Module P2

# PL/pgSQL avancé


22.09

Dalibo SCOP

https://dalibo.com/formations

# PL/pgSQL avancé

Module P2

TITRE : PL/pgSQL avancé
SOUS-TITRE : Module P2

REVISION: 22.09
DATE: 02 septembre 2022


---

**Remerciements :** Ce manuel de formation est une aventure collective qui se transmet au sein de notre société depuis des années. Nous remercions chaleureusement ici toutes les personnes qui ont contribué directement ou indirectement à cet ouvrage, notamment : Jean-Paul Argudo, Alexandre Anriot, Carole Arnaud, Alexandre Baron, David Bidoc, Sharon Bonan, Franck Boudehen, Arnaud Bruniquel, Damien Clochard, Christophe Courtois, Marc Cousin, Gilles Darold, Jehan-Guillaume de Rorthais, Ronan Dunklau, Vik Fearing, Stefan Fercot, Pierre Giraud, Nicolas Gollet, Dimitri Fontaine, Florent Jardin, Virginie Jourdan, Luc Lamarle, Denis Laxalde, Guillaume Lelarge, Benoit Lobréau, Jean-Louis Louër, Thibaut Madelaine, Adrien Nayrat, Alexandre Pereira, Flavie Perette, Robin Portigliatti, Thomas Reiss, Maël Rimbault, Julien Rouhaud, Stéphane Schildknecht, Julien Tachoires, Nicolas Thauvin, Be Hai Tran, Christophe Truffier, Cédric Villemain, Thibaud Walkowiak, Frédéric Yhuel.

---

# LICENCE CREATIVE COMMONS BY-NC-SA 2.0 FR

**Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions**

*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

Dalibo 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 Dalibo vous soutient ou soutient la façon dont vous avez utilisé ce document.

*Pas d'Utilisation Commerciale* : Vous n'êtes pas autorisé à faire un usage commercial de ce document, tout ou partie du matériel le composant.

*Partage dans les Mêmes Conditions* : Dans le cas où vous effectuez un remix, que vous transformez, ou créez à partir du matériel composant le document original, vous devez diffuser le document modifié dans les même conditions, c'est à dire avec la même licence avec laquelle le document original a été diffusé.

*Pas de restrictions complémentaires* : Vous n'êtes pas autorisé à appliquer des conditions légales ou des mesures techniques qui restreindraient légalement autrui à utiliser le document dans les conditions décrites par la licence.

Note : Ceci est un résumé de la licence. Le texte complet est disponible ici :

https://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode

Pour toute demande au sujet des conditions d'utilisation de ce document, envoyez vos questions à contact@dalibo.com[1] !

[1] mailto:contact@dalibo.com

**Chers lectrices & lecteurs,**

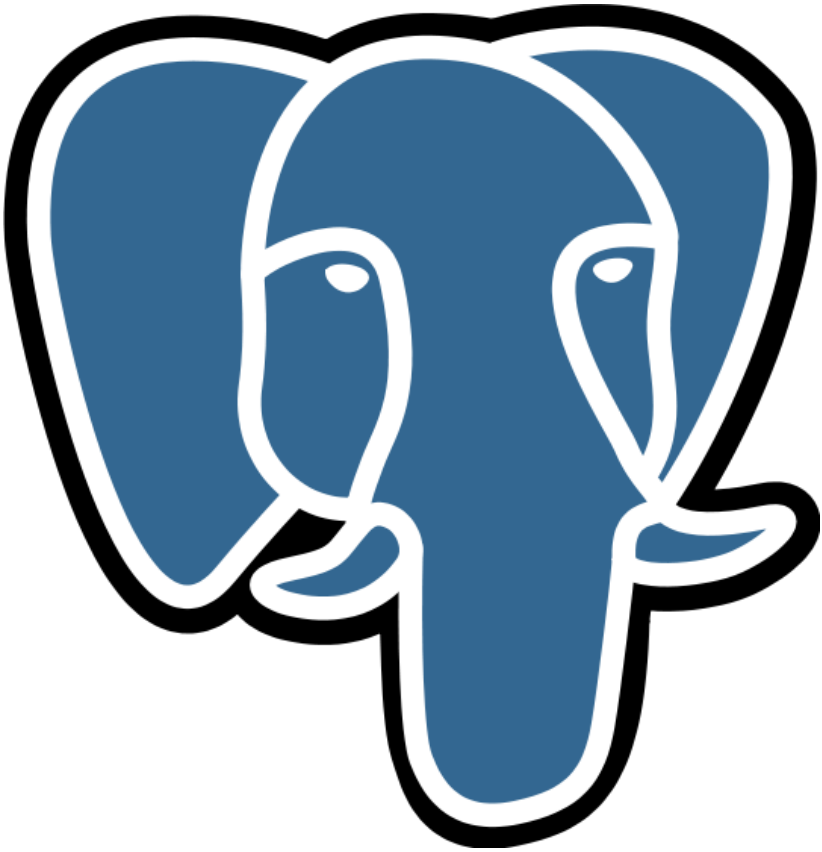
Nos formations PostgreSQL sont issues de nombreuses années d'études, d'expérience de terrain et de passion pour les logiciels libres. Pour Dalibo, l'utilisation de PostgreSQL n'est pas une marque d'opportunisme commercial, mais l'expression d'un engagement de longue date. Le choix de l'Open Source est aussi le choix de l'implication dans la communauté du logiciel.

Au-delà du contenu technique en lui-même, notre intention est de transmettre les valeurs qui animent et unissent les développeurs de PostgreSQL depuis toujours : partage, ouverture, transparence, créativité, dynamisme... Le but premier de nos formations 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 via l'adresse formation@dalibo.com !

# Table des Matières

# 1 PL/PGSQL AVANCÉ



## 1.1 PRÉAMBULE

### 1.1.1 AU MENU

- Routines « variadic » et polymorphes
- Fonctions trigger
- Curseurs
- Récupérer les erreurs
- Messages d'erreur dans les logs
- Sécurité
- Optimisation
- Problèmes fréquents

### 1.1.2 OBJECTIFS

- Connaître la majorité des possibilités de PL/pgSQL
- Les utiliser pour étendre les fonctionnalités de la base
- Écrire du code robuste
- Éviter les pièges de sécurité
- Savoir optimiser une routine

## 1.2 ROUTINES VARIADIC

### 1.2.1 ROUTINES VARIADIC : INTRODUCTION

- Permet de créer des routines avec un nombre d'arguments variables
- … mais du même type

L'utilisation du mot clé `VARIADIC` dans la déclaration des routines permet d'utiliser un nombre variable d'arguments dans la mesure où tous les arguments optionnels sont du même type de données. Ces arguments sont passés à la fonction sous forme de tableau d'arguments du même type.

```
VARIADIC tableau text[]
```

Il n'est pas possible d'utiliser d'autres arguments en entrée à la suite d'un paramètre `VARIADIC`.

### 1.2.2 ROUTINES VARIADIC : EXEMPLE

Récupérer le minimum d'une liste :

```
CREATE FUNCTION pluspetit(VARIADIC numeric[])
RETURNS numeric AS $$
SELECT min($1[i]) FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;

SELECT pluspetit(10, -1, 5, 4.4);
 pluspetit
-----------
        -1
(1 row)
```

Quelques explications sur cette fonction :

- SQL est un langage de routines stockées
  - une routine SQL ne contient que des ordres SQL exécutés séquentiellements
  - le résultat de la fonction est le résultat du dernier ordre
- `generate_subscript()` prend un tableau en premier paramètre et la dimension de ce tableau (un tableau peut avoir plusieurs dimensions), et elle retourne une série d'entiers allant du premier au dernier indice du tableau dans cette dimension
- g(i) est un alias : `generate_subscripts` est une SRF (`set-returning function`, retourne un `SETOF`), g est donc le nom de l'alias de table, et i le nom de l'alias de colonne.

### 1.2.3 ROUTINES VARIADIC : EXEMPLE PL/PGSQL

- En PL/pgSQL, cette fois-ci
- Démonstration de `FOREACH xxx IN ARRAY aaa LOOP`
- Précédemment, obligé de convertir le tableau en relation pour boucler (`unnest`)

En PL/pgSQL, il est possible d'utiliser une boucle `FOREACH` pour parcourir directement le tableau des arguments optionnels.

```
CREATE OR REPLACE FUNCTION pluspetit(VARIADIC liste numeric[])
 RETURNS numeric
 LANGUAGE plpgsql
AS $function$
DECLARE
  courant numeric;
  plus_petit numeric;
```

```
BEGIN
FOREACH courant IN ARRAY liste LOOP
  IF plus_petit IS NULL OR courant < plus_petit THEN
    plus_petit := courant;
  END IF;
END LOOP;
RETURN plus_petit;
END
$function$;
```

Auparavant, il fallait développer le tableau avec la fonction `unnest()` pour réaliser la même opération.

```
CREATE OR REPLACE FUNCTION pluspetit(VARIADIC liste numeric[])
 RETURNS numeric
 LANGUAGE plpgsql
AS $function$
DECLARE
  courant numeric;
  plus_petit numeric;
BEGIN
FOR courant IN SELECT unnest(liste) LOOP
  IF plus_petit IS NULL OR courant < plus_petit THEN
    plus_petit := courant;
  END IF;
END LOOP;
RETURN plus_petit;
END
$function$;
```

---

## 1.3 ROUTINES POLYMORPHES

---

### 1.3.1 ROUTINES POLYMORPHES : INTRODUCTION

- Typer les variables oblige à dupliquer les routines communes à plusieurs types
- PostgreSQL propose des types polymorphes
- Le typage se fait à l'exécution

Pour pouvoir utiliser la même fonction en utilisant des types différents, il est nécessaire de la redéfinir avec les différents types autorisés en entrée. Par exemple, pour autoriser l'utilisation de données de type `integer` ou `float` en entrée et retournés par une même fonction, il faut la dupliquer.

```
CREATE OR REPLACE FUNCTION
  addition(var1 integer, var2 integer)
RETURNS integer
AS $$
DECLARE
  somme integer;
BEGIN
  somme := var1 + var2;
  RETURN somme;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION
  addition(var1 float, var2 float)
RETURNS float
AS $$
DECLARE
  somme float;
BEGIN
  somme := var1 + var2;
  RETURN somme;
END;
$$ LANGUAGE plpgsql;
```

L'utilisation de types polymorphes permet d'éviter ce genre de duplications fastidieuses.

---

### 1.3.2 ROUTINES POLYMORPHES : ANYELEMENT

- Remplace tout type de données simple ou composite
  - pour les paramètres en entrée comme pour les paramètres en sortie
- Tous les paramètres et type de retour de type `anyelement` se voient attribués le même type
- Donc un seul type pour tous les `anyelement` autorisés
- Paramètre spécial `$0` : du type attribué aux éléments `anyelement`

---

### 1.3.3 ROUTINES POLYMORPHES : ANYARRAY

- `anyarray` remplace tout tableau de type de données simple ou composite
  - pour les paramètres en entrée comme pour les paramètres en sortie
- Le typage se fait à l'exécution
- Tous les paramètres de type `anyarray` se voient attribués le même type

---

### 1.3.4 ROUTINES POLYMORPHES : EXEMPLE

L'addition est un exemple fréquent :

```
CREATE OR REPLACE FUNCTION
  addition(var1 anyelement, var2 anyelement)
RETURNS anyelement
AS $$
DECLARE
  somme ALIAS FOR $0;
BEGIN
  somme := var1 + var2;
  RETURN somme;
END;
$$ LANGUAGE plpgsql;
```

---

### 1.3.5 ROUTINES POLYMORPHES : TESTS

```
# SELECT addition(1, 3);
 addition
----------
        4
(1 row)

# SELECT addition(1.3, 3.5);
 addition
----------
      4.8
(1 row)
```

L'opérateur `+` étant défini pour les entiers comme pour les `numeric`, la fonction ne pose aucun problème pour ces deux types de données, et retourne une donnée du même type que les données d'entrée.

---

### 1.3.6 ROUTINES POLYMORPHES : PROBLÈME

- Attention lors de l'utilisation de type polymorphe...

```
# SELECT addition('un'::text, 'mot'::text);
ERREUR:  L'opérateur n'existe pas : text + text
LIGNE 1 : SELECT   $1  +  $2
^
ASTUCE : Aucun opérateur correspond au nom donné et aux types d'arguments.
    Vous devez ajouter des conversions explicites de type.
REQUÊTE : SELECT   $1  +  $2
CONTEXTE : PL/pgSQL function "addition" line 4 at assignment
```

Le typage n'étant connu qu'à l'exécution, c'est aussi à ce moment que se déclenchent les erreurs.

De même, l'affectation du type unique pour tous les éléments se fait sur la base du premier élément, ainsi :

```
# SELECT addition(1, 3.5);
ERROR:  function addition(integer, numeric) does not exist
LIGNE 1 : SELECT addition(1, 3.5);
                ^
ASTUCE : No function matches the given name and argument types.
         You might need to add explicit type casts.
```

génère une erreur car du premier argument est déduit le type `integer`, ce qui n'est évidemment pas le cas du deuxième. Il peut donc être nécessaire d'utiliser une conversion explicite pour résoudre ce genre de problématique.

```
# SELECT addition(1::numeric, 3.5);
 addition
----------
      4.5
```

## 1.4 FONCTIONS TRIGGER

### 1.4.1 FONCTIONS TRIGGER : INTRODUCTION

- Fonction stockée
- Action déclenchée par `INSERT` (incluant `COPY`), `UPDATE`, `DELETE`, `TRUNCATE`
- Mode **par ligne** ou **par instruction**
- Exécution d'une fonction stockée codée à partir de tout langage de procédure activée dans la base de données

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

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

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

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`.

À noter que les problématiques de visibilité et de volatilité depuis un trigger sont assez complexes dès lors que l'on lit ou modifie les données. Voir la documentation[2] pour plus de détails à ce sujet.

[2] https://docs.postgresql.fr/current/trigger-datachanges.html

### 1.4.2 FONCTIONS TRIGGER : VARIABLES (1/5)

- `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`

---

### 1.4.3 FONCTIONS TRIGGER : VARIABLES (2/5)

- Ces deux variables sont valables uniquement pour les triggers en mode ligne
  - pour les triggers en mode instruction, la version 10 propose les tables de transition
- Accès aux champs par la notation pointée
  - `NEW.champ1` pour accéder à la nouvelle valeur de `champ1`

---

### 1.4.4 FONCTIONS TRIGGER : VARIABLES (3/5)

- `TG_NAME`
  - nom du trigger qui a déclenché l'appel de la fonction
- `TG_WHEN`
  - chaîne valant `BEFORE`, `AFTER` ou `INSTEAD OF` suivant le type du trigger
- `TG_LEVEL`
  - chaîne valant `ROW` ou `STATEMENT` suivant le mode du trigger
- `TG_OP`
  - chaîne valant `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE` suivant l'opération qui a déclenché le trigger

---

### 1.4.5 FONCTIONS TRIGGER : VARIABLES (4/5)

- `TG_RELID`
  - `OID` de la table qui a déclenché le trigger
- `TG_TABLE_NAME`
  - nom de la table qui a déclenché le trigger
- `TG_TABLE_SCHEMA`
  - nom du schéma contenant la table qui a déclenché le trigger

Vous pourriez aussi rencontrer dans du code la variable `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`.

---

### 1.4.6 FONCTIONS TRIGGER : VARIABLES (5/5)

- `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);
```

---

### 1.4.7 FONCTIONS TRIGGER : RETOUR

- Une fonction trigger a un type de retour spécial, `trigger`
- Trigger `ROW`, `BEFORE` :
  - si retour NULL, annulation de l'opération, sans déclencher d'erreur
  - sinon, poursuite de l'opération avec cette valeur de ligne
  - attention au `RETURN NEW;` avec trigger `BEFORE DELETE`
- Trigger `ROW`, `AFTER` : valeur de retour ignorée
- Trigger `STATEMENT` : valeur de retour ignorée
- Pour ces deux derniers cas, annulation possible dans le cas d'une erreur à l'exécution de la fonction (que vous pouvez déclencher dans le code du trigger)

Une fonction trigger retourne le type spécial `trigger`. Pour cette raison, ces fonctions ne peuvent être utilisées que dans le contexte d'un ou plusieurs triggers. Pour pouvoir être utilisée comme valeur de retour dans la fonction (avec `RETURN`), une variable doit être de structure identique à celle de la table sur laquelle le trigger a été déclenché. Les variables spéciales `OLD` (ancienne valeur avant application de l'action à l'origine du déclenchement) et `NEW` (nouvelle valeur après application de l'action) sont également disponibles, utilisables et même modifiables.

La valeur de retour d'un trigger de type ligne (`ROW`) déclenché avant l'opération (`BEFORE`) peut changer complètement l'effet de la commande ayant déclenché le trigger. Par exemple, il est possible d'annuler complètement l'action sans erreur (et d'empêcher également tout déclenchement ultérieur d'autres triggers pour cette même action) en retournant `NULL`. Il est également possible de changer les valeurs de la nouvelle ligne créée par une action `INSERT` ou `UPDATE` en retournant une des valeurs différentes de `NEW` (ou en modifiant `NEW` directement). Attention, dans le cas d'une fonction trigger `BEFORE` déclenchée par une action `DELETE`, in faut prendre en compte que `NEW` contient `NULL`, en conséquence

`RETURN NEW;` provoquera l'annulation du `DELETE` ! Dans ce cas, si on désire laisser l'action inchangée, la convention est de faire un `RETURN OLD;`.

En revanche, la valeur de retour utilisée n'a pas d'effet dans les cas des triggers `ROW` et `AFTER`, et des triggers `STATEMENT`. À noter que bien que la valeur de retour soit ignorée dans ce cas, 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.

---

### 1.4.8 FONCTIONS 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);
```

---

### 1.4.9 FONCTIONS 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;
```

---

### 1.4.10 OPTIONS DE CREATE TRIGGER

`CREATE TRIGGER` permet quelques variantes :

- `CREATE TRIGGER name WHEN ( condition )`
- `CREATE TRIGGER name BEFORE UPDATE OF colx ON my_table`
- `CREATE CONSTRAINT TRIGGER` : exécuté qu'au moment de la validation de la transaction
- `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 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 en le déclarant comme étant un trigger de contrainte. Il peut alors être *deferrable*, *deferred*, comme tout autre contrainte, c'est-à-dire n'être exécuté 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.
- 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.

---

### 1.4.11 TABLES DE TRANSITION

- Pour les triggers de type `AFTER` et de niveau statement
- Possibilité de stocker les lignes avant et/ou après modification
  - `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.

La version 10 de PostgreSQL permet donc de rattraper le retard à ce sujet par rapport au

standard SQL et SQL Server.

Voici un exemple de leur utilisation.

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

```
CREATE TABLE t1 (c1 integer, c2 text);

CREATE TABLE archives (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 $$
      BEGIN
        INSERT INTO archives (t1_c1, t1_c2) SELECT c1, c2 FROM oldtable;
        RETURN null;
      END
      $$;
```

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;

DELETE FROM t1;
Time: 2141.871 ms (00:02.142)
```

La suppression avec le trigger prend 2 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 archives;

INSERT INTO t1 SELECT i, 'Ligne '||i FROM generate_series(1, 1000000) i;

EXPLAIN (ANALYZE) DELETE FROM t1;
                                QUERY PLAN
-------------------------------------------------------------------------
 Delete on t1  (cost=0.00..14241.98 rows=796798 width=6)
               (actual time=781.612..781.612 rows=0 loops=1)
```

```
   ->  Seq Scan on t1  (cost=0.00..14241.98 rows=796798 width=6)
                       (actual time=0.113..104.328 rows=1000000 loops=1)
 Planning time: 0.079 ms
 Trigger tr1: time=1501.688 calls=1
 Execution time: 2287.907 ms
(5 rows)
```

Donc la suppression des lignes met 0,7 seconde alors que l'exécution du trigger met 1,5 seconde.

Pour comparer, voici l'ancienne façon de faire (configuration d'un trigger en mode ligne) :

```
CREATE OR REPLACE FUNCTION log_delete() RETURNS trigger LANGUAGE plpgsql AS $$
      BEGIN
        INSERT INTO archives (t1_c1, t1_c2) VALUES (old.c1, old.c2);
        RETURN null;
      END
      $$;

DROP TRIGGER tr1 ON t1;

CREATE TRIGGER tr1
      AFTER DELETE ON t1
      FOR EACH ROW
      EXECUTE PROCEDURE log_delete();

TRUNCATE archives;

TRUNCATE t1;

INSERT INTO t1 SELECT i, 'Ligne '||i FROM generate_series(1, 1000000) i;

DELETE FROM t1;
Time: 8445.697 ms (00:08.446)

TRUNCATE archives;

INSERT INTO t1 SELECT i, 'Ligne '||i FROM generate_series(1, 1000000) i;

EXPLAIN (ANALYZE) DELETE FROM t1;
                              QUERY PLAN
-----------------------------------------------------------------------
 Delete on t1  (cost=0.00..14241.98 rows=796798 width=6)
               (actual time=1049.420..1049.420 rows=0 loops=1)
   ->  Seq Scan on t1  (cost=0.00..14241.98 rows=796798 width=6)
                       (actual time=0.061..121.701 rows=1000000 loops=1)
 Planning time: 0.096 ms
```

```
 Trigger tr1: time=7709.725 calls=1000000
 Execution time: 8825.958 ms
(5 rows)
```

Donc avec un trigger en mode ligne, la suppression du million de lignes met presque 9 secondes à s'exécuter, dont 7,7 pour l'exécution du trigger. Sur le trigger en mode instruction, il faut compter 2,2 secondes, dont 1,5 sur le trigger. Les tables de transition nous permettent de gagner en performance.

Le gros intérêt des tables de transition est le gain en performance que cela apporte.

---

## 1.5 CURSEURS

---

### 1.5.1 CURSEURS : INTRODUCTION

- Exécuter une requête en une fois peut ramener beaucoup de résultats
- Tout ce résultat est en mémoire
  - risque de dépassement mémoire
- La solution : les curseurs
- Un curseur permet d'exécuter la requête sur le serveur mais de ne récupérer les résultats que petit bout par petit bout
- Dans une transaction ou une routine

À noter que la notion de curseur existe aussi en SQL pur, sans passer par une routine PL/pgSQL. On les crée en utilisant la commande `DECLARE`, et les règles de manipulation sont légèrement différentes (on peut par exemple créer un curseur `WITH HOLD`, qui persistera après la fin de la transaction). Voir la documentation pour plus d'informations à ce sujet : https://docs.postgresql.fr/current/sql-declare.html

---

### 1.5.2 CURSEURS : DÉCLARATION D'UN CURSEUR

- Avec le type refcursor
- Avec la pseudo-instruction `CURSOR FOR`
- Avec une requête paramétrée
- Exemples :

```
curseur1 refcursor;
curseur2 CURSOR FOR SELECT * FROM ma_table;
curseur3 CURSOR (param integer) IS
SELECT * FROM ma_table WHERE un_champ=param;
```

La première forme permet la création d'un curseur non lié à une requête.

---

### 1.5.3 CURSEURS : OUVERTURE D'UN CURSEUR

- Lier une requête à un curseur :
  `OPEN curseur FOR requete`
- Plan de la requête mis en cache
- Lier une requête dynamique à un curseur
  `OPEN curseur FOR EXECUTE chaine_requete`

Voici un exemple de lien entre une requête et un curseur :

```
OPEN curseur FOR SELECT * FROM ma_table;
```

Et voici un exemple d'utilisation d'une requête dynamique :

```
OPEN curseur FOR EXECUTE 'SELECT * FROM ' || quote_ident(TG_TABLE_NAME);
```

---

### 1.5.4 CURSEURS : OUVERTURE D'UN CURSEUR LIÉ

- Instruction SQL : `OPEN curseur(arguments)`
- Permet d'ouvrir un curseur déjà lié à une requête
- Impossible d'ouvrir deux fois le même curseur
- Plan de la requête mise en cache
- Exemple

```
curseur CURSOR FOR SELECT * FROM ma_table;
...
OPEN curseur;
```

---

### 1.5.5 CURSEURS : RÉCUPÉRATION DES DONNÉES

- Instruction SQL :
  `FETCH [ direction { FROM | IN } ] curseur INTO cible`
- Récupère la prochaine ligne
- FOUND indique si cette nouvelle ligne a été récupérée
- Cible est
  - une variable RECORD
  - une variable ROW
  - un ensemble de variables séparées par des virgules

### 1.5.6 CURSEURS : RÉCUPÉRATION DES DONNÉES

- `direction` du `FETCH` :
  - `NEXT`, `PRIOR`
  - `FIRST`, `LAST`
  - `ABSOLUTE nombre`, `RELATIVE nombre`
  - nombre
  - `ALL`
  - `FORWARD`, `FORWARD nombre`, `FORWARD ALL`
  - `BACKWARD`, `BACKWARD nombre`, `BACKWARD ALL`

### 1.5.7 CURSEURS : MODIFICATION DES DONNÉES

- Mise à jour d'une ligne d'un curseur :

```
UPDATE une_table SET ...
WHERE CURRENT OF curseur;
```

- Suppression d'une ligne d'un curseur :

```
DELETE FROM une_table
WHERE CURRENT OF curseur;
```

Attention, ces différentes syntaxes ne modifient pas les données dans le curseur en mémoire, mais font réellement la modification dans la table. L'emplacement actuel du curseur est utilisé ici pour identifier la ligne correspondante à mettre à jour.

### 1.5.8 CURSEURS : FERMETURE D'UN CURSEUR

- Instruction SQL : `CLOSE curseur`
- Ferme le curseur
- Permet de récupérer de la mémoire
- Permet aussi de réouvrir le curseur

### 1.5.9 CURSEURS : RENVOI D'UN CURSEUR

- Fonction renvoyant une valeur de type `refcursor`
- Permet donc de renvoyer plusieurs valeurs

Voici un exemple d'utilisation d'une référence de curseur retournée par une fonction :

```
CREATE FUNCTION consult_all_stock(refcursor) RETURNS refcursor AS $$
BEGIN
    OPEN $1 FOR SELECT * FROM stock;
    RETURN $1;
END;
$$ LANGUAGE plpgsql;

-- doit être dans une transaction pour utiliser les curseurs.
BEGIN;

SELECT * FROM consult_all_stock('cursor_a');

FETCH ALL FROM cursor_a;
COMMIT;
```

---

## 1.6 CONTRÔLE TRANSACTIONNEL

- Procédures uniquement !
- `COMMIT` et `ROLLBACK`
- Pas de `BEGIN`
  - automatique après la fin d'une transaction
- Ne fonctionne pas à l'intérieur d'une transaction
- Incompatible avec une clause `EXCEPTION`

Voici un exemple avec `COMMIT` ou `ROLLBACK` suivant que le nombre est pair ou impair :

```
CREATE TABLE test1 (a int) ;

CREATE OR REPLACE PROCEDURE transaction_test1()
LANGUAGE plpgsql
AS $$
BEGIN
  FOR i IN 0..5 LOOP
    INSERT INTO test1 (a) VALUES (i);
    IF i % 2 = 0 THEN
      COMMIT;
    ELSE
      ROLLBACK;
```

```
    END IF;
  END LOOP;
END
$$;

CALL transaction_test1();

SELECT * FROM test1;
 a | b
---+---
 0 |
 2 |
 4 |
 6 |
 8 |
(5 lignes)
```

Une exemple plus fréquemment utilisé est celui d'une procédure effectuant un traitement de modification des données par lots, et donc faisant un `COMMIT` à intervalle régulier.

Noter qu'il n'y a pas de `BEGIN` explicite dans la gestion des transactions. Après un `COMMIT` ou un `ROLLBACK`, un `BEGIN` est immédiatement exécuté.

On ne peut pas imbriquer des transactions :

```
BEGIN ; CALL transaction_test1() ;

ERROR:  invalid transaction termination
CONTEXTE : PL/pgSQL function transaction_test1() line 6 at COMMIT
```

On ne peut pas utiliser en même temps une clause `EXCEPTION` et le contrôle transactionnel :

```
DO LANGUAGE plpgsql $$
BEGIN
  BEGIN
    INSERT INTO test1 (a) VALUES (1);
  COMMIT;
  INSERT INTO test1 (a) VALUES (1/0);
COMMIT;
EXCEPTION
  WHEN division_by_zero THEN
    RAISE NOTICE 'caught division_by_zero';
  END;
END;
$$;

ERREUR:  cannot commit while a subtransaction is active
CONTEXTE : fonction PL/pgSQL inline_code_block, ligne 5 à COMMIT
```

## 1.7 GESTION DES ERREURS

### 1.7.1 GESTION DES ERREURS : INTRODUCTION

- Sans exceptions :
  - toute erreur provoque un arrêt de la fonction
  - toute modification suite à une instruction SQL (`INSERT`, `UPDATE`, `DELETE`) est annulée
  - d'où l'ajout d'une gestion personnalisée des erreurs avec le concept des exceptions

### 1.7.2 GESTION DES ERREURS : UNE EXCEPTION

- La fonction comporte un bloc supplémentaire, `EXCEPTION` :

```
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
```

### 1.7.3 GESTION DES ERREURS : 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

### 1.7.4 GESTION DES ERREURS : 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

---

### 1.7.5 GESTION DES ERREURS : 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 documentation[3]

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 documentation[4] .

[3] https://docs.postgresql.fr/current/errcodes-appendix.html
[4] https://docs.postgresql.fr/current/plpgsql-control-structures.html#plpgsql-exception-diagnostics-values

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 $$
DECLARE
    v_state   TEXT;
    v_msg     TEXT;
    v_detail  TEXT;
    v_hint    TEXT;
    v_context TEXT;
BEGIN
    BEGIN
        INSERT INTO t5 (c1) VALUES ($1);
    EXCEPTION WHEN others THEN
        GET STACKED DIAGNOSTICS
            v_state   = RETURNED_SQLSTATE,
            v_msg     = MESSAGE_TEXT,
            v_detail  = PG_EXCEPTION_DETAIL,
            v_hint    = PG_EXCEPTION_HINT,
            v_context = PG_EXCEPTION_CONTEXT;
        raise notice E'Et une exception :
            state  : %
            message: %
            detail : %
            hint   : %
            context: %', v_state, v_msg, v_detail, v_hint, v_context;
    END;
    RETURN;
END;
$$ 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)
```

---

### 1.7.6 MESSAGES D'ERREURS : RAISE - 1

- Envoyer une trace dans les journaux applicatifs et/ou vers le client
  - `RAISE niveau message`
- Niveau correspond au niveau d'importance du message
  - `DEBUG`, `LOG`, `INFO`, `NOTICE`, `WARNING`, `EXCEPTION`
- Message est la trace à enregistrer
- Message dynamique... tout signe `%` est remplacé par la valeur indiquée après le message
- Champs `DETAIL` et `HINT` disponibles

Il convient de noter qu'un message envoyé de cette manière ne fera pas partie de l'éventuel résultat d'une fonction, et ne sera donc pas exploitable en SQL. Pour cela, il faut utiliser l'instruction `RETURN` avec un type de retour approprié.

Le traitement des messages de ce type et leur destination d'envoi sont contrôlés par le serveur à l'aide des paramètres `log_min_messages` et `client_min_messages`.

---

### 1.7.7 MESSAGES D'ERREURS : RAISE - 2

Exemples :

```
RAISE WARNING 'valeur % interdite', valeur;
RAISE WARNING 'valeur % ambigue',
              valeur
              USING HINT = 'Controlez la valeur saisie en amont';
```

Les autres niveaux pour `RAISE` ne sont que des messages, sans déclenchement d'exception.

---

### 1.7.8 MESSAGES D'ERREURS : CONFIGURATION DES LOGS

- Deux paramètres importants pour les traces
- `log_min_messages`
  - niveau minimum pour que la trace soit enregistrée dans les journaux
- `client_min_messages`
  - niveau minimum pour que la trace soit envoyée au client
- Dans le cas d'un `RAISE NOTICE message`, il faut avoir soit `log_min_messages`, soit `client_min_messages`, soit les deux à la valeur `NOTICE` au minimum.

---

### 1.7.9 MESSAGES D'ERREURS : RAISE EXCEPTION - 1

- Annule le bloc en cours d'exécution
  - `RAISE EXCEPTION message`
- Sauf en cas de présence d'un bloc `EXCEPTION` gérant la condition `RAISE_EXCEPTION`
- message est la trace à enregistrer, et est dynamique... tout signe % est remplacé par la valeur indiquée après le message

---

### 1.7.10 MESSAGES D'ERREURS : RAISE EXCEPTION - 2

Exemple :

```
RAISE EXCEPTION 'erreur interne';
-- La chose à ne pas faire !
```

Le rôle d'une exception est d'intercepter une erreur pour exécuter un traitement permettant soit de corriger l'erreur, soit de remonter une erreur pertinente. Intercepter un problème pour retourner « erreur interne » n'est pas une bonne idée.

---

### 1.7.11 FLUX DES ERREURS DANS DU CODE PL

- Les exceptions non traitées «remontent»
  - de bloc `BEGIN`/`END` imbriqués vers les blocs parents (fonctions appelantes comprises)
  - jusqu'à ce que personne ne puisse les traiter
  - voir note pour démonstration

Démonstration en plusieurs étapes :

```
# CREATE TABLE ma_table (
    id integer unique
);
CREATE TABLE

# CREATE OR REPLACE FUNCTION public.demo_exception()
 RETURNS void
 LANGUAGE plpgsql
AS $function$
DECLARE
BEGIN
  INSERT INTO ma_table VALUES (1);
  -- Va déclencher une erreur de violation de contrainte d'unicité
  INSERT INTO ma_table VALUES (1);
END
$function$;
CREATE FUNCTION

# SELECT demo_exception();
ERROR:  duplicate key value violates unique constraint "ma_table_id_key"
DETAIL:  Key (id)=(1) already exists.
CONTEXT:  SQL statement "INSERT INTO ma_table VALUES (1)"
PL/pgSQL function demo_exception() line 6 at SQL statement
```

Une exception a été remontée avec un message explicite.

```
# SELECT * FROM ma_table ;
 a
---
(0 row)
```

La fonction a bien été annulée.

---

### 1.7.12 FLUX DES ERREURS DANS DU CODE PL - 2

- Les erreurs remontent
- Cette fois-ci, on rajoute un bloc PL pour intercepter l'erreur

```
# CREATE OR REPLACE FUNCTION public.demo_exception()
 RETURNS void
 LANGUAGE plpgsql
AS $function$
DECLARE
BEGIN
  INSERT INTO ma_table VALUES (1);
  -- Va déclencher une erreur de violation de contrainte d'unicité
```

```
  INSERT INTO ma_table VALUES (1);
EXCEPTION WHEN unique_violation THEN
  RAISE NOTICE 'violation d''unicite, mais celle-ci n''est pas grave';
  RAISE NOTICE 'erreur: %',sqlerrm;
END
$function$;
CREATE FUNCTION

# SELECT demo_exception();
NOTICE:  violation d'unicite, mais celle-ci n'est pas grave
NOTICE:  erreur: duplicate key value violates unique constraint "ma_table_id_key"
 demo_exception
----------------

(1 row)
```

L'erreur est bien devenue un message de niveau `NOTICE`.

```
# SELECT * FROM ma_table ;
 a
---
(0 row)
```

La table n'en reste pas moins vide pour autant puisque le bloc a été annulé.

---

### 1.7.13 FLUX DES ERREURS DANS DU CODE PL - 3

- Cette fois-ci, on rajoute un bloc PL indépendant pour gérer le second `INSERT`

Voici une nouvelle version de la fonction :

```
# CREATE OR REPLACE FUNCTION public.demo_exception()
 RETURNS void
 LANGUAGE plpgsql
AS $function$
DECLARE
BEGIN
  INSERT INTO ma_table VALUES (1);
  -- L'operation suivante pourrait échouer.
  -- Il ne faut pas perdre le travail effectué jusqu'à ici
  BEGIN
  -- Va déclencher une erreur de violation de contrainte d'unicité
    INSERT INTO ma_table VALUES (1);
  EXCEPTION WHEN unique_violation THEN
    -- Cette exception est bien celle du bloc imbriqué
    RAISE NOTICE 'violation d''unicite, mais celle-ci n''est pas grave';
```

```
    RAISE NOTICE 'erreur: %',sqlerrm;
  END; -- Fin du bloc imbriqué
END
$function$;
CREATE FUNCTION

# SELECT demo_exception();
NOTICE:  violation d'unicite, mais celle-ci n'est pas grave
NOTICE:  erreur: duplicate key value violates unique constraint "ma_table_id_key"
 demo_exception
----------------

(1 row)
```

En apparence, le résultat est identique.

```
# SELECT * FROM ma_table ;
 a
---
 1
(1 row)
```

Mais cette fois-ci, le bloc `BEGIN` parent n'a pas eu d'exception, il s'est donc bien terminé.

---

### 1.7.14 FLUX DES ERREURS DANS DU CODE PL - 4

- Illustrons maintenant la remontée d'erreurs
- Nous avons deux blocs imbriqués
- Une erreur non prévue va se produire dans le bloc intérieur

On commence par ajouter une contrainte sur la colonne pour empêcher les valeurs supérieures ou égales à 10 :

```
# ALTER TABLE ma_table ADD CHECK (id < 10 ) ;
ALTER TABLE
```

Puis, on recrée la fonction de façon à ce qu'elle déclenche cette erreur dans le bloc le plus bas, et la gère uniquement dans le bloc parent :

```
CREATE OR REPLACE FUNCTION public.demo_exception()
 RETURNS void
 LANGUAGE plpgsql
AS $function$
DECLARE
BEGIN
  INSERT INTO ma_table VALUES (1);
```

```
  -- L'operation suivante pourrait échouer.
  -- Il ne faut pas perdre le travail effectué jusqu'à ici
  BEGIN
    -- Va déclencher une erreur de violation de check (col < 10)
    INSERT INTO ma_table VALUES (100);
  EXCEPTION WHEN unique_violation THEN
    -- Cette exception est bien celle du bloc imbriqué
    RAISE NOTICE 'violation d''unicite, mais celle-ci n''est pas grave';
    RAISE NOTICE 'erreur: %',sqlerrm;
  END; -- Fin du bloc imbriqué
EXCEPTION WHEN check_violation THEN
  RAISE NOTICE 'violation de contrainte check';
  RAISE EXCEPTION 'mais on va remonter une exception à l''appelant, '
                  'juste pour le montrer';
END
$function$;
```

Exécutons la fonction :

```
# SELECT demo_exception();
ERROR:  duplicate key value violates unique constraint "ma_table_id_key"
DETAIL:  Key (id)=(1) already exists.
CONTEXT:  SQL statement "INSERT INTO ma_table VALUES (1)"
PL/pgSQL function demo_exception() line 4 at SQL statement
```

C'est normal, nous avons toujours l'enregistrement à 1 du test précédent. L'exception se déclenche donc dans le bloc parent, sans espoir d'interception: nous n'avons pas d'exception pour lui.

Nettoyons donc la table, pour reprendre le test :

```
# TRUNCATE ma_table ;
TRUNCATE TABLE
# SELECT demo_exception();
NOTICE:  violation de contrainte check
ERREUR:  mais on va remonter une exception à l'appelant, juste pour le montrer
CONTEXT:  PL/pgSQL function demo_exception() line 17 at RAISE
```

Le gestionnaire d'exception qui intercepte l'erreur est bien ici celui de l'appelant. Par ailleurs, comme nous retournons nous-même une exception, la requête ne retourne pas de résultat, mais une erreur : il n'y a plus personne pour récupérer l'exception, c'est donc PostgreSQL lui-même qui s'en charge.

---

# 1.8 SÉCURITÉ

## 1.8.1 SÉCURITÉ : DROITS

- L'exécution de la routine dépend du droit EXECUTE
- Par défaut, ce droit est donné à la création de la routine
  - au propriétaire de la routine
  - au groupe spécial PUBLIC

## 1.8.2 SÉCURITÉ : AJOUT

- Ce droit peut être donné avec l'instruction SQL GRANT :

```
GRANT { EXECUTE | ALL [ PRIVILEGES ] }
    ON { { FUNCTION | PROCEDURE | ROUTINE } routine_name
    [ ( [ [ argmode ] [ arg_name ] arg_type [, ...] ] ) ] [, ...  ]
         | ALL { FUNCTIONS | PROCEDURES | ROUTINES } IN SCHEMA schema_name [, ...] }
    TO role_specification [, ...] [ WITH GRANT OPTION ]
```

## 1.8.3 SÉCURITÉ : SUPPRESSION

- Un droit peut être révoqué avec l'instruction SQL REVOKE

```
REVOKE [ GRANT OPTION FOR ]
    { EXECUTE | ALL [ PRIVILEGES ] }
    ON { { FUNCTION | PROCEDURE | ROUTINE } function_name
    [ ( [ [ argmode ] [ arg_name ] arg_type [, ...] ] ) ] [, ... ]
         | ALL { FUNCTIONS | PROCEDURES | ROUTINES } IN SCHEMA schema_name [, ...] }
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

### 1.8.4 SÉCURITÉ : SECURITY INVOKER/DEFINER

- `SECURITY INVOKER`
  - la routine s'exécute avec les droits de l'utilisateur qui l'exécute
- `SECURITY DEFINER`
  - la routine s'exécute avec les droits de l'utilisateur qui en est le propriétaire
  - équivalent du `sudo` Unix
- Il faut impérativement sécuriser les variables d'environnement (surtout le *search path*) en `SECURITY DEFINER`

Exemple d'une fonction en `SECURITY DEFINER` avec un *search path* sécurisé :

```
CREATE OR REPLACE FUNCTION instance_is_in_backup ( )
RETURNS BOOLEAN AS $$
DECLARE is_exists BOOLEAN;
BEGIN
        -- Set a secure search_path: trusted schemas, then 'pg_temp'.
        PERFORM pg_catalog.set_config('search_path', 'pg_temp', true);
        SELECT ((pg_stat_file('backup_label')).modification IS NOT NULL)
        INTO is_exists;
        RETURN is_exists;
EXCEPTION
WHEN undefined_file THEN
  RETURN false;
END
$$ LANGUAGE plpgsql SECURITY DEFINER;
```

---

### 1.8.5 SÉCURITÉ : LEAKPROOF

- `LEAKPROOF`
  - indique au planificateur que la routine ne peut pas faire fuiter d'information de contexte
  - réservé aux superutilisateurs
  - si on la déclare telle, s'assurer que la routine est véritablement sûre !
- Option utile lorsque l'on utilise des vues avec l'option `security_barrier`

Certains utilisateurs créent des vues pour filtrer des lignes, afin de restreindre la visibilité sur certaines données. Or, cela peut se révéler dangereux si un utilisateur malintentionné a la possibilité de créer une fonction car il peut facilement contourner cette sécurité si cette option n'est pas utilisée, notamment en jouant sur des paramètres de fonction comme `COST`, qui permet d'indiquer au planificateur un coût estimé pour la fonction.

En indiquant un coût extrêmement faible, le planificateur aura tendance à réécrire la

requête, et à déplacer l'exécution de la fonction dans le code même de la vue, avant l'application des filtres restreignant l'accès aux données : la fonction a donc accès à tout le contenu de la table, et peut faire fuiter des données normalement inaccessibles, par exemple à travers l'utilisation de la commande `RAISE`.

L'option `security_barrier` des vues dans PostgreSQL bloque ce comportement du planificateur, mais en conséquence empêche le choix de plans d'exécutions potentiellement plus performants. Déclarer une fonction avec l'option `LEAKPROOF` permet d'indiquer à PostgreSQL que celle-ci ne peut pas occasionner de fuite d'informations. Ainsi, le planificateur de PostgreSQL sait qu'il peut en optimiser l'exécution. Cette option n'est accessible qu'aux superutilisateurs.

---

### 1.8.6 SÉCURITÉ : VISIBILITÉ DES SOURCES - 1

- Le code d'une fonction est visible par tout le monde
  - y compris ceux qui n'ont pas le droit d'exécuter la fonction
- Vous devez donc écrire un code robuste
  - pas espérer que, comme personne n'en a le code, personne ne trouvera de faille
- **Surtout pour les fonctions SECURITY DEFINER**

---

### 1.8.7 SÉCURITÉ : VISIBILITÉ DES SOURCES - 2

```
# SELECT proargnames, prosrc
FROM pg_proc WHERE proname='addition';

-[ RECORD 1 ]-------------------------
proargnames | {var1,var2}
prosrc      |
            :   DECLARE
            :     somme ALIAS FOR $0;
            :   BEGIN
            :     somme := var1 + var2;
            :     RETURN somme;
            :   END;
            :
```

La méta-commande psql `\df+ public.addition` permet également d'obtenir cette information.

---

### 1.8.8 SÉCURITÉ : INJECTIONS SQL

- Les paramètres d'une routine doivent être considérés comme hostiles :
  - ils contiennent des données non validées (qui appelle la routine ?)
  - ils peuvent, si l'utilisateur est imaginatif, être utilisés pour exécuter du code
- Utiliser `quote_ident`, `quote_literal` et `quote_nullable`
- Utiliser aussi `format`

Voici un exemple simple :

```
CREATE TABLE ma_table_secrete1 (b integer, a integer);
INSERT INTO ma_table_secrete1 SELECT i,i from generate_series(1,20) i;

CREATE OR REPLACE FUNCTION demo_injection ( param1 text, value1 text )
 RETURNS SETOF ma_table_secrete1
 LANGUAGE plpgsql
 SECURITY DEFINER
AS $function$
-- Cette fonction prend un nom de colonne variable
-- et l'utilise dans une clause WHERE
-- Il faut donc une requête dynamique
-- Par contre, mon utilisateur 'normal' qui appelle
-- n'a droit qu'aux enregistrements où a<10
DECLARE
  ma_requete text;
  ma_ligne record;
BEGIN
  ma_requete := 'SELECT * FROM ma_table_secrete1 WHERE ' || param1 || ' = ' ||
                value1 || ' AND a < 10';
    RETURN QUERY EXECUTE ma_requete;
END
$function$;

# SELECT * from demo_injection ('b','2');
 a | b
---+---
 2 | 2
(1 row)

# SELECT * from demo_injection ('a','20');
 a | b
---+---
(0 row)
```

Tout va bien, elle effectue ce qui est demandé.

Par contre, elle effectue aussi ce qui n'est pas prévu :

```
# SELECT * from demo_injection ('1=1 --','');
  a  |  b
-----+-----
   1 |   1
   2 |   2
   3 |   3
   4 |   4
   5 |   5
   6 |   6
   7 |   7
   8 |   8
   9 |   9
  10 |  10
  11 |  11
  12 |  12
  13 |  13
  14 |  14
  15 |  15
  16 |  16
  17 |  17
  18 |  18
  19 |  19
  20 |  20
(20 lignes)
```

Cet exemple est évidemment simplifié.

Une règle demeure : ne jamais faire confiance aux paramètres d'une fonction. Au minimum, un `quote_ident` pour param1 et un `quote_literal` pour val1 étaient obligatoires, pour se protéger de ce genre de problèmes.

---

## 1.9 OPTIMISATION

---

### 1.9.1 FONCTIONS IMMUABLES, STABLES OU VOLATILES - 1

- Par défaut, PostgreSQL considère que les fonctions sont `VOLATILE`
- `volatile` : fonction dont l'exécution ne peut ni ne doit être évitée

Les fonctions de ce type sont susceptibles de renvoyer un résultat différent à chaque appel, comme par exemple `random()` ou `setval()`.

Toute fonction ayant des effets de bords doit être qualifiée `volatile` dans le but d'éviter

que PostgreSQL utilise un résultat intermédiaire déjà calculé et évite ainsi d'exécuter le code de la fonction.

À noter qu'il est possible de « forcer » le pré-calcul du résultat d'une fonction volatile dans une requête SQL en utilisant une sous-requête. Par exemple, dans l'exemple suivant, `random()` est exécutée pour chaque ligne de la table `ma_table`, et renverra donc une valeur différente par ligne :

```
SELECT random() FROM ma_table;
```

Par contre, en utilisant une sous-requête, l'optimiseur va pré-calculer le résultat de `random()`... l'exécution sera donc plus rapide, mais le résultat différent, puisque la même valeur sera affichée pour toutes les lignes !

```
SELECT ( SELECT random() ) FROM ma_table;
```

---

### 1.9.2 FONCTIONS IMMUABLES, STABLES OU VOLATILES - 2

- `immutable` : fonctions déterministes, dont le résultat peut être précalculé avant de planifier la requête.

Certaines fonctions que l'on écrit sont déterministes. C'est-à-dire qu'à paramètre(s) identique(s), le résultat est identique.

Le résultat de telles fonctions est alors remplaçable par son résultat avant même de commencer à planifier la requête.

Voici un exemple qui utilise cette particularité :

```
create function factorielle (a integer) returns bigint as
$$
declare
  result bigint;
begin
  if a=1 then
    return 1;
  else
    return a*(factorielle(a-1));
  end if;
end;
$$
language plpgsql immutable;

# CREATE TABLE test (a bigint UNIQUE);
CREATE TABLE
# INSERT INTO test SELECT generate_series(1,1000000);
```

```
INSERT 0 1000000
# ANALYZE test;
# EXPLAIN ANALYZE SELECT * FROM test WHERE a < factorielle(12);
                                  QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on test  (cost=0.00..16925.00 rows=1000000 width=8)
                   (actual time=0.032..130.921 rows=1000000 loops=1)
   Filter: (a < '479001600'::bigint)
 Planning time: 896.039 ms
 Execution time: 169.954 ms
(4 rows)
```

La fonction est exécutée une fois, remplacée par sa constante, et la requête est ensuite planifiée.

Si on déclare la fonction comme stable :

```
# EXPLAIN ANALYZE SELECT * FROM test WHERE a < factorielle(12);
                       QUERY PLAN
-------------------------------------------------------------
 Index Only Scan using test_a_key on test
       (cost=0.68..28480.67 rows=1000000 width=8)
             (actual time=0.137..115.592 rows=1000000 loops=1)
   Index Cond: (a < factorielle(12))
   Heap Fetches: 0
 Planning time: 4.682 ms
 Execution time: 153.762 ms
(5 rows)
```

La requête est planifiée sans connaître `factorielle(12)`, donc avec une hypothèse très approximative sur la cardinalité. `factorielle(12)` est calculé, et la requête est exécutée. Grâce au `Index Only Scan`, le requête s'effectue rapidement.

Si on déclare la fonction comme volatile :

```
# EXPLAIN ANALYZE SELECT * FROM test WHERE a < factorielle(12);
                                  QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on test  (cost=0.00..266925.00 rows=333333 width=8)
                   (actual time=1.005..57519.702 rows=1000000 loops=1)
   Filter: (a < factorielle(12))
 Planning time: 0.388 ms
 Execution time: 57573.508 ms
(4 rows)
```

La requête est planifiée, et factorielle(12) est calculé pour chaque enregistrement de la table, car on ne sait pas si elle retourne toujours le même résultat.

### 1.9.3 FONCTIONS IMMUABLES, STABLES OU VOLATILES - 3

- `stable` : fonction ayant un comportement **stable** au sein d'un même ordre SQL.

Ces fonctions retournent la même valeur pour la même requête SQL, mais peuvent retourner une valeur différente dans la prochaine instruction.

Il s'agit typiquement de fonctions dont le traitement dépend d'autres valeurs dans la base de données, ou bien de réglages de configuration. Les fonctions comme `to_char()`, `to_date()` sont `STABLE` et non `IMMUTABLE` car des paramètres de configuration (*locale* utilisée pour `to_char()`, *timezone* pour les fonctions temporelles, etc.) pourraient influer sur le résultat.

À noter au passage que les fonctions de la famille de `current_timestamp` (et donc le fréquemment utilisé `now()`) renvoient de plus une valeur constante au sein d'une même transaction.

PostgreSQL refusera de déclarer comme `STABLE` toute fonction modifiant des données : elle ne peut pas être stable si elle modifie la base.

### 1.9.4 OPTIMISATION : RIGUEUR

- Fonction `STRICT`
- La fonction renvoie `NULL` si au moins un des arguments est `NULL`

Les fonctions définies comme `STRICT` ou `RETURNS NULL ON NULL INPUT` annule l'exécution de la requête si l'un des paramètres passés est `NULL`. Dans ce cas, la fonction est considérée comme ayant renvoyé `NULL`.

Si l'on reprend l'exemple de la fonction `factorielle()` :

```
create or replace function factorielle (a integer) returns bigint as
$$
declare
  result bigint;
begin
  if a=1 then
    return 1;
  else
    return a*(factorielle(a-1));
  end if;
end;
```

```
$$
language plpgsql immutable STRICT;
```

on obtient le résultat suivant si elle est exécutée avec la valeur `NULL` passée en paramètre :

```
# EXPLAIN ANALYZE SELECT * FROM test WHERE a < factorielle(NULL);
                  QUERY PLAN
-----------------------------------------------------
 Result  (cost=0.00..0.00 rows=0 width=8)
         (actual time=0.002..0.002 rows=0 loops=1)
   One-Time Filter: false
 Planning time: 0.100 ms
 Execution time: 0.039 ms
(4 rows)
```

---

### 1.9.5 OPTIMISATION : EXCEPTION

- Un bloc contenant une clause `EXCEPTION` est plus coûteuse en entrée/sortie qu'un bloc sans
  - un `SAVEPOINT` est créé à chaque fois pour pouvoir annuler le bloc uniquement.
- À utiliser avec parcimonie
- Un bloc `BEGIN` imbriqué a un coût aussi
  - un `SAVEPOINT` est créé à chaque fois.

---

### 1.9.6 REQUÊTE STATIQUE OU DYNAMIQUE ?

- Les requêtes statiques :
  - sont écrites « en dur » dans le code PL/pgSQL
  - donc pas d'`EXECUTE` ou `PERFORM`
  - sont préparées une fois par session, à leur première exécution
  - peuvent avoir un plan générique lorsque c'est jugé utile par le planificateur

Avant la version 9.2, un plan générique (indépendant des paramètres de l'ordre SQL) était systématiquement généré et utilisé. Ce système permet de gagner du temps d'exécution si la requête est réutilisée plusieurs fois, et qu'elle est coûteuse à planifier.

Toutefois, un plan générique n'est pas forcément idéal dans toutes les situations, et peut conduire à des mauvaises performances.

Par exemple :

```
SELECT * FROM ma_table WHERE col_pk = param_function ;
```

est un excellent candidat à être écrit statiquement : le plan sera toujours le même : on attaque l'index de la clé primaire pour trouver l'enregistrement.

```
SELECT * FROM ma_table WHERE col_timestamp > param_function ;
```

est un moins bon candidat : le plan, idéalement, dépend de param_function : on ne parcourt pas la même fraction de la table suivant la valeur de param_function.

Par défaut, un plan générique ne sera utilisé dès la première exécution d'une requête statique que si celle-ci ne dépend d'aucun paramètre. Dans le cas contraire, cela ne se produira qu'au bout de plusieurs exécutions de la requête, et seulement si le planificateur détermine que les plans spécifiques utilisés n'apportent pas d'avantage par rapport au plan générique.

---

### 1.9.7 REQUÊTE STATIQUE OU DYNAMIQUE ? - 2

- Les requêtes dynamiques :
  - sont écrites avec un `EXECUTE`, `PERFORM`...
  - sont préparées à chaque exécution
  - ont un plan optimisé
  - sont donc plus coûteuses en planification
  - mais potentiellement plus rapides à l'exécution

L'écriture d'une requête dynamique est par contre un peu plus pénible, puisqu'il faut fabriquer un ordre SQL, puis le passer en paramètre à `EXECUTE`, avec tous les quote_* que cela implique pour en protéger les paramètres.

Pour se faciliter la vie, on peut utiliser `EXECUTE query USING param1, param2 …,` qui est même quelquefois plus lisible que la syntaxe en dur : les paramètres de la requête sont clairement identifiés dans cette syntaxe.

Par contre, la syntaxe `USING` n'est utilisable que si le nombre de paramètres est fixe.

---

### 1.9.8 REQUÊTE STATIQUE OU DYNAMIQUE ? -3

- Alors, statique ou dynamique ?
- Si la requête est simple : statique
  - peu de `WHERE`
  - peu ou pas de jointure
- Sinon dynamique

La limite est difficile à placer, il s'agit de faire un compromis entre le temps de planification d'une requête (quelques dizaines de microsecondes pour une requête basique à potentiellement plusieurs secondes si on dépasse la dizaine de jointures) et le temps d'exécution.

Dans le doute, réalisez un test de performance de la fonction sur un jeu de données représentatif.

---

## 1.10 OUTILS

- Deux outils disponibles
  - un debugger
  - un pseudo-profiler

Tous les outils d'administration PostgreSQL permettent d'écrire des routines stockées en PL/pgSQL, la plupart avec les fonctionnalités habituelles (comme le surlignage des mots clés, l'indentation automatique, etc.).

Par contre, pour aller plus loin, l'offre est restreinte. Il existe tout de même un debugger qui fonctionne avec pgAdmin 4, sous la forme d'une extension.

---

### 1.10.1 PLDEBUGGER

- License Artistic 2.0
- Développé par EDB et intégrable dans pgAdmin
- Installé par défaut avec le one-click installer
  - mais non activé
- Compilation nécessaire pour les autres systèmes

pldebugger est un outil initialement créé par Dave Page et Korry Douglas au sein d'EnterpriseDB, repris par la communauté. Il est proposé sous license libre (Artistic 2.0).

Il fonctionne grâce à des hooks implémentés dans la version 8.2 de PostgreSQL.

Il est assez peu connu, ce qui explique que peu l'utilisent. Seul l'outil d'installation « one-click installer » l'installe par défaut. Pour tous les autres systèmes, cela réclame une compilation supplémentaire. Cette compilation est d'ailleurs peu aisée étant donné qu'il n'utilise pas le système pgxs.

---

### 1.10.2 PLDEBUGGER - COMPILATION

- Récupérer le source avec git
- Copier le répertoire dans le répertoire contrib des sources de PostgreSQL
- Et les suivre étapes standards
  - make
  - make install

Voici les étapes à réaliser pour compiler pldebugger en prenant pour hypothèse que les sources de PostgreSQL sont disponibles dans le répertoire `/usr/src/postgresql-10` et qu'ils ont été préconfigurés avec la commande `./configure` :

- Se placer dans le répertoire contrib des sources de PostgreSQL :

```
$ cd /usr/src/postgresql-10/contrib
```

- Cloner le dépôt git :

```
$ git clone git://git.postgresql.org/git/pldebugger.git
Cloning into 'pldebugger'...
remote: Counting objects: 441, done.
remote: Compressing objects: 100% (337/337), done.
remote: Total 441 (delta 282), reused 171 (delta 104)
Receiving objects: 100% (441/441), 170.24 KiB, done.
Resolving deltas: 100% (282/282), done.
```

- Se placer dans le nouveau répertoire `pldebugger` :

```
$ cd pldebugger
```

- Compiler pldebugger :

```
$ make
```

- Installer pldebugger :

```
# make install
```

L'installation copie le fichier `plugin_debugger.so` dans le répertoire des bibliothèques partagées de PostgreSQL. L'installation copie ensuite les fichiers SQL et de contrôle de l'extension `pldbgapi` dans le répertoire `extension` du répertoire `share` de PostgreSQL.

### 1.10.3 PLDEBUGGER - ACTIVATION

- Configurer `shared_preload_libraries`
  - `shared_preload_libraries = 'plugin_debugger'`
- Redémarrer PostgreSQL
- Installer l'extension `pldbgapi` :

```
CREATE EXTENSION pldbgapi;
```

La configuration du paramètre `shared_preload_libraries` permet au démarrage de PostgreSQL de laisser la bibliothèque `plugin_debugger` s'accrocher aux hooks de l'interpréteur PL/pgSQL. Du coup, pour que la modification de ce paramètre soit prise en compte, il faut redémarrer PostgreSQL.

L'interaction avec pldebugger se fait par l'intermédiaire de procédures stockées. Il faut donc au préalable créer ces procédures stockées dans la base contenant les procédures PL/pgSQL à débugguer. Cela se fait en créant l'extension :

```
$ psql
psql (13.0)
Type "help" for help.

postgres# create extension pldbgapi;
CREATE EXTENSION
```

### 1.10.4 AUTO_EXPLAIN

- Mise en place globale (traces) :
  - `shared_preload_libraries='auto_explain'` si global
  - `ALTER DATABASE erp SET auto_explain.log_min_duration = '3s'`
- Ou par session :
  - `LOAD 'auto_explain'`
  - `SET auto_explain.log_analyze TO true;`
  - `SET auto_explain.log_nested_statements TO true;`

`auto_explain` est une « contrib » officielle de PostgreSQL (et non une extension). Il permet de tracer le plan d'une requête. En général, on ne trace ainsi que les requêtes dont la durée d'exécution dépasse la durée configurée avec le paramètre'`auto_explain.log_min_duration`. Par défaut, ce paramètre est à -1 pour ne tracer aucun plan.

Comme dans un `EXPLAIN` classique, on peut activer toutes les options (par exemple `ANALYZE` ou `TIMING` avec, respectivement `SET auto_explain.log_analyze TO true;` et `SET auto_explain.log_timing TO true;`) mais l'impact en performance peut être important même pour les requêtes qui ne seront pas tracées.

D'autres options existent, qui reprennent les paramètres habituels d'`EXPLAIN`, notamment `auto_explain.log_buffers` et `auto_explain.log_settings` (voir la documentation[5] ).

L'exemple suivant utilise deux fonctions imbriquées mais cela marche pour une simple requête :

```
CREATE OR REPLACE FUNCTION table_nb_indexes (tabname IN text, nbi OUT int)
RETURNS int
LANGUAGE plpgsql
AS $$
BEGIN
    SELECT  COUNT(*) INTO nbi
    FROM    pg_index i INNER JOIN pg_class c ON (c.oid=indrelid)
    WHERE   relname LIKE tabname ;
    RETURN ;
END ;
$$
;
CREATE OR REPLACE FUNCTION table_nb_col_indexes
                  (tabname IN text, nb_cols OUT int, nb_indexes OUT int)
RETURNS record
LANGUAGE plpgsql
AS $$
BEGIN
    SELECT  COUNT(*) INTO nb_cols
    FROM    pg_attribute
    WHERE   attname LIKE tabname ;

    SELECT nbi INTO nb_indexes FROM table_nb_indexes (tabname) ;

    RETURN ;
END ;
$$
;
```

Chargement dans la session d'`auto_explain` (si pas déjà présent dans `shared_preload_libraries`) :

```
LOAD 'auto_explain' ;
```

[5] https://docs.postgresql.fr/current/auto-explain.html

Activation pour toutes les requêtes, avec les options `ANALYZE` et `BUFFERS`, puis affichage dans la console (si la sortie dans les traces ne suffit pas) :

```
SET auto_explain.log_min_duration TO 0 ;
SET auto_explain.log_analyze TO on ;
SET auto_explain.log_buffers TO on ;
SET client_min_messages TO log ;
```

Test de la première fonction : le plan s'affiche, mais les compteurs (ici juste *shared hit*), ne concernent que la fonction dans son ensemble.

```
postgres=# SELECT * FROM table_nb_col_indexes ('pg_class') ;

LOG:  duration: 2.208 ms  plan:
Query Text: SELECT * FROM table_nb_col_indexes ('pg_class') ;
Function Scan on table_nb_col_indexes  (cost=0.25..0.26 rows=1 width=8)
                              (actual time=2.203..2.203 rows=1 loops=1)
  Buffers: shared hit=294

 nb_cols | nb_indexes
---------+------------
       0 |          3
```

En activant `auto_explain.log_nested_statements`, on voit clairement les plans de chaque requête exécutée :

```
SET auto_explain.log_nested_statements TO on ;

postgres=# SELECT * FROM table_nb_col_indexes ('pg_class') ;

LOG:  duration: 0.235 ms  plan:
Query Text: SELECT  COUNT(*)                  FROM    pg_attribute
    WHERE   attname LIKE tabname
Aggregate  (cost=65.95..65.96 rows=1 width=8)
    (actual time=0.234..0.234 rows=1 loops=1)
  Buffers: shared hit=24
  ->  Index Only Scan using pg_attribute_relid_attnam_index on pg_attribute
                                      (cost=0.28..65.94 rows=1 width=0)
                            (actual time=0.233..0.233 rows=0 loops=1)
        Index Cond: ((attname >= 'pg'::text) AND (attname < 'ph'::text))
        Filter: (attname ~~ 'pg_class'::text)
        Heap Fetches: 0
        Buffers: shared hit=24

LOG:  duration: 0.102 ms  plan:
Query Text: SELECT  COUNT(*)              FROM    pg_index i
    INNER JOIN pg_class c ON (c.oid=indrelid)
    WHERE   relname LIKE tabname
```

```
Aggregate  (cost=24.48..24.49 rows=1 width=8)
    (actual time=0.100..0.100 rows=1 loops=1)
  Buffers: shared hit=18
  ->  Nested Loop  (cost=0.14..24.47 rows=1 width=0)
          (actual time=0.096..0.099 rows=3 loops=1)
        Buffers: shared hit=18
        ->  Seq Scan on pg_class c  (cost=0.00..23.30 rows=1 width=4)
                        (actual time=0.091..0.093 rows=1 loops=1)
              Filter: (relname ~~ 'pg_class'::text)
              Rows Removed by Filter: 580
              Buffers: shared hit=16
        ->  Index Only Scan using pg_index_indrelid_index on pg_index i
                                    (cost=0.14..1.16 rows=1 width=4)
                         (actual time=0.003..0.004 rows=3 loops=1)
              Index Cond: (indrelid = c.oid)
              Heap Fetches: 0
              Buffers: shared hit=2

LOG:  duration: 0.703 ms  plan:
Query Text: SELECT nbi                 FROM table_nb_indexes (tabname)
Function Scan on table_nb_indexes  (cost=0.25..0.26 rows=1 width=4)
                        (actual time=0.702..0.702 rows=1 loops=1)
  Buffers: shared hit=26

LOG:  duration: 1.524 ms  plan:
Query Text: SELECT * FROM table_nb_col_indexes ('pg_class') ;
Function Scan on table_nb_col_indexes  (cost=0.25..0.26 rows=1 width=8)
                            (actual time=1.520..1.520 rows=1 loops=1)
  Buffers: shared hit=59

 nb_cols | nb_indexes
---------+------------
       0 |          3
```

Cet exemple permet de mettre le doigt sur un petit problème de performance dans la fonction : le _ est interprété comme critère de recherche. En modifiant le paramètre on peut constater le changement de plan au niveau des index :

```
postgres=# SELECT * FROM table_nb_col_indexes ('pg\_class') ;

LOG:  duration: 0.141 ms  plan:
Query Text: SELECT  COUNT(*)                  FROM    pg_attribute
    WHERE   attname LIKE tabname
Aggregate  (cost=56.28..56.29 rows=1 width=8)
    (actual time=0.140..0.140 rows=1 loops=1)
  Buffers: shared hit=24
```

```
  ->  Index Only Scan using pg_attribute_relid_attnam_index on pg_attribute
                                        (cost=0.28..56.28 rows=1 width=0)
                                  (actual time=0.138..0.138 rows=0 loops=1)
        Index Cond: (attname = 'pg_class'::text)
        Filter: (attname ~~ 'pg\_class'::text)
        Heap Fetches: 0
        Buffers: shared hit=24

LOG:  duration: 0.026 ms  plan:
Query Text: SELECT  COUNT(*)               FROM    pg_index i
    INNER JOIN pg_class c ON (c.oid=indrelid)
    WHERE   relname LIKE tabname
Aggregate  (cost=3.47..3.48 rows=1 width=8) (actual time=0.024..0.024 rows=1 loops=1)
  Buffers: shared hit=8
  ->  Nested Loop  (cost=0.42..3.47 rows=1 width=0) (…)
        Buffers: shared hit=8
        ->  Index Scan using pg_class_relname_nsp_index on pg_class c
                                                  (cost=0.28..2.29 rows=1 width=4)
                                            (actual time=0.017..0.018 rows=1 loops=1)
              Index Cond: (relname = 'pg_class'::text)
              Filter: (relname ~~ 'pg\_class'::text)
              Buffers: shared hit=6
        ->  Index Only Scan using pg_index_indrelid_index on pg_index i  (…)
              Index Cond: (indrelid = c.oid)
              Heap Fetches: 0
              Buffers: shared hit=2

LOG:  duration: 0.414 ms  plan:
Query Text: SELECT nbi                 FROM table_nb_indexes (tabname)
Function Scan on table_nb_indexes  (cost=0.25..0.26 rows=1 width=4)
                          (actual time=0.412..0.412 rows=1 loops=1)
  Buffers: shared hit=16

LOG:  duration: 1.046 ms  plan:
Query Text: SELECT * FROM table_nb_col_indexes ('pg\_class') ;
Function Scan on table_nb_col_indexes  (cost=0.25..0.26 rows=1 width=8)
                              (actual time=1.042..1.043 rows=1 loops=1)
  Buffers: shared hit=56

 nb_cols | nb_indexes
---------+------------
       0 |          3
```

Pour les procédures, il est possible de mettre en place cette trace avec `ALTER PROCEDURE … SET auto_explain.log_min_duration = 0`. Cela ne fonctionne pas pour les fonctions.

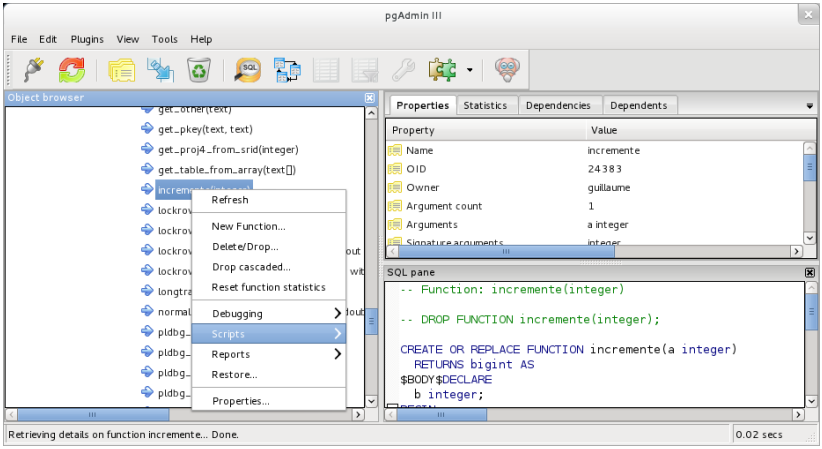
pgBadger est capable de lire les plans tracés par `auto_explain`, de les intégrer à son rapport et d'inclure un lien vers depesz.com[6] pour une version plus lisible.

---

### 1.10.5 PLDEBUGGER - UTILISATION

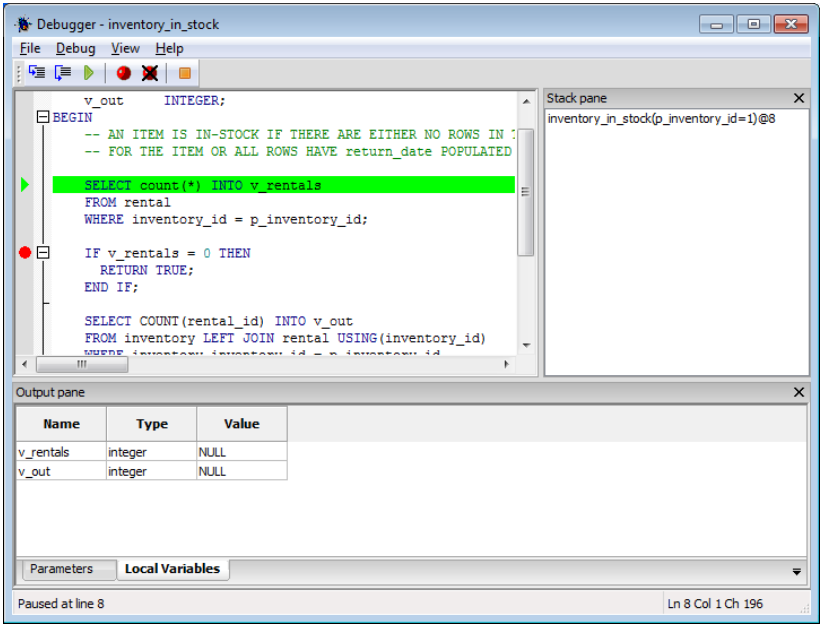
- Via pgAdmin

Le menu contextuel pour accéder au débuggage d'une fonction :




La fenêtre du débugger :

[6] https://explain.depesz.com/




---

### 1.10.6 LOG_FUNCTIONS

- Créé par Dalibo
- License BSD
- Compilation nécessaire

`log_functions` est un outil créé par Guillaume Lelarge au sein de Dalibo. Il est proposé sous license libre (BSD).

---

### 1.10.7 LOG_FUNCTIONS - COMPILATION

- Récupérer l'archive sur PGXN.org
- Décompresser l'archive puis : `make USE_PGXS=1 && make USE_PGXS=1 install`

Voici les étapes à réaliser pour compiler log_functions en prenant pour hypothèse que les sources de PostgreSQL sont disponibles dans le répertoire `/home/guillaume/postgresql-9.1.4` et qu'ils ont été préconfigurés avec la commande `./configure` :

- Se placer dans le répertoire contrib des sources de PostgreSQL :

```
$ cd /home/guillaume/postgresql-9.1.4/contrib
```

- Récupérer le dépôt git de log_functions :

```
$ git://github.com/gleu/log_functions.git

Cloning into 'log_functions'...
remote: Counting objects: 24, done.
remote: Compressing objects: 100% (15/15), done.
remote: Total 24 (delta 8), reused 24 (delta 8)
Receiving objects: 100% (24/24), 11.71 KiB, done.
Resolving deltas: 100% (8/8), done.
```

- Se placer dans le nouveau répertoire `log_functions` :

```
$ cd log_functions
```

- Compiler log_functions :

```
$ make
```

- Installer log_functions :

```
$ make install
```

L'installation copie le fichier `log_functions.o` dans le répertoire des bibliothèques partagées de PostgreSQL.

Si la version de PostgreSQL est supérieure ou égale à la 9.2, alors l'installation est plus simple et les sources de PostgreSQL ne sont plus nécessaires.

Téléchargement de log_functions :

```
wget http://api.pgxn.org/dist/log_functions/1.0.0/log_functions-1.0.0.zip
```

puis décompression et installation de l'extension :

```
unzip log_functions-1.0.0.zip
cd log_functions-1.0.0/
make USE_PGXS=1 && make USE_PGXS=1 install
```

L'installation copie aussi le fichier `log_functions.so` dans le répertoire des bibliothèques partagées de PostgreSQL.

---

### 1.10.8 LOG_FUNCTIONS - ACTIVATION

- Permanente
  - `shared_preload_libraries = 'log_functions'`
  - Redémarrage de PostgreSQL
- Au cas par cas
  - `LOAD 'log_functions'`

Le module log_functions est activable de deux façons.

La première consiste à demander à PostgreSQL de le charger au démarrage. Pour cela, il faut configurer la variable `shared_preload_libraries`, puis redémarrer PostgreSQL pour que le changement soit pris en compte.

La deuxième manière de l'activer est de l'activer seulement au moment où son utilisation s'avère nécessaire. Il faut utiliser pour cela la commande `LOAD` en précisant le module à charger.

La première méthode a un coût en terme de performances car le module s'exécute à chaque exécution d'une procédure stockée écrite en PL/pgSQL. La deuxième méthode rend l'utilisation du profiler un peu plus complexe. Le choix est donc laissé à l'administrateur.

---

### 1.10.9 LOG_FUNCTIONS - CONFIGURATION

- 5 paramètres en tout
- À configurer
  - dans Postgresql.conf
  - ou avec `SET`

Les informations de profilage récupérées par log_functions sont envoyées dans les traces de PostgreSQL. Comme cela va générer plus d'écriture, et donc plus de lenteurs, il est possible de configurer chaque trace.

La configuration se fait soit dans le fichier `postgresql.conf` soit avec l'instruction `SET`.

Voici la liste des paramètres et leur utilité :

- `log_functions.log_declare`, à mettre à true pour tracer le moment où PL/pgSQL exécute la partie `DECLARE` d'une procédure stockée ;
- `log_functions.log_function_begin`, à mettre à true pour tracer le moment où PL/pgSQL exécute la partie `BEGIN` d'une procédure stockée ;
- `log_functions.log_function_end`, à mettre à true pour tracer le moment où PL/pgSQL exécute la partie `END` d'une procédure stockée ;
- `log_functions.log_statement_begin`, à mettre à true pour tracer le moment où PL/pgSQL commence l'exécution d'une instruction dans une procédure stockée ;
- `log_functions.log_statement_end`, à mettre à true pour tracer le moment où PL/pgSQL termine l'exécution d'une instruction dans une procédure stockée.

Par défaut, seuls `log_statement_begin` et `log_statement_end` sont à false pour éviter la génération de traces trop importantes.

---

### 1.10.10 LOG_FUNCTIONS - UTILISATION

- Exécuter des procédures stockées en PL/pgSQL
- Lire les journaux applicatifs
  - `grep` très utile

Voici un exemple d'utilisation de cet outil :

```
b2# SELECT incremente(4);
 incremente
------------
          5
(1 row)

b2# LOAD 'log_functions';
LOAD
b2# SET client_min_messages TO log;
LOG:  duration: 0.136 ms  statement: set client_min_messages to log;
SET
b2# SELECT incremente(4);
LOG:  log_functions, DECLARE, incremente
LOG:  log_functions, BEGIN, incremente
CONTEXT:  PL/pgSQL function "incremente" during function entry
LOG:  valeur de b : 5
LOG:  log_functions, END, incremente
CONTEXT:  PL/pgSQL function "incremente" during function exit
LOG:  duration: 118.332 ms  statement: select incremente(4);
 incremente
------------
```

```
          5
(1 row)
```

---

## 1.11 CONCLUSION

- PL/pgSQL est un langage puissant
- Seul inconvénient
  - sa lenteur par rapport à d'autres PL comme PL/perl ou C
  - PL/perl est très efficace pour les traitements de chaîne notamment
- Permet néanmoins de traiter la plupart des cas, de façon simple et efficace

---

### 1.11.1 POUR ALLER PLUS LOIN

- Documentation officielle
  - « Chapitre 40. PL/pgSQL - Langage de procédures SQL »

Quelques liens utiles dans la documentation de PostgreSQL :

- Chapitre 40. PL/pgSQL - Langage de procédures SQL[7]
- Annexe A. Codes d'erreurs de PostgreSQL[8]

---

### 1.11.2 QUESTIONS

N'hésitez pas, c'est le moment !

---

[7] https://docs.postgresql.fr/current/plpgsql.html
[8] https://docs.postgresql.fr/current/errcodes-appendix.html

## 1.12 TRAVAUX PRATIQUES

### TP2.1

Ré-écrire la fonction de division pour tracer le problème de division par zéro (vous pouvez aussi utiliser les exceptions).

---

### TP2.2

Tracer dans une table toutes les modifications du champ `nombre` dans `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.

---

### TP2.3

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.

---

### TP2.4

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.

---

### TP2.5

Ré-écrire la fonction `nb_bouteilles` du TP précédent de façon à ce qu'elle prenne désormais en paramètre d'entrée une liste variable d'années à traiter.

## 1.13 TRAVAUX PRATIQUES (SOLUTIONS)

**TP2.1** Solution :

```
CREATE OR REPLACE FUNCTION division(arg1 integer, arg2 integer)
RETURNS float4 AS
$BODY$
  BEGIN
    RETURN arg1::float4/arg2::float4;
    EXCEPTION WHEN OTHERS THEN
      -- attention, division par zéro
      RAISE LOG 'attention, [%]: %', SQLSTATE, SQLERRM;
      RETURN 'NaN';
  END $BODY$
LANGUAGE 'plpgsql' VOLATILE;
```

Requêtage :

```
cave=# SET client_min_messages TO log;
SET
cave=# SELECT division(1,5);
 division
----------
      0.2
(1 ligne)

cave=# SELECT division(1,0);
LOG:  attention, [22012]: division par zéro
 division
----------
      NaN
(1 ligne)
```

---

**TP2.2** Solution :

1.

La table de log :

```
CREATE TABLE log_stock (
id serial,
utilisateur text,
dateheure timestamp,
operation char(1),
vin_id integer,
contenant_id integer,
annee integer,
```

```
  anciennevaleur integer,
  nouvellevaleur integer);
```

### La fonction trigger :

```
CREATE OR REPLACE FUNCTION log_stock_nombre()
  RETURNS TRIGGER AS
$BODY$
  DECLARE
    v_requete text;
    v_operation char(1);
    v_vinid integer;
    v_contenantid integer;
    v_annee integer;
    v_anciennevaleur integer;
    v_nouvellevaleur integer;
    v_atracer boolean := false;
  BEGIN

    -- ce test a pour but de vérifier que le contenu de nombre a bien changé
    -- c'est forcément le cas dans une insertion et dans une suppression
    -- mais il faut tester dans le cas d'une mise à jour en se méfiant
    -- des valeurs NULL
    v_operation := substr(TG_OP, 1, 1);
    IF TG_OP = 'INSERT'
    THEN
      -- cas de l'insertion
      v_atracer := true;
      v_vinid := NEW.vin_id;
      v_contenantid := NEW.contenant_id;
      v_annee := NEW.annee;
      v_anciennevaleur := NULL;
      v_nouvellevaleur := NEW.nombre;
    ELSEIF TG_OP = 'UPDATE'
    THEN
      -- cas de la mise à jour
      v_atracer := OLD.nombre != NEW.nombre;
      v_vinid := NEW.vin_id;
      v_contenantid := NEW.contenant_id;
      v_annee := NEW.annee;
      v_anciennevaleur := OLD.nombre;
      v_nouvellevaleur := NEW.nombre;
    ELSEIF TG_OP = 'DELETE'
    THEN
      -- cas de la suppression
      v_atracer := true;
      v_vinid := OLD.vin_id;
```

```
    v_contenantid := OLD.contenant_id;
    v_annee := NEW.annee;
    v_anciennevaleur := OLD.nombre;
    v_nouvellevaleur := NULL;
  END IF;

  IF v_atracer
  THEN
    INSERT INTO log_stock
     (utilisateur, dateheure, operation, vin_id, contenant_id,
      annee, anciennevaleur, nouvellevaleur)
    VALUES
     (current_user, now(), v_operation, v_vinid, v_contenantid,
      v_annee, v_anciennevaleur, v_nouvellevaleur);
  END IF;

  RETURN NEW;

END $BODY$
LANGUAGE 'plpgsql' VOLATILE;
```

Le trigger :

```
CREATE TRIGGER log_stock_nombre_trig
AFTER INSERT OR UPDATE OR DELETE
ON stock
FOR EACH ROW
EXECUTE PROCEDURE log_stock_nombre();
```

2.

On commence par supprimer le trigger :

```
DROP TRIGGER log_stock_nombre_trig ON stock;
```

La fonction trigger :

```
CREATE OR REPLACE FUNCTION log_stock_nombre()
  RETURNS TRIGGER AS
$BODY$
  DECLARE
    v_requete text;
    v_operation char(1);
    v_vinid integer;
    v_contenantid integer;
    v_annee integer;
    v_anciennevaleur integer;
    v_nouvellevaleur integer;
```

```
  v_atracer boolean := false;
BEGIN

  v_operation := substr(TG_OP, 1, 1);
  IF TG_OP = 'INSERT'
  THEN
    -- cas de l'insertion
    v_atracer := true;
    v_vinid := NEW.vin_id;
    v_contenantid := NEW.contenant_id;
    v_annee := NEW.annee;
    v_anciennevaleur := NULL;
    v_nouvellevaleur := NEW.nombre;
  ELSEIF TG_OP = 'UPDATE'
  THEN
    -- cas de la mise à jour
    v_atracer := OLD.nombre != NEW.nombre;
    v_vinid := NEW.vin_id;
    v_contenantid := NEW.contenant_id;
    v_annee := NEW.annee;
    v_anciennevaleur := OLD.nombre;
    v_nouvellevaleur := NEW.nombre;
  END IF;

  IF v_atracer
  THEN
    INSERT INTO log_stock
     (utilisateur, dateheure, operation, vin_id, contenant_id,
      anciennevaleur, nouvellevaleur)
    VALUES
     (current_user, now(), v_operation, v_vinid, v_contenantid,
      v_anciennevaleur, v_nouvellevaleur);
  END IF;

  RETURN NEW;

END $BODY$
  LANGUAGE 'plpgsql' VOLATILE;
```

Le trigger :

```
CREATE TRIGGER trace_nombre_de_stock
AFTER INSERT OR UPDATE
ON stock
FOR EACH ROW
EXECUTE PROCEDURE log_stock_nombre();
```

La deuxième fonction trigger :

```
CREATE OR REPLACE FUNCTION empeche_suppr_stock()
  RETURNS TRIGGER AS
$BODY$
  BEGIN

    IF TG_OP = 'DELETE'
    THEN
      RAISE WARNING 'Tentative de suppression du stock (%, %, %)',
                    OLD.vin_id, OLD.contenant_id, OLD.annee;
      RETURN NULL;
    ELSE
      RETURN NEW;
    END IF;

  END $BODY$
    LANGUAGE 'plpgsql' VOLATILE;
```

Le deuxième trigger :

```
  CREATE TRIGGER empeche_suppr_stock_trig
  BEFORE DELETE
  ON stock
  FOR EACH ROW
  EXECUTE PROCEDURE empeche_suppr_stock();
```

3.

La fonction trigger :

```
CREATE OR REPLACE FUNCTION log_stock_nombre()
  RETURNS TRIGGER AS
$BODY$
  DECLARE
    v_requete text;
    v_operation char(1);
    v_vinid integer;
    v_contenantid integer;
    v_annee integer;
    v_anciennevaleur integer;
    v_nouvellevaleur integer;
    v_atracer boolean := false;
  BEGIN

    v_operation := substr(TG_OP, 1, 1);
    IF TG_OP = 'INSERT'
    THEN
```

```
    -- cas de l'insertion
    v_atracer := true;
    v_vinid := NEW.vin_id;
    v_contenantid := NEW.contenant_id;
    v_annee := NEW.annee;
    v_anciennevaleur := NULL;
    v_nouvellevaleur := NEW.nombre;
  ELSEIF TG_OP = 'UPDATE'
  THEN
    -- cas de la mise à jour
    v_atracer := OLD.nombre != NEW.nombre;
    v_vinid := NEW.vin_id;
    v_contenantid := NEW.contenant_id;
    v_annee := NEW.annee;
    v_anciennevaleur := OLD.nombre;
    v_nouvellevaleur := NEW.nombre;
  END IF;

  IF v_nouvellevaleur < 1
  THEN
    RAISE WARNING 'Il ne reste plus que % bouteilles dans le stock (%, %, %)',
                  v_nouvellevaleur, OLD.vin_id, OLD.contenant_id, OLD.annee;
  ELSEIF v_nouvellevaleur < 5
  THEN
    RAISE LOG 'Il ne reste plus que % bouteilles dans le stock (%, %, %)',
              v_nouvellevaleur, OLD.vin_id, OLD.contenant_id, OLD.annee;
  END IF;

  IF v_atracer
  THEN
    INSERT INTO log_stock
     (utilisateur, dateheure, operation, vin_id, contenant_id,
      annee, anciennevaleur, nouvellevaleur)
    VALUES
     (current_user, now(), v_operation, v_vinid, v_contenantid,
      v_annee, v_anciennevaleur, v_nouvellevaleur);
  END IF;

  RETURN NEW;

END $BODY$
  LANGUAGE 'plpgsql' VOLATILE;
```

Requêtage :

Faire des INSERT, DELETE, UPDATE pour jouer avec.

---

**TP2.3** Solution :

```
 CREATE ROLE admin;
 ALTER TABLE log_stock OWNER TO admin;
 ALTER TABLE log_stock_id_seq OWNER TO admin;
 REVOKE ALL ON TABLE log_stock FROM public;

 cave=> insert into stock (vin_id, contenant_id, annee, nombre)
        values (3,1,2020,10);
ERROR:  permission denied for relation log_stock
CONTEXT:  SQL statement "INSERT INTO log_stock
       (utilisateur, dateheure, operation, vin_id, contenant_id,
        annee, anciennevaleur, nouvellevaleur)
      VALUES
       (current_user, now(), v_operation, v_vinid, v_contenantid,
        v_annee, v_anciennevaleur, v_nouvellevaleur)"
PL/pgSQL function log_stock_nombre() line 45 at SQL statement

 ALTER FUNCTION log_stock_nombre() OWNER TO admin;
 ALTER FUNCTION log_stock_nombre() SECURITY DEFINER;

 cave=> insert into stock (vin_id, contenant_id, annee, nombre)
        values (3,1,2020,10);
  INSERT 0 1
```

Que constatez-vous dans `log_stock` ? (un petit indice : regardez l'utilisateur)

---

**TP2.4** Solution :

```
CREATE OR REPLACE FUNCTION verif_nombre(maxnombre integer)
  RETURNS integer AS
$BODY$
  DECLARE
    v_curseur refcursor;
    v_resultat stock%ROWTYPE;
    v_index integer;
  BEGIN

    v_index := 0;
    OPEN v_curseur FOR SELECT * FROM stock WHERE nombre > maxnombre;
    LOOP
      FETCH v_curseur INTO v_resultat;
      IF NOT FOUND THEN
        EXIT;
```

```
      END IF;
      v_index := v_index + 1;
      RAISE NOTICE 'nombre de (%, %) : % (supérieur à %)',
        v_resultat.vin_id, v_resultat.contenant_id, v_resultat.nombre, maxnombre;
    END LOOP;

  RETURN v_index;

END $BODY$
  LANGUAGE 'plpgsql' VOLATILE;
```

Requêtage:

```
  SELECT verif_nombre(16);
  INFO:  nombre de (6535, 3) : 17 (supérieur à 16)
  INFO:  nombre de (6538, 3) : 17 (supérieur à 16)
  INFO:  nombre de (6541, 3) : 17 (supérieur à 16)
  [...]
  INFO:  nombre de (6692, 3) : 18 (supérieur à 16)
  INFO:  nombre de (6699, 3) : 17 (supérieur à 16)
   verif_nombre
  --------------
         107935
  (1 ligne)
```

---

## TP2.5

```
CREATE OR REPLACE FUNCTION
  nb_bouteilles(v_typevin text, VARIADIC v_annees integer[])
RETURNS SETOF record
AS $BODY$
  DECLARE
    resultat record;
    i integer;
  BEGIN
    FOREACH i IN ARRAY v_annees
    LOOP
      SELECT INTO resultat i, nb_bouteilles(v_typevin, i);
      RETURN NEXT resultat;
    END LOOP;
    RETURN;
  END
$BODY$
LANGUAGE plpgsql;
```

Exécution:

```
-- ancienne fonction
cave=# SELECT * FROM nb_bouteilles('blanc', 1990, 1995)
  AS (annee integer, nb integer);
 annee |  nb
-------+------
  1990 | 5608
  1991 | 5642
  1992 | 5621
  1993 | 5581
  1994 | 5614
  1995 | 5599
(6 lignes)

cave=# SELECT * FROM nb_bouteilles('blanc', 1990, 1992, 1994)
  AS (annee integer, nb integer);
 annee |  nb
-------+------
  1990 | 5608
  1992 | 5621
  1994 | 5614
(3 lignes)

cave=# SELECT * FROM nb_bouteilles('blanc', 1993, 1991)
  AS (annee integer, nb integer);
 annee |  nb
-------+------
  1993 | 5581
  1991 | 5642
(2 lignes)
```

## NOTES

## NOTES

# NOS AUTRES PUBLICATIONS

## FORMATIONS

- **DBA1 : Administration PostgreSQL**
  https://dali.bo/dba1
- **DBA2 : Administration PostgreSQL avancé**
  https://dali.bo/dba2
- **DBA3 : Sauvegarde et réplication avec PostgreSQL**
  https://dali.bo/dba3
- **DEVPG : Développer avec PostgreSQL**
  https://dali.bo/devpg
- **PERF1 : PostgreSQL Performances**
  https://dali.bo/perf1
- **PERF2 : Indexation et SQL avancés**
  https://dali.bo/perf2
- **MIGORPG : Migrer d'Oracle à PostgreSQL**
  https://dali.bo/migorpg
- **HAPAT : Haute disponibilité avec PostgreSQL**
  https://dali.bo/hapat

## LIVRES BLANCS

- **Migrer d'Oracle à PostgreSQL**
- **Industrialiser PostgreSQL**
- **Bonnes pratiques de modélisation avec PostgreSQL**
- **Bonnes pratiques de développement avec PostgreSQL**

## TÉLÉCHARGEMENT GRATUIT

Les versions électroniques de nos publications sont disponibles gratuitement sous licence open-source ou sous licence Creative Commons. Contactez-nous à l'adresse contact@dalibo.com pour plus d'information.

## DALIBO, L'EXPERTISE POSTGRESQL

Depuis 2005, DALIBO met à la disposition de ses clients son savoir-faire dans le domaine des bases de données et propose des services de conseil, de formation et de support aux entreprises et aux institutionnels.

En parallèle de son activité commerciale, DALIBO contribue aux développements de la communauté PostgreSQL et participe activement à l'animation de la communauté francophone de PostgreSQL. La société est également à l'origine de nombreux outils libres de supervision, de migration, de sauvegarde et d'optimisation.

Le succès de PostgreSQL démontre que la transparence, l'ouverture et l'auto-gestion sont à la fois une source d'innovation et un gage de pérennité. DALIBO a intégré ces principes dans son ADN en optant pour le statut de SCOP : la société est contrôlée à 100 % par ses salariés, les décisions sont prises collectivement et les bénéfices sont partagés à parts égales.