Module P1

PL/pgSQL : Les bases

22.09

Dalibo SCOP

https://dalibo.com/formations

PL/pgSQL : Les bases

Module P1

TITRE : PL/pgSQL : Les bases
SOUS-TITRE : Module P1

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,

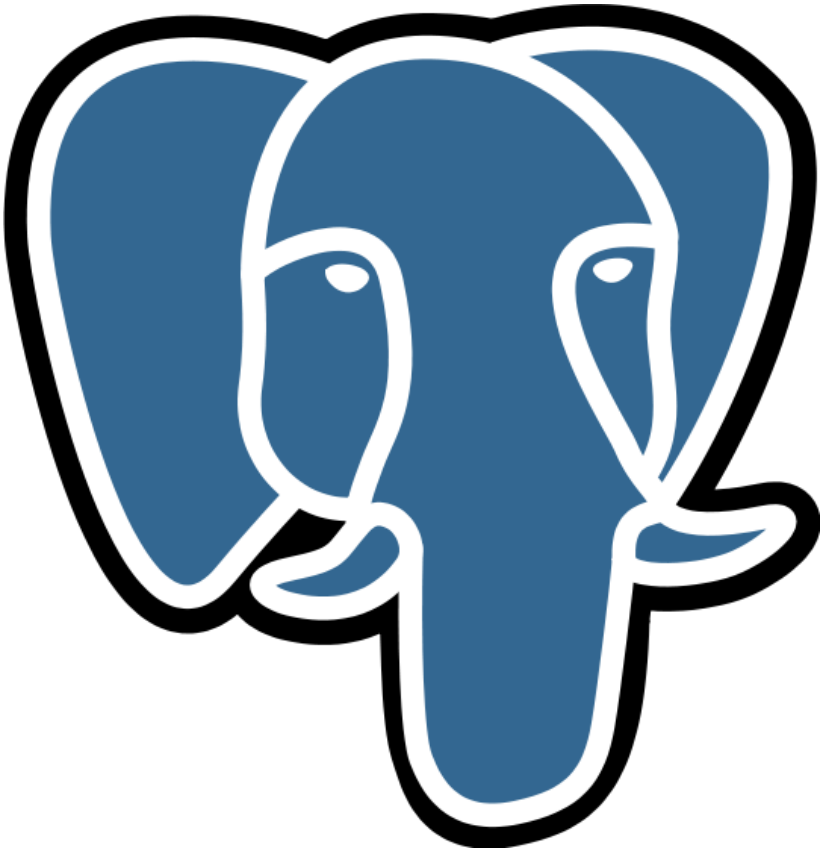
Nos formations PostgreSQL sont issues de nombreuses années d'études, d'expérience de terrain et de passion pour les logiciels libres. Pour Dalibo, l'utilisation de PostgreSQL n'est pas une marque d'opportunisme commercial, mais l'expression d'un engagement de longue date. Le choix de l'Open Source est 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 : LES BASES



1.1 PRÉAMBULE

- Vous apprendrez :
  - à choisir si vous voulez écrire du PL
  - à choisir votre langage PL
  - les principes généraux des langages PL autres que PL/pgSQL
  - les bases de PL/pgSQL

Ce module présente la programmation PL/pgSQL. Il commence par décrire les routines

stockées et les différents langages disponibles. Puis il aborde les bases du langage PL/pgSQL, autrement dit :

- comment installer PL/pgSQL dans une base PostgreSQL ;
- comment créer un squelette de fonction ;
- comment déclarer des variables ;
- comment utiliser les instructions de base du langage ;
- comment créer et manipuler des structures ;
- comment passer une valeur de retour de la fonction à l'appelant.

1.1.1 AU MENU

- Présentation du PL et des principes
- Présentations de PL/pgSQL et des autres langages PL
- Installation d'un langage PL
- Détails sur PL/pgSQL

1.1.2 OBJECTIFS

- Comprendre les cas d'utilisation d'une routine PL/pgSQL
- Choisir son langage PL en connaissance de cause
- Comprendre la différence entre PL/pgSQL et les autres langages PL
- Écrire une routine simple en PL/pgSQL
  - et même plus complexe

1.2 INTRODUCTION

1.2.1 QU'EST-CE QU'UN PL ?

- PL = *Procedural Language*
- 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.

1.2.2 QUELS LANGAGES PL SONT DISPONIBLES ?

- Installé par défaut :
  - PL/pgSQL
- Intégrés au projet :
  - PL/Perl
  - PL/Python
  - PL/Tcl
- Extensions tierces :
  - PL/java, PL/R, PL/v8 (Javascript), PL/sh ...
  - extensible à volonté

Les quatre langages PL supportés nativement (en plus du C et du SQL bien sûr) sont décrits en détail dans la documentation officielle :

- PL/PgSQL[2] est intégré par défaut dans toute nouvelle base (de par sa présence dans la base modèle **template1**) ;
- PL/Tcl[3] (existe en version *trusted* et *untrusted*) ;
- PL/Perl[4] (existe en version *trusted* et *untrusted*) ;
- PL/Python[5] (uniquement en version *untrusted*).

[2] https://docs.postgresql.fr/current/plpgsql.html
[3] https://docs.postgresql.fr/current/pltcl.html
[4] https://docs.postgresql.fr/current/plperl.html
[5] https://docs.postgresql.fr/current/plpython.html

D'autres langages PL sont accessibles en tant qu'extensions tierces. Les plus stables sont mentionnés dans la documentation[6] , comme PL/Java[7] ou PL/R[8] . Ils réclament généralement d'installer les bibliothèques du langage sur le serveur.

Une liste plus large est par ailleurs disponible sur le wiki PostgreSQL[9] , Il en ressort qu'au moins 16 langages sont disponibles, dont 10 installables en production. De plus, il est possible d'en ajouter d'autres, comme décrit dans la documentation[10] .

1.2.3 LANGAGES TRUSTED VS UNTRUSTED

- *Trusted* = langage de confiance :
  - ne permet que l'accès à la base de données
  - donc pas aux systèmes de fichiers, aux sockets réseaux, etc.
  - SQL, PL/pgSQL, PL/Perl, PL/Tcl
- *Untrusted*:
  - PL/Python, C…
  - PL/TclU, PL/PerlU

Les langages de confiance ne peuvent accéder qu'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 utilisables pour compromettre le système. PL/pgSQL est l'exemple typique. Mais de ce fait, ils offrent moins de possibilités que les autres langages.

Seuls les superutilisateurs peuvent créer une routine dans un langage *untrusted*. Par contre, ils peuvent ensuite donner les droits d'exécution à ces routines aux autres rôles dans la base :

```
GRANT EXECUTE ON FUNCTION nom_fonction TO un_role ;
```

---

[6] https://docs.postgresql.fr/current/external-pl.html
[7] https://tada.github.io/pljava/
[8] https://github.com/postgres-plr/plr
[9] https://wiki.postgresql.org/wiki/PL_Matrix
[10] https://docs.postgresql.fr/current/plhandler.html

1.2.4 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

La question se pose souvent de placer la logique applicative du côté de la base, dans un langage PL, ou des clients. Il peut y avoir de nombreuses raisons en faveur de la première option. 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. Les avantages sont :

Centralisation du code :

Si plusieurs applications ont potentiellement besoin d'opérer un même traitement, à fortiori dans des langages différents, porter cette logique dans la base réduit d'autant les risques de *bugs* et facilite la maintenance.

Une règle peut être que tout ce qui a trait à l'intégrité des données devrait être exécuté au niveau de la base.

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 messages réseaux dus à l'exécution de nombreux ordres SQL consécutifs. L'impact de la latence due au trafic réseau de la base au client est souvent sous-estimée.

Les langages PL permettent aussi d'accéder à leurs bibliothèques spécifiques (extrêmement nombreuses en python ou perl, entre autres).

Une fonction en PL peut également servir à l'indexation des données. Cela est impossible si elle se calcule sur une autre machine.

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, là aussi utilisables dans la base.

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.

1.2.5 INTÉRÊTS DE PL/PGSQL EN PARTICULIER

- Inspiré de l'ADA, proche du Pascal
- Ajout de structures de contrôle au langage SQL
- **Dédié au traitement des données et au SQL**
- Peut effectuer des traitements complexes
- Hérite de tous les types, fonctions et opérateurs définis par les utilisateurs
- *Trusted*
- Facile à utiliser

Le langage étant assez ancien, proche du Pascal et de l'ADA, sa syntaxe ne choquera personne. Elle est d'ailleurs très proche de celle du PLSQL d'Oracle.

Le PL/pgSQL 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 procédural de SQL.

PL/pgSQL étant intégré à PostgreSQL, il hérite de tous les types déclarés dans le moteur, même ceux rajoutés par l'utilisateur. Il peut les manipuler de façon transparente.

PL/pgSQL est *trusted*. Tous les utilisateurs peuvent donc créer des routines 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, et il dispose d'une gestion des erreurs évoluée (gestion d'exceptions).

1.2.6 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 :
  - pas spécialisés dans le traitement de requêtes
  - types différents
  - interpréteur séparé

Les langages PL « autres », comme PL/perl[11] 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, possèdent des structures avancées comme des tables de hachage, permettent l'utilisation de variables statiques pour maintenir des caches, voire, pour leur version *untrusted*, peuvent effectuer des appels systèmes. Dans ce cas, il devient possible d'appeler un service web 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[12] . 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, et avec une latence dans le transfert des données.

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[13] , 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.

Mais attention : toute erreur dans le 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 : ils ne sont pas spécialement conçus 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 ; de même en C. L'accès aux données est assez fastidieux au niveau syntaxique, comparé à PL/pgSQL.

[11] https://docs.postgresql.fr/current/plperl-builtins.html
[12] https://github.com/postgres-plr/plr/blob/master/userguide.md
[13] https://docs.postgresql.fr/current/xfunc-c.html

Un autre problème des langages PL (autre que C et PL/pgSQL), 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 aux 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).

1.2.7 ROUTINES / PROCÉDURES STOCKÉES / FONCTIONS

- **Procédure** stockée
  - pas de retour
  - contrôle transactionnel : `COMMIT` / `ROLLBACK`
  - PostgreSQL 11 ou +
- **Fonction**
  - peut renvoyer des données (même des lignes)
  - utilisable dans un `SELECT`
  - peut être de type `TRIGGER`, agrégat, fenêtrage
- **Routine**
  - procédure ou fonction

Les programmes écrits à l'aide des langages PL sont habituellement enregistrés sous forme de « routines » :

- procédures ;
- fonctions ;
- fonctions *trigger* ;
- fonctions d'agrégat ;
- fonctions de fenêtrage (*window functions*).

Le code source de ces objets est stocké dans la table `pg_proc` du catalogue.

Les procédures, apparues avec PostgreSQL 11, sont très similaires aux fonctions. Les principales différences entre les deux sont :

- Les fonctions doivent déclarer des arguments en sortie (`RETURNS` ou arguments `OUT`). Elles peuvent renvoyer n'importe quel type de donnée, ou des ensembles de lignes. Il est possible d'utiliser `void` pour une fonction sans argument de sortie ; c'était d'ailleurs la méthode utilisée pour émuler le comportement d'une procédure avant leur introduction avec PostgreSQL 11. Les procédures n'ont pas de code retour (on

peut cependant utiliser des paramètres `OUT` ou `INOUT` /* selon version, voir plus bas */).

- Les procédures offrent le support du contrôle transactionnel, c'est-à-dire la capacité de valider (`COMMIT`) ou annuler (`ROLLBACK`) les modifications effectuées jusqu'à ce point par la procédure. L'intégralité d'une fonction s'effectue dans la transaction appelante.
- Les procédures sont appelées exclusivement par la commande SQL `CALL` ; les fonctions peuvent être appelées dans la plupart des ordres DML/DQL (notamment `SELECT`), mais pas par `CALL`.
- Les fonctions peuvent être déclarées de telle manière qu'elles peuvent être utilisées dans des rôles spécifiques (`TRIGGER`, agrégat ou fonction de fenêtrage).

1.3 INSTALLATION

1.3.1 INSTALLATION DES BINAIRES NÉCESSAIRES

- SQL, C et PL/pgSQL
  - compilés et installés par défaut
- Paquets du PGDG pour la plupart des langages :

```
yum|dnf install postgresql14-plperl
apt     install postgresql-plpython3-14
```

- Autres langages :
  - à compiler soi-même

Pour savoir si PL/Perl ou PL/Python a été compilé, on peut 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'
```

Si besoin, les emplacements exacts d'installation des bibliothèques peuvent être récupérés à l'aide des options `--libdir` et `--pkglibdir` de `pg_config`.

Cependant, dans les paquets fournis par le PGDG, il faudra installer explicitement le paquet dédié à `plperl` pour la version majeure de PostgreSQL concernée. Pour PostgreSQL 14, les paquets sont `postgresql14-plperl` (depuis yum.postgresql.org) ou `postgresql-plperl-14` (depuis apt.postgresql.org).

Ainsi, la bibliothèque `plperl.so` que l'on trouvera dans ces répertoires contiendra les fonctions qui permettent l'utilisation du langage PL/Perl. Elle est chargée par le moteur à la première utilisation d'une procédure utilisant ce langage.

De même pour Python 3 (paquets `postgresql14-plpython3` ou `postgresql-plython3-14`).

La plupart des langages intéressants sont disponibles sous forme de paquets. Des versions très récentes, ou des langages plus exotiques, peuvent nécessiter une compilation de l'extension.

1.3.2 ACTIVER/DÉSACTIVER UN LANGAGE

- Activer un langage passe par la création de l'extension :
  `CREATE EXTENSION plperl ;`
- Supprimer l'extension désactive le langage :
  `DROP EXTENSION plperl ;`

Le langage est activé uniquement dans la base dans laquelle la commande est lancée. S'il faut l'activer sur plusieurs bases, il sera nécessaire d'exécuter cet ordre SQL sur les différentes bases ciblées.

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

1.3.3 LANGAGE DÉJÀ INSTALLÉ ?

- Interroger le catalogue système `pg_language`
  - ou `\dx` avec `psql`
- Une ligne par langage installé
- *Trusted* ou *untrusted* ?

Voici un exemple d'interrogation de `pg_language` :

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

 lanname | lanpltrusted
---------+--------------
 plpgsql | t
```

Si un langage est *trusted*, tous les utilisateurs peuvent créer des procédures dans ce langage. Sinon seuls les superutilisateurs le peuvent. Il existe par exemple deux variantes de PL/Perl : PL/Perl et PL/PerlU. La seconde est la variante *untrusted* et est un Perl « complet ». La version *trusted* n'a pas le droit d'ouvrir des fichiers, des sockets, ou autres appels systèmes qui seraient dangereux.

SQL, PL/pgSQL, PL/Tcl, PL/Perl (mais pas PL/Python) sont *trusted*.

C, PL/TclU, PL/PerlU, et PL/PythonU (et les variantes spécifiques aux versions PL/Python2U et PL/Python3U) sont *untrusted*.

Les langages PL sont généralement installés par le biais d'extensions :

```
base=# \dx
                        Liste des extensions installées
     Nom     | Version |   Schéma   |            Description
-------------+---------+------------+-----------------------------------
 plpgsql     | 1.0     | pg_catalog | PL/pgSQL procedural language

                        ____________________________________________
```

1.4 EXEMPLES DE FONCTIONS & PROCÉDURES

1.4.1 FONCTION PL/PGSQL SIMPLE

Une fonction simple en PL/pgSQL :

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

```
SELECT addition (1,2);

 addition
----------
        3
```

1.4.2 EXEMPLE DE FONCTION SQL

Même fonction en SQL pur :

```
CREATE FUNCTION addition (entier1 integer, entier2 integer)
RETURNS integer
LANGUAGE sql
IMMUTABLE
AS '
    SELECT entier1 + entier2;
' ;
```

- Intérêt : planification !
- Syntaxe allégée possible en v14

Les fonctions simples peuvent être écrites en SQL pur. La syntaxe est plus claire, mais bien plus limitée qu'en PL/pgSQL (ni boucles, ni conditions, ni exceptions notamment).

À partir de PostgreSQL 14, il est possible de se passer des guillemets encadrants, pour les fonctions SQL uniquement. La même fonction devient donc :

```
CREATE OR REPLACE FUNCTION addition (entier1 integer, entier2 integer)
RETURNS integer
```

```
LANGUAGE sql
IMMUTABLE
RETURN entier1 + entier2 ;
```

Cette nouvelle écriture respecte mieux le standard SQL. Surtout, elle autorise un *parsing* et une vérification des objets impliqués dès la déclaration, et non à l'utilisation. Les dépendances entre fonctions et objets utilisés sont aussi mieux tracées.

L'avantage principal des fonctions en pur SQL est, si elles sont assez simples, leur intégration lors de la réécriture interne de la requête (*inlining*) : elles ne sont donc pas pour l'optimiseur des « boîtes noires ». À l'inverse, l'optimiseur ne sait rien du contenu d'une fonction PL/pgSQL.

Dans l'exemple suivant, la fonction sert de filtre à la requête. Comme elle est en pur SQL, elle permet d'utiliser l'index sur la colonne `date_embauche` de la table `employes_big` :

```
CREATE OR REPLACE function employe_eligible_prime_sql (service int, date_embauche date)
RETURNS boolean
LANGUAGE sql
AS $$
  SELECT ( service !=3 AND date_embauche < '2003-01-01')  ; -- ancien employé, sauf un service
$$ ;

EXPLAIN (ANALYZE) SELECT matricule, num_service, nom, prenom
FROM    employes_big
WHERE   employe_eligible_prime_sql (num_service, date_embauche) ;

                           QUERY PLAN
--------------------------------------------------------------------------------
 Index Scan using employes_big_date_embauche_idx on employes_big
       (cost=0.42..1.54 rows=1 width=22) (actual time=0.008..0.009 rows=1 loops=1)
   Index Cond: (date_embauche < '2003-01-01'::date)
   Filter: (num_service <> 3)
   Rows Removed by Filter: 1
 Planning Time: 0.102 ms
 Execution Time: 0.029 ms
```

Avec une version de la même fonction en PL/pgSQL, le planificateur ne voit pas le critère indexé. Il n'a pas d'autre choix que de lire toute la table et d'appeler la fonction pour chaque ligne, ce qui est bien sûr plus lent :

```
CREATE OR REPLACE function employe_eligible_prime_pl (service int, date_embauche date)
RETURNS boolean
LANGUAGE plpgsql AS $$
BEGIN
 RETURN ( service !=3 AND date_embauche < '2003-01-01') ;
```

```
END ;
$$ ;

EXPLAIN (ANALYZE) SELECT matricule, num_service, nom, prenom
FROM employes_big
WHERE  employe_eligible_prime_pl (num_service, date_embauche) ;

                           QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on employes_big   (cost=0.00..134407.90 rows=166338 width=22)
                            (actual time=0.069..269.121 rows=1 loops=1)
   Filter: employe_eligible_prime_pl(num_service, date_embauche)
   Rows Removed by Filter: 499014
 Planning Time: 0.038 ms
 Execution Time: 269.157 ms
```

Le wiki[14] décrit les conditions pour que l'*inlining* des fonctions SQL fonctionne : obligation d'un seul `SELECT`, interdiction de certains fonctionnalités...

1.4.3 EXEMPLE DE FONCTION PL/PGSQL UTILISANT LA BASE

```
CREATE OR REPLACE FUNCTION nb_lignes_table (sch text, tbl text)
 RETURNS bigint
 STABLE
AS '
DECLARE     n bigint ;
BEGIN
    SELECT n_live_tup
    INTO n
    FROM pg_stat_user_tables
    WHERE schemaname = sch AND relname = tbl ;
    RETURN n ;
END ; '
LANGUAGE plpgsql ;
```

Dans cet exemple, on récupère l'estimation du nombre de lignes actives d'une table passée en paramètres.

L'intérêt majeur du PL/pgSQL et du SQL sur les autres langages est la facilité d'accès aux données. Ici, un simple `SELECT <champ> INTO <variable>` suffit à récupérer une valeur depuis une table dans une variable.

```
SELECT nb_lignes_table ('public', 'pgbench_accounts');
```

[14] https://wiki.postgresql.org/wiki/Inlining_of_SQL_functions

```
 nb_lignes_table
-----------------
        10000000
```

1.4.4 EXEMPLE DE FONCTION PL/PERL COMPLEXE

- Permet d'insérer une facture associée à un client
- Si le client n'existe pas, une entrée est créée
- Utilisation fréquente de `spi_exec`

Voici l'exemple de la fonction :

```
CREATE OR REPLACE FUNCTION
    public.demo_insert_perl(nom_client text, titre_facture text)
 RETURNS integer
 LANGUAGE plperl
 STRICT
AS $function$
  use strict;
  my ($nom_client, $titre_facture)=@_;
  my $rv;
  my $id_facture;
  my $id_client;

  # Le client existe t'il ?
  $rv = spi_exec_query('SELECT id_client FROM mes_clients WHERE nom_client = '
    . quote_literal($nom_client)
  );
  # Sinon on le crée :
  if ($rv->{processed} == 0)
  {
    $rv = spi_exec_query('INSERT INTO mes_clients (nom_client) VALUES ('
        . quote_literal($nom_client) . ') RETURNING id_client'
    );
  }
  # Dans les deux cas, l'id client est dans $rv :
  $id_client=$rv->{rows}[0]->{id_client};

  # Insérons maintenant la facture
  $rv = spi_exec_query(
    'INSERT INTO mes_factures (titre_facture, id_client) VALUES ('
    . quote_literal($titre_facture) . ", $id_client ) RETURNING id_facture"
  );

  $id_facture = $rv->{rows}[0]->{id_facture};
```

```
  return $id_facture;
$function$ ;
```

Cette fonction n'est pas parfaite, elle ne protège pas de tout. Il est tout à fait possible d'avoir une insertion concurrente entre le `SELECT` et le `INSERT` par exemple.

Il est clair que l'accès aux données est malaisé en PL/Perl, comme dans la plupart des langages, puisqu'ils ne sont pas prévus spécifiquement pour cette tâche. Par contre, on dispose de toute la puissance de Perl pour les traitements de chaîne, les appels système…

PL/Perl, c'est :

- Perl, moins les fonctions pouvant accéder à autre chose qu'à PostgreSQL (il faut utiliser PL/PerlU pour passer outre cette limitation) ;
- un bloc de code anonyme appelé par PostgreSQL ;
- des fonctions d'accès à la base, `spi_*`

1.4.5 EXEMPLE DE FONCTION PL/PGSQL COMPLEXE

- Même fonction en PL/pgSQL que précédemment
- 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 maintenant 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;
$function$ ;
```

1.4.6 EXEMPLE DE PROCÉDURE

```
CREATE OR REPLACE PROCEDURE vide_tables (dry_run BOOLEAN)
AS '
BEGIN
    TRUNCATE TABLE pgbench_history ;
    TRUNCATE TABLE pgbench_accounts CASCADE ;
    TRUNCATE TABLE pgbench_tellers  CASCADE ;
    TRUNCATE TABLE pgbench_branches CASCADE ;
    IF dry_run THEN
        ROLLBACK ;
    END IF ;
END ;
 ' LANGUAGE plpgsql ;
```

Cette procédure tronque des tables de la base d'exemple **pgbench**, et annule si `dry_run` est vrai.

Les procédures sont récentes dans PostgreSQL (à partir de la version 11). Elles sont à utiliser quand on n'attend pas de résultat en retour. Surtout, elles permettent de gérer les transactions (`COMMIT`, `ROLLBACK`), ce qui ne peut se faire dans des fonctions, même si celles-ci peuvent modifier les données.

> Une procédure ne peut utiliser le contrôle transactionnel que si elle est appelée en dehors de toute transaction.

Comme pour les fonctions, il est possible d'utiliser le SQL pur dans les cas les plus simples,

sans contrôle transactionnel notamment :

```
CREATE OR REPLACE PROCEDURE vide_tables ()
AS '
    TRUNCATE TABLE pgbench_history ;
    TRUNCATE TABLE pgbench_accounts CASCADE ;
    TRUNCATE TABLE pgbench_tellers  CASCADE ;
    TRUNCATE TABLE pgbench_branches CASCADE ;
' LANGUAGE sql;
```

Toujours pour les procédures en SQL, il existe une variante sans guillemets, à partir de PostgreSQL 14, mais qui ne supporte pas tous les ordres. Comme pour les fonctions, l'intérêt est la prise en compte des dépendances entre objets et procédures.

```
CREATE OR REPLACE PROCEDURE vide_tables ()
BEGIN ATOMIC
    DELETE FROM pgbench_history ;
    DELETE FROM pgbench_accounts ;
    DELETE FROM pgbench_tellers ;
    DELETE FROM pgbench_branches ;
END ;
```

1.4.7 EXEMPLE DE BLOC ANONYME EN PL/PGSQL

- Bloc procédural anonyme en PL/pgSQL :

```
DO $$
DECLARE r record;
BEGIN
    FOR r IN (SELECT schemaname, relname
              FROM pg_stat_user_tables
              WHERE coalesce(last_analyze, last_autoanalyze) IS NULL
              ) LOOP
        RAISE NOTICE 'Analyze %.%', r.schemaname, r.relname ;
        EXECUTE 'ANALYZE ' || quote_ident(r.schemaname)
                           || '.' || quote_ident(r.relname) ;
    END LOOP;
END$$;
```

Les blocs anonymes sont utiles pour des petits scripts ponctuels qui nécessitent des boucles ou du conditionnel, voire du transactionnel, sans avoir à créer une fonction ou une procédure. Ils ne renvoient rien. Ils sont habituellement en PL/pgSQL mais tout langage procédural installé est possible.

L'exemple ci-dessus lance un `ANALYZE` sur toutes les tables où les statistiques n'ont pas été calculées d'après la vue système, et donne aussi un exemple de SQL dynamique. Le

résultat est par exemple :

```
NOTICE:  Analyze public.pgbench_history
NOTICE:  Analyze public.pgbench_tellers
NOTICE:  Analyze public.pgbench_accounts
NOTICE:  Analyze public.pgbench_branches
DO
Temps : 141,208 ms
```

(Pour ce genre de SQL dynamique, si l'on est sous `psql` , il est souvent plus pratique d'utiliser `\gexec`[15] .)

Noter que les ordres constituent une transaction unique, à moins de rajouter des `COMMIT` ou `ROLLBACK` explicitement (ce n'est autorisé qu'à partir de la version 11).

---

[15] https://docs.postgresql.fr/current/app-psql.html#R2-APP-PSQL-4

1.5 UTILISER UNE FONCTION OU UNE PROCÉDURE

1.5.1 INVOCATION D'UNE FONCTION OU PROCÉDURE

- Appeler une procédure : ordre spécifique `CALL`

```
CALL ma_procedure('arg1');
```

- Appeler une fonction : dans une requête

```
SELECT ma_fonction('arg1', 'arg2') ;

SELECT * FROM ma_fonction('arg1', 'arg2') ;

INSERT INTO matable
SELECT ma_fonction( champ1, champ2 )   FROM ma_table2 ;

CALL ma_procedure( mafonction() );

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

Demander l'exécution d'une procédure se fait en utilisant un ordre SQL spécifique : `CALL`[16] . Il suffit de fournir les paramètres. Il n'y a pas de code retour.

Les fonctions ne sont quant à elles pas directement compatibles avec la commande `CALL`, il faut les invoquer dans le contexte d'une commande SQL. Elles sont le plus couramment appelées depuis des commandes de type DML (`SELECT`, `INSERT`, etc.), mais on peut aussi les trouver dans d'autres commandes.

Voici quelques exemples :

- dans un `SELECT` (la fonction ne doit renvoyer qu'une seule ligne) :

```
SELECT ma_fonction('arg1', 'arg2');
```

- dans un `SELECT`, en passant en argument les valeurs d'une colonne d'une table :

```
SELECT ma_fonction(ma_colonne) FROM ma_table;
```

- dans le `FROM` d'un `SELECT`, la fonction renvoit ici généralement plusieurs lignes (`SETOF`), et un résultat de type `RECORD` :

```
SELECT result FROM ma_fonction() AS f(result);
```

- dans un `INSERT` pour générer la valeur à insérer :

```
INSERT INTO ma_table(ma_colonne) VALUES ( ma_fonction() );
```

[16] https://docs.postgresql.fr/current/sql-call.html

- dans une création d'index (index fonctionnel, la fonction sera réellement appelée lors des mises à jour de l'index… attention la fonction doit être déclarée `IMMUTABLE`) :

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

- appel d'une fonction en paramètre d'une autre fonction ou d'une procédure, par exemple ici le résultat de la fonction `ma_fonction()` (qui doit renvoyer une seule ligne) est passé en argument d'entrée de la procédure `ma_procedure()` :

```
CALL ma_procedure( ma_fonction() );
```

Par ailleurs, certaines fonctions sont spécialisées et ne peuvent être invoquées que dans le contexte pour lequel elles ont été conçues (fonctions trigger, d'agrégat, de fenêtrage, etc.).

1.6 CRÉATION ET MAINTENANCE DES FONCTIONS ET PROCÉDURES

1.6.1 CRÉATION

- `CREATE FUNCTION`
- `CREATE PROCEDURE`

Voici la syntaxe complète pour une fonction d'après la documentation[17] :

```
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
    | SUPPORT support_function
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
    | sql_body
  } …
```

Voici la syntaxe complète pour une procédure d'après la documentation[18] :

```
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
  } …
```

[17] https://www.postgresql.org/docs/current/sql-createfunction.html
[18] https://www.postgresql.org/docs/current/sql-createprocedure.html

Nous allons décrire les clauses importantes ci-dessous.

1.6.2 LANGAGE

- Le langage de la routine doit être précisé
  `LANGUAGE nomlang`
- Nous étudierons `SQL` et `plpgsql`
- Aussi : `plpython3u`, `plperl`, `pl/R`…

Il n'y a pas de langage par défaut. Il est donc nécessaire de le spécifier à chaque création d'une routine.

Ici ce sera surtout : `LANGUAGE plpgsql`.

Une routine en pur SQL indiquera `LANGUAGE sql`. On rencontrera aussi `plperl`, `plpython3u`, etc. en fonction des besoins.

1.6.3 STRUCTURE D'UNE ROUTINE PL/PGSQL

- Reprenons le code montré plus haut :

```
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 `:=`

1.6.4 STRUCTURE D'UNE ROUTINE PL/PGSQL (SUITE)

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

Une routine est composée d'un bloc de déclaration des variables locales et d'un bloc de code. Le bloc de déclaration commence par le mot clé `DECLARE` et se termine avec le mot clé `BEGIN`. Ce mot clé est celui qui débute le bloc de code. La fin est indiquée par le mot clé `END`.

Toutes les instructions se terminent avec des points-virgules. Attention, `DECLARE`, `BEGIN` et `END` ne sont pas des instructions.

Il est possible d'ajouter des commentaires. `--` indique le début d'un commentaire qui se terminera en fin de ligne. Pour être plus précis dans la délimitation, il est aussi possible d'utiliser la notation C : `/*` est le début d'un commentaire et `*/` la fin.

1.6.5 BLOCS NOMMÉS

- Labels de bloc possibles
- Plusieurs blocs d'exception possibles dans une routine
- 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 ];
```

Bien sûr, il est aussi 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;
```

1.6.6 MODIFICATION DU CODE D'UNE ROUTINE

- `CREATE OR REPLACE FUNCTION`
- `CREATE OR REPLACE PROCEDURE`
- Une routine est définie par son nom et ses arguments
- Si type de retour différent, la fonction doit d'abord être supprimée puis recréée

Une routine est surchargeable. La seule façon de les différencier est de prendre en compte les arguments (nombre et type). Les noms des arguments peuvent être indiqués mais ils seront ignorés.

Deux routines identiques aux arguments près (on parle de prototype) ne sont pas identiques, mais bien deux routines distinctes.

`CREATE OR REPLACE` a principalement pour but de modifier le code d'une routine, mais il est aussi possible de modifier les méta-données.

1.6.7 MODIFICATION DES MÉTA-DONNÉES D'UNE ROUTINE

- `ALTER FUNCTION` / `ALTER PROCEDURE`
- Une routine est définie par son nom et ses arguments
- Permet de modifier nom, propriétaire, schéma et autres options

Toutes les méta-données discutées plus haut sont modifiables avec un `ALTER`.

1.6.8 SUPPRESSION D'UNE ROUTINE

- Une routine est définie par son nom et ses arguments :

```
DROP FUNCTION addition (integer, integer) ;
DROP PROCEDURE public.vide_tables (boolean);
DROP PROCEDURE public.vide_tables ();
```

La suppression se fait avec l'ordre `DROP`.

Une fonction pouvant exister en plusieurs exemplaires, avec le même nom et des arguments de type différents, il faudra parfois parfois préciser ces derniers.

1.6.9 UTILISATION DES GUILLEMETS

- Les guillemets deviennent très rapidement pénibles
  - préférer `$$`
  - ou `$fonction$`, `$toto$`...

Définir une fonction entre guillemets simples (`'`) devient très pénible dès que la fonction doit en contenir parce qu'elle contient elle-même des chaînes de caractères. PostgreSQL permet de remplacer les guillemets par `$$`, ou tout mot encadré de `$`.

Par exemple, on peut reprendre la syntaxe de déclaration de la fonction `addition()` précédente en utilisant cette méthode :

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

Ce peut être utile aussi dans tout code réalisant une concaténation de chaînes de caractères contenant des guillemets. La syntaxe traditionnelle impose de les multiplier pour les protéger, et le code devient difficile à lire. :

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

En voilà une 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 routine plus complexe, par exemple incluant le nom de la fonction.

1.7 PARAMÈTRES ET RETOUR DES FONCTIONS ET PROCÉDURES

1.7.1 VERSION MINIMALISTE

```
CREATE FUNCTION fonction (entier integer, texte text)
RETURNS int  AS …
```

Ceci une forme de fonction très simple (et très courante) : deux paramètres en entrée (implicitement en entrée seulement), et une valeur en retour.

Dans le corps de la fonction, il est aussi possible d'utiliser une notation numérotée au lieu des noms de paramètre : le premier argument a pour nom `$1`, le deuxième `$2`, etc. C'est à éviter.

Tous les types sont utilisables, y compris les types définis par l'utilisateur. En dehors des types natifs de PostgreSQL, PL/pgSQL ajoute des types de paramètres spécifiques pour faciliter l'écriture des routines.

1.7.2 PARAMÈTRES IN, OUT & RETOUR

```
CREATE FUNCTION cree_utilisateur (nom text, type_id int DEFAULT 0)
 RETURNS id_utilisateur int AS …
CREATE FUNCTION explose_date (IN d date, OUT jour int, OUT mois int, OUT annee int)
 AS …
```

- `IN` / `OUT` / `INOUT` : entrée/sortie/les 2
- `VARIADIC` : nombre variable
- nom (libre et optionnel)
- type (parmi tous les types de base et les types utilisateur)
- `DEFAULT` : valeur par défaut

Si le mode d'un argument est omis, `IN` est la valeur implicite : la valeur en entrée ne sera pas modifiée.

Un paramètre `OUT` sera modifié. S'il s'agit d'une variable d'un bloc PL appelant, sa valeur sera modifiée. Un paramètre `INOUT` est un paramètre en entrée mais sera également modifié.

Dans le corps d'une fonction, `RETURN` est inutile avec des paramètres `OUT` parce que c'est la valeur des paramètres `OUT` à la fin de la fonction qui est retournée, comme dans l'exemple plus bas.

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 comme renvoyant une table (clause `RETURNS TABLE`, voir plus loin).

Jusque PostgreSQL 13 inclus, les procédures ne supportent pas les arguments `OUT`, seulement `IN` et `INOUT`.

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.

1.7.3 TYPE EN RETOUR : 1 VALEUR SIMPLE

- Fonctions uniquement

```
RETURNS type     -- int, text, etc
```

- Tous les types de base & utilisateur
- Rien : `void`

Le type de retour (clause `RETURNS` dans l'entête) est obligatoire pour les fonctions et interdit pour les procédures.

Avant la version 11, il n'était pas possible de créer une procédure, mais il était possible de créer une fonction se comportant globalement comme une procédure en utilisant le type de retour `void`.

Des exemples plus haut utilisent des types simples, mais tous ceux de PostgreSQL ou les types créés par l'utilisateur sont utilisables.

Depuis le corps de la fonction, le résultat est renvoyé par un appel à `RETURN` (PL/pgSQL) ou `SELECT` (SQL).

1.7.4 TYPE EN RETOUR : 1 LIGNES, PLUSIEURS CHAMPS

3 options :

- Type composé dédié

```
CREATE TYPE ma_structure AS ( … ) ;
CREATE FUNCTION …  RETURNS ma_structure ;
```

- Paramètres OUT

```
CREATE FUNCTION explose_date (IN d date, OUT jour int, OUT mois int, OUT annee int) AS …
```

- RETURNS TABLE

```
CREATE FUNCTION explose_date_table (d date)
RETURNS TABLE (jour integer, mois integer, annee integer) AS…
```

S'il y a besoin de renvoyer plusieurs valeurs à la fois, une possibilité est de renvoyer un type composé défini auparavant.

Une alternative courante est d'utiliser plusieurs paramètres OUT (et pas de clause RETURN dans l'entête) pour obtenir un enregistrement composite :

```
CREATE OR REPLACE FUNCTION explose_date (IN d date, OUT jour int, OUT mois int, OUT annee int)
AS $$
SELECT extract ('d' FROM d)::int, extract('m' FROM d)::int, extract ('y' FROM d)::int
$$
LANGUAGE SQL;

SELECT * FROM explose_date ('31-12-2020');

 jour | mois | annee
------+------+-------
   31 |    0 |  2020
```

(Noter que l'exemple ci-dessus est en simple SQL.)

La clause TABLE est une autre alternative, sans doute plus claire. Cet exemple devient alors, toujours en pur SQL :

```
CREATE OR REPLACE FUNCTION explose_date_table (d date)
RETURNS TABLE (jour integer, mois integer, annee integer)
LANGUAGE sql
AS $function$
  SELECT  extract ('d' FROM d)::int, extract('m' FROM d)::int, extract ('y' FROM d)::int ;
$$ ;
```

1.7.5 RETOUR MULTI-LIGNES

- 1 seul champ ou plusieurs ?

```
RETURNS SETOF type   -- int, text, type personnalisé
RETURNS TABLE ( col1 type, col2 type … )
```

- Ligne à ligne ou en bloc ?

```
RETURN NEXT …
RETURN QUERY   SELECT …
RETURN QUERY   EXECUTE …
```

Pour renvoyer plusieurs lignes, la première possibilité est de déclarer un type de retour `SETOF`. Cet exemple utilise `RETURN NEXT` pour renvoyer les lignes une à une :

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

SELECT * FROM liste_entiers_setof (3) ;
 liste_entiers_setof
---------------------
                   1
                   2
                   3
(3 lignes)
```

S'il y plusieurs champs à renvoyer, une possibilité est d'utiliser un type dédié (composé), qu'il faudra cependant créer auparavant. L'exemple suivant utilise aussi un `RETURN QUERY` pour éviter d'itérer sur toutes les lignes du résultat :

```
CREATE TYPE pgt AS (schemaname text, tablename text) ;

CREATE OR REPLACE FUNCTION tables_by_owner (p_owner text)
RETURNS SETOF pgt
LANGUAGE plpgsql
AS $$
BEGIN
  RETURN QUERY SELECT schemaname::text, tablename::text
               FROM pg_tables WHERE tableowner=p_owner
               ORDER BY tablename ;
END ; $$ ;
```

```
SELECT * FROM tables_by_owner ('pgbench');

 schemaname |    tablename
------------+------------------
 public     | pgbench_accounts
 public     | pgbench_branches
 public     | pgbench_history
 public     | pgbench_tellers
(4 lignes)
```

On a vu que la clause `TABLE` permet de renvoyer plusieurs champs. Or, elle implique aussi `SETOF`, et les deux exemples ci-dessus peuvent devenir :

```
CREATE OR REPLACE FUNCTION liste_entiers_table (limite int)
RETURNS TABLE (j int)
AS $$
BEGIN
  FOR i IN 1..limite LOOP
    j = i ;
    RETURN NEXT ;  -- renvoie la valeur de j en cours
  END LOOP;
END $$ LANGUAGE plpgsql;

SELECT * FROM liste_entiers_table (3) ;
 j
---
 1
 2
 3
(3 lignes)
```

(Noter ici que le nom du champ retourné dépend du nom de la variable utilisée, et n'est pas forcément le nom de la fonction. En effet, chaque appel à `RETURN NEXT` retourne un enregistrement composé d'une copie de toutes les variables, au moment de l'appel à `RETURN NEXT`.)

```
CREATE OR REPLACE FUNCTION tables_by_owner (p_owner text)
RETURNS TABLE (schemaname text, tablename text)
LANGUAGE plpgsql
AS $$
BEGIN
  RETURN QUERY SELECT schemaname::text, tablename::text
               FROM pg_tables WHERE tableowner=p_owner
               ORDER BY tablename ;
END ; $$ ;
```

La variante `RETURN QUERY EXECUTE …` est destinée à des requêtes en SQL dynamique.

Les fonctions avec `RETURN QUERY` ou `RETURN NEXT` stockent tout le résultat avant de le retourner en bloc. Le paramètre `work_mem` permet de définir la mémoire utilisée avant l'utilisation d'un fichier temporaire, qui a bien sûr un impact sur les performances.

Si `RETURNS TABLE` est peut-être le plus souple et clair, le choix entre toutes ces méthodes est affaire de goût, ou de compatibilité avec du code ancien ou converti d'un produit concurrent.

Quand plusieurs lignes sont renvoyées, tout est conservé en mémoire jusqu'à la fin de la fonction. Donc, si beaucoup de données sont renvoyées, cela poser des problèmes de latence, voire de mémoire.

En général, l'appel se fait ainsi pour obtenir des lignes :

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

Une alternative est d'utiliser :

```
SELECT ma_fonction();
```

pour récupérer un résultat d'une seule colonne, scalaire, type composite ou `RECORD` suivant la fonction.

Cette différence concerne aussi les fonctions système :

```
# SELECT * FROM pg_control_system () ;

 pg_control_version | catalog_version_no |  system_identifier  | pg_control_last_modified
--------------------+--------------------+---------------------+--------------------------
               1201 |          201909212 | 6744959735975969621 | 2021-09-17 18:24:05+02
(1 ligne)

# SELECT pg_control_system () ;

                      pg_control_system
-------------------------------------------------------------
 (1201,201909212,6744959735975969621,"2021-09-17 18:24:05+02")
 (1 ligne)
```

1.7.6 GESTION DES VALEURS NULL

- Comment gérer les paramètres à `NULL` ?
- `STRICT` :
  - 1 paramètre `NULL` : retourne `NULL` immédiatement
- Défaut :
  - gestion par la fonction

Si une fonction est définie comme `STRICT` et qu'un des arguments d'entrée 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).

Dans la définition d'une fonction, les options sont `STRICT` ou son synonyme `RETURNS NULL ON NULL INPUT`, ou le défaut implicite `CALLED ON NULL INPUT`.

1.8 VARIABLES EN PL/PGSQL

1.8.1 CLAUSE DECLARE

- Dans le source, partie `DECLARE` :

```
DECLARE
  i  integer;
  j  integer := 5;
  k  integer NOT NULL DEFAULT 1;
  ch text    COLLATE "fr_FR";
```

- Blocs `DECLARE`/`BEGIN`/`END` imbriqués possible
  - restriction de scope de variable

En PL/pgSQL, pour utiliser une variable dans le corps de la routine (entre le `BEGIN` et le `END`), il est obligatoire de l'avoir déclarée précédemment :

- soit dans la liste des arguments (`IN`, `INOUT` ou `OUT`) ;
- soit dans la section `DECLARE`.

La déclaration doit impérativement préciser le nom et le type de la variable.

En option, il est également possible de préciser :

- sa valeur initiale (si rien n'est précisé, ce sera `NULL` par défaut) :

```
answer integer := 42;
```

- sa valeur par défaut, si on veut autre chose que `NULL` :

```
answer integer DEFAULT 42;
```

- une contrainte `NOT NULL` (dans ce cas, il faut impérativement un défaut différent de `NULL`, et toute éventuelle affectation ultérieure de `NULL` à la variable provoquera une erreur) :

```
answer integer NOT NULL DEFAULT 42;
```

- le collationnement à utiliser, pour les variables de type chaîne de caractères :

```
question text COLLATE "en_GB";
```

Pour les fonctions complexes, avec plusieurs niveaux de boucle par exemple, il est possible d'imbriquer les blocs `DECLARE`/`BEGIN`/`END` en y déclarant des variables locales à ce bloc. Si une variable est par erreur utilisée hors du *scope* prévu, une erreur surviendra.

1.8.2 CONSTANTES

- Clause supplémentaire `CONSTANT` :

```
DECLARE
  eur_to_frf   CONSTANT numeric := 6.55957 ;
  societe_nom  CONSTANT text    := 'Dalibo SARL';
```

L'option `CONSTANT` permet de définir une variable pour laquelle il sera alors impossible d'assigner une valeur dans le reste de la routine.

1.8.3 TYPES DE VARIABLES

- Récupérer le type d'une autre variable avec `%TYPE` :

```
quantite    integer ;
total       quantite%TYPE ;
```

- Récupérer le type de la colonne d'une table :

```
quantite    ma_table.ma_colonne%TYPE ;
```

Cela permet d'écrire des routines plus génériques.

1.8.4 TYPE ROW - 1

- Pour renvoyer 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 …;
```

1.8.5 TYPE ROW - 2

- 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 seul nom de la table est donc suffisant.

1.8.6 TYPE RECORD

- `RECORD` identique au type `ROW`
  - …sauf que son type n'est connu que lors de son affectation
- `RECORD` peut changer de type au cours de l'exécution de la routine
- Curseur et boucle sur une requête

`RECORD` est beaucoup utilisé pour manipuler des curseurs, ou dans des boucles `FOR … LOOP` : cela évite de devoir se préoccuper de déclarer un type correspondant exactement aux colonnes de la requête associée à chaque curseur.

1.8.7 TYPE RECORD : EXEMPLE

```
CREATE FUNCTION ma_fonction () RETURNS integer
AS $$
DECLARE
  ligne RECORD;
BEGIN
  -- récupération de la 1è ligne uniquement
  SELECT * INTO ligne FROM ma_première_table;
  -- ou : traitement ligne à ligne
  FOR ligne IN SELECT * FROM ma_deuxième_table  LOOP
    …
  END LOOP ;
  RETURN … ;
END $$ ;
```

Dans ces exemples, on récupère la première ligne de la fonction avec `SELECT … INTO`, puis on ouvre un curseur implicite pour balayer chaque ligne obtenue d'une deuxième table. Le type `RECORD` permet de ne pas déclarer une nouvelle variable de type ligne.

1.9 EXÉCUTION DE REQUÊTE DANS UN BLOC PL/PGSQL

1.9.1 REQUÊTE DANS UN BLOC PL/PGSQL

- Toutes opérations sur la base de données
- Et calculs, comparaisons, etc.
- Toute expression écrite en PL/pgSQL sera passée à `SELECT` pour interprétation par le moteur
- `PREPARE` implicite, avec cache

Par expression, on entend par exemple des choses comme :

```
IF myvar > 0 THEN
    myvar2 := 1 / myvar;
END IF;
```

Dans ce cas, l'expression `myvar > 0` sera préparée par le moteur de la façon suivante :

```
PREPARE statement_name(integer, integer) AS SELECT $1 > $2;
```

Puis cette requête préparée sera exécutée en lui passant en paramètre la valeur de `myvar` et la constante `0`.

Si `myvar` est supérieur à `0`, il en sera ensuite de même pour l'instruction suivante :

```
PREPARE statement_name(integer, integer) AS SELECT $1 / $2;
```

Comme toute requête préparée, son plan sera mis en cache.

Pour les détails, voir les dessous de PL/pgSQL[19] .

1.9.2 AFFECTATION D'UNE VALEUR À UNE VARIABLE

- Utiliser l'opérateur `:=` :

```
un_entier := 5;
```

- Utiliser `SELECT INTO` :

```
SELECT 5 INTO un_entier;
```

Privilégiez la première écriture pour la lisibilité, la seconde écriture est moins claire et n'apporte rien puisqu'il s'agit ici d'une affectation de constante.

À noter que l'écriture suivante est également possible pour une affectation :

[19] https://docs.postgresql.fr/current/plpgsql-implementation.html#PLPGSQL-PLAN-CACHING

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

Cette méthode profite du fait que toutes les expressions du code PL/pgSQL vont être passées au moteur SQL de PostgreSQL dans un `SELECT` pour être résolues. Cela va fonctionner, mais c'est très peu lisible, et donc non recommandé.

1.9.3 EXÉCUTION D'UNE REQUÊTE

- Affectation de la ligne :

```
SELECT *
INTO ma_variable_ligne  -- type ROW ou RECORD
FROM …;
```

- `INTO STRICT` pour garantir unicité
  - `INTO` seul : juste 1è ligne !
- Plus d'un enregistrement :
  - écrire une boucle
- Ordre statique :
  - colonnes, clause `WHERE`, tables figées

Récupérer une ligne de résultat d'une requête dans une ligne de type `ROW` ou `RECORD` se fait avec `SELECT … INTO`. La première ligne est récupérée. Généralement on préférera utiliser `INTO STRICT` pour lever une de ces erreurs si la requête renvoie zéro ou plusieurs lignes :

```
ERROR:  query returned no rows
ERROR:  query returned more than one row
```

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.

Il est possible d'utiliser `SELECT INTO` avec une simple variable si l'on n'a qu'un champ d'une ligne à récupérer.

Cette fonction compte les tables, et la trace les liste (les tables ne font pas partie du résultat) :

```
CREATE OR REPLACE FUNCTION compte_tables () RETURNS int LANGUAGE plpgsql AS $$
DECLARE
  n int ;
  t RECORD ;
BEGIN
  SELECT count(*) INTO STRICT n
  FROM pg_tables ;
```

```
  FOR t IN SELECT * FROM pg_tables LOOP
    RAISE NOTICE 'Table %.%', t.schemaname, t.tablename;
  END LOOP ;

  RETURN n ;
END ;
$$ ;

# SELECT compte_tables ();

NOTICE:  Table pg_catalog.pg_foreign_server
NOTICE:  Table pg_catalog.pg_type
…
NOTICE:  Table public.pgbench_accounts
NOTICE:  Table public.pgbench_branches
NOTICE:  Table public.pgbench_tellers
NOTICE:  Table public.pgbench_history
 compte_tables
---------------
           186
(1 ligne)
```

1.9.4 EXÉCUTION D'UNE REQUÊTE SANS BESOIN DU RÉSULTAT

- `PERFORM` : résultat ignoré

```
PERFORM * FROM ma_table WHERE une_colonne>0 ;
PERFORM mafonction (argument1) ;
```

- Variable `FOUND`
  - si une ligne est affectée par l'instruction
- Nombre de lignes :

```
GET DIAGNOSTICS variable = ROW_COUNT;
```

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

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

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

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

Pour récupérer le nombre de lignes affectées par l'instruction exécutée, il faut récupérer la variable de diagnostic `ROW_COUNT` :

```
GET DIAGNOSTICS variable = ROW_COUNT;
```

Il est à noter que le `ROW_COUNT` récupéré ainsi 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.

1.10 SQL DYNAMIQUE

1.10.1 EXECUTE D'UNE REQUÊTE

- `EXECUTE 'chaine' [INTO [STRICT] cible] [USING (paramètres)] ;`
- Exécute la requête dans `chaine`
- `chaine` peut être construite à partir d'autres variables
- `cible` : résultat (une seule ligne)

`EXECUTE` dans un bloc PL/pgSQL permet notamment du SQL dynamique : l'ordre peut être construit dans une variable.

1.10.2 EXECUTE & REQUÊTE DYNAMIQUE : INJECTION SQL

Si `nom` vaut : « `'Robert' ; DROP TABLE eleves ;` »
que renvoie ceci ?

```
EXECUTE 'SELECT * FROM eleves WHERE nom = '|| nom ;
```

Un danger du SQL dynamique est de faire aveuglément confiance aux valeurs des variables en construisant un ordre SQL :

```
CREATE TEMP TABLE eleves (nom text, id int) ;
INSERT INTO eleves VALUES ('Robert', 0) ;

-- Mise à jour d'un ID
DO $f$
DECLARE
    nom text := $$'Robert' ; DROP TABLE eleves;$$  ;
    id  int ;
BEGIN
RAISE NOTICE 'A exécuter : %','SELECT * FROM eleves WHERE nom = '|| nom ;
EXECUTE 'UPDATE eleves SET id = 327 WHERE nom = '|| nom ;
END ;
$f$ LANGUAGE plpgsql ;

NOTICE:  A exécuter : SELECT * FROM eleves WHERE nom = 'Robert' ; DROP TABLE eleves;

\d+ eleves
Aucune relation nommée « eleves » n'a été trouvée.
```

Cet exemple est directement inspiré d'un dessin très connu de XKCD[20] .

[20] https://xkcd.com/327/

Dans la pratique, la variable `nom` (entrée ici en dur) proviendra par exemple d'un site web, et donc contient potentiellement des caractères terminant la requête dynamique et en insérant une autre, potentiellement destructrice.

Moins grave, une erreur peut être levée à cause d'une apostrophe (*quote*) dans une chaîne texte. Il existe effectivement des gens avec une apostrophe dans le nom.

Ce qui suit concerne le SQL dynamique dans des routines PL/pgSQL, mais le principe concerne tous les langages et clients, y compris `psql` et sa méta-commande `\gexec`[21] . En SQL pur, la protection contre les injections SQL est un argument pour utiliser les requêtes préparées[22] , dont l'ordre `EXECUTE` diffère de celui-ci du PL/pgSQL ci-dessous.

1.10.3 EXECUTE & REQUÊTE DYNAMIQUE : 3 POSSIBILITÉS

```
EXECUTE 'UPDATE tbl SET '
    || quote_ident(nom_colonne)
    || ' = '
    || quote_literal(nouvelle_valeur)
    || ' WHERE cle = '
    || quote_literal(valeur_cle) ;
EXECUTE format('UPDATE matable SET %I = %L '
   'WHERE clef = %L', nom_colonne, nouvelle_valeur, valeur_clef);
EXECUTE format('UPDATE table SET %I = $1 '
   'WHERE clef = $2', nom_colonne) USING nouvelle_valeur, valeur_clef;
```

Les trois exemples précédents sont équivalents.

Le premier est le plus simple au premier abord. Il utilise `quote_ident` et `quote_literal` pour protéger des injections SQL[23] (voir plus loin).

Le second est plus lisible grâce à la fonction de formatage `format`[24] qui évite ces concaténations et appelle implicitement les fonctions `quote_%` Si un paramètre ne peut pas prendre la valeur NULL, utiliser `%L` (équivalent de `quote_nullable`) et non `%I` (équivalent de `quote_ident`).

La troisième alternative avec `USING` et les paramètres numériques `$1` et `$2` est considérée comme la plus performante.

(Voir les détails dans la documentation[25]).

[21] https://docs.postgresql.fr/current/app-psql.html#APP-PSQL-META-COMMANDS
[22] https://docs.postgresql.fr/current/sql-prepare.html
[23] https://fr.wikipedia.org/wiki/Injection_SQL
[24] https://docs.postgresql.fr/current/functions-string.html#FUNCTIONS-STRING-FORMAT
[25] https://docs.postgresql.fr/current/plpgsql-statements.html#PLPGSQL-QUOTE-LITERAL-EXAMPLE

L'exemple complet suivant tiré de la documentation officielle[26] utilise `EXECUTE` pour rafraîchir des vues matérialisées en masse.

```
CREATE FUNCTION rafraichir_vuemat() RETURNS integer AS $$
DECLARE
    mviews RECORD;
BEGIN
    RAISE NOTICE 'Rafraichissement de toutes les vues matérialisées…';

    FOR mviews IN
       SELECT n.nspname AS mv_schema,
              c.relname AS mv_name,
              pg_catalog.pg_get_userbyid(c.relowner) AS owner
         FROM pg_catalog.pg_class c
    LEFT JOIN pg_catalog.pg_namespace n ON (n.oid = c.relnamespace)
        WHERE c.relkind = 'm'
     ORDER BY 1
    LOOP
        -- Maintenant "mviews" contient un enregistrement avec les informations sur la vue matérialisé
        RAISE NOTICE 'Rafraichissement de la vue matérialisée %.% (owner: %)…',
                     quote_ident(mviews.mv_schema),
                     quote_ident(mviews.mv_name),
                     quote_ident(mviews.owner);
        EXECUTE format('REFRESH MATERIALIZED VIEW %I.%I', mviews.mv_schema, mviews.mv_name);
    END LOOP;

    RAISE NOTICE 'Done refreshing materialized views.';
    RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

1.10.4 EXECUTE & REQUÊTE DYNAMIQUE (SUITE)

- `EXECUTE 'chaine' [INTO STRICT cible] [USING (paramètres)] ;`
- `STRICT` : 1 résultat
  - sinon `NO_DATA_FOUND` ou `TOO_MANY_ROWS`
- Sans `STRICT` :
  - 1ère ligne ou `NO_DATA_FOUND`
- Nombre de lignes :
  - `GET DIAGNOSTICS integer_var = ROW_COUNT`

De la même manière que pour `SELECT … INTO`, utiliser `STRICT` permet de garantir qu'il y a

[26] https://www.postgresql.org/docs/current/plpgsql-statements.html#PLPGSQL-QUOTE-LITERAL-EXAMPLE

exactement une valeur comme résultat de `EXECUTE`, ou alors une erreur sera levée.

Nous verrons plus loin comment traiter les exceptions.

1.10.5 OUTILS POUR CONSTRUIRE UNE REQUÊTE DYNAMIQUE

- `quote_ident ()`
  - pour mettre entre guillemets un identifiant d'un objet PostgreSQL (table, colonne, etc.)
- `quote_literal ()`
  - pour mettre entre guillemets une valeur (chaîne de caractères)
- `quote_nullable ()`
  - pour mettre entre guillemets une valeur (chaîne de caractères), sauf NULL qui sera alors renvoyé sans les guillemets
- `||` : concaténer
- Ou fonction `format(…)`, équivalent de `sprintf` en C

La fonction `format` est l'équivalent de la fonction `sprintf` en C : elle formate une chaîne en fonction d'un patron et de valeurs à appliquer à ses paramètres et la retourne. Les types 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 de guillemets doubles si nécessaire ;
- `%L` : est remplacé par une valeur littérale. C'est l'équivalent de la fonction `quote_literal`. Des guillemets simples 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 paramètres 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'
```

1.11 STRUCTURES DE CONTRÔLE EN PL/PGSQL

- But du PL : les traitements procéduraux

1.11.1 TESTS CONDITIONNELS - 2

Exemple :

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

1.11.2 TESTS CONDITIONNELS : CASE

```
    CASE nombre
    WHEN nombre = 0  THEN 'zéro'
    WHEN variable > 0  THEN 'positif'
    WHEN variable < 0  THEN 'négatif'
    ELSE 'indéterminé'
    END CASE
```

ou :

```
CASE current_setting ('server_version_num')::int/10000
    WHEN 8,9            THEN RAISE NOTICE 'Version non supportée !!' ;
    WHEN 10,11,12,13,14 THEN RAISE NOTICE 'Version supportée' ;
    ELSE                     RAISE NOTICE 'Version inconnue au 1/11/2021 ?' ;
END CASE ;
```

L'instruction `CASE WHEN` est proche de l'expression `CASE`[27] des requêtes SQL dans son principe (à part qu'elle se clôt par `END` en SQL, et `END CASE` en PL/pgSQL).

Elle est parfois plus légère à lire que des `IF` imbriqués.

Exemple complet :

```
DO $$
BEGIN
CASE current_setting ('server_version_num')::int/10000
```

[27] https://docs.postgresql.fr/current/functions-conditional.html#FUNCTIONS-CASE

```
    WHEN 8,9            THEN RAISE NOTICE 'Version non supportée !!' ;
    WHEN 10,11,12,13,14 THEN RAISE NOTICE 'Version supportée' ;
    ELSE                     RAISE NOTICE 'Version inconnue au 1/11/2021 ?' ;
END CASE ;
END ;
$$ LANGUAGE plpgsql ;
```

1.11.3 BOUCLE LOOP/EXIT/CONTINUE : SYNTAXE

- Boucle :
  - `LOOP` / `END LOOP`
  - label possible
- En sortir :
  - `EXIT [label] [WHEN expression_booléenne]`
- Commencer une nouvelle itération de la boucle
  - `CONTINUE [label] [WHEN expression_booléenne]`

Des boucles simples s'effectuent avec `LOOP`/`END LOOP`.

Pour les détails, voir la documentation officielle[28] .

1.11.4 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 résultat de 1 à chaque itération tant que la valeur du résultat est inférieure à 50. Ensuite, le résultat est incrémenté de 1 à deux reprises pour chaque tour de boucle. On incrémente donc de 2 par tour de boucle. Arrivée à 100, la procédure sort de la boucle.

---

[28] https://docs.postgresql.fr/current/plpgsql-control-structures.html#PLPGSQL-CONTROL-STRUCTURES-LOOPS

1.11.5 BOUCLE WHILE

```
WHILE condition LOOP
  …
END LOOP;
```

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

1.11.6 BOUCLE FOR : SYNTAXE

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

- `variable` va obtenir les différentes valeurs entre entier1 et entier2
- Label possible

La boucle `FOR` n'a pas d'originalité par rapport à d'autres langages.

L'option `BY` permet d'augmenter l'incrémentation :

```
FOR variable in 1..10 BY 5…
```

L'option `REVERSE` permet de faire défiler les valeurs en ordre inverse :

```
FOR variable in REVERSE 10..1 …
```

1.11.7 BOUCLE FOR ... IN ... LOOP : PARCOURS DE RÉSULTAT DE REQUÊTE

```
FOR ligne IN ( SELECT * FROM ma_table ) LOOP
  …
END LOOP;
```

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

Cette syntaxe très pratique permet de parcourir les lignes résultant d'une requête sans avoir besoin de créer et parcourir un curseur. Souvent on utilisera une variable de type `ROW` ou `RECORD` (comme dans l'exemple de la fonction `rafraichir_vuemat` plus haut), mais l'utilisation directe de variables (déclarées préalablement) est possible :

```
FOR a, b, c, d IN
  (SELECT col_a, col_b, col_c, col_d FROM ma_table)
```

```
LOOP
  -- instructions utilisant ces variables
  …
END LOOP;
```

Attention de ne pas utiliser les variables en question hors de la boucle, elles auront gardé la valeur acquise dans la dernière itération.

1.11.8 BOUCLE FOREACH

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

- Pour 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
- Label possible

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}
```

et avec `SLICE 2`, on obtient :

```
INFO:  var: {{1,2},{3,4},{5,6}}
```

1.12 AUTRES PROPRIÉTÉS DES FONCTIONS

- Sécurité
- Optimisations
- Parallélisation

1.12.1 POLITIQUE DE SÉCURITÉ

- `SECURITY INVOKER` : défaut
- `SECURITY DEFINER`
  - « sudo de la base de données »
  - potentiellement dangereux
  - ne pas laisser à **public** !

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.

Deux points importants sont à noter pour `SECURITY DEFINER` :

- Par défaut, toute fonction créée dans **public** est exécutable par le rôle **public**. La première chose à faire est donc de révoquer ce droit. Créer la fonction dans un schéma séparé permet aussi de gérer plus finalement les accès.

- Il faut se protéger des variables de session qui pourraient être utilisées pour modifier le comportement de la fonction, en particulier le *search_path* (qui pourrait faire pointer vers des tables de même nom dans un autre schéma). 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`) ; ou bien les fonctions doivent préciser systématiquement le schéma (`SELECT … FROM nomschema.nomtable …`).

1.12.2 OPTIMISATION DES FONCTIONS

- Fonctions uniquement
- À destination de l'optimiseur
- `COST cout_execution`
  - coût estimé pour l'exécution de la fonction
- `ROWS nb_lignes_resultat`
  - nombre estimé de lignes que la fonction renvoie

`COST` est un coût représenté en unité de `cpu_operator_cost` (100 par défaut).

`ROWS` vaut par défaut 1000 pour les fonctions `SETOF` ou `TABLE`, et 1 pour les autres.

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.

1.12.3 PARALLÉLISATION

- Fonctions uniquement
- La fonction peut-elle être exécutée en parallèle ?
  - `PARALLEL UNSAFE` (défaut)
  - `PARALLEL RESTRICTED`
  - `PARALLEL SAFE`

`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.

1.13 UTILISATION DE FONCTIONS DANS LES INDEX

- Fonctions uniquement !
- `IMMUTABLE | STABLE | VOLATILE`
- Ce mode précise la « volatilité » de la fonction.
- Permet de réduire le nombre d'appels
- Index : fonctions immuables uniquement (sinon problèmes !)

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. Les fonctions arithmétiques simples (`+`, `*`, `abs`...) sont immuables.

À l'inverse, `now()` n'est évidemment pas immuable. Une fonction sélectionnant des données d'une table non plus. `to_char()` n'est pas non plus immuable, car 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. Une fonction n'exécutant que des `SELECT` sur des tables (pas des fonctions !) sera stable. `to_char()` est stable. L'optimiseur peut réduire ainsi le nombre d'appels sans que ce soit en pratique toujours le cas.

Une fonction est `VOLATILE` dans tous les autres cas. `random()` est volatile. Une fonction volatile peut même modifier les donneés. Une fonction non déclarée comme stable ou immuable est volatile par défaut.

La volatilité des fonctions intégrées à PostgreSQL est déjà définie. C'est au développeur de préciser la volatilité des fonctions qu'il écrit. Ce n'est pas forcément évident. Une erreur peut poser des problèmes quand le plan est mis en cache, ou, on le verra, dans des index.

Quelle importance cela a-t-il ?

Prenons une table d'exemple sur les heures de l'année 2020 :

```
-- Une ligne par heure dans l année, 8784 lignes
CREATE TABLE heures
AS
```

```
SELECT i, '2020-01-01 00:00:00+01:00'::timestamptz + i * interval '1 hour'  AS t
FROM generate_series (1,366*24) i;
```

Définissons une fonction un peu naïve ramenant le premier jour du mois, volatile faute de mieux :

```
CREATE OR REPLACE FUNCTION premierjourdumois(t timestamptz)
RETURNS timestamptz
LANGUAGE plpgsql
VOLATILE
AS $$
BEGIN
    RAISE notice 'appel premierjourdumois' ; -- trace des appels
    RETURN date_trunc ('month', t);
END $$ ;
```

Demandons juste le plan d'un appel ne portant que sur le dernier jour :

```
EXPLAIN SELECT * FROM heures
WHERE t > premierjourdumois('2020-12-31 00:00:00+02:00'::timestamptz)
LIMIT 10 ;
```

```
                                QUERY PLAN
---------------------------------------------------------------------------
Limit  (cost=0.00..8.04 rows=10 width=12)
 ->  Seq Scan on heures  (cost=0.00..2353.80 rows=2928 width=12)
       Filter: (t > premierjourdumois(
                   '2020-12-30 23:00:00+01'::timestamp with time zone))
```

Le nombre de lignes attendues (2928) est le tiers de la table, alors que nous ne demandons que le dernier mois. Il s'agit de l'estimation forfaitaire que PostgreSQL utilise faute d'informations sur ce que va retourner la fonction.

Demander à voir le résultat mène à l'affichage de milliers de `NOTICE` : la fonction est appelée à chaque ligne pour calculer s'il faut filtrer la valeur. En effet, une fonction volatile sera systématiquement exécutée à chaque appel, et, selon le plan, ce peut être pour chaque ligne parcourue !

Cependant notre fonction ne fait que des calculs à partir du paramètre, sans effet de bord. Déclarons-la donc stable :

```
ALTER FUNCTION premierjourdumois(timestamp with time zone) STABLE ;
```

Une fonction stable peut en théorie être remplacée par son résultat pendant l'exécution de la requête. Mais c'est impossible de le faire plus tôt, car on ne sait pas forcément dans quel contexte la fonction va être appelée (par exemple, en cas de requête préparée,

les paramètres de la session ou les données de la base peuvent même changer entre la planification et l'exécution).

Dans notre cas, le même `EXPLAIN` simple mène à ceci :

```
NOTICE:  appel premierjourdumois
                                  QUERY PLAN
------------------------------------------------------------------------------
Limit  (cost=0.00..32.60 rows=10 width=12)
 ->  Seq Scan on heures  (cost=0.00..2347.50 rows=720 width=12)
       Filter: (t > premierjourdumois(
                    '2020-12-30 23:00:00+01'::timestamp with time zone))
```

Comme il s'agit d'un simple `EXPLAIN`, la requête n'est pas exécutée. Or le message `NOTICE` est renvoyé : la fonction est donc exécutée pour une simple planification. Un appel unique suffit, puisque la valeur d'une fonction stable ne change pas pendant toute la durée de la requête pour les mêmes paramètres (ici une constante). Cet appel permet d'affiner la volumétrie des valeurs attendues, ce qui peut avoir un impact énorme.

Cependant, à l'exécution, les `NOTICE` apparaîtront pour indiquer que la fonction est à nouveau appelée à chaque ligne. Pour qu'un seul appel soit effectué pour toute la requête, il faudrait déclarer la fonction comme immuable, ce qui serait faux, puisqu'elle dépend implicitement du fuseau horaire.

Dans l'idéal, une fonction immuable peut être remplacée par son résultat avant même la planification d'une requête l'utilisant. C'est le cas avec les calculs arithmétiques par exemple :

```
EXPLAIN SELECT * FROM heures
WHERE i > abs(364*24) AND t > '2020-06-01'::date + interval '57 hours' ;
```

La valeur est substituée très tôt, ce qui permet de les comparer aux statistiques :

```
Seq Scan on heures  (cost=0.00..179.40 rows=13 width=12)
  Filter: ((i > 8736) AND (t > '2020-06-03 09:00:00'::timestamp without time zone))
```

Pour forcer un appel unique quand on sait que la fonction renverra une constante, du moins le temps de la requête, même si elle est volatile, une astuce est de signifier à l'optimiseur qu'il n'y aura qu'une seule valeur de comparaison, même si on ne sait pas laquelle :

```
EXPLAIN (ANALYZE) SELECT * FROM heures
WHERE t > (SELECT premierjourdumois('2020-12-31 00:00:00+02:00'::timestamptz)) ;

NOTICE:  appel premierjourdumois
                                  QUERY PLAN
------------------------------------------------------------------------------
```

```
Seq Scan on heures  (cost=0.26..157.76 rows=2920 width=12)
                    (actual time=1.090..1.206 rows=721 loops=1)
  Filter: (t > $0)
  Rows Removed by Filter: 8039
  InitPlan 1 (returns $0)
    ->  Result  (cost=0.00..0.26 rows=1 width=8)
                (actual time=0.138..0.139 rows=1 loops=1)
Planning Time: 0.058 ms
Execution Time: 1.328 ms
```

On note qu'il n'y a qu'un appel. On comprend donc l'intérêt de se poser la question à l'écriture de chaque fonction.

La volatilité est encore plus importante quand il s'agit de créer des fonctions sur index :

```
CREATE INDEX ON heures (premierjourdumois( t )) ;

ERROR:  functions in index expression must be marked IMMUTABLE
```

Ceci n'est possible que si la fonction est immuable. En effet, si le résultat de la fonction dépend de l'état de la base ou d'autres paramètres, la fonction exécutée au moment de la création de la clé d'index pourrait ne plus retourner le même résultat quand viendra le moment de l'interroger. PostgreSQL n'acceptera donc que les fonctions immuables dans la déclaration des index fonctionnels.

Déclarer hâtivement une fonction comme immuable juste pour pouvoir l'utiliser dans un index est dangereux : en cas d'erreur, les résultats d'une requête peuvent alors dépendre du plan d'exécution, selon que les index seront utilisés ou pas !

Cela est particulièrement fréquent quand les fuseaux horaires ou les dictionnaires sont impliqués. Vérifiez bien que vous n'utilisez que des fonctions immuables dans les index fonctionnels, les pièges sont nombreux.

Par exemple, si l'on veut une version immuable de la fonction précédente, il faut fixer le fuseau horaire dans l'appel à `date_trunc`. En effet, on peut voir avec `df+ date_trunc` que la seule version immuable de `date_trunc` n'accepte que des `timestamp` (sans fuseau), et en renvoie un. Notre fonction devient donc :

```
CREATE OR REPLACE FUNCTION premierjourdumois_utc(t timestamptz)
RETURNS timestamptz
LANGUAGE plpgsql
IMMUTABLE
AS $$
DECLARE
    jour1   timestamp ; --sans TZ
BEGIN
    jour1 := date_trunc ('month', (t at time zone 'UTC')::timestamp) ;
```

```
    RETURN jour1 AT TIME ZONE 'UTC';
END $$ ;
```

Testons avec une date dans les dernières heures de septembre en Alaska, qui correspond au tout début d'octobre en temps universel, et par exemple aussi au Japon :

```
\x

SET timezone TO 'US/Alaska';

SELECT d,
       d AT TIME ZONE 'UTC' AS d_en_utc,
       premierjourdumois_utc (d),
       premierjourdumois_utc (d) AT TIME ZONE 'UTC' as pjm_en_utc
FROM (SELECT '2020-09-30 18:00:00-08'::timestamptz AS d) x;

-[ RECORD 1 ]---------+-----------------------
d                     | 2020-09-30 18:00:00-08
d_en_utc              | 2020-10-01 02:00:00
premierjourdumois_utc | 2020-09-30 16:00:00-08
pjm_en_utc            | 2020-10-01 00:00:00

SET timezone TO 'Japan';

SELECT d,
       d AT TIME ZONE 'UTC' AS d_en_utc,
       premierjourdumois_utc (d),
       premierjourdumois_utc (d) AT TIME ZONE 'UTC' as pjm_en_utc
FROM (SELECT '2020-09-30 18:00:00-08'::timestamptz AS d) x;

-[ RECORD 1 ]---------+-----------------------
d                     | 2020-10-01 11:00:00+09
d_en_utc              | 2020-10-01 02:00:00
premierjourdumois_utc | 2020-10-01 09:00:00+09
pjm_en_utc            | 2020-10-01 00:00:00
```

Malgré les différences d'affichage dues au fuseau horaire, c'est bien le même moment (la première seconde d'octobre en temps universel) qui est retourné par la fonction.

Pour une fonction aussi simple, la version SQL est même préférable :

```
CREATE OR REPLACE FUNCTION premierjourdumois_utc(t timestamptz)
RETURNS timestamptz
LANGUAGE sql
IMMUTABLE
AS $$
    SELECT (date_trunc ('month', (t at time zone 'UTC')::timestamp)) AT TIME ZONE 'UTC';
$$ ;
```

Enfin, la volatilité a également son importance lors d'autres opérations d'optimisation, comme l'exclusion de partitions. Seules les fonctions immuables sont compatibles avec le *partition pruning* effectué à la planification, mais les fonctions stable sont éligibles au *dynamic partition pruning* (à l'exécution) apparu avec PostgreSQL 11.

1.14 CONCLUSION

- Grand nombre de structure de contrôle (test, boucle, etc.)
- Facile à utiliser et à comprendre
- Attention à la compatibilité ascendante

1.14.1 POUR ALLER PLUS LOIN

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

La documentation officielle sur le langage PL/pgSQL peut être consultée en français à cette adresse[29] .

1.14.2 QUESTIONS

```
FOR q IN (SELECT * FROM questions ) LOOP

  répondre (q) ;

END LOOP ;
```

---

[29] https://docs.postgresql.fr/current/plpgsql.html

1.15 QUIZ

https://dali.bo/p1_quiz

1.16 TRAVAUX PRATIQUES

1.16.1 HELLO

> 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.

1.16.2 DIVISION

> Écrire en PL/pgSQL une fonction de division appelée `division`. Elle acceptera en entrée deux arguments de type entier et renverra un nombre réel (`numeric`).

> Écrire cette même fonction en SQL.

> Comment corriger le problème de la division par zéro ? Écrire cette nouvelle fonction dans les deux langages.
> (Conseil : dans ce genre de calcul impossible, il est possible d'utiliser la constante `NaN` (*Not A Number*)).

1.16.3 SELECT SUR DES TABLES DANS LES FONCTIONS

Ce TP utilise les tables de la base **employes_services**.

Le script de création de la base peut être téléchargé depuis https://dali.bo/tp_employes_services. Il ne fait que 3,5 ko. Le chargement se fait de manière classique :

```
$ psql < employes_services.sql
```

Les quelques tables occupent environ 80 Mo sur le disque.

> Créer une fonction qui ramène le nombre d'employés embauchés une année donnée (à partir du champ `employes.date_embauche`).

> Utiliser la fonction `generate_series()` pour lister le nombre d'embauches pour chaque année entre 2000 et 2010.

> Créer une fonction qui fait la même chose avec deux années en paramètres une boucle `FOR … LOOP`, `RETURNS TABLE` et `RETURN NEXT`.

1.16.4 MULTIPLICATION

> Écrire une fonction de multiplication dont les arguments sont des chiffres en toute lettre, inférieurs ou égaux à « neuf ». Par exemple, `multiplication ('deux','trois')` doit renvoyer 6.

> Si ce n'est déjà fait, faire en sorte que `multiplication` appelle une autre fonction pour faire la conversion de texte en chiffre, et n'effectue que le calcul.

> Essayer de multiplier « deux » par 4. Qu'obtient-on et pourquoi ?

> Corriger la fonction pour tomber en erreur si un argument est numérique (utiliser `RAISE EXCEPTION <message>`).

1.16.5 SALUTATIONS

> Écrire une fonction en PL/pgSQL qui prend en argument le nom de l'utilisateur, puis lui dit « Bonjour » ou « Bonsoir » suivant l'heure de la journée. Utiliser la fonction `to_char()`.

> Écrire la même fonction avec un paramètre `OUT`.

> Pour calculer l'heure courante, utiliser plutôt la fonction `extract`.

> Réécrire la fonction en SQL.

1.16.6 INVERSION DE CHAÎNE

> Écrire une fonction `inverser` qui inverse une chaîne (pour « toto » en entrée, afficher « otot » en sortie), à l'aide d'une boucle `WHILE` et des fonctions `char_length` et `substring`.

1.16.7 JOURS FÉRIÉS

Le calcul de la date de Pâques est complexe[30] . On peut écrire la fonction suivante :

```
CREATE OR REPLACE FUNCTION paques(annee integer)
RETURNS date
AS $$
  DECLARE
    a integer;
    b integer;
    r date;
  BEGIN
    a := (19*(annee % 19) + 24) % 30;
    b := (2*(annee % 4) + 4*(annee % 7) + 6*a + 5) % 7;
    SELECT (annee::text||'-03-31')::date + (a+b-9) INTO r;
    RETURN r;
  END;
$$
LANGUAGE plpgsql;
```

Principe : Soit `m` l'année. On calcule successivement :

- le reste de `m/19` : c'est la valeur de `a`.
- le reste de `m/4` : c'est la valeur de `b`.
- le reste de `m/7` : c'est la valeur de `c`.
- le reste de `(19a + p)/30` : c'est la valeur de `d`.
- le reste de `(2b + 4c + 6d + q)/7` : c'est la valeur de `e`.

Les valeurs de `p` et de `q` varient de 100 ans en 100 ans. De 2000 à 2100, `p` vaut 24, `q` vaut 5. La date de Pâques est le `(22 + d + e)` mars ou le `(d + e - 9)` avril.

> Afficher les dates de Pâques de 2018 à 2022.

> Écrire une fonction qui calcule la date de l'Ascension, soit le jeudi de la sixième semaine après Pâques. Pour simplifier, on peut aussi considérer que l'Ascension se déroule 39 jours après Pâques.

> Enfin, écrire une fonction qui renvoie tous les jours fériés d'une année (libellé et date).
> Prévoir un paramètre supplémentaire pour l'Alsace-Moselle, où le Vendredi saint (précédant le dimanche de Pâques) et le 26

[30] https://fr.wikipedia.org/wiki/Calcul_de_la_date_de_P%C3%A2ques

> décembre sont aussi fériés.
> Cette fonction doit renvoyer plusieurs lignes : utiliser `RETURN NEXT`. Plusieurs variantes sont possibles : avec `SETOF record`, avec des paramètres `OUT`, ou avec `RETURNS TABLE (libelle, jour)`.
> Enfin, il est possible d'utiliser `RETURN QUERY`.

1.17 TRAVAUX PRATIQUES (SOLUTIONS)

1.17.1 HELLO

> Écrire une fonction `hello` qui renvoie la chaîne de caractère « Hello World! » en SQL.

```
CREATE OR REPLACE FUNCTION hello()
RETURNS text
AS $BODY$
  SELECT 'hello world !'::text;
$BODY$
LANGUAGE SQL;
```

> Écrire une fonction `hello_pl` qui renvoie la chaîne de caractère « Hello World! » en PL/pgSQL.

```
CREATE OR REPLACE FUNCTION hello_pl()
RETURNS text
AS $BODY$
  BEGIN
    RETURN 'hello world !';
  END
$BODY$
LANGUAGE plpgsql;
```

> Comparer les coûts des deux plans d'exécutions de ces requêtes. Expliquer les coûts.

Requêtage :

```
EXPLAIN SELECT hello();

                QUERY PLAN
-----------------------------------------
 Result  (cost=0.00..0.01 rows=1 width=32)

EXPLAIN SELECT hello_pl();

                QUERY PLAN
-----------------------------------------
 Result  (cost=0.00..0.26 rows=1 width=32)
```

Par défaut, si on ne précise pas le coût (`COST`) d'une fonction, cette dernière a un coût par défaut de 100. Ce coût est à multiplier par la valeur du paramètre `cpu_operator_cost`, par défaut à 0.0025. Le coût total d'appel de la fonction `hello_pl` est donc par défaut de :

100*cpu_operator_cost + cpu_tuple_cost

Ce n'est pas valable pour la fonction en SQL pur, qui est ici intégrée à la requête.

1.17.2 DIVISION

> Écrire en PL/pgSQL une fonction de division appelée `division`. Elle acceptera en entrée deux arguments de type entier et renverra un nombre réel (`numeric`).

Attention, sous PostgreSQL, la division de deux entiers est par défaut entière : il faut donc transtyper.

```
CREATE OR REPLACE FUNCTION division (arg1 integer, arg2 integer)
RETURNS numeric
AS $BODY$
  BEGIN
    RETURN arg1::numeric / arg2::numeric;
  END
$BODY$
LANGUAGE plpgsql;

SELECT division (3,2) ;

      division
--------------------
 1.5000000000000000
```

> Écrire cette même fonction en SQL.

```
CREATE OR REPLACE FUNCTION division_sql (a integer, b integer)
RETURNS numeric
AS $$
    SELECT a::numeric / b::numeric;
$$
LANGUAGE SQL;
```

> Comment corriger le problème de la division par zéro ? Écrire cette nouvelle fonction dans les deux langages.
> (Conseil : dans ce genre de calcul impossible, il est possible d'utiliser la constante `NaN` (*Not A Number*)).

Le problème se présente ainsi :

```
SELECT division(1,0);
ERROR:  division by zero
CONTEXTE : PL/pgSQL function division(integer,integer) line 3 at RETURN
```

Pour la version en PL :

```
CREATE OR REPLACE FUNCTION division(arg1 integer, arg2 integer)
RETURNS numeric
AS $BODY$
  BEGIN
    IF arg2 = 0 THEN
      RETURN 'NaN';
    ELSE
      RETURN arg1::numeric / arg2::numeric;
    END IF;
  END $BODY$
LANGUAGE plpgsql;

SELECT division (3,0) ;

 division
----------
      NaN
```

Pour la version en SQL :

```
CREATE OR REPLACE FUNCTION division_sql(a integer, b integer)
RETURNS numeric
AS $$
  SELECT CASE $2
    WHEN 0 THEN 'NaN'
    ELSE $1::numeric / $2::numeric
  END;
$$
LANGUAGE SQL;
```

1.17.3 SELECT SUR DES TABLES DANS LES FONCTIONS

> Créer une fonction qui ramène le nombre d'employés embauchés une année donnée (à partir du champ `employes.date_embauche`).

```
CREATE OR REPLACE FUNCTION nb_embauches (v_annee integer)
RETURNS integer
AS $BODY$
  DECLARE
    nb integer;
  BEGIN
    SELECT  count(*)
    INTO    nb
    FROM    employes
    WHERE   extract (year from date_embauche) = v_annee ;
    RETURN nb;
  END
$BODY$
LANGUAGE plpgsql ;
```

Test :

```
SELECT nb_embauches (2006);

 nb_embauches
--------------
            9
```

> Utiliser la fonction `generate_series()` pour lister le nombre d'embauches pour chaque année entre 2000 et 2010.

```
SELECT n, nb_embauches (n)
FROM generate_series (2000,2010) n
ORDER BY n;

  n   | nb_embauches
------+--------------
 2000 |            2
 2001 |            0
 2002 |            0
 2003 |            1
 2004 |            0
 2005 |            2
 2006 |            9
 2007 |            0
 2008 |            0
```

```
 2009 |           0
 2010 |           0
```

> Créer une fonction qui fait la même chose avec deux années en paramètres une boucle `FOR … LOOP`, `RETURNS TABLE` et `RETURN NEXT`.

```
CREATE OR REPLACE FUNCTION nb_embauches (v_anneedeb int, v_anneefin int)
RETURNS TABLE (annee int, nombre_embauches int)
AS $BODY$
  BEGIN
    FOR i in v_anneedeb..v_anneefin
    LOOP
      SELECT i, nb_embauches (i)
      INTO   annee, nombre_embauches ;
      RETURN NEXT ;
    END LOOP;
    RETURN;
  END
$BODY$
LANGUAGE plpgsql;
```

Le nom de la fonction a été choisi identique à la précédente, mais avec des paramètres différents. Cela ne gêne pas le requêtage :

```
SELECT * FROM  nb_embauches (2006,2010);
```

```
 annee | nombre_embauches
-------+------------------
  2006 |                9
  2007 |                0
  2008 |                0
  2009 |                0
  2010 |                0
```

1.17.4 MULTIPLICATION

> Écrire une fonction de multiplication dont les arguments sont des chiffres en toute lettre, inférieurs ou égaux à « neuf ». Par exemple, `multiplication ('deux','trois')` doit renvoyer 6.

```
CREATE OR REPLACE FUNCTION multiplication (arg1 text, arg2 text)
RETURNS integer
AS $BODY$
```

```
DECLARE
  a1 integer;
  a2 integer;
BEGIN
  IF arg1 = 'zéro' THEN
    a1 := 0;
  ELSEIF arg1 = 'un' THEN
    a1 := 1;
  ELSEIF arg1 = 'deux' THEN
    a1 := 2;
  ELSEIF arg1 = 'trois' THEN
    a1 := 3;
  ELSEIF arg1 = 'quatre' THEN
    a1 := 4;
  ELSEIF arg1 = 'cinq' THEN
    a1 := 5;
  ELSEIF arg1 = 'six' THEN
    a1 := 6;
  ELSEIF arg1 = 'sept' THEN
    a1 := 7;
  ELSEIF arg1 = 'huit' THEN
    a1 := 8;
  ELSEIF arg1 = 'neuf' THEN
    a1 := 9;
  END IF;

  IF arg2 = 'zéro' THEN
    a2 := 0;
  ELSEIF arg2 = 'un' THEN
    a2 := 1;
  ELSEIF arg2 = 'deux' THEN
    a2 := 2;
  ELSEIF arg2 = 'trois' THEN
    a2 := 3;
  ELSEIF arg2 = 'quatre' THEN
    a2 := 4;
  ELSEIF arg2 = 'cinq' THEN
    a2 := 5;
  ELSEIF arg2 = 'six' THEN
    a2 := 6;
  ELSEIF arg2 = 'sept' THEN
    a2 := 7;
  ELSEIF arg2 = 'huit' THEN
    a2 := 8;
  ELSEIF arg2 = 'neuf' THEN
    a2 := 9;
```

```
    END IF;

    RETURN a1*a2;
  END
$BODY$
LANGUAGE plpgsql;
```

Test :

```
SELECT multiplication('deux', 'trois');

 multiplication
----------------
              6

SELECT multiplication('deux', 'quatre');

 multiplication
----------------
              8
```

> Si ce n'est déjà fait, créer une autre fonction pour faire la conversion de texte en chiffre, que `multiplication` appellera avant d'effectuer le calcul.

```
CREATE OR REPLACE FUNCTION texte_vers_entier(arg text)
RETURNS integer AS $BODY$
  DECLARE
    ret integer;
  BEGIN
    IF arg = 'zéro' THEN
      ret := 0;
    ELSEIF arg = 'un' THEN
      ret := 1;
    ELSEIF arg = 'deux' THEN
      ret := 2;
    ELSEIF arg = 'trois' THEN
      ret := 3;
    ELSEIF arg = 'quatre' THEN
      ret := 4;
    ELSEIF arg = 'cinq' THEN
      ret := 5;
    ELSEIF arg = 'six' THEN
      ret := 6;
    ELSEIF arg = 'sept' THEN
      ret := 7;
    ELSEIF arg = 'huit' THEN
      ret := 8;
```

```
    ELSEIF arg = 'neuf' THEN
      ret := 9;
    END IF;

    RETURN ret;
  END
$BODY$
LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION multiplication(arg1 text, arg2 text)
RETURNS integer
AS $BODY$
  DECLARE
    a1 integer;
    a2 integer;
  BEGIN
    a1 := texte_vers_entier(arg1);
    a2 := texte_vers_entier(arg2);
    RETURN a1*a2;
  END
$BODY$
LANGUAGE plpgsql;
```

> Essayer de multiplier « deux » par 4. Qu'obtient-on et pourquoi ?

```
SELECT multiplication('deux', 4::text);
 multiplication
----------------
```

Par défaut, les variables internes à la fonction valent NULL. Rien n'est prévu pour affecter le second argument, on obtient donc NULL en résultat.

> Corriger la fonction pour tomber en erreur si un argument est numérique (utiliser `RAISE EXCEPTION <message>`).

```
CREATE OR REPLACE FUNCTION texte_vers_entier(arg text)
RETURNS integer AS $BODY$
  DECLARE
    ret integer;
  BEGIN
    IF arg = 'zéro' THEN
      ret := 0;
    ELSEIF arg = 'un' THEN
      ret := 1;
    ELSEIF arg = 'deux' THEN
```

```
      ret := 2;
    ELSEIF arg = 'trois' THEN
      ret := 3;
    ELSEIF arg = 'quatre' THEN
      ret := 4;
    ELSEIF arg = 'cinq' THEN
      ret := 5;
    ELSEIF arg = 'six' THEN
      ret := 6;
    ELSEIF arg = 'sept' THEN
      ret := 7;
    ELSEIF arg = 'huit' THEN
      ret := 8;
    ELSEIF arg = 'neuf' THEN
      ret := 9;
    ELSE
      RAISE EXCEPTION 'argument "%" invalide', arg;
      ret := NULL;
    END IF;

    RETURN ret;
  END
$BODY$
LANGUAGE plpgsql;

SELECT multiplication('deux', 4::text);
ERROR:  argument "4" invalide
CONTEXTE : PL/pgSQL function texte_vers_entier(text) line 26 at RAISE
PL/pgSQL function multiplication(text,text) line 7 at assignment
```

1.17.5 SALUTATIONS

> Écrire une fonction en PL/pgSQL qui prend en argument le nom de l'utilisateur, puis lui dit « Bonjour » ou « Bonsoir » suivant l'heure de la journée. Utiliser la fonction `to_char()`.

```
CREATE OR REPLACE FUNCTION salutation(utilisateur text)
RETURNS text
AS $BODY$
  DECLARE
    heure integer;
    libelle text;
  BEGIN
    heure := to_char(now(), 'HH24');
```

```
    IF heure > 12
    THEN
      libelle := 'Bonsoir';
    ELSE
      libelle := 'Bonjour';
    END IF;

    RETURN libelle||' '||utilisateur||' !';
  END
$BODY$
LANGUAGE plpgsql;
```

Test :

```
SELECT salutation ('Guillaume');

   salutation
---------------------
 Bonsoir Guillaume !
```

> Écrire la même fonction avec un paramètre `OUT`.

```
CREATE OR REPLACE FUNCTION salutation(IN utilisateur text, OUT message text)
AS $BODY$
  DECLARE
    heure integer;
    libelle text;
  BEGIN
    heure := to_char(now(), 'HH24');
    IF heure > 12
    THEN
      libelle := 'Bonsoir';
    ELSE
      libelle := 'Bonjour';
    END IF;

    message := libelle||' '||utilisateur||' !';
  END
$BODY$
LANGUAGE plpgsql;
```

Elle s'utilise de la même manière :

```
SELECT salutation ('Guillaume');

    salutation
---------------------
 Bonsoir Guillaume !
```

> Pour calculer l'heure courante, utiliser plutôt la fonction `extract`.

```
CREATE OR REPLACE FUNCTION salutation(IN utilisateur text, OUT message text)
AS $BODY$
  DECLARE
    heure integer;
    libelle text;
  BEGIN
    SELECT INTO heure extract(hour from now())::int;
    IF heure > 12
    THEN
      libelle := 'Bonsoir';
    ELSE
      libelle := 'Bonjour';
    END IF;

    message := libelle||' '||utilisateur||' !';
  END
$BODY$
LANGUAGE plpgsql;
```

> Réécrire la fonction en SQL.

Le `CASE … WHEN` remplace aisément un `IF … THEN` :

```
CREATE OR REPLACE FUNCTION salutation_sql(nom text)
RETURNS text
AS $$
  SELECT CASE extract(hour from now()) > 12
    WHEN 't' THEN 'Bonsoir '|| nom
    ELSE 'Bonjour '|| nom
  END::text;
$$ LANGUAGE SQL;
```

1.17.6 INVERSION DE CHAÎNE

> Écrire une fonction `inverser` qui inverse une chaîne (pour « toto » en entrée, afficher « otot » en sortie), à l'aide d'une boucle `WHILE` et des fonctions `char_length` et `substring`.

```
CREATE OR REPLACE FUNCTION inverser(str_in varchar)
RETURNS varchar
AS $$
  DECLARE
    str_out varchar ;   -- à renvoyer
    position integer ;
  BEGIN
    -- Initialisation de str_out, sinon sa valeur reste à NULL
    str_out := '';
    -- Position initialisée ç la longueur de la chaîne
    position := char_length(str_in);
    -- La chaîne est traitée ç l'envers
    -- Boucle: Inverse l'ordre des caractères d'une chaîne de caractères
    WHILE position > 0 LOOP
    -- la chaîne donnée en argument est parcourue
    -- à l'envers,
    -- et les caractères sont extraits individuellement
      str_out := str_out || substring(str_in, position, 1);
      position := position - 1;
    END LOOP;
    RETURN str_out;
  END;
$$
LANGUAGE plpgsql;

SELECT inverser (' toto ') ;

 inverser
----------
  otot
```

1.17.7 JOURS FÉRIÉS

La fonction suivante calcule la date de Pâques d'une année :

```
CREATE OR REPLACE FUNCTION paques(annee integer)
RETURNS date
AS $$
  DECLARE
    a integer;
    b integer;
    r date;
  BEGIN
    a := (19*(annee % 19) + 24) % 30;
    b := (2*(annee % 4) + 4*(annee % 7) + 6*a + 5) % 7;
    SELECT (annee::text||'-03-31')::date + (a+b-9) INTO r;
    RETURN r;
  END;
$$
LANGUAGE plpgsql;
```

> Afficher les dates du dimanche de Pâques de 2018 à 2022.

```
SELECT paques (n) FROM generate_series (2018, 2022) n ;
```

```
   paques
------------
 2018-04-01
 2019-04-21
 2020-04-12
 2021-04-04
 2022-04-17
```

> Écrire une fonction qui calcule la date de l'Ascension, soit le jeudi de la sixième semaine après Pâques. Pour simplifier, on peut aussi considérer que l'Ascension se déroule 39 jours après Pâques.

Version complexe :

```
CREATE OR REPLACE FUNCTION ascension(annee integer)
RETURNS date
AS $$
  DECLARE
    r    date;
  BEGIN
    SELECT paques(annee)::date + 40 INTO r;
    SELECT r + (4 - extract(dow from r))::integer INTO r;
```

```
    RETURN r;
  END;
$$
LANGUAGE plpgsql;
```

Version simple :

```
CREATE OR REPLACE FUNCTION ascension(annee integer)
RETURNS date
AS $$
    SELECT (paques (annee) + INTERVAL '39 days')::date ;
$$
LANGUAGE sql;
```

Test :

```
SELECT paques (n), ascension(n) FROM generate_series (2018, 2022) n ;
```

```
   paques    | ascension
-------------+------------
 2018-04-01 | 2018-05-10
 2019-04-21 | 2019-05-30
 2020-04-12 | 2020-05-21
 2021-04-04 | 2021-05-13
 2022-04-17 | 2022-05-26
```

> Enfin, écrire une fonction qui renvoie tous les jours fériés d'une année (libellé et date).
> Prévoir un paramètre supplémentaire pour l'Alsace-Moselle, où le Vendredi saint (précédant le dimanche de Pâques) et le 26 décembre sont aussi fériés.
> Cette fonction doit renvoyer plusieurs lignes : utiliser `RETURN NEXT`. Plusieurs variantes sont possibles : avec `SETOF record`, avec des paramètres `OUT`, ou avec `RETURNS TABLE (libelle, jour)`.
> Enfin, il est possible d'utiliser `RETURN QUERY`.

Version avec SETOF record :

```
CREATE OR REPLACE FUNCTION vacances(annee integer, alsace_moselle boolean DEFAULT false)
RETURNS SETOF record
AS $$
  DECLARE
    f integer;
    r record;
  BEGIN
```

```
    SELECT 'Jour de l''an'::text, (annee::text||'-01-01')::date INTO r;
    RETURN NEXT r;
    SELECT 'Pâques'::text, paques(annee)::date + 1 INTO r;
    RETURN NEXT r;
    SELECT 'Ascension'::text, ascension(annee)::date INTO r;
    RETURN NEXT r;
    SELECT 'Fête du travail'::text, (annee::text||'-05-01')::date INTO r;
    RETURN NEXT r;
    SELECT 'Victoire 1945'::text, (annee::text||'-05-08')::date INTO r;
    RETURN NEXT r;
    SELECT 'Fête nationale'::text, (annee::text||'-07-14')::date INTO r;
    RETURN NEXT r;
    SELECT 'Assomption'::text, (annee::text||'-08-15')::date INTO r;
    RETURN NEXT r;
    SELECT 'La toussaint'::text, (annee::text||'-11-01')::date INTO r;
    RETURN NEXT r;
    SELECT 'Armistice 1918'::text, (annee::text||'-11-11')::date INTO r;
    RETURN NEXT r;
    SELECT 'Noël'::text, (annee::text||'-12-25')::date INTO r;
    RETURN NEXT r;
    IF alsace_moselle THEN
      SELECT 'Vendredi saint'::text, paques(annee)::date - 2 INTO r;
      RETURN NEXT r;
      SELECT 'Lendemain de Noël'::text, (annee::text||'-12-26')::date INTO r;
      RETURN NEXT r;
    END IF;

    RETURN;
  END;
$$
LANGUAGE plpgsql;
```

Le requêtage implique de nommer les colonnes :

```
SELECT *
FROM vacances(2020, true) AS (libelle text, jour date)
ORDER BY jour ;
```

```
      libelle       |    jour
--------------------+------------
 Jour de l'an       | 2020-01-01
 Vendredi saint     | 2020-04-10
 Pâques             | 2020-04-13
 Fête du travail    | 2020-05-01
 Victoire 1945      | 2020-05-08
 Ascension          | 2020-05-21
 Fête nationale     | 2020-07-14
```

```
 Assomption        | 2020-08-15
 La toussaint      | 2020-11-01
 Armistice 1918    | 2020-11-11
 Noël              | 2020-12-25
 Lendemain de Noël | 2020-12-26
```

Version avec paramètres OUT :

Une autre forme d'écriture possible consiste à indiquer les deux colonnes de retour comme des paramètres `OUT` :

```
CREATE OR REPLACE FUNCTION vacances(
            annee integer,
            alsace_moselle boolean DEFAULT false,
            OUT libelle text,
            OUT jour date)
 RETURNS SETOF record
 LANGUAGE plpgsql
AS $function$
  DECLARE
    f integer;
    r record;
  BEGIN
    SELECT 'Jour de l''an'::text, (annee::text||'-01-01')::date
        INTO libelle, jour;
    RETURN NEXT;
    SELECT 'Pâques'::text, paques(annee)::date + 1 INTO libelle, jour;
    RETURN NEXT;
    SELECT 'Ascension'::text, ascension(annee)::date INTO libelle, jour;
    RETURN NEXT;
    SELECT 'Fête du travail'::text, (annee::text||'-05-01')::date
        INTO libelle, jour;
    RETURN NEXT;
    SELECT 'Victoire 1945'::text, (annee::text||'-05-08')::date
        INTO libelle, jour;
    RETURN NEXT;
    SELECT 'Fête nationale'::text, (annee::text||'-07-14')::date
        INTO libelle, jour;
    RETURN NEXT;
    SELECT 'Assomption'::text, (annee::text||'-08-15')::date
        INTO libelle, jour;
    RETURN NEXT;
    SELECT 'La toussaint'::text, (annee::text||'-11-01')::date
        INTO libelle, jour;
    RETURN NEXT;
    SELECT 'Armistice 1918'::text, (annee::text||'-11-11')::date
        INTO libelle, jour;
```

```
    RETURN NEXT;
    SELECT 'Noël'::text, (annee::text||'-12-25')::date INTO libelle, jour;
    RETURN NEXT;
    IF alsace_moselle THEN
      SELECT 'Vendredi saint'::text, paques(annee)::date - 2 INTO libelle, jour;
      RETURN NEXT;
      SELECT 'Lendemain de Noël'::text, (annee::text||'-12-26')::date
        INTO libelle, jour;
      RETURN NEXT;
    END IF;

    RETURN;
  END;
$function$;
```

La fonction s'utilise alors de façon simple :

```
SELECT *
FROM vacances(2020)
ORDER BY jour ;

     libelle     |    jour
-----------------+------------
 Jour de l'an    | 2020-01-01
 Pâques          | 2020-04-13
 Fête du travail | 2020-05-01
 Victoire 1945   | 2020-05-08
 Ascension       | 2020-05-21
 Fête nationale  | 2020-07-14
 Assomption      | 2020-08-15
 La toussaint    | 2020-11-01
 Armistice 1918  | 2020-11-11
 Noël            | 2020-12-25
```

Version avec `RETURNS TABLE` :

Seule la déclaration en début diffère de la version avec les paramètres `OUT` :

```
CREATE OR REPLACE FUNCTION vacances(
            annee integer,alsace_moselle boolean DEFAULT false)
 RETURNS TABLE (libelle text, jour date)
 LANGUAGE plpgsql
AS $function$
…
```

L'utilisation est aussi simple que la version précédente.

Version avec RETURN QUERY :

C'est peut-être la version la plus compacte :

```
CREATE OR REPLACE FUNCTION vacances(annee integer,alsace_moselle boolean DEFAULT false)
  RETURNS TABLE (libelle text, jour date)
  LANGUAGE plpgsql
  AS $function$
  BEGIN
    RETURN QUERY SELECT 'Jour de l''an'::text, (annee::text||'-01-01')::date ;
    RETURN QUERY SELECT 'Pâques'::text, paques(annee)::date + 1 ;
    RETURN QUERY SELECT 'Ascension'::text, ascension(annee)::date ;
    RETURN QUERY SELECT 'Fête du travail'::text, (annee::text||'-05-01')::date ;
    RETURN QUERY SELECT 'Victoire 1945'::text, (annee::text||'-05-08')::date ;
    RETURN QUERY SELECT 'Fête nationale'::text, (annee::text||'-07-14')::date ;
    RETURN QUERY SELECT 'Assomption'::text, (annee::text||'-08-15')::date ;
    RETURN QUERY SELECT 'La toussaint'::text, (annee::text||'-11-01')::date ;
    RETURN QUERY SELECT 'Armistice 1918'::text, (annee::text||'-11-11')::date ;
    RETURN QUERY SELECT 'Noël'::text, (annee::text||'-12-25')::date ;
    IF alsace_moselle THEN
      RETURN QUERY SELECT 'Vendredi saint'::text, paques(annee)::date - 2 ;
      RETURN QUERY SELECT 'Lendemain de Noël'::text, (annee::text||'-12-26')::date ;
    END IF;
    RETURN;
  END;
$function$;
```

NOTES

NOTES

NOTES

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.