mise-à-jour de la base de données (insertion, suppression, modification) qui déclenche le trigger
  - Condition, devant être vérifiée pour exécuter le trigger
  - Action : procédure exécutée sur la base lorsque le trigger est declenché et la condition satisfaite

Cas d'utilisation

- Opérations de maintenance
  - Génération de logs
  - Synchronisation de données répliquées
- Mise-à-jour automatiques
  - Complétion de tuples avant insertion
  - Traduction de mise-à-jour dans des vues
- Définition de contraintes complexes
  - Annulation de modifications invalides
  - Signalement d'erreurs, exceptions, avertissements

Un premier trigger

```
CREATE TRIGGER suiviPrix
AFTER UPDATE ON stocke
FOR EACH ROW
WHEN (OLD.prixUnit != NEW.prixUnit)
EXECUTE PROCEDURE logprix();
```

Un premier trigger

```
CREATE TRIGGER suiviPrix
AFTER UPDATE ON stocke
FOR EACH ROW
WHEN (OLD.prixUnit != NEW.prixUnit)
EXECUTE PROCEDURE logprix();
```

et la fonction logprix() :

```
CREATE OR REPLACE FUNCTION logprix()
RETURNS TRIGGER AS
$$
  BEGIN
  INSERT INTO historiquePrix(dateH,idmag,idpro,ancienPrix,nouveauPrix)
  VALUES (now(), OLD.idmag, OLD.idpro, OLD.prixUnit, NEW.prixUnit);
  RETURN NEW;
  END;
$$
LANGUAGE plpgsql;
```

Un peu de recul...

1. Définition de triggers
   - Concepts communs à de nombreux RDBMS
   - Quelques différences subsistent :
     - Row vs statement
     - Insertion dans des vues
     - Modification du schéma
   - Référence pour ce cours : PostgresQL

2. Programmation de procédures
   - Nombreux choix de langages : C, Python, Perl...
   - Ceci n'est pas un cours de programmation.
     Vous savez tous déjà programmer !
   - Nous allons voir (un tout petit peu de) PL/pgSQL

Définition de triggers

Syntaxe simplifiée

```
CREATE TRIGGER nomTrig
{ BEFORE | AFTER | INSTEAD OF } événement [ OR ... ]
ON nomTable
FOR EACH { ROW | STATEMENT }
WHEN (condition)
EXECUTE PROCEDURE nomFunc(arguments);
```

où événement est de l'un des types suivants :

```
INSERT | UPDATE [ OF nomCol ] | DELETE | TRUNCATE
```

- Structure ECA :
  - événement est l'événement déclenchant
  - condition spécifie la condition du trigger
  - nomFunc est l'action déclenchée par le trigger
- Le trigger nomTrig est attaché à la table nomTable
  (sous Postgres, mais pas sous Oracle...)
- **DROP TRIGGER** nomTrig **ON** nomTable; pour supprimer le trigger

Syntaxe simplifiée

```
CREATE TRIGGER nomTrig
{ BEFORE | AFTER | INSTEAD OF } événement [ OR ... ]
ON nomTable
FOR EACH { ROW | STATEMENT }
WHEN (condition)
EXECUTE PROCEDURE nomFunc(arguments);
```

où événement est de l'un des types suivants :

```
INSERT | UPDATE [ OF nomCol ] | DELETE | TRUNCATE
```

Remarques :

- **BEFORE** / **AFTER** : spécifie si l'action est exécutée avant ou après l'événement déclenchant
- **INSTEAD OF** : uniquement utilisé pour les mises-à-jour et insertions dans les vues
- Pas de sous-requêtes dans la clause **WHEN** !

Types de triggers : statement

Triggers **FOR EACH STATEMENT**

- Déclenché une seule fois par événement
  Même si aucune ou plusieurs lignes sont affectées
- Utile pour maintenance ne dépendant pas des données affectées
  Écriture d'un log, mise-à-jour d'une vue matérialisée...
- La valeur de retour de l'action n'est pas utilisée
  Si l'action lève une exception, tout l'événement est annulé

Ex :

```
CREATE TRIGGER majMoyenne
AFTER INSERT OR DELETE OR UPDATE OF prixUnit
ON stocke
FOR EACH STATEMENT
EXECUTE PROCEDURE refreshViewMoy();
-- Rafraichit la vue prixMoy quand un produit est ajouté ou supprimé d'un magasin
-- ou qu'un prix est modifié
```

♣ Types de triggers : statement

(Sous Postgres 10) Accès aux lignes affectées

```
CREATE TRIGGER majNbVente
AFTER INSERT ON contient
REFERENCING NEW TABLE AS nouvellesVentes
FOR EACH STATEMENT
EXECUTE PROCEDURE ajoutVentes();
-- Mise à jour du nombre de produits vendus après chaque nouvelle vente
```

- Uniquement à partir de Postgres 10
- Tables temporaires **OLD TABLE** et **NEW TABLE**
  - Contiennent les lignes affectées, avant et après modification
  - Consultées comme des tables normales dans la fonction déclenchée
- Plus performant que **FOR EACH ROW** si le trigger concerne l'ensemble des lignes modifiées (agrégats, décompte, etc.)

Types de triggers : row

Triggers **FOR EACH ROW**

- Déclenché pour chaque ligne insérée, modifiée ou supprimée
- **OLD** et **NEW** font référence à la ligne avant et après modification
  Définis dans la condition et dans l'action du trigger
- Pour un trigger **BEFORE**, la valeur de retour de l'action :
  - Remplace la ligne affectée (cas **INSERT** ou **UPDATE**)
  - Annule la modification si l'action renvoie **NULL**
- Pour un trigger **AFTER** la valeur de retour est ignorée
- Si l'action lève une exception, l'événement au complet est annulé

Ex :

```
CREATE TRIGGER suiviPrix
BEFORE UPDATE ON stocke
FOR EACH ROW
WHEN (OLD.prixUnit != NEW.prixUnit)
EXECUTE PROCEDURE logprix();
-- suivi du prix des produits, déclenché à chaque fois qu'un produit change de prix
```

Intermède : rappel du cours de L2 sur les vues

Vues

Une vue est une relation virtuelle définie par une requête

- Les vues ne stockent pas de données
  Elles sont recalculées à partir de leur définition à chaque utilisation
- Les requêtes **SELECT** sur les vues se comportent comme sur les tables
  On ne peut en général pas faire de **INSERT** / **DELETE** / **UPDATE**

Objectifs :

- Adapter le schéma relationnel pour une application spécifique
- Servir de raccourci pour une requête complexe
- Restreindre l'accès à certaines données pour certains utilisateurs

Création et suppression de vues

- Création de la vue et choix de ses attributs
  Si non spécifiés, les attributs choisis sont ceux de la requête

```
CREATE VIEW nomVue (attr1, ..., attrn) AS (requête);
```

- Suppression de la vue

```
DROP VIEW nomVue;
```

Ex :

```
CREATE VIEW nouveauL3 AS
  (
    SELECT numEtud, nom
    FROM etudiant NATURAL JOIN examen NATURAL JOIN cours
    WHERE note >= 10 AND numLic = 2
    GROUP BY numEtud, nom
    HAVING sum(ects) >= 60
  );
-- les étudiants de L2 qui passent en L3
```

```
SELECT * FROM nouveauL3 NATURAL JOIN examen NATURAL JOIN cours
WHERE intitule = 'Base de données';
-- Les futurs L3 qui ont suivi le cours de base de données
```

Utilisation des vues

- Pour garantir l'indépendance logique
  - Isolation logique des programmes par rapport aux données
  - Les programmes accèdent seulement aux données transmises au travers de vues
  - Si le schéma de la base est modifié, l'admistrateur ajustera la définition des vues, le changement est invisible de l'extérieur

- Pour restreindre l'accès aux données
  - Les utilisateurs peuvent consulter uniquement les données transmises par certaines vues choisies par l'adminsitrateur
  - Les données en dehors des vues sont protégées

Vues matérialisées

- Vue matérialisée : vue dont le contenu est stocké à la création
- Permet de mémoriser le résultat de requêtes coûteuses

- Création de la vue et choix de ses attributs

```
CREATE MATERIALIZED VIEW nomVue (attr1, ..., attrn) AS (requête);
```

- Mise-à-jour de la vue (en cas de modification des tables d'origine)

```
REFRESH MATERIALIZED VIEW nomVue;
```

- Suppression de la vue

```
DROP MATERIALIZED VIEW nomVue;
```

Insertion et mise-à-jour dans une vue ?

```
CREATE VIEW nouveauL3 AS
 (
   SELECT numEtud, nom
   FROM etudiant NATURAL JOIN examen NATURAL JOIN cours
   WHERE note >= 10 AND numLic = 2
   GROUP BY numEtud, nom
   HAVING sum(ects) >= 60
 );
```

```
INSERT INTO nouveauL3 values (12549, 'Delacour');
```

Insertion et mise-à-jour dans une vue ?

```
CREATE VIEW nouveauL3 AS
 (
   SELECT numEtud, nom
   FROM etudiant NATURAL JOIN examen NATURAL JOIN cours
   WHERE note >= 10 AND numLic = 2
   GROUP BY numEtud, nom
   HAVING sum(ects) >= 60
 );
```

```
INSERT INTO nouveauL3 values (12549, 'Delacour');
```

```
ERROR: cannot insert into view "nouveauL3"
DETAIL: Views containing GROUP BY are not automatically updatable.
```

Insertion et mise-à-jour dans une vue ?

```
CREATE VIEW nouveauL3 AS
 (
   SELECT numEtud, nom
   FROM etudiant NATURAL JOIN examen NATURAL JOIN cours
   WHERE note >= 10 AND numLic = 2
   GROUP BY numEtud, nom
   HAVING sum(ects) >= 60
 );
```

```
INSERT INTO nouveauL3 values (12549, 'Delacour');
```

```
ERROR: cannot insert into view "nouveauL3"
DETAIL: Views containing GROUP BY are not automatically updatable.
```

(et aussi : pas de jointures, sous-requêtes, opérations ensemblistes, distinct, ...)

Insertion et mise-à-jour dans une vue ?

```
CREATE VIEW nouveauL3 AS
 (
   SELECT numEtud, nom
   FROM etudiant NATURAL JOIN examen NATURAL JOIN cours
   WHERE note >= 10 AND numLic = 2
   GROUP BY numEtud, nom
   HAVING sum(ects) >= 60
 );
```

```
INSERT INTO nouveauL3 values (12549, 'Delacour');
```

```
ERROR: cannot insert into view "nouveauL3"
DETAIL: Views containing GROUP BY are not automatically updatable.
```

(et aussi : pas de jointures, sous-requêtes, opérations ensemblistes, distinct, ...)

Le système ne peut pas deviner si vous voulez :

Insertion et mise-à-jour dans une vue ?

```
CREATE VIEW nouveauL3 AS
 (
   SELECT numEtud, nom
   FROM etudiant NATURAL JOIN examen NATURAL JOIN cours
   WHERE note >= 10 AND numLic = 2
   GROUP BY numEtud, nom
   HAVING sum(ects) >= 60
 );
```

```
INSERT INTO nouveauL3 values (12549, 'Delacour');
```

```
ERROR: cannot insert into view "nouveauL3"
DETAIL: Views containing GROUP BY are not automatically updatable.
```

(et aussi : pas de jointures, sous-requêtes, opérations ensemblistes, distinct, ...)

Le système ne peut pas deviner si vous voulez :

- Augmenter les notes de l'étudiant jusqu'à ce qu'il valide 60 ECTS ?

Insertion et mise-à-jour dans une vue ?

```
CREATE VIEW nouveauL3 AS
 (
   SELECT numEtud, nom
   FROM etudiant NATURAL JOIN examen NATURAL JOIN cours
   WHERE note >= 10 AND numLic = 2
   GROUP BY numEtud, nom
   HAVING sum(ects) >= 60
 );
```

```
INSERT INTO nouveauL3 values (12549, 'Delacour');
```

```
ERROR: cannot insert into view "nouveauL3"
DETAIL: Views containing GROUP BY are not automatically updatable.
```

(et aussi : pas de jointures, sous-requêtes, opérations ensemblistes, distinct, ...)

Le système ne peut pas deviner si vous voulez :

- Augmenter les notes de l'étudiant jusqu'à ce qu'il valide 60 ECTS ?
- Lui faire valider une matière factice octroyant suffisament d'ECTS ?

Insertion et mise-à-jour dans une vue ?

```
CREATE VIEW nouveauL3 AS
 (
   SELECT numEtud, nom
   FROM etudiant NATURAL JOIN examen NATURAL JOIN cours
   WHERE note >= 10 AND numLic = 2
   GROUP BY numEtud, nom
   HAVING sum(ects) >= 60
 );
```

```
INSERT INTO nouveauL3 values (12549, 'Delacour');
```

```
ERROR: cannot insert into view "nouveauL3"
DETAIL: Views containing GROUP BY are not automatically updatable.
```

(et aussi : pas de jointures, sous-requêtes, opérations ensemblistes, distinct, ...)

Le système ne peut pas deviner si vous voulez :

- Augmenter les notes de l'étudiant jusqu'à ce qu'il valide 60 ECTS ?
- Lui faire valider une matière factice octroyant suffisament d'ECTS ?
- Changer les quantités d'ECTS des matières qu'il valide déjà ?

Insertion et mise-à-jour dans une vue ?

```
CREATE VIEW nouveauL3 AS
 (
   SELECT numEtud, nom
   FROM etudiant NATURAL JOIN examen NATURAL JOIN cours
   WHERE note >= 10 AND numLic = 2
   GROUP BY numEtud, nom
   HAVING sum(ects) >= 60
 );
```

```
INSERT INTO nouveauL3 values (12549, 'Delacour');
```

```
ERROR: cannot insert into view "nouveauL3"
DETAIL: Views containing GROUP BY are not automatically updatable.
```

(et aussi : pas de jointures, sous-requêtes, opérations ensemblistes, distinct, ...)

Le système ne peut pas deviner si vous voulez :

- Augmenter les notes de l'étudiant jusqu'à ce qu'il valide 60 ECTS ?
- Lui faire valider une matière factice octroyant suffisament d'ECTS ?
- Changer les quantités d'ECTS des matières qu'il valide déjà ?

... d'autres idées ?

Triggers et vues

Les triggers de type **INSTEAD OF** (événement) permettent de spécifier le comportement des insertions, suppressions et mises-à-jour dans les vues.

```
CREATE TRIGGER insereL3
INSTEAD OF INSERT ON nouveauL3
FOR EACH ROW
EXECUTE PROCEDURE pointsJury();
```

Triggers et vues

Les triggers de type **INSTEAD OF** (événement) permettent de spécifier le comportement des insertions, suppressions et mises-à-jour dans les vues.

```
CREATE TRIGGER insereL3
INSTEAD OF INSERT ON nouveauL3
FOR EACH ROW
EXECUTE PROCEDURE pointsJury();
```

Remarques :

- On doit spécifier comment traiter chaque ligne :
  - Le trigger est forcément de type **FOR EACH ROW**
  - La clause **WHEN** n'est pas autorisée

Triggers et vues

Les triggers de type **INSTEAD OF** (événement) permettent de spécifier le comportement des insertions, suppressions et mises-à-jour dans les vues.

```
CREATE TRIGGER insereL3
INSTEAD OF INSERT ON nouveauL3
FOR EACH ROW
EXECUTE PROCEDURE pointsJury();
```

Remarques :

- On doit spécifier comment traiter chaque ligne :
  - Le trigger est forcément de type **FOR EACH ROW**
  - La clause **WHEN** n'est pas autorisée
- La valeur de retour de la fonction est utilisée comme signalement au système (pour **RETURNING**) et aux autres triggers :
  - **NEW** : l'insertion ou la mise à jour ont été effectuées
  - **OLD** : la suppression a été effectuée
  - **NULL** : la modification demandée a été annulée
  - Un autre enregistrement : valeurs de la ligne qui a été affectée

La fonction pointsJury

```
CREATE FUNCTION pointsJury() RETURNS TRIGGER AS
$$
  DECLARE etu etudiant%ROWTYPE; deja int; jury int;

  BEGIN
  SELECT * INTO etu FROM etudiant WHERE numEtud = NEW.numEtud;
  IF NOT FOUND THEN
    RAISE NOTICE 'L''étudiant n''existe pas.' ; RETURN NULL ;
  END IF;
  IF etu.numLic != 2 THEN
    RAISE NOTICE 'L''étudiant n''est pas en L2.' ; RETURN NULL ;
  END IF ;

  SELECT numEtud INTO deja FROM nouveauL3 WHERE numEtud = NEW.numEtud;
  IF FOUND THEN
    RAISE NOTICE '% passe déjà en L3.', etu.nom; RETURN NULL ;
  END IF;

  SELECT codeCours INTO jury FROM cours WHERE intitule = 'Points de jury';
  INSERT INTO examen VALUES (etu.numEtud, jury, 10);
  RETURN NEW ;
  END ;
$$
LANGUAGE plpgsql;
```

La fonction pointsJury

```
CREATE FUNCTION pointsJury() RETURNS TRIGGER AS
$$
  DECLARE etu etudiant%ROWTYPE; deja int; jury int;

  BEGIN
  SELECT * INTO etu FROM etudiant WHERE numEtud = NEW.numEtud;
  IF NOT FOUND THEN
    RAISE NOTICE 'L''étudiant n''existe pas.' ; RETURN NULL ;
  END IF;
  IF etu.numLic != 2 THEN
    RAISE NOTICE 'L''étudiant n''est pas en L2.' ; RETURN NULL ;
  END IF ;

  SELECT numEtud INTO deja FROM nouveauL3 WHERE numEtud = NEW.numEtud;
  IF FOUND THEN
    RAISE NOTICE '% passe déjà en L3.', etu.nom; RETURN NULL ;
  END IF;

  SELECT codeCours INTO jury FROM cours WHERE intitule = 'Points de jury';
  INSERT INTO examen VALUES (etu.numEtud, jury, 10);
  RETURN NEW ;
  END ;
$$
LANGUAGE plpgsql;
```

Questions : Que se passe-t-il si le cours n'existe pas ?
S'il n'octroie pas assez d'ECTS ?

Ordre d'exécution des événements et des triggers

Lorsqu'un événement se produit :

- Étape 1 : les triggers **BEFORE** et **INSTEAD OF** sont déclenchés
- Étape 2 : l'événement (ou son remplaçant pour les vues) est effectué
- Étape 3 : les triggers **AFTER** sont déclenchés

Ordre d'exécution des événements et des triggers

Lorsqu'un événement se produit :

- Étape 1 : les triggers **BEFORE** et **INSTEAD OF** sont déclenchés
- Étape 2 : l'événement (ou son remplaçant pour les vues) est effectué
- Étape 3 : les triggers **AFTER** sont déclenchés

Si plusieurs triggers sont déclenchés par le même événement :

- Sous Postgres, les triggers déclenchés à la même étape sont exécutés par ordre alphabétique

Ordre d'exécution des événements et des triggers

Lorsqu'un événement se produit :

- Étape 1 : les triggers **BEFORE** et **INSTEAD OF** sont déclenchés
- Étape 2 : l'événement (ou son remplaçant pour les vues) est effectué
- Étape 3 : les triggers **AFTER** sont déclenchés

Si plusieurs triggers sont déclenchés par le même événement :

- Sous Postgres, les triggers déclenchés à la même étape sont exécutés par ordre alphabétique
- Les triggers **FOR EACH ROW** reçoivent les lignes telles que modifiées par les retours des triggers **BEFORE** ou **INSTEAD OF** qui les précèdent

Ordre d'exécution des événements et des triggers

Lorsqu'un événement se produit :

- Étape 1 : les triggers **BEFORE** et **INSTEAD OF** sont déclenchés
- Étape 2 : l'événement (ou son remplaçant pour les vues) est effectué
- Étape 3 : les triggers **AFTER** sont déclenchés

Si plusieurs triggers sont déclenchés par le même événement :

- Sous Postgres, les triggers déclenchés à la même étape sont exécutés par ordre alphabétique
- Les triggers **FOR EACH ROW** reçoivent les lignes telles que modifiées par les retours des triggers **BEFORE** ou **INSTEAD OF** qui les précèdent
- Si un trigger **BEFORE** ... **FOR EACH ROW** ou **INSTEAD OF** renvoie **NULL**, les triggers **FOR EACH ROW** ne sont pas exécutés

Ordre d'exécution des événements et des triggers

Lorsqu'un événement se produit :

- Étape 1 : les triggers **BEFORE** et **INSTEAD OF** sont déclenchés
- Étape 2 : l'événement (ou son remplaçant pour les vues) est effectué
- Étape 3 : les triggers **AFTER** sont déclenchés

Si plusieurs triggers sont déclenchés par le même événement :

- Sous Postgres, les triggers déclenchés à la même étape sont exécutés par ordre alphabétique
- Les triggers **FOR EACH ROW** reçoivent les lignes telles que modifiées par les retours des triggers **BEFORE** ou **INSTEAD OF** qui les précèdent
- Si un trigger **BEFORE** ... **FOR EACH ROW** ou **INSTEAD OF** renvoie **NULL**, les triggers **FOR EACH ROW** ne sont pas exécutés
- Si un trigger lève une exception, toutes les étapes sont annulées

Triggers en cascade

Un trigger peut effectuer une mise-à-jour...

Triggers en cascade

Un trigger peut effectuer une mise-à-jour...
... qui déclenche un trigger...

Triggers en cascade

Un trigger peut effectuer une mise-à-jour...
... qui déclenche un trigger...
... qui effectue lui même une mise-à-jour...

Triggers en cascade

Un trigger peut effectuer une mise-à-jour...
... qui déclenche un trigger...
... qui effectue lui même une mise-à-jour...
... qui déclenche à son tour un nouveau trigger...

Triggers en cascade

Un trigger peut effectuer une mise-à-jour...
... qui déclenche un trigger...
... qui effectue lui même une mise-à-jour...
... qui déclenche à son tour un nouveau trigger...

Les triggers sont des outils puissants à utiliser avec parcimonie.

L'abus de triggers :

- peut provoquer des boucles infinies et des erreurs
- rend le code difficile à lire et à maintenir
- peut créer des interdépendances complexes et imprédictibles

Triggers en cascade

Un trigger peut effectuer une mise-à-jour...
... qui déclenche un trigger...
... qui effectue lui même une mise-à-jour...
... qui déclenche à son tour un nouveau trigger...

Les triggers sont des outils puissants à utiliser avec parcimonie.

L'abus de triggers :

- peut provoquer des boucles infinies et des erreurs
- rend le code difficile à lire et à maintenir
- peut créer des interdépendances complexes et imprédictibles

Question : traitement du côté applicatif ou par un trigger ?

Exercice : contrainte ou trigger ?

On considère le schéma de la base de données `magasin`.

Expliquer comment assurer les comportements suivants :

1. Lorsqu'on ajoute un produit au stock d'un magasin, si la quantité n'est pas renseignée, elle est automatiquement mise à 0.
2. Lorsque des post-its sont ajoutés à une commande, si la quantité n'est pas renseignée, elle est automatiquement mise à 50.
3. Les numéros des nouveaux clients sont tirés automatiquement.
4. Aucune facture ne contient plus de 200 exemplaires d'un produit.
5. Le total d'une commande ne peut pas dépasser plus de 5000 euros.
6. Il n'y a jamais deux magasins de même nom dans la même ville.
7. Les magasins qui ont le même nom pratiquent des tarifs identiques.

Procédures trigger en PL/pgSQL

Le langage PL/pgSQL

PL/pgSQL : langage procédural de PostgresQL

Utilisé pour l'écriture de fonctions et procédures stockées

Objectifs :

- Définition d'opérations complexes en plusieurs étapes
- Fonctions exécutées localement sur le serveur
  - Efficacité : moins de cycles de communication client / serveur
  - Sécurité : résultats intermédiaires non divulgués au client
- Écriture de la partie action des triggers

Dans ce cours :

- Minimum pour écrire et tester des triggers simples
- Ceci n'est pas un cours de programmation !
- Plus de détails sur `https://www.postgresql.org/docs/`

Création d'une procédure trigger

```
CREATE [ OR REPLACE ] FUNCTION nomFonction()
RETURNS TRIGGER AS
$$
  [ DECLARE
    nomVar typeVar ; ... ]

  BEGIN
    statement ; ...

  END ;
$$
LANGUAGE plpgsql;
```

Remarques :

- Les procédures trigger sont toujours déclarées sans arguments
- Elles doivent renvoyer soit **NULL** soit un enregistrement compatible avec l'événement déclenchant
- Toutes les variables sont déclarées dans la clause **DECLARE**

Variables prédéfinies

Variables automatiquement affectées à l'appel de la procédure :

- **OLD** et **NEW**
  Ligne avant et après modification (cas **FOR EACH ROW**)
- **TG_WHEN** : 'BEFORE', 'AFTER' ou 'INSTEAD OF'
  chaine indiquant le moment de déclenchement du trigger
- **TG_OP** : 'INSERT', 'UPDATE', 'DELETE' ou 'TRUNCATE'
  chaine indiquant le type de l'événement qui a déclenché le trigger
- **TG_NARGS**
  entier représentant le nombre d'arguments passés à la procédure par le trigger (similaire à la variable `int argc` du langage C)
- **TG_ARGV[ ]**
  tableau de chaînes représentant les arguments passés à la procédure par le trigger (similaire à la variable `char* argv[]` du langage C)

Et bien d'autres, renseignant le nom du trigger, de la table affectée...

Structures de contrôle

- Retour de fonction

```
RETURN expression ;
```

- Tests conditionnels

```
IF condition THEN statement ; ...
[ ELSE IF condition THEN statement ; ... ]
[ ELSE statement ; ... ]
END IF ;
```

- Boucles

```
FOR nomVar IN debut..fin LOOP
  statement ; ...
END LOOP ;
-- nomVar est automatiquement déclaré comme un entier
```

```
WHILE condition LOOP
  statement ; ...
END LOOP ;
-- on peut aussi utiliser les commandes usuelles CONTINUE et EXIT
```

Utiliser le résultat d'une requête

Requête renvoyant une seule ligne

```
SELECT expression INTO nomVar1, ... , nomVarN FROM ... ;
```

- Les nombres et types des variables doivent être compatibles avec le type des lignes renvoyées par la requêtes
- Le type nomTable%**ROWTYPE** représente un n-uplet compatible avec le type des lignes de la table nomTable
- Si la requête renvoie plusieurs lignes, seule la première est affectée
  Si la requête renvoie zéro ligne, les variables sont mises à **NULL**
- La variable spéciale **FOUND** contient **True** ou **False** selon que la dernière requête exécutée a renvoyé ou non au moins une ligne

Ex :

```
DECLARE etu1 etudiant%ROWTYPE;
BEGIN
  SELECT * FROM etudiant INTO etu1 WHERE nom = 'Delacour';
  IF NOT FOUND THEN RAISE 'Etudiant inconnu';
  -- suite du code ici...
END;
```

Utiliser le résultat d'une requête

Requête renvoyant plusieurs lignes

```
FOR nomVar1, ..., nomVarN IN requete LOOP
  statement ; ...
END LOOP
```

Remarques :

- Les valeurs des variables parcourent les lignes renvoyées
  Le corps de la boucle est exécutée une fois pour chaque ligne
  (similaire à la construction `for var in list:` de Python)
- Les variables doivent être compatibles avec la requête (cf. **INTO**)

Ex :

```
DECLARE etu etudiant%ROWTYPE;
BEGIN
FOR etu IN SELECT * FROM etudiant WHERE numlic = 2 LOOP
  -- suite du code à compléter ici, exécuté pour chaque etudiant en L2...
END LOOP;
END;
```

Messages et erreurs

```
RAISE [ LEVEL ] message, var1, ... , varN;
```

où **LEVEL** est un niveau de priorité parmi :

```
DEBUG | LOG | INFO | NOTICE | WARNING | EXCEPTION
```

Remarques :

- **EXCEPTION** est le niveau par défaut
- **EXCEPTION** lève une exception et annule donc toute la mise-à-jour
- Les autres niveaux génèrent simplement des messages
- L'administrateur du système peut choisir le niveau des messages auxquels les utilisateurs ont accès ou qui sont reportés dans les logs
- PL/pgSQL permet de rattraper les exceptions, référez-vous à la doc

Traitement automatique des factures

On considère une base de données respectant le schéma suivant :

client(numcli, nom, prenom, ville, tel)
produit(idpro, libelle, couleur)
magasin(idmag, nom, ville, tel)
stocke(idmag, idpro, prixUnit, quantite)
facture(idfac, date, numcli, idmag)
contient(idfac, idpro, prixUnit, quantite)
fidelite(numcarte, dateCreation, points, numcli, idmag)

FK : les attributs numcli, idpro, idmag et idfac dans les tables *stocke*, *facture*, *contient* et *fidelite* font référence aux attributs du même nom dans les tables *client*, *produit*, *magasin* et *facture*.

Écrire des triggers pour automatiser autant que raisonnable les conséquences de l'insertion d'une ligne dans la table *contient*.

Suggestions :

- Vérification et mise-à-jour des stocks
- Création et mise-à-jour des cartes de fidélité

Exercice : écriture de triggers

On considère une base de données respectant le schéma suivant :

utilisateur(`uid`, `nom`, `prenom`, `email`)
projet(`pid`, `titre`, `statut`, `requis`)
soutient(`uid`, `pid`, `montant`)

FK : les attributs `uid` et `pid` dans la table *soutient* font référence aux attributs `uid` et `pid` dans les tables *utilisateur* et *projet*.

Écrire des triggers pour automatiser les opérations suivantes :

1. Lorsque le montant total des soutiens d'un projet passe au dessus du montant requis, le statut du projet passe à *"validé"*
2. Lorsqu'un projet est *"annulé"*, les soutiens du projet sont supprimés
3. Seuls les projets *"en attente"* peuvent recevoir de nouveaux soutiens
4. Lorsqu'un projet reçoit un soutien, une vue matérialisée contenant les montants totaux promis à chaque projet est mise-à-jour
5. Lorsque le statut d'un projet passe à *"validé"*, la liste des utilisateurs qui l'ont soutenu est écrite dans une table de log