Bases de données : « Intégrité, Confidentialité & Administration »

David Célestin FAYE

UFR SAT/UGB
(Version provisoire)

28 octobre 2021

Plan 2

Plan 3

Motivations 4

- Un SGBD est un ensemble coordonnés et complexe de logiciels permettant de gérer une base de données
- La compléxité est liée à l'organisation
- Administration de qui ? De quoi ?
  - des utilisateurs de la base
  - des données
  - des bases de données
- l'administration de donnée consiste à :
  - créer la base de données
  - définir les objets de la base
  - veiller à la bonne utilisation des données.

Mettre en place et à gérer de manière satisfaisante un environnement dans lequel les données sont utilisées de manière efficace et sûre

Plan 5

Fonctions d'un DBA (*Database Administrator*) 6

- Définition de toute l'information nécessaire d'un point de vue gestion
- Développement et administration des règles, des procédures, des pratiques et des plans pour la définition, l'organisation, la protection et l'utilisation la plus efficace possible des données
- Cela inclut toutes les données d'une organisation, informatique ou non
- Fonction de gestion et de coordination
- Responsable de la partie physique de la conception :
  - la sélection du logiciel et du serveur
  - l'installation/mise à jour du SGBD
  - la sécurité des données, intimité (privacy), intégrité
  - la sauvegarde et la restauration
  - l'amélioration des performances (requêtes, transactions, indexes)

Fonctions d'un DBA 7

- administration et gestion du contenu des données
  ☞**quelles** données doit-on gérer ?
- administration et gestion de la structure des données
  ☞**comment** doit-on les gérer ?
- administration et gestion des aspects physique de la base
  ☞**où** doit-on les gérer ?

Fonctions d'un DBA 8

Administration et gestion du contenu des données

- quelles données seront dans la base ?
- quels utilisateurs et quel droits d'accès sur la base ?
- quels contrôles de cohérence à maintenir ?
- quand les données peuvent elles être supprimées ?

Administration et gestion de la structure des données

- combien de bases/tables/... sont nécessaires ?
- quelle est la structure des tables ?
  - normalisation (3NF, BCNF, 4NF, 5NF ?)
  - analyse fonctionnelle (quelles fonctions ?)
  - dé-normalisation (performances)
- comment implanter la sécurité ?
- comment sont effectués les contrôles de cohérence ?
- comment les accès sont-ils optimisés ?
  - transactions, cluster, indexes, requêtes
- comment et quand réorganiser la base de données ?

Administration et gestion des aspects physiques de la base

- combien de serveurs logiciels/matériels ?
  - dimensionnement des serveurs
  - performances du système d'exploitation support
- quand et comment procéder pour les sauvegardes ?
  - bases de données
  - fichiers de recouvrement , fichiers de logs
- procédure de démarrage/arrêt
- procédures pour la récupération en cas de problèmes
  - export/import, réplication, rollforward/rollback

Rôle d'un DBA

- l'administrateur peut avoir un double rôle :
  - rôle organisationnel
  - rôle technique
- ces deux rôles peuvent être assurés par une ou plusieurs personnes.

- Installation du SGBD et des outils associés
- Création de la base de données et assurer son évolution
- Gestion des privilèges d'accès
- Amélioration des performances
- Sécurité, intégrité et cohérence des données
- Surveillance et optimisation
- Réduction de l'espace de stockage
- Gestion du partage des données
- Echange de données entre la base et le monde extérieur
- Sauvegardes, restauration
- Assistance aux utilisateurs
- Gestion du changement

Relation avec les utilisateurs

- besoins quant aux données
- applications prioritaires
- possession des données
- besoin en termes d'archivage
- documentation

Relation avec les développeurs d'applications

- contrôle de sécurité
- règles pour l'intégrité des données
- informations sur la base
- plans de tests
- formation

Relations avec l'équipe maintenance

- besoins en terme de disponibilité
- priorités sur les interventions
- procédures de sécurité
- surveillance de la performance

Relations avec développeurs de SGBD

- veille technologique
- besoins matériels
- documentation
- support en termes de services/maintenance
- utilitaires

Interactions 13

Relations avec les vendeurs de matériel

- capacité d'un serveur/disque/...
- capacité d'extension
- incompatibilités
- capacités en termes de services/maintenance
- formation

Relations avec l'équipe de direction

- objectifs de l'entreprise
- contraintes (logiciel,matériel, temps) pour le développement, mise à jour,...
- budget
- évolution
- changements en termes d'organisation

Utilisateur final 14

Utilisateur final

- Celui qui utilise l'information
- sait ce qu'il veut,
- ne sait pas forcément ce qu'il faut faire pour l'obtenir,
- qui ne sait rien du fonctionnement interne
- n'a accès qu'aux données qui lui sont utiles

l'information doit donc

- être correcte
- être accessible lorsque l'utilisateur en a besoin
- être en phase avec les besoins des utilisateurs
- permettre d'être productif

Les métiers 15

La séparation entre les métiers : Administrateur, Concepteur, Analyste, Programmeur, Technicien dépend

- de la taille de l'entreprise,
- de la complexité de l'application
- du SGBD choisi
- des utilisateurs finaux
- de l'étape du développement

Plan 16

Sécurisation

Sécurité

- protection de la base de données contre les utilisateurs *non autorisés*
- vérifie que ce que les utilisateurs veulent faire est *autorisé*

Intégrité

- protection de la base de données contre les utilisateurs *autorisés*
- vérifie que ce que les utilisateurs veulent faire est *correct*

Protection des données contre les pertes accidentelles ou intentionnelles, les destructions, ou les mauvaises utilisations

Exemples

Pertes accidentelles

- erreur humaine, bug logiciel ou matériel,
- procédures pour les autorisations utilisateurs, pour les installations logicielles, pour les maintenances matérielles

Fraudes

- utilisateur, réseau, système, matériel
- contrôle des accès physiques, pare-feu, mise à jour sécurité...

Initimité/confidentialité

- données personnelles, données de l'entreprise

Intégrité des données

- données invalides/corrompues
- procédures de sauvegarde/restauration

Niveau de sécurisation

Au niveau du SGBD

- authentification/autorisation d'accès pour les utilisateurs de la base
- attaques sur le SGBD : failles connues, failles dans les applis associées (programmes setuid root installés par le SGBD,...)
- mauvaises configurations (mot de passe par défaut,...)

Au niveau de l'OS

- authentification/autorisation d'accès pour les utilisateurs du système
- fichiers de la base non sécurisés (lecture par tous)

Au niveau du réseau

- cryptage pour éviter interception des mots de passe, lecture des messages, lecture des fichiers de config,

Niveau de sécurisation

Au niveau applicatif

- utilisateurs, rôles, droits, actions possibles,...
- vues
  restriction et vérification dynamique pour la consultation et la modification
- procédures stockées
  contrôle des opérations et des requêtes effectuées sur la base
- injection SQL sur les applications Web

Au niveau physique

- sécurisation du stockage
- accès aux serveurs, feu, sauvegarde...

Au niveau humain

- mot de passe,...

Possibilités pour assurer la sécurité

- autorisations (identification, restriction) et contrôles
- vues
- procédures de cryptage pour rendre les données illisibles

Avec le client/serveur et le couplage Web/SGBD il faut :

- identification : qui
- authentification : l'identité est vérifiée
- autorisation : permission pour effectuer cette action
- intégrité : les données envoyées sont celles reçues
- confidentialité : personne n'a lu les données envoyées
- audit : pour vérifier a posteriori

Différentes autorisations sur la base

- lecture
- inser tion
- mise à jour
- suppression

Différentes autorisations sur la structure de la base

- création/suppression d'index
- ressources : création/suppression de nouvelles relations
- altération des schémas : ajout/suppression d'attributs

Privilège

droit d'effectuer une action précise sur un objet donné

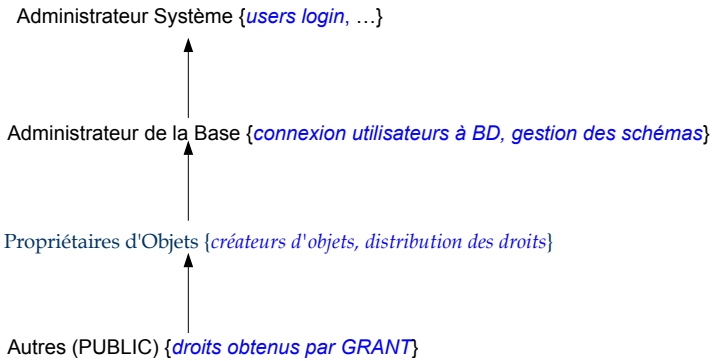
Actions possibles

- créer
- accéder
- modifier
- supprimer...

Notion de rôle

- possibilité de gérer un ensemble de privilèges acquis
- défini en fonction des besoins d'un certain groupe d'utilisateurs

Hiérarchie des privilèges



- attribution de privilèges sur des objets

```
GRANT droits ON objet
TO usagers
[WITH GRANT OPTION ] ;
```

- droits :

```
SELECT|INSERT|UPDATE|DELETE|ALTER| ALL [PRIVILEGES]
```

- objet

```
TABLE <relation> | VIEW <vue>
```

- usagers

```
PUBLIC| <username>
```

- Un utilisateur ayant reçu un privilège avec l'option facultative `[WITH GRANT OPTION]` peut la transmettre à son tour.

- suppression de privilèges

```
REVOKE liste-systeme-privileges
FROM usagers
```

- suppression de privilèges sur des objets

```
REVOKE liste-privileges
ON objet
FROM usagers
```

- Un utilisateur ayant accordé un privilège peut le reprendre avec l'ordre

```
REVOKE droit ON table/vue FROM utilisateur
```


Règles d'octroi et de suppression des droits

- On ne peut transmettre que les droits que l'on possède ou qui nous ont été transmis avec "grant option"
- On ne peut supprimer que les droits que l'on a transmis
- La révocation des droits est récursive
- Si un utilisateur U1 a reçu le droit D de la part de plusieurs utilisateurs (U2, U3, ...), il ne perd ce droit que si tous les utilisateurs lui retirent

règles d'autorisation

Contrôles inclus dans le système qui permettent de restreindre l'accès aux données et les actions des utilisateurs sur ces données

Utilisation de

- tables d'autorisation pour les rôles
- tables d'autorisation pour les objets

pour indiquer quel sujet est autorisé à effectuer une quelle action sur quel un objet de la base

	Client	Commandes
Read	Y	Y
Insert	Y	Y
Modify	Y	N
delete	N	N

	Service Ventes	Service Commandes	Service Compta
Read	Y	Y	Y
Insert	Y	Y	N
Modify	Y	N	Y
delete	N	N	N

GERER UN UTILISATEUR

```
CREATE ROLE nom_utilisateur
[ [WITH] option [...]] ;
```

(dans les anciennes version : create user)

Voici quelques options possibles :

```
login
password 'mot_de_passe'
superuser | nosuperuser
createdb | nocreatedb
createrole | nocreaterole
```

(CREATE USER est équivalent à CREATE ROLE sauf que CREATE USER utilise l'option LOGIN par défaut alors que CREATE ROLE ne le fait pas).

Exemple

```
postgres=# create role uti3 login password 'uti3';
CREATE ROLE

postgres=# create role  uti4
postgres-# password 'uti4'
postgres-# nocreatedb
postgres-# nocreateuser;
CREATE ROLE
```

Pour connaître les rôles existants :


```
postgres=# select rolname from pg_roles ;
 rolname
----------
 dude
 postgres
 alice
 babou
 bob
 uti3
 uti4
(7 rows)
```

Il est également possible d'utiliser : \du


```
postgres=# \du
                             List of roles
 Role name |                   Attributes                   | Member of
-----------+------------------------------------------------+-----------
 alice     | Create DB                                      | {bob}
 babou     | Superuser, Cannot login                        | {}
 bob       | Superuser, Create DB                           | {}
 dude      |                                                | {}
 postgres  | Superuser, Create role, Create DB, Replication | {}
 uti3      |                                                | {}
 uti4      | Cannot login                                   | {}
```

SUPPRIMER UN UTILISATEUR

```
 DROP ROLE nom_utilisateur ;
```

Exemple

```
postgres=# drop role uti3;
DROP ROLE
```

MODIFIER UN UTILISATEUR

```
ALTER ROLE nom option ;
```

Exemple

```
postgres=# alter user uti4 createdb;
ALTER ROLE
```


GERER UN GROUPE

- Les groupes sont une manière logique de réunir des utilisateurs pour faciliter la gestion des privilèges : les privilèges peuvent être accordés ou révoqués à un groupe entier.
- Attention, un groupe est en fait un rôle (sans mot de passe) pouvant contenir d'autres rôles (utilisateurs ou groupes), on pourrait donc utiliser la commande create role.

CREER UN GROUPE (un rôle)

```
CREATE ROLE nom_groupe ;
```

(dans les anciennes version : create group)
Exemple

```
postgres=# create role gr2;
CREATE ROLE
```


MODIFIER L'ATTRIBUTION D'UN GROUPE (un rôle)

```
GRANT nom_role TO nom_groupe ou nom_uti ;
```

```
REVOKE nom_role FROM nom_groupe ou nom_uti;
```

Exemple

```
postgres=# grant gr2 to uti4 ;
GRANT ROLE
```

```
postgres=# revoke gr2 from uti4 ;
REVOKE ROLE
```


GERER LES DROITS

ATTRIBUER DES DROITS

- Ce sont les commandes GRANT et REVOKE qui permettent d'assigner des droits aux utilisateurs.
- Par défaut, le propriétaire d'un objet a tous les droits sur celui-ci et les autres n'ont aucun droit.
- Les droits portent sur : les bases de données, les schémas, les tables, les vues, et les fonctions.

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | RULE
       | REFERENCES | TRIGGER } [,...] | ALL [ PRIVILEGES ] }
ON TABLE nom_objet [, ...]
TO { nom_role | PUBLIC } [, ...] ;
```

```
GRANT { { CREATE | TEMPORARY | TEMP } [,...]
      ON DATABASE nom_base [, ...]
```

```
TO { nom_role | PUBLIC } [, ...] ;
```

- Il existe également ON TABLESPACE ou ON SCHEMA par exemple.
- En ajoutant, WITH GRANT OPTIONS ou WITH ADMIN OPTION à la fin de la commande, cela permet d'attribuer le droit de donner des droits.
- Les différents droits sont listés ci-dessous. Le droit ALL PRIVILEGES attribue tous les droits possibles sur un objet.

Les privilèges octroyés dépendent du type d'objet

- tables : SELECT | INSERT | UPDATE | DELETE | REFERENCES | TRIGGER
- séquences : USAGE | SELECT | UPDATE
- base de données : CREATE | CONNECT | TEMPORARY

- fonctions : EXECUTE
- langage : USAGE
- schémas : CREATE | USAGE
- espaces de tables : CREATE

Exemple
L'administrateur, connecté sous la base postgres, créé une base de donnée mabd et un utilisateur toto :

```
postgres=# create database mabd;
CREATE DATABASE
postgres=# create role  toto login password 'toto';
CREATE ROLE
postgres=#
```


Attention, si un utilisateur n'a pas de mot de passe, il ne pourra pas se connecter.
L'administrateur créé une table tab2 dans mabd :

```
mabd=# create table tab1( id integer primary key,
mabd(# nom varchar(10));
CREATE TABLE
mabd=#
```

L'utilisateur toto se connecte et tente une insertion :

```
Server [localhost]:
Database [postgres]:
Port [5432]:
Username [postgres]: toto
Password for user toto:
psql (9.4.4)
```



```
Type "help" for help.

postgres=> \c mabd
You are now connected to database "mabd" as user "toto".
mabd=>
mabd=> insert into  tab1 values (7,'fanta');
ERROR:  permission denied for relation tab1
mabd=>
```

Il faut donc lui ajouter des droits, en tant qu'administrateur :

```
mabd=# grant insert, select on tab1 to toto;
GRANT
mabd=#
```

L'utilisateur toto peut alors insérer et consulter la table tab1 :

```
Server [localhost]:
Database [postgres]:
Port [5432]:
Username [postgres]: toto
Password for user toto:
psql (9.4.4)
Type "help" for help.

postgres=> \c mabd
You are now connected to database "mabd" as user "toto".
mabd=>

mabd=> insert into  tab1 values (7,'fanta');
INSERT 0 1
mabd=> select * from tab1;
 id |  nom
```



```
----+-------
  7 | fanta
(1 row)

mabd=>
```

Les commandes de l'interpréteur \z ou \dp affichent les droits d'un objet.

```
mabd=# \z
                              Access privileges
 Schema | Name | Type  | Access privileges | Column access privil
--------+------+-------+-------------------+--------------------
 public | tab1 | table | toto=arwdDxt/toto |
(1 row)

mabd=#
```


r	SELECT ("read")
w	UPDATE ("write")
a	INSERT ("append")
d	DELETE
R	RULE
x	REFERENCES
t	TRIGGER
arwdRxt	ALL PRIVILEGES

SUPPRIMER DES DROITS

La commande REVOKE permet d'ôter des droits.

```
REVOKE { { SELECT | INSERT | UPDATE | DELETE | RULE
      | REFERENCES | TRIGGER } [,...] | ALL [ PRIVILEGES ] }
ON [ TABLE ] objet [, ...]
FROM { nom_uti | GROUP nom_groupe | PUBLIC } [, ...] ;
```

L'utilisateur administrateur de la base retire des droits à toto :


```
mabd=# revoke insert on tab1 from toto;
REVOKE
```

Vérification avec l'utilisateur toto

```
mabd=> insert into  tab1 values (8,'Samba');
ERROR:  permission denied for relation tab1
mabd=>
```

- Elles supportent le niveau externe des SGBD selon l'architecture ANSI/SPARC.
- le niveau externe offre une perception de la base plus proche des besoins des utilisateurs, en termes de structures et formats de données.
- Elles assurent donc une indépendance logique des applications par rapport à la base (aux tables)
- Le programme restera invariant aux modifications de schéma s'il accède à la base via une vue qui l'isole de celle-ci.
- Lors de modifications du schéma de la base, l'administrateur changera seulement la définition des vues.

Objectif

- assurer une indépendance logique des applications par rapport à la base (aux tables)

Moyen

- les vues : relations virtuelles dont la définition (schéma) est une requête
- Vues et sous-schémas permettant de restreindre les vues de l'utilisateur sur la base
- elles sont interrogées et mises à jour (sous certaines conditions) comme des relations normales

Problème

Une BD peut contenir des *centaines de tables* avec des *milliers d'attributs* :

- difficile de formuler des requêtes.
- Les requêtes sont complexes.
- Une modification du schéma nécessite la modification de beaucoup de programmes.

Solution

Adapter le schéma et les données à des applications spécifiques $\longrightarrow$ **vues**

Définition

Une vue est une table virtuelle résultat d'une requête. Elle ne stocke donc aucune donnée dans la base, mais qui utilise les informations extraites de la requête.

- ☞ rôle d'une vue
  - réduire la complexité syntaxique des requêtes
  - définir les schémas externes.
  - définir des contraintes d'intégrité.
  - définir un niveau additionnel de sécurité en restreignant l'accès à un sous ensemble de lignes et/ ou de colonnes.(accorder des privilèges d'accès à la vue plutôt qu'à la table)
- ☞ Vue : comparable à une table, et on peut l'interroger par SQL.
- ☞ différence : une vue est le résultat d'une requête, avec la caractéristique essentielle que ce résultat est réévalué à chaque fois que l'on accède à la vue. Une vue est donc dynamique

- On utilise les termes **table** et **vue** pour distinguer des relations « matérialisées »et les relations « virtuelles »
- Les vues permettent d'implémenter l'indépendance logique en permettant de créer des relations virtuelles
- Vue = Question stockée
  Le SGBD stocke la définition et non le résultat
  La définition de la vue est enregistrée dans la base de données, mais les lignes correspondant à la vue ne le sont pas
- Le contenu d'une vue est recalculé à chaque utilisation de la vue par SQL
- Exemple :
  - La vue des étudiants thiessois
  - La vue des enseignants avec leurs matières enseignées

- Création d'une vue

  CREATE VIEW <nomVue>[(nom_col1,...)]
  AS <requêteSQL>
  [WITH CHECK OPTION]
- Le nom de la vue doit être un nom d'objet unique dans la BD
- Par défaut les noms des colonnes de la vue sont les mêmes que les noms des colonnes du résultat SELECT
- Si certaines colonnes résultat du SELECT sont des expressions, il faut renommer ces colonnes dans le SELECT ou spécifier les noms de colonne de la vue.
- RequêteSQL instruction SELECT(portant sur des tables/vues) ne comprenant pas de clause ORDER BY, UNION ou INTO
- [WITH CHECK OPTION] permet de ne pas autoriser d'insertion ou de modification de données ne répondant pas aux critères de la requête
- Une vue permet de donner un nom à une requête

- Vue SQL ≠Vue ANSI-SPARC = Base virtuelle
- Vue SQL : Une expression de sélection mémorisée
- On utilise les termes *table* et *vue* pour distinguer des relations « matérialisées »et les relations « virtuelles »
- création de la vue correspondant aux clients de dakar

```
CREATE VIEW ClientsDeDakar
AS SELECT numClient, nomClient
   FROM Clients
   WHERE Ville= 'Dakar' ;
```


- **Interroger une vue**
  interrogation de la vue correspondant aux clients qui habitent Dakar
  `SELECT * FROM ClientsDeDakar;`
- **Supprimer une vue**
  `DROP VIEW nomDeVue ;`
  suppression de la vue correspondant aux clients qui habitent Dakar
  DROP VIEW ClientsDeDakar ;
- **Renommer une vue** `RENAME ancien_nom TO nouveau_nom;`

- pour connaître les clients qui ont fait le plus grand nombre de commandes

```
CREATE VIEW nbCommandesParClients
AS SELECT codeClient, count(*) AS nbCommandes
FROM Commandes C
GROUP BY codeClient;

SELECT codeClient FROM nbCommandesParClients
WHERE nbCommandes =
            (SELECT max( nbCommandes)
            (FROM nbreCommandesParClients) ;
```


- une vue est une relation virtuelle et toutes les modifications de cette relation doivent être transmises aux relations (tables) utilisées dans sa définition.
- La plupart du temps *il n'est pas possible de mettre à jour une vue* (insérer un n-uplet, ...).
- opérations sur les vues INSERT - UPDATE - DELETE
- Restrictions : Ces instructions ne s'appliquent pas aux vues qui contiennent :
  - une jointure
  - un opérateur ensembliste : UNION, INTERSECT, MINUS
  - une clause GROUP BY, ORDER BY ou
  - la clause DISTINCT,
- Restrictions : De plus, Une vue n'est pas modifiable quand elle ne contient pas tous les attributs définis comme NON NULL dans la table interrogée

Règle

Une vue est *modifiable* quand elle est définie comme une sélection/projection sur une relation R (qui peut aussi être une vue modifiable) sans utilisation de SELECT DISTINCT.

Conditions de mise à jour pour les vues(1)

- Pour UPDATE, DELETE, INSERT la vue ne doit pas contenir :
  - Un opérateur ensembliste (UNION, MINUS, INTERSECT)
  - Un opérateur DISTINCT
  - Une fonction d'agrégation comme attribut
  - Une clause GROUP BY
  - Une jointure (la vue doit être construite sur une seule table)

Conditions de mise à jour pour les vues(2)

- Pour UPDATE, DELETE, INSERT :
  - Les colonnes résultats de l'ordre SELECT doivent être des colonnes réelles d'une table de la base et non des expressions
  - Si la vue est construite à partir d'une autre vue, cette dernière doit elle-même vérifier les conditions ci-dessus

NB : Il peut y avoir des cas particuliers suivant les SGBD.

- création de la vue pour la personne qui définit les commandes

```
CREATE VIEW defCommandes
AS SELECT numCommande, dateCommande
FROM Commandes
WHERE numCommande IS NULL AND codeClient IS NULL ;
```

- définir une nouvelle commande

```
INSERT INTO  defCommandes
VALUES('C09', #10/03/2009#) ;
```

- supprimer une commande non attribuée

```
DELETE FROM defCommandes
WHERE numCommande = 'C18' ;
```

- modifier une commande non attribuée

```
UPDATE defCommandes
SET dateCommande= dateCommande+ 1 WHERE
numCommande = 'C18' ;
```

- connaître les commandes non attribuées

```
SELECT * FROM defCommandes ;
```


- création de la vue pour la personne qui affecte une commande à un client

```
CREATE VIEW attribCommande
AS SELECT numCommande, codeClient,dateCommande
FROM Commandes ;
```

- attribuer une nouvelle commande à un client

```
UPDATE attribCommande
SET codeClient = 1,dateCommande=#17/04/2009#
WHERE numCommande = 'C15'
AND codeClient IS NULL AND dateCommande IS NULL;
```


- affecter un nouveau client à une commande

```
UPDATE attribCommande
SET codeClient = 202
WHERE numCommande = 'C15' ;
```

- permuter l'attribution des clients à 2 commandes

```
UPDATE attribCommandes A1
SET codeClient =
    (SELECT codeClient
    FROM attribCommandes A2
    WHERE A1.numCommande = 'C10' AND A2.numCommande = 'C20')
    OR (A1.numCommande = 'C20' AND A2.numCommande = 'C10'))
WHERE numCommande = 'C10' OR numCommande = 'C20' ;
```

Les vues : contrôle de mise à jour 61

- création d'une vue de vérification : contrôle de l'insertion ou de la modification de ligne. **WITH CHECK OPTION** protège contre les « disparitions de n-uplets »causées par des mise-à-jour :

```
CREATE VIEW nom de vue
AS requête
WITH CHECK OPTION;
```

- vérification des contraintes de domaine (interdiction des valeurs inconnues)

Vues et tables 62

- Similitudes :
  - Interrogation SQL
  - UPDATE, INSERT et DELETE sur vues modifiables
  - Autorisations d'accès
- Différences
  - On ne peut pas créer des index
  - On ne peut pas définir des contraintes (clés)
  - Une vue est recalculée à chaque fois qu'on l'interroge → vues matérialisées

Sécurité : les vues 63

Avantages

- flexibles et efficaces
- Effet macro : remplacer une requête compliquée par des requêtes plus simples
- définition de critères très proches de ce que les applications ont besoin
- Confidentialité : définition de politiques de sécurité qui prennent en compte les données et le contexte
- Contraintes d'intégrité(CHECK OPTION)
- Augmenter l'indépendance logique (Les applications utilisant les tables de la base ne doivent pas être modifiées si on change le schéma de la base)
- mise à jour possible (sous conditions)

Sécurité : les vues 64

Désavantages

- vérification des conditions d'accès peut devenir lourde
- complétude des vues par rapport à la politique de sécurité voulue
- superposition de vues (incohérences possibles)
- à compléter avec des procédures stockées

- Ne jamais exposer un serveur DB sur Internet
- Ne jamais partager un serveur BD
- Configuration de l'OS
  - permet de limiter l'exploitation des failles
  - appliquer les procédures classiques
    - non installation des composants inutiles
    - limitation des services réseaux
    - application régulière des correctifs de sécurité
- Configuration installation SGBD
  - changer les mots de passe par défaut
  - supprimer les comptes par défaut
  - ne pas installer les exemples, les applications annexes,...

intégrité sémantique d'une BD

- concerne la qualité de l'information stockée dans la BD,
- consiste à s'assurer les informations stockées sont cohérentes par rapport à la signification qu'elles ont, par le respect de règles de cohérence

différents niveaux de règles de cohérence :

Domaine d'application	Règle de gestion (business rule)
Modèle conceptuel (Merise, UML, ...)	Règle d'intégrité
Base de données	Contraintes d'intégrité
Mécanisme de support	Procédures stockées , index, trigger, ...

contrainte d'intégrité

assertion qui doit être vérifiée par des données à des instants déterminés.

Exemple : la note de l'étudiant est comprise entre 0 et 20

base de données (sémantiquement) INTEGRE

Une base de données est sémantiquement intègre si l'ensemble des contraintes d'intégrité (implicites et explicites) est respecté par toutes les données de la base

intégrité d'une base de données (sémantiquement) INTEGRE

concordance des données avec le monde réel que la base de données modélise

- Mise à jour invalide
  Une mise à jour invalide sur une base de données est une opération qui ne respecte pas une contrainte d'intégrité définie dans la base de données.
- Origines des Mises à jour invalides
  - entrée de données erronées,
  - opérations concurrentes,
  - défaillance d'un logiciel ou d'un matériel,
  - opérations illicites (non-autorisées),

But : maintenir la cohérence/l'intégrité de la BD :

- Vérifier/valider automatiquement (en dehors de l'application) les données lors des mises-à-jour (insertion, modification, effacement)
- Déclencher automatiquement des mises-à-jour entre tables pour maintenir la cohérence globale.

Maintenance de la consistance
La maintenance de la consistance d'une base de données consiste à s'assurer que chaque opération sur la base de données respecte l'ensemble des contraintes d'intégrité définie dans celle-ci.

La maintenance de la consistance d'une base de données comprend :

- Contrôle des droits d'accés,
- Contrôle des accès concurrents,
- Contrôle de l'intégrité sémantique,
- Procédures de reprise

SQL : Analyse des contraintes d'intégrité 71

- Les contraintes sont-elles cohérentes entre elles ?
- Quand vérifier une contrainte ?
  - selon le type de mise-à-jour(surtout en insertion et modification) et
  - à chaque opération : contraintes d'attribut, domaine, n- uplet, etc.
  - à la validation de la transaction : contraintes référentielles, assertions

Type de contraintes d'intégrité 72 I

intra-table/inter-table :

- la contrainte porte sur une ou plusieurs tables

mono-attribut/multi-attribut :

- la contrainte concerne un ou plusieurs attributs

individuelle/ensembliste :

- la contrainte concerne un tuple ou une table

statique/dynamique : la contrainte s'applique à un état ou à un changement d'état :

- *statique* : condition vérifiée pour tout état.
  Exemple :domaine des notes (toute note doit être comprise entre 0 et 20)
- *dynamique* : condition entre l'état avant et l'état après le changement.
  Exemple : Mise à jour des notes (Aucune note ne peut être diminué)

différées/immédiates :

- *Contrainte Immédiate :* Une Contrainte Immédiate est une condition évaluée pendant l'exécution de la transaction.
- *Contrainte Différée* : Une Contrainte Différée est une condition évaluée à la fin de la transaction de la transaction.

Intrinsèque / Générique

- *Contrainte Intrinséque* : Une Contrainte Intrinséque est contrainte structurelle.
  Exemple : contraintes de clé primaire, unicité, non-nullité
- *Contrainte Générique* : Une Contrainte Générique est une contrainte active
  Exemple : contraintes de domaine, d'intégrité référencielle.

- conformité avec le schéma de la base
  tout attribut ou table cité dans la C.I. appartient au schéma
- conformité avec les données de la base
  toutes les données déjà existantes dans la base doivent respecter la nouvelle C.I.
- conformité avec les autres C.I. détection de contradiction ou redondances, ..

CODD (90) propriétés « CRUDE »de l'intégrité d'une BDR :

- C : "Colonne" :intégrité du typage d'un attribut
- R : "Référence" : intégrité référentielle
- U : "Utilisateur" : intégrité applicative sur les règles de gestion
- D : "Domaine" : intégrité de domaine
- E : "Entité" :intégrité de clé primaire

En SQL :

- Prédicats de contrainte SQL(*check, assertions*)
- Procédures déclenchées SQL(*triggers*)
- Vues filtrantes SQL(*views with check option*)
- Procédures SQL associées au schéma(*stored procedures*)

Hors SQL :

- Modules d'accès
- Sections de code distribuées dans les programmes d'application
- Procédure de validation attachées aux écrans de saisies
- Programme de validation avant ou après chargement ...

1. Contraintes inhérentes au modèle relationnel :
   - intégrité des domaines
   - intégrité de relations (clés identifiantes)
   - intégrité de références (clés étrangères)
     Requêtes SQL, CREATE TABLE et ALTER TABLE ...
2. Contraintes spécifiques :
   - contraintes générales (évaluées pour toute modification de la base de données)
   - contraintes attachées aux tables de base
   - 3 types de requêtes SQL :
     - CREATE ASSERTION ... CHECK
     - CHECK
     - TRIGGER

- Contraintes de domaines : Types, NOT NULL, CHECK, CREATE DOMAIN, DEFAULT
- Contraintes de clé primaire : PRIMARY KEY
- Contraintes d'unicité : UNIQUE
- Contraintes d'intégrité référentielle : FOREIGN KEY
- Désactivation et réactivation de contraintes

Désactivation d'une clé primaire :
ALTER TABLE client DESACTIVATE PRIMARY KEY

- les insertions, suppressions et mises à jour sont suspendues tant que la clé est désactivée
- les contraintes référentielles concernées par la clé primaire désactivées sont aussi désactivées
- aucun autre utilisateur ne peut accéder à la table tant que la clé est désactivée

Réactivation d'une clé primaire :
ALTER TABLE client ACTIVATE PRIMARY KEY

Désactivation/réactivation d'une clé étrangère (fk1 est le nom de la contrainte référentielle concernée)
ALTER TABLE Commande DESACTIVATE/ACTIVATE FOREIGN KEY fk1
Désactivation/réactivation d'une contrainte d'unicité :
ALTER TABLE client DESACTIVATE/ACTIVATE UNIQUE fu1
insertions et mises à jour sont suspendues tant que la clé est désactivée.

Contraintes de domaines : Types SQL

INTEGER, CHAR, DATE, ...

Contraintes de domaines : NOT NULL, CHECK

contrainte NOT NULL

```
CREATE TABLE Clients(
     codeClient INTEGER,
     nomClient CHAR(20) NOT NULL,
     prenomClient CHAR(20),
     ville CHAR(16));
```

contrainte CHECK sur une colonne

```
CREATE TABLE Clients(
     codeClient INTEGER,
     nomClient CHAR(20) NOT NULL,
     prenomClient CHAR(20),
     ville CHAR(16)
     sexe CHAR(1)  CHECK (sexe IN ('M', 'F') );
```


Contraintes de domaines : CREATE DOMAIN DEFAULT

```
CREATE DOMAIN domaineSexe
     CHAR(1)  CHECK (sexe IN ('M', 'F');

CREATE DOMAIN villeLivraison
     CHAR(3)  DEFAULT 'DK' CHECK (VALUE IN ('DK', 'TH','LG');

CREATE TABLE Clients(
     codeClient COUNTER,
     nomClient CHAR(20) NOT NULL,
     prenomClient CHAR(20),
     ville villeLivraison,
     sexe domaineSexe );
```

DEFAULT - Valeur par défaut

```
CREATE TABLE Clients(
     ...,
     statut VARCHAR(20) DEFAULT 'célibataire' NOT NULL
     ..... );
```


Il est possible de donner un nom à une contrainte pour pouvoir la supprimer ultérieurement

```
CREATE DOMAIN villeLivraison
     CHAR(3)  DEFAULT 'DK' CONSTRAINT contVille CHECK (VALUE
IN ('DK', 'TH','LG');
```

Modification et suppression de domaine :

- ALTER DOMAIN
- DROP DOMAIN
- DROP CONSTRAINT

déclaration d'une clé primaire à de la création d'une table :

```
CREATE TABLE Clients(
     codeClient INTEGER NOT NULL,
     nomClient CHAR(20) NOT NULL,
     prenomClient CHAR(20)
     ville CHAR(20),
     PRIMARY KEY(codeClient));
```

ajouter une clé primaire à une table existante :

```
CREATE TABLE Clients(
codeClient INTEGER,
nomClient CHAR(30),
precomClient CHAR(30),
adresse CHAR(50));

ALTER TABLE Clients
ADD CONSTRAINT cle_primaire PRIMARY KEY(codeClient);
```

Clé primaire composée

```
CREATE TABLE ligneCommandes(
     numCommande INTEGER ,
     codeProduit CHAR(20),
     CONSTRAINT pk PRIMARY KEY(numCommande, codeProduit));
```

Contrainte d'unicité : UNIQUE

```
CREATE TABLE Clients(
   codeClient INTEGER,
   nomClient CHAR(20) NOT NULL, UNIQUE,
   prenomClient CHAR(20),
   ville CHAR(16));
```

un attribut déclaré en contrainte d'unicité doit être défini avec l'option NOT NULL

Déclaration d'une clé étrangère à la création d'une table :

```
CREATE TABLE Commandes(
     numCommande INTEGER PRIMARY KEY,
     dateCommande DATE,
     codeClient INTEGER,
     CONSTRAINT relation_commander FOREIGN KEY (codeClient)
     REFERENCES Clients ON DELETE CASCADE ON UPDATE CASCADE);
```

- la table parente doit déjà exister
- l'attribut de la table parente doit avoir été définie en clé primaire
- option `ON DELETE CASCADE` : si un tuple de la table Clients est détruit, tous les tuples de la table Commandes s'y référant sont détruits automatiquement

Options de suppression sur clé étrangère

Considérons la suppression d'un enregistrement de la table MERE :

- RESTRICT
  - l'enregistrement de la table MERE est supprimé SSI aucun autre enregistrement des tables FILLE ne dépend de lui
  - les enregistrements dépendants (des tables FILLES) doivent être supprimées ou modifiées AVANT !
- SET NULL (seulement pour clés étrangères autorisant le NULL) :
  - l'enregistrement de la table MERE est supprimé
  - toutes les clés étrangères de valeur égale à la clé primaire de l'enregistrement sont automatiquement mises à NULL
  - si la clé étrangère contient plus qu'un attribut, au moins un doit permettre le NULL
- CASCADE :
  - le tuple de la table parent est supprimé ainsi que tous les tuples dépendants des tables filles

Options de suppression sur clé étrangère

- Restreindre les suppressions et les cascader sont les deux options les plus communes.
- RESTRICT empêche la suppression d'une ligne référencée.
- NO ACTION impose la levée d'une erreur si des lignes référençant existent lors de la vérification de la contrainte.
- La différence entre RESTRICT et NO ACTION est l'autorisation par NO ACTION du report de la vérification à la fin de la transaction, ce que RESTRICT ne permet pas.

Désactivation et réactivation de contraintes

- Les contraintes existent toujours dans le dictionnaire de données mais ne sont pas actives
- Chargement de données volumineuses extérieures à la base

- Contraintes générales :
  - CHECK ASSERTION, CHECK ...
- Contraintes attachées aux tables :
  - CONSTRAINT
  - DEFERRABLE / NOT DEFERRABLE
- - Triggers :
  - définition " usages

Création d'une contrainte générale :

```
CREATE ASSERTION nomContrainte CHECK (prédicat) ;
```

- prédicat = une propriété des données qui doit être vérifiée à chaque instant
- Si une commande SQL viole le prédicat, cette commande est annulée et un message d'erreur (exception) est généré

Suppression d'une contrainte générale :

```
DROP ASSERTION nomContrainte;
```

ND :

- Les assertions sont des contraintes qui peuvent porter sur plusieurs tables.
- Syntaxe en SQL standard. La majeur partie des SGBD ne supportent pas ASSERTION et la contourne par des triggers


```
Etudiant(numEtudiant,nom,...) ;
Matiere (codeMat,libellé,...,matricule ;
Evaluation(numEtudiant, codeMat,note) ;
Enseignant (matricule,...,salaireBase,salaireActuel,specialite)
;
```

Chaque Enseignant doit posséder un salaire de base positif :

```
CREATE ASSERTION c1 CHECK
      (NOT EXISTS (SELECT * FROM Enseignant
                   WHERE (salaireBase<0))
```

Tout enseignant doit donner au moins un cours

```
CREATE ASSERTION c2 CHECK
      (NOT EXISTS (SELECT * FROM Enseignant
                   WHERE matricule NOT IN
                  (SELECT DISTINCT matricule FROM Matiere)) );
```

```
CREATE TABLE nomTable (listeDefColonne) [listeContrainteTable];
contrainteTable ::=
CONSTRAINT nomContrainte typeContrainteTable modeContrainte
typeContrainteTable ::= PRIMARY KEY
| UNIQUE (listeColonne)
| CHECK (condition)
| FOREIGN KEY (listeColonne)
| REFERENCES nomTable (listeColonne)
modeContrainte ::= [NOT] DEFERRABLE
```


Modes de contrainte :

- DEFERRABLE : évaluation ultérieure (fin d'une transaction)
- NOT DEFERRABLE : évaluation lors de l'instruction de la mise à jour

Contrainte sur le domaine de valeurs :
Exemple : Les notes sont toujours comprises entre 0 et 20 :

```
CREATE TABLE Evaluations (
      numEtudiant CHAR(8),
      codeMat CHAR(5),
      note integer
      CONSTRAINT intervallePoints
      CHECK (note BETWEEN 0 AND 20) NOT DEFERRABLE );
```


Contrainte horizontale :
la valeur d'un attribut est fonction des valeurs apparaissant dans les autres attributs
Ex : Le salaire actuel d'un enseignant doit être supérieur ou égal à son salaire de base :

```
CREATE TABLE Enseignant (
      ...
     CONSTRAINT contrainteSalaire
     CHECK (salaireActuel >= salaireBase)
);
```


Contrainte verticale : la valeur d'un attribut d'une ligne est fonction des valeurs de cet attribut dans les autres lignes
Ex : Le salaire actuel d'un professeur ne peut dépasser le double de la moyenne des salaires de sa spécialité :

```
CREATE TABLE Enseignant (
     ...
     CONSTRAINT salaireRaisonnable
     CHECK (salaireActuel <= 2*(SELECT AVG(salaireActuel)
                                FROM Enseignant E
                                 WHERE E.specialite =
                                Enseignant.specialite)
                                )
     )
)
```

Contrainte faisant référence à une autre table : :
Le salaire actuel d'un professeur donnant le cours numéro M2 doit être supérieur à 1000 euros :

```
CREATE TABLE Enseignant (
   ...
   CONSTRAINT contrSalaire
   CHECK (1000 <= (SELECT salaireActuel
                   FROM Enseignant E
                   WHERE matricule IN
                                (SELECT matricule
                                 FROM Matiere
                                 WHERE codeMatiere = M2)
                   )
          )
)
```


- Certaines contraintes d'intégrité, doivent être exécutées dès l'apparition d'un événement donné
- Ces contraintes étaient traitées de façon procédurale par le programmeur.
- Dans de certains SGBD relationnels, le programmeur peut dans son application déclencher la contrainte dès la survenance d'un événement par la création d'un trigger.
- La norme SQL-2 ne normalise pas l'expression de ces triggers, longtemps chaque SGBD a proposé sa propre syntaxe et ses propres extensions de triggers, maintenant partiellement normalisé dans SQL3.
  On peut donner une syntaxe générale

```
CREATE TRIGGER nomTrigger
EVENEMENT ON nomTable
WHEN (condition à vérifier)
- INSTRUCTIONS -
FOR EACH ROW / STATEMENT
```


Triggers(déclencheur ou démon)

- « Règles actives »généralisant les contraintes d'intégrité
- Procédure stockée dans la base qui est déclenchée automatiquement par des événements spécifiés par le programmeur et ne s'exécutant que lorsqu'une condition est satisfaite.

Définition ECA :

- Événement(E) : une mise-à-jour de la BD qui active le trigger
- Condition(C) : un test ou une requête devant être vérifié lorsque le trigger est activé (une requête est vraie si sa réponse n'est pas vide)
- Action(A) : une procédure exécutée lorsque le trigger est activé et que la condition est vraie


```
BEFORE | AFTER E
WHEN C
BEGIN
  A
END
```

Les déclencheurs permettent :

- d'éviter les **risques d'incohérence** dus à la présence de redondance.
- L'**enregistrement automatique** de certains événements.
- La spécification de **contraintes liées à l'évolution de données.** Exemple : un age ne peut qu'augmenter.

Caractéristiques

- **Un seul déclencheur par événement sur une table**.
- Les déclencheurs permettent de rendre une base de données **dynamique**.
  Une opération peut en déclencher d'autres, qui elles-mêmes peuvent entraîner en cascade d'autres déclencheurs. . .
- **Conséquence**
  Risque de boucle infinie.
- Manipulation simultanée de l'**ancienne** et de la **nouvelle** valeur d'un attribut .
- Un déclencheur peut être exécuté :
  Une fois pour un seul ordre SQL, Ou **à chaque ligne concernée** par cet ordre.
- L'action peut être réalisée **avant** ou **après** l'événement.

Caractéristiques

- SQL n'est pas procédural ⇒ Les SGBD fournissent une extension du langage SQL pour les **actions** à effectuer instructions de sélection, boucles, affectations, . . .
- Langage impératif permettant de créer des véritables procédures (procédures stockées)
  - PL/SQL pour ORACLE
  - PL/pgSQL pour PostgreSQL
  - T-SQL pour SQL Server
  - norme SQL2003 pour DB2 et MySQL5.

- Moment de déclenchement par rapport à la requête activante :
  - avant (before) ou après (after)
  - à la place de (instead of)
  - à la fin de la transaction (deferred)
  - dans une transaction séparée (decoupled)
- Nombre d'exécutions par déclenchement :
  - for each statement : une exécution par requête activante
  - for each row : une exécution par n-uplet modifie


```
CREATE TRIGGER  <nom>
// Evénement
{BEFORE | AFTER | INSTEAD OF}
{INSERT | DELETE | UPDATE [OF <liste d'attributs>]}
ON <table>
[REFERENCING {NEW | OLD } AS <nom>]...
// Condition
([WHEN (<condition SQL>) THEN]
// Action
[FOR EACH {ROW | STATEMENT}]
<Procédure SQL>)
```

SQL : triggers -Syntaxe 105

- `INSERT |UPDATE | DELETE`
  - Précise le ou les événements provoquant l'exécution du trigger
- `BEFORE | AFTER`
  - Spécifie l'exécution de la procédure avant(BEFORE) ou après (AFTER) l'exécution de l'événement déclencheur
  - `BEFORE` : vérification de l'insertion
  - `AFTER` : copie ou archivage des opérations
- `[FOR EACH ROW ]`
  - Indique que le trigger est exécuté pour chaque ligne modifiée/insérée/supprimée
  - Si cette clause est omise, le trigger est exécuté une seule fois pour chaque commande UPDATE, INSERT, DELETE, quelque soit le nombre de lignes modifiées, insérées ou supprimées.
- `[WHEN (condition)]`
  - conditions optionnelles permettant de restreindre le déclenchement du trigger
  - !!! Ne peut contenir de requêtes

SQL : triggers -Contrôle d'intégrite 106

Soit la relation

`Enseignant(numEns,nomEns,prenomEns,grade, adresse)`

On veut vérifier la contrainte de clé à l'insertion d'un nouveau enseignant :

```
CREATE TRIGGER InsertEns
BEFORE INSERT ON Enseignant
REFERENCING NEW AS N
(WHEN EXISTS
   (SELECT * FROM Enseignant WHERE numEns=N.numEns)
THEN FOR EACH ROW
   ABORT);
```

SQL : triggers -Contrôle d'intégrite 107

Soit le schéma relationnel

`Enseignant(numEns,nomEns,prenomEns,grade, adresse)`
`Remuneration(grade,salaire)`

Suppression d'un grade et des employés correspondants (ON DELETE CASCADE) :

```
CREATE TRIGGER EffacerGrade
AFTER DELETE ON Remuneration
REFERENCING OLD AS O
FOR EACH ROW
   (BEGIN
     DELETE FROM Enseignant WHERE grade=O.grade
   END);
```

SQL : triggers -Mise-à-jour automatique 108

Soit le schéma relationnel

`Enseignant(numEns,nomEns,prenomEns,grade, adresse)`

Création automatique de clé :

```
CREATE TRIGGER AffecterCléEns
BEFORE INSERT ON Enseignant
REFERENCING NEW AS N
FOR EACH ROW
  (BEGIN
     N.numEns := SELECT COUNT(*) FROM Enseignant
  END);
```

Soit le schéma relationnel

Remuneration(grade,salaire,hausse)

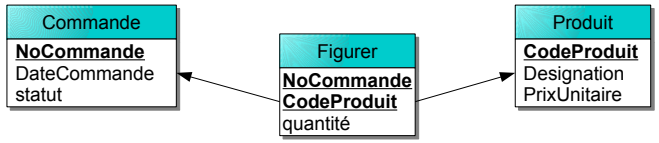
Maintenance des augmentations de salaire :

```
CREATE TRIGGER MajHausse
AFTER UPDATE OF salaire ON Remuneration
REFERENCING OLD AS O,NEW AS N
FOR EACH ROW
  (UPDATE Remuneration
  SET hausse = O.hausse + (N.salaire - O.salaire)
  WHERE grade = N.grade);
```


- l'intérêt des triggers concerne le fait que le contrôle est effectué
  - pour tous les utilisateurs
  - pour toutes les activités
- le contrôle n'a pas à être codé pour chaque requête ou programme
- un programme ne peut pas détourner la règle établie par un trigger

- Disponible dans Oracle et dans SQL-3 (pas dans SQL-2)
- Un trigger activé peut en activer un autre :
  - longues chaînes d'activation ⇒ problème de performances
  - boucles d'activation ⇒ problème de terminaison
- Recommandations :
  - pour l'intégrité, utiliser si possible le mécanisme des contraintes plus facile à optimiser par le système.
  - associer les triggers à des règles de gestion.

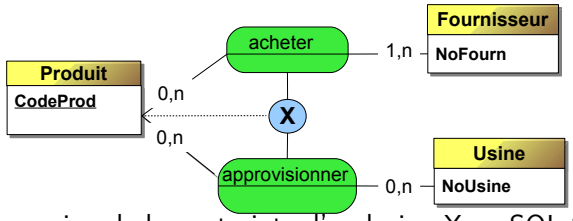
- Certaines contraintes apparaissant dans un MCD sont souvent difficile à traiter de façon déclarative.
- Les triggers permettent aussi de traiter la plupart de ces contraintes surtout celle de stabilité. des propriétés et des associations
- certaines de ces contraintes ne pourront être prise en compte que de façon procédurale dans le développement des applications, en L3G ou L4G.
- Dans la suite,nous donnons quelques exemplesde contraintes graphique et leur traduction en trigger avec l PL/SQL d'oracle.




```
Produits(codeProd,designation,prix)
Commandes(numCom, dateCom,codeClient)
LignesCommande(numCom,codeProd, quantite)
```

On veut un trigger assurant la stabilité de l'attribut dateCommande de la table Commandes
Dans le SGBD Oracle, un tel trigger s'exprimerait ainsi :

```
CREATE TRIGGER dateComStable
AFTER INSERT ON dateCom OF Commandes
ON EACH ROW
BEGIN
   IF :new.dateCom != :old.dateCom
      raise_application_error(-20001,'Date non modifiable');
   END IF;
END;
```

Exepression de la contrainte d'exclusion X en SQL-2, avec l'assertion CX suivante :

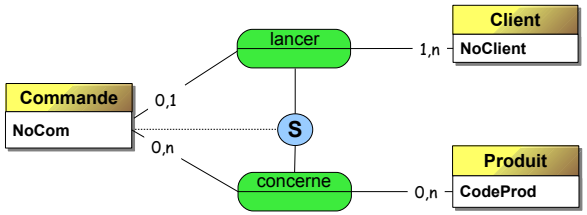
```
CREATE TABLE Produit            CREATE TABLE Fournisseur
CREATE TABLE Acheter            CREATE TABLE Approvisionner
CREATE TABLE Usine
...
CREATE ASSERTION CX
CHECK (NOT EXISTS
       (SELECT * FROM Acheter WHERE codeProd IN
            (SELECT codeProd FROM Approvisionner))
       UNION
       (SELECT * FROM Approvisionner WHERE codeProd IN
            (SELECT codeProd FROM Acheter ));
```


Autre solution avec 2 triggers Oracle suivants :
Trigger 1

```
CREATE TRIGGER ExcluAcheterAppro
BEFORE INSERT ON Acheter
   ON EACH ROW
   DECLARE
       nbProduitsApprovisionner number;
   BEGIN
      SELECT COUNT(*) INTO nbProduitsApprovisionner
      FROM Approvisonner
      WHERE codeProd = :new.codeProd;
      IF nbProduitsApprovisionner > 0 THEN
          raise_application_error (-20002, 'Produit N°',
         :new.codeProd,'est un produit d'approvisionnement' ));
      END IF;
   END;
```


Autre solution avec 2 triggers Oracle suivants :
Trigger 2

```
CREATE TRIGGER ExcluApproAcheter
BEFORE INSERT ON Approvisionner
   ON EACH ROW
   DECLARE
       nbProduitsAcheter number;
   BEGIN
      SELECT COUNT(*) INTO nbProduitsAcheter
      FROM Acheter
      WHERE codeProd = :new.codeProd;
      IF nbProduitsAcheter > 0 THEN
          raise_application_error (-20002, 'Produit N°',
         :new.codeProd,'est un produit à acheter' ));
      END IF;
   END;
```



une Commande portant sur des Produits est obligatoirement destinée à un Client
Pour exprimer la contrainte d'exclusion S en SQL-2, on déclarera l'assertion CS suivante :


```
CREATE TABLE Commande
CREATE TABLE Client
CREATE TABLE Produit
CREATE TABLE Concerner
...
CREATE ASSERTION CS
CHECK (NOT EXISTS
    (SELECT noCom FROM Commande
    WHERE noCom NOT IN
          (SELECT noCom FROM Concerner)
    AND noClient IS NOT NULL
    UNION
    (SELECT noCom FROM Concerner
    WHERE noCom NOT IN
         (SELECT noCom FROM Commande)
          WHERE noClient IS NOT NULL
```

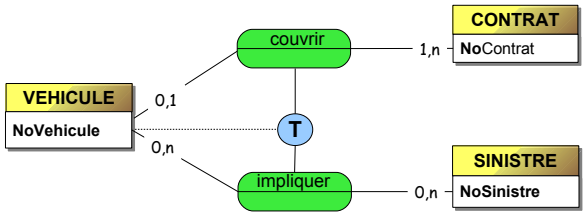

La résolution d'une contrainte de simultanéité par trigger sera partielle au risque de blocage. Ainsi en Oracle, on pourra retenir l'un de ces deux triggers :

Trigger 1

```
CREATE TRIGGER SimultaneitéPasserConcerner
BEFORE INSERT OR UPDATE noClient ON Commande
ON EACH ROW
WHEN new.noClient IS NOT NULL
DECLARE
    nbArticlesLancer number;
BEGIN
   SELECT COUNT(*) INTO nbArticlesLancer
   FROM Lancer
   WHERE noCom = :new.noCom;
   IF nbArticlesLancer = 0
      raise_application_error (-20004, 'La commande n'est pas
      porteuse d'articles' );
   END IF;
END;
```


Trigger 2

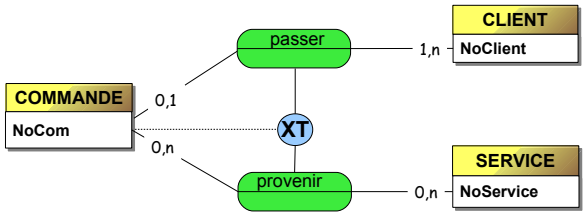
```
CREATE TRIGGER SimultaneitéConcernerPasser
BEFORE INSERT ON Concerner
ON EACH ROW
WHEN new.noClient IS NOT NULL
DECLARE
    nbArticlesConcerner number;
BEGIN
   SELECT COUNT(*) INTO nbArticlesConcerner
   FROM Commande
   WHERE noCclient IS NOT NULL;
   IF nbArticlesConcerner = 0
      raise_application_error (-20005, 'La commande n'est
      pas encore passée par un client' );
   END IF;
END;
```



Tout Véhicule est reliée soit à Contrat par la relation Couvrir, soit à Sinistre par la relation Impliquer, soit les deux
Pour exprimer la contrainte d'exclusion (T) en SQL-2, on déclarera l'assertion CT suivante :


```
CREATE TABLE Véhicule
CREATE TABLE Contrat
CREATE TABLE Sinistre
CREATE TABLE Impliquer
...
CREATE ASSERTION CT
CHECK (NOT EXISTS
        (SELECT n°véhicule FROM Véhicule
         WHERE n°véhicule NOT IN
              (SELECT n°véhicule FROM Véhicule
              WHERE n°contrat IS NOT NULL)
              UNION
              SELECT n°véhicule FROM Impliquer));
```

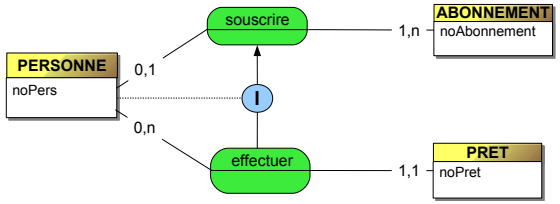
On ne peut traiter cette contrainte de totalité T par triggers, car on doit d'abord insérer un tuple de véhicule avant de l'affecter soit par la relation "couvrir" ou la relation "impliquer" (Primary Key et Foreign Key). Un trigger sur la table Véhicule sur insertion ou mise à jour ne peut ainsi assurer aucun contrôle pertinent. Une telle contrainte sera donc traitée de façon traditionnelle dans l'application.



une Commande est, soit destinée à un Client, soit à un Service, mais pas au deux à la fois,
Pour exrpimer la contrainte de partition XT en SQL-2, on déclarera l'assertion CXT suivante


```
CREATE TABLE Commande
CREATE TABLE Service
CREATE TABLE Client
...
CREATE ASSERTION CXT
CHECK (NOT EXISTS
         SELECT noCom FROM Commande
        WHERE noCom NOT IN
              (SELECT noCom FROM Commande
              WHERE noClient IS NOT NULL)
               UNION
               SELECT noCom FROM Commande))
               WHERE noService IS NOT NULL)
```

On ne peut totalement traiter cette contrainte de partition de participation XT par triggers.
Si la contrainte d'exclusion X peut être traitée par deux triggers sur le modèle déjà présenté, le traitement de la contrainte de totalité T n'est pas possible pour les raisons déjà évoquées précédemment.

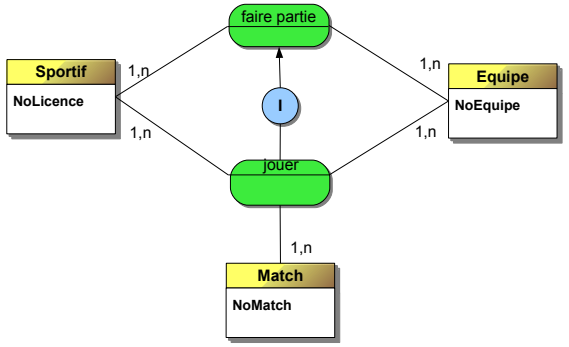


Toute personne qui effectue un prêt doit avoir souscrit un abonnement
Pour exprimer la contrainte d'inclusion I en SQL-2, on déclarera l'assertion CI suivante :

```
CREATE ASSERTION I
CHECK (NOT EXISTS
        (SELECT DISTINCT noPers FROM Personne A
        WHERE NOT EXISTS
              (SELECT DISTINCT noPers FROM Prêt B
              WHERE (A.noPers= B.noPers))
        AND noAbonnement IS NOT NULL));
```


Utiisation d'un trigger dans Oracle traiter cette contrainte d'inclusion I

```
CREATE TRIGGER InclusionEffectuerSouscrire
BEFORE INSERT ON Pret
   ON EACH ROW
      WHEN new.noPers IS NOT NULL
   DECLARE
      nbAbonnement number;
   BEGIN
      SELECT COUNT(*) INTO nbAbonnement FROM Personne
      WHERE noPers = :new.noPers;
      IF nbAbonnement = 0 THEN
         raise_application_error (-20006, 'Un abonnement n'a
         pas été souscrit' ));
      END IF;
   END;
```

L'ensemble des couples (*Sportif* , *Equipe*) qui participent à l'association *jouer* est inclus dans l'ensemble de ceux qui participent à l'association *faire partie*. Autrement dit : tout sportif qui joue un match avec une équipe doit faire partie de cette équipe.

Dans Oracle, on pourrait traiter cette contrainte d'inclusion I par le trigger suivant :

```
CREATE TRIGGER InclusionJouerFairePartie
BEFORE INSERT ON Jouer
   ON EACH ROW
   DECLARE
      faitPartie number;
   BEGIN
      SELECT COUNT(*) INTO faitPartie
      FROM FairePartie
      WHERE noSportif = :new.noSportif
      AND noEquipe = :new.noEquipee;
      IF faitPartie = 0 THEN
         raise_application_error (-20007, 'Ce sportif ne fait
         pas partie de cette équipe' ));
      END IF;
   END;
```

Plan 129

Sauvegarde 130

dump

Copie de la base qui permet d'avoir une image (partielle ou totale) de la base à un instant donné

Périodicité

- journalière
- hebdomadaire
- mensuelle

Type

- cold backup : la base est arrétée pendant la sauvegarde
- hot backup : un fragment est arrété et sauvegardé

Sauvegarde : méthodologie 131

à chaud (on-line)

- minimise la durée de la sauvegarde
- peut provoquer des blocages (en fonction du SGBD)
- nécessite la mise en œuvre de méthodologie spécifique en fonction du SGBD

à froid (off-line)

- minimise la durée de la restauration
- nécessite un arrêt de la base
- effectuée par copie de l'ensemble des éléments de la base
- compatible avec tous les SGBD

Souvent une combinaison de sauvegarde à chaud et à froid

- à froid une fois tous les N jours
- à chaud plus régulièrement

En résumé 132

DBA

- Gestion de la qualité des données
- Lien avec tous les utilisateurs
- qualités
  - sécurité et récupération
  - contrôle des utilisateurs
  - contrôle des accès
  - maintenance (logicielle et matérielle)
  - protection des données

Evolution

- Moins centré sur les aspects traditionnels(backups, optimisation , ...)
- Les SGBDs offrent maintenant des outils permettant une maintenance 'automatique'
- Plus orienté vers gérer les applications, fournir une disponibilité maximale, , extraire une plus-value des informations,etc.)

Open source($\neq$ gratuit)

- MySQL
- PostgreSQL
- Firebird
- MaxDB
- Ingres

Quelques critères

- licence
- plates-formes
- compatibilité SQL standard
- vitesse
- stabilité
- compatibilité ACID
- intégrité
- sécurité
- ...

	MySQL	PostgreSQL
Licence	GPL si le code source de l'application fourni Commerciale sinon	BSD licence (pas besoin d'inclure le source)
Plate-formes	Linux, MacOS, Windows, FreeBSD, OpenBSD,...	Linux, MacOS, Windows, FreeBSD, OpenBSD,...
Conformité SQL	moyenne	très bonne
Vitessse	moyenne/bonne	bonne/moyenne

Remarques- Conformité SQL

- MySQL : sous-ensembles de SQL 92 et SQL 99
- PostgreSQL : sous-ensemble de SQL 2003

Remarques-Vitesse

- MySQL utilise un serveur avec des threads, PostgreSQL utilise les processus
- MySQL supérieur pour SELECT simple, moins pour UPDATE
- PostgreSQL largement optimisable

	MySQL	PostgreSQL
Stabilité	bonne	bonne
Sécurité	bonne	bonne
Stockage	MyISAM, BerkeleyBD, MERGE, InnoDB	une seule possibilité
Intégrité des données	ACID classique	MVCC : Multi Version Concurrency Control
Procédures stockées	oui, supérieure à 5.0	oui
Triggers	oui, rudimentaire, version supérieure à 5.0	oui

Remarques-Stockage

- MySQL : seul InnoDB est transaction-safe
- PostgreSQL : un seul modèle, plus cohérent

Remarques-Procédures stockées

- PostgreSQL : langage pl/pgSQL, plus pl/Perl, pl/TCL, pl/Python

Remarques-Triggers

- pas de contraintes pour MySQL

Il existe 2 moteurs principaux de stockage dans MySQL :

- Mylsam : c'est le moteur par défaut (qui ne gère pas l'intégrité référentielle).
- InnoDB : c'est le moteur le plus évolué qui gère l'intégrité référentielle.

	MySQL	PostgreSQL
Backups	InnoDB seulement pour la cohérence réplication	Write Ahead Logging Recovery après un problème disque
Interface	JDBC, ODBC, C++, Python, Perl, PHP, ADO.Net	ODBC,JDBC,C++,Python, Perl,PHP,Tcl/Tk

Remarques-Interface

- PostgreSQL possède en plus son propre langage

Plan 137

Limites de SQL 138

- Langage non procédural
- Il n'a pas de :
  - Variables
  - Itérations
  - Branchements conditionnels
- Impossible de lier plusieurs requêtes SQL : regrouper un bloc de commandes et le soumettre au SGBD

De SQL à PL/(pg)SQL 139

- Structures de contrôle
  - Branchement conditionnel - Itération
  - Affectations
- Principes de la programmation
  - Fonctions
  - Procédures

Principales caractéristiques 140

- Des requêtes SQL cohabitent avec les structures de contrôle habituelles de la programmation structurée
- Langage (initialement) interprété syntaxiquement proche d'Ada ou Pascal
- Des variables permettent l'échange d'information entre les requêtes SQL et le reste du programme

- Ecriture de procédures stockées et de triggers
- Ecriture de fonctions utilisateurs qui peuvent être utilisées dans les requêtes SQL (en plus des fonctions prédéfinies)
- Contrôle de l'intégrité des données
  - procédures ou fonctions déclenchées sur des événements spécifiques (insertion, suppression, mise-à-jour, etc.)
- Définition de traitements complexes
  - combinaison de requêtes et d'instructions procédurales
  - manipulation de données liées au résultat d'une ou plusieurs requêtes
- Optimisation des traitements fréquents
  - stockage de procédures ou de fonctions (temps d'execution minimisé vs. opération au niveau applicatif)

- Oracle(PL/SQL)
- PostgreSQL(PL/pgSQL)

Ressemble au langage normalisé SQL/PSM : Persistant Stored Modules

- MySQL(SQL/PSM)
- DB2(SQL-PL)
- Sybase et SQLServer(Transact-SQL)
- PostgreSQL(PL/pgPSM)

Mais tous les langages L4G des différents SGBD se ressemblent

Une procédure stockée est similaire à une procédure dans les langages de programmation usuels i.e. :

- 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)

Fonctions stockées sont similaires aux procédures mais :

- peuvent être utilisées dans les requêtes, fonctions, procédures, et vues.
- dans certains SGBD, elles ne peuvent pas modifiées les données ou ont des limitations au niveau ddl/dml.
- en general, elles ne prennent pas des arguments passés par référence. Sous PostgreSQL, il n'y a pas de différence entre procédures et fonctions.

Procédures Stockées 145

Atouts des procédures stockées :

- Sécurité : accès plus sécurisé à une base données et empêche les applications de modifier les règles sémantiques (logique) de la base de données.
- Organisation :interface d'accès utilisable par différentes applications.
- Maintenance : peuvent être modifiées sans modifier les applications.

Questions à prendre en compte

- Portabilité : Est ce que la fonction est utilisée par différentes BDs.
- Modularité : Est ce que la fonction est utilisée dans différentes parties d'une application ou différentes applications.
- Nombre de requêtes : Est ce que la fonction comporte un nombre important (ou réduit) de requêtes.
- Paramètres : Est ce que la fonction nécessite beaucoup de Paramètres et retourne un scalaire ou un ensemble de valeurs.

...

Types de fonctions utilisateur 146

Il y a quatre types de fonctions utilisateurs :

1. SQL : fonctions écrites en langage de requêtes SQL. Une fonction SQL exécute une suite arbitraire de commandes SQL, et retourne le resultat de la dernière requête (SELECT) de la liste.
2. procédurale : fonctions écrites dans un langage procédurale, (PL/Perl, PL/pgSQL, PL/Tcl, ...) : Les langages Procéduraux ne sont pas intégrés au serveur PostgreSQL, Ils sont fournis sous forme de modules chargeables.
3. C : fonctions en langage C.
4. interne : fonctions internes sont écrites en langage C mais liées statiquement au ser- veur PostgreSQL.

Fonctions en SQL 147

fonction SQL

Une fonction SQL exécute une liste arbitraire d'instructions SQL, séparées par des points-virgule et renvoit le résultat de la dernière requête de cette liste.

- instructions autorisées : SELECT, INSERT, UPDATE et DELETE ainsi que d'autres commandes SQL
- commandes interdites : commandes de contrôle de transaction, telles que COMMIT, SAVEPOINT, et certaines commandes utilitaires, comme VACUUM

Fonctions en SQL 148

- Dans le cas d'un résultat simple (pas d'ensemble), la première ligne du résultat de la dernière requête sera renvoyée (« la première ligne » d'un résultat multiligne n'est pas bien définie à moins d'utiliser ORDER BY). Si la dernière requête de la liste ne renvoie aucune ligne, la valeur NULL est renvoyée.
- Une fonction SQL peut renvoyer un ensemble de lignes. Dans ce cas, toutes les lignes de la dernière requête sont renvoyées.
- Sauf si la fonction déclare renvoyer void, la dernière instruction doit être un SELECT ou un INSERT, UPDATE ou un DELETE qui a une clause RETURNING.
- une fonction SQL qui réalise des actions mais n'a pas de valeur utile à renvoyer, renvoie void.

- Deux nouvelles commandes : `Create Function` ou mieux `Create or Replace Function` et `Drop Function`
- Définir la signature de la fonction :

```
CREATE OR REPLACE FUNCTION name ([ftype [ , ...]])
RETURNS rtype AS $$
        definition
$$
LANGUAGE ' langname '
```


- Type de Base type implémenté au niveau interne (langage C).
- Type Composé liste de types associés à des champs (ou type tuple) et défini par les Commandes suivantes
  - implicitement : CREATE TABLE
  - explicitement : CREATE TYPE
- Types Domaines
  type de base dont le domaine des valeurs est restreint par une contrainte.
  - La commande CREATE DOMAIN .

- Pseudo-Types
  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 être utilisés comme type d'un attribut d'une table ou d'un type composé.
- Types Polymorphiques
  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.

Type void
Exemple

```
CREATE FUNCTION supprimeAdmis()
RETURNS void AS '
    DELETE FROM candidat WHERE points>= 290;
' LANGUAGE SQL;

SELECT supprimeAdmis();
  supprimeAmdis
  -----------
  (105 row)
```

Types de Base

La fonction SQL la plus simple possible n'a pas d'argument et retourne un type de base tel que integer :

```
CREATE FUNCTION un()
RETURNS integer AS $$
    SELECT 1 AS resultat;
$$ LANGUAGE SQL;

postgres=# select  un();
 un
----
  1
(1 row)
```

NB le prompt postgres=# indique la base de donnée qui a été selectionné en l'occurrence est la base par défaut

Types de Base

– Autre syntaxe pour les chaînes littérales :

```
CREATE FUNCTION un()
RETURNS integer AS '
    SELECT 1 AS resultat;
' LANGUAGE SQL;

postgres=# select un();
 deux
------
    1
(1 row)
```

- l' alias de colonne avec le nom `resultat` dans le corps de la fonction permet de se référer au résultat de la fonction mais cet alias n'est pas visible hors de la fonction.
- En effet, le résultat est nommé `un` au lieu de `resultat`.

Types de Base

```
CREATE or replace FUNCTION plusDix(in int)
RETURNS integer AS $$
        SELECT $1+10 AS Resultat;
$$ LANGUAGE SQL;

postgres=# Select plusDix(3);
 plusdix
---------
      13
(1 row)
```

- `$i` désigne le ième paramètre
- Les colonnes de tables conformes au type de paramètre sont applicables

Types de Base

- On peut de définir des fonctions SQL acceptant des types de base comme arguments.

```
CREATE FUNCTION add(integer, integer)
RETURNS integer AS $$
    SELECT $1 + $2;
$$ LANGUAGE SQL;

postgres=# SELECT add(7, 9) AS reponse;
 reponse
---------
      16
(1 row)
```

Types de Base

```
create function ajoute(a integer, b integer) returns integer
as 'select a + b;'
language sql;

mydb=# select ajoute(1, 2);
 ajoute
--------
      3
(1 row)

CREATE FUNCTION sum (text, text)
RETURNS text AS $$
  SELECT $1 || ' ' || $2
$$ LANGUAGE SQL;
mydb=# SELECT sum('hello', 'world');
     sum
-------------
 hello world
(1 row)
```


Types de Base

- Fonction, qui pourrait être utilisée pour augmenter une note d'étudiant et renvoyer la nouvelle valeur :

```
CREATE FUNCTION plusNote (integer, numeric)
RETURNS integer AS $$
    UPDATE evaluation
        SET note = note + $2
        WHERE noEtu = $1;
    SELECT note FROM evaluation WHERE noEtu = $1;
$$ LANGUAGE SQL;
```

Un utilisateur pourra exécuter cette fonction pour augmenter la note de l'étudiant 8 de 2,5 points ainsi :

```
SELECT plusNote(8, 2.5);
```


Types Composés (ou tuples)

```
 CREATE TABLE maTable (
         id       int,
         nom      varchar(30)
     );
insert into maTable values(4,'mada');
insert into maTable values(9,'modou');
CREATE or replace FUNCTION fIdTable(maTable)
RETURNS int AS $$
    SELECT $1.id AS id;
$$ LANGUAGE SQL;

postgres=# select fIdTable(maTable) from maTable;
fIdTable
--------
         4
         9
(2 rows)
```

Notez l'utilisation de la syntaxe `$1.id` pour sélectionner un champ dans la valeur de la ligne argument

Types Composés (ou tuples)

```
CREATE FUNCTION fIdTable(maTable)
RETURNS maTable AS $$
    SELECT $1.id, $1.nom;
$$ LANGUAGE SQL;

postgres=# select fIdTable(maTable) from maTable;
fidtable
----------
 (4,mada)
 (9,modou)
(2 rows)

postgres=#
```

Fonctions avec des paramètres en sortie :IN, OUT, INOUT

- Les param. peuvent être marqués comme IN (par défaut), OUT ou INOUT.
- Un param. INOUT sert à la fois de param. en entrée (fait partie de la liste d'arguments en appel) et comme param. de sortie (fait partie du type d'enregistrement résultat).

```
CREATE FUNCTION addEtMult(a int, b int, OUT som int,
                                        OUT prod int)
        AS 'SELECT $1 + $2, $1 * $2'
LANGUAGE SQL;
postgres=# SELECT * FROM addEtMult(5,9);
 som | prod
-----+------
  14 |   45
(1 row)
```

Ici nous avons créé un type composite anonyme pour le résultat de la fonction

Fonctions avec des paramètres en sortie :IN, OUT, INOUT

cet exemple donne le même résultat que le précédent

```
CREATE TYPE sommeMult AS (somme int, mult int);

CREATE FUNCTION addEtMult(int, int)
RETURNS sommeMult AS
        'SELECT $1 + $2, $1 * $2'
LANGUAGE SQL;
```


Fonctions avec des paramètres en sortie :IN, OUT, INOUT

Remarque sur l'exemple précédent

- Les paramètres en sortie ne sont pas inclus dans la liste d'arguments lors de l'appel d'une fonction de ce type en SQL.
- Explication PostgreSQL considère seulement les paramètres en entrée pour définir la signature d'appel de la fonction.
- Cela signifie aussi que seuls les paramètres en entrée sont importants lors de références de la fonction pour des buts comme sa suppression.

Nous pouvons supprimer la fonction avec l'un des deux appels ci-dessous :

```
DROP FUNCTION addEtMult(a int, b int, OUT somme int, OUT produit int);
DROP FUNCTION addEtMult(int, int);
```


Fonctions avec des paramètres en sortie :IN, OUT, INOUT

- Reprenons notre fonction

```
CREATE FUNCTION fIdTable(maTable)
RETURNS maTable AS $$
    SELECT $1.id, $1.nom;
$$ LANGUAGE SQL;
```

- Comme maTable est à la fois en entrée et en sortie nous réécrivons la fonction comme suit

```
CREATE FUNCTION fIdTableBis(INOUT maTable)
AS $$
     SELECT $1.id, $1.nom;
$$ LANGUAGE SQL;
```

Type composé indéfini

```
CREATE FUNCTION EtOu(IN boolean, IN boolean,
                                        OUT boolean, OUT boolean)
AS $$
        SELECT $1 and $2, $1 or $2;
$$ LANGUAGE SQL;

postgres=# SELECT EtOu(true,false);
 etou
-------
 (f,t)
(1 row)
```


Fonctions SQL comme sources de table

- les fonctions SQL peuvent être utilisées dans la clause FROM d'une requête
- utiles pour les fonctions renvoyant des types composites.
- Si la fonction est définie pour renvoyer un type de base, la fonction table produit une table d'une seule colonne.
- Si la fonction est définie pour renvoyer un type composite, la fonction table produit une colonne pour chaque attribut du type composite.

Fonctions SQL comme sources de table

```
CREATE TABLE matiere (id int, nom text, age int);
INSERT INTO matiere VALUES (1, 'yoyo', 18);
INSERT INTO matiere VALUES (1, 'popo', 45);
INSERT INTO matiere VALUES (2, 'mimi', 6);

CREATE FUNCTION recupMatiere(int)
RETURNS matiere AS $$
    SELECT * FROM  matiere WHERE id = $1;
$$ LANGUAGE SQL;

SELECT *, upper(nom) FROM recupMatiere(1) AS t1;
 id | nom  | age | upper
----+------+-----+-------
  1 | yoyo |  18 | YOYO
(1 row)
```

Conclusions,on peut travailler avec les colonnes du résultat de la fonction comme s'il s'agissait des colonnes d'une table normale.

Fonctions SQL renvoyant un ensemble

- Si une fonction SQL est déclarée renvoyer un `SETOF un_type`,
  - la requête finale SELECT de la fonction est complètement exécutée et
  - chaque ligne extraite est renvoyée en tant qu'élément de l'ensemble résultat.
- Cette caractéristique est normalement utilisée lors de l'appel d'une fonction dans une clause FROM.
- Dans ce cas, chaque ligne renvoyée par la fonction devient une ligne de la table vue par la requête.

Fonctions SQL renvoyant un ensemble

- Par exemple, supposons que la table matiere ait le même contenu que précédemment et écrivons :

```
CREATE FUNCTION recupMatiereBis(int)
RETURNS SETOF matiere AS $$
    SELECT * FROM matiere WHERE id = $1;
$$ LANGUAGE SQL;

SELECT * FROM recupMatiereBis(1) AS t1;
 id | nom  | age
----+------+-----
  1 | yoyo |  18
  1 | popo |  45
(2 rows)
```


Fonctions SQL renvoyant un ensemble

```
 CREATE FUNCTION test1 (OUT f1 int, OUT f2 text) AS $$
  VALUES (42, 'hello'), (64, 'world');
$$ language sql;
SELECT * FROM test1();
 f1 |  f2
----+-------
 42 | hello
(1 row)

CREATE FUNCTION test2 (OUT f1 int, OUT f2 text)
RETURNS SETOF RECORD AS $$
  VALUES (42, 'hello'), (64, 'world');
$$ language sql;
mydb=# SELECT * FROM test2();
 f1 |  f2
----+-------
 42 | hello
 64 | world
(2 rows)
```


Fonctions SQL renvoyant un ensemble : Liste cible

```
  mydb=# SELECT test2();
  test2
------------
 (42,hello)
 (64,world)
(2 rows)

mydb=# SELECT (test2()).*;
 f1 |  f2
----+-------
 42 | hello
 64 | world
(2 rows)
```


Fonctions SQL renvoyant un ensemble

```
 CREATE TYPE test3_type AS (f1 int, f2 text);
CREATE FUNCTION test3 ()
RETURNS SETOF test3_type AS $$
  VALUES (42, 'hello'), (64, 'world');
$$ language sql;
mydb=# SELECT * FROM test3();
 f1 |  f2
----+-------
 42 | hello
 64 | world
(2 rows)
```

Fonctions SQL renvoyant une table

```
CREATE FUNCTION test4 ()
RETURNS TABLE (f1 int, f2 text) AS $$
  VALUES (42, 'hello'), (64, 'world');
$$ language sql;
mydb=# SELECT * FROM test4();
 f1 |  f2
----+-------
 42 | hello
 64 | world
(2 rows)
```


Fonctions SQL renvoyant un RECORD non spécifié

```
CREATE FUNCTION test5 ()
RETURNS SETOF RECORD AS $$
  VALUES (42, 'hello'), (64, 'world');
$$ language sql;

mydb=# SELECT * FROM test5() as t(f1 int, f2 text);
 f1 |  f2
----+-------
 42 | hello
 64 | world
(2 rows)
```


Fonctions SQL renvoyant des scalaires

```
 CREATE FUNCTION test6 ()
RETURNS SETOF int AS $$
  VALUES (42), (64);
$$ language sql;
mydb=# SELECT * FROM test6();
 test6
----------
       42
       64
(2 rows)
CREATE FUNCTION test7 ()
RETURNS SETOF int AS $$
  VALUES (42), (64);
$$ language sql;
mydb=# SELECT * FROM test7() AS t(f1);
 f1
----
 42
 64
(2 rows)
```


Fonctions SQL polymorphes

- Les fonctions SQL peuvent être déclarées pour accepter et renvoyer les types « polymorphe »`anyelement` et `anyarray`
- Exemple : fonction polymorphe `creeTableau` qui construit un tableau à partir de deux éléments de type arbitraire :

```
CREATE FUNCTION creeTableau(anyelement, anyelement)
RETURNS anyarray AS $$
    SELECT ARRAY[$1, $2];
$$ LANGUAGE SQL;

SELECT creeTableau(1, 2) AS tableauEntier,
           creeTableau('a'::text, 'b') AS  tableauTexte;

 tableauentier | tableautexte
---------------+--------------
 {1,2}         | {a,b}
(1 row)
```

Fonctions SQL polymorphes

Remarques

- Le transtypage 'a'::text permet de spécifier le type text de l'argument.
- Ceci est nécessaire si l'argument est une chaîne de caractères car, autrement, il serait traité comme un type
- unknown, et un tableau de type unknown n'est pas un type valide.
- Sans le transtypage, vous obtiendrez ce genre d'erreur :

```
SELECT creeTableau(1, 2) AS tableauEntier,
           creeTableau('a', 'b') AS  tableauTexte;

ERROR:  could not determine polymorphic type because
input has type "unknown"
```


Fonctions SQL polymorphes

- Il est permis d'avoir des arguments polymorphes avec un type de renvoi fixe, mais non l'inverse. Par exemple

```
CREATE FUNCTION bigger(anyelement, anyelement)
RETURNS bool AS $$
    SELECT $1 > $2;
$$ LANGUAGE SQL;
SELECT bigger(5, 7);
 bigger
--------
 f
(1 row)

CREATE FUNCTION fonctionInvalide()
RETURNS anyelement AS $$
    SELECT 1;
$$ LANGUAGE SQL;
ERROR:  cannot determine result data type
DETAIL:  A function returning a polymorphic type must have
at least one polymorphic argument.
```


Débite un compte bancaire et retourne la nouvelle valeur du solde (notez la clause RETURNING) :

```
create function debiter(no_compte integer, val numeric)
     returns integer as $$
  update compte
  set solde = solde - val
  where no_compte = debiter.no_compte
  returning solde;
$$ language sql;
```

Pour débiter le compte 34 de 300 000 FCFA

```
> select debiter(34, 300000);
```


Les plus gros emprunteurs (notez le SETOF)

```
create function max_emprunteurs()
returns setof adherent as $$
select a.*
from adherent a
natural join emprunt
group by id_adherent
having count(id_adherent) >=
      (select max(adherent.nb_emprunts)
      from adherent);
$$ language sql;

select max_emprunteurs();
   max_emprunteurs
-------------------------
(4,Diop,Aly,0)
(1 row)
> select * from max_emprunteurs();
 id_adherent | nom_adherent | prenom_adherent | amende_adherent
-------------+--------------+-----------------+-----------------
```

```
      4      |      Diop     |       Aly        |       0
(1 row
> select nom_adherent from max_emprunteurs();
 nom_adherent
--------------
   Diop
(1 row)
> select nom_adherent, titre
from max_emprunteurs() natural join emprunt
natural join livre
natural join oeuvre;

 nom_adherent |          titre
--------------+--------------------------
     Diop     | Maîtriser les bases de données
     Diop     | Java programming with Oracle
     Diop     | Database Management Systems
     Diop     | PL/SQL pour Oracle 10g
(3 rows)
```


- Langage procédural intégré à PostgreSQL
- Utilisé essentiellement pour écrire des déclencheurs
- Variante (très proche) du langage PL/SQL du SGBD Oracle.
- Activé par défaut sur PostgreSQL
- Fonction stockée dans la base de données
- Interprétée au niveau du serveur
- Utilisation possible de tous les types, opérateurs et fonctions du SQL.

Les objectifs de la conception de PL/pgSQL ont été de créer un langage de procédures chargeable qui

- est utilisé pour créer des fonctions standards et triggers,
- ajoute des structures de contrôle au langage SQL,
- permet d'effectuer des traitements complexes,
- hérite de tous les types, fonctions et opérateurs définis par les utilisateurs,
- est défini comme digne de confiance par le serveur,
- est facile à utiliser.

Chaque application client doit

- envoyer chaque requête au serveur de bases de données,
- attendre que celui-ci la traite,
- recevoir et traiter les résultats, faire quelques traitements,
- et enfin envoyer d'autres requêtes au serveur

$\Rightarrow$ surcharge nécessaire à la communication client/serveur.

PL/pgSQL permet

- de grouper un bloc de traitement et une série de requêtes au sein du serveur de bases de données,
- de bénéficier ainsi de la puissance d'un langage de procédures,

$\Rightarrow$ de supprimer la surcharge nécessaire à la communication C/S

- Élimination des allers/retours entre le client et le serveur
- Pas nécessaire de traiter ou transférer entre le client et le serveur les résultats intermédiaires dont le client n'a pas besoin

- Fonction stockée dans la base de données
- Interprétée au niveau du serveur
- Exemple (création + appel de fonction) :

```
CREATE FUNCTION ajoute(a integer, b integer)
RETURNS integer AS $$
BEGIN
  RETURN a+b;
END;
$$
LANGUAGE plpgsql;

SELECT ajoute(5,3);
 ajoute
--------
      8
(1 row)
```



```
create function prix_ttc(prix_ht real)
     returns real as $$
declare
  taxe real;
begin
  taxe := 0.196;
  return prix_ht * (1 + taxe);
end
$$ language plpgsql;
```


- Le script est encadré par des $$
  - fonctione pour toutes les chaînes de caratères
  - Particulierement utile pour les corps des fonctions

```
CREATE OR REPLACE FUNCTION fun () RETURNS text AS
$Q$
  DECLARE
      result text;
  BEGIN
      PERFORM 'SELECT 1+1';
      RETURN 'ok';
END; $Q$
LANGUAGE plpgsql
mydb=# select fun();
 fun
-------
 ok
(1 row)
```


- l'écriture des instructions du langage pas sensible à la casse.
- Toute déclaration, bloc ou instruction se termine par un ";".
- Les commentaires sont compris entre /* et */
- Tout ce qui n'est pas reconnu comme une instruction PL/pgSQL est présumé être une commande SQL et est envoyé au moteur principal de bases de données.

- PL/pgSQL est un langage structuré en blocs. Un bloc est une suite de commandes SQL parenthésée par les commandes DECLARE/BEGIN et END.

```
[ <<label>> ]
[ DECLARE
      -- définition des variables]
 BEGIN
      -- corps de la fonction
 END;
```

- Les blocs peuvent être imbriqués entre eux et/ou se suivre
- Les blocs déterminent la portée (ou visibilité) des variables.

- Chaque expression de la section expression d'un bloc peut être un sous-bloc.
- Les sous-blocs peuvent être utilisés pour des groupements logiques ou pour localiser des variables locales à un petit groupe d'instructions.
- Les variables déclarées dans la section déclaration précédant un bloc sont initialisées à leur valeur par défaut chaque fois qu'on entre dans un bloc et pas seulement une fois à chaque appel de fonction.


```
CREATE FUNCTION somefunc() RETURNS integer AS $$
<< outerblock >>
DECLARE
  quantity integer := 30;
BEGIN
  RAISE NOTICE 'Quantity : %', quantity; -- Prints 30
  quantity := 50;
  -- Create a subblock
  DECLARE
    quantity integer := 80;
  BEGIN
    RAISE NOTICE 'Quantity : %', quantity;
    --Prints 80
    RAISE NOTICE 'Outer quantity : %', outerblock.quantity;
    --Prints 50
  END;

  RAISE NOTICE 'Quantity : %', quantity;   --Prints 50

  RETURN quantity;
END;
$$ LANGUAGE plpgsql;
```


- NE pas confondre l'utilisation de BEGIN/END pour grouper les instructions dans PL/pgSQL avec les commandes de bases de données pour le contrôle des transactions.
- Les BEGIN/END de PL/pgSQL ne servent qu'au groupement ; ils ne débutent ni ne terminent une transaction.
- Les procédures fonctions et déclencheurs sont toujours exécutées à l'intérieur d'une transaction établie par une requête extérieure (ils ne peuvent pas être utilisés pour commencer ou effectuer un commit sur une transaction puisque PostgreSQL ne gère pas les transactions imbriquées).

- Une fonction PL/pgSQL est une chaine de caractères : les chaines de caractères apparaissant dans le corps de la fonction sont délimitées par \' ou '' ou encore \047.

```
CREATE FUNCTION name ( [ argtype [, ...] ] )
RETURNS return_type AS '
definition
' LANGUAGE plpgsql
[ WITH ( attribute [, ...] ) ]
```


- Les paramètres sont définis uniquement par leur type `argtype` et sont automatiquement nommés `$i` où `i` est le rang du paramètre dans la liste des paramètres, le premier est `$1`.
- Les paramètres peuvent être renommés par la commande : `parameter_name ALIAS FOR $rang`
  où rang est un entier désignant le rang du paramètre à renommer en `parameter_name` .
- a valeur de retour d'une fonction :
  - est définie par son type dans la clause RETURNS `return_type`,
  - est retournée par la commande RETURN expression.

Suppression d'une fonction
DROP FUNCTION nomFonction()

Deux styles de commentaires :

1. `--` : double tiret introduit un commentaire sur le reste de la ligne.
2. `/* */` : commentaire type C pour plusieurs lignes.

```
-- cette ligne est un commentaire
/* Ces
   lignes
   sont des
   commentaires
*/
```

Les blocs de commentaires ne peuvent pas être imbriqués, mais les commentaires de lignes (double tiret) peuvent être contenus dans un bloc de commentaire et un double tiret peut cacher les délimiteurs du bloc de commentaire /* et */.

Quatre façons d'introduire de nouvelles variables :

1. Paramètres Positionnels : Chaque paramètre définit une nouvelle variable.
   Exemple : définition de deux variables `$1` de type `text` et `$2` de type `decimal(4,2)`.
   `CREATE or replace FUNCTION moyenne(text,decimal(4,2))`
2. ALIAS : Un paramètre peut recevoir un autre nom en utilisant la commande ALIAS .
   Exemple :

```
DECLARE
Nom ALIAS FOR $1;
Note ALIAS FOR $2;
```

   Utile si les variables sont NEW et OLD dans les triggers. A noter que c'est une déclaration alors à mettre dans DECLARE.

```
CREATE FUNCTION add(integer, integer)
       RETURNS INTEGER AS $$
DECLARE
  q INTEGER := 0;
BEGIN
q := $1+$2;
RETURN q; END
$$ LANGUAGE plpgsql;

postgres=# select add(1,2);
 add
-----
   3
(1 row)
```


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

```
DECLARE
  chaine       VARCHAR(8);
  attribut     table.colonne%TYPE;
  tuple        table%ROWTYPE;
  indef         record;
  ....
```

Si pas de valeur par défaut, la variable est assignée la valeur NULL.

4 FOR : La commande d'itération FOR permet de déclarer automatiquement une va- riable entière. Dans l'exemple suivant, deux variables i sont definies.

```
DECLARE
   i INT;
BEGIN
   FOR i IN 1 .. 12 LOOP
     ....
   END LOOP;
   ....
```


- Tous les types du langage SQL peuvent êtres utilisés.
- Le type correspondant à la ligne d'une table :

```
DECLARE
e Etudiant%ROWTYPE;
BEGIN
  return e.age+1;
```

- Le type d'une colonne d'une table :
  `nomE Personne.nom%TYPE;`
- Un type enregistrement sans structure prédéfinie `RECORD`
  - Les variables record sont similaires aux variables de type ligne, mais n'ont pas de structure prédéfinie.
  - Elles empruntent la structure effective de type ligne de la ligne à laquelle elles sont assignées durant une commande SELECT ou FOR jusqu'à ce qu'elle ait été assignée,
  - elle n'a pas de sous-structure, et toutes les tentatives pour accéder à un de ses champs entraineront une erreur d'exécution.

`variable%TYPE`

- `%TYPE` fournit le type de données d'une variable ou d'une colonne de table.
- On peut l'utiliser pour déclarer des variables qui contiendront des valeurs de bases de données.
- Par exemple, disons que vous avez une colonne nommée `user_id` dans votre table users. Pour déclarer une variable du même type de données que `users.user_id` vous pouvez écrire :
  `user_id users.user_id%TYPE;`
- En utilisant `%TYPE` vous n'avez pas besoin de connaître le type de données de la structure à laquelle vous faites référence
- Si le type de données de l'objet référencé change dans le futur (par exemple : vous changez le type de `user_id` de integer à real), vous pouvez ne pas avoir besoin de changer votre définition de fonction.

- `target := valeur;`
  Exemple

```
start_date := current_date;
```

- `target := select-query;`

```
Nom := SELECT nomEtu
           FROM etudiant
           WHERE codEtu = 345634;
```

- `SELECT list INTO destination FROM ...;`

```
SELECT * INTO myrecord
FROM etudiant
WHERE codEtu = 345634;
```

La construction

```
expression::type
```

permet de faire un « cast »et changer le type d'une expression

`SELECT liste-select INTO destination FROM ...;`

- Stocke dans une variable de type RECORD ou ROWTYPE le résultat d'une sélection.
- Si cette sélection renvoie plusieurs lignes (tuples), seule la première est conservée. Le résultat d'un select renvoyant 1 enregistrement peut être affecté
  - à une variable de type ROWTYPE
  - à une liste de variables
- A T T E N T I O N : Il ne faut pas confondre cette instruction avec le SELECT INTO du SQL pur, qui elle stocke le résultat de la sélection dans une nouvelle table


```
CREATE OR REPLACE FUNCTION format ()
RETURNS text AS $$
  DECLARE
    tmp RECORD;
  BEGIN
    SELECT INTO tmp 1 + 1 AS a, 2 + 2 AS b;
    RETURN 'a = ' || tmp.a || ' b = ' || tmp.b;
  END;
$$ LANGUAGE plpgsql;

mydb=# select format();
   format
-------------
 a = 2 b = 4
(1 row)
```



```
DECLARE
  p1 personne%ROWTYPE;
  nump integer;
  nomp varchar;
BEGIN
  select * into p1
  from personne
  where nom_personne='Toto';

  select * into nump,nomp
  from personne
  where nom_personne='Toto';
```

Avec FOUND on pourra vérifier si la requête a effectivement retourné une ligne ou non.

```
CREATE OR REPLACE FUNCTION intot(ingredients.ingredientid%TYPE)
   RETURNS ingredients.unit%TYPE AS $$
DECLARE
   a ingredients%ROWTYPE; --Declaration d'une variable du type:
                          --ligne de la table ingredients
BEGIN
   -- Maintenant on peut faire SELECT dans a:
   SELECT * INTO a
   FROM ingredients
   WHERE $1 = ingredients.ingredientid ;
   IF FOUND THEN --FOUND est TRUE si le select
                 --a trouve des lignes
      RETURN a.unit;
   ELSE
      RETURN 'unknown' ;
   END IF;
END;
$$ LANGUAGE 'plpgsql';
```


Destination peut-être une liste de variables :

```
CREATE OR REPLACE FUNCTION intot(ingredients.ingredientid%TYPE)
   RETURNS TEXT AS $$
DECLARE
   a ingredients.unit%TYPE;
   b ingredients.name%TYPE;
BEGIN
   SELECT ingredients.unit, ingredients.name  INTO a,b
   FROM ingredients
   WHERE $1 = ingredients.ingredientid ;
   IF FOUND THEN
      RETURN a||' '||b ;
   ELSE
      RETURN 'unknown' ;
   END IF;
END;
$$ LANGUAGE 'plpgsql';
```


- RETURN permet de renvoyer les données d'une fonction.
- Le type doit correspondre à celui indiqué dans le RETURNS
- Le type de retour VOID permet de se passer du RETURN.

Exemple :

```
CREATE FUNCTION getNote (numE INTEGER)
    RETURNS INT AS $$
DECLARE
       valeur INT ;
BEGIN
    SELECT note INTO valeur
    FROM evaluation
    WHERE codeEtu= numE;
    RETURN valeur;
 END ;
 $$ LANGUAGE plpgsql;
```

NB : pour supprimer la fonction : Drop function getNote(integer) ;

Les procédures PostgreSQL sont des fonctions qui retournent void.

```
drop table if exists scores;
create table scores ( equipe varchar(25), points int );
create or replace function
     majPoints( iequipe varchar, iscore int )
returns void as $$
begin
  update scores set points = points + iscore
  where equipe = iequipe;
  if not found then
    insert into scores ( equipe, points )
    values ( iequipe, iscore );
  end if;
end;
$$
language plpgsql;
```

Cette procédure stockée est créée et utilisée ainsi :

```
postgres=# select majPoints( 'Les vétérans', 15 );
 majPoints
------------
(1 row)
postgres=# select * from scores;
   equipe     | points
--------------+--------
Les vétérans |     15
(1 row)
```


Retourner un ensemble de données

- Type de retour (spécifiés dans RETURNS) :
  - SETOF nom_table ;
  - TABLE(var1 TYPE1, var2 TYPE2...)
- RETURN NEXT
  - Renvoie un élément d'une séquence
- RETURN QUERY
  - Renvoie le résultat d'une requête
  - Une suite de plusieurs RETURN NEXT et RETURN QUERY doit finir par RETURN ;

Retourner un ensemble de données

```
CREATE TABLE truc (id_truc INT, sousid_truc INT, nom_truc TEXT);
INSERT INTO truc VALUES (1, 2, 'trois');
INSERT INTO truc VALUES (4, 5, 'six');
CREATE OR REPLACE FUNCTION TousLesTrucs()
         RETURNS SETOF truc AS $$
  DECLARE
         r truc%rowtype;
  BEGIN
    FOR r IN SELECT * FROM truc WHERE id_truc > 0
    LOOP
        -- quelques traitements
        RETURN NEXT r; -- renvoie la ligne courante du SELECT
    END LOOP;
    RETURN;
  END
$$LANGUAGE plpgsql;
```


Retourner un ensemble de données

```
 SELECT * FROM TousLesTrucs();
 id_truc | sousid_truc | nom_truc
---------+-------------+----------
       1 |           2 | trois
       4 |           5 | six
(2 rows)
```

```
SELECT expressions_select INTO [STRICT] cible FROM ...;
INSERT ... RETURNING expressions INTO [STRICT] cible;
UPDATE ... RETURNING expressions INTO [STRICT] cible;
DELETE ... RETURNING expressions INTO [STRICT] cible;
```

où `cible` peut être une variable de type `record`, `row` ou une liste de variables ou de champs `record/row` séparées par des virgules.
Exemple

```
select * into monrec from emp where nom = mon_nom;
select * into strict monrec from emp where nom = mon_nom;
```

- Si l'option STRICT est indiquée, la requête doit retourner exactement une ligne. Dans le cas contraire, une erreur sera rapportée à l'exécution.
- Si STRICT n'est pas spécifié dans la clause INTO, alors cible sera configuré avec la première ligne retournée par la requête ou à NULL si la requête n'a retourné aucune ligne.

- Notez que « la première ligne » n'est bien définie que si vous avez utilisé ORDER BY.
- Vous pouvez vérifier la valeur de la variable spéciale FOUND pour déterminer si une ligne a été retournée :

```
select * into monrec from emp where nom = mon_nom;
if not found then
   raise exception 'employe introuvable';
end if;
```



```
PERFORM requête;
```

- écrivez la requête de la même façon qu'une instruction SELECT mais remplacez le mot clé initial SELECT avec PERFORM.
- Une instruction SQL qui peut retourner des lignes comme SELECT sera rejetée comme une erreur si elle n'a pas de clause INTO.


```
CREATE FUNCTION myFonct() RETURNS void AS $$
   BEGIN
        PERFORM *
        FROM etudiants E
        WHERE E.noEtu = 12;
    IF NOT FOUND THEN
        RAISE NOTICE 'Pas trouve...';
    END IF;
  END;
$$ LANGUAGE plpgsql;
```

`EXECUTE chaîne [ INTO [STRICT] cible ] USING exp, ... ];`
Les expressions `USING` fournissent des valeurs à insérer dans l'instruction.

```
execute
'select * from adherent where prix < $1 and date < $2'
 into resultat
using max_prix, max_date;
```

Notez que USING peut seulement être utilisé avec des valeurs de données.
Si vous voulez utiliser des noms de tables ou colonnes déterminés dynamiquement, vous devez les insérer dans la chaîne.

```
execute 'select * from '
|| nomtable::regclass
|| ' where prix < $1 and date < $2' into resultat
using max_prix, max_date;
```


La première méthode pour déterminer l'effet d'une instruction est d'utiliser `GET DIAGNOSTICS`.
`GET [ CURRENT ] DIAGNOSTICS variable = élément , ...;`
Les éléments d'état actuellement disponibles sont :

- `ROW_COUNT` :le nombre de lignes traitées par la dernière instruction.
- `RESULT_OID` : l'OID de la dernière ligne insérée par l'instruction SQL la plus récente dans une table contenant des OID.

La seconde méthode est la variable spéciale FOUND de type boolean.
FOUND est initialisée à false au début de chaque instruction PL/pgSQL, ensuite :

- SELECT INTO, PERFORM et FETCH :
  true si au moins une ligne est retournée, *false* sinon.
- UPDATE,INSERT et DELETE :
  true si au moins une ligne est affectée, *false* sinon.
- MOVE : *true* si repositionne le curseur avec succès, *false* sinon.
- FOR et FOREACH : *true* s'il y a au moins une itération *false* sinon.
  FOUND n'est pas modifié à l'intérieur de la boucle, bien qu'il pourrait être modifié par l'exécution d'autres requêtes dans la boucle.

- RETURN QUERY et RETURN QUREY EXECUTE :
  true si au moins une ligne est retournée, *false* sinon.

Les autres instructions PL/pgSQL ne changent pas l'état de FOUND.
EXECUTE modifie la sortie de GET DIAGNOSTICS mais pas FOUND.
FOUND est une variable locale à l'intérieur de chaque fonction PL/pgSQL
Instructions de base : Ne rien faire du tout

```
NULL;
```

Commande IF-THEN-ELSE

```
IF expression-booleenne THEN
   instructions
ELSE
   instructions
END IF;
```

- Le bloc ELSE est facultatif
- On peut ajout un bloc ELSEIF pour enchaîner les IF


```
CREATE OR REPLACE FUNCTION pair (i int)
RETURNS boolean AS $$
  DECLARE
    tmp int;
  BEGIN
    tmp := i % 2;
    IF tmp = 0 THEN RETURN true;
    ELSE RETURN false;
    END IF;
 END;
$$ LANGUAGE plpgsql;
SELECT pair(3), pair(42);
 pair | pair
------+------
   f  |   t
(1 row)
```


CASE

Deux formes de CASE

- Basée sur les valeurs des expressions :

```
CASE x
   WHEN 1, 2 THEN
      msg := 'un ou deux';
   ELSE
      msg := 'autre valeur que un ou deux';
END CASE;
```

- Basée sur des expressions booléennes :

```
CASE
   WHEN x BETWEEN 0 AND 10 THEN
       msg := 'valeur entre zéro et dix';
   WHEN x BETWEEN 11 AND 20 THEN
       msg := 'valeur entre onze et vingt';
END CASE;
```


Les boucles LOOP

- boucle répétée indéfiniment jusqu'à ce qu'elle soit terminée par une instruction EXIT ou RETURN.

```
LOOP
   statements
   EXIT WHEN bool_expression;
END LOOP;
```

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

Exemple

```
LOOP
  -- quelques traitements
  IF nombre > 0 THEN
       EXIT; -- sortie de boucle
  END IF;
END LOOP;

LOOP
   -- quelques traitements
   EXIT WHEN nombre > 0; -- même chose qu'avec IF
END LOOP;
```


Les boucles WHILE

- répéte une séquence d'instructions aussi longtemps qu'une expression booléenne est vraie.

```
WHILE bool_expression LOOP
       statements
END LOOP;
```

Exemple :

```
WHILE n > 0 AND m > 0 LOOP
  -- quelques traitements ici
END LOOP;
```



```
drop function if exists pgcd(integer);
create or replace function pgcd(a integer, b integer) returns
    integer as $$
declare
va integer;
vb integer; -- car ils ne peuvent pas être modifiés.
begin
va:=a;
vb:=b;
 while va != vb loop
        if va > vb then
            va:=va-vb;
        else
            vb:=vb-va;
        end if;
    end loop;
return va;
end $$ language 'plpgsql';
mydb=# select pgcd(57, 135);
 pgcd
------
    3
(1 row)
mydb=# select * from pgcd(57, 135) ;
```



```
CREATE OR REPLACE FUNCTION factorial (i numeric)
RETURNS numeric AS $$
  DECLARE tmp numeric; result numeric;
  BEGIN
    result := 1; tmp := 1;
    WHILE tmp <= i LOOP
      result := result * tmp;
      tmp := tmp + 1;
    END LOOP;
    RETURN result;
  END;
$$ LANGUAGE plpgsql;
mydb=# SELECT factorial(39::numeric);
                    factorial
-----------------------------------------------------
 20397882081197443358640281739902897356800000000
(1 row)
```

Les boucles FOR (Avec entiers)

- Itération sur une plage de valeurs entières.
- La variable utilisée est de type integer
- Qui n'existe que dans la boucle
- La clause BY permet de régler l'itération (1 par défaut)
- Avec REVERSE la valeur de l'étape est soustraite, plutôt qu'ajoutée, après chaque itération.

```
FOR iterator IN [ REVERSE ] start_expr .. end_expr LOOP
   statements
END LOOP;
```

CONTINUE 'a l'intérieur de la boucle aura le même effet qu'en langage C, on retourne au début de la boucle. La forme générale : `CONTINUE [ etiquette ] [ WHEN expression-booleenne ];`

Exemples :

```
FOR i IN 1..10 LOOP
   -- instructions
END LOOP;

FOR i IN REVERSE 10..1 BY 2 LOOP
-- prend les valeurs 10,8,6,4,2 dans la boucle
END LOOP;
```

REVERSE indique que la valeur de l'étape est soustraite, plutôt qu'ajoutée. Si la limite inférieure est supérieure à la limite supérieure (ou inférieure dans le cas du REVERSE), le corps de la boucle n'est pas exécuté du tout. Aucune erreur n'est renvoyée.

Exemples :

```
CREATE OR REPLACE FUNCTION factorial (i numeric)
RETURNS numeric AS $$
  DECLARE
    tmp numeric; result numeric;
  BEGIN
    result := 1;
    FOR tmp IN 1 .. i LOOP
      result := result * tmp;
    END LOOP;
    RETURN result;
  END;
$$ LANGUAGE plpgsql;

mydb=# SELECT factorial(39::numeric);
                    factorial
-----------------------------------------------------
 20397882081197443358640281739902897356800000000
(1 row)
```


Par exemple, si le fichier repete.sql contient ce code :

```
create function repete(str varchar, nb int)
returns varchar as $$
declare
  res varchar := '';
begin
  for i in 1..nb loop
    res := res || str;
end loop;
  return res;
end;
$$
language plpgsql;
```

Alors, la création et l'exécution de cette fonction se font ainsi :

```
postgres=# \i repete.sql

CREATE FUNCTION
postgres=# select repete('@', 20);
        repete
----------------------
 @@@@@@@@@@@@@@@@@@@@
(1 row)
```

Les boucles FOR (dans les résultats d'une requête)

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

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

- iterator est une variable de type record, row ou une liste de variables scalaires séparées par une virgule.
- L'itérateur est affectée successivement à chaque ligne résultant de la requête et le corps de la boucle est exécuté pour chaque ligne.
- Si la boucle est terminée par une instruction EXIT, la dernière valeur ligne affectée est toujours accessible après la boucle.

Les boucles FOR (dans les résultats d'une requête)
Exemple

```
CREATE FUNCTION sauvegarde_client()
        RETURNS integer AS $$
DECLARE
        un_client RECORD;
BEGIN
    RAISE NOTICE 'Copie de table client...';
    FOR un_client IN SELECT * FROM client LOOP
        -- un_client contient une ligne de la table client
        INSERT INTO client_sauv
        VALUES (un_client.id, unclient.nom);
        RAISE NOTICE 'Copie du client %', un_client.id
    END LOOP;
    RETURN 1;
END;
 $$ LANGUAGE plpgsql;
```


FOREACH :

```
FOREACH cible [ SLICE nombre ] IN ARRAY expr LOOP
  instructions
END LOOP ;
```

- cible est peut être une variable scalaire ou une liste de variables lors d'une boucle dans un tableau de valeurs composites.
- Avec une valeur SLICE positive, FOREACH itère au travers des morceaux du tableau plutôt que des éléments seuls. La variable cible doit être un tableau et elle reçoit les morceaux successifs de la valeur du tableau.

Exemple

```
create function print_lignes(int[][]) returns void as $$
declare
x int[];
begin
foreach x slice 1 in array $1 loop
raise notice 'ligne = %', x; end loop;
end;
$$ language plpgsql;

mydb=# select print_lignes(array[[1,2,3],[4,5,6],[7,8,9],
                [10 ,11 ,12]]);
NOTICE:  ligne = {1,2,3}
NOTICE:  ligne = {4,5,6}
NOTICE:  ligne = {7,8,9}
NOTICE:  ligne = {10,11,12}
 print_lignes
--------------

(1 row)
```

```
 CREATE OR REPLACE FUNCTION factorial (i numeric)
RETURNS numeric AS $$
  BEGIN
    IF i = 0 THEN
        RETURN 1;
    ELSIF i = 1 THEN
        RETURN 1;
    ELSE
        RETURN i * factorial(i - 1);
    END IF;
 END;
$$ LANGUAGE plpgsql;
SELECT factorial(39::numeric);
                      factorial
-----------------------------------------------------
 20397882081197443358640281739902897356800000000
(1 row)
```


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

```
CREATE FUNCTION name([[argname][argmode] argtype [, ...]])
    RETURNS return_type AS $$
            définition
$$ LANGUAGE plpgsql
```

On distingue trois modes IN, OUT, INOUT

Paramètres IN

- permet de transmettre une valeur à la fonction,
- ne permet pas de retourner une valeur ,
- ne peut être modifié.
- mode par défaut

Paramètres OUT

- permet de définir une valeur de retour,
- se comporte comme une variable non initialisée : valeur intiale indéterminée (NULL).
- sa valeur doit étre déterminer par la fonction.

Paramètres INOUT

- permet à la fois de transmettre une valeur à la fonction, et de définir une valeur de retour,
- comporte comme une variable initialisée : par la valeur transmise en argument.


```
create table devoir(id integer primary key, nom text,
note decimal(4,2));
insert into devoir values(1,'Rama',10);
insert into devoir values(2,'Lamarana',12);
insert into devoir values(3,'Salim',14);
CREATE FUNCTION  note (idEl IN int,nomEl INOUT text,noteEl
                       OUT decimal(4,2)) AS $body$
begin
  SELECT devoir.note into noteEl
  FROM devoir
  WHERE devoir.id=idEl and devoir.nom=nomEl;
end;
$body$ LANGUAGE plpgsql;
 select note(2,'Lamarana');
       note
------------------
 (Lamarana,12.00)
(1 row)
 select * from  note(2,'Lamarana');
  nomel    | noteel
----------+--------
 Lamarana |  12.00
(1 row)
```

```
CREATE FUNCTION ParametresINOUT (
a IN int,b INOUT int ,c OUT int)
AS  $body$
begin
   c:=a; -- a : en lecture seule
   c:=c*b;
   b:=a+b; -- b : en lecture/écriture
 end;
$body$ LANGUAGE plpgsql;

select ParametresINOUT(4,6);
 parametresinout
-----------------
 (10,24)
(1 row)
```



```
CREATE FUNCTION LigneComplete(tab devoir)
    RETURNS text AS  $body$
BEGIN
  RETURN tab.id ||' '||tab.nom||' '||tab.note;
END;
$body$ LANGUAGE plpgsql;

select LigneComplete(devoir.*) from devoir;
  lignecomplete
------------------
 1 Rama 10.00
 2 Lamarana 12.00
 3 Salim 14.00
(3 rows)
```



```
CREATE or replace FUNCTION note (INOUT nomEl text, OUT noteEl decimal(
   AS $body$
 declare
   tuple record;
begin
   SELECT devoir.note into noteEl
   FROM devoir
   WHERE devoir.nom=nomEl;
  RETURN;
end;
$body$ LANGUAGE plpgsql;

select note('Lamarana');
       note
------------------
 (Lamarana,12.00)
(1 row)
```


Comme Java PL/pgSQL supporte plusieurs fonctions avec le même nom si les signatures différentes. our supprimer une fonction définie par utilisateur on utilise :
DROP FUNCTION nom-fonction(parametres) ; l'exemple suivant définit trois fonctions qui portent le même nom.

Plusieurs fonctions avec le même nom : surcharge 245 I

```
------------------------- compute_date.sql --------------------------
CREATE OR REPLACE FUNCTION compute_due_date(DATE) RETURNS DATE AS $$
   DECLARE            ------ declaration de variables
     due_date        DATE;
     rental_period   INTERVAL := '7 days';
   BEGIN
     due_date := $1 + rental_period;
     RETURN due_date;
   END;
 $$ LANGUAGE 'plpgsql';
---------------------------------------------------------------------
CREATE OR REPLACE FUNCTION compute_due_date(DATE, INTERVAL) RETURNS DA
   BEGIN
-- c'est un commentaire dans une fonction
     RETURN( $1 + $2 );
   END;
 $$ LANGUAGE 'plpgsql';
-- SELECT compute_due_date(current_date,INTERVAL '1 MONTH');
---------------------------------------------------------------------
CREATE OR REPLACE FUNCTION compute_due_date(dans INTERVAL) RETURNS DATE
```

Plusieurs fonctions avec le même nom : surcharge 246 II

```
  BEGIN
    RETURN current_date + dans;
  END
$$ LANGUAGE 'plpgsql';
-- select compute_due_date(interval '3 months');
---------------------------------------------------------------------


CREATE OR REPLACE FUNCTION foo (text) RETURNS text AS $$
  SELECT $1
$$ LANGUAGE sql;
CREATE OR REPLACE FUNCTION foo (int) RETURNS text AS $$
  SELECT ($1 + 1)::text
$$ LANGUAGE sql;
SELECT foo('52'), foo(51);
 foo | foo
-----+-----
 52  | 52
(1 row)
```

Comportement avec NULL en entrée 247 I

- CALLED ON NULL INPUT (par défaut) Fonction appelée normalement pour les valeurs NULL en entrée
- RETURNS NULL ON NULL INPUT
  Fonction pas appelée quand des valeurs NULL sont présentes.
  A la place NULL est renvoyée automatiquement.

```
CREATE FUNCTION add1 (int, int) RETURNS int AS $$
  SELECT $1 + $2
$$ LANGUAGE SQL RETURNS NULL ON NULL INPUT;
CREATE FUNCTION add2 (int, int) RETURNS int AS $$
  SELECT COALESCE($1, 0) + COALESCE($2, 0)
$$ LANGUAGE SQL CALLED ON NULL INPUT;
SELECT add1(7, NULL) IS NULL AS "true", add2(7, NULL);
 true | add2
------+------
   t  |  7
(1 row)
```

Comportement avec NULL en entrée 248 II

La fonction COALESCE accepte un nombre illimité d'arguments. Elle retourne le premier argument non nul. Si tous les arguments sont non nuls le fonction COALESCE retournera NULL

- Rapporter des messages et lever des erreurs.
  Syntaxe :
  `RAISE niveau 'format' [, expression [, ...]];`
- L'option niveau indique la sévérité de l'erreur.
  - DEBUG
  - LOG
  - INFO
  - NOTICE
  - WARNING
  - EXCEPTION (Niveau par défaut, seul niveau qui lève une erreur)
- Exemples :

```
RAISE EXCEPTION 'Erreur';
RAISE NOTICE 'Appel de creer_user(%)', id_user;
```


Syntaxe : `RAISE niveau 'format' [, expression [, ...]];`

- On peut ajouter des expressions optionnelles à insérer dans le message.
  - % est remplacé par la représentation de la valeur du prochain argument.
  - %% sert pour afficher une autre chaîne de caractères

  Exemples :

```
RAISE EXCEPTION 'Erreur';
RAISE NOTICE 'Appel de creer_user(%)', id_user;
```



```
insert into mon_tableau(prenom, nom)
values('Tom', 'Jones');
begin
 update mon_tableau
 set prenom = 'Joe'
 where nom = 'Jones';
 x := x + 1;
 y := x / 0;
 exception
 when division_by_zero then
    raise notice 'recuperation de l''erreur...';
 return x;
end;
```

L'exécution de la ligne (y := x / 0 ;) échouera avec une erreur division_by_zero. Cela sera récupérée par la clause EXCEPTION. La valeur retournée par RETURN sera la valeur incrémentée de x mais les effets de l'instruction UPDATE auront été annulés. L'instruction INSERT précédant le bloc ne sera pas annulée


```
CREATE OR REPLACE FUNCTION plus(v_a integer,v_b integer)
RETURNS integer AS $$
BEGIN
  RAISE INFO 'fonction plus : addition de % et %', v_a, v_b;
  -- RAISE WARNING 'fonction plus : addition de % et %', v_a, v_b
  -- RAISE NOTICE 'fonction plus : addition de % et %', v_a, v_b;
  Return v_a + v_b;
END $$ LANGUAGE 'plpgsql' ;

mydb=# select plus(2,5);
INFO:  fonction plus : addition de 2 et 5
 plus
------
    7
(1 row)
```

```
CREATE OR REPLACE
FUNCTION main(v_a integer,v_b integer) RETURNS void AS $$
DECLARE
    res integer;
    BEGIN
      RAISE INFO 'main : appel à la fonction plus';
      res = plus(v_a, v_b) ;
      RAISE NOTICE 'résultat = %', res ;
    END
$$ LANGUAGE 'plpgsql' ;

mydb=# select main(2,5);
INFO:  main : appel à la fonction plus
INFO:  fonction plus : addition de 2 et 5
CONTEXT:  PL/pgSQL function main(integer,integer) line 6 at assignment
NOTICE:  résultat = 7
 main
------

(1 row)
```


- SECURITY INVOKER (par défaut)
  Fonction executée avec les droits de l'utilisateur courant
- SECURITY DEFINER
  Fonction executée avec les droits du créateur

```
CREATE TABLE zik (f1 int);
REVOKE ALL ON zik FROM public;
CREATE FUNCTION see_zik() RETURNS SETOF zik AS $$
  SELECT * FROM zik
$$ LANGUAGE SQL SECURITY DEFINER;
\c - guest
You are now connected to database "postgres" as user "guest".
SELECT * FROM zik;
ERROR:  permission denied for relation zik
SELECT * FROM see_zik();
f1
----
(0 rows)
```


- Programme stocké dans une base de données.
- Associé à une table (ou une vue)de la base de donnée.
- Associé à un événement qui se produit sur cette table.
- Exécuté automatiquement lorsque l'événement auquel il est attaché se produit sur la table

Formellement, une procédure trigger est une règle de la forme :
Evenement $\rightarrow$ *Action*

- L'action*Action* est définie par une fonction ayant une signature spécifique :
  `function_name () RETURNS TRIGGER ...`
  1. la fonction associée à un trigger n'a pas d'arguments
  2. et sa valeur de retour est de type particulier : TRIGGER.
- L'évenement *Evenement* est spécifié par un déclencheur (trigger). Un trigger est un mécanisme qui permet d'associer une action à un évenement. l'événement peut être soit INSERT, UPDATE ou DELETE.

Déclencheurs : Le choix du langage 257

- Plusieurs langages peuvent être utilisés pour écrire des triggers.
- Les langages supportés varient d'un SGBD à l'autre.
- Oracle : PL/SQL , Java
- PostgreSQL : PL/PGSQL, python, ruby, C...

Création d'une procédure trigger

```
CREATE FUNCTION function_identifier ()
   RETURNS TRIGGER AS '
DECLARE
   -- declarations;
  BEGIN
    -- statements;
  END;
'LANGUAGE 'plpgsql'';
```

Création d'une procédure trigger 258

```
CREATE TRIGGER avant_inserer_personne
   BEFORE INSERT ON personne
   FOR EACH ROW
   EXECUTE PROCEDURE generer_cle();
   CREATE FUNCTION generer_cle ()
      RETURNS TRIGGER AS $$
      DECLARE
          nopers integer;
      BEGIN
         select into nopers max(num_personne )
         from personne;
         IF nopers IS NULL THEN
            nopers:=0;
         END IF;
         NEW.num_personne:=nopers+1;
         RETURN NEW;
      END;
$$LANGUAGE plpgsql;
```

Création d'une procédure trigger 259

```
CREATE TRIGGER nom_declencheur
{ BEFORE | AFTER | INSTEAD OF }
{ événement [ OR ... ] } ON nom_table
FOR EACH { ROW | STATEMENT } ]
EXECUTE PROCEDURE procedure_trigger( args )
```

où événement fait partie de : INSERT, UPDATE, DELETE,TRUNCATE. Le trigger peut agir :

- Avantl'événement(BEFORE) : Il peut alors :
  - modifier le tuple inséré ou mis à jour,
  - annuler l'opération.
- Après l'événement (AFTER) : Il ne peut pas :
  - modifier le tuple inserré ou mis à jour
  - annuler l'opération.
- INSTEAD OF (A la place)–INSTEAD OF ne s'applique que pour les vues
- Au niveau tuple :FOR EACH ROW :Pour chaque ligne de la table
- Au niveau requête : FOR EACH STATEMENT : Pour chaque instruction (une fois)

Déclencheurs : Arguments 260 I

Quand une procédure trigger est appelé par le "gestionnaire de triggers", aucun argument normal ne lui est transmis.
Les arguments d'une fonction trigger sont transmis dans une structure particulière comportant :

- `TG_NARGS` (integer) : le nombre d'arguments donnés à la fonction trigger dans la commande CREATE TRIGGER.
- `TG_ARGV[]` (text) : les arguments de la commande CREATE TRIGGER. l'index débute à 0. Les indices invalides (inférieurs à 0 ou supérieurs ou égaux à `TG_NARGS`) auront une valeur nulle.
- `NEW` (record) : variable contenant le nouveau tuple pour les opérations INSERT/UPDATE dans les trigger de niveau tuple. Cette variable est NULL dans un trigger de niveau commande.
- `OLD` (record) : variable contenant l'ancien tuple pour les opérations DELETE/UPDATE dans les trigger de niveau tuple. Cette variable est NULL dans un trigger de niveau commande.

- `TG_LEVEL` (text) : vaut soit ROW soit STATEMENT selon la définition du trigger.
- `TG_WHEN` (text) : vaut soit BEFORE soit AFTER selon la définition du trigger.
- `TG_OP` (text) : vaut soit INSERT, UPDATE, ou DELETE indiquant pour quel évene- ment a déclenché le trigger.
- `TG_RELID` (oid) : l'identifiant de la table qui a déclenché le trigger.
- `TG_RELNAME` (name) : le nom de la table qui a déclenché le trigger.
- `TG_NAME` (name) : le nom du trigger déclenché.

Dans les procédures trigger PLpgSQL, le tuple pour lequel la trigger est déclenché est accessible par des variables SQL.

- NEW : est l'image du tuple à insérer ou après mis à jour. NEW variable enregistrement (type RECORD) contenant la nouvelle ligne → pour les opérations INSERT/UPDATE.
- OLD : est l'image du tuple à supprimer ou avant mise à jour. OLD variable enregistrement contenant l'ancienne ligne → pour les opérations UPDATE/DELETE

Chaque attribut de l'image du tuple est accessible en utilisant la notation pointé de la forme NEW.champ ou OLD.champ où champ est le nom d'un attribut. La valeur d'un attribut peut être modifié par une expression d'affection.

- Une fonction pour déclencheurs doit renvoyer soit NULL soit une valeur RECORD ayant exactement la structure de la table pour laquelle le déclencheur a été lancé.
- Déclencheurs BEFORE :
  - NULL : annule l'action qui a déclenché la fonction ;
  - non NULL :l'opération se déroule avec la valeur renvoyée
- Pour les déclencheurs AFTER, la valeur de retour est toujours ignorée. Néanmoins, un déclencheur peut toujours annuler l'opération en cours avec une exception.

La valeur de retour d'une fonction trigger consiste à la fois en

- un tuple (données)
- une donnée de contrôle

Les valeurs de retour possibles sont OLD,NEW ou NULL.

Evénement	variable tuple
INSERT	NEW
DELETE	OLD
UPDATE	OLD NEW

Une procédure trigger (comme toute autre procédure) est toujours éxécutée dans une tran- saction : celle de la requête qui l'a déclenchée.

Type	Evenement	Valeur de Retour	
		tuple	controle
BEFORE	INSERT	NULL	Ignorer le tuple sans avorter la Transaction
		NEW	Insere le tuple
	UPDATE	NULL	Ignorer le tuple sans avorter la Transaction
		OLD	Ignorer le tuple sans avorter la Transaction
		NEW	Mettre à jour du tuple
	DELETE	NULL	Ignorer le tuple sans avorter la Transaction
		OLD	Supprimer le tuple
AFTER	INSERT	NULL	Valeur de retour ignorée
	UPDATE	NULL	Valeur de retour ignorée
	DELETE	NULL	Valeur de retour ignorée


```
CREATE TABLE Evaluation (
        numEtu      int,
        codeMat      char(5),
        note        decimal(4,2)
);
CREATE FUNCTION newNote() returns TRIGGER as '
BEGIN
   IF NEW.note < OLD.note THEN
      NEW.note = OLD.note;
   END IF;
   RETURN NEW;
END;
' language 'plpgsql';
CREATE TRIGGER newNote
BEFORE UPDATE on Evaluation
FOR EACH ROW EXECUTE PROCEDURE newNote();
```



```
create table commande_article(
    id_article int,
    id_commande int ,
   qte int);
create function au_mois_dix() returns trigger
 as $$
 begin
   if (new.qte < 10) then
      new.qte = 10;
   end if;
   return new;
 end;
 $$ language plpgsql
create trigger tr_commande
  before insert or update on commande_article
  for each row
    execute procedure au_moins_dix ()
```


- Plusieurs déclencheurs peuvent être lancés par une même action. Ils sont exécutés les uns à la suite des autre par ordre alphabétique (! !) sur leur nom. L'élément retourné par un déclencheur devient l'élément entrant (NEW) du suivant.
- Le premier déclencheur qui retourne NULL annule l'ensemble de l'action sur l'enregistrement courant.
- La programmation des déclencheurs nécessite donc d'une analyse fine de leurs enchaînements (et de préférence simple, avec peu de niveau de déclenchement en cascade), et doit être bien documentée.

- Les déclencheurs par instruction (FOR EACH STATEMENT) sont exécutés une seule fois pour une instruction, même si cette instruction concerne plusieurs enregistrements.
- Les déclencheurs BEFORE par instruction sont exécutés aprés tous les déclencheurs BEFORE par enregistrement (et de même pour les déclencheurs AFTER par enregistrement et par instruction).
- Les pseudo-variables NEW et OLD ne sont pas accessibles dans un déclencheur par instruction.
- Les déclencheurs par instruction sont utiles notamment pour les actions qui nécessitent d'accéder à la table concernée par le déclencheur.

Par exemple, dans la base biblio, nous voulons enregistrer quelques informations statistiques dans la table proprietes(id, nom, valeur). Notamment, le nombre de livres empruntés actuellement :


```
create function maj_nb_emprunts() returns trigger
as $$
declare
  nb int;
begin
  select count(id_livre) into nb
  from emprunt;
  update proprietes
  set valeur = nb
  where nom = 'nb_emprunts';
end;
$$ language plpgsql;

create trigger tr_maj_nb_emprunts
after insert or delete
on emprunt
for each statement
 execute procedure maj_nb_emprunts ();
```


Règles de base

- Trigger ROW - BEFORE : ne voit pas les majs (tuple) de la requête qui l'a déclenché. Par contre, il voit les majs des autres tuples.
- ROW - AFTER : voit les majs (tuple) de la requête qui l'a déclenché.
- STATEMENT - AFTER : voit tous les tuples majs.
- STATEMENT - BEFORE : ne voit aucune maj.

```
create or replace function ajout_emprunt() returns trigger as $$
declare
   v_du Adherent.du%TYPE;
   v_nb_emprunts integer;
   v_sorti Livre.sorti%TYPE;
begin
  select du into v_du
  from Adherent
  where id_adherent = new.id_adherent;
  if not found then
    raise exception 'Adherent inconnu.';
  end if;
  if v_du <> 0 then
      raise notice 'L adherent doit %', v_du;
      return null;
  end if;
  select sorti into v_sorti
  from Livre
  where id_livre = new.id_livre;
  if not found then
     raise notice 'Livre inconnu : %', new.id_livre;
```



```
     return null;
  end if;
  if v_sorti then
     raise exception 'Livre sorti.';
  end if;
  select count(*) into v_nb_emprunts
  from Emprunt
  where id_adherent = new.id_adherent;
  if ((v_nb_emprunts is not null) and (v_nb_emprunts > 4)) then
      raise notice 'Trop d emprunts en cours.';
      return null;
  end if;
  return new;
end;
$$ language plpgsql;
create trigger ajout_emprunt
before inserton table Emprunt
for each row
        execute procedure ajout_emprunt ();
```



```
CREATE TABLE maTable (
   maDonnee int
);

CREATE  or REPLACE FUNCTION maFonction()
  RETURNS TRIGGER  AS '
  DECLARE
    i         int;
    Tab     name;
    Quand      text;
BEGIN
     Quand := TG_WHEN;
     Tab := TG_RELNAME;
     IF Tab!=''matable'' THEN
        raise exception ''% différé sur %'', TG_NAME,Tab;
     END IF;

     SELECT count(*) into i FROM maTable;
     raise notice ''% [lancé %] : Il y a % lignes dans % '',
                  TG_NAME, Quand ,i , Tab;
     IF TG_OP != ''DELETE'' AND Quand = ''BEFORE'' THEN
```



```
          IF NEW.maDonnee is NULL THEN
             return NULL;
          END IF;
     END IF;
     IF TG_OP= ''DELETE'' THEN
           RETURN OLD;
     else -- UPDATE INSERT
           RETURN NEW;
     end if;
END;
' LANGUAGE 'plpgsql';

CREATE TRIGGER trig_before
   BEFORE INSERT OR UPDATE OR DELETE ON maTable
   FOR EACH ROW EXECUTE PROCEDURE maFonction();
CREATE TRIGGER trig_after
   AFTER INSERT OR UPDATE OR DELETE ON maTable
   FOR EACH ROW EXECUTE PROCEDURE maFonction();
```

```
# select * from maTable;
 madonnee
----------
(0 rows)

# INSERT INTO matable VALUES (NULL);
NOTICE:  trig_before [lancé BEFORE] : Ily a 0 lignes dans matable
INSERT 0 0
-- Insertion ignorée => trig_after non déclenché
# SELECT * FROM matable;
 madonnee
----------
(0 rows)
INSERT INTO matable VALUES (1);
NOTICE:trig_before [lancé BEFORE]: Il y a 0 lignes dans matable
NOTICE:trig_after [lancé AFTER]: Il y a 1 lignes dans matable
INSERT 0 1
--se rappeler de ce qui a été dit sur la visibiité

 SELECT * FROM matable;
 madonnee
```



```
----------
        1
(1 row)
# INSERT INTO matable  SELECT 2*maDonnee FROM matable;
NOTICE:  trig_before [lancé BEFORE] : Il y a 1 lignes dans matabl
NOTICE:  trig_after [lancé AFTER] : Il y a 2 lignes dans matable
INSERT 0 1
--se rappeler de ce qui a été dit sur la visibiité
=#  SELECT * FROM matable;
 madonnee
----------
        1
        2
(2 rows)
# INSERT INTO matable  SELECT maDonnee+3 FROM matable;
NOTICE: trig_before [lancé BEFORE] : Il y a 2 lignes dans matable
NOTICE: trig_before [lancé BEFORE] : Il y a 3 lignes dans matable
NOTICE: trig_after [lancé AFTER] : Il y a 4 lignes dans matable
NOTICE: trig_after [lancé AFTER] : Il y a 4 lignes dans matable
INSERT 0 2
```



```
SELECT * FROM matable;
 madonnee
----------
        1
        2
        4
        5
(4 rows)
```



```
 SELECT * FROM matable;
 madonnee
----------
        1
        2
(2 rows)
#UPDATE maTable set maDonnee=NULL WHERE maDonnee=2;
NOTICE: trig_before [lancé BEFORE] : Il y a 2 lignes dans matable
UPDATE 0
#SELECT * from matable;
 madonnee
----------
        1
        2
(2 rows)
```

```
#UPDATE maTable set maDonnee=6 WHERE maDonnee=2;
NOTICE: trig_before [lancé BEFORE] : Il y a 2 lignes dans matable
NOTICE: trig_after [lancé AFTER] : Il y a 2 lignes dans matable
UPDATE 1
#SELECT * FROM matable;
 madonnee
----------
        1
        6
(2 rows)
```



```
#  SELECT * FROM matable;
 madonnee
----------
        1
        6
(2 rows)
# DELETE FROM maTable;
NOTICE:  trig_before [lancé BEFORE] : Ily a 2 lignes dans matable
NOTICE:  trig_before [lancé BEFORE] : Ily a 1 lignes dans matable
NOTICE:  trig_after [lancé AFTER] : Ily a 0 lignes dans matable
NOTICE:  trig_after [lancé AFTER] : Ily a 0 lignes dans matable
DELETE 2
```


Une contrainte trigger est une contrainte implémentée par un trigger.

```
CREATE CONSTRAINT TRIGGER name
  AFTER events ON relation constraint attributes
  FOR EACH ROW EXECUTE PROCEDURE func ( args )
```

où :

- `name` : nom de la contrainte trigger,
- `events` : les types d'évenement pour lesquels le trigger doit être déclenché,
- `relation` : nom de la table associé aux évenements `events`,
- `constraint` : nom de la contrainte
- `attributes` : attributs de la contrainte
- `func(args)` : procédure trigger à appeler


```
create table classe (
   codeClasse varchar(20) primary key,
   respClasse varchar(30)
);

create or replace function verifResponsable()
   returns trigger as '
   DECLARE
     nb int;
   BEGIN
      SELECT count(*) into nb
      FROM classe where respClasse = NEW.respClasse;
      IF nb>1 THEN
         RAISE EXCEPTION ''étudiant %  est déjà responsable de cl
      END IF;
      return NULL;
   END;
'language 'plpgsql';

create constraint trigger verifResp
```

```
after UPDATE OR INSERT on classe
initially deferred
for each row execute procedure verifResponsable();

INSERT INTO classe values ('L1-MASS','Diba');
INSERT INTO classe values ('L2-MPI','Bira');
INSERT INTO classe values ('L3-INFO','Sira');

SELECT * FROM classe;
 codeclasse | respclasse
------------+------------
 L1-MASS    | Diba
 L2-MPI     | Bira
 L3-INFO    | Sira
(3 rows)

#UPDATE classe SET respClasse='Sira' WHERE codeClasse='L2-MPI';
ERROR:  étudiant Sira  est déjà responsable de classe

#begin;
```



```
BEGIN
# UPDATE classe SET respClasse='Sira' WHERE codeClasse='L2-MPI';
UPDATE 1
# UPDATE classe SET respClasse='Bira' WHERE codeClasse='L3-INFO';
UPDATE 1
# commit;
COMMIT

SELECT * FROM classe;
 codeclasse | respclasse
------------+------------
 L1-MASS    | Diba
 L2-MPI     | Sira
 L3-INFO    | Bira
(3 rows)
```


- Par défault, si une fonction PL/pgSQL produit une erreur, celle-ci est avortée, et la transaction qui l'a exécutée est avortée aussi.
- Il est possible d'intercepter ces erreurs et d'effectuer une reprise sur erreur en utilisant un bloc BEGIN avec une clause EXCEPTION.
- La syntaxe est une extension de la syntaxe normal du bloc BEGIN vers une structure de type try-catch :

```
BEGIN
    -- traitement ...
    EXCEPTION
    WHEN condition [OR condition ] THEN
        -- Reprise erreur i...
    WHEN condition [OR condition ] THEN
        -- Reprise erreur j...
END;
```


Reprise sur erreur

Pas d' Erreur durant le "traitement"

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

Une Erreur est survenu durant le "traitement"

- l'exécution des commandes de traitement est suspendue,
- Les commandes de traitement sont annulées : ROLLBACK,
- Les variables locales de la fonction PL/pgSQL reste comme elles étaient au moment de l'erreur.
- Le controle est ensuite transféré à la clause EXCEPTION,
- La première condition qui correspond à l'erreur est recherchée : l'erreur détectée est commparée successivement aux listes de conditions WHEN.

- Si l'erreur correspond à une condition d'une clause WHEN, les commandes Reprise erreur associées sont exécutées, et le controle est ensuite transféré à la première commande après END.
- Si l'erreur ne correspond à aucune condition des clauses WHEN, l'erreur est propagée vers le bloc parent.
- S'il l'erreur ne peut être traitée alors la fonction est avortée.

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

Reprise sur erreur

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

- les 2 premiers caractères : désigne une classe d'erreurs,
- les 3 derniers caractères : indique l'erreur spécifique à la classe. Une application qui désire savoir quelle erreur elle a généré, doit tester ce code (plutôt que le message textuel).

Quelques exemples :
Class 23 : Integrity Constraint Violation
Class 40 : Transaction Rollback
Class P0 : PL/pgSQL Error

Code Erreur [1]	Signification	Constante
23000	INTEGRITY CONSTRAINT VIOLATION	integrity_constraint_violation
23001	RESTRICT VIOLATION	restrict_violation
23502	NOT NULL VIOLATION	not_null_violation
23503	FOREIGN KEY VIOLATION	foreign_key_violation
23505	UNIQUE VIOLATION	unique_violation
23514	CHECK VIOLATION	check_violation
40000	TRANSACTION ROLLBACK	transaction_rollback
40002 40002	TRANSACTION INTEGRITY CONSTRAINT VIOLATION	transaction_integrity_constraint _violation "
40001	SERIALIZATION FAILURE	serialization_failure
40003	STATEMENT COMPLETION	statement_completion_unknown
40P01	DEADLOCK DETECTED	deadlock_detected
P0000	PLPGSQL ERROR	plpgsql_error
P0001	RAISE EXCEPTION	raise_exception
P0002	NO DATA FOUND	no_data_found
P0003	TOO MANY ROWS	too_many_rows

1. http ://docs.postgresqlfr.org/8.1/errcodes-appendix.html

- Une condition spéciale appelée OTHERS satisfait tout type d'erreur sauf QUERY_CANCELED.
- Dans la clause EXCEPTION, deux variables locales sont accessibles :
  1. SQLSTATE : contient le code d'erreur correspondant à l'exception,
  2. SQLERRM : contient le message d'erreur correspondant à l'exception.
- (Ces variables sont inacessibles à l'extérieur de la clause EXCEPTION.)

Exemple

```
 create table Telephone (
         noTel       text PRIMARY KEY,
         abonne      text
);

CREATE FUNCTION InsererTel( noTel text, abonne text)
  RETURNS integer AS '
  DECLARE
    nbEnreg integer;
  BEGIN
     BEGIN
        INSERT INTO Telephone VALUES (noTel, abonne);
        EXCEPTION
        WHEN UNIQUE_VIOLATION THEN
           RAISE NOTICE ''Clé dupliquée  % pour % ignorée.'',
           noTel, abonne;
     END;
     GET DIAGNOSTICS nbEnreg = ROW_COUNT;
     RETURN nbEnreg;
```



```
  END;
'LANGUAGE 'plpgsql';

#begin;
BEGIN
# select InsererTel( '752082088','Bara');
 insererтel
------------
          1
(1 row)

# select InsererTel( '752082088','Bara');
NOTICE:  Clé dupliquée  752082088 pour Bara ignorée.
 insererтel
------------
          0
(1 row)
# end;
COMMIT
```



```
# select * from telephone;
   notel    | abonne
------------+--------
 752082088 | Bara
(1 row)
```


Erreurs et Messages

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

- où level peut être : DEBUG, LOG, INFO, NOTICE, WARNING ou EXCEPTION.
- Seul EXCEPTION avorte (normalement) la transaction.
- format est une chaine pouvant comporter le symbole de substitution

```
-------------------------  creer_tables.sql  ---------------------
DROP TABLE clients;
DROP TABLE archive_clients;

CREATE TABLE clients (
      client_id         integer primary key,
      nom_client        character varying(50) not null,
      teleph            character(10),
      naissance_date    date,
      solde             decimal(7,2)
);

CREATE TABLE archive_clients(
  client_id       integer,
  nom_client      character varying(50) not null,
  teleph          character(10),
  naissance_date  date,
  solde           decimal(7,2),
  qui_change      varchar,
  quand_change    date,
```



```
  operation       varchar
);

--------------------------------------- function.sql ------------
-- La fonction associee au trigger.

CREATE OR REPLACE FUNCTION archive_clients() RETURNS TRIGGER AS $
  DECLARE
     mindec CONSTANT DECIMAL(7,2) DEFAULT -10000;
  BEGIN
     IF NEW.solde IS NULL THEN
        RAISE EXCEPTION 'solde de % ne peut pas etre null', OLD.n
     END IF;
     IF NEW.solde < mindec THEN
        RAISE NOTICE 'solde de % inferieur a %', OLD.nom_client,
     END IF;
     INSERT INTO archive_clients
        VALUES
       (
         OLD.client_id,      --on met dans l'archive les anciennes
```



```
         OLD.nom_client,
         OLD.teleph,
         OLD.naissance_date,
         OLD.solde,
         CURRENT_USER,      --qui execute INSERT/UPDATE
         now(),             --quand
         TG_OP              --type d'operation
       );
     RETURN NULL;
  END;
$$ LANGUAGE 'plpgsql';
--------------------  creer_trig.sql -------------------------------
-- creation d'un trigger ------------------------------------
DROP TRIGGER IF EXISTS archiver_clients ON clients;

CREATE TRIGGER archiver_clients     --nom du trigger
   AFTER DELETE OR UPDATE           --pour DELETE et UPDATE
   ON clients                       --executer si l'evenement dans la t
   FOR EACH ROW                     --pour chaque ligne
     EXECUTE PROCEDURE archive_clients();
```



```
-------------------------------------------------------------------
INSERT INTO clients VALUES
      (2,'Toto Dupont','0156316267','1956-07-29',55643.32),
      (3,'Modou SY','0556314567','1961-09-29',2361.00)
;

--update clients set solde=334.11 where nom_client like '%Dupont%

--delete from clients where client_id=3;

--update clients set solde=-12000 where nom_client like '%Dupont%
update clients set solde = NULL where nom_client like '%Dupont%';
```