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

Hocine ABIR

25 janvier 2014

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

TABLE DES MATIÈRES

1	Procédures Stockées (1-SQL)	1
1.1	Introduction	1
1.2	Programmation SQL : Rupture de séquence et Code déclaratif . . .	2
1.3	Caractéristiques des Fonctions en SQL	7
1.4	Modes Opératoires des Fonctions SQL	12
1.5	Tableau	19
1.6	Procédures Stockées et Catalogue	21

Procédures Stockées (1-SQL)

1.1 Introduction

1.1.1 procédure vs Fonctions

Procédure

Une procédure stockée est un des nombreux mécanismes d'encapsulation de la logique (sémantique) dans une base de données.

Une procédure stockée est similaire à une procédure dans les langages de programmation usuels c'est à dire qu'elle :

- prend des arguments
- effectue un traitement
- retourne éventuellement un résultat et peut éventuellement modifier les valeurs de ses arguments

Les procédures stockées ne peuvent pas être utilisées dans des requêtes à cause des arguments passés par référence (plusieurs résultats)

Fonction

Fonctions stockées sont similaires aux procédures mais :

1. peuvent être utilisées dans les requêtes, fonctions, procédures, et vues.
2. dans certains SGBD, elles ne peuvent pas modifier les données ou ont des limitations au niveau ddl/dml.
3. en général, elles ne prennent pas des arguments passés par référence.

Sous PostgreSQL (comme dans le langage C), il n'y a pas de distinction entre procédures et fonctions.

1.1.2 Languages et Fonctions

Plusieurs langages peuvent être disponibles :

```
1 SELECT tmplname as "Name",  
2     Case when (tmpldbcreate)  
3         then 'dba'  
4         else 'db owner'  
5 end as "CREATE"
```

```
6 FROM pg_pltemplate;
```

Name	CREATE
plpgsql	dba
pltcl	dba
pltclu	db owner
plperl	dba
plperl	db owner
plpythonu	db owner

(6 rows)

Certains sont déjà installés :

```
1 SELECT lanname,
2     CASE WHEN lanispl
3         THEN 'User defined'
4         ELSE 'Internal'
5     END AS lanispl,
6     CASE when not lanpltrusted
7         then 'Not Trusted'
8     end as lanpltrusted
9 FROM pg_language;
```

lanname	lanispl	lanpltrusted
internal	Internal	Not Trusted
c	Internal	Not Trusted
sql	Internal	
plpgsql	User defined	

(4 rows)

1.2 Programmation SQL : Rupture de séquence et Code déclaratif

Les langages de programmation structurée classiques utilisent deux concepts pour mettre en oeuvre une rupture de séquence :

1. IF-THEN-ELSE : alternative
2. WHILE-DO : itération

Dans ce qui suit, nous allons donner un aperçu sur l'implémentation de ces deux concepts en SQL.

1.2.1 Contrôle de Séquence : Case

- La clause SQL CASE est une clause CASE de recherche (par opposition à une clause CASE simple).

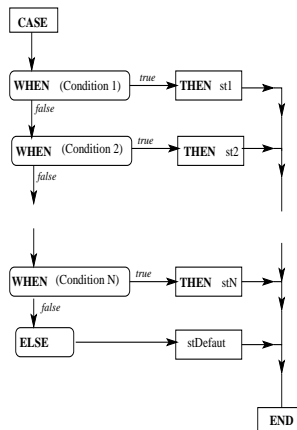
- La clause SQL CASE permet d'introduire un peu de logique complexe dans les clauses SELECT, WHERE etc ..
- Dans une clause CASE, l'ordre des clauses WHEN est important.
- La clause finale ELSE est facultative : il y a toujours une implicite clause ELSE NULL si la clause ELSE n'est pas définie.

```

1 CASE
2   WHEN Condition1 THEN st1
3   [
4     WHEN Condition2 THEN st2
5     ...
6     WHEN ConditionN THEN stN
7   ]
8   [
9     ELSE stDefault
10  ]
11 END

```

Expression Conditionnelle CASE



Exemple 1 : Une requête au lieu de deux

```

1 SELECT SUM(CASE WHEN i%2=0 then i else 0 end)
2         "Somme des Nombres Paires",
3         SUM(CASE WHEN i%2=0 then 0 else i end)
4         "Somme des Nombres Impaires"
5 FROM generate_series(1,5) t(i);

```

	Somme des Nombres Paires	Somme des Nombres Impaires
(1 row)	6	9

Exemple 2 : Jour de Semaine

```

1 --Zeller's congruence
2 -- 14/11/2023
3 \set j '14'
4 \set m '11'
5 \set a '2013'
6
7 SELECT CASE
8     WHEN j=0 THEN 'Dimanche'
9     WHEN j=1 THEN 'Lundi'
10    WHEN j=2 THEN 'Mardi'
11    WHEN j=3 THEN 'Mercredi'
12    WHEN j=4 THEN 'Jeudi'
13    WHEN j=5 THEN 'Vendredi'
14    WHEN j=6 THEN 'Samedi'
15 END "Jour"

```

```

16 FROM (SELECT (:j+ya+ ya/4 -2*sa + sa/4 + (26*m1-2)/10)%7
17        FROM (SELECT m1,a1,a1/100 sa, a1%100 ya
18              FROM (SELECT CASE WHEN(:m<3) THEN :m+10 else :m-2 end)
19                    tm1(m1),
20                    (SELECT CASE WHEN(:m<3) THEN :a-1 else :a end)
21                    ta1(a1)
22              ) f
23        ) jour(j);

```

```

Jour
-----
Jeudi
(1 row)

```

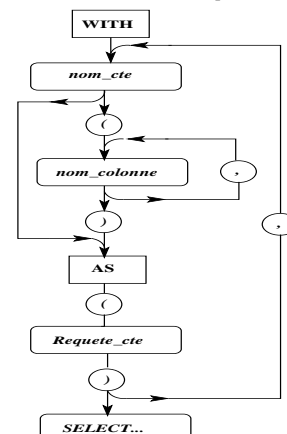
1.2.2 Itération : Common Table Expression

```

1 WITH [ RECURSIVE ]
2   nom_cte
3   [ ( nom_colonne [, ...] ) ]
4   AS ( requete_cte )
5   SELECT ...

```

WITH Common Table Expression



Exemple 1 : Forme élémentaire

```

1 create table t(i int);
2 insert into t (select generate_series(1,3));
3
4 WITH ctetab(colonne) AS      -- clause WITH
5   ( SELECT i from t)
6   SELECT * FROM ctetab;      -- clause SELECT

```

```

colonne
-----
1
2
3
(3 rows)

```

La requête ci-dessus est constituée de deux clauses :

- la clause **WITH** : utilise une table temporaire (ou Common Table Expression) **ctetab** ayant un attribut **colonne** et constitué des tuples **t.i** de la table **t**.
- la clause **SELECT** : lit séquentiellement la table **ctetab**.

```

                                QUERY PLAN
-----
CTE Scan on ctetab  (cost=1.03..1.09 rows=3 width=4)
  CTE ctetab
    -> Seq Scan on t  (cost=0.00..1.03 rows=3 width=4)
(3 rows)

```

Exemple 2 (récursion) : Boucle

```

1  WITH RECURSIVE boucle (i) AS
2  (
3      SELECT 1 -- valeur initiale
4      UNION
5      SELECT i + 1 -- pas
6      FROM boucle
7      WHERE i < 4 -- condition fin iteration
8  )
9  SELECT * FROM boucle;

```

```

i
---
1
2
3
4
(4 rows)

```

Comment ça marche :

```

                                QUERY PLAN
-----
CTE Scan on boucle  (cost=2.96..3.58 rows=31 width=4)
  CTE boucle
    -> Recursive Union (cost=0.00..2.96 rows=31 width=4)
      -> Result  (cost=0.00..0.01 rows=1 width=0)
      -> WorkTable Scan on boucle  (cost=0.00..0.23 rows=3 width=4)
          Filter: (boucle.i < 4)
(6 rows)

```

Exemple 3 : Dessin d'art ASCII


```

1 WITH RECURSIVE pyramide (h) AS
2 (
3     SELECT 1
4     UNION
5     SELECT h + 1
6         FROM pyramide
7         WHERE h < 4
8 )
9 SELECT repeat(' ', 4-h) ||
10     repeat('*', 2*h-1) as "Pyramide"
11 FROM pyramide;

```

```

Pyramide
-----
      *
     ***
    *****
   ********
(4 rows)

```

Exemple 4 : (union all) Facteurs premiers

```

1 -- Facteurs premiers d'un entier N
2
3 WITH RECURSIVE Decomposition (facteur, Nombre, est_facteur) AS
4 (
5     SELECT 2, 60, false
6     UNION ALL
7     SELECT
8         CASE WHEN Nombre % facteur = 0 THEN facteur
9             WHEN facteur * facteur > Nombre THEN Nombre
10            WHEN facteur = 2 THEN 3
11            ELSE facteur + 2
12         END,
13         CASE WHEN Nombre % facteur = 0 THEN Nombre / facteur
14            ELSE Nombre
15         END,
16         CASE WHEN Nombre % facteur = 0 THEN true
17            ELSE false
18         END
19     FROM Decomposition
20     WHERE Nombre <> 1
21 )
22 SELECT facteur, Nombre
23     FROM Decomposition
24     WHERE est_facteur;

```

facteur	nombre
2	30
2	15
3	5
5	1

(4 rows)

1.3 Caractéristiques des Fonctions en SQL

```

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 SQL
16
17 [ IMMUTABLE | STABLE | VOLATILE ]
18 [ CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT]
19 [ SECURITY INVOKER | SECURITY DEFINER]

```

1.3.1 Introduction

Les fonctions SQL permettent d'exécuter une liste arbitraire de commandes SQL, et retournent le résultat de la dernière commande de cette liste.

La dernière commande doit être une commande SELECT ou une commande INSERT, DELETE, UPDATE ayant une clause RETURNING.

```

1 CREATE table tab(
2   id    serial
3       primary key,
4   b     int
5 );
6
7 create or replace function ajout
8 (   bvalue int
9 ) returns int as
10 $$

```

```

11      insert into tab (id,b) values (default,$1)
12      returning id
13  $$ language sql;
14
15  select ajout(8);
16  select ajout(9);
17  select * from tab;

```

```
ajout
```

```
-----
```

```
      1
(1 row)
```

```
ajout
```

```
-----
```

```
      2
(1 row)
```

```
id | b
```

```
----+---
```

```
  1 | 8
```

```
  2 | 9
```

```
(2 rows)
```

- Fonction Scalaire :

Les fonctions scalaires retournent une "seule" valeur de donnée dont le type est défini dans la clause RETURNS (ou paramètres OUT).

```

1 CREATE FUNCTION scal_func(int,out a int,out b int)
2 AS
3 $$
4     SELECT $1, $1+1;
5 $$ LANGUAGE sql;
6
7 SELECT * FROM scal_func(4);
8 SELECT (scal_func(4)).b;

```

```
a | b
```

```
----+---
```

```
  4 | 5
```

```
(1 row)
```

```
b
```

```
---
```

```
  5
```

```
(1 row)
```

- Fonction Ensemble : SETOF

```

1 CREATE FUNCTION setof_func(a int,OUT b int, OUT c int )
2 RETURNS SETOF record AS
3 $$
4     SELECT i, i+1 FROM generate_series(1, $1) g(i);
5 $$ LANGUAGE sql;
6

```

```

7 SELECT * FROM setof_func(4);
8 SELECT (setof_func(4)).b;

```

```

  b | c
---+---
  1 | 2
  2 | 3
  3 | 4
  4 | 5
(4 rows)

```

```

  b
---
  1
  2
  3
  4
(4 rows)

```

- Fonction Table :

Les fonctions table retournent un type de données table. la table est le résultat d'une commande SELECT (implique SETOF).

```

1 CREATE FUNCTION tab_func(a int)
2   RETURNS TABLE(b int, c int) AS
3 $$
4   SELECT i, i+1 FROM generate_series(1, $1) g(i);
5 $$ LANGUAGE sql;
6
7 SELECT * FROM tab_func(4);
8 SELECT (tab_func(4)).b;

```

```

  b | c
---+---
  1 | 2
  2 | 3
  3 | 4
  4 | 5
(4 rows)

```

```

  b
---
  1
  2
  3
  4
(4 rows)

```

1.3.2 Classification Comportementale

Volatilité

- VOLATILE (défaut) : Chaque appel peut retourner un résultat différent. Les fonctions qui modifient des tables doivent être déclarées VOLATILE.

- STABLE : Chaque appel retourne le même résultat *dans une même requête (transaction)*
- IMMUTABLE : Chaque appel retourne le même résultat *pour les mêmes arguments*.

```

1 WITH RECURSIVE boucle (i,"Volatilite","Stabilite","Immutabilite") AS
2 (
3     SELECT 1 ,random() ,now()::time,pi()
4     UNION
5     SELECT i + 1,random() ,now()::time,pi()
6     FROM boucle
7     WHERE i < 4
8 )
9 SELECT * FROM boucle;

```

i	Volatilite	Stabilite	Immutabilite
1	0.159957596100867	07:21:17.820671	3.14159265358979
2	0.736195099074394	07:21:17.820671	3.14159265358979
3	0.750766322482377	07:21:17.820671	3.14159265358979
4	0.491170474328101	07:21:17.820671	3.14159265358979

(4 rows)

```

1 SELECT pronom, provolatile
2 FROM pg_proc
3 WHERE pronom IN ('random','now','pi');

```

pronom	provolatile
now	s
random	v
pi	i

(3 rows)

Argument indéfini : NULL

- CALLED ON NULL INPUT (default)
- RETURNS NULL ON NULL INPUT / STRICT

```

1 CREATE FUNCTION add1 (int, int)
2 RETURNS int AS
3 $$
4     SELECT $1 + $2
5 $$ LANGUAGE SQL
6 RETURNS NULL ON NULL INPUT;
7
8 CREATE FUNCTION add2 (int, int)
9 RETURNS int AS
10 $$
11     SELECT COALESCE($1, 0) + COALESCE($2, 0)
12 $$ LANGUAGE SQL ;

```

```
(abir) [demodb] => SELECT add1(3, NULL) , add2(3, NULL);
add1 | add2
-----+-----
NULL |    3
(1 row)
```

```
1 SELECT pronom,
2     CASE WHEN pronom IS NULL
3         THEN 'RETURNS NULL ON NULL INPUT / STRICT'
4         ELSE 'CALLED ON NULL INPUT (default)'
5     END "NULL INPUT"
6 FROM pg_proc
7 WHERE pronom IN ('add1','add2');
```

```
pronom | NULL INPUT
-----+-----
add2    | CALLED ON NULL INPUT (default)
add1    | RETURNS NULL ON NULL INPUT / STRICT
(2 rows)
```

Sécurité

- SECURITY INVOKER (default)
- SECURITY DEFINER

```
1 CREATE TABLE notes
2 (
3     Nom varchar,
4     note decimal(4,2)
5 );
6 INSERT INTO notes VALUES
7     ('toto',12.5),('guest',13.25),('titi',9.5);
8 -----
9 CREATE or replace FUNCTION notes()
10 RETURNS SETOF notes AS
11 $$
12     SELECT * FROM notes;
13 $$ LANGUAGE SQL SECURITY DEFINER;
```

```
(abir) [demodb] => select * from notes;
  nom | note
-----+-----
  toto | 12.50
  guest | 13.25
  titi | 9.50
(3 rows)

(abir) [demodb] => \c - guest
Password for user guest:
psql (8.4.18)
You are now connected to database "demodb" as user "guest".

(guest) [demodb] => select * from notes;
ERROR:  permission denied for relation notes

(guest) [demodb] => select * from notes();
  nom | note
-----+-----
  toto | 12.50
  guest | 13.25
  titi | 9.50
(3 rows)
```

1.4 Modes Opérateurs des Fonctions SQL

1.4.1 Fonction Normale

```
(abir) [demodb] => \df add*
          List of functions
 Schema | Name | Result data type | Argument data types | Type
-----+-----+-----+-----+-----
 public | add1 | integer           | integer, integer    | normal
 public | add2 | integer           | integer, integer    | normal
(2 rows)
```

Arguments par Défaut

```
1 create function add12
2   ( int ,
3     int default 1
4   ) returns int as
5   $$
6     select $1 + $2;
7   $$ language sql;
8
9 select add12(2,4);
10 select add12(2);
```

```

add12
-----
      6
(1 row)

add12
-----
      3
(1 row)

```

1.4.2 Fonction d'Aggrégation

```

(abir) [demodb] => \df pmoy
                List of functions
 Schema | Name | Result data type | Argument data types | Type
-----+-----+-----+-----+-----
 public | pmoy | numeric          | numeric, numeric    | agg
(1 row)

```

Création

```

1 CREATE AGGREGATE name
2 (
3     input_data_type
4     [ , ... ]
5 )
6 (
7     SFUNC = state_func,
8     STYPE = state_data_type
9     [ , FINALFUNC = final_func ]
10    [ , INITCOND = initial_condition ]
11    [ , SORTOP = sort_operator ]
12 )

```

Exemple

```

1 -- etat
2 CREATE TYPE state AS
3 (
4     notes    decimal,
5     coefs    decimal
6 );
7
8 -- somme pondere et somme coefficients
9 CREATE FUNCTION spmoy (INOUT state,IN decimal,IN decimal)
10 AS $$
11     SELECT $1.notes+COALESCE($2,0)*COALESCE($3,0),
12            $1.coefs+COALESCE($3,0);
13 $$ LANGUAGE sql;

```



```

14
15 -- moyenne pondere
16 CREATE FUNCTION fpmoy (state)
17     RETURNS decimal AS $$
18     SELECT $1.notes / $1.coefs;
19 $$ LANGUAGE sql;
20
21 -- pmoy (notes,coefficients) retourne moyenne ponderee
22 CREATE AGGREGATE pmoy (decimal,decimal)
23 (
24     STYPE = state,
25     SFUNC = spmoy,
26     FINALFUNC = fpmoy,
27     INITCOND = '(0,0)'
28 );

```

Comment ça Marche

```

1 CREATE TABLE evaluations
2 (
3     nom          varchar,
4     matiere      varchar,
5     note         numeric,
6     coeff        numeric
7 );
8 INSERT INTO evaluations VALUES
9 ('toto','MI41',15,2),('titi','MI41',13,2),
10 ('titi','MI43',10,3),('toto','MI43',10,3),
11 ('toto','MI42',6,1);

```

```

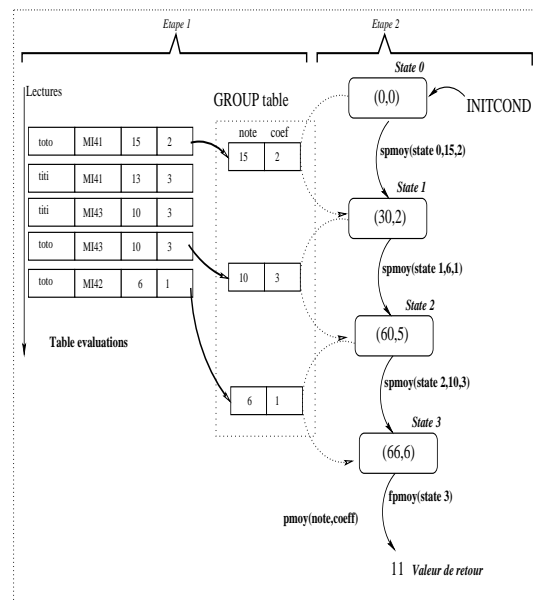
1 SELECT pmoy(note,coeff) ::
2     decimal(4,2) "Toto"
3 FROM evaluations
4 WHERE nom='toto';

```

```

Toto
-----
11.00
(1 row)

```



Clause GROUP BY

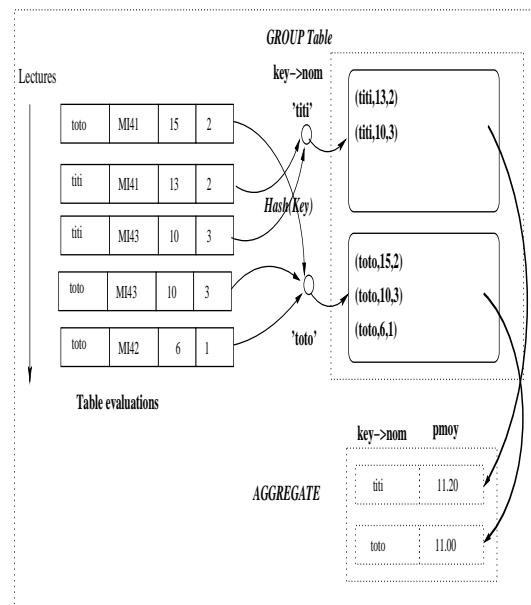
```

1 SELECT nom ,pmoy(note,coeff) ::
2     decimal(4,2) "Moy"
3 FROM evaluations
4 GROUP BY nom;

```

nom	Moy
titi	11.20
toto	11.00

(2 rows)



1.4.3 Fonction "Windows"

```
(abir) [demodb] => \df row_number
```

List of functions

Schema	Name	Result data type	Argument data types	Type
pg_catalog	row_number	bigint		window

Clause OVER

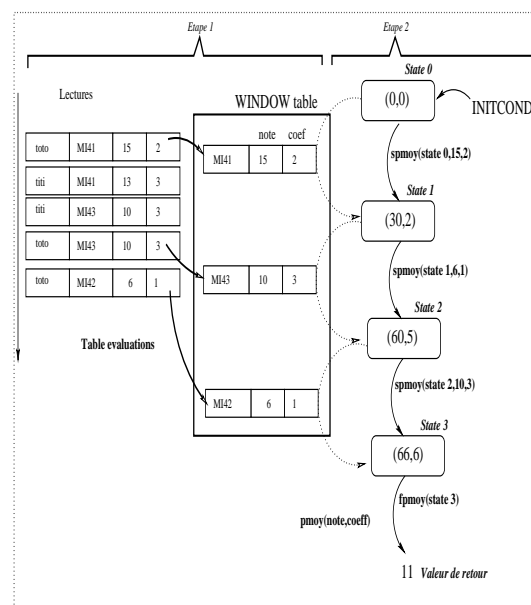
```

1 SELECT matiere,note,coeff,
2       pmoy(note,coeff)
3       over() :: decimal(4,2) "Toto"
4 FROM evaluations
5 WHERE nom='toto';

```

matiere	note	coeff	Toto
MI41	15	2	11.00
MI43	10	3	11.00
MI42	6	1	11.00

(3 rows)



Partition

1.4.4 Fonction "Trigger"

```
(abir) [abir] => \df log_data()
                        List of functions
 Schema |   Name   | Result data type | Argument data types | Type
-----+-----+-----+-----+-----
 public | log_data | trigger          |                     | trigger
(1 row)
```

1.4.5 Fonction "Variadic"

```
1 CREATE FUNCTION or replace concat(VARIADIC param_args text[])
2 RETURNS text AS
3 $$
4 SELECT array_to_string($1, ' ');
5 $$
6 LANGUAGE SQL;
```

```
(abir) [abir] => \df concat
                        List of functions
 Schema | Name | Result data type | Argument data types | Type
-----+-----+-----+-----+-----
 public | concat | text          | VARIADIC param_args text[] | normal
(1 row)
```

```
(abir) [abir] => select concat('Paul','adore','programmer','en' , 'SQL');
concat
-----
Paul adore programmer en SQL
(1 row)
```

1.4.6 Types de Données SQL

Type de Base

Est un type implémenté au niveau interne (langage C).

Type Composé

Est une liste de types associés à des champs (ou type tuple). Les commandes suivantes permettent de définir un type composé :

- tableau
- tuple (implicitement) : CREATE TABLE
- tuple (explicitement) : CREATE TYPE

Types Domaines

Est un type de base dont le domaine des valeurs est restreint par une contrainte.
La commande `CREATE DOMAIN` permet de définir un type domaine.

Pseudo-Types

Sont des types spéciaux utilisés pour définir les types des arguments et des valeurs de retour de fonctions :

- `void` : la fonction ne retourne pas de valeur,
- `trigger` : la fonction retourne un `trigger`,
- `record` : la fonction retourne un type composé indéfini.

Ils ne peuvent pas être utilisés comme type d'un attribut d'une table ou d'un type composé.

Types Polymorphiques

Sont des Pseudo-Types spéciaux (fonctions polymorphiques) :

- `any` : la fonction accepte tout type de donnée en entrée,
- `anyelement` : la fonction accepte tout type de donnée,
- `anyarray` : la fonction accepte tout tableau de types.

1.4.7 Paramètres de Fonctions SQL

Types de Base

```
1 CREATE FUNCTION pair(  
2     in int  
3 ) RETURNS boolean AS  
4 $$  
5     SELECT $1%2=0 AS Resultat;  
6 $$ LANGUAGE SQL;  
7  
8 SELECT pair(3) "3" ,pair(4) "4";
```

```
3 | 4  
---+---  
f | t  
(1 row)
```

Types Composés (ou tuples)

```
1 CREATE TABLE tabn (  
2     id      int,  
3     nom     varchar(30)  
4 );  
5  
6 insert into tabn values(2,'toto');  
7 insert into tabn values(7,'titi');  
8
```

```

9 CREATE or replace FUNCTION tab_id(
10     tabn
11 ) RETURNS int AS
12 $$
13     SELECT $1.id AS id;
14 $$ LANGUAGE SQL;
15
16 select tab_id(tabn) from tabn;

```

```

    tab_id
-----
         2
         7
(2 rows)

```

Paramètres IN, OUT, INOUT

Redondances :

```

CREATE FUNCTION tab_id(INOUT tab)
AS $$
    SELECT $1.id, $1.nom;
$$ LANGUAGE SQL;

```

Type composé indéfini :

```

CREATE FUNCTION and_or(in boolean, in boolean,
    OUT boolean, OUT boolean) AS
$$
    SELECT $1 and $2, $1 or $2;
$$ LANGUAGE SQL;

# select and_or(true,false);
and_or
-----
(f,t)
(1 row)

```

Fonction comme table :

```

# select * from and_or(true,false) as t(e,o);
 e | o
---+---
 f | t
(1 row)

```

Returning Sets

```
CREATE FUNCTION setof_avec_out(OUT a int, OUT b text)
    RETURNS SETOF RECORD AS
$$
    values (1,'a'),(2,'b')
$$ LANGUAGE SQL;

(abir) [demodb] => SELECT * FROM setof_avec_out();
 a | b
---+---
 1 | a
 2 | b
(2 rows)
```

1.5 Tableaux

1.5.1 Initialisation

Exemple :

```
\set tableau1 '\{1,2,3}\'
\set tableau2 array[5,6,7]
\set tableau3 '\[1:3]={9,10,11}\':int[]'
SELECT :tableau1 as t1,:tableau2 as t2,
       :tableau3 as t3;
   t1   |   t2   |   t3
-----+-----+-----
{1,2,3} | {5,6,7} | {9,10,11}
(1 row)
```

1.5.2 Opérateurs

- `t1 @ > t2` : t1 contient t2

```
\set t1 ARRAY[1,4,3]
\set t2 ARRAY[3,1]
SELECT :t1 @> :t2 as "t1 contient t2"
       t1 contient t2
-----
t
(1 row)
```

- `t1 < @ t2` : t1 est contenu dans t2

```

\set t1 ARRAY[2,7]
\set t2 ARRAY[1,7,4,2,6]
SELECT :t1 <@ :t2 as "t1 est contenu dans t2";
t1 est contenu dans t2
-----
t
(1 row)

```

- Concaténation

```

\set t1 ARRAY[2,3]
\set t2 ARRAY[4,5,6]
SELECT :t1||:t2 as "t1||t2", 1||:t1 as "1||t1";
t1||t2 | 1||t1
-----+-----
{2,3,4,5,6} | {1,2,3}
(1 row)

```

1.5.3 Quelques Caractéristiques Internes

- ndim : Nombre de dimensions :

`array_ndims(anyarray)`

Exemple :

```

\set tableau ARRAY[[1,2,3], [4,5,6]]
SELECT array_ndims(:tableau)
array_ndims
-----
2
(1 row)

```

- dimensions : longueur (nombre d'élément) de chaque dimension (axe) [ndim éléments].

`array_length(anyarray, int)`

Exemple :

```

\set tableau ARRAY[[1,2,3], [4,5,6]]
SELECT array_length(:tableau,1),array_length(:tableau,2);
array_length | array_length
-----+-----
2 | 3
(1 row)

```

- Borne inférieure Indice du premier élément [ndim éléments].
`array_lower(anyarray, int)`

Exemple :

```

\set tableau '\'[0:2]={1,2,3}\'::int[]
SELECT array_lower(:tableau,1),array_upper(:tableau,1);
array_lower | array_upper
-----+-----
0 | 2
(1 row)

```

1.5.4 Autres Fonctions

- Conversion tableau-Tuples

`unnest(anyarray)`

Exemple :

```
\set tableau array[2,1,3]
SELECT unnest(:tableau) as "Tuples";
Tuples
-----
      2
      1
      3
(3 rows)
```

- Conversion tableau-chaine de caractères

`array_to_string(anyarray, text)`
`string_to_array(text, text)`

Exemple :

```
\set tableau array[2,1,3]
\set string '\3-9-6-2\'
SELECT array_to_string(:tableau,',' ) as "Chaine",
       string_to_array(:string,'-') as Tableau;
Chaine | tableau
-----+-----
 2,1,3 | {3,9,6,2}
(1 row)
```

- etc ...

1.6 Procédures Stockées et Catalogue

```
(abir) [demodb] => SELECT pronom, prosrc FROM pg_proc WHERE pronom~'.pmoy';
pronom | prosrc
-----+-----
spmoy  |
      : SELECT $1.notes+COALESCE($2,0)*COALESCE($3,0),
      : $1.coefs+COALESCE($3,0);
      :
fpmoy  |
      : SELECT $1.notes / $1.coefs;
      :
(2 rows)
```