

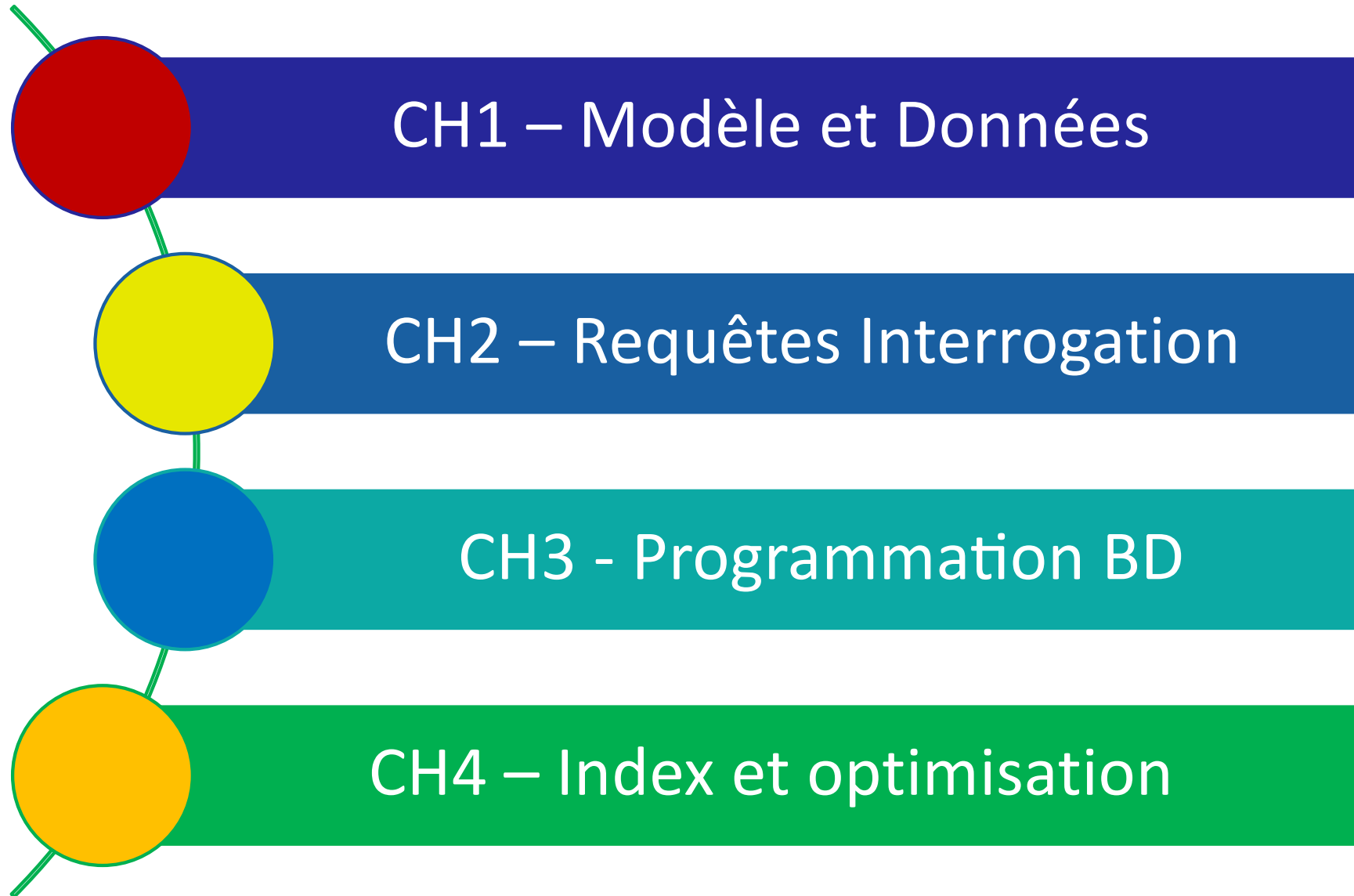
Bases de données et Optimisation

CH3 – Programmation BD



Evelyne VITTORI
vittori_e@univ-corse.fr

Optimisation et Bases de données



CH3- Programmation BD

Objectifs

- Découvrir la programmation procédurale « bases de données »
 - Procédures et fonctions stockées programmes en PL/pgSQL
 - Triggers



Comment dialoguer avec un SGBDr?

■ Requêtes

- Chaînes de caracteres interprétées par le SGBD
- Langage de requête relationnel
- SQL (DDL, DML, ...)

■ Appel de Procédures stockées

- Programmes compilés stockés sur le SGBD
- Langage de programmation BD
 - Langage procédural spécifique à un SGBD
 - Ex: PL/pgSQL Oracle – PL/PGSQL PostGres

Comment dialoguer avec un SGBD?

- **Interface interactive**

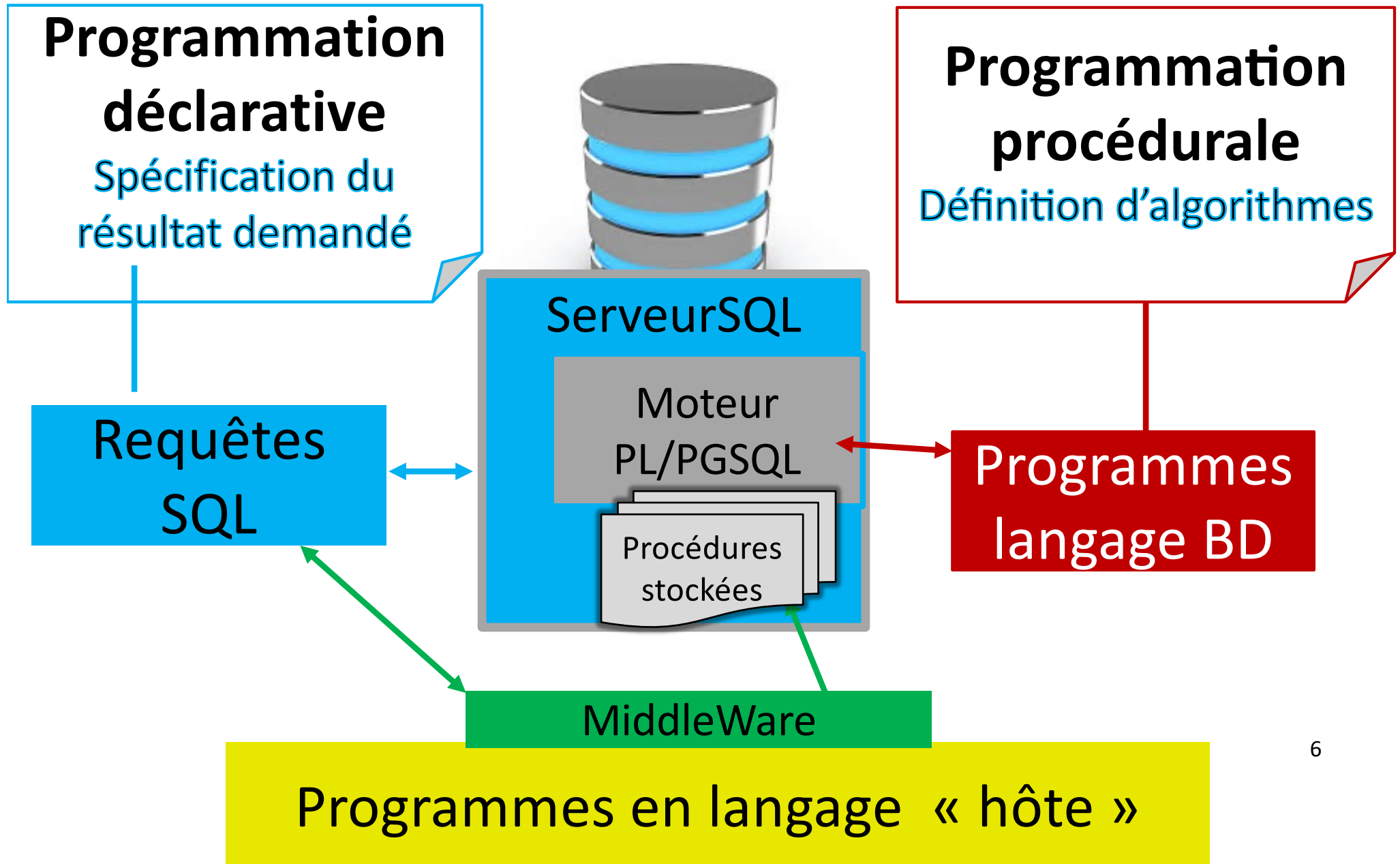
- Ex: client graphique

par ex phpMyAdmin - MySQL, SQLDeveloper – Oracle, pgAdmin -Postgres

- **Programme en langage hôte**

- Langage programmation classique (Java, Python, php, python, ...)
 - API d'accès aux données
 - Ex: ODBC, JDBC,

Requêtes et Programmes



Langage de programmation BD

- Langage procédural spécifique à un SGBD
 - PL/SQL Oracle
 - PL/PGSQL PostGres
 - SQL/PSM (PersistentStoredModules) sous MySQL
- Amélioration des performances
 - Code compilé exécuté sur le moteur de BD :

Notion de procédure stockée

CH3 – Programmation BD

1. Introduction à la programmation PL/pgSQL
2. Définition de blocs PL/pgSQL anonymes
3. Manipulation de curseurs
4. Procédures et fonctions stockées
5. Triggers





Introduction à la programmation PL/PGSQL

- Notion de procédure stockée
- Le langage PL/PGSQL



Qu'est-ce qu'une procédure stockée?

- Suite d'instructions désignée par un nom, compilée et stockée de manière durable dans la base de données

Disponible sous Oracle, PostgreSQL, MySQL depuis la version 5

- Une procédure stockée peut contenir:
 - Des requêtes SQL
 - Des blocs d'instructions dans le langage procédural du SGBD

```
BEGIN
  instructions
END
```

PL/pgSQL sous Oracle,
PL/PGSQL sous PostgreSQL

Pour ou contre les procédures stockées

■ Avantages

- Améliorer les **performances** en réduisant les allers-retours entre le client et le serveur
- **Sécuriser** la base de données en limitant les possibilités de requetage aux procédures stockées
- **Faciliter** l'utilisation de la BD sur plusieurs plateformes

■ Inconvénients

- Surcharger le serveur
- Syntaxes différentes selon le SGBD choisi (portage problématique)

Qu'est ce que PL/pgSQL?

Procedural Language/PostgreSQL Structured Query Language

- Un langage **procédural** qui intègre des commandes SQL
 - Manipulation des données (DML)
 - Gestion de transactions
- Un véritable langage de **programmation structurée**
 - variables, types, structures de contrôle
 - procédures, fonctions
- Un langage avec des concepts spécifiques pour manipuler les requêtes SQL (**curseurs**, ...) ¹²

≠ SQL qui est un langage déclaratif

POO
intégrée à
partir de la
version 8

Pourquoi PL/pgSQL?

- Encapsuler l'exécution des requetes fréquentes.
- Certaines **vérifications** ne peuvent pas être définies par des contraintes standards:
 - Exemple : *toute nouvelle croisière doit avoir une date de début supérieure à la date de fin des autres croisières effectuées par le même bateau.*
- L'insertion, la suppression ou la MAJ de certaines données peut nécessiter des **calculs** préalables.
- L'utilisation de fonctions personnalisées dans les requêtes peut être utile .

Triggers

Procédures et fonctions
stockées

Trois Types de Bloc PL/pgSQL

Anonyme

```
DO $$  
DECLARE  
BEGIN  
    ....  
EXCEPTION  
END $$;
```

Procedure

```
PROCEDURE nomP (p1  
    type1, ...)  
IS  
BEGIN  
    ...  
EXCEPTION  
END;  
$$ LANGUAGE plpgsql;
```

+ triggers

Function

```
FUNCTION nomF (p1  
    type1, ...)  
RETURNS type  
AS $$  
BEGIN  
    ...  
    RETURN expr;  
EXCEPTION  
END;  
$$ LANGUAGE plpgsql;
```



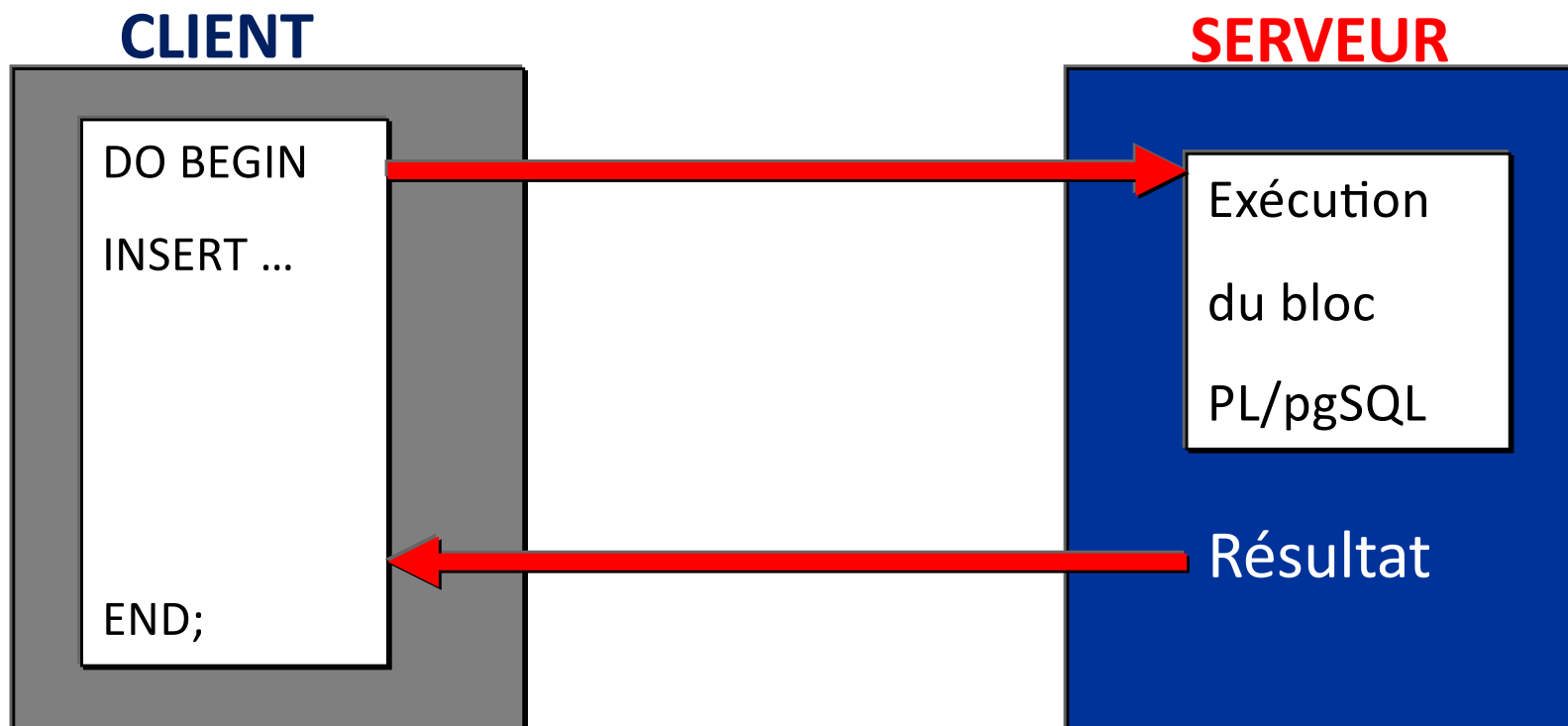

Blocs PL/pgSQL anonymes



UNIVERSITÀ DI CORSICA
PASQUALE PAOLI

Principe d'exécution d'un Bloc anonyme PL/pgSQL

- Le bloc d'instructions est envoyé sur le serveur. Celui-ci exécute le bloc et renvoie un résultat final.



Structure d'un bloc PL/pgSQL anonyme

DO \$\$

[DECLARE

--déclarations de variables, constantes

-- exceptions et curseurs]

BEGIN [nombloc]

--instructions

[EXCEPTIONS *-- Traitement des exceptions*]

END; --ou END nombloc

END \$\$; ←

A ajouter obligatoirement
dans l'exécution d'un script

Affectation et Structures de contrôle

- Affectation
 - variable:= valeur
- IF *condition* THEN ... ELSIF *condition* THEN ... ELSE ... END IF ;
- WHILE *condition* LOOP ... END LOOP ;
- FOR *compteur* IN min .. max
 - LOOP ... END LOOP ;
- LOOP ... EXIT WHEN *condition* END LOOP ;

Déclaration de variables

■ DECLARE

variable1 type1;

variable2 type2 :=valeur;

- types SQL
- types structurés (tableaux record)
- référence à un type du dictionnaire de données

```
DECLARE
```

```
    NOM          CHAR(15);
```

```
    SALAIRE      NUMERIC(7,2);
```

```
    EMBAUCHE     DATE ;
```

```
    REPONSE      BOOLEAN;
```

Remarque :PL/pgSQL n'est pas sensible à la casse

Types référant le dictionnaire de données

- Type d'une **colonne** existante
 - nom_var **TABLE.COLONNE%TYPE**
- type record correspondant à une ligne d'une table existante
 - nom_var2 **TABLE%ROWTYPE**
- Type d'une variable existante
 - nom_var 3 **nom_var1%TYPE**

type record
(colonnes de la
table)


```
NOM      EMP.ename%TYPE;  
ENREG    EMP%ROWTYPE;  
COMMI    NUMERIC(7,2);  
SALAIRE  COMMI%TYPE;
```

Record (enregistrements)

DECLARE

```
LIGNETU RECORD;
```

déclaration d'une
variable de type
record



BEGIN


```
LIGNETU.nom:='Toto';
```

```
LIGNETU.moyenne:=18;
```

```
-- Affichage
```

```
RAISE NOTICE 'Nom: %, Moyenne: %',  
                LIGNETU.nom, LIGNETU.moyenne;
```

Affichage à
l'écran



END

Tableaux


DO \$\$

DECLARE

TABNOTES NUMERIC(4,2)[];

i INTEGER;

déclaration d'une
variable de type
tableau



BEGIN

i:=1;

-- Initialisation du tableau

TABNOTES := ARRAY[12.5, 13.0, 15.5];

FOR i IN 1 .. array_length(tabnotes, 1) LOOP

RAISE NOTICE 'Valeur dans tabnotes[%] : %', i, tabnotes[i];

END LOOP;

END \$\$;

SELECT INTO

Pour récupérer les données
d'une sélection mono-ligne

- Clause permettant de récupérer le résultat d'une requête dans une (ou des) variables

```
SELECT col1, col2, . . . INTO var1, var2, . . .  
FROM table  
WHERE .....
```

ATTENTION!

Le SFW ne doit retourner qu'une
seule ligne

SELECT INTO

Exemple

```
DO $$  
DECLARE  
    NOMSKIP          VARCHAR(50);  
    SAL              SKIPERS.SALAIRE%TYPE  
    PORT             SKIPERS.SKPORT%TYPE;  
BEGIN  
    SELECT SKNOM, SKPORT, SALAIRE  
        INTO NOMSKIP, PORT, SAL  
    FROM SKIPERS  
    WHERE SKNUM='1';  
    ...  
END $$;
```


Exemple

Variables de type %ROWTYPE

```
DO $$
```

```
DECLARE
```

```
    V_BATEAUX BATEAUX%ROWTYPE;
```

```
BEGIN
```

```
    SELECT * INTO V_BATEAUX  
    FROM BATEAUX
```

```
    WHERE BATNUM='B001';
```

```
    RAISE NOTICE 'Nom: %', V_BATEAUX.BATNOM;
```


```
    RAISE NOTICE 'Port: %', V_BATEAUX.BATPORT;
```

```
    RAISE NOTICE 'Capacité: %', V_BATEAUX.CAPACITE;
```

```
END $$;
```

NOTICE:	Nom: LIBERTE
NOTICE:	Port: ANTIBES
NOTICE:	Capacité: 10

Affichage des valeurs
de la ligne



Exercice 1 –

- Définissez un bloc anonyme affichant les caractéristiques du skipper barrant la croisière C001 sous la forme suivante :

```
Numéro: 1  
Nom: JEAN  
Port d'attache: AJACCIO  
Salaire: 3000
```





Manipulation de curseurs



UNIVERSITÀ DI CORSICA
PASQUALE PAOLI

Notion de Curseur

Pour récupérer les données
d'une sélection multi-ligne

- Un curseur est un pointeur sur un ensemble de lignes correspondant au résultat d'une requête SFW
- Deux types de curseurs:
 - curseur **implicite**: généré et géré par le noyau pour chaque ordre SQL d'un bloc
 - curseur **explicite**: défini par l'utilisateur



1. Déclaration
2. Ouverture du curseur
3. Traitement des lignes
4. Fermeture du curseur

Déclaration d'un curseur explicite

- **DECLARE**

- *nom_curseur* **CURSOR FOR** *SFW* ;

```
DO $$  
DECLARE  
    SKIPAJA CURSOR FOR  
        SELECT SKNOM, SALAIRE FROM SKIPPER  
        WHERE SKPORT='AJACCIO'  
        ORDER BY SALAIRE;  
  
BEGIN  
    ...  
  
END $$;
```

Ouverture d'un curseur

- **BEGIN**

OPEN nom_curseur;

- L'ouverture déclenche:

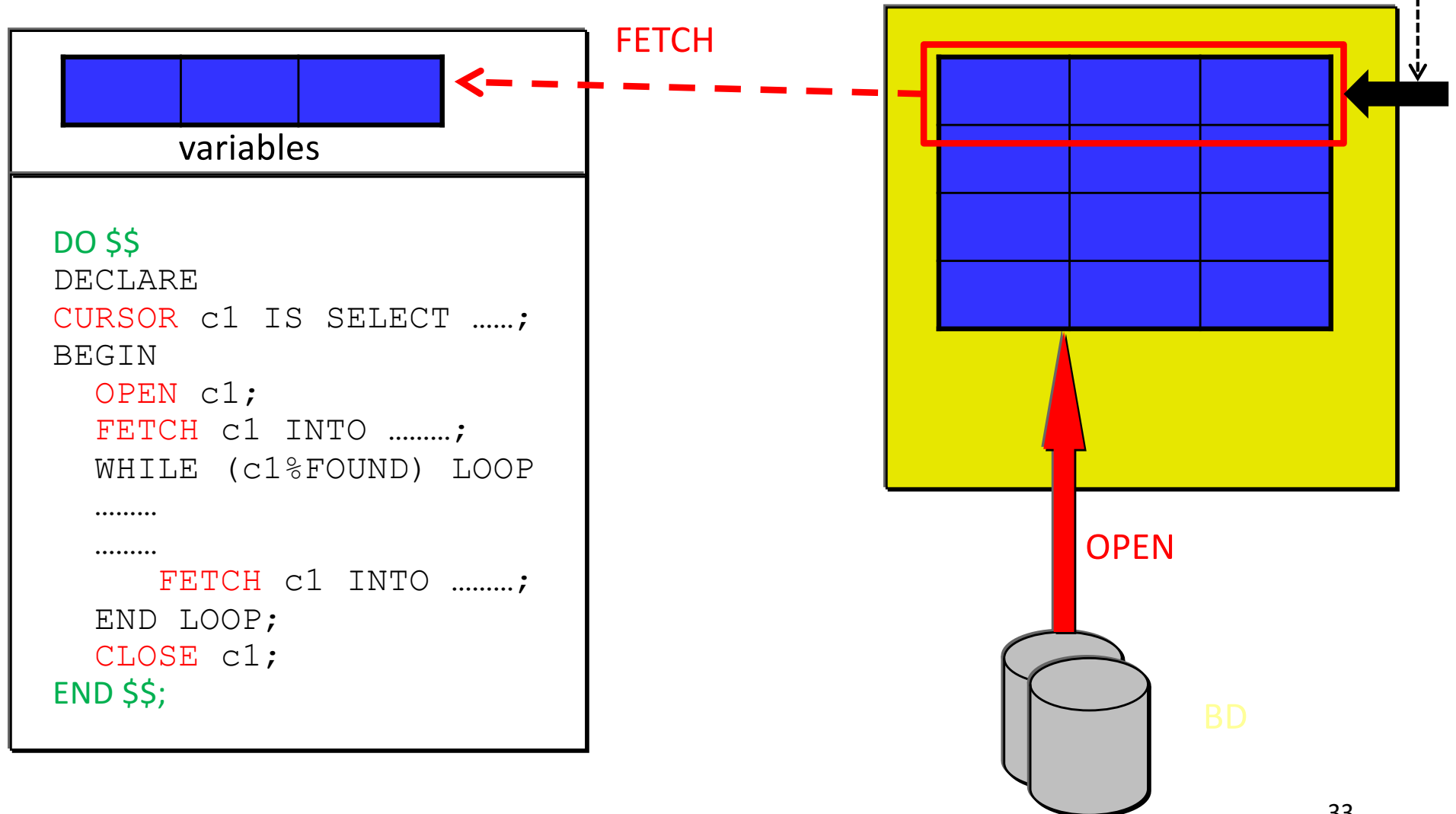
- Allocation mémoire pour les lignes du curseur
- Analyse syntaxique et sémantique du SFW
- Positionnement de verrous éventuels

Parcours d'un curseur

- Instruction **FETCH**
 - Parcourt une à une les lignes du curseur
 - Située en général dans une boucle
- la valeur de chaque colonne doit être stockée dans une **variable réceptrice**
- FETCH permet d'avancer le pointeur du curseur sur la ligne suivante et de récupérer cette ligne
FETCH nom_curseur **INTO** liste_variables;

A tout moment le curseur pointe sur une ligne spécifique (la ligne courante)

Parcours d'un curseur



Attributs d'un curseur

Indicateurs permettant de connaître l'état d'un curseur.

■ FOUND

- TRUE: le curseur pointe sur une ligne
- FALSE: le parcours est terminé

le dernier FETCH n'a pas ramené de ligne

■ NOT FOUND

- TRUE: le parcours est terminé

BOUCLES DE PARCOURS

WHILE FOUND

LOOP....END LOOP;

OU

LOOP ...

EXIT WHEN NOTFOUND

END LOOP;

Exemple de parcours d'un curseur

```
DO $$  
DECLARE  
    SKIPAJA CURSOR FOR  
        SELECT SKNOM, SALAIRE FROM SKIPPERS  
        WHERE SKPORT='AJACCIO' ORDER BY SALAIRE;  
    NOM SKIPPERS.SKNOM%TYPE; SAL SKIPPERS.SALAIRE%TYPE;  
BEGIN  
    OPEN SKIPAJA;  
    LOOP  
        FETCH SKIPAJA INTO NOM, SAL;  
        EXIT WHEN NOT FOUND;  
        IF SAL > 2500 THEN  
            INSERT INTO RESULTAT VALUES (NOM,SAL);  
        END IF;  
    END LOOP;  
    CLOSE SKIPAJA;  
END $$;
```


variables réceptrices



table préalablement créée



Fermeture du curseur
Libération mémoire



Boucle simplifiée de parcours d'un curseur

```
FOR varLigne IN nom_curseur  
LOOP  
...  
END LOOP ;
```



Variable de type RECORD

- Pas de déclaration de variables réceptrices
- Pas d'ouverture
- Pas de Fetch
- Pas de close

Exemple de parcours simplifié

```
DO $$  
DECLARE  
    SKIPAJA CURSOR FOR  
        SELECT SKNOM, SALAIRE FROM SKIPPERS  
        WHERE SKPORT='AJACCIO' ORDER BY SALAIRE;  
BEGIN  
    FOR L IN SKIPAJA LOOP  
        IF L.SALAIRE > 2500 THEN  
            INSERT INTO RESULTAT VALUES (L.SKNOM,L.SALAIRE);  
        END IF;  
    END LOOP;  
END $$;
```

Variable RECORD de réception
(n'a pas à être déclarée)
ici 2 champs: SKNOM, SALAIRE

Exemple de parcours avec calculs

```
DO $$  
DECLARE  
  C1 CURSOR FOR SELECT SKNOM, SALAIRE FROM SKIPPERS;  
  E1 INTEGER := 0;  
  E2 INTEGER := 0;  
BEGIN  
  FOR LIGNE IN C1 LOOP  
    IF LIGNE.SALAIRE < 2000 THEN  
      E1 := E1 + 1;  
      INSERT INTO LISTE_BAS_SAL (SKNOM, SALAIRE) VALUES (LIGNE.SKNOM, LIGNE.SALAIRE);  
    ELSE  
      E2 := E2 + 1;  
      INSERT INTO LISTE_HAUT_SAL (SKNOM, SALAIRE) VALUES (LIGNE.SKNOM, LIGNE.SALAIRE);  
    END IF;  
  END LOOP;  
  RAISE NOTICE 'Nombre de salaires < 2000 : %', E1;  
  RAISE NOTICE 'Nombre de salaires >= 2000 : %', E2;  
END $$;
```

Insertion de lignes dans LISTE_BAS_SAL et
LISTE_HAUT_SAL préalablement créées

```
NOTICE:  Nombre de salaires < 2000 : 1  
NOTICE:  Nombre de salaires >= 2000 : 3
```

Curseur paramétré

```
DO $$  
DECLARE  
  NOMSKIP SKIPPER.SKNOM%TYPE;  
  
  C_SKIP CURSOR ( NUMSKIP SKIPPER.SKNUM%TYPE ) FOR  
  SELECT SKNOM FROM SKIPPER WHERE SKNUM = NUMSKIP;  
  
BEGIN  
  OPEN C_SKIP ( '2' );  
  FETCH C_SKIP INTO NOMSKIP;  
  RAISE NOTICE 'NOM DU SKIPPER 1 : %',NOMSKIP;  
  CLOSE C_SKIP;  
END $$;
```

paramètre

NOTICE: NOM DU SKIPPER 1 : PAUL

Mises à jour avec Curseur

```
DO $$  
DECLARE  
    C CURSOR FOR SELECT SKNUM, SALAIRE  
      FROM SKIPPERS FOR UPDATE OF SKIPPERS;  
    NB INTEGER :=0;  
BEGIN  
    FOR I IN C LOOP  
        IF I.SALAIRE < 2500 THEN  
            UPDATE SKIPPERS SET SALAIRE = SALAIRE*1.3  
            WHERE CURRENT OF C;  
            NB:=NB+1;  
        END IF;  
    END LOOP;  
    RAISE NOTICE ' % salaires modifiés ', NB;  
END $$;
```

Le curseur pourra
être utilisé
directement pour
des MAJ

Modifie la ligne
courante du
curseur

Exercice 2 –

- Définissez un bloc anonyme affichant les noms des skippers effectuant une croisière au départ d'Ajaccio sous la forme suivante :



```
Skippers au départ d'Ajaccio : LAURA  JEAN
```



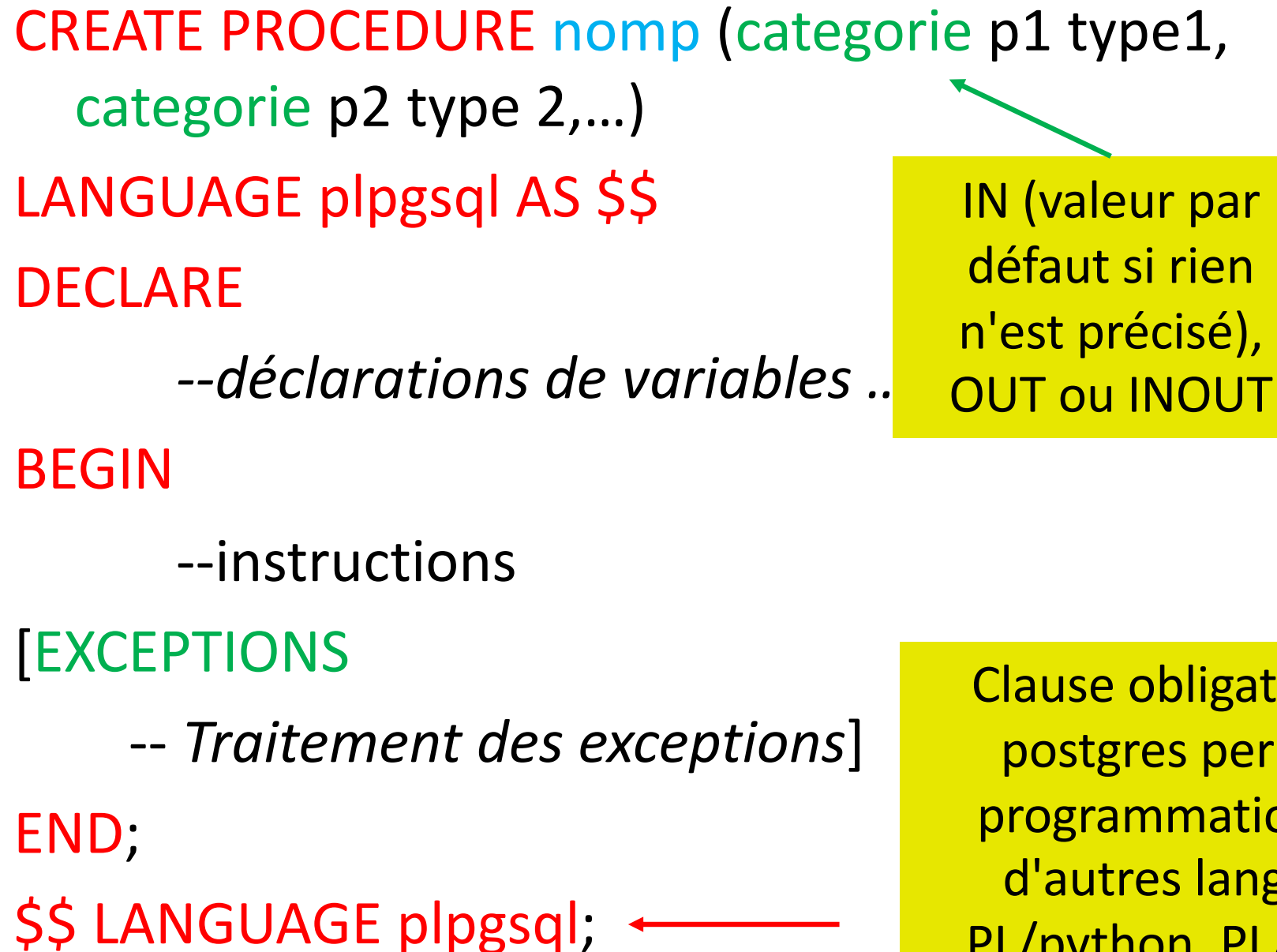

Procédures et fonctions stockées



UNIVERSITÀ DI CORSICA
PASQUALE PAOLI

Structure d'une procédure stockée

```
CREATE PROCEDURE nomp (categorie p1 type1,  
    categorie p2 type 2,...)  
LANGUAGE plpgsql AS $$  
DECLARE  
    --déclarations de variables ..  
BEGIN  
    --instructions  
[EXCEPTIONS  
    -- Traitement des exceptions]  
END;  
$$ LANGUAGE plpgsql;
```



The diagram illustrates the syntax of a stored procedure. A green arrow points from the 'categorie' parameter in the parameter list to a yellow box explaining its mode. A red arrow points from the 'LANGUAGE plpgsql;' line to another yellow box explaining its necessity and alternatives.

IN (valeur par défaut si rien n'est précisé), OUT ou INOUT

Clause obligatoire car postgres permet la programmation dans d'autres langages : PL/python, PL/PERL ...

Structure d'une fonction stockée

```
CREATE FUNCTION nomf (p1 type1,  
    p2 type 2,...) RETURNS type AS $$
```

```
DECLARE
```

```
    --déclarations de variables, constantes
```

```
BEGIN          --instructions
```

```
    RETURN ...;
```

```
END;
```

```
$$ LANGUAGE plpgsql;
```

Clause obligatoire car
postgres permet la
programmation dans
d'autres langages :
PL/python, PL/PERL ...

Exemple de fonction stockée

```
create or replace FUNCTION  NBSKIPPER() RETURNS INTEGER AS $$  
DECLARE  
    NB INTEGER;  
BEGIN  
    SELECT COUNT(SKNUM) INTO NB  
    FROM SKIPPERS;  
    RETURN NB;  
END;  
$$ LANGUAGE plpgsql;
```

Tests de la fonction
✓ dans un bloc anonyme
✓ dans une requête

SELECT **NBSKIPPER**();

```
DO $$  
BEGIN  
    RAISE NOTICE 'Nombre de skippers :  
%', NBSKIPPER();  
END $$;
```

```
SELECT SKNUM, COUNT(1)  
FROM CROISIERES  
GROUP BY SKNUM  
HAVING COUNT(1)>NBSKIPPER()/2;
```

Paramètres des procédures et fonctions stockées

- Un paramètre est caractérisé par un **NOM**, un **TYPE** (INTEGER, VARCHAR, ect...) et une **CATEGORIE**.
- On distingue 3 catégories de paramètres:
 - IN : paramètre **donnée** ou « entrant ». Sa valeur doit être donnée lors de l'appel de la procédure et il n'est pas modifié dans son corps
 - OUT : paramètre **résultat** ou « sortant ». Sa valeur va être calculée dans la procédure.
 - INOUT : paramètre **donnée-résultat**. Sa valeur doit être donnée lors de l'appel de la procédure mais elle est modifiée dans la procédure.

Exemple de procédure stockée paramétrée

```
CREATE OR REPLACE PROCEDURE SUPP_SKIP (IN NUM VARCHAR)
LANGUAGE plpgsql AS $$
BEGIN
    DELETE FROM CROISIERES WHERE SKNUM = NUM;
    DELETE FROM SKIPPERS WHERE SKNUM = NUM;
END;
$$;
```

Tests de la procédure



```
DO $$
BEGIN
    CALL SUPP_SKIP ('6');
END $$;
```

Procédures et fonctions stockées: ce qu'il faut éviter

- Eviter les invocations de fonctions stockées dans les clauses where des requêtes:
 - Exécution pour chaque ligne!
- Eviter l'utilisation intensive de curseurs (pénalisent les performances)
 - Préférer la définition de requêtes déclaratives dès que cela est possible

Exercice 3 – fonction moyenne

- Définissez une fonction stockée MOYSAL qui renvoie la moyenne des salaires de la table SKIPPER.
- Testez le fonctionnement de votre fonction en l'invoquant dans un bloc anonyme faisant afficher son résultat.



```
moyenne salaires : 2375 euros
```




Déclencheurs (Triggers)



UNIVERSITÀ DI CORSICA
PASQUALE PAOLI

Notion de déclencheur

- Un déclencheur (ou Trigger) est un bloc d'instructions:
 - associé à une table,
 - exécuté automatiquement juste avant ou juste après une opération de mise à jour:
 - insertion, suppression ou modification.
- Il est associé à un **évènement déclencheur** (Insert, Update ou Delete)
- Il est exécuté globalement ou **pour chaque ligne** mise à jour.

Que peut faire un trigger?

Un **trigger** est une procédure stockée attachée à une table et à un évènement.

Il peut:

- Lire et modifier la ligne sur laquelle il a été déclenché
- Lire, modifier et/ou insérer des données dans une table quelconque qui n'est pas utilisée dans requête qui l'a déclenché

Autres types de triggers (non disponibles dans PostgreSQL)

Avec Oracle8i, un trigger peut aussi être déclenché dans d'autres cas:

- Lors d'évènements liés à la manipulation de la base de données ou du schéma
 - démarrer, arrêter, créer un objet schéma
- Lors d'évènements liés à des actions de l'utilisateur
 - se connecter, se déconnecter

Exemples d'utilisation de triggers

- Avant une modification ou une insertion
 - Vérifier les données
 - Faire des corrections sur les données
 - Faire échouer la requête en cas de problème
- Faire des vérifications avant suppression en cas de conditions complexes
- Tenir à jour un historique en insérant des données dans une table spécifique à chaque opération de mise à jour

Deux types de triggers

- Triggers exécutés pour chaque ligne impactée par la mise à jour

- Triggers de **type ligne**

Row-level triggers

- Dans ces triggers, on peut manipuler explicitement les nouvelles valeurs de la ligne modifiée

- *Exemple : Si une requête UPDATE modifie 100 lignes, un trigger de type ligne s'exécutera 100 fois, une fois pour chaque ligne modifiée.*

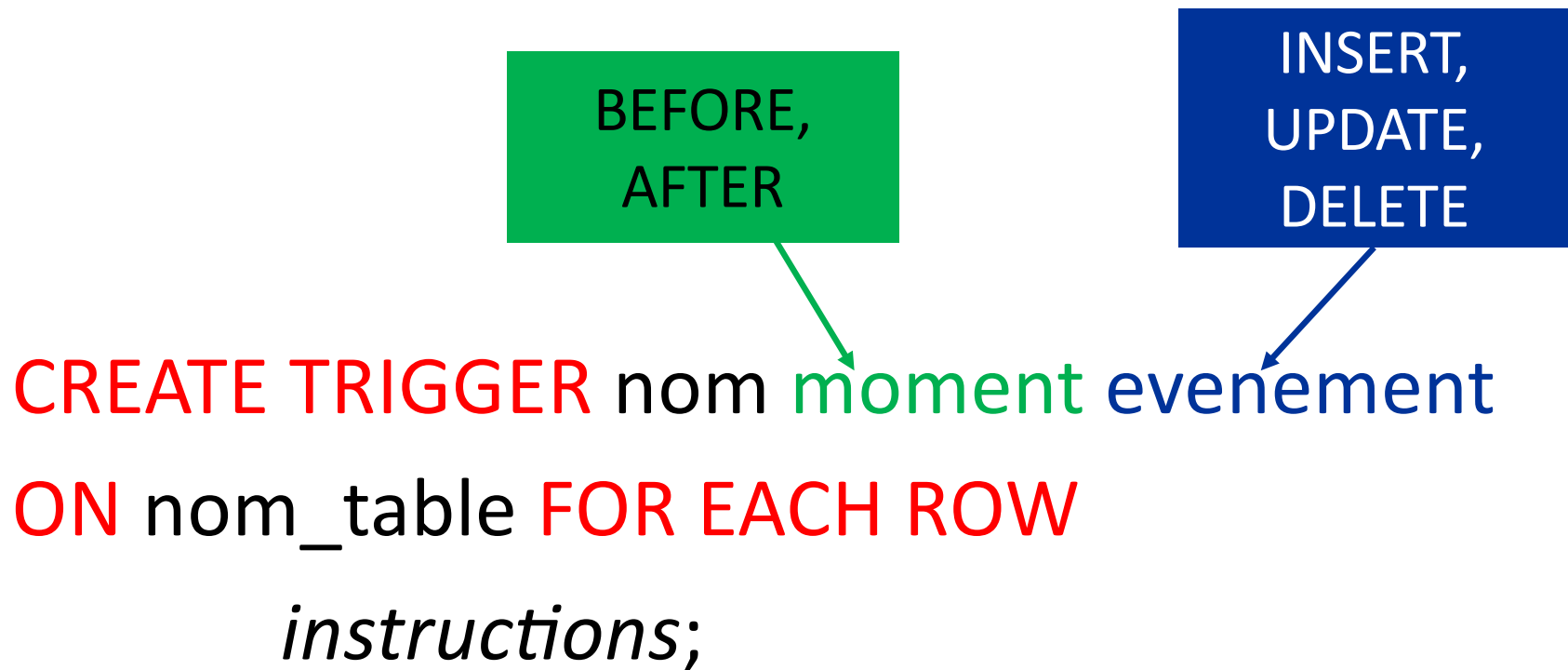
- Triggers exécutés globalement lors d'une mise à jour

- Triggers de **type Instruction**

Statement-level triggers

- *Exemple : Si une requête UPDATE modifie 100 lignes, un trigger de type instruction s'exécutera une seule fois.*

Définition d'un trigger ligne



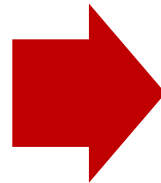
- PostgreSQL: Dans pgAdmin, fenetre spécifique (Clic droit sur la table, puis créer → déclencheur)
- Oracle: Dans sqldeveloper, onglet spécifique associé à une table
- MySQL: Dans php myAdmin, fenêtre spécifique

SPECIFICITES DES TRIGGERS

postgreSQL

- Le code d'un trigger doit être défini dans une fonction spéciale : "trigger function"
- Deux étapes pour définir un trigger :

Définition de la
fonction trigger



Définition du trigger

```
CREATE FUNCTION nom_fonction_trig ()  
RETURNS TRIGGER AS $$  
BEGIN  
    --instructions ...  
    RETURN NEW;  
END;  
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER nom_trigger  
BEFORE INSERT OR UPDATE OF  
nom_colonne ON nom_table  
FOR EACH ROW  
EXECUTE FUNCTION nom_fonction_trig();
```


Manipulation de la ligne courante

- Le trigger peut manipuler les valeurs des colonnes de la ligne sur laquelle il a été déclenché.
- **OLD.col** désigne la valeur de la colonne col avant la mise à jour (:OLD.col en Oracle)
- **NEW.col** désigne la nouvelle valeur de la colonne col après la mise à jour (:NEW.col en Oracle)

Pour pouvoir modifier la valeur d'une colonne, le trigger doit être exécuté avant la mise à jour (BEFORE)

Exemple de fonction trigger

Vérification avant insertion

```
CREATE OR REPLACE FUNCTION force_salaire_min()  
RETURNS TRIGGER AS $$  
BEGIN  
    IF NEW.salaire < 1500 THEN  
        NEW.salaire := 1500;  
    END IF;  
    RETURN NEW;  
END;  
$$ LANGUAGE plpgsql;
```

Mise en place d'un
salaire minimum

Si le salaire est inférieur à 1500 euros, il est automatiquement
initialisé à 1500 euros

Exemple de trigger

Vérification avant insertion

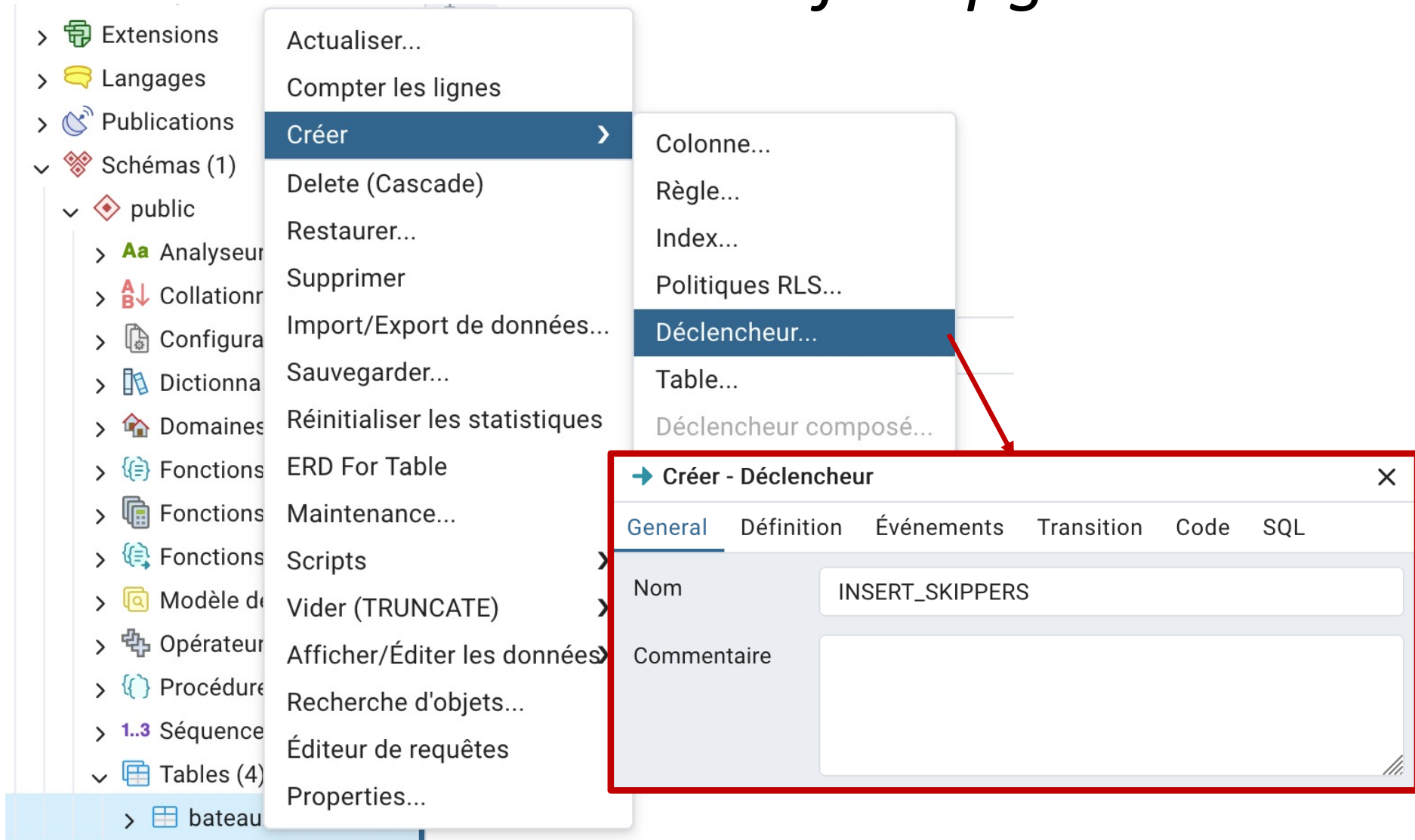
```
CREATE TRIGGER insert_skippers  
BEFORE INSERT OR UPDATE OF salaire ON skippers  
FOR EACH ROW  
EXECUTE FUNCTION force_salaire_min() ;
```

Mise en place d'un
salaire minimum

Si le salaire est inférieur à 1500 euros, il est automatiquement
initialisé à 1500 euros

Exemple de trigger

Utilisation de l'interface pgAdmin



Exemple de trigger

Utilisation de l'interface pgAdmin

→ insert_skippers

General Définition Événements Transition Code SQL

Déclencheur activé ? Activer

Déclencheur niveau ligne ? ☒

Déclencheur contrainte ? ☐

Différable ? ☐

Différable ? ☐

Fonction déclencheur public.force_salaire_min

Arguments

force_salaire_min()

General Définition Code Options Paramètres Sécurité SQL

```
1
2 BEGIN
3     IF NEW.salaire < 1500 THEN
4         NEW.salaire := 1500;
5     END IF;
6     RETURN NEW;
7 END;
```

→ Créer - Déclencheur

General Définition Événements Transition Code SQL

Déclencheur BEFORE

Événements

INSERT ☒

UPDATE ☒

DELETE ☐

TRUNCATE ☐

Quand 1

Colonnes salaire x

Fermer Réinitialiser Enregistrer

Exercice 4



```
CREATE TABLE EXERCICE(x NUMERIC, y NUMERIC, z NUMERIC);
```

```
CREATE OR REPLACE FUNCTION modif()
```

```
RETURNS TRIGGER AS $$
```

```
BEGIN
```

```
    NEW.x := OLD.y; NEW.y := OLD.z;
```

```
    RETURN NEW;
```

```
END;
```

```
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER BEF_UPD_EXERCICE BEFORE UPDATE ON EXERCICE
```

```
FOR EACH ROW
```

```
    EXECUTE FUNCTION modif();
```

```
INSERT INTO EXERCICE(x,y,z) VALUES (1,2,3);
```

```
UPDATE EXERCICE SET x=x+2, y=y+2, z=z+2;
```

```
UPDATE EXERCICE SET x=x+1, y=y+1, z=z+1;
```

Après avoir exécuté ce code,
quel sera le résultat affiché
par la requête suivante :

```
SELECT x, y, z FROM  
EXERCICE;
```

Arrêt de transaction dans un trigger

- Les triggers permettent de tester la validité d'une mise à jour et de la bloquer en cas d'erreur
- Levée d'une exception bloquant la transaction

RAISE EXCEPTION message

Message d'erreur à afficher

Exemple de mise en échec d'une insertion

```
CREATE OR REPLACE FUNCTION verif_dates ()  
RETURNS TRIGGER AS $$  
BEGIN  
    IF (NEW.DEPPDATE>NEW.ARRDATE) THEN  
        RAISE EXCEPTION 'Dates incorrectes';  
    END IF;  
    RETURN NEW;  
END;  
$$ LANGUAGE plpgsql;
```

Vérification de la cohérence des
dates d'une croisière

```
CREATE TRIGGER insert_crois  
BEFORE INSERT OR UPDATE OF ARRDATE, DEPPDATE ON CROISIERES  
FOR EACH ROW  
EXECUTE FUNCTION verif_dates ();
```


Exemple de trigger d'insertion avec vérifications plus complexes

```
CREATE OR REPLACE FUNCTION verifier_disponibilite_bateau()  
RETURNS TRIGGER AS $$  
DECLARE  
    nbBat INTEGER;  
BEGIN  
    -- Recherche de conflit bateau  
    SELECT COUNT(CROISNUM) INTO nbBat  
    FROM CROISIERES  
    WHERE BATNUM = NEW.BATNUM  
        AND ((DEPDATE <= NEW.DEPDATE AND ARRDATE >= NEW.DEPDATE)  
            OR (DEPDATE >= NEW.DEPDATE AND DEPDATE <=  
NEW.ARRDATE)) ;  
  
    IF nbBat > 0 THEN  
        RAISE EXCEPTION 'Bateau Incompatible avec % croisières  
existantes', nbBat;  
    END IF;  
  
    RETURN NEW;  
END;
```








Vérification que le bateau n'est pas déjà
impliqué dans des croisières aux mêmes
dates

Exemple de trigger d'insertion avec vérifications plus complexes

Vérification que le bateau n'est pas déjà impliqué dans des croisières aux mêmes dates

```
CREATE TRIGGER insert_crois  
BEFORE INSERT OR UPDATE OF ARRDATE, DEPDATE ON CROISIERES  
FOR EACH ROW  
EXECUTE FUNCTION verifier_disponibilite_bateau();
```

Test du trigger d'insertion

croisnum [PK] character varying (5) 	deport character varying (50) 	arrport character varying (50) 	depdate date 	arrdate date 	batnum character varying (5) 	sknum character varying (5) 
C010	BASTIA	NICE	2024-08-02	2024-08-04	B001	1
C001	BASTIA	CALVI	2024-07-10	2024-07-11	B002	1
C002	ANTIBES	AJACCIO	2024-07-15	2024-07-16	B002	2
C003	AJACCIO	BASTIA	2024-08-21	2024-08-22	B002	1
C004	CALVI	MARSEILLE	2024-09-02	2024-08-22	B002	1
C005	AJACCIO	CALVI	2024-10-10	2024-10-11	B002	1
C007	BASTIA	MARSEILLE	2024-08-02	2024-08-03	B002	1
C008	BASTIA	MARSEILLE	2024-10-02	2024-10-03	B002	1
C009	AJACCIO	MARSEILLE	2024-08-02	2024-08-03	B002	1

Le bateau B001 est déjà impliqué dans une croisière aux dates données



Bateau Incompatible avec 1 croisières existantes CONTEXT: PL/pgSQL function verifier_disponibilite_bateau() line 13 at RAISE



Tests d'évènement dans un trigger

- Pour les triggers traitants d'évènements multiples (ex: Insert ou Update), il est possible de tester l'évènement considéré:

- INSERT, DELETE, UPDATE

Dans postgresSQL il faut utiliser la variable TG_OP pour définir ces tests

- Exemple

```
IF TG_OP = 'INSERT' THEN
```

```
    -- Logique pour l'insertion
```

```
ELSIF TG_OP = 'UPDATE' THEN
```

```
    -- Logique pour la modification
```

Exercice 5 – Trigger SKIPPERS

- Définissez un trigger **avant Insertion et modification** pour la table SKIPPERS afin que dans le cas où le salaire du skipper à insérer s'avère inférieur à la moyenne des salaires des skippers, le salaire prenne automatiquement la valeur de cette moyenne.
- Contrainte: le trigger devra faire appel à la fonction MOYSAL définie précédemment



Exercice 6 – Trigger CROISIERES



- Définissez un trigger (et sa fonction associée) **avant insertion et modification** pour la table CROISIERES afin qu'il soit impossible d'insérer une nouvelle croisière si les dates de celle-ci ne sont pas compatibles avec les disponibilités du **skipper** et du **bateau**.
- Si des conflits sont trouvés, l'insertion doit être annulée et un message de la forme suivante affiché:
 - Par exemple
'Skipper Incompatible avec 10 croisières existantes'

Exercice 7 – table HISTORIQUE (1/2)

- Définissez une table HISTORIQUE permettant de sauvegarder l'historique des mises à jour sur la table CROISIERES:
- La table HISTORIQUE aura la structure suivante:
 - id: numérique auto incrémenté (clé primaire)
 - nom_table: nom de la table concernée (chaîne de caractères)
 - type_maj: Insertion (INS), Modification(MOD) ou suppression (SUP)(chaîne de caractères)
 - date_maj: date



Exercice 7 – table HISTORIQUE (2/2)

- On souhaite à présent faire en sorte que chaque opération de mise à jour sur la table CROISIERES donne lieu à la création d'une ligne dans la table HISTORIQUE.
- Définissez-les fonctions et triggers nécessaires.
- Testez le fonctionnement de vos triggers en effectuant plusieurs insertions, modifications et suppressions dans la table CROISIERES et en visualisant la table HISTORIQUE.



Définissez une seule fonction trigger pour les 3 événements (avec test)

Définition d'un trigger de type instruction

- Les triggers de type instruction sont exécutés une seule fois lors d'une mise à jour même si plusieurs lignes sont modifiées.
- La syntaxe est similaire avec l'option "**FOR EACH STATEMENT**" à la place de "**FOR EACH ROW**".
- Ces triggers ne peuvent pas référencer les nouvelles valeurs (NEW.colonne).

Exemple

```
CREATE FUNCTION check_croisieres_before_delete()  
RETURNS TRIGGER AS $$  
DECLARE  
    total_croisieres INT;  
BEGIN  
    -- Compter le nombre de croisières  
    SELECT COUNT(*) INTO total_croisieres FROM CROISIERES;  
  
    IF total_croisieres = 0 THEN  
        RAISE EXCEPTION 'Vous ne pouvez pas supprimer  
toutes les croisières. Il doit en rester au moins une.';  
    END IF;  
  
    RETURN NULL;  
END;  
$$ LANGUAGE plpgsql;
```

Que fait ce trigger?
Pourquoi est-ce un trigger de type
instruction?

```
CREATE TRIGGER before_delete_croisieres  
BEFORE DELETE ON CROISIERES  
FOR EACH STATEMENT  
EXECUTE FUNCTION check_croisieres_before_delete();
```

Triggers : Conseils

- Utiliser les triggers lorsque les vérifications à réaliser ne portent pas que sur une seule ligne
- Dès que cela est possible:
 - Préférer la définition de contraintes Check à l'utilisation de triggers.
 - Préférer l'utilisation des options de suppression et maj en cascade à la définition de triggers
- Pour être utiles et efficaces, les triggers doivent être utilisés avec mesure!
 - Il faut notamment limiter leur taille