
PL/PgSQL et Triggers

Bases de données 2

Thibaut MADELAINE

janvier 2022

Chers lectrices & lecteurs,

Cette formation PostgreSQL est issue des manuels Dalibo. Ils ont été repris par Thibaut MADELAINE pour rentrer dans le format universitaire avec Cours Magistraux, Travaux Dirigés (sans ordinateurs) et Travaux Pratiques (avec ordinateur).

Au-delà du contenu technique en lui-même, l'intention des auteurs est de transmettre les valeurs qui animent et unissent les développeurs de PostgreSQL depuis toujours : partage, ouverture, transparence, créativité, dynamisme... Le but premier de cette formation est de vous aider à mieux exploiter toute la puissance de PostgreSQL mais nous espérons également qu'elles vous inciteront à devenir un membre actif de la communauté en partageant à votre tour le savoir-faire que vous aurez acquis avec nous.

Nous mettons un point d'honneur à maintenir nos manuels à jour, avec des informations précises et des exemples détaillés. Toutefois malgré nos efforts et nos multiples relectures, il est probable que ce document contienne des oublis, des coquilles, des imprécisions ou des erreurs. Si vous constatez un souci, n'hésitez pas à le signaler sur le site gitlab https://gitlab.com/madtibo/cours_dba_pg_universite/-/issues !

Licence Creative Commons Attribution - Partage dans les Mêmes Conditions (CC BY-SA 4.0)

Vous êtes autorisé à :

- Partager — copier, distribuer et communiquer le matériel par tous moyens et sous tous formats
- Adapter — remixer, transformer et créer à partir du matériel pour toute utilisation, y compris commerciale.

Cette formation (diapositives, manuels et travaux pratiques) est sous licence **CC BY-SA**.

C'est un résumé (et non pas un substitut) de la licence.

L'Offrant ne peut retirer les autorisations concédées par la licence tant que vous appliquez les termes de cette licence.

Selon les conditions suivantes :

- Attribution — Vous devez créditer l'Œuvre, intégrer un lien vers la licence et indiquer si des modifications ont été effectuées à l'Œuvre. Vous devez indiquer ces informations par tous les moyens raisonnables, sans toutefois suggérer que l'Offrant vous soutient ou soutient la façon dont vous avez utilisé son Œuvre.
- Partage dans les Mêmes Conditions — Dans le cas où vous effectuez un remix, que vous transformez, ou créez à partir du matériel composant l'Œuvre originale, vous devez diffuser l'Œuvre modifiée dans les même conditions, c'est à dire avec la même licence avec laquelle l'Œuvre

originale a été diffusée. Vous êtes libres de redistribuer et/ou modifier cette création selon les conditions suivantes :

Pas de restrictions complémentaires — Vous n’êtes pas autorisé à appliquer des conditions légales ou des mesures techniques qui restreindraient légalement autrui à utiliser l’Oeuvre dans les conditions décrites par la licence.

Le texte complet de la licence est disponible à cette adresse: <http://creativecommons.org/licenses/by-sa/4.0/fr/legalcode>

PL/PgSQL et triggers



Programme de ce cours

- Semaine 1 : découverte de PostgreSQL
 - Semaine 2 : transactions et accès concurrents
 - Semaine 3 : missions du DBA
 - Semaine 4 : optimisation et indexation
 - **Semaine 5** : *PL/PgSQL* et triggers
 - Présentation du PL et des principes
 - Présentations de *PL/PgSQL* et des autres langages PL
 - Installation d'un langage PL
 - Détails sur *PL/PgSQL*
 - Gestion des erreurs
-

Introduction aux PL

- PL = Procedural Languages
- 3 langages activés par défaut :
 - C
 - SQL
 - *PL/PgSQL*

PL est l'acronyme de « Procedural Languages ». En dehors du C et du SQL, tous les langages acceptés par PostgreSQL sont des PL.

Par défaut, trois langages sont installés et activés : C, SQL et *PL/PgSQL*.

Autres langages de PL

- Nombreux autres langages disponibles
- Voici une liste non exhaustive :
 - *PL/Tcl*
 - *PL/Perl* et *PL/PerlU*
 - *PL/python*

- *PL/php*
- *PL/java*
- *PL/ruby*
- *PL/V8*
- *PL/R*
- *PL/sh*

Une quinzaine de langages sont disponibles, ce qui fait que la plupart des langages connus sont couverts. De plus, il est possible d'en ajouter d'autres.

Des langages de confiance... ou non

- les langages sont dits :
 - *de confiance (trusted)*
 - *ou non (untrusted)*
- Langage de confiance
 - ne permet que l'accès à la base de données
 - donc pas d'accès aux systèmes de fichiers, aux sockets réseaux, etc.

Les langages de confiance ne peuvent qu'accéder à la base de données. Ils ne peuvent pas accéder aux autres bases, aux systèmes de fichiers, au réseau, etc. Ils sont donc confinés, ce qui les rend moins facilement utilisable pour compromettre le système. *PL/PgSQL* est l'exemple typique. Mais du coup, ils offrent moins de possibilités que les autres langages.

Seuls les superutilisateurs peuvent créer une fonction dans un langage *Untrusted*. Par contre, ils peuvent ensuite donner les droits d'exécution à ces fonctions aux autres utilisateurs.

Qui est de confiance ?

- *Trusted* : *PL/PgSQL*, *SQL*, *PL/Perl*
 - *Untrusted* : *PL/PerlU*, *C*, *PL/Python...*
-

Les langages PL de PostgreSQL

- Les langages PL fournissent :
 - Des fonctionnalités procédurales dans un univers relationnel
 - Des fonctionnalités avancées du langage PL choisi
 - Des performances de traitement souvent supérieures à celles du même code côté client

Il peut y avoir de nombreuses raisons différentes à l'utilisation d'un langage PL. Simplifier et centraliser des traitements clients directement dans la base est l'argument le plus fréquent. Par exemple, une insertion complexe dans plusieurs tables, avec mise en place d'identifiants pour liens entre ces tables, peut évidemment être écrite côté client. Il est quelquefois plus pratique de l'écrire sous forme de PL. On y gagne :

- La centralisation du code : si plusieurs applications ont potentiellement besoin d'écrire le traitement, cela réduit d'autant les risques de bugs
- Les performances : le code s'exécute localement, directement dans le moteur de la base. Il n'y a donc pas tous les changements de contexte et échanges de message réseaux dûs à l'exécution de nombreux ordres *SQL* consécutifs
- La simplicité : suivant le besoin, un langage PL peut être bien plus pratique que le langage client.

Il est par exemple très simple d'écrire un traitement d'insertion/mise à jour en *PL/PgSQL*, le langage étant créé pour simplifier ce genre de traitements, et la gestion des exceptions pouvant s'y produire. Si vous avez besoin de réaliser du traitement de chaîne puissant, ou de la manipulation de fichiers, *PL/Perl* ou *PL/Python* seront probablement des options plus intéressantes, car plus performantes.

La grande variété des différents langages PL supportés par PostgreSQL permet normalement d'en trouver un correspondant aux besoins et aux langages déjà maîtrisés dans l'entreprise.

Les langages PL permettent donc de rajouter une couche d'abstraction et d'effectuer des traitements avancés directement en base.

Intérêts de *PL/PgSQL* en particulier

- Ajout de structures de contrôle au langage *SQL*
- Peut effectuer des traitements complexes
- Hérite de tous les types, fonctions et opérateurs définis par les utilisateurs
- Est « *Trusted* »
 - et facile à utiliser

La structure du *PL/PgSQL* est inspirée de l'ADA, donc proche du Pascal. La plupart des développeurs vieux développeurs ont eu l'occasion de faire du Pascal ou de l'ADA, et sont donc familiers avec la syntaxe de *PL/PgSQL*. Aussi, cette syntaxe est très proche de celle de PLSQL d'Oracle.

Elle permet d'écrire des requêtes directement dans le code PL sans déclaration préalable, sans appel à des méthodes complexes, ni rien de cette sorte. Le code SQL est mélangé naturellement au code PL, et on a donc un sur-ensemble de SQL qui est procédural.

PL/PgSQL étant intégré à PostgreSQL, il hérite de tous les types déclarés dans le moteur, même ceux que vous aurez rajouté. Il peut les manipuler de façon transparente.

PL/PgSQL est trusted. Tous les utilisateurs peuvent donc créer des procédures dans ce langage (par défaut). Vous pouvez toujours soit supprimer le langage, soit retirer les droits à un utilisateur sur ce langage (via la commande SQL «REVOKE»).

PL/PgSQL est donc raisonnablement facile à utiliser : il y a peu de complications, peu de pièges, il dispose d'une gestion des erreurs évoluée (gestion d'exceptions).

Les autres langages PL ont toujours leur intérêt

- Avantages des autres langages PL par rapport à *PL/PgSQL* :
 - Beaucoup plus de possibilités
 - Souvent plus performants pour la résolution de certains problèmes
- Mais un gros défaut :
 - Pas spécialisés dans le traitement de requêtes

Les langages PL «autres», comme *PL/Perl* et *PL/Python* (les deux plus utilisés après *PL/PgSQL*), sont bien plus évolués que *PL/PgSQL*. Par exemple, ils sont bien plus efficaces en matière de traitement de chaînes de caractères, ils possèdent des structures avancées comme des tables de hachages, permettent l'utilisation de variables statiques pour maintenir des caches, voire, pour leurs versions untrusted, peuvent effectuer des appels systèmes. Dans ce cas, il devient possible d'appeler un Webservice par exemple, ou d'écrire des données dans un fichier externe.

Il existe des langages PL spécialisés. Le plus emblématique d'entre eux est *PL/R*. R est un langage utilisé par les statisticiens pour manipuler de gros jeux de données. *PL/R* permet donc d'effectuer ces traitements R directement en base, traitements qui seraient très pénibles à écrire dans d'autres langages.

Il existe aussi un langage qui est, du moins sur le papier, plus rapide que tous les langages cités précédemment : vous pouvez écrire des procédures stockées en C, directement. Elles seront compilées à l'extérieur de PostgreSQL, en respectant un certain formalisme, puis seront chargées en indiquant la bibliothèque C qui les contient et leurs paramètres et types de retour. Attention, toute erreur dans votre code C est susceptible d'accéder à toute la mémoire visible par le processus PostgreSQL qui l'exécute, et donc de corrompre les données. Il est donc conseillé de ne faire ceci qu'en dernière extrémité.

Le gros défaut est simple et commun à tous ces langages : vous utilisez par exemple *PL/Perl*. Perl n'est pas spécialement conçu pour s'exécuter en tant que langage de procédures stockées. Ce que vous utilisez quand vous écrivez du *PL/Perl* est donc du code Perl, avec quelques fonctions supplémentaires (préfixées par *spi*) pour accéder à la base de données. L'accès aux données est donc rapide, mais assez malaisé au niveau syntaxique, comparé à *PL/PgSQL*.

Un autre problème des langages PL (autre que C et *PL/PgSQL*), c'est que ces langages n'ont pas les mêmes types natifs que PostgreSQL, et s'exécutent dans un interpréteur relativement séparé. Les performances sont donc moindres que *PL/PgSQL* et C pour les traitements dont le plus consommateur est l'accès des données. Souvent, le temps de traitement dans un de ces langages plus évolués est tout de même meilleur grâce au temps gagné par les autres fonctionnalités (la possibilité d'utiliser un cache, ou une table de hachage par exemple).

Installation

- *PL/PgSQL* compilé et installé par défaut
- Paquets Debian et RedHat :
 - *PL/PgSQL* par défaut
 - de nombreux autres disponibles
- Autres langages à compiler explicitement

Voici la liste des langages PL disponibles par paquets sur Debian :

```
$ apt-cache search postgresql | grep -i procedural | grep 9.6
postgresql-plperl-9.6 - PL/Perl procedural language for PostgreSQL 9.6
postgresql-plpython-9.6 - PL/Python procedural language for PostgreSQL 9.6
postgresql-plpython3-9.6 - PL/Python 3 procedural language for PostgreSQL
  ↪ 9.6
postgresql-pltcl-9.6 - PL/Tcl procedural language for PostgreSQL 9.6
postgresql-9.6-pllua - Lua procedural language for PostgreSQL 9.6
```


postgresql-9.6-plr - Procedural language interface between PostgreSQL and R
postgresql-9.6-plsh - PL/sh procedural language for PostgreSQL 9.6
postgresql-9.6-plv8 - Procedural language interface between PostgreSQL and
↪ JavaScript

L'installateur Windows contient aussi *PL/PgSQL*. *PL/Perl* et *PL/Python* sont par contre plus compliqués de mise en œuvre. Il faut disposer d'exactly la version qui a été utilisée par le packageur au moment de la compilation de la version windows.

Pour savoir si *PL/Perl* ou *PL/Python* a été compilé, on peut à nouveau demander à `pg_config` :

```
> pg_config --configure  
'--prefix=/usr/local/pgsql-10_icu' '--enable-thread-safety'  
'--with-openssl' '--with-libxml' '--enable-nls' '--with-perl' '--  
enable-debug'  
'ICU_CFLAGS=-I/usr/local/include/unicode/'  
'ICU_LIBS=-L/usr/local/lib -licui18n -licuuc -licudata' '--with-icu'
```

Vérification de la disponibilité

Comment vérifier la présence de la bibliothèque :

```
find $(pg_config --libdir) -name "plpgsql.so"
```

```
find $(pg_config --pkglibdir) -name "plpgsql.so"
```

La bibliothèque `plpgsql.so` contient les fonctions qui permettent l'utilisation du langage *PL/PgSQL*. Elle est installée par défaut avec le moteur PostgreSQL.

Elle est chargée par le moteur à la première utilisation d'une procédure utilisant ce langage.

Toutefois, il est encore plus simple de demander à PostgreSQL d'activer le langage. Il sera bien temps ensuite, si cela échoue, de lancer cette commande de diagnostic.

Ajout d'un langage dans une base de données

- Commande similaire pour tous les langages
- Activer : `CREATE EXTENSION plpython;`
- Désactiver : `DROP EXTENSION plpython;`

Activer le langage dans la base modèle template1 l'activera aussi pour toutes les bases créées par la suite.

PostgreSQL fournit un outil, appelé `createlang`, pour activer un langage :

```
createlang plpgsql la_base_a_activer
```

L'outil se connecte à la base indiquée et exécute la commande `CREATE LANGUAGE` pour le langage précisé en argument. Son pendant Windows se nomme `createlang.exe`. Il existe aussi un outil pour désactiver un langage (`droplang`).

Si vous optez pour exécuter vous-même l'ordre *SQL*, le langage est créé dans la base dans laquelle la commande est lancée.

Pour installer un autre langage, utilisez la même commande tout en remplaçant `plpgsql` par `plperl`, `plperlu`, `plpython`, `pltcl`, `plsh`...

PL/mon_langage est-il déjà installé ?

- Vérification dans `psql` avec la commande `\dx`
- Ou interroger le catalogue système `pg_language`
 - Il contient une ligne par langage installé
 - Un langage peut avoir `lanpltrusted` à `false`

Le langage *PL/PgSQL* apparaît comme une extension :

```
base=# \dx
```

Liste des extensions installées			
Nom	Version	Schéma	Description
-----+-----+-----+-----			
↳ -			
plpgsql	1.0	pg_catalog	_PL/PgSQL_ procedural language

C'est évidemment aussi applicable aux autres langages PL.

Voici un exemple d'interrogation de `pg_language` :

```
SELECT lanname, lanpltrusted
FROM pg_language
WHERE lanname='plpgsql';
```

```
lanname | lanpltrusted
-----+-----
plpgsql | t
(1 ligne)
```

Une fonction *PL/PgSQL*

- Fonction effectuant la somme de deux entiers :

```
CREATE FUNCTION addition(entier1 integer, entier2 integer)
RETURNS integer
LANGUAGE plpgsql
IMMUTABLE
AS '
DECLARE
    resultat integer;
BEGIN
    resultat := entier1 + entier2;
    RETURN resultat;
END';
```

Le langage *PL/PgSQL* n'est pas sensible à la casse, tout comme *SQL* (sauf les noms de colonnes, si vous les mettez entre des guillemets doubles).

L'opérateur de comparaison est =, l'opérateur d'affectation :=

Création et structure

- Ordre *SQL* : **CREATE FUNCTION** ou **CREATE PROCEDURE**
- Le langage est un paramètre de la définition de l'objet

Voici la syntaxe complète :

Syntax :

```
CREATE [ OR REPLACE ] FUNCTION
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ]
          [, ...] ] )
    [ RETURNS rettype
      | RETURNS TABLE ( column_name column_type [, ...] ) ]
    { LANGUAGE lang_name
      | TRANSFORM { FOR TYPE type_name } [, ... ]
      | WINDOW
      | IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF
```

```

| CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
| [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
| PARALLEL { UNSAFE | RESTRICTED | SAFE }
| COST execution_cost
| ROWS result_rows
| SET configuration_parameter { TO value | = value | FROM CURRENT }
| AS 'definition'
| AS 'obj_file', 'link_symbol'
} ...
[ WITH ( attribute [, ...] ) ]

```

Et un exemple pour *PL/PgSQL* :

```

CREATE FUNCTION ma_fonction () RETURNS integer
LANGUAGE plpgsql
(...)

```

Les objets PROCEDURE sont apparus dans la version 11 de PostgreSQL. À la différence des objets FUNCTION, ils ne renvoient rien mais on peut utiliser dans le corps de la fonction des ordres de contrôle de transaction (comme COMMIT et ROLLBACK). La syntaxe complète :

```

CREATE [ OR REPLACE ] PROCEDURE
  name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ]
↪  [, ...] ] )
{ LANGUAGE lang_name
  | TRANSFORM { FOR TYPE type_name } [, ... ]
  | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
  | SET configuration_parameter { TO value | = value | FROM CURRENT }
  | AS 'definition'
  | AS 'obj_file', 'link_symbol'
  | sql_body
} ...

```

Modification d'une fonction

- Dans le cas d'une modification : CREATE OR REPLACE FUNCTION
- Une fonction est définie par son nom et ses arguments
- Si type de retour différent, la fonction doit d'abord être supprimée puis recréée

Une fonction est surchargeable. La seule façon de les différencier est de prendre en compte les arguments (nombre et type). Deux fonctions identiques aux arguments près (on parle de prototype) ne

sont pas identiques, mais bien deux fonctions distinctes.

Suppression d'une fonction

- Ordre SQL : DROP FUNCTION
- Arguments (en entrée) nécessaires à l'identification de la fonction à supprimer :

```
DROP FUNCTION addition(integer, integer);  
DROP FUNCTION public.addition(integer, integer);
```

Les types des arguments en entrée doivent être indiqués. Par contre, leur nom n'a aucune importance. Vous pouvez toutefois les passer quand même mais, comme pour un CREATE FUNCTION, ils sont simplement ignorés.

Arguments d'une fonction

```
[ [ mode_argument ] [ nom_argument ] type_argument  
[ { DEFAULT | = } expr_default ] [, ...] ]
```

- *mode_argument* : en entrée (IN), en sortie (OUT)
- *nom_argument* : nom
- *type_argument* : type
 - parmi tous les types de base et les types utilisateur
- valeur par défaut : clause DEFAULT

Si le mode de l'argument est omis, IN est la valeur implicite.

Il existe d'autres modes pour les arguments : , en entrée/sortie (INOUT) ou à nombre variant (VARIADIC).

L'option VARIADIC permet de définir une fonction avec un nombre d'arguments libres à condition de respecter le type de l'argument (comme printf en C par exemple).

Seul un argument OUT peut suivre un argument VARIADIC : l'argument VARIADIC doit être le dernier de la liste des paramètres en entrée puisque tous les paramètres en entrée suivant seront considérées comme faisant partie du tableau variadic. Seuls les arguments IN et VARIADIC sont utilisables avec une fonction déclarée renvoyant une table (clause RETURNS TABLE). S'il y a

plusieurs paramètres en OUT, un enregistrement composite de tous ces types est renvoyé (c'est donc équivalent sémantiquement à un RETURNS TABLE).

La clause DEFAULT permet de rendre les paramètres optionnels. Après le premier paramètre ayant une valeur par défaut, tous les paramètres qui suivent doivent avoir une valeur par défaut. Pour rendre le paramètre optionnel, il doit être le dernier argument ou alors les paramètres suivants doivent aussi avoir une valeur par défaut.

Retour d'une fonction

RETURNS type_ret

- Il faut aussi indiquer un type de retour
- sauf si un ou plusieurs paramètres sont en mode OUT ou INOUT
- *type_ret* : type de la valeur en retour
- *void* est un type de retour valide

Voici comment s'utilise le retour de type table :

RETURNS TABLE (nom_colonne nom_type [, ...])

On peut aussi indiquer que la fonction ne retourne pas un enregistrement (un scalaire en termes relationnels) mais un jeu d'enregistrements (c'est-à-dire une relation, une table). Il faut utiliser le mot clé SETOF

Language d'une fonction

LANGUAGE <nomlang>

- Le langage de la fonction doit être précisé
- Dans notre cas, nous utiliserons `plpgsql`.
- Mais il est possible de créer des fonctions en `plphp`, `plruby`, voire des langages spécialisés comme `plproxy`.

Politique de sécurité

[EXTERNAL] SECURITY INVOKER | [EXTERNAL] SECURITY DEFINER

- Permet de déterminer l'utilisateur avec lequel sera exécutée la fonction
- Le « *sudo* » de la base de données
 - Potentiellement dangereux

Une fonction SECURITY INVOKER s'exécute avec les droits de l'appelant. C'est le mode par défaut.

Une fonction SECURITY DEFINER s'exécute avec les droits du créateur. Cela permet, au travers d'une fonction, de permettre à un utilisateur d'outrepasser ses droits de façon contrôlée.

Bien sûr, une fonction SECURITY DEFINER doit faire l'objet d'encore plus d'attention qu'une fonction normale. Elle peut facilement constituer un trou béant dans la sécurité de votre base.

Des choses importantes sont à noter pour SECURITY DEFINER :

- Toute fonction, par défaut, est exécutable par public. La première chose à faire est donc de révoquer ce droit.
- Il faut se protéger des variables de session qui pourraient être utilisées pour modifier le comportement de la fonction, en particulier le **search_path**. Il doit donc **impérativement** être positionné en dur dans cette fonction (soit d'emblée, avec un SET dans la fonction, soit en positionnant un SET dans le CREATE FUNCTION).

Le mot clé EXTERNAL est facultatif, et n'est là que pour être en conformité avec la norme SQL : en effet, dans PostgreSQL, on peut modifier le security definer pour toutes les fonctions, qu'elles soient externes ou pas.

Mode de la fonction

IMMUTABLE | STABLE | VOLATILE

- Ce mode précise la « volatilité » de la fonction.

On peut indiquer à PostgreSQL le niveau de volatilité (ou de stabilité) d'une fonction. Ceci permet d'aider PostgreSQL à optimiser les requêtes utilisant ces fonctions, mais aussi d'interdire leur utilisation dans certains contextes.

- Une fonction est IMMUTABLE (immuable) si son exécution ne dépend que de ses paramètres. Elle ne doit donc dépendre ni du contenu de la base (pas de SELECT, ni de modification de donnée de quelque sorte), ni d'**aucun** autre élément qui ne soit pas un de ses paramètres. Par exemple, now() n'est évidemment pas immuable. Une fonction sélectionnant des données d'une table non plus. to_char() n'est pas non plus immuable : son comportement dépend des paramètres de session, par exemple to_char(timestamp with time zone, text) dépend du paramètre de session timezone...
- Une fonction est STABLE si son exécution donne toujours le même résultat sur toute la durée d'un ordre SQL, pour les mêmes paramètres en entrée. Cela signifie que la fonction ne modifie pas les données de la base. to_char() est STABLE.
- Une fonction est VOLATILE dans tous les autres cas. now() est VOLATILE. Une fonction non déclarée comme STABLE ou IMMUTABLE est VOLATILE par défaut.

Quelle importance ?

- Une fonction IMMUTABLE peut être remplacée par son résultat avant même la planification d'une requête l'utilisant. L'exemple le plus simple est une simple opération arithmétique. Si vous exécutez :

```
SELECT * FROM ma_table WHERE mon_champ > abs(-2)
```

et PostgreSQL substitue abs(-2) par 2 et planifie ensuite la requête. Cela fonctionne aussi, bien sûr, avec les opérateurs (comme +), qui ne sont qu'un habillage syntaxique au-dessus d'une fonction.

- Une fonction STABLE peut être remplacée par son résultat pendant l'exécution de la requête. On n'a par contre aucune idée de sa valeur avant de commencer à exécuter la requête parce qu'on ne sait pas encore, à ce moment là, quelles sont les données qui seront visibles en base. Par exemple, avec :

```
SELECT * FROM ma_table WHERE mon_timestamp > now()
```

PostgreSQL sait que now() (le timestamp de démarrage de la transaction) va être constant pendant toute la durée de la transaction. Néanmoins, now() n'est pas IMMUTABLE, il ne va donc pas le remplacer par sa valeur avant d'exécuter la requête. Il n'exécutera par contre now() qu'une seule fois.

- Une fonction VOLATILE doit systématiquement être exécutée à chaque appel.

On comprend donc l'intérêt de se poser la question à l'écriture de chaque fonction.

Une autre importance existe, pour la création d'index sur fonction. Par exemple,

```
CREATE INDEX mon_index ON ma_table ((ma_fonction(ma_colonne)))
```


Ceci n'est possible que si la fonction est IMMUTABLE. En effet, si le résultat de la fonction dépend de l'état de la base, la fonction calculée au moment de la création de la clé d'index ne retournera plus le même résultat quand viendra le moment de l'interroger. PostgreSQL n'acceptera donc que les fonctions IMMUTABLE dans la déclaration des index fonctionnels.

Autres paramètres

- Coût d'appel de la fonction avec les paramètres COST et ROWS
- la fonction peut-elle être exécutée en mode parallèle ? PARALLEL [UNSAFE | RESTRICTED | SAFE]
- Gestion des valeurs *NULL* dans les arguments

Coût d'appel de la fonction

- COST `cout_execution`
 - coût estimé pour l'exécution de la fonction
 - COST est représenté en unité de `cpu_operator_cost` (100 par défaut).
- ROWS `nb_lignes_resultat`
 - nombre estimé de lignes que la fonction renvoie
 - ROWS vaut par défaut 1000 pour les fonctions SETOF. Pour les autres fonctions, la valeur de ce paramètre est ignorée et remplacée par 1.

Ces deux paramètres ne modifient pas le comportement de la fonction. Ils ne servent que pour aider l'optimiseur de requête à estimer le coût d'appel à la fonction, afin de savoir, si plusieurs plans sont possibles, lequel est le moins coûteux par rapport au nombre d'appels de la fonction et au nombre d'enregistrements qu'elle retourne.

Exécution en parallèle PARALLEL UNSAFE indique que la fonction ne peut pas être exécutée dans le mode parallèle. La présence d'une fonction de ce type dans une requête SQL force un plan d'exécution en série. C'est la valeur par défaut.

Une fonction est non parallélisable si elle modifie l'état d'une base ou si elle fait des changements sur la transaction.

PARALLEL RESTRICTED indique que la fonction peut être exécutée en mode parallèle mais l'exécution est restreinte au processus principal d'exécution.

Une fonction peut être déclarée comme restreinte si elle accède aux tables temporaires, à l'état de connexion des clients, aux curseurs, aux requêtes préparées.

`PARALLEL SAFE` indique que la fonction s'exécute correctement dans le mode parallèle sans restriction.

En général, si une fonction est marquée sûre ou restreinte à la parallélisation alors qu'elle ne l'est pas, elle pourrait renvoyer des erreurs ou fournir de mauvaises réponses lorsqu'elle est utilisée dans une requête parallèle.

En cas de doute, les fonctions doivent être marquées comme `UNSAFE`, ce qui correspond à la valeur par défaut.

Gestion des valeurs NULL

`CALLED ON NULL INPUT` | `RETURNS NULL ON NULL INPUT` | `STRICT`

- `CALLED ON NULL INPUT` : fonction appelée même si certains arguments sont `NULL`.
- `RETURNS NULL ON NULL INPUT` ou `STRICT` : la fonction renvoie `NULL` à chaque fois qu'au moins un argument est `NULL`.

Si un des arguments est `NULL`, PostgreSQL n'exécute même pas la fonction et utilise `NULL` comme résultat.

Dans la logique relationnelle, `NULL` signifie «la valeur est inconnue». La plupart du temps, il est logique qu'une fonction ayant un paramètre à une valeur inconnue retourne aussi une valeur inconnue, ce qui fait que cette optimisation est très souvent pertinente.

On gagne à la fois en temps d'exécution, mais aussi en simplicité du code (il n'y a pas à gérer les cas `NULL` pour une fonction dans laquelle `NULL` ne doit jamais être injecté).

Code de la fonction

`AS 'definition'` | `AS 'fichier_obj', 'symbole_lien'`

- Premier cas : chaîne « definition » contenant le code réel de la fonction
- Deuxième cas : `fichier_obj` est le nom de la bibliothèque, `symbole_lien` est le nom de la fonction dans le code source C

`symbole_lien` n'est à utiliser que quand le nom de la fonction diffère du nom de la fonction C qui l'implémente.

Structure de la définition

- DECLARE : déclaration des variables locales
 - BEGIN : début du code de la fonction
 - END : fin du code de la fonction
 - Instructions séparées par des points-virgules
 - Commentaires :
 - commencent par --
 - ou compris entre /* et */
-

Utilisation des guillemets

- L'usage des guillemets problématique.
- le source se place directement entre guillemets
 - besoin de doubler les guillemets du code source
- Encadrement du source par \$\$

Exemple :

Syntaxe utilisant uniquement des guillemets :

```
requete := requete || ' ' AND vin LIKE ''''bordeaux%'''' AND xyz''
```

Simplification grâce aux dollars :

```
requete := requete || $sql$ AND vin LIKE 'bordeaux%' AND xyz$sql$
```

Si vous avez besoin de mettre entre guillemets du texte qui inclut \$\$, vous pouvez utiliser *Q*, et ainsi de suite. Le plus simple étant de définir un marqueur de fin de fonction plus complexe...

Déclaration de variables

- Variables déclarées dans le source, dans la partie DECLARE :

DECLARE

```
nombre integer;  
contenu text;
```

- Les variables peuvent se voir associées une valeur initiale :

```
nombre integer := 5;
```

Type des variables

- Types natifs de PostgreSQL intégralement supportés
- Quelques types spécifiques à *PL/PgSQL*

En dehors des types natifs de PostgreSQL, *PL/PgSQL* y ajoute des types spécifiques pour faciliter l'écriture des fonctions.

Type ROW

- But :
 - utilisation de structures,
 - renvoi de plusieurs valeurs à partir d'une fonction
- Utiliser un type composite :

```
CREATE TYPE ma_structure AS (  
  un_entier integer,  
  une_chaine text,  
  ...);  
CREATE FUNCTION ma_fonction ()  
  RETURNS ma_structure...;
```

Type RECORD

- Identique au type ROW
 - mais type connu que lors de son affectation
- Une variable de type RECORD peut changer de type au cours de l'exécution de la fonction, suivant les affectations réalisées

RECORD est beaucoup utilisé pour manipuler des curseurs : cela évite de devoir se préoccuper de déclarer un type correspondant exactement aux colonnes de la requête associée à chaque curseur.

Ici la variable `ligne` change de type entre les 2 traitements :

```
CREATE FUNCTION ma_fonction () RETURNS integer
AS $$
DECLARE
    ligne RECORD;
    (...)
BEGIN
    SELECT INTO ligne * FROM ma_premiere_table;
    -- traitement de la ligne
    FOR ligne IN SELECT * FROM ma_deuxieme_table LOOP
    -- traitement de cette nouvelle ligne
    (...)

```

Type ROWTYPE

- Possible d'utiliser le type composite défini par la ligne d'une table :

```
CREATE FUNCTION ma_fonction () RETURNS integer
AS $$
DECLARE
    ligne ma_table%ROWTYPE;
    (...)

```

L'utilisation de « %ROWTYPE » permet de définir une variable qui contient la structure d'un enregistrement de la table spécifiée. « %ROWTYPE » n'est pas obligatoire, il est néanmoins préférable d'utiliser cette forme, bien plus portable. En effet, dans PostgreSQL, toute création de table crée un type associé de même nom, le nom de la table seul est donc suffisant.

Affectation d'une valeur à une variable

- À l'ordre `SELECT INTO` :

```
SELECT INTO un_entier 5;
```

- Préférez l'opérateur `:=` :

```
un_entier := 5;
```

```
un_entier := une_colonne FROM ma_table WHERE id = 5;
```

Affectation via une requête

```
SELECT * INTO ma_variable_ligne FROM ma_table...;
```

- Affectation dans une variable `RECORD` ou `ROW`
- Si plusieurs enregistrements renvoyés, seul le premier est récupéré
- Pour récupérer plus d'un enregistrement : écrire une boucle

Dans le cas du type `ROW`, la définition de la ligne doit correspondre parfaitement à la définition de la ligne renvoyée. Utiliser un type `RECORD` permet d'éviter ce type de problème. La variable obtient directement le type `ROW` de la ligne renvoyée.

Pour contrôler qu'un seul enregistrement est renvoyé, remplacer `INTO` par `INTO STRICT`.

L'ordre SQL est statique : on ne peut pas faire varier les colonnes retournées, la clause `WHERE`, les tables...

Exécution d'une requête via *EXECUTE*

```
EXECUTE '<chaîne>' [INTO [STRICT] cible];
```

- Exécute la requête comprise dans la variable chaîne
- La variable chaîne peut être construite à partir d'autres variables
- `cible` contient le résultat de l'exécution de la requête

Ne fonctionne que si la requête ne retourne qu'une unique ligne. Si plusieurs lignes sont retournées, utiliser une boucle.

Mot clé USING Le mot clé USING peut-être utiliser. Il permet de créer une requête dynamique avec des variables de substitution. Il est de plus beaucoup plus lisible que des quote_nullable.

```
EXECUTE command-string [ INTO [STRICT] target ] [ USING expression [, ... ] ];
```

```
EXECUTE 'SELECT count(*) FROM mytable  
WHERE inserted_by = $1 AND inserted <= $2'  
INTO c  
USING checked_user, checked_date;
```

- Le nombre de paramètres de la requête doit être fixe, ainsi que leur type
- Ne concerne pas les identifiants !

Mot clé STRICT

- Sans STRICT, cible contient la première ligne d'un résultat multi-lignes ou NULL s'il n'y a pas de résultat.
- Avec STRICT, une exception est levée si le résultat ne contient aucune ligne (NO_DATA_FOUND) ou en contient plusieurs (TOO_MANY_ROWS).

Nous verrons comment traiter les exceptions plus loin.

Exécution d'une requête via *PERFORM*

PERFORM <query>

- Permet l'exécution
 - d'un INSERT, UPDATE, DELETE
 - ou même SELECT, si le résultat importe peu
- Permet aussi d'appeler une autre fonction sans en récupérer de résultat

Pour appeler une fonction, il suffit d'utiliser PERFORM de la manière suivante :

```
PERFORM mafonction(argument1);
```

Ne fonctionne que si la clause RETURNING n'est pas utilisée.

Protection contre l'injection

- `quote_ident` : met entre guillemets un identifiant d'un objet PostgreSQL
- `quote_literal` : met entre guillemets une valeur (chaîne de caractères)
- Pour concaténer tous les morceaux de la requête :
 - Utiliser l'opérateur de concaténation `||`
 - Ou utiliser `format(...)`, équivalent de `sprintf`

Une autre fonction, `quote_nullable`, met entre guillemets une valeur (chaîne de caractères), sauf `NULL` qui sera alors renvoyé sans les guillemets

La fonction `format` est l'équivalent de la fonction `sprintf` : elle formate une chaîne en fonction d'un patron et de valeurs à appliquer à ses paramètres et la retourne. Les type de paramètre reconnus par `format` sont :

- `%I` : est remplacé par un identifiant d'objet. C'est l'équivalent de la fonction `quote_ident`. L'objet en question est entouré en double-guillemet si nécessaire ;
- `%L` : est remplacé par une valeur littérale. C'est l'équivalent de la fonction `quote_literal`. Des simple-guillemet sont ajoutés à la valeur et celle-ci est correctement échappée si nécessaire ;
- `%s` : est remplacé par la valeur donnée sans autre forme de transformation ;
- `%%` : est remplacé par un simple `%`.

Voici un exemple d'utilisation de cette fonction, utilisant des paramètre positionnels :

```
select format(  
    'SELECT %I FROM %I WHERE %1$I=%3$L',  
    'MaColonne',  
    'ma_table',  
    $$l'été$$  
);  
  
format  
-----  
SELECT "MaColonne" FROM ma_table WHERE "MaColonne"='l'été'
```

Diagnostics d'une exécution

- Affectation de la variable `FOUND` si une ligne est affectée par l'instruction

- Pour obtenir le nombre de lignes affectées : `GET DIAGNOSTICS variable = ROW_COUNT;`

On peut déterminer qu'aucune ligne n'a été trouvé par la requête en utilisant la variable FOUND :

```
PERFORM * FROM ma_table WHERE une_colonne>0;
IF NOT FOUND THEN
...
END IF;
```

Il est à noter que ROW_COUNT s'applique à l'ordre SQL précédent, quel qu'il soit :

- PERFORM ;
- EXECUTE ;
- Ou même à un ordre statique directement dans le code PL/PgSQL.

Fonction renvoyant un ensemble

- Doit renvoyer un ensemble d'un type SETOF
- Chaque ligne sera récupérée par l'instruction RETURN NEXT

Exemple d'une fonction SETOF

```
CREATE FUNCTION liste_entier (limite integer)
RETURNS SETOF integer
AS $$
BEGIN
    FOR i IN 1..limite LOOP
        RETURN NEXT i;
    END LOOP;
END
$$ LANGUAGE plpgsql;
```

- Le retour de l'exécution de cette requête :

```
ma_base=# SELECT * FROM liste_entier(3);
```

```
liste_entier
```

```
-----
```

```
1
```

```
2
```

```
3
```

```
(3 lignes)
```

Structures de contrôles

- Pourquoi du PL?
 - Le but du PL est de pouvoir effectuer des traitements procéduraux.
 - Nous allons donc maintenant aborder les structures de contrôle :
 - Tests
 - Boucles
-

Tests IF/THEN/ELSE/END IF

```
IF condition THEN
  instructions
[ELSEIF condition THEN
  instructions]
[ELSEIF condition THEN
  instructions]
[ELSE
  instructions]
END IF
```

Ce dernier est l'équivalent d'un CASE en C pour une vérification de plusieurs alternatives.

Tests IF/THEN/ELSE/END IF - exemple

```
IF nombre = 0 THEN
    resultat := 'zero';
ELSEIF nombre > 0 THEN
    resultat := 'positif';
ELSEIF nombre < 0 THEN
    resultat := 'négatif';
ELSE
    resultat := 'indéterminé';
END IF;
```

Tests CASE

Deux possibilités :

```
CASE variable
WHEN expression THEN instructions
ELSE instructions
END CASE

CASE
WHEN expression-booléene THEN instructions
ELSE instructions
END CASE
```

Quelques exemples :

```
CASE x
    WHEN 1, 2 THEN
        msg := 'un ou deux';
    ELSE
        msg := 'autre valeur que un ou deux';
END CASE;

CASE
    WHEN x BETWEEN 0 AND 10 THEN
        msg := 'la valeur est entre 0 et 10';
    WHEN x BETWEEN 11 AND 20 THEN
        msg := 'la valeur est entre 11 et 20';
END CASE;
```

Boucle LOOP/EXIT/CONTINUE

- Créer une boucle (label possible)
 - LOOP / END LOOP :
- Sortir de la boucle
 - EXIT [label] [WHEN expression_booléenne]
- Commencer une nouvelle itération de la boucle
 - CONTINUE [label] [WHEN expression_booléenne]

Label de blocs

Dans les boucles :

- Labels de bloc possibles
- Plusieurs blocs d'exception possibles dans une fonction
- Permet de préfixer des variables avec le label du bloc
- De donner un label à une boucle itérative et de préciser de quelle boucle on veut sortir, quand plusieurs d'entre elles sont imbriquées

Indiquer le nom d'un label ainsi :

```
<<mon_label>>  
-- le code (blocs DECLARE, BEGIN-END, et EXCEPTION)
```

ou bien (pour une boucle)

```
[ <<mon_label>> ]  
LOOP  
    ordres ...  
END LOOP [ mon_label ];
```

Il est aussi bien sûr possible d'utiliser des labels pour des boucles FOR, WHILE, FOREACH.

On sort d'un bloc ou d'une boucle avec la commande EXIT, on peut aussi utiliser CONTINUE pour passer à l'exécution suivante d'une boucle sans terminer l'itération courante.

Par exemple :

```
EXIT [mon_label] WHEN compteur > 1;
```

Boucle LOOP/EXIT/CONTINUE - exemple

LOOP

```
resultat := resultat + 1;  
EXIT WHEN resultat > 100;  
CONTINUE WHEN resultat < 50;  
resultat := resultat + 1;  
END LOOP;
```

Cette boucle incrémente le resultat de 1 à chaque itération tant que la valeur de resultat est inférieur à 50. Ensuite, resultat est incrémenté de 1 deux fois. Arrivé à 100, la procédure sort de la boucle.

Boucle WHILE

```
WHILE condition LOOP instructions END LOOP;
```

- Boucle jusqu'à ce que la condition soit fausse
 - Label possible
-

Boucle FOR

```
FOR variable in [REVERSE] entier1..entier2 [BY incrément]  
LOOP  
instructions  
END LOOP;
```

- variable va obtenir les différentes valeurs entre entier1 et entier2
 - Label possible.
 - L'option BY permet d'augmenter l'incrément: `sql FOR variable in 1..10 BY 5 ... s`
 - L'option REVERSE permet de faire défiler les valeurs en ordre inverse: `sql FOR variable in REVERSE 10..1 ...`
-

Boucle FOR... IN... LOOP

```
FOR ligne IN SELECT * FROM ma_table LOOP
    instructions
END LOOP;
```

- Permet de boucler dans les lignes résultats d'une requête
- Label possible
- Ligne de type RECORD, ROW ou liste de variables séparées par des virgules
- Utilise un curseur en interne

Exemple :

```
FOR a, b, c, d IN SELECT col_a, col_b, col_c, col_d FROM ma_table
LOOP
    -- instructions
END LOOP;
```

Boucle FOREACH

```
FOREACH variable [SLICE n] IN ARRAY expression LOOP
    instructions
END LOOP
```

- Permet de boucler sur les éléments d'un tableau
- variable va obtenir les différentes valeurs du tableau retourné par expression
- SLICE permet de jouer sur le nombre de dimensions du tableau à passer à la variable

Voici deux exemples permettant d'illustrer l'utilité de SLICE :

- sans SLICE :

```
do $$
declare a int[] := ARRAY[[1,2],[3,4],[5,6]];
        b int;
begin
    foreach b in array a loop
        raise info 'var: %', b;
    end loop;
end $$;
INFO:  var: 1
```

```
INFO: var: 2
INFO: var: 3
INFO: var: 4
INFO: var: 5
INFO: var: 6
```

- Avec SLICE :

```
do $$
declare a int[] := ARRAY[[1,2],[3,4],[5,6]];
        b int[];
begin
    foreach b slice 1 in array a loop
        raise info 'var: %', b;
    end loop;
end $$;
INFO: var: {1,2}
INFO: var: {3,4}
INFO: var: {5,6}
```

Retour d'une fonction

- RETURN [expression]
 - Renvoie cette expression à la requête appelante
 - expression optionnelle si argument(s) déclarés OUT
 - RETURN lui-même optionnel si argument(s) déclarés OUT
-

RETURN NEXT

- Fonction SETOF, aussi appelé fonction SRF (Set Returning Function)
- RETURN NEXT renvoie une ligne du SETOF
- Cette fonction s'appelle de cette façon :

```
SELECT * FROM ma_fonction();
```

RETURN NEXT fonctionne avec des types scalaires (normaux) et des types composites.

Tout est conservé en mémoire jusqu'à la fin de la fonction. Donc, si beaucoup de données sont renvoyées, cela pourrait occasionner quelques lenteurs.

Par ailleurs, il est possible d'appeler une SRF par

```
SELECT ma_fonction();
```

Dans ce cas, on récupère un résultat d'une seule colonne, de type composite.

RETURN QUERY

- Fonctionne comme RETURN NEXT

```
RETURN QUERY la_requete
```

```
RETURN QUERY EXECUTE chaine_requete
```

Par ce mécanisme, on peut très simplement produire une fonction retournant le résultat d'une requête complexe fabriquée à partir de quelques paramètres.

Exemple de Fonction PL/PgSQL

- Permet d'insérer une facture associée à un client
- Si le client n'existe pas, une entrée est créée
- L'accès aux données est simple et naturel
- Les types de données SQL sont natifs
- La capacité de traitement est limitée par le langage
- **Attention** au nommage des variables et paramètres

Pour éviter les conflits avec les objets de la base, il est conseillé de préfixer les variables.

CREATE OR REPLACE FUNCTION

```
public.demo_insert_plpgsql(p_nom_client text, p_titre_facture text)
  RETURNS integer
  LANGUAGE plpgsql
  STRICT
AS $function$
```



```
DECLARE
  v_id_facture int;
  v_id_client int;
BEGIN
  -- Le client existe-t-il ?
  SELECT id_client
  INTO v_id_client
  FROM mes_clients
  WHERE nom_client = p_nom_client;
  -- Sinon on le crée :
  IF (NOT FOUND) THEN
    INSERT INTO mes_clients (nom_client)
    VALUES (p_nom_client)
    RETURNING id_client INTO v_id_client;
  END IF;
  -- Dans les deux cas, l'id client est dans v_id_client

  -- Insérons maintenant la facture
  INSERT INTO mes_factures (titre_facture, id_client)
  VALUES (p_titre_facture, v_id_client)
  RETURNING id_facture INTO v_id_facture;

  return v_id_facture;
END;
$function$ ;
```

Gestion des erreurs

! Sans exceptions !

- Toute erreur provoque un arrêt de la fonction
 - Toute modification suite à une instruction SQL (INSERT, UPDATE, DELETE) est annulée
 - gestion personnalisée des erreurs possible grâce au bloc EXCEPTION
-

Gestion des erreurs : bloc EXCEPTION

La fonction comporte un bloc supplémentaire :

DECLARE

-- déclaration des variables locales

BEGIN

-- instructions de la fonction

EXCEPTION**WHEN** condition **THEN**

-- instructions traitant cette erreur

WHEN condition **THEN**

-- autres instructions traitant cette autre erreur
-- etc.

END**Flot dans une fonction**

- L'exécution de la fonction commence après le BEGIN
- Si aucune erreur ne survient, le bloc EXCEPTION est ignoré
- Si une erreur se produit
 - tout ce qui a été modifié dans la base dans le bloc est annulé
 - les variables gardent par contre leur état
 - l'exécution passe directement dans le bloc de gestion de l'exception

Flot dans une exception

- Recherche d'une condition satisfaisante
- Si cette condition est trouvée
 - exécution des instructions correspondantes
- Si aucune condition n'est compatible
 - sortie du bloc BEGIN/END comme si le bloc d'exception n'existait pas
 - passage de l'exception au bloc BEGIN/END contenant (après annulation de ce que ce bloc a modifié en base)
- Dans un bloc d'exception, les instructions INSERT, UPDATE, DELETE de la fonction ont été annulées
- Dans un bloc d'exception, les variables locales de la fonction ont gardé leur ancienne valeur

Codes d'erreurs

- SQLSTATE : code d'erreur
- SQLERRM : message d'erreur

- par exemple :
 - Data Exception : division par zéro, overflow, argument invalide pour certaines fonctions, etc.
 - Integrity Constraint Violation : unicité, CHECK, clé étrangère, etc.
 - Syntax Error
 - PL/pgsql Error : RAISE EXCEPTION, pas de données, trop de lignes, etc.
- Les erreurs sont contenues dans des classes d'erreurs plus génériques, qui peuvent aussi être utilisées

Toutes les erreurs sont référencées dans la documentation¹

Attention, des codes d'erreurs nouveaux apparaissent à chaque version.

La classe `data_exception` contient de nombreuses erreurs, comme `datetime_field_overflow`, `invalid_escape_character`, `invalid_binary_representation`... On peut donc, dans la déclaration de l'exception, intercepter toutes les erreurs de type `data_exception` d'un coup, ou une par une.

L'instruction `GET STACKED DIAGNOSTICS` permet d'avoir une vision plus précise de l'erreur récupéré par le bloc de traitement des exceptions. La liste de toutes les informations que l'on peut collecter est disponible dans la documentation².

La démonstration ci-dessous montre comment elle peut être utilisée.

```
# CREATE TABLE t5(c1 integer PRIMARY KEY);
CREATE TABLE
# INSERT INTO t5 VALUES (1);
INSERT 0 1
# CREATE OR REPLACE FUNCTION test(INT4) RETURNS void AS $$
DECLARE
    v_state TEXT;
    v_msg TEXT;
    v_detail TEXT;
    v_hint TEXT;
    v_context TEXT;
BEGIN
    BEGIN
        INSERT INTO t5 (c1) VALUES ($1);
    EXCEPTION WHEN others THEN
        GET STACKED DIAGNOSTICS
            v_state = RETURNED_SQLSTATE,
```

¹<http://docs.postgresql.fr/current/errcodes-appendix.html>

²<http://docs.postgresql.fr/current/plpgsql-control-structures.html#plpgsql-exception-diagnostics-values>

```
        v_msg      = MESSAGE_TEXT,
        v_detail   = PG_EXCEPTION_DETAIL,
        v_hint     = PG_EXCEPTION_HINT,
        v_context  = PG_EXCEPTION_CONTEXT;
    raise notice E'Et une exception :
        state : %
        message: %
        detail : %
        hint   : %
        context: %', v_state, v_msg, v_detail, v_hint, v_context;
END;
RETURN;
END;
$$ LANGUAGE plpgsql;
# SELECT test(2);
test
-----

(1 row)

# SELECT test(2);
NOTICE:  Et une exception :
        state : 23505
        message: duplicate key value violates unique constraint "t5_pkey"
        detail : Key (c1)=(2) already exists.
        hint   :
        context: SQL statement "INSERT INTO t5 (c1) VALUES ($1)"
PL/pgSQL function test(integer) line 10 at SQL statement
test
-----

(1 row)
```

Pour aller plus loin

- Documentation officielle
 - « Chapitre 43. PL/PgSQL - Langage de procédures SQL »

La documentation officielle sur le langage PL/PgSQL peut être consultée en français à cette

adresse³.

Les triggers

- Aperçu du comportement des déclencheurs
 - Les variables disponibles
 - Traitement du retour
 - Options de CREATE TRIGGER
-

Procédures trigger : introduction

- Action déclenchée par un ordre de *DML* :
 - INSERT (incluant COPY), UPDATE, DELETE ou TRUNCATE
- Mode **par ligne** ou **par instruction**
- Exécution d'une procédure stockée
 - codée dans un des langages de procédure disponible

Un déclencheur est une spécification précisant que la base de données doit exécuter une fonction particulière quand un certain type d'opération est traité. Les fonctions déclencheurs peuvent être définies pour s'exécuter avant ou après une commande INSERT, UPDATE, DELETE ou TRUNCATE.

Un déclencheur peut être attaché à une table ou à une vue.

La fonction déclencheur doit être définie avant que le déclencheur lui-même puisse être créé. La fonction déclencheur doit être déclarée comme une fonction ne prenant aucun argument et retournant un type trigger.

Une fois qu'une fonction déclencheur est créée, le déclencheur est créé avec CREATE TRIGGER. La même fonction déclencheur est utilisable par plusieurs déclencheurs.

Un trigger TRUNCATE ne peut utiliser que le mode par instruction, contrairement aux autres triggers pour lesquels vous avez le choix entre « par ligne » et « par instruction ».

Enfin, l'instruction COPY est traitée comme s'il s'agissait d'une commande INSERT.

³<http://docs.postgresql.fr/14/plpgsql.html>

Les variables OLD et NEW (1/2)

- OLD :
 - type de données RECORD correspondant à la ligne avant modification
 - valable pour un DELETE et un UPDATE
 - NEW :
 - type de données RECORD correspondant à la ligne après modification
 - valable pour un INSERT et un UPDATE
-

Les variables OLD et NEW (2/2)

- Ces deux variables sont valables uniquement pour les triggers en mode ligne
 - pour les triggers en mode instruction : tables de transition
 - Accès aux champs par la notation pointée
 - NEW.champ1 pour accéder à la nouvelle valeur de champ1
-

Les variables d'opération

- TG_NAME : nom du trigger ayant déclenché l'appel de la fonction
 - TG_WHEN : type du trigger (BEFORE, AFTER ou INSTEAD OF)
 - TG_LEVEL : mode du trigger (ROW ou STATEMENT)
 - TG_OP : opération ayant déclenché le trigger (INSERT, UPDATE, DELETE, TRUNCATE)
-

Les variables de relations

- TG_TABLE_SCHEMA : nom du schéma contenant la table...
- TG_TABLE_NAME : nom de la table...
- TG_RELID : OID de la table...

... ayant déclenché le trigger

Vous pourriez aussi rencontrer dans du code `TG_RELNAME`. C'est aussi le nom de la table qui a déclenché le trigger. Attention, cette variable est obsolète, il est préférable d'utiliser maintenant `TG_TABLE_NAME`

Les variables d'arguments

- `TG_NARGS` : nombre d'arguments donnés à la fonction trigger
- `TG_ARGV` : les arguments donnés à la fonction trigger (le tableau commence à 0)

La fonction trigger est déclarée sans arguments mais il est possible de lui en passer dans la déclaration du trigger. Dans ce cas, il faut utiliser les deux variables ci-dessus pour y accéder. Attention, tous les arguments sont convertis en texte. Il faut donc se cantonner à des informations simples, sous peine de compliquer le code.

```
CREATE OR REPLACE FUNCTION verifier_somme()  
RETURNS trigger AS $$  
DECLARE  
    fact_limit integer;  
    arg_color varchar;  
BEGIN  
    fact_limit := TG_ARGV[0];  
  
    IF NEW.somme > fact_limit THEN  
        RAISE NOTICE 'La facture % necessite une verification. '  
            'La somme % depasse la limite autorisee de %.',  
            NEW.idfact, NEW.somme, fact_limit;  
    END IF;  
  
    NEW.datecreate := current_timestamp;  
  
    return NEW;  
END;  
$$ LANGUAGE plpgsql;  
  
CREATE TRIGGER trig_verifier_debit  
    BEFORE INSERT OR UPDATE ON test  
    FOR EACH ROW  
    EXECUTE PROCEDURE verifier_somme(400);
```

```
CREATE TRIGGER trig_verifier_credit
  BEFORE INSERT OR UPDATE ON test
  FOR EACH ROW
  EXECUTE PROCEDURE verifier_somme(800);
```

Traitement du retour

Type de retour spécial : le type trigger

```
CREATE OR REPLACE FUNCTION mon_trigger()
  RETURNS trigger AS $$
  (...)
  $$ LANGUAGE plpgsql;
```

Une fonction de trigger retourne le type spécial `trigger`, pour cette raison ces fonctions ne peuvent être utilisées que dans le contexte d'un ou plusieurs triggers.

Retour des type ROW - BEFORE

- Si retour NULL, annulation de l'opération, sans déclencher d'erreur
- Sinon, poursuite de l'opération avec cette valeur de ligne

Il est possible d'annuler l'action d'un trigger de type ligne avant l'opération en retournant NULL. Ceci annule purement et simplement le trigger sans déclencher d'erreur.

Retour des type ROW - AFTER et STATEMENT

- Valeur de retour ignorée
- Annulation possible si erreur à l'exécution de la fonction

Pour les triggers de type ligne intervenant après l'opération, comme pour les triggers à l'instruction, une valeur de retour est inutile. Elle est ignorée.

Il est possible d'annuler l'action d'un trigger de type ligne intervenant après l'opération ou d'un trigger à l'instruction, en remontant une erreur à l'exécution de la fonction.

Options de CREATE TRIGGER

- Déclenchement conditionnel :

```
CREATE TRIGGER name WHEN ( condition )
```

- Déclenchement limité à une colonne d'une table :

```
CREATE TRIGGER name BEFORE UPDATE OF colx ON my_table
```

- Action de *DML* sur une vue :

```
CREATE TRIGGER view_insert INSTEAD OF INSERT ON my_view`
```

On peut ne déclencher un trigger que si une condition est vérifiée. Cela simplifie souvent le code du trigger, et gagne en performances : plus du tout besoin pour le moteur d'aller exécuter la fonction.

On peut ne déclencher un trigger que si une colonne spécifique a été modifiée. Il ne s'agit donc que de triggers sur update. Encore un moyen de simplifier le code et de gagner en performances en évitant les déclenchements inutiles.

On peut créer un trigger sur une vue. C'est un trigger `INSTEAD OF`, qui permet de programmer de façon efficace les `INSERT/UPDATE/DELETE/TRUNCATE` sur les vues. Auparavant, il fallait passer par le système de règles (`RULES`), complexe et sujet à erreurs.

On peut créer un trigger en le déclarant comme étant un trigger de contrainte. Il peut alors être 'deferred', 'deferred', comme tout autre contrainte, c'est à dire n'être exécutée qu'au moment de la validation de la transaction, ce qui permet de ne vérifier les contraintes implémentées par le trigger qu'au moment de la validation finale :

```
CREATE CONSTRAINT TRIGGER
```

Procédures trigger : exemple - 1

Horodater une opération sur une ligne :

```
CREATE TABLE ma_table (  
  id serial,  
  -- un certain nombre de champs informatifs  
  date_ajout timestamp,  
  date_modif timestamp);
```

Procédures trigger : exemple - 2

```
CREATE OR REPLACE FUNCTION horodatage() RETURNS trigger
AS $$
BEGIN
    IF TG_OP = 'INSERT' THEN
        NEW.date_ajout := now();
    ELSEIF TG_OP = 'UPDATE' THEN
        NEW.date_modif := now();
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Tables de transition

- REFERENCING OLD TABLE
- REFERENCING NEW TABLE
- Par exemple :

```
CREATE TRIGGER tr1
AFTER DELETE ON t1
REFERENCING OLD TABLE AS oldtable
FOR EACH STATEMENT
EXECUTE PROCEDURE log_delete();
```

Dans le cas d'un trigger en mode instruction, il n'est pas possible d'utiliser les variables OLD et NEW car elles ciblent une seule ligne. Pour cela, le standard SQL parle de tables de transition. Ces dernières ont été implémentées dans PostgreSQL pour la version 10. Voici un exemple de leur utilisation.

Nous allons créer une table t1 qui aura le trigger et une table poubelle qui a pour but de récupérer les enregistrements supprimés de la table t1.

```
CREATE TABLE t1 (c1 integer, c2 text);
CREATE TABLE poubelle (id integer GENERATED ALWAYS AS IDENTITY,
    dlog timestamp DEFAULT now(),
    t1_c1 integer, t1_c2 text);
```

Maintenant, il faut créer le code de la procédure stockée :

```
CREATE OR REPLACE FUNCTION log_delete() RETURNS trigger LANGUAGE plpgsql AS $$
BEGIN
    INSERT INTO poubelle (t1_c1, t1_c2) SELECT c1, c2 FROM oldtable;
    RETURN null;
END
$$;
```

Et ajouter le trigger sur la table t1 :

```
CREATE TRIGGER tr1
AFTER DELETE ON t1
REFERENCING OLD TABLE AS oldtable
FOR EACH STATEMENT
EXECUTE PROCEDURE log_delete();
```

Maintenant, insérons un million de ligne dans t1 et supprimons-les :

```
$ INSERT INTO t1 SELECT i, 'Ligne ' || i FROM generate_series(1, 1000000) i;
INSERT 0 1000000
$ \timing on
$ DELETE FROM t1;
DELETE 1000000
Time: 6753.294 ms (00:06.753)
```

La suppression avec le trigger prend 6.7 secondes. Il est possible de connaître le temps à supprimer les lignes et le temps à exécuter le trigger en utilisant l'ordre EXPLAIN ANALYZE :

```
$ TRUNCATE poubelle;
TRUNCATE TABLE
Time: 545.840 ms
$ INSERT INTO t1 SELECT i, 'Ligne ' || i FROM generate_series(1, 1000000) i;
INSERT 0 1000000
Time: 4259.430 ms (00:04.259)
$ EXPLAIN (ANALYZE) DELETE FROM t1;
               QUERY PLAN
```

```
-----  
---  
Delete on t1 (cost=0.00..17880.90 rows=1160690 width=6)  
          (actual time=2552.210..2552.210 rows=0 loops=1)  
-> Seq Scan on t1 (cost=0.00..17880.90 rows=1160690 width=6)  
      (actual time=0.071..183.026 rows=1000000 loops=1)  
Planning time: 0.094 ms  
Trigger tr1: time=4424.651 calls=1  
Execution time: 6982.290 ms  
(5 rows)
```

Donc la suppression des lignes met 2,5 secondes alors que l'exécution du trigger met 4,4 secondes.

Le gros intérêt des tables de transition est le gain en performance que cela apporte. Sur la suppression d'un million de lignes, il est deux fois plus rapide de passer par un trigger en mode instruction que par un trigger en mode ligne.

Déclencheurs sur évènement

- Trigger global à une base de données
 - Capable de capturer tous les évènements *DDL*
- Type de retour spécial de la fonction : `event_trigger`
- ne peut pas être écrit en *SQL*

La syntaxe de l'ordre de création est le suivant :

```
CREATE EVENT TRIGGER nom  
ON evenement  
[ WHEN variable_filtre IN (valeur_filtre [, ... ]) [ AND ... ] ]  
EXECUTE PROCEDURE nom_fonction()
```

Les évènements possibles sont :

- `ddl_command_start` : se déclenche juste avant l'exécution d'une commande `CREATE`, `ALTER`, `DROP`, `SECURITY LABEL`, `COMMENT`, `GRANT` ou `REVOKE`.

- `ddl_command_end` : se déclenche juste après l'exécution de ces même ensembles de commandes.
 - `sql_drop` : se déclenche juste avant le trigger sur évènement `ddl_command_end` pour toute opération qui supprime des objets de la base.
 - `table_rewrite` : se déclenche juste avant qu'une table soit modifiée par certaines actions des commandes `ALTER TABLE` et `ALTER TYPE`.
-

Pour aller plus loin

- Documentation officielle
 - « Chapitre 39. Déclencheurs (*triggers*) »
 - « Chapitre 40. Déclencheurs (*triggers*) sur évènement »

La documentation officielle sur le langage *PL/PgSQL* peut être consultée en français à cette adresse⁴.

Travaux Dirigés 5

- *PL/PgSQL* et triggers
-

Question de cours

- Quand est-il préférable d'utiliser un langage PL plutôt que SQL ?
 - Pourquoi utiliser PL/pgSQL plutôt que de traiter les données côté application ?
 - Sur quels types d'opération peut-on définir un trigger ?
-

⁴<https://docs.postgresql.fr/14/triggers.html>

Enoncés

Fonctions PL/PgSQL Syntaxe de création d'une fonction :

```
CREATE [ OR REPLACE ] FUNCTION
    nom ( [ [ mode_argument ] [ nom_agrégat ] type_argument
          [ { DEFAULT | = } expression_par_défaut ]
          [, ...] ] )
    [ RETURNS type_en_retour
      | RETURNS TABLE ( nom_colonne type_colonne [, ...] ) ]
{ LANGUAGE nom_langage
  | TRANSFORM { FOR TYPE nom_type } [, ... ]
  | WINDOW
  | IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF
  | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
  | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
  | PARALLEL { UNSAFE | RESTRICTED | SAFE }
  | COST coût_exécution
  | ROWS lignes_de_résultat
  | SET paramètre_configuration { TO valeur | = valeur | FROM CURRENT }
  | AS 'définition'
  | AS 'fichier_objet', 'symbole_link'
} ...
[ WITH ( attribut [, ...] ) ]
```

Exercice 1

- Écrire une fonction de division appelée `division`. Elle acceptera en entrée deux arguments de type entier et renverra un nombre flottant.
- Ré-écrire la fonction de division pour tracer le problème de division par zéro.

Conseil : dans ce genre de calcul impossible, il est possible d'utiliser avec PostgreSQL la constante NaN (*Not A Number*).

- Modifier la fonction pour tracer le problème de division par zéro via les exceptions.

Exercice 2

Écrire une fonction de multiplication dont les arguments sont des chiffres en toute lettre.

Par exemple, appeler la fonction avec comme arguments les chaînes « deux » et « trois » doit renvoyer 6.

Exercice 3

- En utilisant la base de données *cave*, créer une fonction `nb_bouteilles` qui renvoie le nombre de bouteilles en stock suivant une année et un type de vin (donc passés en paramètre).
- Utiliser la fonction `generate_series()` pour extraire le nombre de bouteilles en stock sur plusieurs années, de 1990 à 1999.

Triggers Syntaxe de création d'un trigger :

```
CREATE [ CONSTRAINT ] TRIGGER nom { BEFORE | AFTER | INSTEAD OF }
    { événement [ OR ... ] }
    ON nom_table
    [ FROM nom_table_référencée ]
    [ NOT DEFERRABLE
      | [ DEFERRABLE ] [ INITIALLY IMMEDIATE | INITIALLY DEFERRED ] ]
    [ REFERENCE { { OLD | NEW } TABLE [ AS ] nom_relation_transition } [
    ↪ ... ] ]
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( condition ) ]
    EXECUTE PROCEDURE nom_fonction ( arguments )
```

Exercice 4

Tracer dans une table toutes les modifications du champ nombre dans stock. On veut garder le trace de l'heure, de l'opération, du type de vin, du contenant, de l'année, de l'ancienne valeur et de la nouvelle valeur.

Afficher un message NOTICE quand nombre devient inférieur à 5, et WARNING quand il vaut 0.

Travaux Pratiques 5

- PL/PgSQL et triggers

Rappel

Durant ces travaux pratiques, nous allons utiliser la machine virtuelle du TP 1 pour héberger notre serveur de base de données PostgreSQL.

Effectuez les manipulations nécessaires pour réaliser les actions listées dans la section *Énoncés*.

Vous pouvez vous aider du cours, de l'annexe de ce TP, des derniers TP, ainsi que de l'aide en ligne ou des pages de manuels (man).

Énoncés

Fonctions Hello World

Fonctions traitant des chaînes de caractères :

- Écrire une fonction hello qui renvoie la chaîne de caractère « Hello World! » en SQL.
- Écrire une fonction hello pl qui renvoie la chaîne de caractère « Hello World! » en PL/pgSQL.
- Comparer les coûts des deux plans d'exécutions de ces requêtes. Expliquer les coûts.

Arithmétique

- Écrivez une fonction getbordeaux qui retournent les libelles tous les vins de Bordeaux.
- Écrivez une fonction vin region par type qui prend en paramètre un type de vin (rouge, rose, blanc) et retourne les noms et régions de production des vins correspondant.
- Écrivez une fonction nb_par_recoltant par an qui prend en paramètre une année (int) et retourne le nombre de vins par récoltant de l'année en paramètre
- Écrivez la fonction nb_bouteilles_1 qui prend en paramètre d'entrée une année et un type de vin et retourne le nombre de bouteille de ce type produite.
- Écrivez la fonction nb_bouteilles_2 qui prend en paramètre d'entrée une liste variable d'années à traiter et un type de vin et retourne le nombre de bouteille du type produite par année.

Triggers Dans cette partie, nous allons travailler avec la base de données cave

Triggers-1

Tracer dans une table toutes les modifications du champ nombre de la table stock.

On veut conserver l'ancienne et la nouvelle valeur. On veut aussi savoir qui a fait la modification et quand. Interdire la suppression des lignes dans stock. Afficher un message dans les logs dans ce cas.

Afficher aussi un message NOTICE quand nombre devient inférieur à 5, et WARNING quand il vaut 0.

Triggers-2

Interdire à tout le monde, sauf un compte admin, l'accès à la table des logs précédemment créée . En conséquence, le trigger fonctionne-t-il ? Le cas échéant, le modifier pour qu'il fonctionne.

Triggers-3

Lire toute la table stock avec un curseur.

Afficher dans les journaux applicatifs toutes les paires (vin id, contenant id) pour chaque nombre supérieur à l'argument de la fonction.
