
SGBD : Programmation et administration des bases de données [M2106]

Hocine ABIR

12 février 2014

IUT Villetaneuse
E-mail: abir@iutv.univ-paris13.fr

TABLE DES MATIÈRES

2	Procédures Stockées (2-PLpgSQL)	1
2.1	Introduction	1
2.2	Déclarations et Variables	4
2.3	Initialisation de Variables et Constantes	5
2.4	Commandes	5
2.5	Paramètres d'une fonction	13
2.6	Requête SQL dynamique	17
2.7	Exceptions	19

Procédures Stockées (2-PL_{pg}SQL)

2.1 Introduction

PL/pgSQL est un langage procédural structuré en blocs. Un *bloc* est une constitué de trois (3) sections :

1. DECLARE : déclarations des données (optionnel).
2. BEGIN : commandes de traitements (obligatoire).
3. EXCEPTION : commandes de gestion des erreurs (optionnel).

```
1  -- Forme general d'un bloc
2  DECLARE
3
4      -- definition des variables, constantes, types, curseurs,...
5
6  BEGIN
7
8      -- corps de la fonction (code)
9
10 EXCEPTION
11
12     -- gestion des erreurs
13
14 END;
```

Les blocs peuvent être imbriqués entre eux et/ou se suivre, comme dans l'exemple :

```
1  CREATE FUNCTION factoriel(int)
2  RETURNS INTEGER AS $$
3      -- Bloc 1
4      DECLARE
5          arg int;
6      BEGIN
7          arg := $1;
8          IF arg IS NULL OR arg < 0 THEN
9              RAISE NOTICE 'Invalid Number';
10             RETURN NULL;
11         ELSE
12             IF arg = 1 THEN
13                 RETURN 1;
14             ELSE
15                 -- Bloc 2
16                 DECLARE
17                     next_value INTEGER;
```

```

18         BEGIN
19             next_value := factoriel (arg - 1) * arg;
20             RETURN next_value;
21         END;
22     END IF;
23 END IF;
24 END;
25 $$ LANGUAGE 'plpgsql';

```

La fonction PL/pgSQL `factoriel` ci-dessus comporte deux blocs.

Les blocs déterminent la portée (ou visibilité) des variables.

```

1 CREATE FUNCTION nested_bloc()
2 RETURNS integer AS $$
3 DECLARE      --- Bloc -----
4     Variable integer := 1;
5 BEGIN
6     RAISE NOTICE 'Bloc 1 : Variable = %', Variable;
7
8     DECLARE  --- Bloc -----
9         Variable integer := 11;
10    BEGIN
11        RAISE NOTICE 'Bloc 11 : Variable = %', Variable;
12    END;
13
14    DECLARE  --- Bloc -----
15        Variable integer := 12;
16    BEGIN
17        RAISE NOTICE 'Bloc 12 : Variable = %', Variable;
18    END;
19
20    RAISE NOTICE 'Bloc 1 : Variable = %', Variable;
21    RETURN Variable;
22 END;
23 $$ LANGUAGE plpgsql;

```

```

(abir) [abir] => select nested_bloc();
NOTICE:  Bloc 1 : Variable = 1
NOTICE:  Bloc 11 : Variable = 11
NOTICE:  Bloc 12 : Variable = 12
NOTICE:  Bloc 1 : Variable = 1
 nested_bloc
-----
              1
(1 row)

```

2.1.1 Commande de création

```

1 CREATE [ OR REPLACE ] FUNCTION name
2 (
3     [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ]
4     [, ...] ]
5 )
6

```

```

7      [ RETURNS rettype |
8        RETURNS TABLE ( column_name column_type [, ...] ) ]
9
10 AS $$
11
12      Corps_de_la_Fonction
13
14      $$
15 LANGUAGE PLpgSQL
16
17 [ IMMUTABLE | STABLE | VOLATILE ]
18 [ CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT]
19 [ SECURITY INVOKER | SECURITY DEFINER]

```

Chaque fonction a un nom **name**. Un nom peut être surchargé, l'identification d'une fonction se fait donc par sa *signature* ou version **name ([argtype [, ...]])**.

2.1.2 Paramètres

Les paramètres sont définis uniquement par leur type **argtype** et sont automatiquement nommés **\$i** où **i** est le rang du paramètre dans la liste des paramètres, le premier est **\$1**. Les paramètres positionnels peuvent être renommés de deux façons :

1. par la déclaration **ALIAS FOR** :

```

-----
parameter_name ALIAS FOR $rang
-----

```

où **rang** est un entier désignant le rang du paramètre à renommer en **parameter_name**. La commande **ALIAS FOR** permet donc de donner des noms mnémoniques aux noms automatiques des paramètres.

2. par la commande **CREATE FUNCTION**

```

-----
CREATE FUNCTION name( ..., argname argtype ,....
-----

```

Exemple :

```

1 CREATE FUNCTION prix TTC (money, taxe real)
2   RETURNS money AS $$
3 DECLARE
4   prix ALIAS FOR $1;
5 BEGIN
6   RETURN prix + prix * taxe;
7 END;
8 $$ LANGUAGE plpgsql;

```

2.1.3 Commentaires :

Deux styles de commentaires :

1. **--** : double tiret introduit un commentaire sur le reste de la ligne.

2. `/* */` : commentaire type C pour plusieurs lignes.

```
1  -- cette ligne est un commentaire
2  /* Ces
3     lignes
4     sont des
5     commentaires
6  */
```

2.2 Déclarations et Variables

Toutes les variables utilisées dans un bloc doivent être déclarées dans la section déclaration du bloc (SAUF pour la boucle FOR).

2.2.1 Déclarations

Il y a plusieurs (4) façons d'introduire de nouvelles variables :

Paramètres Positionnels

Chaque paramètre définit une nouvelle variable. Dans l'exemple suivant sont définies deux variables \$1 de type `text` et \$2 de type `decimal(4,2)`.

```
1  CREATE or replace FUNCTION moyenne(text,decimal(4,2))
2  ...
```

ALIAS

Un paramètre peut recevoir un autre nom en utilisant la commande `ALIAS` .
Exemple :

```
1  DECLARE
2      Nom ALIAS FOR $1;
3      Moy ALIAS FOR $2;
```

DECLARE

Une nouvelle variable peut être définie dans la section `DECLARE` d'un bloc. Par exemple :

```
1  DECLARE
2      chaine      VARCHAR(8);
3      attribut    table.colonne%TYPE;
4      tuple       table%ROWTYPE;
5      indef       record;
6      ....
```

FOR

La commande d'itération **FOR** permet de déclarer automatiquement une variable entière. Dans l'exemple suivant, deux variables *i* sont définies.

```
1      DECLARE
2          i INT;
3      BEGIN
4          FOR i IN 1 .. 12 LOOP
5              ....
6          END LOOP;
7          ....
```

2.3 Initialisation de Variables et Constantes

```
1      DECLARE
2          nombre integer DEFAULT 23;
3          url varchar := 'http://monsite.fr';
4          Numero CONSTANT integer := 10;
5          ....
```

2.4 Commandes

Toutes les expressions utilisées en PL/pgSQL sont évaluées par le serveur (executor) en utilisant l'interface SPI (Server Programming Interface).

2.4.1 Affectation

* **SELECT** liste **INTO** destination ...;

Exemple :

```
1      CREATE TYPE carre AS
2          (n INT, carre INT);
3
4      CREATE FUNCTION carre(n INT)
5          RETURNS carre
6      AS $$
7      DECLARE
8          v_rec carre;
9      BEGIN
10         SELECT n, POWER(n,2)::INT INTO v_rec;
11         RETURN v_rec;
12     END;
13     $$
14     LANGUAGE 'plpgsql' IMMUTABLE;
```



```

SELECT * FROM carre(5);
n | carre
---+-----
5 |    25
(1 row)

```

* destination := appel_fonction_scalaire | expression;

Exemple :

```

1 CREATE or replace FUNCTION carre(n INT)
2   RETURNS carre
3 AS $$
4 DECLARE
5   v_rec carre;
6 BEGIN
7   v_rec:=(n, POWER(n,2)::INT);
8   RETURN v_rec;
9 END;
10 $$
11 LANGUAGE 'plpgsql' IMMUTABLE;

```

```

SELECT * FROM carre(5);
n | carre
---+-----
5 |    25
(1 row)

```

2.4.2 Alternatives IF-THEN-ELSE

```

1 IF expression_booleenne THEN
2   commandes
3 [
4 ELSIF expression_booleenne THEN
5   commandes
6 ]*
7 [
8 ELSE
9   commandes
10 ]
11 END IF;

```

Exemple :

```

1 CREATE FUNCTION equation2degres(a int, b int, c int)
2   RETURNS SETOF real AS
3 $$
4 DECLARE
5   d real;
6 BEGIN
7   IF a=0 and b=0 and c=0 THEN

```

```

8      raise info 'Tout reel est une solution de cette equation.';
9  ELSIF a=0 and b=0 THEN
10     raise info 'Cette equation ne possede pas de solutions.';
11 ELSIF a=0 THEN
12     raise info 'Cette equation est du premier degre.';
13     RETURN Next c/b::real;
14 ELSE
15     d := power(b,2) - 4.0*a*c;
16     IF d<0 THEN
17         raise info 'Cette equation n''a pas de solutions reelles.';
18     ELSIF d=0 THEN
19         raise info 'Cette equation a une seule solution reelle.';
20         RETURN Next -b/(2*a)::real;
21     ELSE
22         raise info 'Cette equation a deux solutions reelles.';
23         RETURN Next (-b+sqrt(d))/(2*a)::real;
24         RETURN Next (-b-sqrt(d))/(2*a)::real;
25     END IF;
26 END IF;
27 RETURN;
28 END;

```

```

=> select equation2degres(1,1,-2);
INFO: Cette équation a deux solutions réelles.
equation2degres
-----
              1
             -2
(2 rows)

=> select equation2degres(1,1,1);
INFO: Cette équation n'a pas de solutions reelles.
equation2degres
-----
(0 rows)

=> select equation2degres(4,4,1);
INFO: Cette équation a une seule solution réelle.
equation2degres
-----
             -0.5
(1 row)

```

2.4.3 Alternative CASE

CASE simple

Exemple :

```

1 CREATE OR REPLACE FUNCTION simple_case(INT4)
2 RETURNS TEXT as $$
3 BEGIN
4     CASE $1
5     WHEN 1,2 THEN RETURN 'UN ou DEUX';

```

```

6      WHEN 3 THEN RETURN 'TROIS';
7      ELSE RETURN 'ni UN ni DEUX ni TROIS';
8  END CASE;
9  END;
10 $$ language plpgsql;

```

```

=> select simple_case(1);
      simple_case

```

```

-----
UN ou DEUX
(1 row)

```

```

=> select simple_case(2);
      simple_case

```

```

-----
UN ou DEUX
(1 row)

```

```

=> select simple_case(3);
      simple_case

```

```

-----
TROIS
(1 row)

```

```

(abir) [abir] => select simple_case(4);
      simple_case

```

```

-----
ni UN ni DEUX ni TROIS
(1 row)

```

CASE de recherche

Exemple :

```

1  CREATE OR REPLACE FUNCTION search_case(INT4)
2  RETURNS TEXT as $$
3  BEGIN
4      CASE
5          WHEN $1 < 10 THEN RETURN 'inferieur a 10';
6          WHEN $1 = 10 THEN RETURN 'egal a 10';
7          ELSE RETURN 'superieur a 10';
8      END CASE;
9  END;
10 $$ language plpgsql;

```

```

=> select search_case(4);
      search_case
-----
inferieur a 10
(1 row)

=> select search_case(10);
      search_case
-----
egal a 10
(1 row)

=> select search_case(11);
      search_case
-----
superieur a 10
(1 row)

```

2.4.4 Itération

FOR - itération

```

-----
FOR iterator IN [ REVERSE ]
  start_expression .. end_expression LOOP
  statements
END LOOP;
-----

```

Exemple 1 :

```

1 CREATE FUNCTION simple_for(n INT4)
2 RETURNS setof int as $$
3 BEGIN
4   FOR i IN 1..n
5   LOOP
6     RETURN NEXT i;
7   END LOOP;
8   RETURN;
9 END;
10 $$ language plpgsql;

```

```

=> select simple_for(4);
      simple_for
-----
              1
              2
              3
              4
(4 rows)

```

Exemple 2 :

```

1 CREATE FUNCTION reverse_for(n INT4)
2 RETURNS setof int as $$
3 BEGIN
4     FOR i IN REVERSE n..1
5     LOOP
6         RETURN NEXT i;
7     END LOOP;
8     RETURN;
9 END;
10 $$ language plpgsql;

```

```

=> select reverse_for(4);
reverse_for
-----
         4
         3
         2
         1
(4 rows)

```

FOR - query result

```

-----
FOR iterator IN select-query LOOP
    statements
END LOOP;
-----

```

Exemple 1 :

```

1 CREATE FUNCTION query_for(n INT4)
2 RETURNS setof int as $$
3 DECLARE
4     i int;
5 BEGIN
6     FOR i IN SELECT generate_series(1,n)
7     LOOP
8         RETURN NEXT i;
9     END LOOP;
10    RETURN;
11 END;
12 $$ language plpgsql;

```

```

=> select query_for(4);
query_for
-----
         1
         2
         3
         4
(4 rows)

```

```

-----
FOR iterator IN EXECUTE query-string LOOP
    statements
END LOOP;
-----

```

Exemple 2 :

```

1 CREATE FUNCTION execute_for(n INT4)
2 RETURNS setof int as $$
3 DECLARE
4     i int;
5 BEGIN
6     FOR i IN EXECUTE
7         'SELECT generate_series(1,'||n||')'
8     LOOP
9         RETURN NEXT i;
10    END LOOP;
11    RETURN;
12 END;
13 $$ language plpgsql;

```

```

=> select execute_for(4);
   execute_for
-----
           1
           2
           3
           4
(4 rows)

```

WHILE LOOP

```

-----
WHILE bool_expression LOOP
    statements
END LOOP;
-----

```

Exemple :

```

1 CREATE FUNCTION while_loop(n INT4)
2 RETURNS setof int as $$
3 DECLARE
4     i int :=1; -- init cond
5 BEGIN
6     WHILE(i<=n)
7     LOOP
8         RETURN NEXT i;
9         i:=i+1; -- new cond
10    END LOOP;
11    RETURN;
12 END;
13 $$ language plpgsql;

```

```

=> select while_loop(4);
   while_loop
-----
           1
           2
           3
           4
(4 rows)

```

FOR - exit

```

-----
LOOP
    statements
    EXIT WHEN bool_expression;
END LOOP;
-----

```

Exemple :

```

1 CREATE FUNCTION exit_when(n INT4)
2 RETURNS setof int as $$
3 DECLARE
4     i int :=1; -- init cond
5 BEGIN
6     LOOP
7         RETURN NEXT i;
8         i:=i+1; -- new cond
9         EXIT WHEN i>4;
10    END LOOP;
11    RETURN;
12 END;
13 $$ language plpgsql;

```

```

=> select exit_when(4);
    exit_when
-----
           1
           2
           3
           4
(4 rows)

```

```

-----
LOOP
    IF bool_expression THEN
        EXIT;
    END IF;
    statements
END LOOP;
-----

```

Exemple :

```

1 CREATE FUNCTION exit_if(n INT4)
2 RETURNS setof int as $$
3 DECLARE
4     i int :=1; -- init cond
5 BEGIN
6     LOOP
7         IF i>4 THEN
8             EXIT;
9         END IF;
10        RETURN NEXT i;
11        i:=i+1; -- new cond
12    END LOOP;
13    RETURN;
14 END;
15 $$ language plpgsql;

```

```

=> select exit_if(4);
    exit_if
-----
           1
           2
           3
           4
(4 rows)

```

2.4.5 Commandes SQL

```

1 INSERT ... [ RETURNING expressions INTO [STRICT] target];
2 UPDATE ... [ RETURNING expressions INTO [STRICT] target];
3
4 DELETE ... [ RETURNING expressions INTO [STRICT] target];

```

Exemple 1 :

```

1 create table table1
2   ( field1 serial primary key,
3     field2 text not null
4   );
5
6 create function insere(p_val text)
7   returns integer as $$
8 declare
9   t_out integer;
10 begin
11   insert into table1(field2)
12     values ($1)
13   returning field1 into t_out;
14   return t_out;
15 end $$
16 language plpgsql;

```

```

=> select insere('toto');
insere
-----
      1
(1 row)

=> select insere('titi');
insere
-----
      2
(1 row)

=> select * from table1;
 field1 | field2
-----+-----
      1 | toto
      2 | titi
(2 rows)

```

Exemple 2 :

```

1 create table table2
2   (valeur int);
3
4 insert into table2
5   select generate_series(1,30);
6
7 create unction delete_pair(out mini int )
8 as
9 $$
10 declare
11   val int;
12 begin
13   mini=NULL;
14   FOR val in
15     delete from table2
16     where valeur%2=0
17     returning valeur
18   LOOP
19     if mini is null or
20       mini > val then
21       mini= val;
22     end if;
23   END LOOP;
24 end;
25 $$ language plpgsql;

```

```

=> select delete_pair();
delete_pair
-----
      2
(1 row)

=> select delete_pair();
delete_pair
-----
    NULL
(1 row)

```

2.5 Paramètres d'une fonction

Sous PL/pgSQL, on peut aussi définir des paramètres en spécifiant leur usage (ou mode). Le mode d'un paramètre détermine comment la fonction (ou procédure) peut utiliser et manipuler la valeur du paramètre.

On distingue trois modes IN, OUT, INOUT comme illustré par l'exemple suivant :


```

1  create table controle(
2      id          integer primary key,
3      nom         text,
4      note        decimal(4,2)
5  );
6
7  insert into controle values(1,'Robert',11)
8  insert into controle values(2,'Linda',12);
9  insert into controle values(3,'David',13);

```

```

1  CREATE FUNCTION note (
2      id IN int,nom INOUT text,note OUT decimal(4,2))
3  AS $body$
4  begin
5      SELECT controle.note into note
6          from controle
7          WHERE controle.id=id
8          and controle.nom=nom;
9  end;
10 $body$ LANGUAGE plpgsql;

```

```

=> select note(2,'Linda');
      note
-----
(Linda,12.00)
(1 row)

=> select * from  note(2,'Linda');
   nom | note
-----+-----
 Linda | 12.00
(1 row)

```

Paramètres IN

- permet de transmettre une valeur à la fonction,
- ne permet pas de retourner une valeur ,
- se comporte comme une constante (analogue à `#define CONST` en langage C) : ne peut être modifié.
- mode par défaut

Paramètres OUT

- permet de définir une valeur de retour,
- se comporte comme une variable *non initialisée* : valeur initiale indéterminée (NULL).
- sa valeur doit être déterminée par la fonction.

Paramètres INOUT

- permet à la fois de transmettre une valeur à la fonction, et de définir une valeur de retour,

- se comporte comme une variable *initialisée* : par la valeur transmise en argument.

```

1 CREATE FUNCTION inoutpar (
2     i IN int,j INOUT int ,x OUT
3     int)
4 AS $body$
5     begin
6         x:=i;  -- i : en lecture seule
7         x:=x*j;
8         j:=i+j; -- j : en lecture/ecriture
9     end;
$body$ LANGUAGE plpgsql

```

```

=> select inoutpar(2,3);
      inoutpar
-----
      (5,6)
      (1 row)

```

2.5.1 Table entière en paramètre

Exemple 1

```

1 CREATE FUNCTION concat_attr(tab controle)
2     RETURNS text
3 AS $body$
4 BEGIN
5     RETURN tab.id ||' '||tab.nom||' '||tab.note;
6 END;
7 $body$ LANGUAGE plpgsql;

```

```

=> select  concat_attr(controle.*) from controle;
      concat_attr
-----
1 Robert 11.00
2 Linda 12.00
3 David 13.00
(3 rows)

```

```

1 CREATE FUNCTION note (INOUT nom text,
2     OUT note decimal(4,2))
3 AS $body$
4 declare
5     tuple record;
6 begin
7     SELECT controle.note into note
8         from controle
9         WHERE controle.nom=nom;
10    RETURN;
11 end;
12 $body$ LANGUAGE plpgsql;

```

```

=> select note('Linda');
      note
-----
(Linda,12.00)
(1 row)

=> select * from note('Linda');
      nom | note
-----+-----
Linda | 12.00
(1 row)

```

Exemple 2

```

1 CREATE TYPE etudiant_rec as (
2     id          integer,
3     nom         text
4 );
5
6 drop table controle cascade;
7 create table controle(
8     etudiant     etudiant_rec,
9     note         decimal(4,2)
10 );
11
12 insert into controle values(ROW(1,'Robert'),11);
13 insert into controle values(ROW(2,'Linda'),12);
14 insert into controle values(ROW(3,'David'),13);

```

```

=> select * from controle;
      etudiant | note
-----+-----
(1,Robert) | 11.00
(2,Linda) | 12.00
(3,David) | 13.00
(3 rows)

```

```

1 CREATE FUNCTION ctrl (etudiant_rec ,
2                       OUT note decimal(4,2))
3 AS $body$
4 BEGIN
5     SELECT $1.nom, note from controle
6     WHERE controle.id(etudiant)=$1.id;
7 END;
8 $body$ LANGUAGE plpgsql;

```

```

=> select ctrl(ROW(1, 'Robert'));

```

2.6 Requête SQL dynamique

2.6.1 Définition

Une requête SQL dynamique est une requête générée (ou construite) par une fonction avant d'être exécutée. Les différentes clauses sont générées en fonction des paramètres de la procédure (requêtes à clauses variables).

La commande `EXECUTE` permet d'exécuter une requête dynamique.

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

`query-string` est une expression de type `text` ayant pour valeur la requête à exécuter. Si `query-string` est une requête d'interrogation `SELECT`, les résultats de la requête peuvent être récupérés :

- par la construction : `FOR-IN-EXECUTE`,
- ou en ajoutant la clause `into variable_name` à la commande `EXECUTE` dans le cas où la requête `SELECT` n'a qu'un seul résultat. La clause `STRICT` génère une erreur s'il y a plus d'un tuple.

La commande (chaîne) peuvent contenir des paramètres positionnels `$1`, `$2`, ... dont les valeurs seront fournies par la clause `using`.

2.6.2 Caractères d'échappement

#quotes	Utilisation	Exemple	Résultat
2	affectation	<code>x := "chaîne"</code>	<code>'chaîne'</code>
2	chaîne dans une clause	<code>WHERE x = "chaîne"</code>	<code>WHERE x = 'chaîne'</code>
4	simple quote dans une chaîne	<code>x=x " and y=" "chaîne" " "</code>	<code>... and y='chaîne'</code>

- `quote_ident(TEXT)` : variable contenant des attributs ou nom de table.
- `quote_literal(TEXT)` : variable contenant des chaînes littérales.

```
-----  
EXECUTE ''UPDATE tablename SET ''  
      || quote_ident(fieldname)  
      || '' = ''  
      || quote_literal(newvalue);  
-----
```

2.6.3 Exemples

```
1 create type comments AS  
2   (thing text,  
3    oname text,  
4    comment text  
5   );
```

```
1 create or replace function get_comment(text, text)
```

```

2  returns setof comments as
3  ,
4  DECLARE
5      ret comments%ROWTYPE;
6      rec RECORD;
7      objtype alias for $1;
8      iname alias for $2;
9      tbl text;
10     qry text;
11     types text;
12 BEGIN
13     ret.thing := objtype;
14     IF objtype = 'table'
15     THEN
16         tbl := 'pg_class';
17
18         qry := ' select relname as oname , obj_description( p.oid, '' ||
19             quote_literal(tbl) ||
20             '') as comm from pg_class p where relname = '' ||
21             quote_literal(iname) || ''';
22     ELSIF objtype = 'function'
23     THEN
24         tbl := 'pg_proc';
25         qry := ' select proname ||'' || '' '' ''('' '' ||'' ||
26             ''oidvectortypes(proargtypes) ''||
27             ''|| '' '' '' '' as oname, obj_description( oid, '' ||
28             quote_literal(tbl) ||
29             '' ) as comm from '' ||
30             quote_ident(tbl)
31             || '' where proname =''
32             || quote_literal(iname) || ''';
33     ELSE
34         RAISE EXCEPTION ''USAGE: get_comment(''table |
35             function'', object_name )'';
36     END IF;
37     FOR rec IN EXECUTE qry
38     LOOP
39         ret.oname = rec.oname;
40         ret.comment = rec.comm;
41         RETURN NEXT ret;
42     END LOOP;
43     RETURN ;
44 END;
45 ' LANGUAGE plpgsql;

```

```

-----
select * from get_comment('function' , 'date') as t(x,y);
      x      |              y              |              comment
-----+-----+-----
function | date(text)                  | convert text to date
function | date(abstime)               | convert abstime to date
function | date(timestamp without time zone) | convert timestamp to date
function | date(timestamp with time zone)   | convert timestamp with time zone
                                     to date
(4 lignes)

COMMENT ON FUNCTION get_comment(text,text) IS
  'get_comment( ''table | function'', object_name )';
COMMENT

select * from get_comment('function' , 'get_comment') as t(x,y);
      x      |              y              |              comment
-----+-----+-----
function | get_comment(text, text) | get_comment( 'table | function', object_name )
(1 ligne)

select * from get_comment('func' , 'date') as t(x,y);
ERREUR:  USAGE: get_comment( 'table | function', object_name )
-----

```

2.7 Exceptions

2.7.1 Introduction

Par défaut, si une fonction PL/pgSQL produit une erreur, celle-ci est avortée, et la transaction qui l'a exécutée est avortée aussi.

Il est possible d'intercepter ces erreurs et d'effectuer une reprise sur erreur en utilisant un bloc **BEGIN** avec une clause **EXCEPTION**.

La syntaxe est une extension de la syntaxe normal du bloc **BEGIN** vers une structure de type **try-catch** :

```

1 BEGIN
2   -- traitement ...
3 EXCEPTION
4   WHEN condition [OR condition ] THEN
5     -- Reprise erreur i...
6   WHEN condition [OR condition ] THEN
7     -- Reprise erreur j...
8 END;
```

Exemple :

```

1 CREATE FUNCTION except_intro(x int,Y int)
2 RETURNS setof int AS
3 $$
4 BEGIN
5   RETURN NEXT(x);
```

```

6      INSERT INTO t values(x);
7  BEGIN -- debut sous bloc
8      RETURN NEXT(x+y);
9      INSERT INTO t values(x+y);
10     RETURN NEXT(x/y);
11     INSERT INTO t values(x/y);
12     RETURN NEXT(x-y);
13     INSERT INTO t values(x-y);
14 EXCEPTION
15     WHEN division_by_zero THEN
16         raise notice 'division par zero';
17 END; -- fin sous bloc
18 RETURN NEXT(y);
19 INSERT INTO t values(y);
20 RETURN;
21 END;
22 $$ LANGUAGE plpgsql;

```

```

=> select * from t;
c
---
(0 rows)

```

```

=> select * from tt;
c
---
(0 rows)

```

```

=> insert into tt select except_intro(6,0);
NOTICE: division par zero
INSERT 0 3
r=> select * from tt;
c
---
6
6
0
(3 rows)

```

```

=> select * from t;
c
---
6
0
(2 rows)

```

Pas d' Erreur durant le "traitement"

- Toutes les commandes de **traitement** sont exécutées
- Le controle est ensuite transféré à la première commande après **END** : la clause **EXCEPTION** est ignorée.

Une Erreur est survenu durant le "traitement"

- L'exécution des commandes de **traitement** est suspendue,
- Les commandes de **traitement** sont annulées : **ROLLBACK**,
- Les variables locales de la fonction PL/pgSQL reste comme elles étaient au moment de l'erreur.
- Le controle est ensuite transféré à la clause **EXCEPTION**,
- La première condition qui correspond à l'erreur est recherchée : l'erreur détectée est comparée successivement aux listes de conditions **WHEN**.
- Si l'erreur correspond à une **condition** d'une clause **WHEN**, les commandes **Reprise erreur** associées sont exécutées, et le controle est ensuite transféré à la première commande après **END**.
- Si l'erreur ne correspond à aucune **condition** des clauses **WHEN**, l'erreur est propagée vers le bloc parent.
- S'il l'erreur ne peut être traitée alors la fonction est avortée.

Si une nouvelle erreur se produit durant la reprise, l'erreur est propagée (ne peut être intercepter par la clause **EXCEPTION** dans laquelle l'erreur est produite).

2.7.2 Message d'erreur

```
1 CREATE FUNCTION except_disp(x int,Y int)
2 RETURNS boolean AS
3 $$
4 BEGIN
5     INSERT INTO t VALUES(x/y);
6     return true;
7 EXCEPTION
8     WHEN division_by_zero THEN
9         raise notice
10 E'\n\tSQLERRM = %, \n\tSQLSTATE = %', SQLERRM, SQLSTATE;
11     return false;
12 END;
13 $$ LANGUAGE plpgsql;
```

```
r=> select except_disp(6,0);
NOTICE:
SQLERRM  = division by zero,
SQLSTATE = 22012
except_disp
-----
f
(1 row)
```

2.7.3 Comment les exceptions se propagent

```
1 CREATE FUNCTION except_prog(x INT4)
2 RETURNS text as
3 $$
4 BEGIN
5     BEGIN
6         IF x=1 THEN
```



```

7      RAISE USING ERRCODE ='sql_routine_exception';
8  ELSIF x=2 THEN
9      RAISE USING ERRCODE ='data_exception';
10     ELSE
11         RAISE 'Valeur x : %', x
12         USING ERRCODE ='plpgsql_error';
13     END IF;
14 EXCEPTION
15     WHEN sql_routine_exception THEN
16         RETURN 'SQL Routine Exception , x = ' || x;
17 END;
18 EXCEPTION
19     WHEN data_exception THEN
20         RETURN 'DATA EXCEPTION , x = ' || x;
21 END;
22 $$ language plpgsql;

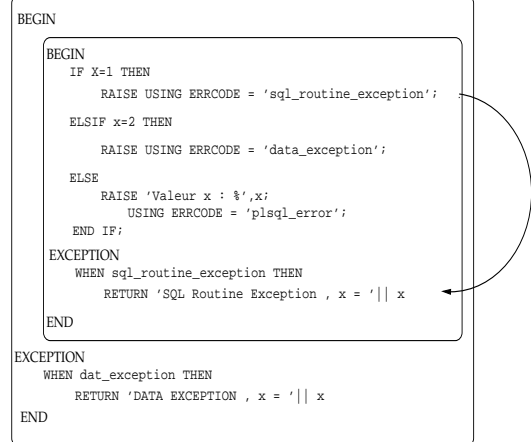
```

Cas 1

```

=> select except_prog(1);
      except_prog
-----
SQL Routine Exception , x = 1
(1 row)

```

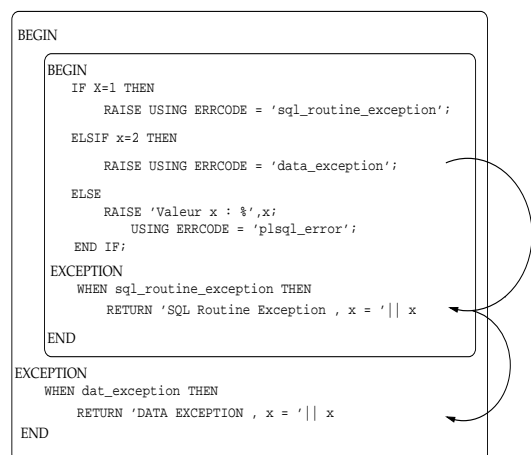


Cas 2

```

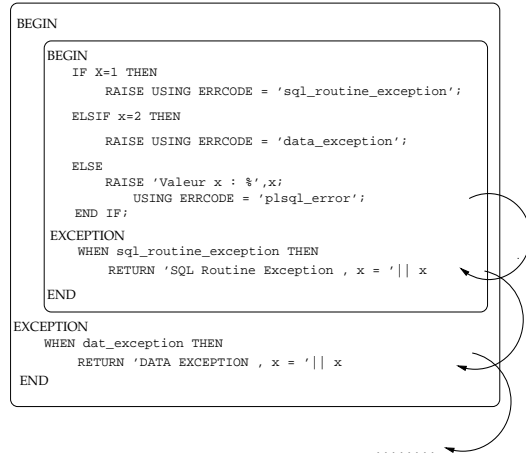
=> select except_prog(2);
      except_prog
-----
DATA EXCEPTION , x = 2
(1 row)

```



Cas 3

```
=> select except_prog(3);
ERROR:  Valeur x : 3
```



2.7.4 Propager une exception (reraising)

```

1 CREATE FUNCTION except_raise() returns void AS $$
2 BEGIN
3   BEGIN
4     RAISE syntax_error;
5   EXCEPTION
6     WHEN syntax_error THEN
7     BEGIN
8       raise notice
9       'syntax_error : bloc interne, reraising';
10      RAISE;
11    END;
12  END;
13 EXCEPTION
14   WHEN syntax_error THEN
15     raise notice
16     'syntax_error : bloc externe';
17 END;
18 $$ LANGUAGE plpgsql

```

```
=> select except_raise();
NOTICE: syntax_error : bloc interne, reraising
NOTICE: syntax_error : bloc externe
except_raise
-----
(1 row)
```

2.7.5 Gestion d'une exception

Annulation de l'opération

```
1 CREATE FUNCTION except_cont(x int,Y int)
```

```

2 RETURNS boolean AS
3 $$
4 begin
5     INSERT INTO t VALUES(x);
6     INSERT INTO t VALUES(x/y);
7     INSERT INTO t VALUES(y);
8     return true;
9 exception
10     WHEN division_by_zero THEN
11         raise notice 'division par zero';
12         return false;
13 end;
14 $$ LANGUAGE plpgsql;

```

```

=> select * from t;
      c
----
(0 rows)

=> select except_cont(6,0);
NOTICE: division par zero
except_cont
-----
      f
(1 row)

=> select * from t;
      c
----
(0 rows)

```

Reprise de l'opération

```

1 CREATE FUNCTION except_trans(x int,Y int)
2 RETURNS boolean AS
3 $$
4 BEGIN
5     LOOP
6         BEGIN -- debut sous bloc
7             INSERT INTO t VALUES(x);
8             INSERT INTO t VALUES(x/y);
9             INSERT INTO t VALUES(y);
10            return true;
11        EXCEPTION
12            WHEN division_by_zero THEN
13                raise notice 'division par zero';
14                y=1; -- reprise avec y!=0
15        END; -- fin sous bloc
16    END LOOP;
17 END;
18 $$ LANGUAGE plpgsql;

```

```

=> select * from t;
 c
---
(0 rows)

=> select except_trans(6,0);
NOTICE:  division par zero
 except_trans
-----
 t
(1 row)

=> select * from t;
 c
---
 6
 6
 1
(3 rows)

```

2.7.6 Codes d'erreur "Postgres"

Tous les messages d'erreurs émis par le server **Postgres** sont identifiés par un code d'erreur à 5 caractères (SQL standard's conventions for "SQLSTATE" codes) :

- les 2 premiers caractères : désigne une classe d'erreurs,
- les 3 derniers caractères : indique l'erreur spécifique à la classe.

Une application qui désire savoir quelle erreur elle a généré, doit tester ce code (plutôt que le message textuel).

Quelques exemples :

Class 23 : Integrity Constraint Violation

Class 40 : Transaction Rollback

Class P0 : PL/pgSQL Error

Code Erreur	Signification	Constant
23000	INTEGRITY CONSTRAINT VIOLATION	integrity_constraint_violation
23001	RESTRICT VIO- LATION	restrict_violation
23502	NOT NULL VIO- LATION	not_null_violation
23503	FOREIGN KEY VIOLATION	foreign_key_violation
23505	UNIQUE VIOLA- TION	unique_violation
23514	CHECK VIOLA- TION	check_violation
40000	TRANSACTION ROLLBACK	transaction_rollback
40002	TRANSACTION INTEGRITY CONSTRAINT VIOLATION	transaction_integrity_constraint_violation
40001	SERIALIZATION FAILURE	serialization_failure
40003	STATEMENT COMPLETION UNKNOWN	statement_completion_unknown
40P01	DEADLOCK DE- TECTED	deadlock_detected

Code Erreur	Signification	Constant
P0000	PLPGSQL ER- ROR	plpgsql_error
P0001	RAISE EXCEP- TION	raise_exception
P0002	NO DATA FOUND	no_data_found
P0003	TOO MANY ROWS	too_many_rows

Une condition spéciale appelée **OTHERS** satisfait tout type d'erreur sauf **QUERY_CANCELED**.

Dans la clause **EXCEPTION**, deux variables locales sont accessibles :

1. **SQLSTATE** : contient le code d'erreur correspondant à l'exception,
2. **SQLERRM** : contient le message d'erreur correspondant à l'exception.

(Ces variables sont inaccessibles à l'extérieur de la clause **EXCEPTION**.)

2.7.7 Exemple

```
create table ip_host (
    ip      text PRIMARY KEY,
    host    text
);

CREATE FUNCTION ins_iphost( ip text, host text)
RETURNS integer AS '
DECLARE
    rcount integer;
BEGIN
    BEGIN
        INSERT INTO ip_host VALUES (ip, host);
    EXCEPTION
        WHEN UNIQUE_VIOLATION THEN
            RAISE NOTICE
                ''Duplicate Key % for % ignored.'', ip, host;
    END;
    GET DIAGNOSTICS rcount = ROW_COUNT;
    RETURN rcount;
END; ' LANGUAGE 'plpgsql';

# begin;
BEGIN

## select  ins_iphost( '194.254.173.155','my_host');
ins_iphost
-----
          1
(1 row)

## select  ins_iphost( '194.254.173.155','my_host');
NOTICE:  Duplicate Key 194.254.173.155 for my_host ignored.
ins_iphost
-----
          0
(1 row)

## end;
COMMIT

# select * from ip_host;
      ip      | host
-----+-----
194.254.173.155 | my_host
(1 row)
```

2.7.8 Erreurs et Messages

```
RAISE level 'format' [, expression [, ...]];
```

où `level` peut être : `DEBUG`, `LOG`, `INFO`, `NOTICE`, `WARNING` ou `EXCEPTION`.

Seul `EXCEPTION` avorte (normalement) la transaction. `format` est une chaîne pouvant comporter le symbole de substitution `%`.