Licence Ingénierie des Systèmes d'Information et des Logiciels (ISIL)

Bases de Données Avancées

http://www.larbiguezouli.com

Présenté par Dr Larbi GUEZOULI

Bases de Données Avancées

1. Contraintes d'intégrité, Triggers et Vues [? Cours + ? TD]
2. Conception et optimisation de schéma relationnel [? Cours + ? TD]
3. Evaluation et optimisation de requête [? Cours + ? TD]
4. Les transactions [? Cours + ? TD]
5. Introduction aux Entrepôts de données [? Cours + ? TD]

Mode d'évaluation :
Interrogations - Travaux pratiques - Examen final

Contraintes d'intégrité, Triggers et Vues

Plan

1. Contraintes d'intégrité
2. Triggers (Déclencheurs)
3. Les vues

Contraintes d'intégrité, Triggers et Vues

1. Contraintes d'intégrité

Une contrainte d'intégrité est une règle qui permet de garantir la cohérence des données lors des mises à jour de la base.

Exemple: La date de naissance d'une personne est une date postérieure de la date actuelle!

1.1. Typologie

- **Contraintes structurelles**:
  - **Contraintes de domaine**: Les valeurs d'un attribut d'une table doivent appartenir à un domaine prédéfini. Exp. La date de naissance est de type Date.
  - **Contraintes d'unicité de clé**: Chaque relation possède au moins une clé avec des valeurs non nulles.

1.1. Typologie (suite)

- **Contraintes référentielles**: Inter-relations, chaque valeur d'une **clé étrangère** doit correspondre à une valeur **existante** de la clé dans la table référencée. Exp. L'ensemble des étudiants doit être **inclus** dans l'ensemble des personnes.

- **Contraintes comportementales**: Contraintes à vérifier lors des mises à jour.
  - **Les dépendances fonctionnelles**: Un attribut (ou groupe d'attributs) **Y** de la relation **R** dépend fonctionnellement d'un attribut (ou groupe d'attributs) **X**, si, pour toute valeur de **X**, il lui correspond une seule valeur de **Y**. On note **X→ Y**.

  Exp. MatriculeEtd → Nom, Prénom.

1.1. Typologie (suite)

- **Les dépendances d'inclusion**: Les valeurs d'un groupe de colonnes d'une table doivent rester incluses dans celles d'un groupe de colonnes d'une autre table. Exp. Les valeurs de la colonne idProd de la table Facture doivent être incluses dans l'ensemble des valeurs de la colonne idProduit de la table Produit.
- **Les contraintes équationnelles**: Comparer deux expressions arithmétiques calculées à partir de données de la base. Exp. La valeur du stock est égale à la somme des quantités achetées moins la somme des quantités vendues.

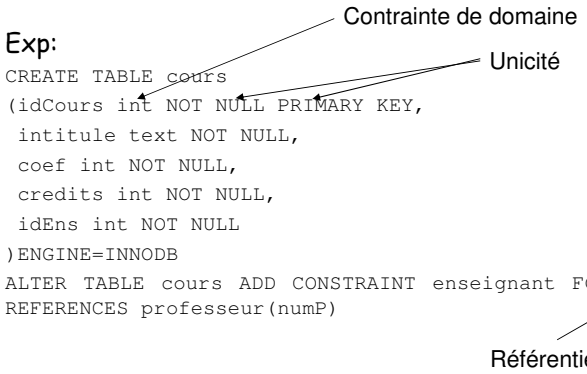
1.1. Typologie (suite)

En SQL:

```
CREATE TABLE <nom de table>
({<Attribut> <Domaine> [<CONTRAINTE D'ATTRIBUT>]}+)
[<CONTRAINTE DE RELATION>]
<CONTRAINTE D'ATTRIBUT> ::=
NOT NULL |
UNIQUE | PRIMARY KEY
```


1.1. Typologie (suite)

Exp:



Contraintes d'intégrité, Triggers et Vues

1.1. Typologie (suite)

Exp: (suite)

Dépendances fonctionnelles: idCours → intitule

Dépendances d'inclusion: Les valeurs de la colonne 'idEns' de la table 'cours' doivent rester incluses dans celles de la colonne 'numP' de la table 'professeur' (Clé étrangère)

Contraintes équationnelles:

```
ALTER TABLE cours ADD CONSTRAINT "PRJ_CHK1" CHECK (coef <= credits);
```


Contraintes d'intégrité, Triggers et Vues

1.2. Vérification (Contrôle) des C.I.

- Détection d'incohérence: Si on retrouve des tuples ne satisfaisant pas une contrainte dans la base après une mise à jour, on doit pouvoir rejeter la mise à jour.
- Post-test: Après une mise à jour appliquer un test pour déterminer si une contrainte d'intégrité est violée.

```
(SELECT COUNT(*)
 FROM R
 WHERE NOT (CI)) = 0
```


Contraintes d'intégrité, Triggers et Vues

1.2. Vérification (Contrôle) des C.I. (suite)

- Pré-test: Avant une mise à jour, appliquer un test pour déterminer si une contrainte d'intégrité est violée.

```
UPDATE cours
SET credit = credit+1
WHERE credit < 6
```

1. Vérification de l'unicité de clé: Tout SGBD applique un pré-test avant la mise à jour pour vérifier que la nouvelle clé ne figure pas dans la table.

Contraintes d'intégrité, Triggers et Vues

1.2. Vérification (Contrôle) des C.I. (suite)

2. Vérification des contraintes de domaine: Pour les contraintes de type `CHECK <condition>` le SGBD applique un pré-test sur la condition avant la mise à jour.
3. Vérification des contraintes référentielles lors de:
   - Insertion dans la table dépendante
   - Mise à jour de la colonne référençante dans la table dépendante
   - Suppression dans la table maître
   - Modification de clé dans la table maître

2. Triggers (Déclencheurs)

2.1. Définition (événement, condition et action)

Un déclencheur (en anglais Trigger) est une action sur la base de données (ensemble d'instructions) qui sera déclenchée par un évènement. Le déclencheur est généralement exprimé sous forme d'un triplet (Evènement, Condition, Action) comme suit:

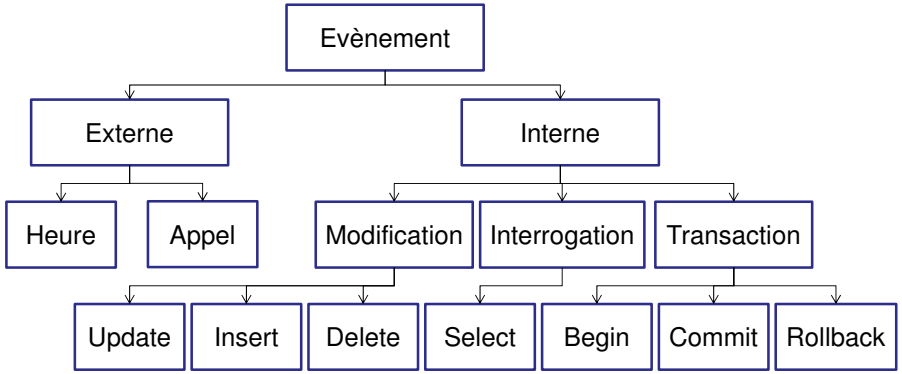
Quand <**Evènement**> Si <**Condition** sur BD> Alors <**Action** sur BD>

2.1.1. Les événements

Il existe plusieurs types d'évènements:

- Début (BEFORE) ou fin (AFTER) d'une recherche (SELECT), d'une mise à jour (UPDATE, DELETE, INSERT) ou d'une transaction (BEGIN, COMMIT, ROLLBACK)
- Ecoulement d'un délai;
- Passage d'une horodate;
- etc.

2.1.1. Les événements (suite)



2.1.1. Les événements (suite)

Evènements externes:

```
AT TIMES <Heure>
IN TIMES <Delta>
BEFORE/AFTER <Procédure>
```

Evènements internes:

```
BEFORE/AFTER UPDATE
BEFORE/AFTER INSERT
BEFORE/AFTER DELETE
BEFORE/AFTER SELECT
BEGIN, COMMIT, ABORT
```

2.1.2. Les conditions

La condition est une expression logique portant sur les variables de contexte d'exécution ou/et sur la base de données.

Exp:

On peut utiliser `EXIST`, `NOT EXIST` ou `COUNT`.

2.1.3. Les actions

L'action du déclencheur est un ensemble de requêtes à exécuter lorsque l'évènement est déclenché et la condition est satisfaite.

2.1.4. Expression en SQL

Pour créer un trigger:

```
CREATE TRIGGER [OR REPLACE] nom_trigger
moment_trigger evenement_trigger
ON nom_table [FOR EACH ROW]
[WHEN condition]
BEGIN
corps_trigger;
END
```

Remarque: MySQL n'accepte pas la clause WHEN. Donc il faut mettre la condition dans le corps du trigger.

2.1.4. Expression en SQL (suite)

`moment_trigger`: BEFORE/AFTER

`evenement_trigger`: <evt> {OR <evt>}

<evt>: DELETE/INSERT/UPDATE [OF {col1,…}]

`nom_table`: Table concernée

`corps_trigger`: Requêtes à exécuter lorsque le trigger est déclenché.

2.1.4. Expression en SQL (suite)

Pour **supprimer** un trigger:

```
DROP TRIGGER nom_trigger;
```

Pour **lister** les triggers définis:

```
SHOW TRIGGERS;
```


2.1.4. Expression en SQL (suite)

Exp:

Soit la table VOL qui contient des informations des vols d'une compagnie:

VOL(id, villeDepart, villeDest, dateDepart, dateArrivee)

```
CREATE TABLE vol
(id int NOT NULL PRIMARY KEY,
 villeDepart VARCHAR(255),
 villeDest VARCHAR(255),
 dateDepart TIMESTAMP (0),
 dateArrivee TIMESTAMP (0)
)
```


Exp: (suite)

Nous allons ajouter une condition sur les dates de départ et d'arrivée:

```
ALTER TABLE vol ADD CONSTRAINT "VOL_CHK1" CHECK
(dateDepart < dateArrivee);
```


Exp: (suite)

L'insertion d'un nouveau vols qui ne satisfait pas la condition précédente sera refusée.

```
INSERT INTO vol(id, villeDepart, villeDest,
dateDepart, dateArrivee) VALUES (1, 'Batna',
'Setif', TIMESTAMP '2016-01-25 08:00:00 +1:00',
TIMESTAMP '2016-01-25 07:00:00 +1:00');
```

Exp: (suite)

Nous allons créer un déclencheur pour ajouter 1h30 au vol qui a la date de départ et supérieure ou égale à la date d'arrivée:

```
CREATE TRIGGER TRIGGER1
BEFORE INSERT ON VOL
FOR EACH ROW
WHEN (new.dateArrivee <= new.dateDepart)
BEGIN
  :new.dateArrivee := :new.dateDepart + interval
'90' MINUTE;
END;
```


Exp: (suite)

A l'insertion d'un nouveau vols, nous devons vérifier que l'heure de départ est inférieur à l'heure d'arrivée, sinon l'insertion déclenche le trigger pour ajouter 90 minutes à l'heure de départ.

```
INSERT INTO vol(id, villeDepart, villeDest,
dateDepart, dateArrivee) VALUES (1, 'Batna',
'Setif', TIMESTAMP '2016-01-25 08:00:00 +1:00',
TIMESTAMP '2016-01-25 07:00:00 +1:00');
```

Dans la table nous allons trouver que l'heure d'arrivée est 09:30 au lieu de 07:00.

3. Les vues

3.1. Définition

En général, une vue est une tables virtuelles dont le schéma et le contenu sont dérivés de la base réelle par un ensemble de requêtes.

Dans la norme SQL, la notion de vue a été **réduite** à **une seule relation**. Une vue est donc finalement une table virtuelle **calculable** par une requête.

3.1. Définition (suite)

La syntaxe générale de la commande SQL de création de vue est :

```
CREATE VIEW <nom de vue> [(liste d'attributs)]
AS <question>
[WITH CHECK OPTION]
```

Tel que:

`(liste d'attributs)` est la liste d'attributs de la vue.

`<question>` une requête SQL pour sélectionner les tuples qui vont remplir la table virtuelle.

`[WITH CHECK OPTION]` sera traitée dans la section suivante.

3.1. Définition (suite)

La suppression de la vue se fait par:

```
DROP VIEW <nom de vue>
```

Exp.

Soit la base MAGASIN contenant les tables suivantes:

VENTES(NUMV, NUMPRO, NUMFOU, DATE_VENTE, QUANTITE, PRIX)

PRODUITS (NUMPRO, NOM, MARQUE, TYPE_PROD, PRIX)

FOURNISSEURS (NUMFOU, NOM, VILLE, REGION, TELEPHONE)

Exp. (suite)

```
CREATE TABLE VENTES
(NUMV INT NOT NULL PRIMARY KEY,
 NUMPRO INT NOT NULL,
 NUMFOU INT NOT NULL,
 DATE_VENTE DATE,
 QUANTITE INT,
 PRIX DECIMAL(10,2) )
CREATE TABLE PRODUITS
(NUMPRO INT NOT NULL PRIMARY KEY,
 NOM VARCHAR(255),
 MARQUE VARCHAR(255),
 TYPE_PROD VARCHAR(255),
 PRIX DECIMAL(10,2) )
```


Exp. (suite)

```
CREATE TABLE FOURNISSEURS
(NUMFOU INT NOT NULL PRIMARY KEY,
 NOM VARCHAR(255),
 VILLE VARCHAR(255),
 REGION VARCHAR(255),
 TELEPHONE VARCHAR(10)
)

ALTER TABLE VENTES ADD CONSTRAINT "Prod" FOREIGN KEY
(NUMPRO) REFERENCES PRODUITS(NUMPRO);
ALTER TABLE VENTES ADD CONSTRAINT "Four" FOREIGN KEY
(NUMFOU) REFERENCES FOURNISSEURS(NUMFOU);
```


Exp. (suite)

Définissons la vue suivante:

```
CREATE VIEW VENTEDOC
(NUMV,NOMPRO,MARQUE,NOMFOU,VILLE,REGION, DATE,
QUANTITE, PRIX)
AS
SELECT
V.NUMV,P.NOM,P.MARQUE,F.NOM,F.VILLE,F.REGION,V.DATE,
V.QUANTITE, V.PRIX
FROM VENTES V, PRODUITS P, FOURNISSEURS F
WHERE V.NUMPRO=P.NUMPRO AND V.NUMFOU=F.NUMFOU
```

3.2. Utilisation des vues pour l'interrogation

Dans les requêtes de consultation, (comme SELECT) une vue est traitée de la même manière qu'une table.

Exp.

```
SELECT NOMPRO,MARQUE FROM VENTEDOC
WHERE PRIX > 100
```

Dans le cas général, les **mises à jour** appliquées sur les **vues** doivent être **traduites** en mises à jour sur les **tables**.

3.2. Utilisation des vues pour l'interrogation (suite)

La clause [WITH CHECK OPTION] de la requête de création de vue, permet de vérifier, lors d'une mise à jour sur une vue, que les lignes concernées sont sélectionnées par la requête de la vue.

Exp.

```
CREATE TABLE ETUDIANT (
 NUM_ETD INT NOT NULL PRIMARY KEY,
 NOM_ETD VARCHAR(255),
 PRENOM_ETD VARCHAR(255),
 ADR VARCHAR(255),
 DATE_NAIS DATE
)
```


3.2. Utilisation des vues pour l'interrogation (suite)

Exp. (suite)

```
CREATE VIEW ETUDIANTS_90
AS
SELECT * FROM ETUDIANTS WHERE DATE_NAIS BETWEEN
TO_DATE('1990/01/01','yyyy/mm/dd') AND
TO_DATE('1999/12/31', 'yyyy/mm/dd');

INSERT INTO ETUDIANTS_90 VALUES
(1,'x','y','Batna',TO_DATE('1978/01/01',
'yyyy/mm/dd'));
```

Cette requête ajoute une ligne dans la table réelle.

3.2. Utilisation des vues pour l'interrogation (suite)

```
SELECT * FROM ETUDIANTS_90;
```

Cette requête ne retourne pas l'étudiant inséré !!!

```
UPDATE ETUDIANTS_90
SET DATE_NAIS = TO_DATE('1979/01/01','yyyy/mm/dd')
WHERE NOM_ETD='x';
```

Cette requête ne fait rien !!!

```
DELETE FROM ETUDIANTS_90 WHERE NOM_ETD='x';
```

Cette requête ne fait rien !!!

3.2. Utilisation des vues pour l'interrogation (suite)

Pour éviter ces problèmes, on utilise la clause [WITH CHECK OPTION]:

```
CREATE VIEW ETUDIANTS_90
AS SELECT * FROM ETUDIANTS WHERE DATE_NAIS BETWEEN
TO_DATE('1990/01/01','yyyy/mm/dd') AND
TO_DATE('1999/12/31', 'yyyy/mm/dd')
WITH CHECK OPTION;
```


3.2. Utilisation des vues pour l'interrogation (suite)

Si on lance la requête d'insertion:

```
INSERT INTO ETUDIANTS_90 VALUES
(1,'x','y','Batna',TO_DATE('1978/01/01',
'yyyy/mm/dd'));
```

nous aurons l'erreur suivante:

```
Erreur SQL : ORA-01402: view WITH CHECK OPTION
where-clause violation
```


3.4. Matérialisation des vues

Problème 1:

L'évaluation d'une vue dérivée à partir de relations est très **couteuse**. Si plusieurs utilisateurs accèdent à la vue, cette dernière doit être **recalculée** pour chacun.

Solution:

Stocker la version actuelle de la vue pour éviter de la recalculer. Une telle vue est appelée vue **matérialisée**.

L'accès à une vue matérialisée est plus rapide parce qu'on ne recalcule pas le contenu de la vue.

3.4. Matérialisation des vues (suite)

Problème 2:

Si les relations utilisées dans la génération de la vue sont mises à jour, le contenu de la vue matérialisée sera **faut**.

Le SGBD doit **maintenir** la vue matérialisée à jour. L'administrateur doit préciser **quand** et **comment** mettre à jour la vue matérialisée.

3.4. Matérialisation des vues (suite)

Quand mettre à jour la vue matérialisée?

La mise à jour peut être **immédiate** ou **différée**.

Dans le mode **immédiat**, la mise à jour de la vue doit être une partie d'une transaction de mise à jour des relations utilisées par la vue.

Avantage: La vue est à jours en **permanence**.

Inconvénient: La mise à jour des relations utilisées par la vue devienne **lente**.

3.4. Matérialisation des vues (suite)

Dans le mode **différé**, la mise à jour de la vue se fait d'une manière **périodique** (1 fois par jour, ou par semaine…) ou d'une manière forcée (par exemple après un certain nombre de mises à jours sur les relations utilisées dans la définition de la vue).

3.4. Matérialisation des vues (suite)

Comment mettre à jour la vue matérialisée?

Le plus simple est de **recalculer** la vue **entière**.

Une solution **meilleure** est de ne recalculer que la **partie modifiée**.

Soit **u** la modification appliquée à la relation **R**.

Et soit R^+ contient les tuples à ajouter à **R** en appliquant **u** (ajout de tuples).

R^- contient les tuples à supprimer de **R** en appliquant **u** (suppression de tuples)

3.4. Matérialisation des vues (suite)

Comment mettre à jour la vue matérialisée? (suite)

La relation **R** après application de **u** contient:

$$(R - R^-) \cup R^+$$

Exp.

Soit une entreprise qui a des annexes dans plusieurs villes: Alger, Batna, Oran. Le gérant veut maintenir une base de données des employés et les projets sur lesquels ils travaillent. Le gérant utilise 4 tables:

Contraintes d'intégrité, Triggers et Vues

Exp. (suite)

```
Emp (idEmp, nomEmp, poste);
Prj (idPrj, nomPrj, designation, budget, ville);
Sal (poste, salaire);
Aff (idEmp, idPrj, resp, dur)
```
//Affectation d'un employé à un projet avec telle responsabilité et telle durée.

Contraintes d'intégrité, Triggers et Vues

Exp. (suite)

Soit la vue qui donne les noms des employés et leurs responsabilités:

```
CREATE VIEW ER(nomEmp, resp)
AS SELECT DISTINCT nomEmp, resp
FROM Emp, Aff
WHERE Emp.idEmp = Aff.idEmp
```

Soit Emp^+ l'ensemble des tuples à ajouter à `Emp`, et soit Aff^+ l'ensemble de tuples à ajouter à `Aff`.

Contraintes d'intégrité, Triggers et Vues

Exp. (suite)

Les changements de la vue ER peuvent être calculés comme suit:

```
ER+ = (SELECT nomEmp, resp
       FROM Emp, Aff+
       WHERE Emp.idEmp = Aff+.idEmp)
       UNION
       (SELECT nomEmp, resp
       FROM Emp+, Aff
       WHERE Emp+.idEmp = Aff.idEmp)
       UNION
       (SELECT nomEmp, resp
       FROM Emp+, Aff+
       WHERE Emp+.idEmp = Aff+.idEmp)
```


Conception et optimisation de schéma relationnel

PLAN

1. Introduction et problématique
2. Dépendance fonctionnelle
3. Déduction (Axiomes d'Armstrong) et couverture minimale
4. Formes normales

1. Introduction et problématique

Le modèle relationnel de données est proposé par Ted Codd (1923-2003).

Exp.

Soit la table suivante, dans une base de données relationnelle, avec le schéma relationnel suivant:

R(Entreprise, CH.Affaire, N°Facture, Client, Montant)

Les contraintes suivantes doivent être respectées:

Entreprise -> Ch.Affaire

Entreprise, N°Facture -> Client

Entreprise, N°Facture -> Montant

1. Introduction et problématique (suite)

Exp. (suite)

Entreprise	Ch.Affaire	N°Facture	Client	Montant
Ent1	80M	01501	C1	450
Ent1	80M	01540	C2	10456
Ent2	50M	01502	C3	500
Ent3	100M	01503	C4	450
Ent4	45M	01510	C5	164
Ent4	45M	01518	C6	9789
…	…	…	…	…

1. Introduction et problématique (suite)

Exp. (suite)

Une nouvelle facture à insérer dans la table <Ent2, 50M, 01541, C7, 450>

Le chiffre d'affaire est une information **redondante**.

Si une entreprise change de nom, **toutes les lignes** contenant cette entreprise doivent être **changées**.

Si une nouvelle entreprise se crée, sans clients au début, on **ne pourra pas** l'ajouter à la table.

1. Introduction et problématique (suite)

Problèmes:

- Problème de redondances.
- Problème de mise à jour.
- Une ligne qui manque de données ne peut être insérée.

1. Introduction et problématique (suite)

Solution:

- Décomposer la table en plusieurs tables.

Pour que cette décomposition soit dans les normes, nous devons voir comment rendre la table en **forme normale**.

La théorie de **normalisation** est basée sur la notion de **dépendance fonctionnelle**.

2. Dépendance fonctionnelle

2.1. Définition

Les dépendances fonctionnelles traduisent des contraintes d'intégrité sur les données.

Exp. Deux personnes différentes peuvent avoir même nom et prénom mais jamais le même numéro de sécurité sociale.

SS → Nom, Prénom

Un numéro de sécurité sociale est associé un seul nom et prénom.

2.2. Propriétés des dépendances fonctionnelles

Les dépendances fonctionnelles possèdent trois propriétés fondamentales qui ont été découvertes par William Armstrong: Réflexivité, Augmentation, Transitivité.

3. Déduction (Axiomes d'Armstrong) et couverture minimale

Soit X, Y, Z trois ensembles d'attributs. L'écriture X,Z est une écriture simplifiée de l'union ensembliste $X \cup Z$.

3.1. Réflexivité

Si $Y \subseteq X \Rightarrow X \rightarrow Y$; tout ensemble d'attributs détermine lui-même ou une partie de lui-même.

Exp:

Nom,Prenom → Prenom càd On ne peut pas trouver une valeur de (Nom,Prenom) qui détermine 2 valeurs différentes de (Prenom).

3.2. Augmentation

Si $X \rightarrow Y \Rightarrow \forall Z$: $X,Z \rightarrow Y,Z$; si X détermine Y, les deux ensembles d'attributs peuvent être enrichis par un même troisième.

Exp:

Nom,Prenom $\rightarrow$ Adr $\Rightarrow$ Nom,Prenom,Tel $\rightarrow$ Adr,Tel

3.3. Transitivité

Si $X \rightarrow Y$ et $Y \rightarrow Z \Rightarrow X \rightarrow Z$; Une valeur de X détermine une seule valeur de Y, et cette valeur de Y détermine une seule valeur de Z, donc la valeur de X détermine la seule valeur de Z.

Exp:

Nom,Prenom $\rightarrow$ TelFixe et TelFixe $\rightarrow$ Adr $\Rightarrow$ Nom,Prenom $\rightarrow$ Adr

À partir de ces trois propriétés on déduit d'autres:

3.4. Union

Si $X \rightarrow Y$ et $X \rightarrow Z \Rightarrow X \rightarrow Y,Z$

Exp:

TelFixe $\rightarrow$ Nom et TelFixe $\rightarrow$ Prenom $\Rightarrow$ TelFixe $\rightarrow$ Nom,Prenom

3.5. Décomposition

Si $X \rightarrow Y$ et $Z \subseteq Y \Rightarrow X \rightarrow Z$

Exp:

TelFixe $\rightarrow$ Nom,Prenom $\Rightarrow$ TelFixe $\rightarrow$ Nom et TelFixe $\rightarrow$ Prenom

3.6. Pseudo-transitivité

Si $X \rightarrow Y$ et $W,Y \rightarrow Z \Rightarrow W,X \rightarrow Z$

Parce que:

Par augmentation: $\mathbf{X \rightarrow Y} \Rightarrow W,X \rightarrow W,Y$

Par transitivité: $W,X \rightarrow W,Y$ et $\mathbf{W,Y \rightarrow Z} \Rightarrow \mathbf{W,X \rightarrow Z}$

Exp:

Nom,Prenom → Adr ⇒ Nom,Prenom,DateNais → Adr,DateNais

Nom,Prenom,DateNais → Adr,DateNais et Adr,DateNais → Tel ⇒ Nom,Prenom,DateNais → Tel

3.7. Dépendance fonctionnelle élémentaire

C'est une dépendance de la forme : $X \rightarrow A$, où A est un attribut **unique non inclus** dans X ($A \notin X$), tel que: $\forall$ X' $\subset$ X, il n'existe pas de dépendance fonctionnelle $X' \rightarrow A$.

Exp:

Soient les deux D.F. :

Num, Nom → Adr (1)

Num → Adr (2)

La D.F. (1) est une D.F. non élémentaire car Num $\subset$ Num,Nom et il existe la D.F. Num → Adr.

Par contre la D.F. (2) est une D.F. élémentaire.

3.8. Fermeture transitive

C'est l'ensemble des DF **élémentaires** enrichi de toutes les DF **élémentaires** déduites par **transitivité**.

Exp:

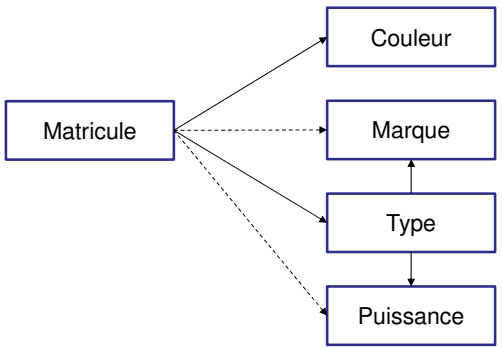
F = { Matricule → Type; Type → Marque; Type → Puissance; Matricule → Couleur}

on déduit la fermeture transitive :

$F^+ = F \cup$ {Matricule → Marque; Matricule → Puissance}

3.8. Fermeture transitive (suite)

Exp: (suite)



3.8. Fermeture transitive (suite)

À partir de la notion de fermeture transitive, on peut définir la notion **d'équivalence** entre deux ensembles de DF.

Deux ensembles de DF sont dits **équivalents** s'ils ont la **même fermeture transitive**.

3.9. Couverture minimale

C'est un **sous-ensemble minimal** de DF permettant de générer toutes les autres.

Exp: idModule → idFilière
idFilière → nomFilière
~~idModule → nomFilière~~ (par transitivité)
jour, heure, salle → idEns
~~jour, heure, salle → idFilière~~ (par transitivité)
jour, heure, salle → section
jour, heure, salle → groupe
jour, heure, salle → an_étude
jour, heure, salle → idModule
~~jour, heure, salle, idEns → idFilière, idEns~~ (par augmentation)

4. Formes normales

Les formes normales sont utiles pour supprimer (ou minimiser) les redondances dans les relations.

4.1. La première forme normale (1FN)

Une relation est en **première forme normale** si aucun de ses domaines (attributs) ne peut contenir des éléments qui soient eux-mêmes des ensembles. Une relation qui n'est pas en première forme normale est dite **non normalisée**.

Exp.

Personne (IdPersonne, Nom, Prénom, Adresse(IdAdr, Num, Rue, Ville, CodePostal, Pays))

4.1. La première forme normale (1FN) (suite)

Exp. (suite)

La relation Personne n'est pas en 1FN.

Transformation en deux relations:

Personne (IdPersonne, Nom, Prénom, IdAdr)

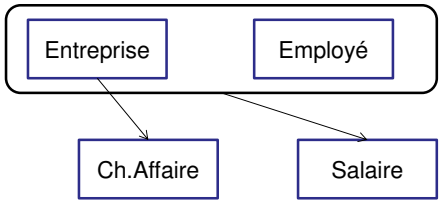
Adresse(IdAdr, IdPersonne, Num, Rue, Ville, CodePostal, Pays)

4.2. La deuxième forme normale (2FN)

Exp.

Dans un serveur d'une entreprise, on trouve la table:

R(Entreprise, Ch.Affaire, Employé, Salaire)



4.2. La deuxième forme normale (2FN) (suite)

R satisfait les dépendances fonctionnelles suivantes:

Entreprise → Ch.Affaire

Entreprise, Employé → Salaire

Problème:

La relation R est en 1FN, mais beaucoup de redondances!

Solution:

Mettre R en 2FN.

4.2. La deuxième forme normale (2FN) (suite)

Une relation **R** définie sur un ensemble d'attributs **U** et sur un ensemble de dépendances fonctionnelles **F** est en **2FN** ssi tout **attribut** n'appartenant pas à une clé **dépend directement** d'une clé.

4.2. La deuxième forme normale (2FN) (suite)

Une relation **R** est en **2FN** si:

- elle est en 1FN;
- tout attribut n'appartenant pas à une clé ne dépend pas d'une partie de cette clé.

Exp.

Dans l'exemple précédent, l'attribut « Salaire » dépend directement de toute la clé, mais l'attribut « Ch.Affaire » dépend d'une partie de la clé.

Donc **R** n'est pas en 2FN.

4.2. La deuxième forme normale (2FN) (suite)

Exp. (suite)

Il faut donc décomposer R(Entreprise, Ch.Affaire, Employé, Salaire) en deux relations:

R1(Entreprise, Ch.Affaire) et

R2(Entreprise, Employé, Salaire)

4.3. La troisième forme normale (3FN)

Exp.

R (NumEtudiant, NomEtudiant, Filière, Année, Salle)

Avec les dépendances:

Filière, Année → Salle

…

Problème: Plusieurs lignes contenant la même valeur pour le triplet < Filière, Année, Salle >

4.3. La troisième forme normale (3FN) (suite)

Soit une relation **R**, soit **X** un sous-ensemble d'attributs de **R** et soit **A** un attribut de **R**.

R est en **3NF** si pour chaque DF : X → A dans **R**, au-moins une des conditions suivantes est remplie :

1. **X** contient **A**.
2. **X** est une clé.
3. **A** fait partie d'une clé de **R**.

4.3. La troisième forme normale (3FN) (suite)

Ou bien:

Une relation **R** est en 3FN si:

- elle est en 2FN;
- tout attribut n'appartenant pas à une clé ne dépend pas d'un attribut non clé.

4.3. La troisième forme normale (3FN) (suite)

Exp. (suite)

La relation R (NumEtudiant, NomEtudiant, Filière, Année, Salle) de l'exemple précédent n'est pas en 3FN.

Il faut la décomposer:

R1(NumEtudiant, NomEtudiant, IdF)

R2(IdF, Filière, Année)

R3(IdF, Salle)

4.3. La troisième forme normale (3FN) (suite)

4.3.1. Algorithme de décomposition en 3FN

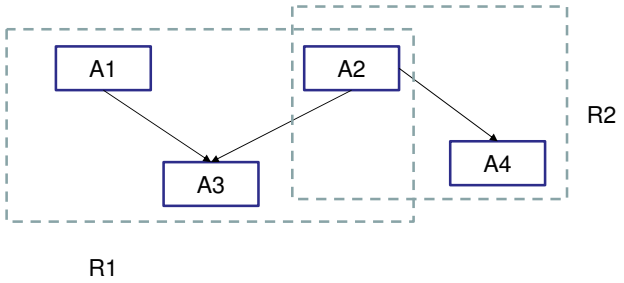
Pour toute relation, il existe donc au moins une **décomposition** en 3FN **préservant** les DF et **sans perte d'information**.

Etapes de l'algorithme (Heath):

- Dessiner le graphe des DF;
- Découper le graphe en gardant toutes les DF et sans perdre d'attributs.

4.3.1. Algorithme de décomposition en 3FN (suite)

Exp.



R1 et R2 sont en 3FN.

4.3.2. Algorithme de synthèse en 3FN

L'algorithme de synthèse permet de décomposer une relation non 3FN en sous-relations en 3FN.

Etapes de l'algorithme:

- Construire une couverture minimale F des DF élémentaires;
- Partitionner F en groupes F_i tels que les DF dans chaque F_i ait le même ensemble d'attributs à gauche.
- Pour chaque ensemble F_i de DF, construire une relation composée des attributs de F_i, la clé de la relation sera la partie gauche commune des DF de F_i;

4.3.2. Algorithme de synthèse en 3FN (suite)

- S'il reste des attributs isolés, on les sort dans une table qui est bien en 3FN puisqu'elle n'a pas de DF entre ses attributs;

4.3.2. Algorithme de synthèse en 3FN (suite)

Exp: idModule → idFilière
idFilière → nomFilière
~~idModule → nomFilière~~ (par transitivité)
jour, heure, salle → idEns
~~jour, heure, salle → idFilière~~ (par transitivité)
jour, heure, salle → section
jour, heure, salle → groupe
jour, heure, salle → an_étude
jour, heure, salle → idModule
~~jour, heure, salle, idEns → idFilière, idEns~~ (par augmentation)

4.3.2. Algorithme de synthèse en 3FN (suite)

Exp: (suite)

F = {**F**$_1$ = {idModule → idFilière}
F$_2$ = {idFilière → nomFilière}
F$_3$ = {jour, heure, salle → idEns
jour, heure, salle → section
jour, heure, salle → groupe
jour, heure, salle → an_étude
jour, heure, salle → idModule}}

R1(idModule, idFilière)
R2(idFilière, nomFilière)
R3(jour, heure, salle, idEns, section, groupe, an_étude, idModule)

4.4. La forme normale Boyce-Codd (BCNF)

Proposée par Raymond Boyce et Ted Codd

Soit une relation **R**, soit **X** un sous-ensemble d'attributs de **R** et soit **A** un attribut de **R**.

R est en **BCNF** si pour chaque DF : X → A dans **R**, au-moins une des conditions suivantes est remplie :

1. **X** contient **A**.
2. **X** est une clé de **R**.

4.4. La forme normale Boyce-Codd (BCNF) (suite)

Ou bien

Une relation **R** est en BCNF si:

- elle est en 3FN;
- aucun attribut ne dépend d'un attribut non clé

4.4. La forme normale Boyce-Codd (BCNF) (suite)

Exp.

R (Etudiant, Module, Enseignant)

Avec les dépendances:

Etudiant, Module → Enseignant

Enseignant → Module

Les clés candidates (possibles) sont:

Etudiant, Module

Etudiant, Enseignant

R est en 3FN, mais n'est pas en BCNF

PLAN

1. Basiques de l'algèbre relationnelle
   1. Opérateurs ensemblistes
   2. Equivalences algébriques
2. Optimisation

1. Basiques de l'algèbre relationnelle

1.1 Opérateurs ensemblistes

L'union:

Soit R et S deux relations de même schéma (même attributs)

R1

X	Y
A	B
B	B
C	D

R2

X	Y
E	F
C	D

R1 ∪ R2

X	Y
A	B
B	B
C	D
E	F

Evaluation et optimisation de requête

1.1 Opérateurs ensemblistes (suite)

L'intersection:

Soit R et S deux relations de même schéma (même attributs)

R1

X	Y
A	B
B	B
C	D

R2

X	Y
E	F
C	D

R1 ∩ R2

X	Y
C	D

Evaluation et optimisation de requête

1.1 Opérateurs ensemblistes (suite)

La différence:

Soit R et S deux relations de même schéma (même attributs)

R1

X	Y
A	B
B	B
C	D

R2

X	Y
E	F
C	D

R1 - R2

X	Y
A	B
B	B

Evaluation et optimisation de requête

1.1 Opérateurs ensemblistes (suite)

Le produit cartésien:

R1

X	Y
A	B
B	B
C	D

R2

Z	W	V
E	F	G
C	D	H

R1 x R2

X	Y	Z	W	V
A	B	E	F	G
A	B	C	D	H
B	B	E	F	G
B	B	C	D	H
C	D	E	F	G
C	D	C	D	H

Evaluation et optimisation de requête

1.1 Opérateurs ensemblistes (suite)

La projection: Π[X,Y](R)

R

X	Y	Z	W	V
A	B	E	F	G
A	B	C	D	H
B	B	E	F	G
B	B	C	D	H
C	D	E	F	G
C	D	C	D	H

Π[X,Y](R)

X	Y
A	B
B	B
C	D

1.1 Opérateurs ensemblistes (suite)

La sélection: σ[C](R)

R

X	Y	Z	W	V
A	B	E	F	G
A	B	C	D	H
B	B	E	F	G
B	B	C	D	H
C	D	E	F	G
C	D	C	D	H

σ[X=A](R)

X	Y	Z	W	V
A	B	E	F	G
A	B	C	D	H

1.1 Opérateurs ensemblistes (suite)

La jointure naturelle:

$$Join[A,B](R,S) = \sigma[A=B](R \times S)$$

1.1 Opérateurs ensemblistes (suite)

La division: A/B

R1

X	Y
A	B
B	B
C	D
A	D
B	D

R2

Y
B
D

R1 / R2

X
A
B

Les éléments de R1.X qui forment un couple **avec tous les** éléments de R2.Y

1.2. Equivalences algébriques

Commutativité et associativité de la jointure

$$Join(R,S) = Join(S,R)$$
$$Join(Join(R,S),T) = Join(R,Join(S,T))$$

Cascade de projections : Soit $\{X_1,\ldots,X_m\} \subseteq \{Y_1,\ldots,Y_n\}$,

$$\Pi[X_1,\ldots,X_m](\Pi[Y_1,\ldots,Y_n](R)) = \Pi[X_1,\ldots,X_m](R)$$

Cascade de sélections

$$\sigma[C_1](\sigma[C_2](R)) = \sigma[C_1 \wedge C_2](R)$$

1.2. Equivalences algébriques (suite)

Exp: Vérifier les propriétés précédentes avec:

R

X	Y
1	4
2	5
3	6
1	7

S

X	Z
1	8
2	9
3	10
4	11

T

X	W
1	4
2	3
5	9
6	8

1.2. Equivalences algébriques (suite)

Commutation de sélection et projection:

Si la condition C ne porte que sur $X_1, \dots, X_n$

$$\Pi[X_1,\dots,X_n](\sigma[C](R)) = \sigma[C](\Pi[X_1,\dots,X_n](R))$$

Et d'une manière générale, si C porte aussi sur $Y_1, \dots, Y_m$

$$\Pi[X_1,\dots,X_n](\sigma[C](R)) =$$
$$\Pi[X_1,\dots,X_n](\sigma[C](\Pi[X_1,\dots,X_n, Y_1, \dots, Y_m](R)))$$

1.2. Equivalences algébriques (suite)

Commutation de sélection et $x \cup$ - Join:

$$\sigma[C](R \text{ op } S) = \sigma[C](R) \text{ op } \sigma[C](S)$$

Commutation de projection et $x \cup$:

$$\Pi[X_1,\dots,X_n](R \text{ op } S) = \Pi[X_1,\dots,X_n](R) \text{ op } \Pi[X_1,\dots,X_n](S)$$

1.2. Equivalences algébriques (suite)

Exp.

```
SELECT *
FROM R
WHERE X1=a OR X2=b OR X2=c
AND NOT (X2=b OR X2=c)
```

Logique:

$(A \vee B \vee C) \wedge \neg(B \vee C) \Leftrightarrow A \wedge \neg(B \vee C)$

1.2. Equivalences algébriques (suite)

Exp. (suite)

La requête devient

```
SELECT *
FROM R
WHERE X1=a
AND NOT (X2=b OR X2=c)
```


1.2. Equivalences algébriques (suite)

Avec contraintes d'intégrité

Exp1:

Join (Π[X,Y](R) , Π[X,Z](R))

avec $X \rightarrow Y$ (dépendance fonctionnelle)

Le résultat est équivalent à: Π[X,Y,Z](R)

Exp:

X	Y	Z
1	4	7
2	5	8
3	6	9
1	7	2

1.2. Equivalences algébriques (suite)

Avec contraintes d'intégrité

Exp2:

```
SELECT *
FROM R
WHERE (X=a) AND (Y≤10)
```

Avec la contrainte: $X=a \Rightarrow Y \geq 12$

```
SELECT *
FROM R
WHERE (X=a) AND (Y≤10) AND (Y≥12)
```


2. Optimisation

Optimiser les ressources suivantes:

- Processeurs;
- Accès disques;
- Communication (dans le cas de BDD distribuées)

En termes de:

- Temps de réponses;
- Le débit des requêtes (nombre de requêtes traitées par unité de temps).

Evaluation et optimisation de requête

2. Optimisation (suite)

Exp.

Soit une base avec deux relations:

Entreprise(nomEnt, adresse, gérant)

Annexe(nomEnt, numSiège, nbEmployés)

Soit la requête:

Adresses des entreprises ayant des annexes de plus de 50 employés.

Evaluation et optimisation de requête

2. Optimisation (suite)

Exp. (suite)

En SQL:

```
SELECT Adresse
FROM Entreprise, Annexe
WHERE nbEmployés > 50
AND Entreprise.nomEnt = Annexe.nomEnt
```


Evaluation et optimisation de requête

2. Optimisation (suite)

Exp. (suite)

En algèbre:

Π[nomEnt](σ[nbEmp > 50](Join(Entreprise, Annexe)))

Π[nomEnt](Join(Entreprise, σ[nbEmp > 50](Annexe)))

Faire la **sélection avant la jointure** permet d'éliminer plusieurs enregistrements avant de faire la jointure.

Evaluation et optimisation de requête

2. Optimisation (suite)

Pour une requête SQL il existe plusieurs expressions algébriques équivalentes.

Pour optimiser une requête il faut choisir, parmi les expressions algébriques équivalentes, la meilleure.

Pour trouver les expressions équivalentes on doit utiliser les **équivalences algébriques** vues dans le chapitre II section 1.2.

Les transactions

Plan

1. Définition
2. Transitions concurrentes
3. Les verrous de tables
4. Les verrous de lignes

Les transactions

1. Définition

La gestion de transactions (ou **atomicité**) permet d'assurer qu'un groupe de mises à jour soit totalement exécuté ou totalement annulé.

Une transaction est construite à l'aide de 3 instructions du SGBD:

Sous mySQL:

`START TRANSACTION` : ouverture d'une transaction.

`COMMIT` : Clôture de la transaction avec confirmation.

`ROLLBACK` : Clôture de la transaction avec annulation.

Les transactions

1. Définition (suite)

Sous Oracle SQL Plus:

Une transaction est démarrée à l'ouverture de la session sous SQL Plus. La commande `START TRANSACTION` est donc **implicite**.

`COMMIT` : Validation des opérations effectuées dans la transaction et termine la transaction.

`ROLLBACK` : Annulation des opérations effectuées dans la transaction et termine la transaction.

Les transactions

1. Définition (suite)

Le principe est comme suit:

Sous mySQL:

```
SET autocommit = 0; //pour annuler le COMMIT automatique
START TRANSACTION;
        acquisition de données;
        modification de données;
        si erreur ROLLBACK et sortie;
COMMIT;
```

Les transactions

1. Définition (suite)

Sous Oracle SQL Plus:

`SET AUTOCOMMIT ON` (commit immédiat après chaque opération)

ou `SET AUTOCOMMIT OFF` (Commit explicite avec la commande `commit` ou à la fermeture de la session)

acquisition de données;

modification de données;

si erreur `ROLLBACK`;

`COMMIT;`

Les transactions

1. Définition (suite)

Remarque:

En cas de panne au milieu d'une transaction, le SGBD doit effacer toute trace de la transaction annulée avant la reprise.

Sous mySQL, les tables de type MyISAM ne sont pas transactionnelles, il faut utiliser le type InnoDB.

Les transactions

2. Transactions concurrentes

Deux transactions sont concurrentes si elles accèdent en même temps aux mêmes données.

Cette concurrence peut générer plusieurs problèmes:

- Perte de mise à jour;
- Lecture impropre;
- Lecture non reproductible.

Les transactions

2. Transactions concurrentes (suite)

2.1. Problème de perte de mise à jour

Soit deux transactions concurrentes T1 et T2. Les deux transactions ajoutent une valeur à la capacité d'une salle de cinéma.

Problème: Mise à jour de T1 écrasée par celle de T2

T1	T2
SELECT Capacité FROM salle WHERE NomCinema='cinema1' AND NumSalle=1	
x = Capacité + 10	
	SELECT Capacité FROM salle WHERE NomCinema='cinema1' AND NumSalle=1
UPDATE salle SET Capacité=x WHERE NomCinema='cinema1' AND NumSalle=1	
	x = Capacité + 20
	UPDATE salle SET Capacité=x WHERE NomCinema='cinema1' AND NumSalle=1

2.2. Problème de lecture impropre

Soit deux transactions concurrentes T1 et T2.

T1	T2
UPDATE ETUDIANTS SET DATE_NAIS = '2090-01-01' WHERE NUM = '1' ;	
	SELECT DATE_NAIS FROM ETUDIANTS WHERE NUM = '1' ;
ROLLBACK	

Problème: T2 lit des données modifiées par T1 qui seront annulés par le ROLLBACK juste après.

2.3. Problème de lecture non reproductible

T1	T2
SELECT DATE_NAIS FROM ETUDIANTS WHERE NUM = '1' ;	
	UPDATE ETUDIANTS SET DATE_NAIS = '2090-05-01' WHERE NUM = '1' ;
SELECT DATE_NAIS FROM ETUDIANTS WHERE NUM = '1' ;	
COMMIT	

Problème: T1 ne trouve pas la même valeur dans les deux lectures.

3. Les verrous de tables

Pour résoudre le problème des transactions concurrentes on doit utiliser les verrous. Il existe deux types de verrous:

- **Verrou exclusif**: ou pour une MAJ. Il interdit à d'autres utilisateurs d'effectuer simultanément des MAJ ou des consultations sur une même partie de la BD.
- **Verrou partagé**: Il permet à d'autres utilisateurs d'effectuer simultanément des consultations sur une même partie de la BD, et interdit toute mise à jour.

Les transactions

3. Les verrous de tables (suite)

La commande `LOCK TABLES` permet de verrouiller l'accès à une table.

```
LOCK TABLES nom_table [AS alias_table] [READ | WRITE] [, ...];
```

Tel que:

`READ` représente verrou exclusif;

`WRITE` représente verrou partagé.

Les transactions

3. Les verrous de tables (suite)

Exp:

```
LOCK TABLES ETUDIANTS READ;
SELECT NUM, NOM FROM ETUDIANTS; //Pas de pb
UNLOCK TABLES;

LOCK TABLES ETUDIANTS READ;
SELECT NUM, NOM FROM ETUDIANTS AS ETD; //#1100 - Table 'ETD' was not locked with LOCK TABLES
UNLOCK TABLES;
```


Les transactions

3. Les verrous de tables (suite)

Exp:

```
LOCK TABLES ETUDIANTS AS ETD READ;
SELECT NUM, NOM FROM ETUDIANTS; //#1100 - Table 'ETUDIANTS' was not locked with LOCK TABLES
UNLOCK TABLES;

LOCK TABLES ETUDIANTS AS ETD READ;
SELECT NUM, NOM FROM ETUDIANTS AS ETD; //Pas de pb
UNLOCK TABLES;
```


Les transactions

3. Les verrous de tables (suite)

Exp:

```
LOCK TABLES ETUDIANTS READ;
UPDATE ETUDIANTS SET DATE_NAIS='1992-01-01' WHERE NUM=1; //#1099 - Table 'ETUDIANTS' was locked with a READ lock and can't be updated
UNLOCK TABLES;

LOCK TABLES ETUDIANTS WRITE;
UPDATE ETUDIANTS SET DATE_NAIS='1993-01-01' WHERE NUM=1; //# 1 ligne affectée.
UNLOCK TABLES;
```

Les transactions

3. Les verrous de tables (suite)

Exp:

```
LOCK TABLES ENSEIGNANTS READ;
LOCK TABLES ETUDIANTS READ;
SELECT * FROM ENSEIGNANTS; //Pas de pb
SELECT * FROM ETUDIANTS; //#1099 – Table 'ETUDIANTS' was locked with LOCK TABLES
UNLOCK TABLES;
```

Le verrou sur ENSEIGNANTS a été **relâché** lorsque l'on a posé les verrous sur ETUDIANTS.

Les transactions

4. Les verrous de lignes

Ça concerne les tables de type InnoDB uniquement.

- Les requêtes de **modification** et de **suppression** posent **automatiquement** un **verrou exclusif** sur les lignes concernées par la clause WHERE.
- Les requêtes **d'insertion** posent **automatiquement** un **verrou exclusif** sur la ligne insérée.
- Les requêtes de **sélection** ne posent pas de verrous. Il faut donc en poser **explicitement** au besoin. Dans ce cas, on pose un verrou partagé en utilisant LOCK IN SHARE MODE, ou un verrou exclusif avec FOR UPDATE.

Les transactions

4. Les verrous de lignes (suite)

Exp 1:

```
SELECT * FROM ETUDIANTS WHERE NUM = 1 LOCK IN SHARE MODE; //Verrou partagé sur la ligne de l'étudiant numéro 1
```

Les autres sessions peuvent lire la ligne mais pas la modifier.

```
SELECT * FROM ETUDIANTS WHERE NUM = 1 FOR UPDATE; //Verrou exclusif sur la ligne de l'étudiant numéro 1
```

Les autres sessions ne peuvent ni lire ni modifier la ligne.

Les transactions

4. Les verrous de lignes (suite)

Dans le cas d'un verrou de ligne dans une **transaction**, le **verrou** sera **levé** dès que l'on fait un COMMIT ou ROLLBACK.

4. Les verrous de lignes (suite)

Exp 2:

Session 1:

```
START TRANSACTION;
UPDATE ETUDIANTS SET DATE_NAIS = '1990-04-04' WHERE NUM = 1; //Verrou exclusif sur la ligne de l'étudiant 1
```

Session 2:

```
START TRANSACTION;
SELECT * FROM ETUDIANTS WHERE NUM = 1; //Pas de verrou
SELECT * FROM ETUDIANTS WHERE NUM = 1 LOCK IN SHARE MODE; //Verrou partagé sur la ligne de l'étudiant 1
```


4. Les verrous de lignes (suite)

Exp 2: (suite)

La session 1 pose un **verrou exclusif** sur la ligne de l'étudiant 1.

La 1ère requête de la session 2 essai de lire la ligne de l'étudiant 1. **Pas de blocage, pourquoi???**

Le démarrage de la **transaction** de la session 2 utilise des **copies des tables**.

4. Les verrous de lignes (suite)

Exp 2: (suite)

La 2ème requête de la session 2 essai de lire la ligne de l'étudiant 1 et y pose un verrou partagé. **La session se bloque, pourquoi???**

Quand la session pose un **verrou** elle **utilise** les données à partir de la **table d'origine**. Et comme la ligne est verrouillée par la session 1, la session 2 se bloque et attend l'enlèvement du verrou.

4. Les verrous de lignes (suite)

Exp 2: (suite)

Pour **enlever** les **verrous** de la transaction de la session 1, il suffit de **valider** (`COMMIT`) ou **annuler** (`ROLLBACK`) la transaction.

4. Les verrous de lignes (suite)

Exp 3:

Session 1:

```
START TRANSACTION;
INSERT INTO ETUDIANTS (NUM, NOM, PRENOM, DATE_NAIS)
VALUES (6, 'N6', 'P6', '1992-05-01');
```

Session 2:

```
START TRANSACTION;
SELECT * FROM ETUDIANTS WHERE NUM < 3 LOCK IN SHARE
MODE;
SELECT * FROM ETUDIANTS WHERE NUM > 3 LOCK IN SHARE
MODE;
```


4. Les verrous de lignes (suite)

Exp 3: (suite)

La session 1 pose un verrou exclusif sur la ligne de l'étudiant 6.

La 1ère requête de la session 2 passe, mais la 2ème se bloque puisque elle utilise la ligne de l'étudiant 6 qui est > 3.

Pour débloquer la session 2, il suffit de faire un `COMMIT` dans la session 1.