Contraintes d'intégrité

- domaines
- triggers
- PL/SQL

Contraintes d'intégrité

- Les SGBD permettent de gérer plusieurs types de contraintes
- Les contraintes déjà vues :
  - Clé primaire **`Primary Key`**
  - Clé étrangère **`Foreign key`**
  - Clé secondaire **`Unique`**
  - Valeurs non nulles
- Autres types de contraintes :
  - Domaines
  - Contraintes sur les valeurs des attributs
  - Contraintes sur les tuples d'une table
  - Assertions
- Les triggers

Domaines

- Soit la table *Cours(NumC, Semestre, Nbr_inscrits)*
- On veut que l'attribut semestre soit de la forme
  'SEM1_ _' ou 'SEM2_ _' par exemple, le premier semestre de 2001 sera saisi 'SEM101'.

- ```
  CREATE DOMAIN SEM
  CHECK ((VALUE LIKE 'SEM1_ _') OR
              (VALUE LIKE 'SEM2_ _'))
  ```

- ```
  CREATE TABLE Cours(
  NumC Integer,
  Semestre SEM,
  Nbr_inscrits Integer)
  ```

Contraintes sur les attributs(1)

- La contrainte précédente peut être exprimée par la clause **CHECK**

- ```
  CREATE TABLE Cours(
  NumC Integer,
  Semestre CHAR(6) CHECK(
       (Semestre LIKE 'SEM1_ _') OR
       (Semestre LIKE 'SEM2_ _')),
  Nbr_Inscrits Integer
  )
  ```
  ```

Contraintes sur les attributs(2)

- On pourrait penser que ces contraintes peuvent exprimer les clés étrangères
- *R(A,B) S(B,C)*

- ```
  CREATE TABLE R (
  A integer,
  B integer Check ( B IN (Select B FROM S))
  )
  ```

- Cette manière de faire permet de vérifier les insertions dans R mais ne réagit pas lors des suppressions dans S
  ```

Contraintes sur les tuples

- Une contrainte peut porter sur plusieurs attributs d'un même tuple
- *Personne (Nom, Sexe, Age)*
- `('M. Dupont','M',40), ('Mme. Dupont','F',NULL), ('Mlle. Dupont,'F',20)`
- Le nom d'une femme ne doit pas commencer par 'M.'
- ```
  CREATE TABLE Personne(
  Nom VARCHAR(30),
  Sexe CHAR(1) CHECK (Sexe IN ('M', 'F')),
  Age Integer,
  CHECK ((Sexe ='M') OR ( NOM NOT LIKE 'M.%'))
  )
  ```
  ```

Nommer les contraintes

- Il est conseillé de donner des noms aux contraintes pour ensuite pouvoir y faire référence

- ```
  CREATE TABLE T(
  ...
  C INTEGER CONSTRAINT CC CHECK (Version1),
  ...
  )
  ```

- ```
  ALTER TABLE T DROP CONSTRAINT CC;
  ALTER TABLE T ADD CONSTRAINT CC CHECK(Version2)
  ```

Les assertions(1)

- Les assertions sont des contraintes qui peuvent porter sur plusieurs tables.
- Elles doivent être vérifiées par le SGBD à chaque fois qu'une des tables mentionnées est modifiée
- **`CREATE ASSERTION <nom> CHECK (<condition>)`**
- La condition doit être vraie lors de la création, sinon l'assertion n'est pas créée.

Les assertions(2)

- *Cours(NumC, Sem, Nb_inscrits)*
  Inscription(NumEt, NumC, Sem)

- On veut que *Nb_inscrits* reflète exactement le nombre d'inscrits

- ```
  CREATE ASSERTION NB_INSCR CHECK(
      NOT EXISTS(select * from Cours C where
            C.Nb_inscrits != (select COUNT(*)
                        FROM Inscription i
                        WHERE i.NumC=C.NumC AND
                              i.Sem=C.Sem
                        GROUP BY (i.NumC, i.Sem))
                    )
      );
  SET ASSERTION NB_INSCR DEFERRED;
  ```
  ```

Les Triggers

- Assertions
  - Les assertions décrivent des contraintes qui doivent être satisfaites par la base à tout moment.
  - Une assertion est vérifiée par le SGBD à chaque fois qu'une des tables qu'elle mentionne est modifiée
  - Si une assertion est violée, alors la modification est rejetée
- Triggers
  - Les triggers spécifient explicitement à quel moment doivent-ils être vérifiés (i.e. **`INSERT`**, **`DELETE`**, **`UPDATE`**)
  - Les triggers sont des règles ECA : Evénement, Condition, Action
  - Quand E a lieu, si C est vérifiée alors A est exécutée

Triggers – activation

- Un trigger peut être activé **BEFORE/AFTER/INSTEAD OF** un événement d'activation qui peut être une des commandes **INSERT/DELETE/UPDATE** dans une ou plusieurs tables

- L'action peut faire référence à l'ancienne et/ou à la nouvelle valeur des tuples insérés/ supprimés/modifiés par l'action.

```
CREATE TRIGGER exemple1
AFTER UPDATE OF solde ON compte
```

- La condition est exprimée par la clause WHEN

```
WHEN (solde < 0)
```

La clause WHEN peut être n'importe quelle condition booléenne.

Trigger - Action

- Le concepteur a le choix entre spécifier l'action à exécuter soit
  - Une fois, pour chaque tuple modifié/inséré/supprimé, ou
  - Une fois pour tous les tuples modifiés lors d'une seule opération

```
FOR EACH ROW
UPDATE compte
       SET date_débit = SYSDATE
       WHERE Compte.IdC = OldTuple.IdC
```

- L'action peut être n'importe quel code écrit dans le langage associé du SGBD
  - sous PostgreSQL, c'est PL/pgSQL, PL/PHP, PL/Java
  - sous Oracle, c'est PL/SQL, Java, C

Triggers - row level

- Les insertions créent de nouveaux tuples
  - On ne peut faire référence à l'ancienne valeur
- Les modifications font passer de ":old" à ":new"
  - On peut faire référence à ces deux valeurs
- Avec les suppressions, on ne peut faire référence aux nouvelles valeurs.
- Les nouvelles/anciennes valeurs ne peuvent être accédées qu'en mode ligne (Row level).

Exemple

```
CREATE TRIGGER Trig_Inscrits
After INSERT ON INSCRPTION
FOR EACH ROW //pour chaque nouvel inscrit
BEGIN
    Update Cours SET Nb_Inscrits=Nb_Inscrits+1 where
    Cours.Sem=:new.Sem AND Cours.NumC=:new.Numc;
END;
```

Exemple

- On veut garder la trace des opérations effectuées par les utilisateurs

- ```
  CREATE TRIGGER TRIG_INSC
  AFTER INSERT ON Inscrits
  BEGIN
       Insert into tab_logs(Utilisateur, operation, heure) values
  (USER, 'insertion', SYSDATE);
       END;
  ```
  ```

Triggers: Activation

- Activation
  - **BEFORE** – la condition **WHEN** est testée sur l'état avant l'arrivée de l'événement déclencheur
    - Si la condition est vraie, l'action est exécutée, ensuite
    - L'événement déclencheur est réalisé
  - **INSTEAD OF** – l'action est exécutée si la condition est vérifiée,
    - L'événement déclencheur n'est jamais effectué.
  - **AFTER** – l'action est exécutée si la condition **WHEN** est vérifiée après la réalisation de l'événement déclencheur.

Exemples

```
CREATE TRIGGER compter
BEFORE INSERT ON Emp
DECLARE nombre number;
BEGIN
     nombre := select COUNT(*) from EMP;
     dbms_output.put(‘On passera de' || :nombre);
     dbms_output.put(‘a ' ||    :nombre+1);
END;
/
```

Exemples

```
CREATE TRIGGER compte
AFTER INSERT ON Emp
DECLARE nombre number;
BEGIN
     nombre := select COUNT(*) from EMP;
     dbms_output.put(‘On est passé de'|| :nombre-1);
     dbms_output.put(‘a ' || :nombre);
END;
/
```

Exemples

```
CREATE TRIGGER compte
AFTER INSERT ON Emp
WHEN ((select COUNT(*) from EMP)=101)
BEGIN
    dbms_output.put('On vient de passer le cap');
    dbms_output.put('de 100 employés');
END;
/
```

Exemple

- *Emp(Nom, Salaire)*
- Le salaire d'un employé ne peut baisser.
- Cette contrainte ne peut être prise en compte qu'en utilisant les triggers ou bien les procédure stockées.
- **CREATE PROCEDURE mod_sal(in nomE, in salE)**

```
a:=select Salaire from Emp where Nom=:nomE;
si Salaire < salE alors
 update Emp set Salaire = :salE where Nom= :nomE;
```

Exemple

```
CREATE TRIGGER Mod_Sal
AFTER UPDATE OF Emp ON Salaire
FOR EACH ROW
BEGIN
    If new.Salaire < old.Salaire Rollback;
END;
```

Conception des triggers

- Ne pas utiliser les triggers pour gérer des contraintes qui sont facilement gérables par le système. Exemple : prise en compte des contraintes d'unicité de clé.

- Limiter la taille des triggers. S'il vous faut plus de 60 lignes, il est préférable de définir une procédure stockée et l'appeler par le trigger.

- Utiliser les triggers pour les opérations qui ne dépendent pas de l'utilisateur ni de l'application

- **Eviter les triggers récursifs.** E.g, un trigger déclenché lors d'une modification de la table employé et dont l'action va elle aussi modifier cette table

Procédures et fonctions stockées
Le langage PL/pgSQL

- Structure d'un programme PL/pgSQL

```
DECLARE
    declarations
BEGIN
    instructions
END;
```

Procédures et fonctions stockées
Le langage PL/pgSQL (2)

- section **DECLARE** :
  - sert à des déclarer des variables d'un type SQL (**integer**, **varchar**,...).
  - Les paramètres de fonction peuvent ainsi être redéclarés avec un nom en utilisant la syntaxe **ALIAS**
  - Exemple :

```
CREATE FUNCTION prix_reduit(numeric) RETURNS numeric AS
DECLARE
  prix ALIAS FOR $1;
BEGIN
  RETURN prix * 0.75;
END;
```

Procédures et fonctions stockées
Le langage PL/pgSQL (3)

- Une façon très simple est d'utiliser **%TYPE** pour utiliser le type d'un champ de table connu.

- Le suffixe **%ROWTYPE** est utilisé pour déclarer une variable du type "ligne d'une table".

- Le type **record** est utilisé pour contenir n'importe quelle ligne d'une requête.

- Exemple :

```
DECLARE
  madate commandes.date%TYPE;
  unecommande commande%ROWTYPE;
  nimporte RECORD;
BEGIN
  ....
END;
```

Alternatives

- La syntaxe générale est de la forme :

```
IF ... THEN
      ...
ELSIF ... THEN
      ...
ELSIF ... THEN
      ...
ELSE
      ...
END IF ;
```

- Les sections **ELSE** et **ELSIF** sont optionnelles.
- Exemple :

```
DECLARE
        prix ALIAS FOR $1;
BEGIN
        IF prix < 150 THEN
                RETURN 0;
        ELSIF prix < 500 THEN
                RETURN 0.02;
        ELSIF prix < 1500 THEN
                RETURN 0.05;
        ELSE RETURN 0.10;
        END IF;
END;
```

Itérations

- PL/pgSQL offre la possibilité d'écrire des boucles à l'aide d'instructions **LOOP**, **WHILE** ou **FOR**.

  - **LOOP** :
    Syntaxe :

```
LOOP
  instructions
  EXIT WHEN expression
END LOOP ;
```

Itérations

- PL/pgSQL offre la possibilité d'écrire des boucles à l'aide d'instructions **LOOP**, **WHILE** ou **FOR**.

  - **FOR** :
    - Exemples :
      - ```
        FOR i IN 1..10 LOOP
            RAISE NOTICE 'i is %', i;
        END LOOP;
        ```
      - ```
        FOR ligne IN requête LOOP
            instructions
        END LOOP;
        ```

instructions complémentaires

- **`RAISE EXCEPTION`** lance une exception avec un message d'erreur. La fonction s'arrête à cet endroit.
- **`RAISE NOTICE`** envoie un message sur la sortie standard.
- **`SELECT INTO`** permet de stocker le résultat d'un select dans une variable.
- **`FOUND`** est une variable prédéfinie qui vaut vrai si le **`select into`** a trouvé un enregistrement.

Exemple jouet

- On veut archiver toutes les commandes d'un mois donné, qui n'est pas le mois en cours, dans une table d'archive.
  - ```
    CREATE TABLE archivescommandes(
      numcommande INTEGER PRIMARY KEY,
      numclient INTEGER REFERENCES clients,
      montant NUMERIC);
    ```
  - Pour cela on doit:
    - Vérifier que le mois en cours n'est pas le mois pour lequel l'archive est demandée.
    - Calculer les montants de ces commandes
    - Ajouter des éléments pour chacune de ces commandes dans la table prévue à cet effet : archivescommandes
    - Supprimer toutes les lignes de procom et commandes correspondantes.
    ```

Exemple jouet (2)

- Solution :

```
DECLARE
  moisarchive ALIAS FOR $1;
  anneearchive ALIAS FOR $2;
  com commandes%ROWTYPE;
BEGIN
  IF anneearchive = EXTRACT(YEAR FROM now())AND moisarchive = EXTRACT(MONTH FROM now()) THEN
    RAISE EXCEPTION 'Impossible d'archiver le mois en cours';
  ELSE
    FOR com IN SELECT *
                FROM commandes
                WHERE anneearchive = EXTRACT(YEAR FROM date)
                AND moisarchive = EXTRACT(MONTH FROM date)
        LOOP
         INSERT INTO archivescommandes
                VALUES(com.numcommande,com.refclient,com.date,montantcommande(com.numcommande));
         DELETE FROM procom
                WHERE refcommande = com.numcommande;
          DELETE FROM commandes
                WHERE numcommande=com.numcommande;
        END LOOP;
    END IF;
    RETURN;
  END;
```

Autre utilisation des triggers

- Gestion de l'implémentation horizontale avec copie de la spécialisation
  - Exemple : documents dans une bibliothèque (voir le tableau)
- Gestion des informations redandantes
  - Même exemple : voir le tableau.