Master IAM
(Intelligence Artificielle & Multimédia)

Bases de données distribuées

http://www.larbiguezouli.com/

Présenté par Dr Larbi GUEZOULI

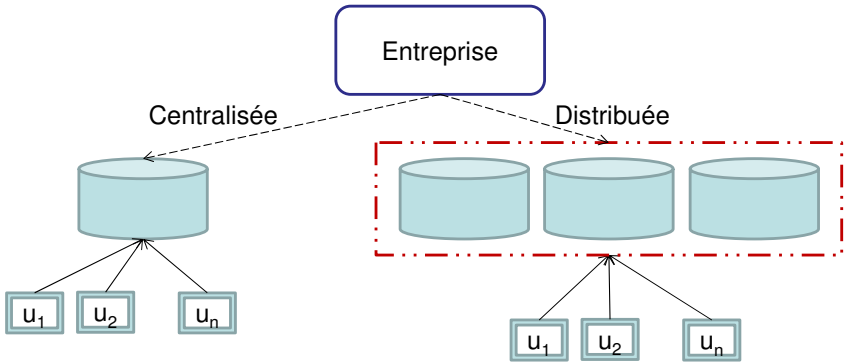
Sommaire

Bibliographie

1. M. Tamer Özsu and Patrick Valduriez, "Principles of Distributed Database Systems", Third Edition, Springer, 2011
2. IBM, "Distributed database programming", Version 6 Release 1, 2008.
3. Georges Gardarin, « Bases de données réparties », Cours BDD Web.

Chapitre I
Introduction

Une entreprise, selon son besoin, peut choisir entre l'utilisation de base de données centralisée ou distribuée.



Quelques définitions

Une base de données **distribuée** est un ensemble de bases de données localisées et gérées par des **sites différents** et apparaissant à l'utilisateur comme une base **unique**. [3]

Une base de données **distribuée** est une collection de multiple bases de données **distribuées** sur un réseau informatique **logiquement inter-reliées**.

Un **client** d'un système de gestion de base de données distribuée est une **application** qui accède aux informations distribuées par les interfaces du système. [3]

Quelques définitions (suite)

Un système de gestion de base de données **distribuées** est un système qui gère des collections de BD **logiquement reliées, distribuées** sur un réseau, en fournissant un mécanisme d'accès qui rend la répartition **transparente** aux utilisateurs. [3]

Un système de gestion de bases de données **distribuées (Distributed DBMS)** est l'application qui permet la gestion de la base de données distribuée et rend la distribution transparente aux utilisateurs.

Pour référencer la BDD distribuée et le DBMS distribuées nous utilisant l'abréviation **DDBS (Distributed DataBase System)**.

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:

```
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.

Exp. (suite)

Si la base de données est centralisée, on peut trouver les noms des employés qui travaillent dans un projet pour une durée supérieure de 12 mois ainsi que leurs salaires avec la requête suivante:

```
SELECT nomEmp, salaire
FROM Emp, Aff, Sal
WHERE Aff.dur > 12
AND Emp.idEmp = Aff.idEmp
AND Sal.poste = Emp.poste
```

Exp. (suite)

Maintenant, si la base est **distribuée** sur les différents annexes, de telle sorte que les information des employés et les projets de chaque annexe sont stockés dans une base de données localisée dans cette annexe.

L'utilisateur doit **posé la même question** au système sans faire attention à la distribution des information **(transparence)**, et c'est le système qui s'occupe de la récupération des informations.

Pour cela, il a été défini plusieurs types de transparence:

1/ Transparence de données (Indépendance)

La définition des données dans une base de données se fait à deux niveaux (structure logique et structure physique), se qui permet de définir une **indépendance de données logique** et **indépendance de données physique**.

1/ Transparence de données (suite)

L'indépendance de données logique signifie que les applications de l'utilisateur ne sont **pas autorisées** à modifier la structure logique des données (le schéma).

L'indépendance de données physique signifie que les détails de la structure de stockage des données seront **cachés** aux applications de l'utilisateur.

Remarque: L'application utilisateur ne doit pas être modifiée quand la structure physique des données change.

2/ Transparence du réseau (ou de distribution)

Ce type de transparence fait référence à ce que **l'existence** du **réseau** est **caché** à l'utilisateur. L'accès aux données de la base doit se faire de la même manière soit on mode centralisé ou distribué.

3/ Transparence de réplication (redondance)

Pour des raisons de performance et disponibilité des données, il est préférable de faire des **copies** de quelques données de la base sur plusieurs sites.

Par exemple, les données qui se trouvent sur un autre site et qui sont réclamées souvent par un utilisateur, sera préférable d'en faire une copie sur la base de données local de cet utilisateur.

En plus, si une **machine s'arrête**, on peut trouver les données sur une autre machine.

3/ Transparence de réplication (redondance)

La décision de réplication des données, et de combien de copies doit-on faire, dépend des applications des utilisateurs.

L'utilisateur **n'a pas besoin de savoir** si les données sont répliquées ou non. Il envoie ses requêtes comme s'il existe une seule copie des données.

3/ Transparence de fragmentation

Dans certains cas, et pour des raisons de performance, on aura besoin de **fragmenter** une table de la base de données et placer chaque fragment sur un site différent.

La fragmentation permet aussi de réduire les **inconvénients** de la **réplication**.

Cette opération de fragmentation doit être faite d'une manière **transparente** à l'utilisateur. Il utilise la table originale comme si elle n'est pas fragmentée.

3/ Transparence de fragmentation (suite)

Il existe deux types de fragmentation:

- **Fragmentation horizontale:** La table est partitionnée en plusieurs sous-tables contenants chacune un sous-ensemble de lignes de la table originale.
- **Fragmentation verticale:** La table est partitionnée en plusieurs sous-tables qui sont définies sur un sous-ensemble d'attributs (de colonnes) de la table originale.

CHAPITRE II
STRUCTURE DE BDD DISTRIBUÉES

I. Introduction
II. Fragmentation
III. Allocation de ressources

CHAPITRE II
STRUCTURE DE BDD DISTRIBUÉES

I. Introduction

L'organisation des systèmes distribués tourne autours de trois axes:

- Niveau de partage:
  - Pas de partage;
  - Partage de données;
  - Partage de données et des applications.
- Type d'accès:
  - Accès statique (localisation des données connue dès le début);
  - Accès dynamique (localisation des données inconnue).

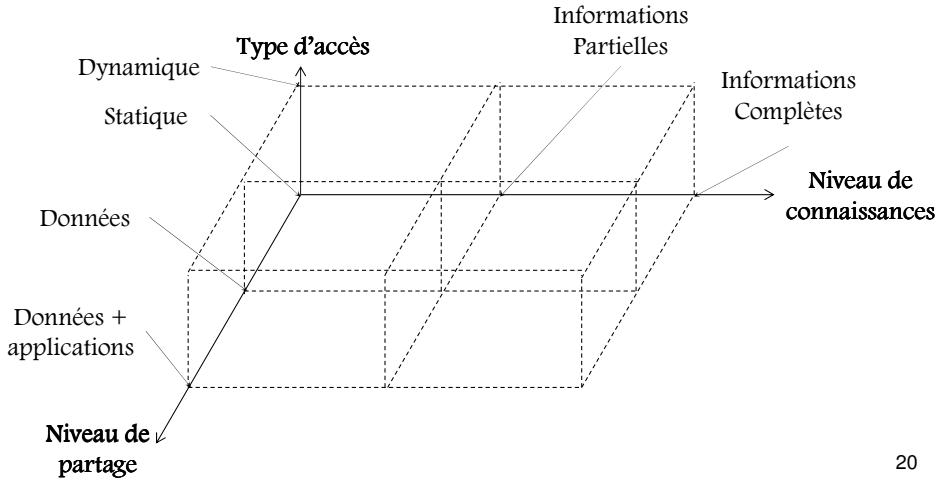
CHAPITRE II
STRUCTURE DE BDD DISTRIBUÉES

I. Introduction (suite)

- Niveau de connaissances sur le comportement (type) d'accès:
  - Pas de connaissance préalable sur le type d'accès (le design ou la conception de la base est très difficile);
  - Information partielles sur le type d'accès;
  - Information complètes sur le type d'accès (Le cas le plus pratique).

CHAPITRE II
STRUCTURE DE BDD DISTRIBUÉES

I. Introduction (suite)



I. Introduction (suite)

Avec cette organisation, et comme indiqué dans le chapitre précédent concernant la distribution des relations de la base, au lieu de distribuer les relations d'une même base sur plusieurs sites, il est préférable de **distribuer** des **fragments** d'une même relation sur plusieurs sites.

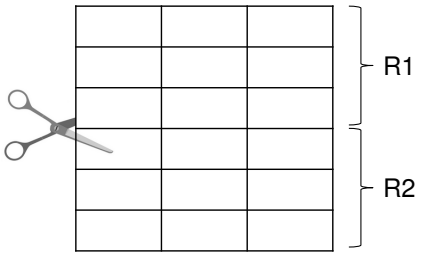
Ainsi, ce processus est divisé en deux parties: **fragmentation** et **allocation**.

II. Fragmentation

Le processus de fragmentation peut se faire sous différentes stratégies: horizontale, verticale ou hybride.

II.1. Fragmentation horizontale

Ce type de fragmentation permet de découper une relation en sous-relations contenants des sous-ensembles des tuples de la relation mère.



II.1. Fragmentation horizontale (suite)

La fragmentation horizontale est basée sur l'opération de **sélection** qui utilise un **prédicat** (condition).

La **reconstitution** de la relation se fait par **l'union** des fragments.

Nous avons besoin de définir deux notions: **prédicat simple**, **prédicat minterm**.

II.1. Fragmentation horizontale (suite)

<u>Prédicat simple</u>

Soit une relation $R(A_1,A_2,\ldots,A_n)$, tel que A_i est un attribut de R défini sur un domaine D_i. Un prédicat simple p_j est défini sur R comme suit:

$$p_j : A_i \ \theta \ val$$

tel que $\theta \in \{=, <, >, \neq, \leq, \geq\}$ et $val \in D_i$

On note P_r l'ensemble de tous les prédicats simples définis sur R.

II.1. Fragmentation horizontale (suite)

Exp.

Soit la relation `Prj` de l'exemple précédent, on peut définir quelques prédicats simples:

`nomPrj = "Construction"`

`budget ≤ 100000`

II.1. Fragmentation horizontale (suite)

Prédicat minterm

Un prédicat minterm *m* est défini par une **conjonction** de prédicats simples.

Une conjonction est une expression booléenne contenant des prédicats simples séparés par des `AND`. On peut utiliser aussi le `NOT`.

$$m = \bigwedge_{p_j \in P_r} p_j^*$$

tel que: $p_j^* = p_j$ ou $p_j^* = \neg p_j$

II.1. Fragmentation horizontale (suite)

Exp. Soit les minterms suivants:

`m1 : poste = "Ing" ∧ salaire ≤ 50000`

`m2 : poste = "Ing" ∧ salaire > 50000`

`m3 : ¬(poste = "Ing") ∧ salaire ≤ 50000`

`m4 : ¬(poste = "Ing") ∧ salaire > 50000`

`m5 : poste = "Comptable" ∧ salaire ≤ 50000`

`m6 : poste = "Comptable" ∧ salaire > 30000`

II.1. Fragmentation horizontale (suite)

Il existe deux types de fragmentation horizontale: primaire et dérivée.

II.1.1. Fragmentation horizontale primaire

La fragmentation horizontale est basée sur l'opération de sélection qui utilise un prédicat minterm.

$$R_i = \sigma_{m_i}(R)$$

tel que:

R_i est un fragment de la relation R

m_i est un minterm

σ est l'opération de sélection.

II.1.1. Fragmentation horizontale primaire (suite)

Exp. 1

Fragmentation de la relation `Prj` en fragmentation horizontale peut être faite comme suit:

$Prj_1 = \sigma_{budget \leq 100000}(Prj)$

$Prj_2 = \sigma_{budget > 100000}(Prj)$

Cet exemple présente un des **problèmes** de la fragmentation horizontale. Si le domaine des attributs est **continu** et **infini** il sera difficile de définir le nombre **maximal** de fragmentations **possibles**. On ne peut pas partager le domaine en un nombre fini de sous-domaines.

II.1.1. Fragmentation horizontale primaire (suite)

Exp. 2

On peut fragmenter la relation `Prj` en se basant sur la localisation:

$Prj_1 = \sigma_{ville="Alger"}(Prj)$

$Prj_2 = \sigma_{ville="Batna"}(Prj)$

$Prj_3 = \sigma_{ville="Oran"}(Prj)$

Dans ce cas, le nombre **maximal** de fragmentations **possibles** est fixe.

II.1.1. Fragmentation horizontale primaire (suite)

Pour éviter les problèmes de fragmentation horizontale, nous avons besoin de **bien choisir** les prédicats simples qui seront à la base des prédicats minterm permettant de fragmenter une relation.

D'une manière générale, cet ensemble doit **contenir seulement** les prédicats avec des **attributs** et **conditions** utilisés par les **applications**.

Cet ensemble de prédicats simples doit être **complet** et **minimal**. Pour cela, nous avons besoin de définir deux aspects importants: **complétude**, **minimalité**.

II.1.1. Fragmentation horizontale primaire (suite)

<u>Complétude</u>

Un ensemble de prédicats simples P_r est dit **complet** ssi la **probabilité d'accès** par chaque application à n'importe quel tuple de n'importe quel (de chaque) fragment-minterm est **égale**.

Autrement dit:

La **complétude** garantit que les **tuples** de chaque fragment-minterm soient **accédés** par **toutes les applications** avec la **même probabilité**.

II.1.1. Fragmentation horizontale primaire (suite)

Exp.

Soit l'ensemble P_r = {ville="Alger", ville="Batna", ville="Oran"} permettant de fragmenter la relation `Prj` en Prj_1, Prj_2, Prj_3 comme dans l'exemple précèdent.

S'il y a une seule application qui cherche à accéder aux tuples des fragments selon leurs locations, l'ensemble P_r est complet.

II.1.1. Fragmentation horizontale primaire (suite)

Complétude (suite)

Exp. (suite)

Par contre, s'il y a une 2ème application qui cherche à accéder aux projets qui ont un budget inférieur à 100000, dans ce cas, P_r n'est **plus complet**.

Cause: Les tuples des fragments qui ont un budget < 100000 ont une probabilité d'accès supérieur aux tuples qui ont un budget > 100000.

II.1.1. Fragmentation horizontale primaire (suite)

Complétude (suite)

Exp. (suite)

Solution: Pour rendre l'ensemble P_r complet, nous devons ajouter le prédicat (budget<100000) et sa négation (budget≥100000) comme suit:

P_r = {ville="Alger", ville="Batna", ville="Oran", budget<100000 , budget≥100000}

II.1.1. Fragmentation horizontale primaire (suite)

Solution: (suite)

Les minterms à utiliser dans la fragmentation sont:

m1: ville = "Alger" ∧ budget < 100000

m2: ville = "Batna " ∧ budget < 100000

m3: ville = "Oran " ∧ budget < 100000

m4: ville = "Alger" ∧ budget ≥ 100000

m5: ville = "Batna " ∧ budget ≥ 100000

m6: ville = "Oran " ∧ budget ≥ 100000

Les deux applications accèdent aux tuples de chaque fragment-minterm avec une probabilité égale (0 ou 1/nb_tuples_fragment)

II.1.1. Fragmentation horizontale primaire (suite)

Minimalité

- Définition 1 -

Un ensemble de prédicats simples est dit **minimal** si tous les prédicats sont **pertinents**.

Un prédicat est dit **pertinent** s'il cause la fragmentation d'un fragment F en F_1 et F_2 et qu'il **existe au-moins** une application qui accède aux deux fragments **différemment** (individuellement) (c-à-d F_1 et F_2 sont disjoints).

II.1.1. Fragmentation horizontale primaire (suite)

Minimalité

Autrement dit:

- Définition 2 -

La **minimalité** d'un ensemble de prédicats exprime que si un prédicat dans cet ensemble permet de partitionner un fragment F en deux fragments F_1 et F_2 , alors il existe au-moins une application accède à F_1 et F_2 individuellement. Cela veut dire que le prédicat est pertinent dans la détermination de F_1 et F_2 .

II.1.1. Fragmentation horizontale primaire (suite)

Minimalité

Autrement dit:

- Définition 3 -

Le but de la **minimalité** est d'assurer qu'il y a une cause pour effectuer la fragmentation (F_1, F_2) par un prédicats.

Il existe au-moins une application qui n'a besoin que d'un seul fragment (parmi les deux F1 et F2) à un moment donné.

II.1.1. Fragmentation horizontale primaire (suite)

Exp.

Dans l'exemple précèdent, l'ensemble de prédicats P_r = {ville="Alger", ville="Batna", ville="Oran", budget<100000 , budget≥100000} est aussi minimal parce que tous les prédicats sont pertinents.

II.1.1. Fragmentation horizontale primaire (suite)

Exp.

Soit la relation `Aff(idEmp, idPrj, resp, dur)` et soit une application qui s'intéresse aux tuples du projet n°5. L'ensemble P_r = {idPrj=5, idPrj≠5} est minimal parce que le prédicat à l'intérieur est pertinent.

S'il existe une 2ème application qui s'intéresse aux tuples dont la durée > 12, dans ce cas, l'application accède aux tuples des 2 fragments en même temps, donc P_r n'est pas minimal.

II.1.1. Fragmentation horizontale primaire (suite)

<u>Remarque:</u>

La meilleure fragmentation est celle qui utilise un ensemble de prédicats simples P_r complet et minimal.

II.1.2. Fragmentation horizontale dérivée

La fragmentation horizontale dérivée d'une relation (R) est basée sur des fragments horizontaux d'une autre relation (S) à condition qu'il y est un lien entre R et S.

Les nouveaux fragments sont définis par semi-jointure.

<u>Définition</u>

La **semi-jointure** entre deux relations R et S ($R \propto S$) permet de garder les tuples de R qui ont relation avec des tuples de S.

II.1.2. Fragmentation horizontale dérivée (suite)

Exp.

Dans l'exemple précédent, nous voulons grouper les employés de la relation `Emp(idEmp, nomEmp, poste)` en 2 groupes selon leurs salaires. Un groupe pour ceux qui ont un salaire inférieur à 30000, et les autres dans le deuxième groupe.

On fragmente la relation `Sal(poste, salaire)` en deux fragments:

$Sal1=\sigma_{salaire<30000}(Sal)$

$Sal2=\sigma_{salaire\geq30000}(Sal)$

II.1.2. Fragmentation horizontale dérivée (suite)

Exp. (suite)

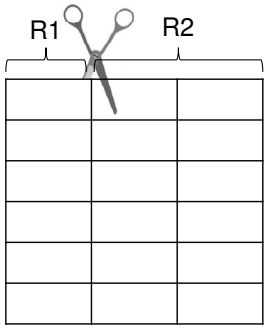
Les deux fragments contenant les deux groupes de salariés sont définis par semi-jointure sur la colonne `'poste'` comme suit:

`Emp1 = Emp ∝ Sal1`

`Emp2 = Emp ∝ Sal2`

II.2. Fragmentation verticale

La fragmentation verticale permet de découper une relation R en plusieurs relations. Chaque fragment contient un sous-ensemble d'attributs ainsi que la clé primaire de R.



II.2. Fragmentation verticale (suite)

Exp.

Soient deux applications qui utilisent la relation `Emp(idEmp, nomEmp, poste)`. La première utilise les noms des employés, et l'autre utilise les postes des employés.

On cherche à fragmenter la relation pour que chaque application accède seulement au fragment qui l'intéresse.

Pour cela, on utilise la projection:

$Emp1 = \pi_{[idEmp,\ nomEmp]}(Emp)$

$Emp2 = \pi_{[idEmp,\ poste]}(Emp)$

II.2. Fragmentation verticale (suite)

Exp. (suite)

La reconstruction de la relation d'origine se fait par la jointure:

$Emp = Emp1 \bowtie Emp2$

II.3. Fragmentation hybride

La fragmentation hybride est une combinaison des deux fragmentations précédentes.

Exp.

Soient trois applications qui utilisent les relations

`Emp(idEmp, nomEmp, poste)` et

`Personne(id, dateNaiss, adr, tel).`

La première utilise les noms des employés ≤ 30ans, la deuxième utilise les noms des employés > 30ans et l'autre utilise les postes des employés.

II.3. Fragmentation hybride (suite)

Exp. (suite)

On cherche à fragmenter la relation `Emp` pour que chaque application accède seulement au fragment qui l'intéresse.

$Personne1 = \sigma_{(dateNaiss<dateActuelle - 30ans)}(Personne)$

$Personne2 = \sigma_{(dateNaiss\geq dateActuelle - 30ans)}(Personne)$

$Emp1 = \pi_{[idEmp, nomEmp]}\ Emp\ \alpha\ Personne1$

$Emp2 = \pi_{[idEmp, nomEmp]}\ Emp\ \alpha\ Personne2$

$Emp3 = \pi_{[idEmp, poste]}\ Emp$

$Emp4 = \pi_{[idEmp, adr, tel]}\ Emp \infty Personne$

$Emp5 = Personne - (Personne\ \alpha\ Emp)$

III. Allocation de ressources

Soit un ensemble de fragments $F=\{F_1, F_2, \ldots, F_n\}$ distribués sur un ensemble de sites $S=\{S_1, S_2, \ldots, S_m\}$ et un ensemble de requêtes $Q=\{q_1, q_2, \ldots, q_p\}$ utilisant les fragments F situés sur les sites S.

Le problème d'allocation de ressources se résume dans la recherche d'une distribution optimale des F sur les S.

III. Allocation de ressources (suite)

Pour assurer l'optimalité, nous devons respecter les deux conditions:

1. Le coût minimal

Les coûts traités dans cette condition sont:

- Le coût de stockage de chaque fragment F_i sur le site S_j;
- Le coût de consultation de F_i localisé sur S_j;
- Le coût de mise à jour de F_i sur tous les sites où il est stocké;
- Le coût de communication.

III. Allocation de ressources (suite)

2. La performance

Dans cette condition nous essayons de:

- Minimiser le temps de réponse;
- Maximiser le débit de chaque site.

Remarque:

Suite à la complexité du problème, on ne peut pas satisfaire toutes ces conditions. Les modèles existants essayent de satisfaire les conditions qui les intéresses.

III. Allocation de ressources (suite)

Pour simplifier la modélisation de ce problème, nous introduisons quelques définitions sur un fragment F_k.

Soit $R=\{r_1, r_2, \ldots, r_m\}$ et $U=\{u_1, u_2, \ldots, u_m\}$

tel que:

r_i: le trafic de lecture à partir de F_k généré sur le site S_i.

u_i: le trafic de mise à jour dans F_k généré sur le site S_i.

III. Allocation de ressources (suite)

Pour le coût de communication nous définissons:

$$C(R) = \begin{bmatrix} 0 & c_{12} & \cdots & c_{1,m-1} & c_{1m} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{m,m-1} & 0 \end{bmatrix}$$

$$C'(U) = \begin{bmatrix} 0 & c'_{12} & \cdots & c'_{1,m-1} & c'_{1m} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ c'_{m1} & c'_{m2} & \cdots & c'_{m,m-1} & 0 \end{bmatrix}$$

tel que: c_{ij} représente le coût de communication pour une requête de consultation entre les sites S_i et S_j.

c'_{ij} représente le coût de communication pour une requête de mise à jour entre les sites S_i et S_j.

III. Allocation de ressources (suite)

Pour le coût de stockage nous définissions:

$$D = \{d_1, d_2, \ldots, d_m\}$$

tel que: d_i représente le coût de stockage du fragment F_k sur le site S_i.

Définissons aussi une variable de décision:

$$x_j = \begin{cases} 1 & \text{si le fragment } F_k \text{ est assigné au site } S_j \\ 0 & \text{sinon} \end{cases}$$

III. Allocation de ressources (suite)

Pour minimiser le coût d'allocation de ressources du fragment F_k qui sera placé sur les sites S_j tel que $x_j=1$, nous allons chercher [ref. 1]:

$$\min\left[\sum_{i=1}^{m}\left(\sum_{j|x_j=1}\left(u_j\cdot c'_{ij}+r_j\cdot\min_{j|x_j=1}c_{ij}\right)\right)+\sum_{j|x_j=1}d_j\right]$$

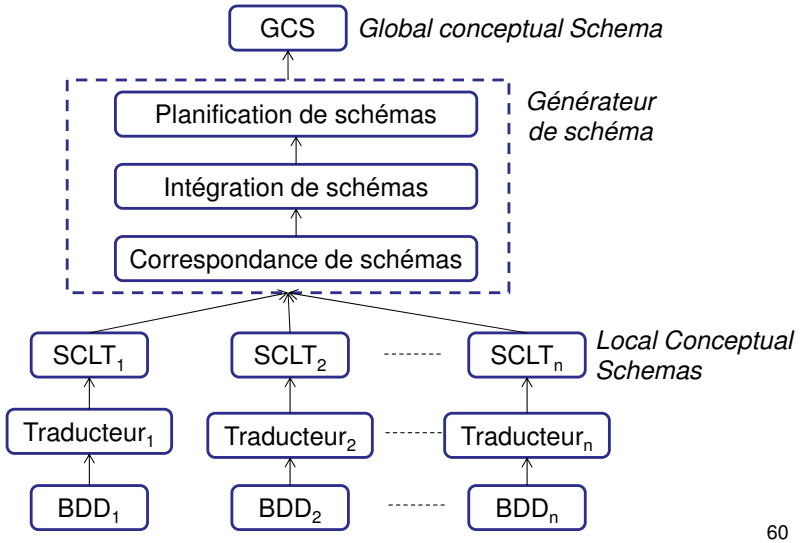
I. Méthodologie de la conception Bottom-Up
II. Correspondance de schémas
III. Intégration de schémas
IV. Planification de schémas

I. Méthodologie de la conception Bottom-Up

Dans cette partie nous allons nous intéressés à la conception Bottom-Up.

On considère qu'il existe plusieurs bases de données **hétérogènes**, et on cherche à les **intégrer** dans une seule base de données distribuée.

Le processus d'intégration est divisé en plusieurs étapes:



Comme indiqué sur la figure, le processus de génération de schémas est composé de 3 étapes:

I.1. Correspondance de schémas

Permet de déterminer les correspondances **syntaxiques** et **sémantiques** entre les éléments traduits.

Si le schéma conceptuel global (GCS) est **défini**, les éléments traduits doivent **lui correspondre**, sinon on établi une correspondance entre les schémas conceptuels locaux traduits (SCLT*i*).

I.1. Correspondance de schémas (suite)

Nous expliquons la problématique par un exemple.

Exp. Soit la table:

```
Prj (idPrj, nomPrj, designation, budget, ville);
```

Dans cette table, l'attribut `budget` peut être spécifié en DA pour les projets réalisés en Algérie, ou en Euros pour les projets réalisés à l'étranger.

La correspondance doit spécifier une valeur en fonction de l'autre en appliquant une fonction de change.

I.1. Correspondance de schémas (suite)

Pour établir une correspondance entre deux éléments, on défini ce qui suit:

- Une correspondance est définie sous forme d'une règle (r);
- Une règle identifie une correspondance (c) entre deux éléments;
- Un prédicat (p) indique quand la règle (r) sera appliquée;
- Une valeur de similarité (s) entre les deux éléments.

I.1. Correspondance de schémas (suite)

Une correspondance (c) définie (calcule) un élément en **fonction** d'un autre soit par **égalité** (=) s'ils sont similaires, ou par une **expression**.

Exp:

Dans l'exemple précédent, nous devons convertir tous les budget en une seule monnaie (en euro).

Pour cela, nous devons multiplier le budget des projets algériens par le taux de change τ:

```
UPDATE Prj SET budget = budget*τ
WHERE ville IN {'Alger', 'Batna', …}
```

I.1. Correspondance de schémas (suite)

Un prédicat (p) est une **condition**. Quand elle est **vrai** on applique la **règle**, sinon la règle sera ignorée.

Exp:

Dans l'exemple précédent, le prédicat p doit spécifier quand la règle doit être appliquée, et cela peut se faire en fonction de la localisation (l'attribut `ville`) du projet.

I.1. Correspondance de schémas (suite)

La valeur de similarité (s) est une valeur réelle ∈ [0,1]. Elle peut être calculée ou spécifiée.

Si les deux éléments comparés sont similaire (s=1), la correspondance se fera par égalité (=), sinon la correspondance se fait par modification d'au-moins un des deux éléments.

Donc une **règle** est un triplet **<c, p, s>**

Exp: Dans l'exemple précédent:

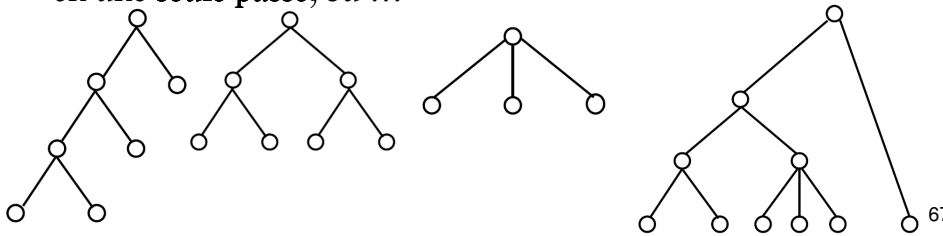
```
< c:(budget = budget*τ), p:(ville IN {'Alger', 'Batna', …} ), s:(sim(monaie du budget, Euro)=0.9) >
```


I.2. Intégration de schémas

Dans le processus d'intégration de schémas, la présence **humaine** est indispensable.

Après la correspondance entre les différents schémas locaux, la prochaine étape est de **créer** un schéma global.

L'intégration des schémas locaux peut se faire **deux à deux**, en une **seule passe**, ou …



I.3. Planification de schémas

Ce processus permet de **placer** les données des bases locales dans la base globale.

Le processus de planification de schémas est divisé en deux étapes: **création** de planification, **maintenance** de planification.

Le processus de **création** de planification consiste à créer des **requêtes** permettant de placer les données provenant des schémas locaux dans le schéma global.

Le processus de **maintenance** de planification permet de **détecter** et **corriger** les incohérences générées par l'évolution du schéma.

Exp.

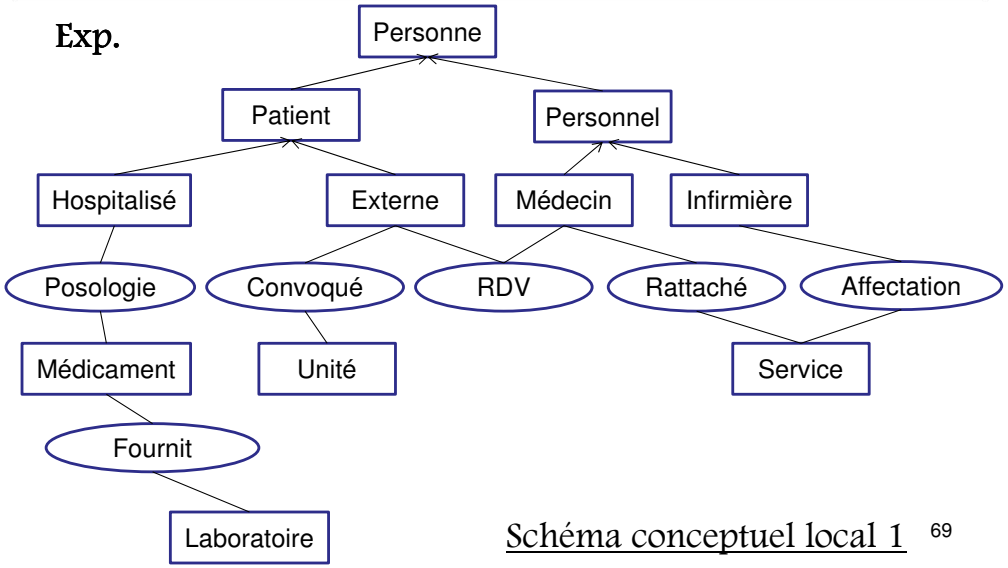


Schéma conceptuel local 1

Exp. (suite)

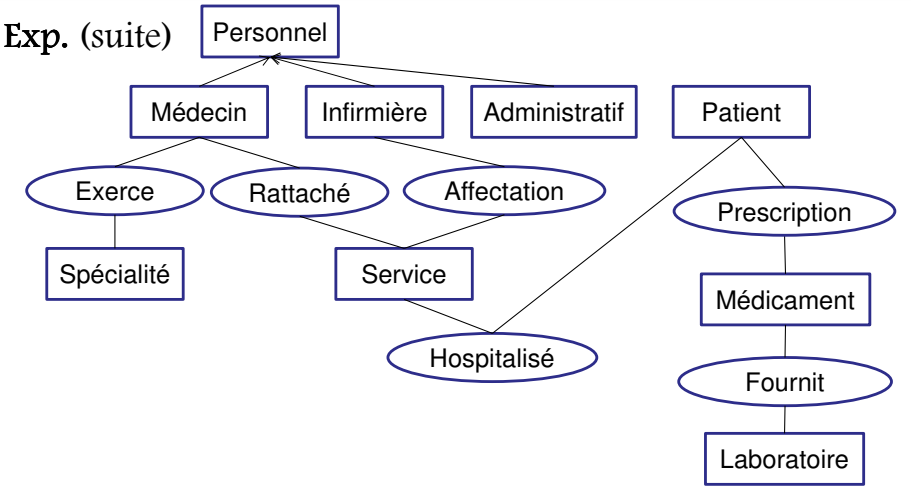
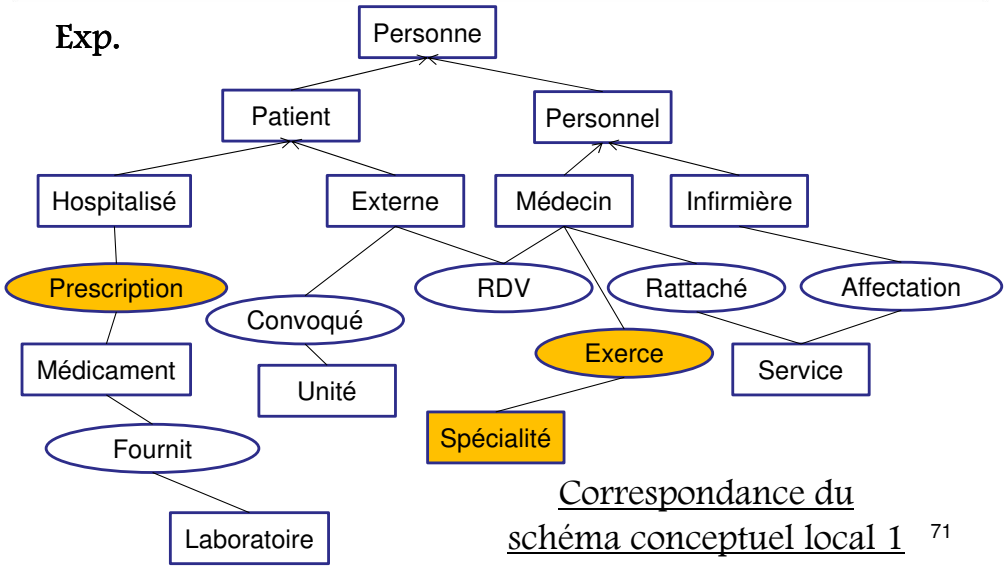


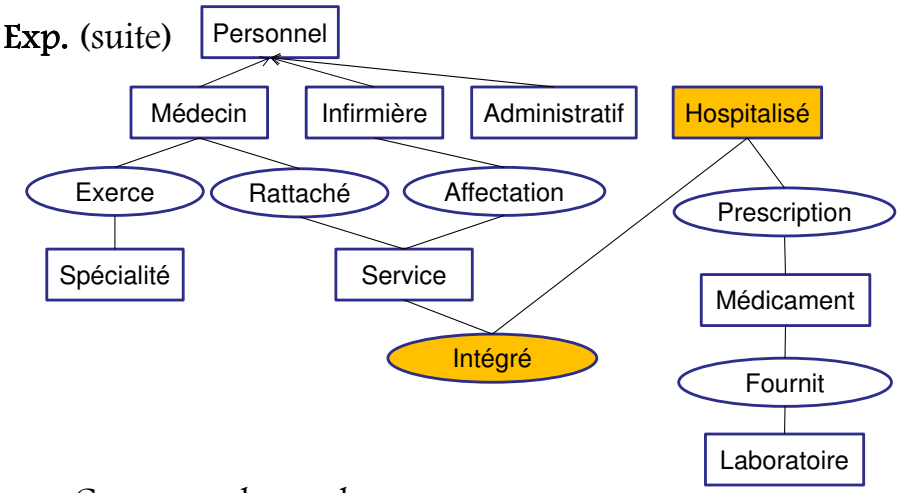
Schéma conceptuel local 2

Exp.



Correspondance du schéma conceptuel local 1

Exp. (suite)



Correspondance du schéma conceptuel local 2

CHAPITRE III
INTÉGRATION DE BASES DE DONNÉES

Exp.

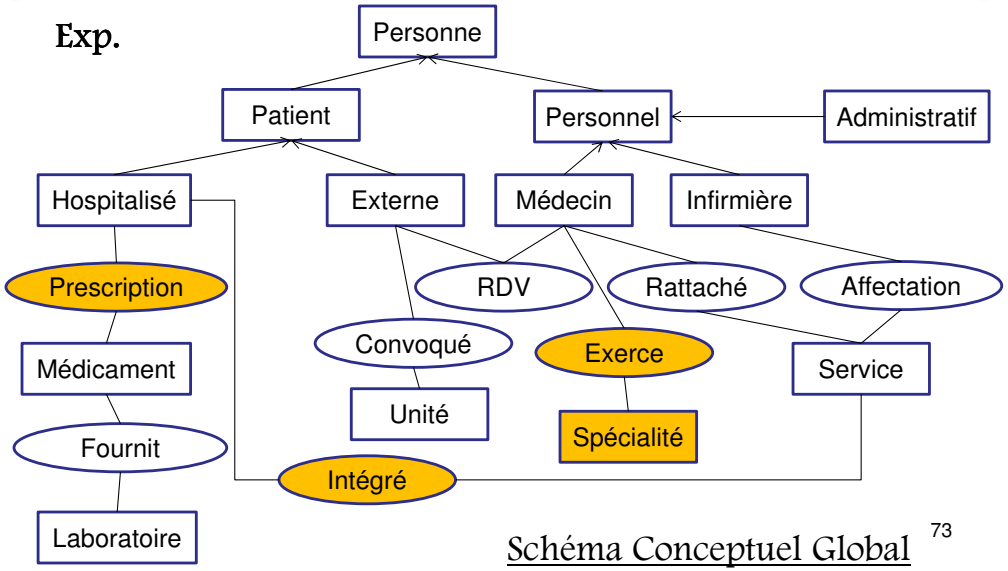


Schéma Conceptuel Global

CHAPITRE IV
CONTRÔLE D'ACCÈS ET DE DONNÉES

I. Gestion de vues
II. Sécurité de données
III. Contrôle d'intégrité sémantique

CHAPITRE IV
CONTRÔLE D'ACCÈS ET DE DONNÉES

Parmi les besoins importants dans les DBMS distribuées on trouve le **contrôle de données sémantique**.

Ce contrôle de données sémantique inclus la gestion des vues, la sécurité des données et le contrôle d'intégrité sémantique.

CHAPITRE IV
CONTRÔLE D'ACCÈS ET DE DONNÉES

I. Gestion des vues

Une vue est une relation virtuelle permettant de cacher une partie des données.

I.1. Les vues dans les SGBD centralisés

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

```
CREATE VIEW <nom de vue> [(liste d'attributs)]
AS <requête>
```

I.1. Les vues dans les SGBD centralisés (suite)

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.

La **suppression** de la vue se fait par:

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


I.1. Les vues dans les SGBD centralisés (suite)

Exp. 1.

Pour créer la vue des employés ingénieurs (`EmpIng`) à partir de la relation `Emp(idEmp, nomEmp, poste)` on utilise:

```
CREATE VIEW EmpIng (idEmp, nomEmp)
AS SELECT idEmp, nomEmp
FROM Emp
WHERE poste = 'Ing'
```


I.1. Les vues dans les SGBD centralisés (suite)

Exp. 2.

Donner les noms des ingénieurs avec les numéros de leurs projets.

```
Emp(idEmp, nomEmp, poste)
Aff(idEmp, idPrj, resp, dur)

CREATE VIEW EmpIng (idEmp, nomEmp)
AS SELECT nomEmp, idPrj
FROM Emp, Aff
WHERE Emp.idEmp = Aff.idEmp
```


I.2. Les vues dans les SGBD distribuées

Le nom et la requête d'une vue sont stockés dans un **catalogue**.

La **requête** peut être appliquée à des **fragments** stockés sur des sites différents.

Problème 1:

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

I.2. Les vues dans les SGBD distribuées (suite)

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.

I.2. Les vues dans les SGBD distribuées (suite)

Problème 2:

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

Le DBMS doit **maintenir** la vue matérialisée à jour.

L'administrateur doit savoir **quand** et **comment** mettre à jour la vue matérialisée.

I.2. Les vues dans les SGBD distribuées (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**.

I.2. Les vues dans les SGBD distribuées (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).

I.2. Les vues dans les SGBD distribuées (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)

I.2. Les vues dans les SGBD distribuées (suite)

Comment mettre à jour la vue matérialisée? (suite)

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

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

Exp.

Dans l'exemple précédent,

```
Emp (idEmp, nomEmp, poste)
Aff (idEmp, idPrj, resp, dur)
```

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

I.2. Les vues dans les SGBD distribuées (suite)

Exp. (suite)

```
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.

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

I.2. Les vues dans les SGBD distribuées (suite)

Exp. (suite)

```
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)
```

II. Sécurité des données

La sécurité des données inclue deux aspects: **Protection des données** et **contrôle d'accès**.

II.1. Protection des données

En général, l'approche principale de protection de données est le **cryptage** des données.

Cet aspect ne concerne pas beaucoup des systèmes de bases de données, mais plutôt le domaine de cryptographie.

Nous allons nous concentrer sur le deuxième aspect.

II.2. Contrôle d'accès

Il existe deux approches principales: contrôle d'accès **discrétionnaire** (contrôle d'autorisation), contrôle d'accès **multi-niveaux**.

II.2.1. Contrôle d'accès discrétionnaire

Cette approche définie des **droits d'accès** basés sur les utilisateurs.

Une autorisation est un triplet **<sujet s, type d'opération t, définition d'objet o>** qui donne le droit au sujet 's' d'établir les opérations de type 't' sur l'objet 'o'.

II.2.1. Contrôle d'accès discrétionnaire (suite)

La définition d'un **sujet** se fait par une paire **<user, pass>** ou **public** pour tout le monde.

Le type d'opération peut être: **SELECT**, **INSERT**, **UPDATE** ou **DELETE, REFERENCES, ALTER, INDEX, ALL**.

Un droit affecté à un sujet peut être: **GRANT** (accorder) ou **REVOKE** (retirer).

```
GRANT (operation) ON (objet) TO (sujet)
REVOKE (operation) FROM (object) TO (sujet)
```


II.2.1. Contrôle d'accès discrétionnaire (suite)

Seuls les utilisateurs de type '**administrateur**' peuvent utiliser les opérations GRANT et REVOKE.

Un **utilisateur** possède **tous** les **droits** sur les objets qu'il a créer et peut utiliser les opérations (GRANT, REVOKE) sur ces objets.

Un utilisateur qui **reçoit les droits** sur un objet, peut ensuite appliquer les opérations (GRANT, REVOKE) sur cet objet.

II.2.1. Contrôle d'accès discrétionnaire (suite)

Récursivité:

Soit (A) qui donne un droit de type 't' sur un objet 'o' à (B) qui donne à son tour ce droit à (C).

Si (A) retire ce droit à (B), le système DBMS doit **retirer** ce droit aussi à (C) d'une manière **récursive**.

Donc le DBMS doit **garder l'hiérarchie** de distribution des droits d'accès. En général, on utilise une **matrice d'autorisations**, tel que les lignes représentent les sujets, une colonne représente un objet et une case représente une autorisation <s,t=*,o>

II.2.1. Contrôle d'accès discrétionnaire (suite)

Exp. Matrice d'autorisations

	Emp	nomEmp	Aff
User1	UPDATE	UPDATE	UPDATE
User2	SELECT	SELECT	SELECT WHERE resp≠'Manager'
user3	NONE	SELECT	NONE

II.2.2. Contrôle d'accès Multi-Niveaux (Mandatory/Multilevel)

Cette approche est basée sur la **limitation** d'accès aux données **classifiées** aux utilisateurs **autorisés**.

Cette approche définie des **niveaux de sécurité** pour les sujets et les données:

Top Secret (TS) > Secret (S) >
Confidentiel (C) > Non-Classifié (NC)

tel que '**>**' veut dire "**plus sécurisé que**".

II.2.2. Contrôle d'accès Multi-Niveaux (suite)

L'accès en lecture et écriture pour un sujet est limité par deux règles:

1. Un sujet **U** est autorisé à lire un objet **O** si:
   niveau(U) ≥ niveau(O)
2. Un sujet **U** est autorisé à écrire un objet **O** si:
   niveau(U) ≤ niveau(O)

La règle 1 interdit un utilisateur de lire les données qui ont un niveau de sécurité supérieur.

La règle 2 interdit un utilisateur d'écrire son objet sécurisé dans un niveau moins sécurisé.

II.2.2. Contrôle d'accès Multi-Niveaux (suite)

Exp. Soit la relation `Prj(idPrj, nomPrj, designation, budget, ville)` avec la définition des niveaux de sécurité suivante:

idPrj	NivSec1	nomPrj	NivSec2	designation	NivSec3	budget	NivSec4	ville	NivSec5
P1	C	Prj1	C	d1	C	150000	C	Batna	C
P2	C	Prj2	C	d2	S	135000	TS	Alger	S
P3	S	Prj3	S	d3	S	250000	TS	Oran	S

II.2.2. Contrôle d'accès Multi-Niveaux (suite)

Exp. (suite) La **table** prend le **niveau le plus bas** de ses données qui est dans notre cas (C)

Mais un utilisateur de niveau (C) ne peux voir que les tuples suivants:

idPrj	NivSec1	nomPrj	NivSec2	designation	NivSec3	budget	NivSec4	ville	NivSec5
P1	C	Prj1	C	d1	C	150000	C	Batna	C
P2	C	Prj2	C	null	C	null	C	null	C

II.2.3. Contrôle d'accès distribué

Dans un DBMS **distribué** on peut **grouper** les utilisateurs en groupes et leurs définir des **niveaux de sécurité.**

Problème 1:

Les **utilisateurs** d'un groupe peuvent être localisés dans **différents sites**;

L'accès à un objet peut être **autorisé** à plusieurs groupes distribués;

Les informations sur les **niveaux de sécurité** des groupes doivent être stockées quelque part !!

II.2.3. Contrôle d'accès distribué (suite)

Le plus simple est de **dupliquer** les informations des groupes (droits d'accès…) sur **tous** les **sites**. Le **maintient** de toutes ces informations à jour est très **difficile**.

Solution 1:

Centraliser les informations des groupes sur un seul nœud et **appliquer** des droits d'accès en envoyant une **requête** à **distance** à ce **nœud**.

Solution 2:

Dupliquer les informations d'un groupe sur les sites **contenant** des objets utilisés par les membres de ce groupe.

II.2.3. Contrôle d'accès distribué (suite)

Problème 2:

Exp. Soit 2 sites tel que le Site1 possède le niveau de sécurité (S) et le Site2 avec le niveau (C).

Site1: Table `Notes(idEtd, Module, Note)` → niveau (S)

Site2: Table `Etudiants(idEtd, nom, prenom)` → niveau (C)

Un utilisateur avec un niveau (S) cherche les noms des étudiants ayant une note de 20 dans le module « BDD ».

Une **jointure** doit être établie entre les 2 tables, et les informations **secrètes** de la table `Notes` doivent être **envoyées** pour réaliser cette jointure.

II.2.3. Contrôle d'accès distribué (suite)

Problème 2: (suite)

Ce problème est appelé **canal caché (covert channel)** pour accéder à des données non-autorisées.

Solution:

Pour **éviter** le problème de canal caché on peut **dupliquer** une **partie** de la base pour qu'un site de niveau de sécurité **(k) contient** toutes les données **nécessaires** pour les utilisateurs de ce niveau.

Donc le site de l'utilisateur de niveau (S) doit contenir des copies des parties des tables (`Notes, Etudiants`) à utiliser dans la requête de jointure.

III. Contrôle d'intégrité sémantique

Exp.

Soit la base de données du premier exemple. La table `Emp` contient les employés avec les identifiants de 1 jusqu'à 50, la table `Prj` contient des projets avec les identifiants de 1 jusqu'à 10. On ajoute un nouveau tuple à la table `Aff(idEmp, idPrj, resp, dur)` comme suit:

```
INSERT INTO Aff VALUES (100, 10, 'Ing', 5)
```


III. Contrôle d'intégrité sémantique (suite)

Problème:

La base n'est pas **cohérente**, il y a un problème **sémantique** à cause de l'affectation de l'employé n°100, qui n'existe pas, à un projet.

Le **contrôle d'intégrité sémantique** assure la **cohérence (consistance)** de la base de données en **rejetant** les mises à jours qui affectent la cohérence de la base ou **activer** des **actions spécifiques** qui corrigent l'incohérence.

III. Contrôle d'intégrité sémantique (suite)

Pour assurer le contrôle d'intégrité sémantique nous allons voir des solution dans les DBMS **centralisés** puis dans les DBMS **distribués**.

III.1. Contrôle d'intégrité sémantique centralisée

Ce contrôle est composé de deux parties principales: spécification des contraintes d'intégrité, application de l'intégrité.

III.1.1. Spécification des contraintes d'intégrité

On peut classifier les contraintes d'intégrité en deux types: contraintes prédéfinies, contraintes pré-conditionnées.

Les **contraintes prédéfinies** utilisent des mots clés comme: attribut non-nul, clé unique, clé étrangère, dépendances fonctionnelles.

Exp. 1

L'identifiant d'un employé ne peut être nul:

```
idEmp NOT NULL IN Emp
```


III.1.1. Spécification des contraintes d'intégrité (suite)

Exp. 2

La paire <idEmp, idPrj> est une clé unique dans Aff:

```
(idEmp, idPrj) UNIQUE IN Aff
```

Exp. 3

L'identifiant du projet est une clé étrangère dans la table Aff, càd le projet utilisé dans une affectation doit exister dans la table Prj:

```
idPrj IN Aff REFERENCES idPrj IN Prj
```


III.1.1. Spécification des contraintes d'intégrité (suite)

Exp. 4

L'identifiant de l'employé définit fonctionnellement le nom de l'employé:

```
idEmp IN Emp DETERMINES nomEmp
```

III.1.1. Spécification des contraintes d'intégrité (suite)

Les **contraintes pré-conditionnées** utilisent le mot clé CHECK.

On utilise le mot clé CHECK de SQL:

```
ALTER TABLE table_name
ADD CONSTRAINT "constraint_name" CHECK (condition) ENABLE
```


III.1.1. Spécification des contraintes d'intégrité (suite)

Exp.

Le budget d'un projet doit être entre 50000 et 100000

```
ALTER TABLE Prj
ADD CONSTRAINT "PRJ_CHK1" CHECK (budget >= 50000 AND budget <= 1000000) ENABLE
```


III.1.2. Application de l'intégrité

Il existe deux méthodes de base permettant de rejeter les mises à jour incohérents:

* La **première méthode** est basée sur la **détection** des **incohérences** après l'exécution des mises à jour.

L'algorithme qui applique l'intégrité sémantique applique des testes après l'exécution des modifications pour détecter les incohérences, ces testes sont appelés **post-testes.**

III.1.2. Application de l'intégrité (suite)

* La **deuxième méthode** est basée sur la **prévention** des incohérences.

Les **modifications** ne seront pas appliquées que si elles ne causent pas d'incohérences.

L'algorithme qui applique l'intégrité sémantique vérifie si les contraintes d'intégrités seront satisfaites avant d'appliquer les modifications. Les testes appliqués sont appelés des **pré-testes.**

III.1.2. Application de l'intégrité (suite)

Une des méthodes préventives est **l'algorithme de modification de requêtes**. Il ajoute des testes à la requête elle-même.

Exp.

La requête qui incrémente le budget du projet prj1 de 20%:

```
UPDATE Prj
SET budget = budget*1.2
WHERE nomPrj = 'prj1'
```


III.1.2. Application de l'intégrité (suite)

Exp. (suite)

Cette requête sera transformée en ajoutant des pré-testes comme suit:

```
UPDATE Prj
SET budget = budget*1.2
WHERE nomPrj = 'prj1'
AND NEW.BUDGET ≥ 50000
AND NEW.BUDGET ≤ 100000
```


III.2. Contrôle d'intégrité sémantique distribuée

Le contrôle d'intégrité sémantique distribuée est une extension du contrôle d'intégrité sémantique centralisée. Il est composé de deux parties principales: spécification des contraintes d'intégrité distribuée, application de l'intégrité distribuée.

III.2.1. Spécification des contraintes d'intégrité distribuée

Les **contraintes** peuvent utiliser des données stockées sur **différents sites**. Une telle contrainte doit être stockée de manière à **minimiser** le coût de l'intégrité.

III.2.1.1. Contraintes individuelle

La définition de la contrainte d'intégrité est **envoyée** aux sites contenants les fragments de la relation utilisée dans la contrainte. La contrainte doit être compatible avec la relation aux niveaux: prédicat et données.

III.2.1.1. Contraintes individuelle (suite)

- **Compatibilité au niveau prédicat**

Une contrainte 'C' n'est **pas compatible** avec un prédicat de fragment 'p' **si 'C est vrai' implique que 'p est toujours faut' ($C \Rightarrow \neg p$)**, sinon 'C' est compatible avec 'p'.

Si on trouve une **incompatibilité** dans un seul site, la contrainte est **rejetée globalement**.

III.2.1.1. Contraintes individuelle (suite)

- **Compatibilité au niveau prédicat (suite)**

Exp.

Soit la relation Emp du premier exemple qui est fragmentée horizontalement sur trois sites en utilisant les prédicats:

p1: $0 \leq idEmp < 3$, p2: $3 \leq idEmp \leq 6$, p3: $idEmp > 6$

et soit la contrainte [C: $idEmp < 4$]

La contrainte 'C' n'est pas compatible avec 'p3' parce que **si C est vrai implique que p3 est faut**.

La contrainte 'C' est compatible avec 'p2' parce que **si C est vrai n'implique pas toujours que p2 est faut.**

III.2.1.1. Contraintes individuelle (suite)

- **Compatibilité au niveau prédicat (suite)**

Exp. (suite)

C	1	1	0	0
p	1	0	1	0
$C \Rightarrow p$	1	0	1	1

III.2.1.1. Contraintes individuelle (suite)

- **Compatibilité au niveau données**

La contrainte est testée avec les **instances** des fragments des sites utilisés. Si la contrainte n'est **pas satisfaite**, elle sera **rejetée globalement**.

Si la **compatibilité existe** aux deux niveaux, la contrainte sera **stockée** dans **tous les sites** utilisés.

III.2.1.1. Contraintes individuelle (suite)

- **Compatibilité au niveau données (suite)**

Exp.

Soit la contrainte de l'exemple précèdent: [C: idEmp < 4]

Et soit les fragments de la table `Emp (idEmp, nomEmp, poste)` sur 2 sites comme suit:

idEmp	nomEmp	poste
1	E1	P1
2	E2	P2

idEmp	nomEmp	poste
3	E1	P1
4	E2	P2

La contrainte 'C' est compatible avec le 1er site au niveau données, et pas avec le 2ème

CHAPITRE IV
CONTRÔLE D'ACCÈS ET DE DONNÉES

III.2.2. Application de l'intégrité distribuée

Le problème principale est de décider dans **quel site** doit-on appliquer la contrainte d'intégrité.

La contrainte est appliquée sur les sites contenant les fragments des relations utilisés dans la contrainte.