INFORMATIQUE Développement d'applications
BLOC 1
UE05 Bases de données

Chapitre 4.3 : Intégrité des données

Vincent Reip

Février 2025

Objectif

- Au terme de ce chapitre, l'étudiant sera capable de :
  - comprendre et expliquer les différents types de contraintes d'intégrité pouvant s'exercer sur des données
  - concevoir des contraintes spécifiques et des contraintes générales (trigger) sur base d'un énoncé simple

Les contraintes d'intégrité

- Pour qu'une base de données soit exploitable, ses tables – et les tuples qu'elles contiennent – doivent demeurer dans un état cohérent
- Les données de la base de données doivent donc respecter – à tout moment - des règles qui garantissent cette cohérence : il s'agit des contraintes d'intégrité
- Nous avons déjà vu plusieurs façons d'exprimer des contraintes d'intégrité
  - Lesquelles ?

Les contraintes d'intégrité structurelles

- Les contraintes d'intégrité structurelles
  - Expriment une propriété fondamentale du modèle (liée au modèle relationnel)
  - Ne sont pas susceptibles de changer au cours du temps. Elles sont donc statiques.
  - Sont déclarées lors de la définition de la structure de la table (CREATE TABLE ou ALTER TABLE)
- Exemples :
  - Les contraintes d'unicité de clé (primaire ou candidate)
    - PRIMARY KEY, UNIQUE
  - Les contraintes de domaine (type d'une donnée, plage de valeurs structurellement possibles)
    - INTEGER, CHAR(3), NUMERIC(6,3)
  - Les contraintes référentielles (clés étrangères)
    - FOREIGN KEY xxx REFERENCES yyy
  - Les contraintes de non-nullité
    - NOT NULL

Les contraintes d'intégrité non-structurelles

- Aussi appelées contraintes comportementale ou applicatives
  - Expriment une règle « métier » qui doit être vérifiée à tout moment
  - Peut faire intervenir des données appartenant à plusieurs tables
  - Vérifiées lors de l'insertion, la modification ou la suppression de tuples

- Exemple
  - Il ne peut pas y avoir plus de 20 étudiants par groupe de laboratoire
  - Un client ne peut pas avoir un crédit d'un montant supérieur à 500 €
  - Un candidat ne peut pas émettre plus de 5 offres actives au même moment

Les contraintes d'intégrité non-structurelles

- On distingue deux catégories de contraintes comportementales
  - Les contraintes générales qui portent sur plusieurs tables
  - Les contraintes spécifiques à une table

- Dans la majorité des cas, une contrainte spécifique à une table est exprimable sous forme de contrainte générale

Contraintes spécifiques

```
…
CONSTRAINT nom_contrainte CHECK ( condition )
```

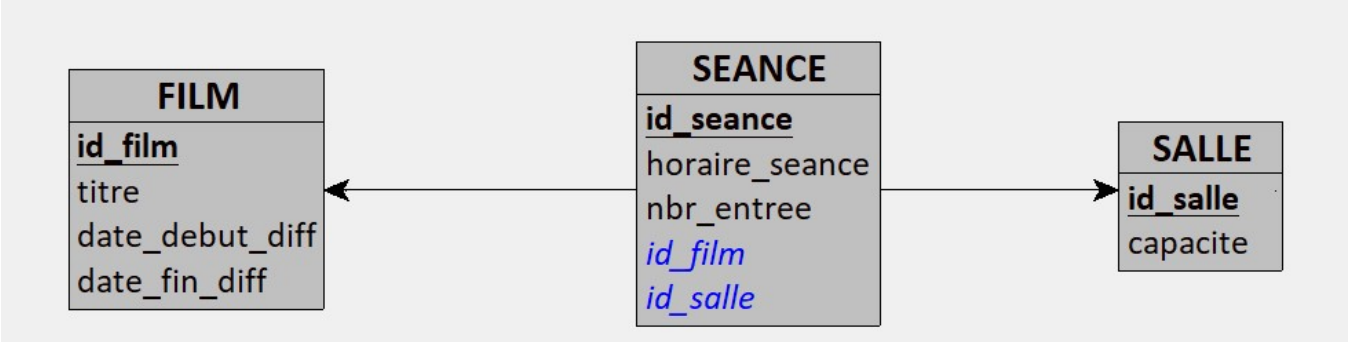
- Concernent les attributs d'une seule et même table
- Appartiennent à la définition d'une table. Si la table disparait, les contraintes spécifiques à cette table disparaissent avec elle.
- Les contraintes spécifiques sont définies
  - au moment de la création d'une table (CREATE TABLE…)
    - CREATE TABLE Salle (
      no_salle INTEGER PRIMARY KEY,
      capacite INTEGER NOT NULL,
      CONSTRAINT capacite_positive CHECK (capacite >0))
  - lors d'une modification du schéma d'une table (ALTER TABLE…)
    - ALTER TABLE Salle
      ADD CONSTRAINT capacite_positive CHECK (capacite >0)

Contraintes spécifiques

- Exemple : soient les tables
  - FILM(id_film, titre, date_debut_diff, date_fin_diff)
  - SEANCE(id_seance, horaire_seance, nbr_entree, id_film, id_salle)
  - SALLE(id_salle, capacite)
- On désire que :
  - la date de début de diffusion d'un film ne soit pas supérieure à sa date de fin de diffusion
  - le nombre d'entrée soit un entier positif ou la valeur null

Contraintes spécifiques

- Exercice :
  - Ajouter une contrainte qui vérifie que le nombre d'entrées vendues pour une séance soit inférieur ou égale à la capacité de la salle
    - Il y a un problème. Lequel ?



Les triggers : introduction

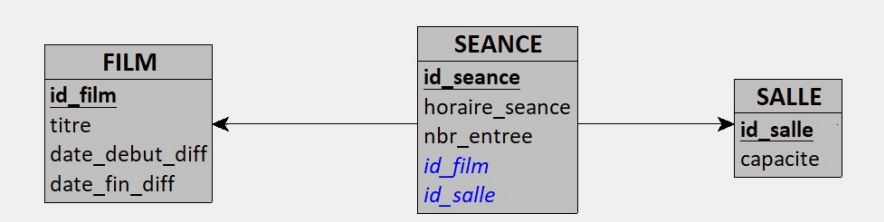
- La définition de contraintes d'intégrité associées à une table présente certaines limites
  - Impossible de spécifier des règles faisant intervenir plusieurs tables
  - Impossible d'effectuer plusieurs requêtes de sélection
  - …
- Les déclencheurs (ou triggers en anglais) permettent notamment de définir des vérifications d'intégrité qui ne souffrent pas de ces limitations

Les triggers : définition

- Un trigger permet de définir un ensemble d'actions qui sont déclenchées automatiquement par le SGBD lorsque certains évènements se produisent
- Les actions définies par un trigger sont stockées dans la base de données. Un trigger constitue une forme de procédure stockée
- Les triggers ont été intégrés dans le langage SQL depuis la version 3 (SQL99). Il est toutefois possible d'observer des variations entre les triggers de différents SGBD
- Nous allons plus particulièrement étudier les triggers sur SQLServer

Contraintes spécifiques : les triggers

- Ajouter une contrainte qui vérifie que le nombre d'entrées vendues pour une séance soit inférieur ou égal à la capacité de la salle



SEANCE

id_seance	horaire_seance	nbr_entree	id_film	id_salle
1	22-02-23 16:00	126	1	1
2	22-02-23 18:00	112	1	1
3	22-02-23 14:00	78	3	2
4	22-02-23 20:00	148	5	1
5	22-02-23 16:00	53	1	3
6	22-02-23 18:00	45	4	3
7	22-02-23 20:00	62	3	2

SALLE

id_salle	capacite
1	150
2	80
3	60

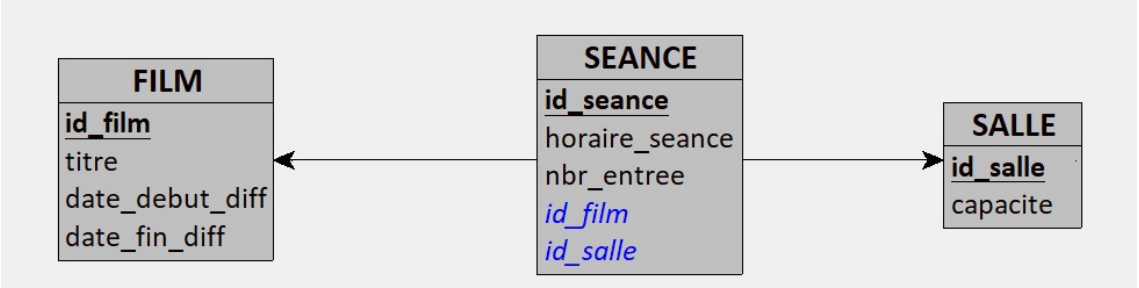
UPDATE SEANCE
SET nbr_entree = nbr_entree + 5
WHERE id_seance = 1

UPDATE SEANCE
SET nbr_entree = nbr_entree + 5
WHERE id_seance = 4

Imaginez un algorithme permettant de procéder à la vérification demandée

Contraintes spécifiques : les triggers

- Ajouter une contrainte qui vérifie que le nombre d'entrées vendues pour une séance soit inférieur ou égal à la capacité de la salle



Stratégie de vérification :

Chaque fois que l'on va modifier la valeur de l'attribut SEANCE.nbr_entree pour une séance particulière, il faudra vérifier qu'elle reste inférieure ou égale à la valeur de l'attribut SALLE.capacite de la salle concernée.

Contraintes spécifiques : les triggers

- Lors de chaque UPDATE sur SEANCE
- Si la valeur de l'attribut nbr_entree a été modifiée
  - Récupérer le tuple mis à jour afin d'obtenir les informations contextuelles nécessaires
    - La nouvelle valeur de nbr_entree que l'on stocke dans une variable @nbr_entree
    - La valeur de id_salle que l'on stocke dans une variable @id_salle
  - Obtenir la capacité de la salle concernée et la stocker dans une variable @capacite
    - SELECT @capacite = capacite FROM SALLE WHERE id_salle = @id_salle
  - Si @nbr_entree > @capacite
    - Annuler l'opération UPDATE

SEANCE

id_seance	horaire_seance	nbr_entree	id_film	id_salle
1	22-02-23 16:00	126	1	1
2	22-02-23 18:00	112	1	1
3	22-02-23 14:00	78	3	2
4	22-02-23 20:00	148	5	1
5	22-02-23 16:00	53	1	3
6	22-02-23 18:00	45	4	3
7	22-02-23 20:00	62	3	2

SALLE

id_salle	capacite
1	150
2	80
3	60

UPDATE SEANCE
SET nbr_entree = nbr_entree + 5
WHERE id_seance = 1

UPDATE SEANCE
SET nbr_entree = nbr_entree + 5
WHERE id_seance = 4

Contraintes spécifiques : les triggers

- Il y a d'autres scénarii à prendre en compte pour atteindre une situation « 100% satisfaisante » :
  - Quid si je change la salle pour une séance ?
    - UPDATE SEANCE SET id_salle = 3 WHERE id_seance = 1
  - Quid si je modifie la capacité d'une salle ?
    - UPDATE SALLE SET capacite = 75 WHERE id_salle = 1
  - Quid si ma requête UPDATE modifie plusieurs tuples ?
    - UPDATE SEANCE SET nbr_entree = nbr_entree + 5 WHERE id_seance IN (1,2,4,6)
  - Quid si j'insère une nouvelle séance ?
    - INSERT INTO SEANCE VALUES (8, CONVERT(smalldatetime, '2023-02-25 16:30:00', 20) , 500, 4,1)

SEANCE

id_seance	horaire_seance	nbr_entree	id_film	id_salle
1	22-02-23 16:00	126	1	1
2	22-02-23 18:00	112	1	1
3	22-02-23 14:00	78	3	2
4	22-02-23 20:00	148	5	1
5	22-02-23 16:00	53	1	3
6	22-02-23 18:00	45	4	3
7	22-02-23 20:00	62	3	2

SALLE

id_salle	capacite
1	150
2	80
3	60

Les triggers : définition

- On nomme parfois les triggers règles E-C-A
- **É**vènement
  - Insertion (INSERT INTO)
  - Mise-à-jour (UPDATE)
  - Suppression d'un tuple (DELETE)
- **C**ondition
  - Qui porte sur l'état de la base de données (éventuellement avant/après l'évènement)
- **A**ction
  - Effectuée par le SGBD si la condition est remplie

Les triggers : en pratique

```
CREATE TRIGGER nom_trigger
ON nom_relation
AFTER  {[INSERT] [,] [UPDATE] [,] [DELETE]}
AS
  [SQL Code]
```

- Cette instruction créera un nouvel objet dans la base de données
- Les évènements déclencheurs (INSERT, UPDATE, DELETE) peuvent être combinés
- [SQL Code] correspond aux conditions et actions liées au déclencheur
  - Conditions : on peut interrompre l'exécution du trigger sous certaines conditions
  - Actions : code Transact-SQL reprenant les différentes actions à effectuer dans le trigger
    - Transact-SQL (T-SQL) est un langage de programmation propriétaire développé par Microsoft qui est utilisé pour interagir avec les bases de données relationnelles.

Les triggers : CONDITION

- Il est possible de vérifier si un / plusieurs attributs ont étés modifiés par le déclencheur et de décider de continuer ou pas l'exécution du trigger.

```
CREATE TRIGGER verifSolde ON COMPTE_COURANT
AFTER UPDATE
AS
BEGIN
   IF NOT UPDATE(solde)
      RETURN
....
```

- La fonction UPDATE() retourne TRUE pour toutes les colonnes si l'événement déclencheur est INSERT et FALSE si c'est DELETE

Les triggers : ACTION

- L'action proprement dite
  - Peut consister en l'application d'une nouvelle requête (ex : pour modifier, insérer ou supprimer des données dans une autre table)
  - Peut faire intervenir un processus plus compliqué
    - Utilisation de curseur : structure de données permettant de gérer le résultat d'une requête ramenant plusieurs tuples
    - Utilisation d'un langage impératif : le PL/SQL sous ORACLE, le Transact/SQL sous MS-SQLServer
      - Affectation, condition, boucle…
    - Vu en détails et exercé lors des laboratoires

Les triggers : coder l'ACTION

- Variables
  - DECLARE @nom VARCHAR(50)
  - DECLARE @dateNaissance SMALLDATETIME
- Affectation
  - SET @nom = 'ROGER'
  - SELECT @nom = nom, @prenom = prenom FROM Etudiant WHERE id=15
- Condition

```
IF (@cote > 8)
BEGIN
    SET @commentaire = 'Très bien !'
END
ELSE
BEGIN
    SET @commentaire = 'Amélioration possible'
END
```

Les triggers : coder l'ACTION

- Boucles

```
WHILE (@i < 10)
  BEGIN
    UPDATE produit
    SET prix = prix + 1
    WHERE numproduit = @i
    @i = @i+1
  END
```

- Requêtes
  - Possibilité d'exécuter des requêtes sur les tables de la BD
    - SELECT : pour retrouver des données nécessaires au traitement
    - INSERT, UPDATE, DELETE : pour modifier l'état de la BD
- Annulation
  - ROLLBACK TRANSACTION (on annule tous les effets de l'événement déclencheur)

Les triggers : coder l'ACTION

- Curseur
  - Le curseur donne la possibilité de parcourir séquentiellement le résultat d'une requête de type SELECT afin de traiter les tuples un par un

```
DECLARE crsrPersonnes CURSOR
FOR SELECT p.nom, p.prenom, r.role
FROM PERSONNE p
JOIN ROLE r ON p.idRole = r.idRole

OPEN crsrPersonnes
FETCH crsrPersonnes INTO @nom, @prenom, @role
WHILE @@fetch_status=0
BEGIN
IF @role = 'Manager'
 BEGIN
      PRINT @prenom + ' ' + @nom + ' est un manager'
 END
 ELSE
 BEGIN
      PRINT @prenom + ' ' + @nom + ' est un employé'
 END
 FETCH crsrPersonnes INTO @nom, @prenom, @role
END
CLOSE crsrPersonnes
DEALLOCATE crsrPersonnes
```

Les curseurs (1/6)

```
DECLARE crsrPersonnes CURSOR
FOR SELECT p.nom, p.prenom, r.role
FROM PERSONNE p
JOIN ROLE r ON p.idRole = r.idRole

OPEN crsrPersonnes
FETCH crsrPersonnes INTO @nom, @prenom, @role
WHILE @@fetch_status=0
BEGIN
IF @role = 'Manager'
  BEGIN
        PRINT @prenom + ' ' + @nom + ' est un manager'
  END
  ELSE
  BEGIN
        PRINT @prenom + ' ' + @nom + ' est un employé'
  END
  FETCH crsrPersonnes INTO @nom, @prenom, @role
END
CLOSE crsrPersonnes
DEALLOCATE crsrPersonnes
```

- Objectif : parcourir l'ensemble des tuples dans la table PERSONNE et faire un affichage spécifique pour chacun des tuples (en fonction de leur rôle)

1) On déclare le curseur : le curseur est un objet nommé que l'on associe à une requête de type SELECT

Les curseurs (2/6)

```
DECLARE crsrPersonnes CURSOR
FOR SELECT p.nom, p.prenom, r.role
FROM PERSONNE p
JOIN ROLE r ON p.idRole = r.idRole

OPEN crsrPersonnes
FETCH crsrPersonnes INTO @nom, @prenom, @role
WHILE @@fetch_status=0
BEGIN
IF @role = 'Manager'
  BEGIN
        PRINT @prenom + ' ' + @nom + ' est un manager'
  END
  ELSE
  BEGIN
        PRINT @prenom + ' ' + @nom + ' est un employé'
  END
  FETCH crsrPersonnes INTO @nom, @prenom, @role
END
CLOSE crsrPersonnes
DEALLOCATE crsrPersonnes
```

2) On « ouvre » le curseur : la requête associée est exécutée et son résultat est stocké en mémoire. Le curseur est prêt à lire la première ligne de ce résultat.

Nom	Prenom	Role
Doe	John	Comptable
Dupont	Roger	Magasinier
Razielg	Léa	Manager

Les curseurs (3/6)

```
DECLARE crsrPersonnes CURSOR
FOR SELECT p.nom, p.prenom, r.role
FROM PERSONNE p
JOIN ROLE r ON p.idRole = r.idRole

OPEN crsrPersonnes
FETCH crsrPersonnes INTO @nom, @prenom, @role
WHILE @@fetch_status=0
BEGIN
IF @role = 'Manager'
  BEGIN
        PRINT @prenom + ' ' + @nom + ' est un manager'
  END
  ELSE
  BEGIN
        PRINT @prenom + ' ' + @nom + ' est un employé'
  END
  FETCH crsrPersonnes INTO @nom, @prenom, @role
END
CLOSE crsrPersonnes
DEALLOCATE crsrPersonnes
```

3a) On lit la ligne courante et on affecte les valeurs aux variables

Nom	Prenom	Role
Doe	John	Comptable
Dupont	Roger	Magasinier
Razielg	Léa	Manager

@nom = ‘Doe’
@prenom = ‘John’
@role = ‘Comptable’

3b) On positionne le curseur sur la ligne suivante

Nom	Prenom	Role
Doe	John	Comptable
Dupont	Roger	Magasinier
Razielg	Léa	Manager

Les curseurs (4/6)

```
DECLARE crsrPersonnes CURSOR
FOR SELECT p.nom, p.prenom, r.role
FROM PERSONNE p
JOIN ROLE r ON p.idRole = r.idRole

OPEN crsrPersonnes
FETCH crsrPersonnes INTO @nom, @prenom, @role
WHILE @@fetch_status=0
BEGIN
IF @role = 'Manager'
  BEGIN
       PRINT @prenom + ' ' + @nom + ' est un manager'
  END
  ELSE
  BEGIN
       PRINT @prenom + ' ' + @nom + ' est un employé'
  END
  FETCH crsrPersonnes INTO @nom, @prenom
END
CLOSE crsrPersonnes
DEALLOCATE crsrPersonnes
```

4) On rentre dans une boucle dont le gardien est une variable système @@fetch_status qui est mise à jour automatiquement après chaque instruction FETCH

- @@fetch_status = 0 si le curseur a pu être positionné sur la ligne suivante
- @@fetch_status = -1 si le curseur n'a pu se positionner sur la ligne suivante
  - Peut être interprété comme un signal que tous les tuples ont étés parcourus et que l'on peut donc sortir de la boucle

La logique du corps de la boucle (hormis la dernière ligne) dépend du traitement à effectuer

Les curseurs (5/6)

```
DECLARE crsrPersonnes CURSOR
FOR SELECT p.nom, p.prenom, r.role
FROM PERSONNE p
JOIN ROLE r ON p.idRole = r.idRole

OPEN crsrPersonnes
FETCH crsrPersonnes INTO @nom, @prenom, @role
WHILE @@fetch_status=0
BEGIN
IF @role = 'Manager'
  BEGIN
        PRINT @prenom + ' ' + @nom + ' est un manager'
  END
  ELSE
  BEGIN
        PRINT @prenom + ' ' + @nom + ' est un employé'
  END
  FETCH crsrPersonnes INTO @nom, @prenom, @role
END
CLOSE crsrPersonnes
DEALLOCATE crsrPersonnes
```

5) Instruction identique à celle de l'étape 3 : on lit la ligne courante, on affecte les valeurs aux variables et on positionne le curseur sur la ligne suivante.

La variable @@fetch_status est mise à jour

/!\ L'oubli de cette instruction mène à une boucle infinie…

Les curseurs (6/6)

```
DECLARE crsrPersonnes CURSOR
FOR SELECT p.nom, p.prenom, r.role
FROM PERSONNE p
JOIN ROLE r ON p.idRole = r.idRole

OPEN crsrPersonnes
FETCH crsrPersonnes INTO @nom, @prenom, @role
WHILE @@fetch_status=0
BEGIN
IF @role = 'Manager'
  BEGIN
        PRINT @prenom + ' ' + @nom + ' est un manager'
  END
  ELSE
  BEGIN
        PRINT @prenom + ' ' + @nom + ' est un employé'
  END
  FETCH crsrPersonnes INTO @nom, @prenom
END
CLOSE crsrPersonnes
DEALLOCATE crsrPersonnes
```

6) On ferme le curseur et on libère les ressources utilisées par le curseur

Les triggers : états AVANT et APRES l'événement

- Un trigger est « attaché » à une (et une seule) table
- L'évènement qui déclenche le trigger est susceptible de modifier la table
- Les triggers (sous SQLServer) s'exécutent APRES que l'évènement déclencheur ait été exécuté avec succès et que toutes les contraintes liées à la table aient pu être vérifiées.
- Il peut être nécessaire de connaître l'état de la table AVANT l'évènement
  - Ex: pour vérifier qu'une augmentation salariale n'est pas supérieure à 10%
- INSERTED et DELETED sont des pseudo-tables de même schéma que la table sur laquelle le trigger est défini et qui contiennent (uniquement) les anciennes (DELETED) et nouvelles (INSERTED) valeurs que l'évènement a affecté.

Les triggers : états AVANT et APRES l'événement

PERSONNE

Id	Nom	Prenom
1	Dupont	Marc
2	Durant	Corinne
3	Craddle	Julius
4	Boli	Ivana

INSERT INTO PERSONNE
VALUES (5, 'Grimard', 'Steve')

PERSONNE

Id	Nom	Prenom
1	Dupont	Marc
2	Durant	Corinne
3	Craddle	Julius
4	Boli	Ivana
5	Grimard	Steve

INSERTED

Id	Nom	Prenom
5	Grimard	Steve

DELETED

Id	Nom	Prenom

PERSONNE

Id	Nom	Prenom
1	Dupont	Marc
2	Durant	Corinne
3	Craddle	Julius
4	Boli	Ivana
5	Grimard	Steve

Rappel : on ne considère ici que les triggers AFTER sous SQLServer

Les triggers : états AVANT et APRES l'événement

PERSONNE

Id	Nom	Prenom
1	Dupont	Marc
2	Durant	Corinne
3	Craddle	Julius
4	Boli	Ivana

DELETE FROM PERSONNE
WHERE Id > 2

PERSONNE

Id	Nom	Prenom
1	Dupont	Marc
2	Durant	Corinne

INSERTED

Id	Nom	Prenom

DELETED

Id	Nom	Prenom
3	Craddle	Julius
4	Boli	Ivana

PERSONNE

Id	Nom	Prenom
1	Dupont	Marc
2	Durant	Corinne

Les triggers : états AVANT et APRES l'événement

PERSONNE

Id	Nom	Prenom
1	Dupont	Marc
2	Durant	Corinne
3	Craddle	Julius
4	Boli	Ivana

```
UPDATE PERSONNE
SET Prenom = 'xxx'
WHERE Id = 2 OR Id = 4
```

PERSONNE

Id	Nom	Prenom
1	Dupont	Marc
2	Durant	xxx
3	Craddle	Julius
4	Boli	xxx

INSERTED

Id	Nom	Prenom
2	Durant	xxx
4	Boli	xxx

DELETED

Id	Nom	Prenom
2	Durant	Corinne
4	Boli	Ivana

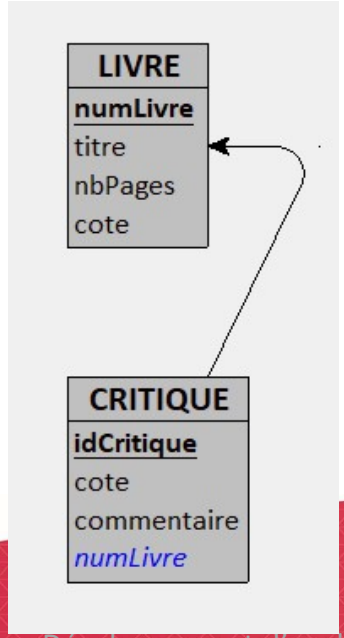
PERSONNE

Id	Nom	Prenom
1	Dupont	Marc
2	Durant	xxx
3	Craddle	Julius
4	Boli	xxx

Les triggers : exemple

Action 1 : La cote d'un livre doit être la moyenne des cotes des critiques pour ce livre.

Action 2 : La cote moyenne d'un livre ne peut pas être négative.




```
CREATE TRIGGER MAJCoteLivre
ON critique
AFTER INSERT
AS
BEGIN

  DECLARE @cotem numeric(2,1)
  DECLARE @numlivre INTEGER

  SELECT @numlivre = numlivre
  FROM INSERTED

  SELECT @cotem = AVG(cote)
  FROM critique
  WHERE numlivre= @numlivre;

  IF (@cotem < 0)
    BEGIN
      RAISERROR('Pas de cote négative !', 12, 1)
      ROLLBACK TRANSACTION
      RETURN
    END
  ELSE
    BEGIN
      UPDATE livre
      SET cote = @cotem
      WHERE numlivre= @numlivre;
    END
END
```

→ Création d'un trigger sur la table 'critique' et qui déclenche sur les insertions

→ Déclaration des variables de travail

→ Récupération du numéro de livre pour lequel une critique a été insérée

→ Calcul de la note moyenne pour ce livre

→ Validation : si la moyenne calculée est négative, on refuse l'insertion

→ Mise à jour de la cote moyenne du livre dans la table 'livre'

Que faire si on modifie la cote d'une critique existante ?

Que faire si on modifie la cote de plusieurs critiques existantes ?

Les triggers : exercices

- On souhaite que l'insertion d'un nouvel acteur ou d'un nouveau réalisateur déclenche l'insertion d'un tuple dans la table personnalités si l'acteur ou le réalisateur ne s'y trouve pas

ACTEUR(idacteur, nom, prenom, sexe, nationalite, regnat)

REALISATEUR(idrealisateur, nom, prenom, sexe, regnat)

PERSONALITE (idpersonalite, nom, prenom, regnat)

Les triggers : exercices

Ecrivez un trigger qui vérifiera qu'un professeur n'a pas plus de 480h d'attributions.