SQL : contraintes, triggers et vues

Dr N. BAME

Compléments sur les contraintes d'intégrité

Contraintes d'intégrité (Rappels)

- *Une contrainte d'intégrité est une **condition logique** qui **doit être satisfaite** par les données stockées dans la BD.*

- **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)
  - La cohérence est liée à la notion de transaction
  - **Déclencher automatiquement** des mises-à-jour entre tables pour maintenir la cohérence globale.

Contraintes d'attributs

- **PRIMARY KEY :**

  désigne un ensemble d'attributs comme la clé primaire de la table

- **FOREIGN KEY :**

  désigne un ensemble d'attributs comme la clé étrangère dans une contrainte référentielle

- **NOT NULL :**

  spécifie qu'un attribut ne peut avoir de valeurs nulles

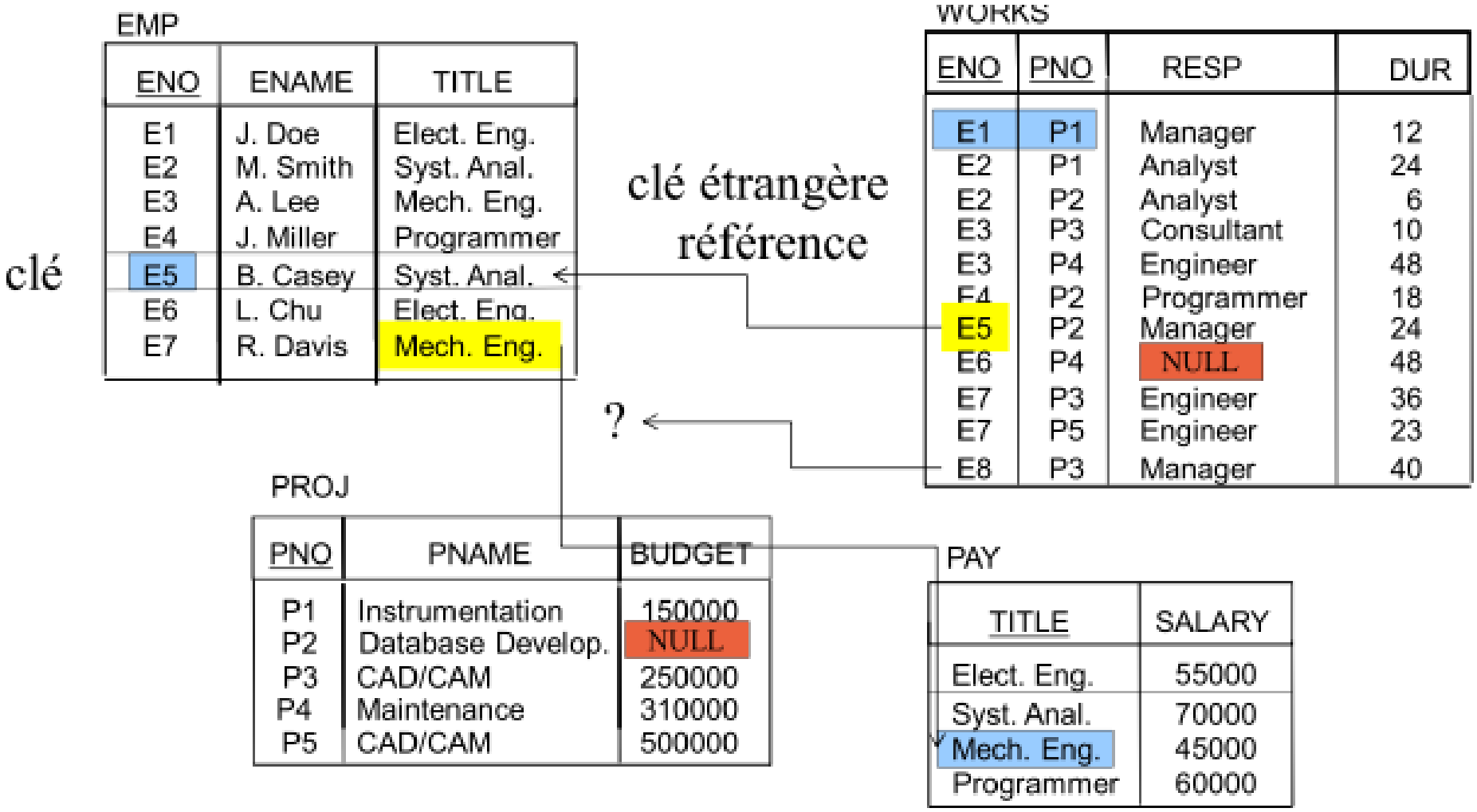
- **UNIQUE :**

  spécifie un ensemble d'attributs dont les valeurs doivent être distinctes pour chaque couple de n-uplets.

- **CHECK :**

  spécifie une **condition** que les colonnes de chaque ligne devront vérifier. On peut ainsi indiquer des contraintes d'intégrité de **domaines.**

Exemple


EMP
ENO
ENAME
TITLE
E1
J. Doe
Elect. Eng.
E2
M. Smith
Syst. Anal.
E3
A. Lee
Mech. Eng.
E4
J. Miller
Programmer
E5
B. Casey
Syst. Anal.
E6
L. Chu
Elect. Eng.
E7
R. Davis
Mech. Eng.
clé
clé étrangère
référence
?
WORKS
ENO
PNO
RESP
DUR
E1
P1
Manager
12
E2
P1
Analyst
24
E2
P2
Analyst
6
E3
P3
Consultant
10
E3
P4
Engineer
48
E4
P2
Programmer
18
E5
P2
Manager
24
E6
P4
NULL
48
E7
P3
Engineer
36
E7
P5
Engineer
23
E8
P3
Manager
40
PROJ
PNO
PNAME
BUDGET
P1
Instrumentation
150000
P2
Database Develop.
NULL
P3
CAD/CAM
250000
P4
Maintenance
310000
P5
CAD/CAM
500000
PAY
TITLE
SALARY
Elect. Eng.
55000
Syst. Anal.
70000
Mech. Eng.
45000
Programmer
60000

Maintenance automatique de l'intégrité référentielle

La suppression (**ON DELETE**) ou la mise-à-jour (**ON UPDATE**) d'un n-uplet référencé (de clé primaire) nécessite une action sur le n-uplet avec la clé étrangère :

- **RESTRICT** : l'opération est rejetée (par défaut)
- **CASCADE** : supprime ou modifie tous les n-uplets avec la clé étrangère si le n-uplet référencé est supprimé ou sa clé est modifiée
- **SET [NULL | DEFAULT]** : mettre à NULL ou à la valeur par défaut quand le n-uplet référencé est effacée/sa clé est modifiée.

Pas de *on update cascade* en Oracle
=> trigger

Maintenance automatique de l'intégrité référentielle

- L'option « **ON DELETE CASCADE** » indique que la suppression d'une ligne de ***tableref*** va entraîner automatiquement la suppression des lignes qui la référencent dans la table. Si cette option n'est pas indiquée, il est impossible de supprimer des lignes de ***tableref*** qui sont référencées par des lignes de la table.

- A la place de «**ON DELETE CASCADE**» on peut donner l'option «**ON DELETE SET NULL**». Dans ce cas, la clé étrangère sera mise à **NULL** si la ligne qu'elle référence dans *tableref* est supprimée.

- Ils existe d'autres options

  ON DELETE SET DEFAULT met une valeur par défaut dans la clé étrangère quand la clé primaire référencée est supprimée.

  ON UPDATE CASCADE modifie la clé étrangère si on modifie la clé primaire (ce qui est à éviter).

  ON UPDATE SET NULL met NULL dans la clé étrangère quand la clé primaire référencée est modifie.

  ON UPDATE SET DEFAULT met une valeur par défaut dans la clé étrangère quand la clé primaire référencée est modifiée.

Exemple

Contraintes de n-uplets : CHECK

Contraintes portant sur une seule table.

(d'autres tables peuvent apparaître dans des sous-requêtes)

- La condition est vérifiée chaque fois qu'un n-uplet est inséré ou modifié dans la table; la mise-à jour (transaction) est rejetée si la condition est fausse.

```
CREATE TABLE Works
(Eno     CHAR(3),
Pno      CHAR(3),
Resp     CHAR(15),
Dur      INT,
PRIMARY KEY (Eno, Pno),
FOREIGN KEY (Eno) REFERENCES Emp(Eno) ON DELETE RESTRICT ON  UPDATE CASCADE,
FOREIGN KEY (Pno) REFERENCES Project(Pno) ON DELETE SET DEFAULT ON UPDATE CASCADE,

CHECK (NOT(PNO<'P5') OR Dur>18));
```

Déclencheurs (Triggers)

Triggers

Définition

- Un trigger est un **programme** qui se ***déclenche automatiquement*** suite à un ***événement***
- Ces triggers **font partie** du schéma de la base.
- Leur **code compilé** est **conservé** (comme pour les programmes stockés)

Pourquoi ?

- **vérification** de certaines **contraintes**
- **lancement** de certains **traitements**

Triggers : utilisation

«**Règles actives**» généralisant les contraintes d'intégrité :

- **génération automatique** de valeurs manquantes
- éviter des **modifications invalides**
- implantation de **règles applicatives**
- **génération de traces** d'exécution, **statistiques**, ...
- **maintenance** de répliques
- propagation de **mises-à-jour** sur des vues vers les tables
- intégrité référentielle entre des données distribuées
- interception d'événements utilisateur / système (LOGIN, STARTUP, …)

Trigger ou règle active ou règle ECA(1)

- **Définition ECA** :
  - ***Événement (E) :***

    une **mise-à-jour** de la BD qui **active le trigger**

    ex.: réservation de place
  - ***Condition (C):***

    un test ou une requête devant être vérifié lorsque le trigger est activé (une requêtes est vraie si sa réponse n'est pas vide)

    ex.: nombre de places disponibles ?
  - ***Action (A):***

    une procédure exécutée lorsque le trigger est activé et la condition est vraie : E,C → A

    ex.: annulation de réservation

Trigger ou règle active ou règle ECA(2)

- Un déclencheur ou **une règle ECA** est de la forme:

  «Quand un **Evénement** se **produit**, **si** une **Condition** est **satisfaite**, une **Action** est **exécutée**.»

- **Exemple**:

  Quand un **employé est évalué**, **si l'évaluation est > 70**, lui **attribuer une augmentation de salaire de 10%**.

Syntaxe

```
CREATE TRIGGER triggerName
BEFORE | AFTER
INSERT | DELETE | UPDATE ON tableName
[REFERENCING variables]
[FOR EACH ROW]
[WHEN  condition]
action
```

événement

condition

action

Syntaxe : VARIABLES

- **OLD ROW** : pour l'ancienne ligne (**delete**, **update**, **for each row**)
- **NEW ROW** : pour la nouvelle ligne (**insert**, **update**, **for each row**)
- **OLD TABLE** : Pour l'ancienne table (**delete**, **update**)
- **NEW TABLE** : Pour la nouvelle table (**insert**, **update**)

Syntaxe

REFERENCING [NEW VAR as nD] [OLD VAR as oD]

Syntaxe : FOR EACH ROW

- Quand «**FOR EACH ROW**» **est spécifié**, le déclencheur **est exécuté pour chaque ligne** insérée, modifiée ou supprimée par le déclencheur

  ⇒Déclencheur **par lignes** (**row-level**)

- Quand «**FOR EACH ROW**» **n'est pas spécifié**, le déclencheur est exécuté une seule fois

  ⇒Déclencheur **par opération** (**statement-level**)

Exécution des triggers (1)

- **Moment de déclenchement** du trigger par rapport à l'événement *E* (maj. activante) :
  - ***avant*** (BEFORE) *E*
  - ***après*** (AFTER) *E*
  - ***à la place*** de (INSTEAD OF) *E* **(spécifique aux vues)**
- **Nombre d'exécutions** de l'action **A** par déclenchement :
  - une exécution de l'action A **par n-uplet** modifié (**ROW TRIGGER**)
  - une exécution de l'action A **par événement** (**STATEMENT TRIGGER**)

Exécution des triggers (2)

Les données considérées par le trigger :

- ***:old*** avant l'événement, : ***new*** après l'événement (peuvent être renommés dans la section **REFERENCING**)
- ***for each row*** *:* un n-uplet**,**
- ***for each sentence*** : un ensemble de n-uplets
- ***:new*** peut être modifié par l'action, mais effet seulement si ***before***
- Pour agir avec un trigger ***after***, il faut modifier directement la base
- ***:old*** (resp. :***new***) n'a pas de sens pour **insert** (resp. ***delete***)

Exemple : mise à jour en cascade

```
AFTER UPDATE ON ETUDIANT
REFERENCING OLD ROW AS old, NEW ROW as new
FOR EACH ROW
UPDATE NOTE SET idEtudiant = new.idEtudiant
WHERE idEtudiant = old.idEtudiant;
```

- NOTE est mise à jour s'il y a une référence entre ETUDIANT et NOTE.
  - Ceci requiert de **ne pas avoir de contraintes d'intégrité référentielle** «RESTRICT» ou «SET NULL» entre NOTE et ETUDIANT.

Plusieurs déclencheurs

- Les **implémentations diffèrent** d'un système à un autre => Il n'y a pas deux systèmes avec le même standard SQL
- MySQL n'autorise qu'un seul déclencheur par opération (INSERT, DELETE, UPDATE) par table, d'autres systèmes autorisent plusieurs => Comportement différent avec résultats différents
- Possibilité d'avoir plusieurs déclencheurs qui s'exécutent à la fois => Différents résultats, des cycles d'exécution infinies, des déclencheurs qui se déclenchent eux-mêmes, des actions qui déclenchent des déclencheurs...
- Faire attention à l'interaction avec les contraintes

Exemples

Contrôle d'intégrité

Emp (Eno, Ename, Title, City)

- Vérification de la contrainte de clé à l'insertion d'un nouvel employé :

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

Contrôle d'intégrité

- **Emp** (Eno, Ename, #Title, City) **Pay**(Title, Salary)

- Suppression d'un titre et des employés correspondants (« ON DELETE CASCADE ») :

```
CREATE TRIGGER DeleteTitle
BEFORE DELETE ON Pay
REFERENCING OLD AS O
FOR EACH ROW
BEGIN
    DELETE FROM Emp WHERE Title=O.Title
END;
```

Mise-à-jour automatique

Emp (Eno, Ename, Title, City)

- Création automatique d'une valeur de clé (autoincrément) :

```
CREATE TRIGGER SetEmpKey
BEFORE INSERT ON Emp
REFERENCING NEW AS N
    FOR EACH ROW
    BEGIN
        N.Eno := SELECT max(Eno)+1 FROM Emp
    END;
```

Mise-à-jour automatique

- **Pay**(Title, Salary, Raise)

- Maintenance des augmentations (**raise**) de salaire :

```
CREATE TRIGGER UpdateRaise
AFTER UPDATE OF Salary ON Pay
REFERENCING OLD AS O, NEW AS N
FOR EACH ROW
BEGIN
    UPDATE Pay
    SET Raise = N.Salary - O.Salary
    WHERE Title = N.Title;
END;
```

Les triggers sous PostgreSQL

- La création d'un trigger sous PostGreSQL se fait en deux étapes :
  - **Définition de la procédure ou fonction** (action) **à exécuter** par le trigger
  - **Définition du trigger** avec **appel de la fonction d'action**

- **Définition de la procédure**
  - Langage : **PL/pgSQL**
  - Fonction : **sans argument**
  - **Type** de **retour : trigger**
  - Des variables locales spéciales, nommées **TG_*quelquechose*** sont automatiquement définies pour décrire la condition qui a déclenché l'appel.

Syntaxe

1. Définition de la **procédure ou fonction (d'action)**

```
CREATE OR REPLACE FUNCTION trigger_function()
  RETURNS trigger AS
$$
    begin
        ....
    end;
$$language plpgsql;
```

2. Définition du **trigger** avec appel de la fonction d'action

Syntaxe

1. Définition de la **procédure ou fonction (d'action)**

```
CREATE OR REPLACE FUNCTION trigger_function()
  RETURNS trigger AS
$$
        begin
                ....
        end;
$$language plpgsql;
```

2. Définition du **trigger** avec appel de la fonction d'action

CREATE TRIGGER trigger_name

{BEFORE | AFTER | INSTEAD OF} {event [OR ...]}

ON table_name

[FOR [EACH] {ROW | STATEMENT}]

EXECUTE PROCEDURE trigger_function()

variables locales spéciales

- Plusieurs variables spéciales sont créées automatiquement dans le bloc de plus haut niveau.
  - **NEW** : Type de données RECORD ; variable contenant **la nouvelle ligne** de base de données pour les opérations **INSERT** / **UPDATE** dans les triggers de niveau ligne.
  - **OLD** : Type de données RECORD ; variable contenant **l'ancienne ligne** de base de données pour les opérations **UPDATE/DELETE** dans les triggers de niveau ligne.
  - **TG_NAME :** variable qui contient le **nom du trigger** réellement lancé.
  - **TG_WHEN** : une chaîne, soit **BEFORE** soit **AFTER**, soit **INSTEAD OF** selon la définition du trigger.

variables locales spéciales

- **TG_LEVEL :** une chaîne, soit **ROW** soit **STATEMENT**, selon la définition du trigger.
- **TG_OP :** une chaîne, **INSERT**, **UPDATE**, **DELETE** ou **TRUNCATE** indiquant pour quelle opération le trigger a été lancé.
- **TG_RELID :** l'ID de l'objet de la table qui a causé le déclenchement du trigger.
- **TG_RELNAME :** le **nom de la table** qui a causé le déclenchement. (TG_TABLE_NAME).
- **TG_TABLE_NAME :** le **nom de la table** qui a déclenché le trigger.
- **TG_TABLE_SCHEMA** : le nom du schéma de la table qui a appelé le trigger.
- **TG_NARGS** : le nombre d'arguments donnés à la procédure trigger dans l'instruction **CREATE TRIGGER**.
- **TG_ARGV[] :** les arguments de l'instruction **CREATE TRIGGER**. L'index débute à 0. Les indices invalides (inférieurs à 0 ou supérieurs ou égaux à tg_nargs) auront une valeur NULL.

Une fonction trigger doit renvoyer soit NULL soit une valeur record ayant exactement la structure de la table pour laquelle le trigger a été lancé.

Exemple

1. Définition de la **procédure ou fonction (d'action)**

```
CREATE OR REPLACE FUNCTION log_lname_changes()
  RETURNS trigger AS
$$
    BEGIN
        IF NEW.last_name <> OLD.last_name THEN
           INSERT INTO employee_audits(employee_id,last_name,changed_on)
           VALUES(OLD.id, OLD.last_name, now());
        END IF;
         RETURN NEW;
    END;
$$language plpgsql;
```

Exemple

2. Définition du **trigger** avec appel de la fonction d'action

```
CREATE TRIGGER last_name_changes
  BEFORE UPDATE
  ON employees
  FOR EACH ROW
  EXECUTE PROCEDURE log_lname_changes();
```

Vues

Pourquoi définir des vues ?

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

1. Les **requêtes** sont **complexes** :
   - **difficiles** à **formuler**
   - ne portent **que sur un sous-ensemble** des attributs
   - **source d'erreurs**

2. Une modification du schéma nécessite la **modification** de **beaucoup de programmes** qui utilisent la base de données.

Solution : ***Adapter*** **le schéma** et les **données** à des **applications spécifiques** → **vues**

Définition d'une vue

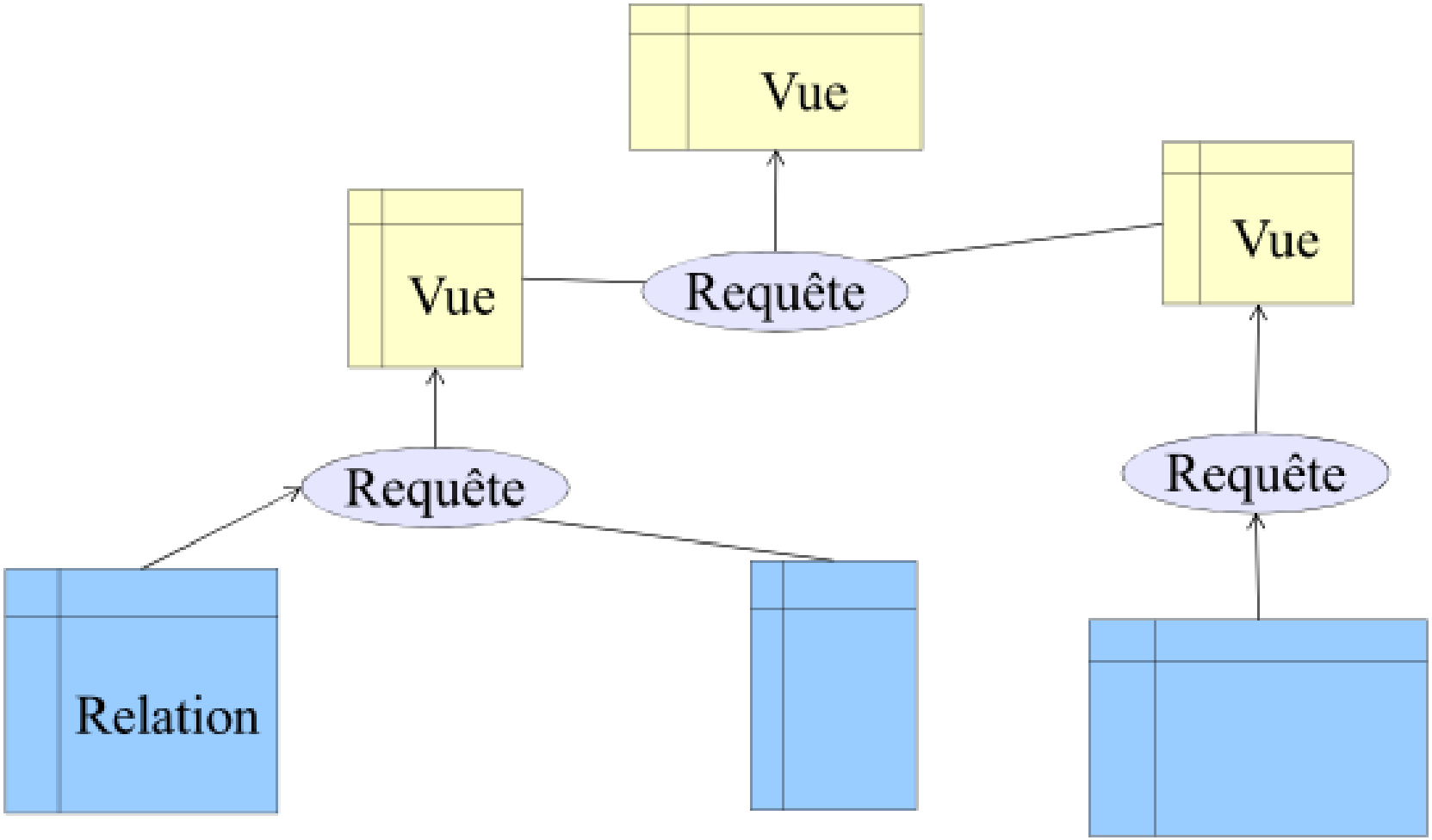
- **Définition** : Une vue **$V(a_1, a_2, \ldots a_n)$** est une relation avec *n* attributs qui **contient le résultat d'une requête $Q(a_1, a_2, \ldots a_n)$** évaluée sur une base de données BD :

$$V(a_1, a_2, \ldots a_n) := Q(x_1, x_2, \ldots, x_n, BD)$$

Remarques :

- V possède un schéma relationnel avec des attributs $a_1, \ldots a_n$.
- V reflète l'état actuel d'une base de données BD
- V peut être interrogée et il est possible de définir des vues à partir d'autres vues.
- On distingue les relations « matérialisées » (tables) et les relations « virtuelles » (vues)

Relations et Vues



Syntaxe

```
CREATE VIEW nom_vue [(att1, att2...)]
AS requête_SQL  [ WITH CHECK OPTION ]
```

- ***nom_vue*** désigne le nom de la relation
- ***att1, ...*** (optionnel) permet de nommer les attributs de la vue (attributs de la requête par défaut)
- ***requête_SQL*** désigne une requête SQL standard qui définit le « contenu » (instance) de la vue
- **WITH CHECK OPTION** (voir mises-à-jour de vues)

Exemple

Emp (Eno, Ename, Title, City)

Project(Pno, Pname, Budget, City)

Pay(Title, Salary)

Works(Eno, Pno, Resp, Dur)

- Définition de la vue EmpProjetsParis des employés travaillant dans des projets à Paris :

```
CREATE VIEW EmpProjetsParis(NumE, NomE, NumP, NomP, Dur)
AS SELECT Emp.Eno, Ename, Works.Pno, Pname, Dur
      FROM Emp, Works, Project
      WHERE Emp.Eno=Works.Eno
      AND Works.Pno = Project.Pno
      AND Project.City = 'Paris'
```

Interrogation de vues

Emp (Eno, Ename, Title, City)

Project(Pno, Pname, Budget, City)

Pay(Title, Salary)

Works(Eno, Pno, Resp, Dur)

- Les noms des employés de projets Parisiens :

Requête sans vue:

```
SELECT Ename
FROM Emp, Works, Project
WHERE Emp.Eno=Works.Eno
AND Works.Pno = Project.Pno
AND Project.City = 'Paris'
```

Requête avec vue:

```
SELECT NomE
FROM EmpProjetsParis
```

On obtient le même résultat

Évaluation de requêtes sur des vues

Vue :

```
CREATE VIEW EmpProjetsParis
AS SELECT Emp.Eno, Ename, Project.Pno,
Pname, Dur
FROM Emp, Works, Project
WHERE Emp.Eno=Works.Eno
AND Works.Pno = Project.Pno
AND Project.City = 'Paris'
```

Requête :

```
SELECT Emp.Eno FROM EmpProjetsParis
WHERE Dur > 3
```

Mise-à-jour de vues

Problème de mise-à-jour : 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, …).

Exemple:

CREATE VIEW V AS SELECT A,C

FROM R,S WHERE R.B = S.B

- Insertion d'un n-uplet [A:1,C:3] dans la vue V
- Quelle est la modification à faire dans R et S (valeur de B) ?

Vues modifiables

Une vue n'est pas modifiable :

- quand elle ne contient pas tous les attributs définis comme **NON NULL** dans la table interrogée
- quand elle contient une **jointure**
- quand elle contient une **fonction agrégat**

- 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**.

Mises-à-jour

Emp (Eno, Ename, Title, City)

Pay(Title, Salary)

Project(Pno, Pname, Budget, City)

Works(Eno, Pno, Resp, Dur)

```
CREATE VIEW ProjetParis
AS SELECT Pno,Pname,Budget
FROM Project
WHERE City='Paris';

UPDATE ProjetParis
SET Budget = Budget*1.2;
```

WITH CHECK OPTION

- **WITH CHECK OPTION** protège contre les « disparitions de n-uplets » causées par des mise-à-jour :
  - pour s'assurer que les données satisfont les conditions de définition de la vue (les nouvelles données sont visibles au travers de la vue).

```
CREATE VIEW ProjetParis
AS SELECT Pno,Pname,Budget,City
FROM Project
WHERE City='Paris'
WITH CHECK OPTION
;
```

```
UPDATE ProjetParis
SET City = 'Lyon'
WHERE Pno=142;
```

← Mise-à-jour rejetée

Vues et tables

Similitudes :

- Interrogation SQL
- UPDATE, INSERT et DELETE sur vues modifiables
- Autorisations d'accès
- Evaluation et optimisation

Différences:

- On ne peut pas créer des index sur les vues
- On ne peut pas définir des contraintes (clés)
- Une vue est recalculée à chaque fois qu'on l'interroge
- *Vue matérialisée :* stocker temporairement la *vue* pour améliorer les performances. => pb de performance si les tables sont mises à jour