SGBD : Exploitation d'une base de données [R206]

Hocine ABIR

February 28, 2022

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

CONTENTS

CHAPTER

UN

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 eventuellement modifier les valeurs de ses arguments

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

Fonction

Fonctions stockées sont similaires aux procédures mais :

1. peuvent être utilisées dans les reqêtes, fonctions, procédures, et vues.
2. dans certains SGBD, elles ne peuvent pas modifiées les données ou ont des limitations au niveau ddl/dml.
3. en general, 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 Langages et Fonctions

Plusieurs langages peuvent être disponibles :

```
SELECT tmplname as "Name",
       Case when(tmpldbacreate)
             then 'dba'
             else 'db owner'
       end as "CREATE"
FROM pg_pltemplate;
```

```
   Name    |  CREATE
-----------+----------
 plpgsql   | dba
 pltcl     | dba
 pltclu    | db owner
 plperl    | dba
 plperlu   | db owner
 plpythonu | db owner
(6 rows)
```

Certains sont déjà installés :

```
SELECT lanname,
      CASE WHEN lanispl
             THEN 'User defined'
             ELSE 'Internal'
      END AS lanispl,
      CASE when not lanpltrusted
             then 'Not Trusted'
      end as lanpltrusted
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

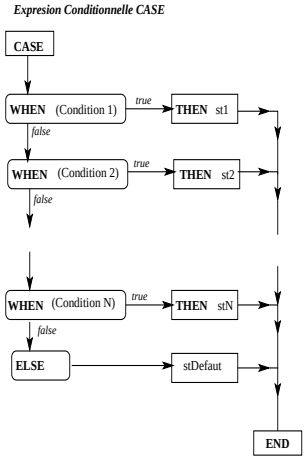
2. WHILE-DO : itération

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

1.2.1 Controle de Séquence : Case

- La clause SQL `CASE` est une 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
```



Exemple 1 : Une requête au lieu de deux

```
1  SELECT SUM(i)
2      "Somme des Nombres Paires"
3    FROM generate_series(1,5) t(i)
4    WHERE i%2=0;
5
6  SELECT SUM(i)
7      "Somme des Nombres Impaires"
8    FROM generate_series(1,5) t(i)
9    WHERE i%2!=0;
```

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

```
 Somme des Nombres Paires | Somme des Nombres Impaires
--------------------------+----------------------------
                        6 |                          9
(1 row)
```

Exemple 2 : Jour de Semaine

```c
// zeller.c
#include <stdio.h>
int main (void)
{
        int j=14; //jours
        int m=11; //mois
        int a=2013; //annee
        int m1,a1,sa,ya;
        int d;
        char * jour[]={"Dimanche","Lundi","Mardi","Mercredi",
                       "Jeudi","Vendredi","Samedi"};

        if (m<3) m1=m+10; else m1=m-2;
        a1=a;
        if (m<3) a1=a-1;
        sa=a1/100;
        ya=a1%100;
        d=j+ya+ya/4 -2*sa +sa/4 + (26*m1 -2)/10;
        d=d%7;
         printf (" Jour : %s\n",jour[d]);

        return 0;
}
```

```
$ gcc zeller.c;./a.out
 Jour : Jeudi
```

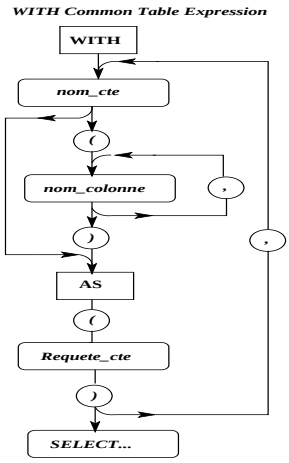
```sql
--Zeller's congruence
-- 14/11/2023
\set j  '14'
\set m '11'
\set a '2013'

 SELECT CASE
         WHEN j=0 THEN 'Dimanche'
         WHEN j=1 THEN 'Lundi'
         WHEN j=2 THEN 'Mardi'
         WHEN j=3 THEN 'Mercredi'
         WHEN j=4 THEN 'Jeudi'
         WHEN j=5 THEN 'Vendredi'
         WHEN j=6 THEN 'Samedi'
       END "Jour"
 FROM (SELECT (:j+ya+ ya/4 -2*sa + sa/4 + (26*m1-2)/10)%7
       FROM (SELECT m1,a1,a1/100 sa, a1%100 ya
             FROM  (SELECT CASE WHEN(:m<3) THEN :m+10 else :m-2 end)
                   tm1(m1),
                   (SELECT CASE WHEN(:m<3) THEN :a-1 else :a end)
                   ta1(a1)
             ) f
     ) 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 ...
```



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`.

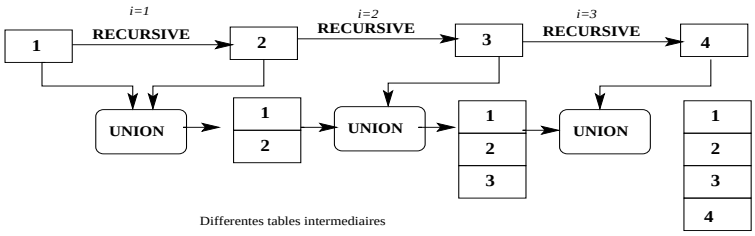
```
                          QUERY PLAN
-------------------------------------------------------------
 CTE Scan on ctetab  (cost=1.03..1.09 rows=3 width=4)
   CTE ctetab
     ->  Seq Scan on t  (cost=0.00..1.03 rows=3 width=4)
(3 rows)
```

Exemple 2 (récursion) : Boucle

```sql
WITH RECURSIVE boucle (i) AS
(
    SELECT 1 -- valeur initiale
    UNION
    SELECT i + 1 -- pas
    FROM boucle
         WHERE i < 4 -- condition fin iteration
)
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

```c
// pyramide.c
#include <stdio.h>
void main (int argc,char * argv[])
{
   int h,i,j;
    h=atoi(argv[1]);
    for(i=1;i<=h;i++)  // hauteur
     {

             for(j=i;j<h;j++)
               {
                     printf (" "); // nombre de blanc h-i
                }
             for(j=1;j<2*i;j++)
               {
                     printf ("*"); // nombre d'etoiles 2*i-1
                }
              printf ("\n");
     }
}
```

```
\set Hauteur 4

 WITH RECURSIVE pyramide (h) AS
 (
     SELECT 1
     UNION
     SELECT h + 1
          FROM pyramide
          WHERE h < :Hauteur
 )
 SELECT repeat(' ', :Hauteur-h) ||
        repeat('*', 2*h-1) as "Pyramide"
  FROM pyramide;
```

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

Exemple 4 : (union all) Décomposition en Produit de Facteurs premiers

```sql
-- Facteurs premiers d'un entier N

\set N 60

WITH RECURSIVE Decomposition (facteur, Nombre, est_facteur) AS
(
    SELECT 2, :N, false
    UNION ALL
    SELECT
         CASE WHEN Nombre % facteur = 0 THEN facteur
              WHEN facteur * facteur > Nombre THEN Nombre
              WHEN facteur = 2 THEN 3
              ELSE facteur + 2
         END , -- facteur
         CASE WHEN Nombre % facteur = 0 THEN Nombre / facteur
              ELSE Nombre
         END,  -- Nombre
         CASE WHEN Nombre % facteur = 0 THEN true
              ELSE false
         END   -- est_facteur
    FROM Decomposition
    WHERE Nombre <> 1
)
SELECT facteur
    FROM Decomposition
    WHERE est_facteur;
```

```
 facteur
---------
       2
       2
       3
       5
(4 lignes)
```

1.3 Caractéristiques des Fonctions en SQL

```sql
CREATE [ OR REPLACE ] FUNCTION  name
```

```
  (
    [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ]
    [, ...] ]
  )

    [ RETURNS rettype |
      RETURNS TABLE ( column_name column_type [, ...] ) ]

AS $$

        Corps_de_la_Fonction

  $$
LANGUAGE SQL

[ IMMUTABLE | STABLE | VOLATILE ]
[ CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT]
[ 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.

```
CREATE table tab(
     id     serial
          primary key,
     b     int
);

 create or replace function ajout
 (   bvalue  int
 )  returns int as
 $$
     insert into tab (id,b) values (default,$1)
        returning  id
 $$ language sql;

select ajout(8);
select ajout(9);
 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).

```
CREATE FUNCTION scal_func(int,out a int,out b int)
 AS
$$
  SELECT $1, $1+1;
$$ LANGUAGE sql;

SELECT * FROM scal_func(4);
SELECT (scal_func(4)).b;
```

```
 a | b
---+---
 4 | 5
(1 row)

 b
---
 5
(1 row)
```

- Fonction Ensemble : SETOF

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

SELECT * FROM setof_func(4);
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 proname,provolatile
2    FROM pg_proc
3    WHERE proname IN ('random','now','pi');
```

```
 proname | 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 $$
```

```
    SELECT $1 + $2
$$ LANGUAGE SQL
   RETURNS NULL ON NULL INPUT;
-------------------------------------
CREATE FUNCTION add2 (int, int)
  RETURNS int AS
$$
    SELECT COALESCE($1, 0) + COALESCE($2, 0)
$$  LANGUAGE SQL ;
```

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

```
SELECT proname,
   CASE WHEN  proisstrict
        THEN 'RETURNS NULL ON NULL INPUT / STRICT'
        ELSE 'CALLED ON NULL INPUT (default)'
   END "NULL INPUT"
   FROM pg_proc
   WHERE proname IN ('add1','add2');
```

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

Sécurité

- SECURITY INVOKER (default)
- SECURITY DEFINER

```
CREATE TABLE notes
(
  Nom  varchar,
  note  decimal(4,2)
) ;
INSERT INTO notes VALUES
    ('toto',12.5),('guest',13.25),('titi',9.5);
-----------------------------------------
CREATE or replace FUNCTION notes()
RETURNS SETOF notes AS
$$
  SELECT * FROM notes;
$$ 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 Opératoires 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

```
CREATE AGGREGATE name
(
   input_data_type
   [ , ... ]
)
(
   SFUNC = state_func,
   STYPE = state_data_type
   [ , FINALFUNC = final_func ]
   [ , INITCOND = initial_condition ]
   [ , SORTOP = sort_operator ]
)
```

Exemple

```
-- etat
CREATE TYPE state AS
(
  notes    decimal,
  coefs       decimal
);

-- somme pondere et somme coefficients
CREATE FUNCTION spmoy (INOUT state,IN decimal,IN decimal)
   AS $$
   SELECT $1.notes+COALESCE($2,0)*COALESCE($3,0),
          $1.coefs+COALESCE($3,0);
$$ LANGUAGE sql;
```

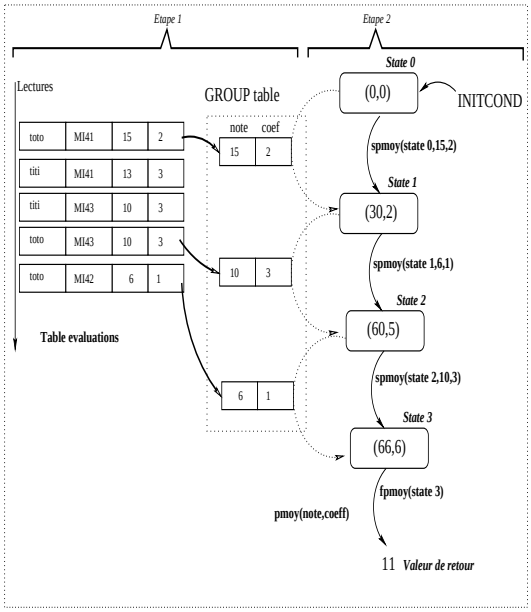
```
14
15 -- moyenne pondere
16 CREATE FUNCTION fpmoy (state)
17     RETURNS decimal AS $$
18    SELECT $1.notes / $1.coefs;
19 $$ LANGUAGE sql;
20
21 -- pmoy (notes,coefficients) retourne moyenne ponderee
22 CREATE AGGREGATE pmoy (decimal,decimal)
23 (
24     STYPE = state,
25     SFUNC = spmoy,
26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

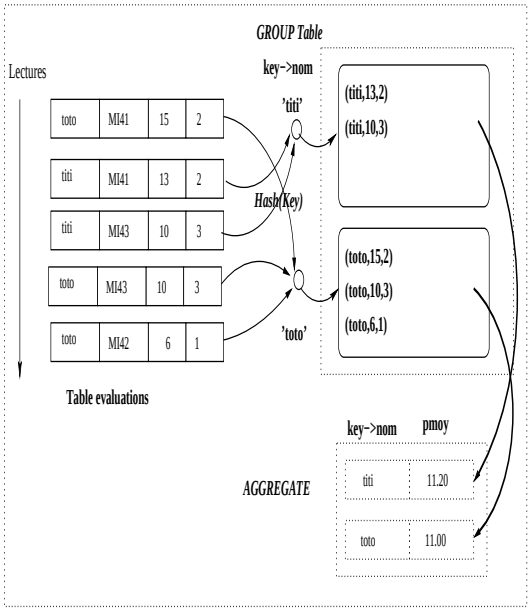
```
 Toto
-------
 11.00
(1 row)
```



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)
```



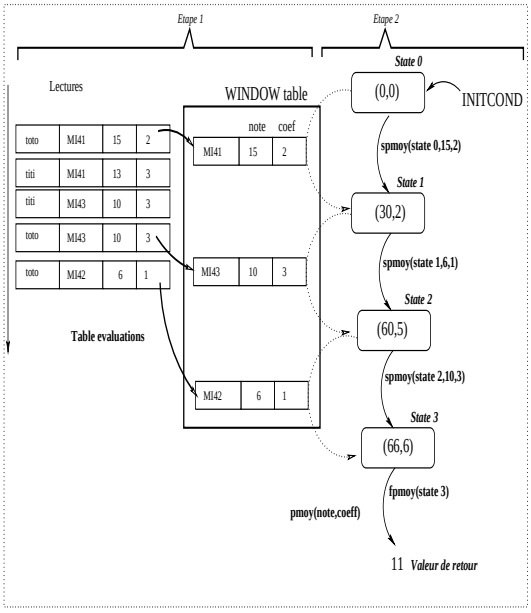
1.4.3 Fonction "Windows"

```
(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)

```
CREATE TABLE tabn (
    id      int,
    nom     varchar(30)
);

insert into tabn values(2,'toto');
insert into tabn values(7,'titi');

CREATE or replace FUNCTION tab_id(
    tabn
 ) RETURNS int AS
 $$
    SELECT $1.id AS id;
$$ LANGUAGE SQL;

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)
```

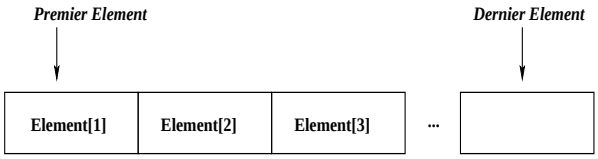
Returning Sets

```
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 Introduction




```
1 CREATE TABLE notes (
2     etudiant       varchar,
3     controle      decimal(4,2)[]
4 );
5
6 INSERT INTO notes values
7    ('Paul','{10.5 , 11, 8 , 12}'),
8    ('Pierre',array[12.5 , 13.5 , 8.5 , 16]);
```

```
=> SELECT * FROM notes;
 etudiant |         controle
----------+--------------------------
 Paul     | {10.50,11.00,8.00,12.00}
 Pierre   | {12.50,13.50,8.50,16.00}
(2 rows)

=> SELECT etudiant, controle[2:3] FROM notes;
 etudiant |   controle
----------+--------------
 Paul     | {11.00,8.00}
 Pierre   | {13.50,8.50}
(2 rows)

=> UPDATE notes set controle[2:3]='{10,11}';
UPDATE 2

=> SELECT etudiant, controle[2:3] FROM notes;
 etudiant |   controle
----------+---------------
 Paul     | {10.00,11.00}
 Pierre   | {10.00,11.00}
(2 rows)
```

1.5.2 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)
```

1.5.3 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)
```

1.5.4 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
```

1.5.5 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 proname, prosrc FROM pg_proc WHERE proname~'.pmoy';
 proname |                       prosrc
---------+-------------------------------------------------
 spmoy   |
         :    SELECT $1.notes+COALESCE($2,0)*COALESCE($3,0),
         :           $1.coefs+COALESCE($3,0);
         :
 fpmoy   |
         :    SELECT $1.notes / $1.coefs;
         :
(2 rows)
```