BDav-M1

Bases de données avancées

Cours de Cristina Sirangelo

IRIF, Université Paris Diderot

Assuré en 2021-2022 par Amélie Gheerbrant

amelie@irif.fr

Triggers et fonctions stockées

Triggers

- Objectif : surveiller l'état d'une BD et réagir quand une condition se présente

- Les *triggers* sont en général exprimés dans une syntaxe similaire aux assertions (voir plus loin) et incluent les parties suivantes:
  - **événement** (e.g., une opération de mise à jour de la BD)
  - **condition** (une condition qui déclenche l'exécution du trigger)
  - **action** (à réaliser quand la condition est satisfaite)

Triggers : un exemple

- Un trigger pour comparer le salaire d'un employé à celui de son encadrant pendant les opérations d'insertion ou mise à jour :

syntaxe SQL standard

```
CREATE TRIGGER Alerte_chef
BEFORE INSERT OR UPDATE OF salaire, nss_chef ON Employé
FOR EACH ROW
WHEN (NEW.salaire > (SELECT salaire FROM Employé
                     WHERE nss = NEW.nss_chef)
      )
INSERT INTO Alerte VALUES (NEW.nss_chef, nss);
```

Triggers

- Opération typiques effectuées par les triggers sur une BD:
  - mises à jour automatiques :
    - maintien d'un log des opérations
    - complétion automatiques de tuples avant insertion , …
  - vérification de contraintes complexes (un trigger peut annuler la mise à jour en cours si elle viole certaines contraintes)
    - pas l'utilisation la plus naturelles de triggers (les assertions sont plus adaptées à cet effet)
    - cependant peu de SGBD implémentent les assertions

Triggers

- Beaucoup de variantes dans la syntaxe et les fonctionnalités
- Plusieurs sémantiques possibles pour l'exécution du trigger:
  - `BEFORE / AFTER` (avant ou après l'événement déclenchent)
  - `FOR EACH ROW / STATEMENT` (pour chaque ligne affectée par la mise à jour ou une fois pour chaque opération de mise à jour)
  - `IMMEDIATE / DEFERRED` (exécution immédiate ou reportée en fin de transaction)
  - etc.
- Le comportement des triggers peut être difficile à prévoir

  peut donner lieu à des exécutions sans terminaison!
- Sous-domaine de recherche en bases de données :
  les "bases de données actives"

Une forme implicite (et "safe") de trigger : cascade

- Peut être associé à une relation avec une clef étrangère
- Fait en sorte que l'intégrité référentielle soit respectée

```
CREATE TABLE Compte
(num      CHAR(10),
id_agence   CHAR(15),
montant INTEGER,
PRIMARY KEY (num),
FOREIGN KEY (id_agence)
             REFERENCES Agence ON DELETE CASCADE )
```

Sémantique : si un tuple de `Agence` qui était référencé par un tuple `t` de `Compte` est supprimé, le tuple `t` est aussi supprimé

Triggers en postgreSQL

```
CREATE TRIGGER nom_trigger
{ BEFORE | AFTER } evenement [ OR … ]
   ON nom_table
[ FOR EACH ROW | FOR EACH STATEMENT   ]
[ WHEN ( condition ) ]
   EXECUTE PROCEDURE nom_fonction ()

evenement := INSERT | DELETE | UPDATE [ OF colonne[,… ]]
```

syntaxe simplifiée

- *événement* : le ou les types de commandes SQL sur la table *nom_table* qui déclenchent le trigger : insertion, élimination ou mise à jour
  - pour `UPDATE` on peut avoir une liste de colonnes : seulement les mises à jour qui affectent au moins une de ces colonnes déclenchent le trigger
- *nom_fonction* : la fonction à exécuter quand l'événement se présente
- *FOR EACH ROW* : le trigger est exécuté une fois pour chaque ligne affectée par la commande déclenchante
- *FOR EACH STATEMENT* : le trigger est exécuté une seule fois pour la commande déclenchante (même si elle affecte 0 lignes)

Triggers en postgreSQL

```
CREATE TRIGGER nom_trigger
{ BEFORE | AFTER } evenement [ OR … ]
  ON nom_table
[ FOR EACH ROW | FOR EACH STATEMENT  ]
[ WHEN ( condition ) ]
  EXECUTE PROCEDURE nom_fonction ()

evenement := INSERT | DELETE | UPDATE [ OF colonne[,… ]]
```

syntaxe simplifiée

- Exemples

```
CREATE TRIGGER verification_comptes
BEFORE INSERT OR UPDATE ON Comptes
FOR EACH ROW
EXECUTE PROCEDURE check_account_update();

CREATE TRIGGER log
AFTER UPDATE ON Comptes
FOR EACH STATEMENT
EXECUTE PROCEDURE log_update();
```

Triggers en postgreSQL

```
CREATE TRIGGER nom_trigger
{ BEFORE | AFTER } evenement [ OR … ]
   ON nom_table
[ FOR EACH ROW | FOR EACH STATEMENT   ]
[ WHEN ( condition ) ]
   EXECUTE PROCEDURE nom_fonction ()

evenement := INSERT | DELETE | UPDATE [ OF colonne[,… ]]
```

syntaxe simplifiée

- *BEFORE / AFTER* : le trigger est exécuté avant / après la mise à jour qui le déclenche, plus précisément
  - *BEFORE … FOR EACH STATEMENT* : trigger exécuté juste avant l'exécution de la commande déclenchante
  - *AFTER … FOR EACH STATEMENT* : trigger exécuté juste après l'exécution de la commande déclenchante
  - *BEFORE … FOR EACH ROW* : trigger exécuté une fois pour chaque ligne mise à jour, juste avant la mise à jour de cette ligne
  - *AFTER … FOR EACH ROW* : trigger exécuté juste après l'exécution de la commande déclenchante, une fois pour chaque ligne affectée

Triggers en postgreSQL - suite

```
CREATE TRIGGER nom_trigger
{ BEFORE | AFTER } evenement [ OR … ]
  ON nom_table
[ FOR EACH ROW | FOR EACH STATEMENT  ]
[ WHEN ( condition ) ]
  EXECUTE PROCEDURE nom_fonction ()

evenement := INSERT | DELETE | UPDATE [ OF colonne[,… ]]
```

- *WHEN (condition)* : la condition est évaluée avant l'exécution du trigger et le trigger est exécuté uniquement si la condition est satisfaite

  - pour triggers *BEFORE/AFTER … FOR EACH STATEMENT* : condition évaluée juste avant / après l'exécution de la commande déclenchante
  - pour triggers *BEFORE/AFTER … FOR EACH ROW* : condition évaluée juste avant / après la mise à jour de chaque ligne affectée par la commande
  - plus efficace que vérifier la condition au début de `nom_fonction()`, surtout pour des triggers de type `AFTER…FOR EACH ROW`

Triggers en postgreSQL - suite

```
CREATE TRIGGER nom_trigger
{ BEFORE | AFTER } evenement [ OR … ]
  ON nom_table
[ FOR EACH ROW | FOR EACH STATEMENT  ]
[ WHEN ( condition ) ]
  EXECUTE PROCEDURE nom_fonction ()

evenement := INSERT | DELETE | UPDATE [ OF colonne[,… ]]
```

- *WHEN (condition)* :

  - Dans les triggers `FOR EACH ROW` la condition peut faire référence à la ligne courante :

    - `OLD` : la ligne avant la modification
      (seulement pour triggers `UPDATE` ou `DELETE`)

    - `NEW` : la ligne après la modification
      (seulement pour triggers `UPDATE` ou `INSERT`)

Triggers en postgreSQL - suite

```
CREATE TRIGGER nom_trigger
{ BEFORE | AFTER } evenement [ OR … ]
  ON nom_table
[ FOR EACH ROW | FOR EACH STATEMENT  ]
[ WHEN ( condition ) ]
  EXECUTE PROCEDURE nom_fonction ()

evenement := INSERT | DELETE | UPDATE [ OF colonne[,… ]]
```

- *WHEN (condition)* : Exemple

```
CREATE TRIGGER trig_comptes
 BEFORE UPDATE ON Comptes
 FOR EACH ROW
 WHEN (OLD.montant IS DISTINCT FROM NEW.montant)
 EXECUTE PROCEDURE check_account_update();
```

Fonctions de trigger en postgreSQL

- À différence du standard SQL, un trigger postgreSQL peut seulement exécuter une fonction

```
EXECUTE PROCEDURE nom_fonction()
```

- Toutes les commandes/code à exécuter en réponse à l'événement déclenchant doivent se trouver dans le corps de `nom_fonction()`

- `nom_fonction` est une fonction définie par l'utilisateur (fonction stockée) avec la syntaxe `CREATE FUNCTION` (cf. plus loin), **avant la création du trigger**

- `nom_fonction` doit retourner le type `trigger` et ne prend pas d'arguments

- Les fonctions des triggers de type `FOR EACH ROW` ont accès aux variables `OLD` et `NEW` tout comme les conditions

  - ainsi que à d'autres variables (`TG_NAME, TG_WHEN, TG_OP, ...`) qui donnent des informations sur le trigger qui les a déclenchées

Fonctions de trigger en postgreSQL

- Le trigger suivant est exécuté après chaque insertion dans la table Emp :

```
CREATE FUNCTION log() RETURNS trigger AS $$
 BEGIN
    INSERT INTO Ins_log SELECT NEW.*, current_timestamp;
    RETURN NULL;
 END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER emp_log
AFTER INSERT ON Emp
FOR EACH ROW EXECUTE PROCEDURE log();
```

  - pour chaque ligne `t` insérée dans `Emp`, le trigger insère dans la table `Ins_log` la ligne (`t, current_timestamp`)

- Les fonctions de triggers (comme toutes les fonctions stockées) en postgreSQL peuvent être définies en plusieurs langages de programmation (C, PL/pgSQL, etc.)
- Les fonctions de type trigger ne peuvent pas être définies en SQL pur

Valeurs de retour des fonctions de trigger

- les triggers de type `STATEMENT` doivent retourner `NULL`
- pour les triggers `FOR EACH ROW` :
  - type `AFTER` : la valeur de retour est ignorée (peut être `NULL`)
  - type `BEFORE` : la valeur de retour peut être `NULL` ou un n-uplet
    - `NULL` : la mise à jour déclenchante **ne sera pas exécutée** (pour la ligne courante)
    - n-uplet `t` pour les triggers `INSERT` ou `UPDATE` :
      - la mise à jour a lieu
      - `t` donne la nouvelle valeur de l'n-uplet inséré /mis à jour
      - `t` doit être `NEW` si on ne veut pas altérer l'insertion / mise à jour
    - n-uplet `t` pour les triggers `DELETE`
      - la mise à jour a lieu, la valeur spécifique (non `NULL`) de `t` est ignorée (typiquement `t = OLD`)

Triggers et erreurs

- Tout trigger (y compris `STATEMENT` ou `AFTER`) qui génère une exception (non capturée) annule la mise à jour déclenchante
  - il annule également l'exécution de tous les triggers suivants sur cette même mise à jour;
  - en cas de trigger `FOR EACH ROW` :
    - l'opération déclenchante est annulée uniquement pour la ligne courante, le reste de la commande est exécutée

Ordre d'exécution des triggers

- Si plusieurs triggers sont déclenchés par le même événement, ils sont exécutés en ordre alphabétique
  - Pour des triggers `BEFORE` de type `FOR EACH ROW` :
    - si un trigger renvoie un n-uplet , celui-là sera le n-uplet d'input (`NEW` pour `INSERT` ou `UPDATE`) du prochain trigger déclenché sur la même ligne par le même événement
    - si un trigger renvoie `NULL`, la mise à jour sur la ligne courante - ainsi que tous les triggers suivants sur cette même ligne - sont annulés

Fonctions en PostgreSQL

- Pour créer des triggers il est nécessaire de créer des fonctions stockées
- Les fonctions stockées ont d'autres utilisations que dans les triggers :
  - elles permettent de stocker sur le serveur une suite de commandes / instructions récurrentes
  - l'exécution d'une fonction par le serveur est plus efficace que l'exécution de la même suite de commandes / instructions par l'application :
    - évite plusieurs cycles de communication application / serveur
- PostgreSQL permet d'écrire des fonctions en plusieurs langages (SQL, C, langages procédurales (PL),…)

Fonctions PostgreSQL

- Syntaxe (simplifiée) pour la création d'une fonction:

```
CREATE [OR REPLACE] FUNCTION nom_fonc ([arg Type [, …]])
RETURNS Type AS

$$ <definition> $$

LANGUAGE langage
```

- `langage` peut être `sql`, `plpgsql`, `C` ,etc…
  - `sql` :
    - `<definition>` doit être une suite de commandes SQL terminées par ;
  - `plpgsql` :
    - `<definition>` doit être un bloc de code PL/pgSQL, un langage procédurale disponible avec la distribution standard de PostgreSQL (cf. plus loin)
- pour exécuter la fonction et afficher sont résultat :

```
SELECT nom_fonc(valeur, …);
```

Types des paramètres et de retour des fonctions PostgreSQL

- Tous les types de base : INT, TEXT, … etc.
- Types variables : nom_table.nom_attr%TYPE
  - le type de l'attribut *nom_attr* de la table *nom_table*
- Types qui dénotent une ligne
  - RECORD :
    n-uplet de structure arbitraire
  - *nom_table*%ROWTYPE :
    le type d'une ligne de la table *nom_table* (%ROWTYPE optionnel)
  - un type composé déclaré avec CREATE TYPE *monType* AS (att Type [, ...]);
- Types qui dénotent un ensemble de lignes
  (uniquement pour le type de retour)
  - SETOF Type_ligne
  - TABLE (*nom_col* Type [, …])
  - void : type de retour pour les fonctions qui ne renvoient aucune valeur

Exemples d'entête de fonctions PostgreSQL

```
CREATE FUNCTION add (i int, j int) RETURNS int AS $$
...
$$ LANGUAGE plpgsql;
```

```
CREATE TABLE Emp (...);

CREATE FUNCTION emp_id (id int) RETURNS Emp%ROWTYPE AS $$
...
$$ LANGUAGE sql;
```

```
CREATE TABLE Emp (...);

CREATE FUNCTION emp_nom (nom TEXT) RETURNS SET OF Emp AS
$$
...
$$ LANGUAGE plpgsql;
```

Exemples d'entête de fonctions PostgreSQL

```
CREATE FUNCTION change (e Emp%ROWTYPE) RETURNS RECORD AS
$$
...

$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION extended_sales(itemno int)
RETURNS TABLE(quantity int, total numeric) AS $$
...
$$ LANGUAGE plpgsql;
```

```
CREATE TYPE Compfoo AS (f1 int, f2 text);

CREATE FUNCTION getfoo() RETURNS Compfoo AS $$
    ...
$$ LANGUAGE sql;
```

Définition de fonctions en langage SQL

La `<definition>` d'une fonction déclarée avec `LANGUAGE sql;` contient une suite de commandes SQL séparées par **;**

- Si la fonction retourne un type différent de `void`, la dernière commande doit avoir une valeur de retour (SELECT ou commandes de mise à jour "RETURNING")
  - la fonction renvoie la valeur retournée par la dernière commande SQL

```
CREATE FUNCTION clean_emp() RETURNS void AS $$
    DELETE FROM Emp
        WHERE salary < 0;
$$ LANGUAGE SQL;
```

```
CREATE FUNCTION transfer (accountno int, debit numeric)
RETURNS numeric AS $$
    UPDATE Account
        SET balance = balance - debit
        WHERE no = accountno;
    SELECT balance FROM Account WHERE no = accountno;
$$ LANGUAGE SQL;
```

Définition de fonctions en langage SQL - suite

```
CREATE FUNCTION getName (e Emp) RETURNS text AS $$
    SELECT e.name;
$$ LANGUAGE SQL;
```

- Si la fonction ne retourne pas un type "ensemble" (SETOF ou TABLE)
  - valeur retournée : la première ligne du résultat de la dernière commande

```
CREATE FUNCTION getEmp(dpt int) RETURNS Emp AS $$
    SELECT * FROM Emp WHERE departement = dpt;
$$ LANGUAGE SQL;

-- renvoie le premier employé du département dpt
```

```
CREATE FUNCTION getEmps(dpt int) RETURNS SETOF Emp AS $$
    SELECT * FROM Emp WHERE departement = dpt;
$$ LANGUAGE SQL;

-- renvoie tous les employés du département dpt
```

Définition de fonctions PL/pgSQL

- La `<definition>` d'une fonction déclarée avec `LANGUAGE plpgsql;` est un bloc de code PL/pgSQL

- Bloc de code PL/pgSQL :

```
[ DECLARE
      <declarations> ]
  BEGIN
      <instructions>
  END;
```

  - Les blocs de code peuvent être imbriqués

Déclarations PL/pgSQL

- `<declarations>` : une suite de déclarations de variable de la forme (syntaxe simplifiée) :

  nom_var Type_var [:= valeur] ;

- Exemples de déclarations

```
user_id int = 7;

quantity numeric(5);

url varchar := 'http://mysite.com';

ligne1 nom_table%ROWTYPE;

champ nom_table.nom_att%TYPE;

ligne2 RECORD;

tab int [] ; -- tableau d'entiers
```

Quelques instructions PL/pgSQL

`<instructions>` : une suite d'instructions PL/pgSQL terminées par `;`

- Affectations : `nom_var := expression;`
- Quelques structures de contrôle :
  - `IF` *`bool-expr`* `THEN ...[ELSE...] END IF;` (`ELSEIF` possible)
  - `WHILE` *`bool-expr`* `LOOP ... END LOOP;`
  - `FOR var IN [REVERSE]` *`expr1..expr2`* `LOOP ... END LOOP;`
    - `var` est une variable entière (pas besoin de la déclarer)
    - `expr1` et `expr2` : deux expressions entières
    - sans `REVERSE`, `var` incrémenté à chaque itération
    - avec `REVERSE`, `var` décrémenté à chaque itération

Quelques instructions PL/pgSQL

- Quelques structures de contrôle (suite) :
  - `FOR var1, ..., vark IN`
    `SELECT a1, ..,ak FROM... WHERE…`
    `LOOP ... END LOOP;`
    - `var1, ..vark` peut être remplacé par une variable de type "ligne" (`RECORD` ou *`nom_table`*`%ROWTYPE` ou type composé)

Structures de contrôle PL/pgSQL - Exemple

```
CREATE FUNCTION moyenne (eid Etudiant.id%TYPE) RETURNS
DECIMAL(4,2) AS $$
DECLARE
   ligne RECORD;
   q Examens.note%TYPE := 0;
   i INTEGER := 0 ;
BEGIN
   FOR ligne IN
      SELECT note, bonus FROM Examens WHERE  etu = eid
   LOOP
      IF ligne.bonus = TRUE THEN q := q + 22;
      ELSE q = q + ligne.note; END IF;
      i := i + 1 ;
   END LOOP;
   IF i = 0 THEN RETURN 0;
   END IF;
   RETURN (q :: DECIMAL / i) :: DECIMAL(4,2);
   -- :: fait un cast
END;
$$ LANGUAGE plpgsql;
```

Quelques instructions PL/pgSQL - suite

- Commandes SQL (possiblement contenant des variables PL/pgSQL)
  - Commandes SQL sans résultat (`INSERT` etc)
  - Commandes `SELECT INTO` :

    ```
    SELECT a1, ...,ak INTO var1, ..vark FROM ...;
    ```

    - la première ligne retournée par la requête est stockée dans `var1, ..vark` (valeurs NULL si le résultat de la requête est vide)
    - `var1, ..vark` peut être remplacé par une variable de type ligne
    - peut être suivi de `IF [NOT] FOUND THEN ... END IF;` pour tester si au moins une ligne a été retournée ou pas
  - ATTENTION : `SELECT` sans `INTO` génère une erreur
  - Requêtes dynamiques :

    ```
    EXECUTE '<requête SQL>' [INTO … ];
    ```

    - équivalent à la commande `<requête SQL> [INTO…]`

Commandes SQL en PL/pgSQL - Exemples

```
CREATE FUNCTION check_password(uname TEXT, pass TEXT)
RETURNS BOOLEAN AS $$
DECLARE passed BOOLEAN;
BEGIN
   SELECT  (pwd = pass) INTO passed FROM  Pwds
   WHERE   username = uname;
   IF NOT FOUND THEN passed:= FALSE; END IF;
   RETURN passed;
END;
$$  LANGUAGE plpgsql;
```

schéma: `Pwds(username, pwd)`

```
CREATE FUNCTION nom (eid Etudiant.id%TYPE)
RETURNS TEXT  AS $$
DECLARE ligne Etudiant%ROWTYPE;
BEGIN
   SELECT  * INTO ligne FROM  Etudiant WHERE id = eid;
   IF NOT FOUND THEN RETURN ''; END IF;
   RETURN ligne.prenom || ' ' || ligne.nom; -- concatenation
END;
$$ LANGUAGE plpgsql;
```

schéma: `Etudiant(id, prenom, nom)`

Quelques instructions PL/pgSQL - suite

- Retour de fonction :
  - fonctions avec type de retour simple ou ligne :

    `RETURN expression ;`

    - `expression` compatible avec le type de retour;
    - termine la fonction
  - fonctions avec type de retour ensemble:

`RETURN QUERY SELECT ...FROM... ;`

- renvoie le résultat de la requête

`RETURN NEXT expression` (`expression` de type ligne)

- renvoie la ligne rendue par l'expression
- `RETURN QUERY` et `RETURN NEXT` ne terminent pas la fonction, elles accumulent leur résultat dans l'ensemble à retourner
- la fonction retourne cet ensemble quand le contrôle arrive à la fin, ou quand un `RETURN;` est rencontré

Retour de fonction PL/pgSQL - Exemples

```
CREATE FUNCTION extended_sales(item int)
RETURNS TABLE(quantity int, total numeric) AS $$
BEGIN
   RETURN QUERY SELECT Sales.quantity, Sales.quantity *
   Sales.price FROM Sales WHERE itemno = item;
END; $$  LANGUAGE plpgsql;
```

```
CREATE FUNCTION all_sales(item int)
RETURNS SETOF NatSales AS $$
DECLARE ligne RECORD;
BEGIN
   RETURN QUERY SELECT * FROM NatSales WHERE itemno = item;
   FOR ligne IN  SELECT * FROM IntSales  WHERE itemno = item
   LOOP
        … -- un calcul qui  modifie ligne
        RETURN NEXT ligne;
   END LOOP;
END;
$$  LANGUAGE plpgsql;
```

schéma:

```
Sales(itemno, quantity, price)
NatSales(itemno, quantity, price)
IntSales(itemno, quantity, price)
```

Curseurs

- Les curseurs sont des variables du langage procédural associées à une requête
- Un curseur permet de parcourir, une par une, les ligne du résultat de la requête
  - un pointeur interne à la prochaine ligne à lire
  - opération d'ouverture, clôture et lecture de la prochaine ligne (avec avancement du pointeur interne)
- En PL/pgSQL l'instruction

```
FOR ligne IN SELECT … FROM … LOOP … END LOOP;
```

crée implicitement un curseur sur la requête et la parcourt entièrement

- Les curseurs peuvent aussi être créés explicitement. Utile dans plusieurs situations :
  - pour parcourir le résultat de la même requête plusieurs fois
  - pour parcourir le résultat d'une requête dans un ordre spécifique
  - pour retourner le résultat d'une requête à une autre fonction

Curseurs en PL/pgSQL

- Un curseur PL/pgSQL est une variable de type `refcursor`
- Trois types de déclarations possibles :

```
DECLARE
        …
  nom_curs1 refcursor; -- curseur non lié

  nom_curs2 CURSOR FOR SELECT … FROM … ; -- curseur lié

  nom_curs3 CURSOR (param Type, … ) FOR
            SELECT … FROM … WHERE att = param;
                                -- curseur lié avec paramètres
```

Curseurs en PL/pgSQL

- Ouverture d'un curseur (nécessaire avant de le parcourir) :
  - curseur lié :

    ```
    OPEN nom_curs2;
    ```

  - curseur lié avec paramètres :

    ```
    OPEN nom_curs3(42, var, …);
    ```

  - curseur non lié :

    ```
    OPEN nom_curs1 FOR SELECT … FROM … WHERE att = var;
    ```

Remarque : un curseur non lié peut être associé à une requête contenant des valeurs et variables du programme

Curseurs en PL/pgSQL

- Clôture d'un curseur :

```
CLOSE nom_curs;
```

  - une fois ouvert un curseur peut être utilisé par toutes les fonctions qui en possèdent une référence et qui sont exécutées dans la même transaction
  - tous les curseurs encore ouverts sont automatiquement fermés à la fin de la transaction

Curseurs en PL/pgSQL

- Utilisation d'un curseur ouvert :
  - `FETCH` lit et stocke la prochaine ligne du curseur dans des variables (la ligne lue devient ensuite la ligne courante):

```
FETCH nom_curs INTO var1, …, vark;

FETCH nom_curs INTO var_ligne;
                        -- var_ligne: variable de type ligne
```

  - s'il n'y a pas de prochaine ligne, les variables prennent la valeur NULL
  - la variable `FOUND` peut être testée pour savoir si la ligne a été trouvée
  - options différentes pour spécifier la ligne à lire :

```
FETCH FIRST FROM curs3 INTO x, y; -- la première ligne
FETCH PRIOR FROM curs3 INTO x, y; -- la ligne précédente
FETCH RELATIVE -2 FROM curs4 INTO x;
 -- la ligne qui précède la ligne courante de 2 positions
```

etc.

Curseurs en PL/pgSQL

- Utilisation d'un curseur ouvert :

  - déplace la ligne courante sans lire

    ```
    MOVE nom_curs ; -- prochaine ligne
    MOVE LAST FROM  nom_curs -- dernière ligne
    etc.
    ```

    mêmes options que `FETCH` pour spécifier la ligne

Curseurs en PL/pgSQL

- Un raccourci pour parcourir entièrement le curseur
  (remplace une séquence de `FETCH nom_curs INTO`...)

  Exemple :

```
DECLARE
      nom_curs CURSOR (param1 INT, param2 INT) FOR
                SELECT * FROM TestTable
                WHERE att1 = param1 AND att2 = param2;
       ligne RECORD;
BEGIN
FOR ligne IN nom_curs(3,2) LOOP
   instructions
END LOOP;
…
END;
```

- Possible uniquement pour des curseurs liés et pas encore ouverts.

Curseurs en PL/pgSQL

- Renvoyer un curseur
  - Une fonction peut renvoyer un curseur pour renvoyer le résultat d'une requête à une autre fonction

Exemple

```
CREATE FUNCTION f() RETURNS refcursor AS $$
DECLARE
    ref refcursor;
BEGIN
    …
    OPEN ref FOR SELECT * FROM maTable WHERE…;
    …
    RETURN ref;
END;
$$ LANGUAGE plpgsql;
```

Curseurs en PL/pgSQL

- Renvoyer un curseur - **Exemple** - suite :

```
CREATE FUNCTION c() RETURNS maTable AS $$
DECLARE
    curs refcursor;
    x maTable;
BEGIN
    curs := f();
    …
    FETCH LAST FROM curs INTO x;
    RETURN x;
END;
$$ LANGUAGE plpgsql;
```

- Attention : Pour que l'appelant de `f()` puisse utiliser le curseur, `f()` doit être exécutée dans la même transaction que l'appelant
- Si l'appelant est une autre fonction c'est toujours le cas

Messages et erreurs en PL/pgSQL

- Afficher un message sur la console pendant l'exécution d'une fonction :

```
RAISE NOTICE 'élève %  absent', eid;
```

`%` est remplacé par la valeur de la variable `eid`

  - n'interrompt pas l'exécution de la fonction

- Soulever une exception

```
RAISE [EXCEPTION] 'élève %  absent', eid;
```

  - Si non-capturée, interrompt l'exécution de la fonction et fait un ROLLBACK de la transaction jusqu'au début de la fonction

- D'autres niveaux disponibles entre `NOTICE` et `EXCEPTION`
- Uniquement le niveau `EXCEPTION` interrompt l'exécution de la fonction et annule la transaction en cours

Messages et erreurs en PL/pgSQL

- Conditions et SQLSTATE :
  - À tous les niveaux (`NOTICE`, `EXCEPTION`, etc) le message peut être remplacé par une condition prédéfinie.
  - Une condition est identifiée
    - soit par un nom : Ex. `division_by_zero`
    - soit par un code : Ex. `SQLSTATE 22012`
  - Ainsi les deux instructions suivantes soulèvent la même exception :

```
RAISE division_by_zero;
RAISE SQLSTATE '22012';
```

Messages et erreurs en PL/pgSQL

- Capturer une exception
  - Entourer les instructions qui peuvent soulever des exceptions dans un sous-bloc de code `BEGIN...END`
  - Avant `END` ajouter une section `EXCEPTION` qui fournit un "*handler*" pour chaque exception qui peut être soulevée par les instructions

```
BEGIN
    instructions
EXCEPTION
    WHEN division_by_zero THEN instructions
    WHEN SQLSTATE 23000 THEN instructions
END;
```

  - Quand ce bloc soulève une exception qui a un *handler*, l'*handler* est exécuté et la fonction continue à partir de `END;`
  - Si l'exception soulevée n'a pas de *handler,* elle annule l'exécution de la fonction et la transaction en cours
    (si elle n'est pas capturée par un bloc de plus haut niveau)

Utilisation des fonctions : exemples de triggers PL/pgSQL

Vérification de contraintes et complétion automatique de ligne

```
CREATE TABLE emp (
empname text, salary integer, last_update timestamp);


CREATE FUNCTION add_stamp() RETURNS trigger AS $$
  BEGIN
    IF NEW.empname IS NULL THEN return NULL; END IF;
    IF NEW.salary IS NULL THEN return NULL; END IF;
    IF NEW.salary < 0 THEN return NULL; END IF;
    NEW.last_update := current_timestamp;
    RETURN NEW;
  END;
$$ LANGUAGE plpgsql;


CREATE TRIGGER emp_stamp
BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE add_stamp();
```

Utilisation des fonctions : exemples de triggers PL/pgSQL

Vérification de contraintes et mise à jour automatique de la BD

Schéma : Compte(num, solde), Virement(orig,dest,date,montant)

```
CREATE FUNCTION update_account() RETURNS trigger AS $$
 DECLARE o Compte%ROWTYPE; d Compte%ROWTYPE;
 BEGIN
    SELECT * INTO o FROM Compte WHERE num = NEW.orig;
    IF NOT FOUND then RAISE EXCEPTION 'compte inexistant';END IF;
    SELECT * INTO d FROM Compte WHERE num = NEW.dest;
    IF NOT FOUND THEN RAISE EXCEPTION ‘compte inexistant’;END IF;
    IF NEW.montant > o.solde THEN
       RAISE EXCEPTION ‘solde insuffisant : %’, o.solde; END IF;
    UPDATE Compte SET solde=solde-NEW.montant WHERE num= o.num;
    UPDATE Compte SET solde=solde+NEW.montant WHERE num= d.num;
    RETURN NEW;
 END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER virement BEFORE INSERT ON Virement
     FOR EACH ROW EXECUTE PROCEDURE update_account();
```

Utilisation des fonctions dans les commandes SQL

- Une fonction peut être utilisée dans une commande SQL là où son type de retour est attendu.

Exemple1 : Dans la partie FROM à la place d'une table, si elle renvoie une table ou un ensemble de lignes

```
CREATE FUNCTION getemp(enom text) RETURNS SETOF Emp AS $$
    SELECT * FROM Emp WHERE nom = enom;
$$ LANGUAGE SQL;

SELECT * FROM getemp('Dupont') AS t1;
```

Utilisation des fonctions dans les commandes SQL

- Une fonction peut être utilisée dans une commande SQL là où son type de retour est attendu.

Exemple 2: Dans la partie SELECT à la place d'un (ou plusieurs) attributs, si elle renvoie une valeur (ou un n-uplet)

```
CREATE FUNCTION nom_complet(e Emp) RETURNS text AS $$
BEGIN
   RETURN e.prenom || ' ' || e.nom;
END;
$$ LANGUAGE plpgsql;


SELECT nom_complet(Emp.*) FROM Emp;
```

Référence

Plus de détail en TP et sur :

- *documentation postgreSQL* : Triggers et PL/pgSQL
  https://www.postgresql.org/docs/13/server-programming.html