

# Procédures stockées et triggers

## Olivier Caron

Polytech Lille  
Avenue Paul Langevin Cité Scientifique Lille 1  
59655 Villeneuve d'Ascq cedex

<http://ocaron.plil.fr>  
[Olivier.Caron@polytech-lille.fr](mailto:Olivier.Caron@polytech-lille.fr)



## Plan global du cours BDA

- Programme GIS 3 : Bases de données (modèle relationnel, SQL, programmation via PHP)

## Plan global du cours BDA

- Programme GIS 3 : Bases de données (modèle relationnel, SQL, programmation via PHP)
- Programme GIS 4 : Système de gestion de bases de données :

## Plan global du cours BDA

- Programme GIS 3 : Bases de données (modèle relationnel, SQL, programmation via PHP)
- Programme GIS 4 : Système de gestion de bases de données :
  - Plusieurs utilisateurs → sécurité, accès concurrents (BDA)

## Plan global du cours BDA

- Programme GIS 3 : Bases de données (modèle relationnel, SQL, programmation via PHP)
- Programme GIS 4 : Système de gestion de bases de données :
  - Plusieurs utilisateurs → sécurité, accès concurrents (BDA)
  - Aspect programmation au sein du SGBD : procédures stockées, triggers (BDA)

## Plan global du cours BDA

- Programme GIS 3 : Bases de données (modèle relationnel, SQL, programmation via PHP)
- Programme GIS 4 : Système de gestion de bases de données :
  - Plusieurs utilisateurs → sécurité, accès concurrents (BDA)
  - Aspect programmation au sein du SGBD : procédures stockées, triggers (BDA)
  - Aspect programmation en dehors du SGBD : le protocole JDBC (cours SIO), JPA (cours SIO, AL)

## Plan global du cours BDA

- Programme GIS 3 : Bases de données (modèle relationnel, SQL, programmation via PHP)
- Programme GIS 4 : Système de gestion de bases de données :
  - Plusieurs utilisateurs → sécurité, accès concurrents (BDA)
  - Aspect programmation au sein du SGBD : procédures stockées, triggers (BDA)
  - Aspect programmation en dehors du SGBD : le protocole JDBC (cours SIO), JPA (cours SIO, AL)
  - D'autres modèles de données : modèle Objet-Relationnel (BDA, SIO, AL), modèle XML (BDA)

## Les procédures stockées

- Procédure intégrée au SGBD



## Les procédures stockées

- Procédure intégrée au SGBD
- Ce n'est pas un interfaçage avec un langage de programmation (JDBC, lib. Postgres Query)

## Les procédures stockées

- Procédure intégrée au SGBD
- Ce n'est pas un interfaçage avec un langage de programmation (JDBC, lib. Postgres Query)
- Peuvent être lancées dans une commande SQL

## Les procédures stockées

- Procédure intégrée au SGBD
- 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 :

## Les procédures stockées

- Procédure intégrée au SGBD
- 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 :
  - SQL

## Les procédures stockées

- Procédure intégrée au SGBD
- 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 :
  - SQL
  - PL/PGSQL (similaire à Oracle PLSQL)

## Les procédures stockées

- Procédure intégrée au SGBD
- 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 :
  - SQL
  - PL/PGSQL (similaire à Oracle PLSQL)
  - PL/TCL, PL/Perl

## Les procédures stockées

- Procédure intégrée au SGBD
- 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 :
  - SQL
  - PL/PGSQL (similaire à Oracle PLSQL)
  - PL/TCL, PL/Perl
  - C (bibliothèques dynamiques)

## Fonctions prédéfinies

- Exemples: `now()`, `length()`, `trim()`, `upper()`, ...

- Soit la table `poste`

num	libelle
1	'directeur'
2	'ingenieur'
3	'agent'

```
select libelle, length(libelle) as longueur from poste
libelle      longueur
-----
directeur    9
ingenieur    9
agent        5
```



## Les fonctions

- Deux nouvelles commandes : Create Function ou mieux 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'
```

## Exemple de fonction SQL

```
CREATE OR REPLACE FUNCTION nouveauSalaire(float)  
RETURNS float  
AS $$ select $1*.10 + $1 ;$$  
LANGUAGE 'sql' ;
```

```
select nouveauSalaire(8000.0) ;  
nouveauSalaire
```

---

8800.0

(1 row)

- $\$i$  désigne le  $i^{\text{ème}}$  paramètre
- Les colonnes de tables conformes au type de paramètre sont applicables

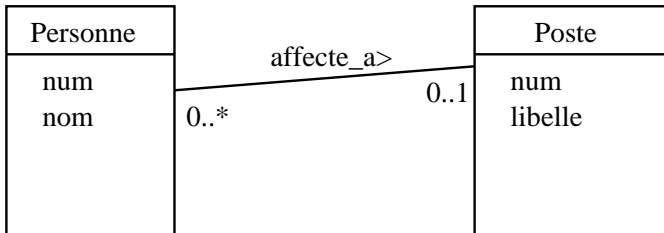
## Idéal pour les données calculées

```
CREATE TABLE produits ( num INT PRIMARY KEY,  
                        designation text UNIQUE, prix_ht float ) ;  
INSERT INTO produits VALUES (1, 'scanner' ,2000.0) ;  
INSERT INTO produits VALUES (2, 'graveur' ,1500.0) ;  
CREATE OR REPLACE FUNCTION taxe(float) RETURNS float  
AS $$ SELECT $1*.196; $$ LANGUAGE 'sql' ;  
  
SELECT designation , prix_ht , taxe(prix_ht),  
       prix_ht+taxe(prix_ht) as prix_ttc FROM produits ;
```

designation	prix_ht	taxe	prix_ttc
scanner	2000	392	2392
graveur	1500	294	1794

(2 rows)

## Idéal pour les associations



```
CREATE OR REPLACE FUNCTION affecte_a(integer)
RETURNS text
AS $$ SELECT libelle from poste
      WHERE cast(num as integer)=cast($1 as integer) ;$$
LANGUAGE 'SQL' ;
```

## L'instruction CASE

```
CREATE OR REPLACE FUNCTION type_carte(integer)  
RETURNS text  
AS $$ SELECT CASE  
    WHEN $1 <= 25 THEN CAST('carte_jeune' AS text)  
    WHEN $1 >=60 THEN CAST('carte_3ieme_age' AS text)  
    WHEN $1>25 AND $1<60 THEN CAST('rien_!' AS text)  
END ; $$  
LANGUAGE 'sql' ;
```

```
select type_carte(22) ;  
type_carte
```

---

```
carte jeune  
(1 row)
```

## L'instruction NULLIF

- Syntaxe : `NULLIF(input, value)`

## L'instruction NULLIF

- Syntaxe : `NULLIF (input, value)`
- Retourne `input` si `input=value`

## L'instruction NULLIF

- Syntaxe : `NULLIF(input, value)`
- Retourne `input` si `input=value`
- Retourne `NULL` sinon



## L'instruction NULLIF

- Syntaxe : `NULLIF (input, value)`
- Retourne `input` si `input=value`
- Retourne `NULL` sinon
- Parfois utile pour comparer des colonnes



## Illustration : les séquences (1/2)

- But : génération de clés automatiques

## Illustration : les séquences (1/2)

- But : génération de clés automatiques
- Chaque séquence définie par un nom :  
`CREATE SEQUENCE nomSequence`



## Illustration : les séquences (1/2)

- But : génération de clés automatiques
- Chaque séquence définie par un nom :  
`CREATE SEQUENCE nomSequence`

- Utilisation de trois fonctions :

<b>nom fonction</b>	<b>signification</b>
<code>int nextval('nom')</code>	retourne le prochain nombre, incrémente le compteur
<code>int currval('nom')</code>	retourne la dernière valeur retournée par <code>nextval</code>
<code>setval('nom', newval)</code>	positionne le compteur



## Illustration : les séquences (2/2)

- Exploitation contrainte DEFAULT :

```
CREATE SEQUENCE personne_seq ;  
CREATE TABLE personne (  
    pers_id integer DEFAULT nextval( 'personne_seq' ),  
    nom varchar(20)  
    age integer  
) ;  
INSERT INTO personne (nom,age) VALUES ( 'dupont' ,23) ;
```

## Le langage PL/PGSQL (analogue à Oracle PLSQL)

- SQL ne fait pas tout (ex : cloture transitive)

## Le langage PL/PGSQL (analogue à Oracle PLSQL)

- SQL ne fait pas tout (ex : cloture transitive)
- PL/PGSQL permet :

## Le langage PL/PGSQL (analogue à Oracle PLSQL)

- SQL ne fait pas tout (ex : cloture transitive)
- PL/PGSQL permet :
  - de déclarer des variables (DECLARE)



## Le langage PL/PGSQL (analogue à Oracle PLSQL)

- SQL ne fait pas tout (ex : cloture transitive)
- PL/PGSQL permet :
  - de déclarer des variables (DECLARE)
  - d'avoir une forme spéciale de `SELECT` qui autorise à stocker le résultat dans des variables (`SELECT INTO`)

## Le langage PL/PGSQL (analogue à Oracle PLSQL)

- SQL ne fait pas tout (ex : cloture transitive)
- PL/PGSQL permet :
  - de déclarer des variables (DECLARE)
  - d'avoir une forme spéciale de `SELECT` qui autorise à stocker le résultat dans des variables (`SELECT INTO`)
  - des structures de contrôle (IF, WHILE)

## Le langage PL/PGSQL (analogue à Oracle PLSQL)

- SQL ne fait pas tout (ex : cloture transitive)
- PL/PGSQL permet :
  - de déclarer des variables (DECLARE)
  - d'avoir une forme spéciale de `SELECT` qui autorise à stocker le résultat dans des variables (`SELECT INTO`)
  - des structures de contrôle (IF, WHILE)
  - quitter et retourner un résultat (RETURN)

## Exemple PL/PGSQL (1)

```
CREATE OR REPLACE FUNCTION affecte_a(integer)  
RETURNS text  
AS $$ DECLARE ch text;  
    BEGIN  
        SELECT INTO ch CAST(libelle AS text)  
        FROM poste  
        WHERE CAST(num as integer)=CAST($1 as integer) ;  
    RETURN ch;  
    END; $$  
LANGUAGE 'plpgsql';
```

## PL/PGSQL : les types de variables

- 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 ;
```
- Clause %rowType pour obtenir le n-uple d'une table  

```
DECLARE recordJoueur joueur%ROWTYPE ;
```

## PL/PGSQL : les variables prédéfinies

- FOUND : variable booléenne, résultat d'une commande select

## PL/PGSQL : les variables prédéfinies

- FOUND : variable booléenne, résultat d'une commande `select`
- Signature de fonction, type de retour :

## PL/PGSQL : les variables prédefinies

- FOUND : variable booléenne, résultat d'une commande `select`
- Signature de fonction, type de retour :
  - OPAQUE : pour définir une procédure



## PL/PGSQL : les variables prédéfinies

- FOUND : variable booléenne, résultat d'une commande `select`
- Signature de fonction, type de retour :
  - OPAQUE : pour définir une procédure
  - TRIGGER :valable pour les déclencheurs

## PL/PGSQL : Les structures de contrôles

```
IF expr THEN      WHILE expr LOOP      FOR record IN expr_select LOOP
  ...              ...              ...
[ ELSE ]          END LOOP ;        END LOOP ;
...
END IF ;

PERFORM nom_procedure(...) ;      nom_variable := expression ;

RAISE EXCEPTION message ;        RAISE NOTICE message ;

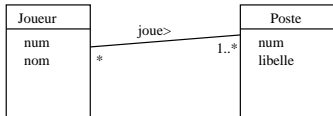
RETURN expression ;
```

- Langage interprété : pas d'erreur détectée à la compilation
- A l'exécution : messages d'erreurs non explicites
- Ex: manque THEN provoque une erreur non détaillée

## Idéal pour toutes les associations

- Les tables joueur et poste :

joueur		poste	
num	nom	num	libelle
1	'Lloris'	1	'goal'
2	'Debuchy'	2	'defenseur'
		3	'milieu'
		4	'attaquant'



```

select nom, affecte_a(num) from joueur ;
  nom      |      affecte_a
-----+-----
  Debuchy  | defenseur, milieu
  Lloris   | goal
  
```

## La fonction psql affecte\_a

```
CREATE FUNCTION affecte_a(integer) RETURNS text AS $$  
  DECLARE   numJoueur alias for $1 ; ch text ;  
           un_poste record ; i integer ;  
  
  BEGIN  
    i := 1 ;  
    for un_poste in select libelle from affecte_a ,poste  
      where numj=numjoueur and affecte_a.nump=poste.num  
    loop  
      if i=1 then  
        ch := un_poste.libelle ;  
      else  
        ch := ch || ', ' || un_poste.libelle ;  
      end if ;  
      i:= i + 1 ;  
    end loop ;  
    return ch ;  
  
END ; $$ LANGUAGE 'plpgsql' ;
```

## Peut assurer les CIR

- La fonction doit vérifier les contraintes d'intégrité

```
select inserer('Deschamps', 'entraîneur') ;  
inserer
```

```
-----
```

```
t  
(1 row)
```

```
select inserer('Deschamps', 'entraîneur') ;  
ERROR: déjà inséré
```

## La fonction plsql inserer (1/2)

```
CREATE OR REPLACE FUNCTION inserer(text , text)
RETURNS boolean AS $$
  DECLARE
    nom_joueur ALIAS for $1 ; nom_poste  ALIAS for $2 ;
    num_poste   integer ; num_joueur   integer ;
    tester      record ;
  BEGIN
    — recherche du poste
    SELECT INTO num_poste num from poste
    where libelle=nom_poste ;
    IF NOT FOUND THEN — creation poste
      SELECT INTO num_poste max(num) from poste ;
      num_poste := num_poste+1 ;
      insert into poste values (num_poste , nom_poste) ;
    END IF ;
```

## La fonction plsql inserer (2/2)

— recherche du joueur

```
SELECT INTO num_joueur num FROM joueur
```

```
WHERE nom=nom_joueur ;
```

IF NOT FOUND THEN — creation joueur

```
  SELECT INTO num_joueur max(num) from joueur ;
```

```
  num_joueur := num_joueur+1 ;
```

```
  insert into joueur values (num_joueur, nom_joueur) ;
```

```
END IF ;
```

— mise a jour affecte\_a

```
SELECT INTO tester * FROM affecte_a
```

```
WHERE numj=num_joueur AND nump=num_poste ;
```

```
IF FOUND THEN RAISE EXCEPTION 'déjà inséré' ;
```

```
END IF ;
```

```
  insert into affecte_a values (num_joueur, num_poste) ;
```

```
  return 't' ;
```

```
END ; $$ LANGUAGE 'plpgsql' ;
```

## Les triggers (déclencheurs)

- Un trigger est un programme qui se déclenche automatiquement suite à un évènement (norme SQL3)



## Les triggers (déclencheurs)

- Un trigger est un programme qui se déclenche automatiquement suite à un évènement (norme SQL3)
- L'évènement est une instruction qui **modifie** la base (insert, delete ou update)

## Les triggers (déclencheurs)

- Un trigger est un programme qui se déclenche automatiquement suite à un évènement (norme SQL3)
- L'évènement est une instruction qui **modifie** la base (insert, delete ou update)
- Les triggers font partie du schéma de la base, leur code compilé est conservé.

## Les triggers (déclencheurs)

- Un trigger est un programme qui se déclenche automatiquement suite à un évènement (norme SQL3)
- L'évènement est une instruction qui **modifie** la base (insert, delete ou update)
- Les triggers font partie du schéma de la base, leur code compilé est conservé.
- Les CIR sont gérés en interne par des triggers



## Création de triggers (1/2)

- On doit définir:

## Création de triggers (1/2)

- On doit définir:
  - La table concernée

## Création de triggers (1/2)

- On doit définir:
  - La table concernée
  - Les instructions qui déclenchent le trigger

## Création de triggers (1/2)

- On doit définir:
  - La table concernée
  - Les instructions qui déclenchent le trigger
  - le moment où le trigger va se déclencher par rapport à l'instruction (avant ou après)

## Création de triggers (1/2)

- On doit définir:
  - La table concernée
  - Les instructions qui déclenchent le trigger
  - le moment où le trigger va se déclencher par rapport à l'instruction (avant ou après)
  - Si le trigger se déclenche:



## Création de triggers (1/2)

- On doit définir:
  - La table concernée
  - Les instructions qui déclenchent le trigger
  - le moment où le trigger va se déclencher par rapport à l'instruction (avant ou après)
  - Si le trigger se déclenche:
    - Une seule fois pour toute l'instruction (**trigger instruction**)

## Création de triggers (1/2)

- On doit définir:
  - La table concernée
  - Les instructions qui déclenchent le trigger
  - le moment où le trigger va se déclencher par rapport à l'instruction (avant ou après)
  - Si le trigger se déclenche:
    - Une seule fois pour toute l'instruction (**trigger instruction**)
    - ligne par ligne concernée (**trigger ligne**)

## Création de triggers (1/2)

- On doit définir:
  - La table concernée
  - Les instructions qui déclenchent le trigger
  - le moment où le trigger va se déclencher par rapport à l'instruction (avant ou après)
  - Si le trigger se déclenche:
    - Une seule fois pour toute l'instruction (**trigger instruction**)
    - ligne par ligne concernée (**trigger ligne**)
  - la procédure stockée concernée par le trigger.

## Création de triggers (2/2)

- Syntaxe :

```
CREATE TRIGGER name { BEFORE | AFTER }  
  { event [OR ...] }  
ON table FOR EACH { ROW | STATEMENT }  
EXECUTE PROCEDURE func ( arguments )
```

## Trigger : des variables prédéfinies (1/2)

- Uniquement valable pour des trigger ligne:

Nom variable	Type	Signification
new	record	la valeur du nouvel enregistrement (insert, update)
old	record	la valeur de l'enregistrement initial (update, delete)

## Trigger : des variables prédéfinies (2/2)

Nom variable	Type	Signification
tg_name	name	le nom du trigger actif
tg_relname	name	le 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' selon la définition du trigger
tg_nargs	integer	nombre d'arguments
tg_argv	text[]	les arguments

## Trigger : un exemple (1/2)

```
CREATE TABLE t_password(numref int PRIMARY KEY,  
                        password text NOT NULL) ;  
CREATE OR REPLACE FUNCTION verif_long_password()  
RETURNS TRIGGER AS $$  
  BEGIN  
    IF length(new.password) < 6 THEN  
      RAISE EXCEPTION 'mot_de_passe_trop_court' ;  
    END IF ;  
    RETURN new ;  
  END ;  
$$ LANGUAGE 'plpgsql' ;  
CREATE TRIGGER verif_long_password  
  BEFORE update OR insert  
  ON t_password FOR EACH ROW  
  EXECUTE PROCEDURE verif_long_password() ;
```

## Trigger : un exemple (2/2)

```
insert into t_password values (1, 'azertyuiop') ; → OK
```

```
insert into t_password values (2, 'azer') ;
```

```
ERROR:  mot de passe trop court
```

```
select * from t_password ;
```

numref	password
--------	----------

1	azertyuiop
---	------------

```
DROP TRIGGER verif_long_password on t_password ;
```



## Aie, un trigger qui boucle ! (1/2)

```
create table log_erreurs(code_erreur int,  
    designation text, quand date) ;  
  
CREATE OR REPLACE FUNCTION insere_erreur()  
RETURNS TRIGGER AS $$  
BEGIN  
    IF new.code_erreur > 10 THEN  
        insert into log_erreurs  
            values (12, 'erreur_déetectée', now()) ;  
    END IF ;  
    return new ;  
END ;  
$$ LANGUAGE 'plpgsql' ;
```

## Aie, un trigger qui boucle ! (2/2)

```
CREATE TRIGGER detecte_erreur  
BEFORE insert ON log_erreurs  
FOR EACH ROW EXECUTE PROCEDURE insere_erreur () ;
```

```
INSERT INTO log_erreurs VALUES  
(100, 'pas_d''erreur' , now()) ;  
→ Kill !
```

## Conclusion

- Mécanisme efficace pour assurer la cohérence

## Conclusion

- Mécanisme efficace pour assurer la cohérence
- Très utilisé (ex: Sybase)

## Conclusion

- Mécanisme efficace pour assurer la cohérence
- Très utilisé (ex: Sybase)
- Simple mais...

## Conclusion

- Mécanisme efficace pour assurer la cohérence
- Très utilisé (ex: Sybase)
- Simple mais...
  - Difficile à déboguer (localisation erreur, triggers)

## Conclusion

- Mécanisme efficace pour assurer la cohérence
- Très utilisé (ex: Sybase)
- Simple mais...
  - Difficile à déboguer (localisation erreur, triggers)
  - Risque de blocage (deadlock)