

Procédures Stockées et Triggers

Anne-Cécile Caron, Anne Étien, Mikaël Monet,
Sylvain Salvati

Outline

① Procédures Stockées

② Trigger

Procédures stockées : Définition

- Une **procédure stockée** est une procédure manipulant des données qui est stockée dans la base de données.
- Correspond à du code métier au cœur de la base de données.
- Ces programmes peuvent être appelés par les programmes clients, et sont **exécutés par le serveur**.
- Norme SQL-99 : Les schémas des SGBD relationnels s'enrichissent de *Persistent Stored Modules (PSM)*.
- Bien que ces procédures stockées soient dans la norme depuis plus de 20 ans, il reste de **nombreuses variantes de langages** selon les éditeurs.
 - **l'application n'est plus indépendante du SGBD**

Procédures stockées : Avantages

- ❶ **Sécurité** : Tous les clients de la base de données, quels qu'ils soient (Client-Serveur, Web, etc...), bénéficient de **la même version du code** de la procédure stockée.

Exemple, considérons une procédure stockée qui permet de connaître le prix d'un article dans une gestion commerciale.

- Si les règles changent, on modifie cette procédure stockée et tous les clients de l'application en bénéficient.
- Si les paramètres d'appel et de sortie de la procédure stockée ne changent pas, les clients n'ont même pas besoin d'être modifiés.

Procédures stockées : Avantages

- ① **Sécurité** : Tous les clients de la base de données, quels qu'ils soient (Client-Serveur, Web, etc...), bénéficient de **la même version du code** de la procédure stockée.

Exemple, considérons une procédure stockée qui permet de connaître le prix d'un article dans une gestion commerciale.

- Si les règles changent, on modifie cette procédure stockée et tous les clients de l'application en bénéficient.
- Si les paramètres d'appel et de sortie de la procédure stockée ne changent pas, les clients n'ont même pas besoin d'être modifiés.

- ② **Performances** : l'usage de procédures stockées permet de **diminuer les communications entre le serveur et le client**, tout en bénéficiant de la puissance d'un langage procédural

- Quand on intègre du SQL dans l'application client : exécution séparée de chaque requête SQL par le serveur de bases de données.
- L'application client envoie chaque requête au serveur de bases de données, attend que celui-ci la traite, reçoit les résultats, fait quelques traitements, et enfin envoie d'autres requêtes au serveur.
- Tout ceci induit des communications interprocessus et peut aussi induire une surcharge du réseau si votre client est sur une machine différente du serveur de bases de données.

Procédures stockées : Particularité

- Ce n'est pas un interfaçage avec un langage de programmation (JDBC, lib. Postgres Query)
- Peuvent être lancées dans une commande SQL
- Plusieurs langages reconnus pour Postgres :
 - SQL
 - PL/pgSQL
 - PL/TCL, PL/Perl
 - C (bibliothèques dynamiques)
- Postgres permettait initialement uniquement la définition de **fonctions** stockées. Les procédures ont été ajoutées dans la version 11 (compatibilité avec la norme). Pour simplifier, et faute de temps, nous ne présenterons ici que les fonctions.

Fonctions

- **Trois nouvelles commandes** : Create Function, Create or Replace Function et Drop Function
- **Définir la signature de la fonction** :

```
CREATE OR REPLACE FUNCTION name ( [ ftype [, ...] ] )  
RETURNS rtype AS definition LANGUAGE 'langname'
```

Fonctions

- **Trois nouvelles commandes** : Create Function, Create or Replace Function et Drop Function

- **Définir la signature de la fonction** :

```
CREATE OR REPLACE FUNCTION name ( [ ftype [, ...] ] )  
RETURNS rtype AS definition LANGUAGE 'langname'
```

- ftype correspond au type des paramètres de la fonction
- rtype correspond au type de retour de la fonction
- definition correspond au corps de la fonction ; C'est une chaîne de caractères, que l'on délimite en général par des doubles dollars plutôt que des quotes.
- langname correspond au langage dans lequel le corps de la fonction est écrit.

Le langage PL/pgSQL

- Analogue à Oracle PLSQL et à la norme SQL-99
- PL/pgSQL est un **langage procédural**, il permet donc
 - de déclarer des variables (DECLARE)
 - des structures de contrôle (IF, WHILE, FOR)
 - de quitter et retourner un résultat (RETURN)
- C'est un langage adapté à SQL, il permet :
 - d'exécuter des instructions SQL
 - d'itérer sur les résultats d'un SELECT grâce à une boucle FOR.
 - d'avoir une forme spéciale de SELECT qui autorise à stocker le résultat dans des variables (SELECT INTO)

Structure de bloc

- Le texte complet de la définition d'une fonction doit être **un bloc**.
- Un bloc est défini comme :
[<<label>>]
[DECLARE
déclarations]
BEGIN
instructions
END;
- Chaque déclaration et chaque expression au sein du bloc est terminée par un point-virgule.

Types de variable

- Les **types simples** : integer, char, char(x), text, . . .
- Le **type enregistrement** : record
- **Déclarer un synonyme pour un paramètre**
`DECLARE numero ALIAS FOR $1 ;`
- **Clause %TYPE** pour obtenir le type d'une colonne :
`DECLARE nomJoueur joueur.nom%TYPE ;`
donne à la variable nomJoueur le type de la colonne nom de la table joueur
- **Clause %rowType** pour obtenir le type de la ligne d'une table, et éviter de passer par un type record explicite.
`DECLARE recordJoueur joueur%ROWTYPE ;`
Donne à la variable recordJoueur la structure des tuples de joueur.

Variables prédéfinies

- TRUE et FALSE variables booléennes
- FOUND : variable booléenne, résultat d'une commande select ;
détermine si une assignation a réussi (c'est à dire qu'au moins une
ligne a été renvoyée par la requête)
- TRIGGER : valable pour les déclencheurs comme type de retour d'une
fonction.

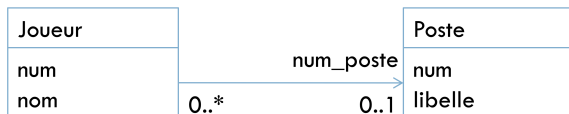
Instructions de base

- Affectation : `nomVariable := expression ;`
- Structures de contrôle :
 - `IF expr THEN ...{ELSIF expr THEN ... } [ELSE ...] END IF;`
 - `WHILE expr LOOP ... END LOOP;`
 - `FOR var IN [REVERSE] expr .. expr LOOP ... END LOOP;`
 - `FOR record IN exprSelect LOOP ... END LOOP;`
- Exécuter une expression ou requête sans récupérer le résultat
`PERFORM expr ;`
- Renvoyer les données d'une fonction
`RETURN expr ;`
- Rapporter des messages et lever des erreurs : `RAISE [level]`
`'format' [USING option = expression];`
`RAISE EXCEPTION 'Joueur inconnu' ;`
`RAISE NOTICE 'Joueur % inexistant', idJoueur USING`
`ERRCODE = '23505';`

Note : Langage interprété, pas d'erreur détectée à la compilation. A l'exécution, les messages d'erreurs ne sont pas très explicites.

Ex : oublier un THEN provoque une erreur non détaillée

Exemple : Association (0,1) - *



```
create table Poste(  
num numeric(2) primary key,  
libelle varchar(20) not null  
);
```

```
create table Joueur(  
num numeric(2) primary key,  
nom varchar(20) not null,  
num_poste numeric(2) references Poste  
);
```

Exemple : Association (0,1) - *

```
CREATE OR REPLACE FUNCTION libPoste(numPoste poste.num%type)
RETURNS varchar
AS $$DECLARE ch varchar;
BEGIN
SELECT libelle INTO ch FROM poste p
WHERE numPoste = p.num ;
RETURN ch;
END;$$
LANGUAGE 'plpgsql';
```

- le code PL/pgSQL est encadré par \$\$
- la requête utilise le paramètre numPoste
- si la requête ne ramène aucun résultat, libPoste retourne NULL.
- Appel de la fonction dans une requête SELECT

```
select libPoste(1) ;
-- renvoie 'goal'
```

```
select nom, libPoste(num_poste) from joueur ;
```

Exemple : Association * - *



```
create table poste(  
  num serial primary key,  
  libelle varchar(20) not null  
); -- serial veut dire integer avec default nextval('poste_num_seq')
```

```
create table Joueur(  
  num serial primary key,  
  nom varchar(20) not null  
);
```

```
create table joueur_poste(  
  numj integer references joueur,  
  nump integer references poste,  
  primary key(numj, nump)  
);
```

```
insert into poste(libelle) values ('goal');  
select setval('joueur_num_seq',10);  
insert into joueur(nom) values ('Durand'); -- Solène Durand, gardienne  
...
```


Exemple : Association * - *

Poste		Joueur		Joueur_Poste	
num	libelle	num	nom	numj	nump
1	'goal'	11	'Durand'	11	1
2	'defenseur.e'	12	'Renard'	14	3
3	'milieu'	13	'Bilbault'	14	4
4	'attaquant.e'	14	'Baltimore'		

La fonction affecteA

```
CREATE OR REPLACE FUNCTION affecteA(numJoueur joueur.num%type)
RETURNS varchar AS $$
DECLARE ch varchar ; unPoste record ;
BEGIN
    ch := '';
    for unPoste in select libelle from joueur_poste
        join poste on joueur_poste.nump=poste.num
        where numj=numjoueur loop
        ch := ch || unPoste.libelle || ', ' ;
    end loop ;
    if length(ch) > 0 then -- on enlève la dernière virgule
        return substring(ch for length(ch)-2) ;
    else return null ; end if;
END ;$$
LANGUAGE 'plpgsql';

select nom, affecteA(num) from joueur ;
```

nom	affecteA
'Durand'	'goal'
'Renard'	NULL
'Bilbault'	NULL
'Baltimore'	'milieu, attaquant.e'

Exemple : Assurer les contraintes d'intégrité référentielle (CIR)

- On définit une fonction qui prend 2 chaînes de caractères en paramètres :

```
FUNCTION inserer(nomJoueur joueur.nom%type,  
                nomPoste poste.libelle%type)  
    RETURNS boolean
```

- Cette fonction insère dans la base un nouveau joueur s'il n'existe pas, un nouveau poste s'il n'existe pas, et le fait que ce joueur est à ce poste.
- Si toutes les informations sont déjà dans la base, provoque une erreur.

```
select inserer('Diacre', 'entraîneur.e') ;
```

inserer

TRUE

```
select inserer('Diacre','entraîneur.e') ;
```

ERROR: déjà inséré

Pour assurer les CIR

```
CREATE OR REPLACE FUNCTION inserer(nomJoueur joueur.nom%type,  
                                   nomPoste poste.libelle%type)  
RETURNS boolean AS $$  
DECLARE  
numPoste integer ; numJoueur integer ; tester record ;  
BEGIN  
-- recherche du poste  
SELECT num INTO numPoste from poste where libelle=nomPoste ;  
IF NOT FOUND THEN -- creation poste  
    insert into poste(libelle) values (nomPoste) RETURNING num INTO numPoste;  
END IF ;  
-- recherche du joueur  
SELECT num INTO numJoueur FROM joueur WHERE nom=nomJoueur ;  
IF NOT FOUND THEN -- creation joueur  
    insert into joueur (nom) values (nomJoueur) RETURNING num INTO numJoueur ;  
END IF ;  
-- mise a jour Joueur_poste  
SELECT * INTO tester FROM Joueur_poste WHERE numj=numJoueur AND nump=numPoste ;  
IF FOUND THEN RAISE EXCEPTION 'déjà inséré' ;  
END IF ;  
insert into Joueur_poste values (numJoueur, numPoste);  
return true ;  
END ;$$  
LANGUAGE 'plpgsql' ;
```

Outline

① Procédures Stockées

② Trigger

Trigger

- **Définition** : Traitement qui est déclenché automatiquement par un événement
- En postgresQL, on définit une fonction trigger, puis on relie cette fonction à une table et un événement.

```
CREATE TRIGGER name BEFORE | AFTER  
event [OR ...]  
ON table FOR EACH ROW | STATEMENT  
EXECUTE FUNCTION func ( arguments )
```
- Les événements déclencheurs sont INSERT, DELETE, UPDATE (on met de côté le TRUNCATE, un peu particulier).
- FOR EACH ROW signifie que la fonction est appelée pour chaque ligne insérée/supprimée/modifiée
- FOR EACH STATEMENT signifie que la fonction est appelée une fois par instruction SQL
- BEFORE | AFTER indique quand la fonction est exécutée par rapport à l'événement déclencheur.

Variables prédéfinies

Les variables suivantes peuvent être utilisées dans le corps de la fonction :

Nom	Type	Signification
new	record	valeur du nouvel enregistrement (insert, update)
old	record	valeur de l'enregistrement initial (update, delete)
tg_name	name	nom du trigger actif
tg_relname	name	nom de la table courante
tg_when	text	'before' ou 'after' selon la définition du trigger
tg_level	text	'row' ou 'statement' selon la définition du trigger
tg_op	text	'insert' , 'update' ou 'delete'
tg_nargs	integer	nombre d'arguments
tg_argv	text[]	les arguments

Remarque : les variables new et old n'ont de sens que pour les triggers
For each row.

Exemple : Vérification de la longueur d'un mot de passe

```
CREATE TABLE password(  
  numref integer PRIMARY KEY,  
  password text NOT NULL) ;
```


Exemple : Vérification de la longueur d'un mot de passe

```
CREATE TABLE password(  
  numref integer PRIMARY KEY,  
  password text NOT NULL) ;  
  
CREATE OR REPLACE FUNCTION verifLongPassword()  
RETURNS TRIGGER AS $$  
BEGIN  
  IF length(new.password) < 6 THEN  
    RAISE EXCEPTION 'mot de passe % trop court', new.password ;  
  END IF ;  
  RETURN new ;  
END ;$$  
LANGUAGE 'plpgsql' ;  
  
CREATE TRIGGER tgVerifLongPassword  
BEFORE update OR insert  
ON password FOR EACH ROW  
EXECUTE FUNCTION verifLongPassword() ;
```

Exemple (suite)

```
insert into password values (1, 'azertyuiop');
```

Exemple (suite)

```
insert into password values (1, 'azertyuiop');  
insert into password values (2, 'azer') ;  
ERROR: mot de passe azer trop court
```

Exemple (suite)

```
insert into password values (1, 'azertyuiop');  
insert into password values (2,'azer') ;  
ERROR: mot de passe azer trop court  
select * from password ;
```

numref	password
1	azertyuiop

Trigger : Synthèse

- Fonctions dont l'exécution **sur le serveur** est déclenchée par les instructions du DML.
- Il existe d'autres types de triggers :
 - triggers **instead of** sur les vues, dont le code vient se substituer à l'instruction SQL.
 - triggers **event** qui se déclenchent sur des événements du DDL
- Les triggers sont pratiques pour
 - **vérifier des contraintes complexes** qu'on ne peut pas déclarer au niveau du schéma.
 - gérer de la **redondance** en cas de dénormalisation du schéma (cf cours de la semaine prochaine)
- Mais ils ont aussi des inconvénients :
 - **complexité cachée** (l'utilisateur pense faire une simple instruction SQL)
 - **interdépendance entre triggers** qui peut être difficile à gérer, cascade de triggers.