Bases de données relationnelles

Michel Rueher

Ouvrages de référence:

- *A First Course in Database Systems*, J. Ullman, J. Widom. 2nd edition, Prentice Hall, 2002
- *Database System Implementation*, H. Garcia-Molina, J. Ullman and J. Widom, Prentice Hall, 2001
- *Database Systems: The Complete Book* H. Garcia-Molina, J. Ullman and J. Widom, Prentice Hall, Prentice-Hall, 2002

Support disponible: ~rueher/BDR

1. Introduction

Plan

- BD … informellement
- Les grands principes
- Bases de données et SI
- Le modèle conceptuel
- Le modèle relationnel
- Passage du modèle conceptuel au modèle relationnel

BD … informellement

… difficile à définir

Un SGBD est

- Un outil pour vérifier, modifier et rechercher ***efficacement*** des données dans un grand ensemble
- Une interface entre les applications "utilisateur" et la mémoire secondaire

Un SGBD comprend

- Un système de gestion de fichiers (gestion du stockage physique)
- Un système interne (placement et accès aux données)
- Un système externe (langage de requête élaboré)

Les grands principes

Les 3 niveaux: physique, logique, externe

- Indépendance entre représentation physique et logique
- Offre différentes vues de la même structure

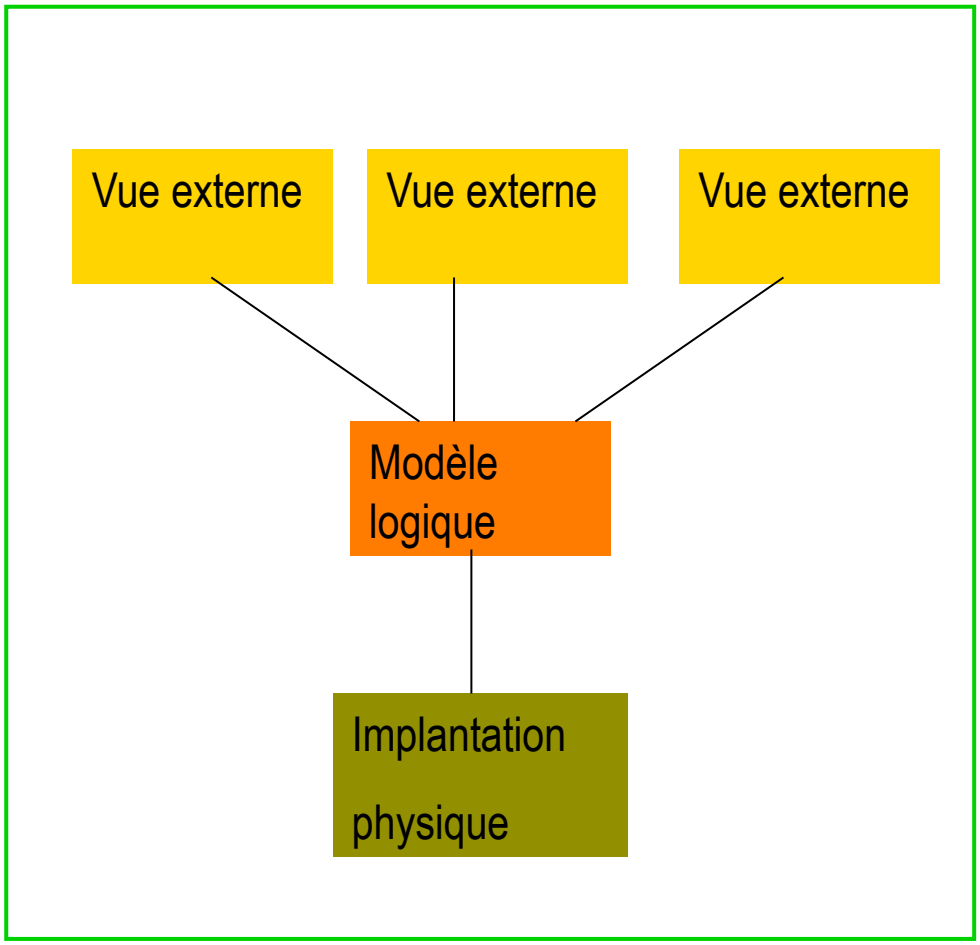
Modèle logique (modèle de données)

- Langage de définition des données (LDD)
- Langage de manipulation des données (LMD)
- Accès à des langages externes C,C++, JDBC

Les grands principes(suite)

Intérêt de l'abstraction des données ANSI- SPARC

- Totalement **déclaratif** (spécification abstraite)
- Aussi "riche" que possible
- Indépendant d'un langage de programmation
- Indépendant de l ’architecture
- Facilitant la conception des applications
- Facilitant l'expression de contraintes
- Faciliter la gestion de l'intégrité de la base



Les grands principes(suite)

Fonctionnalités

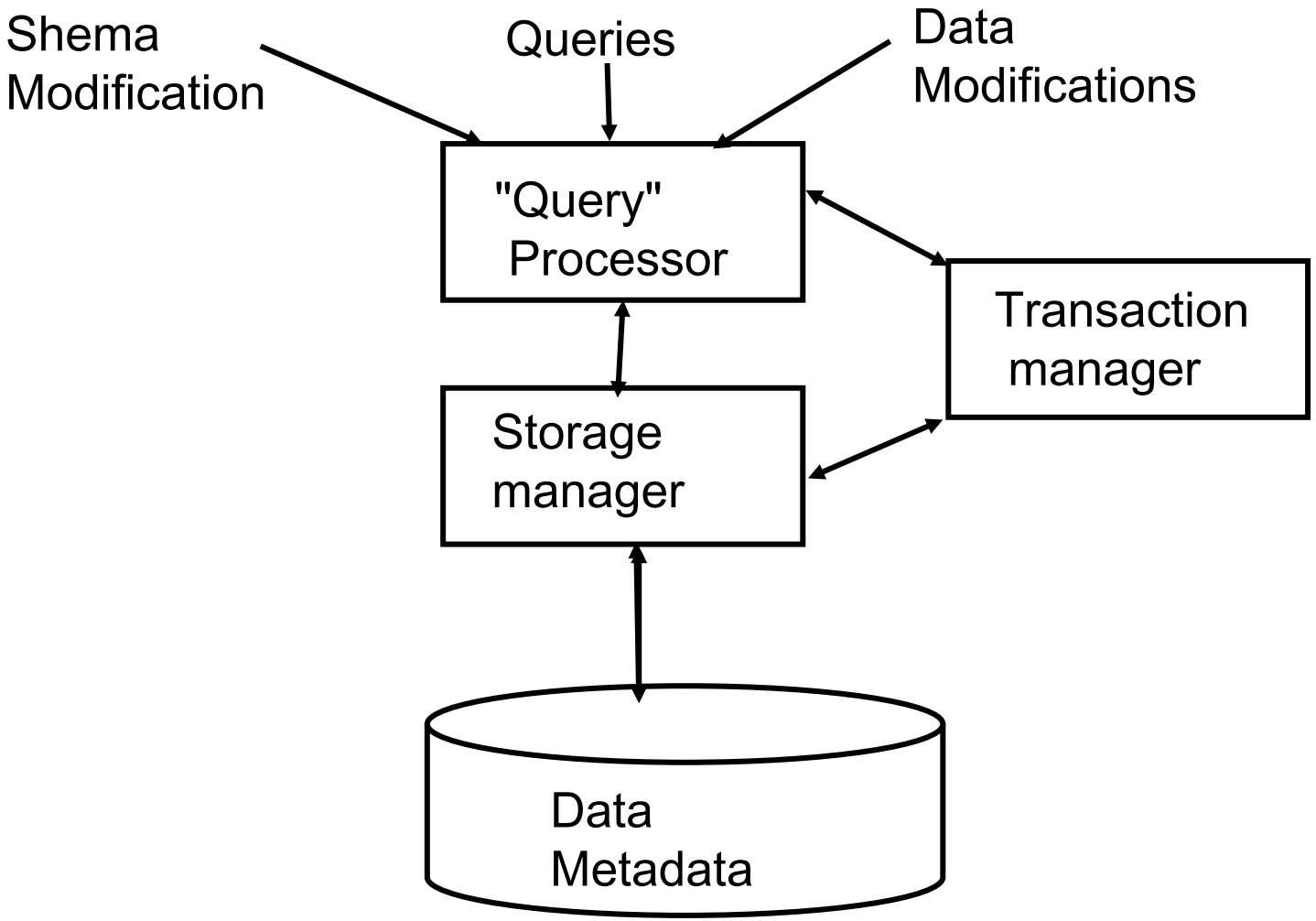
- Gestion du stockage secondaire
- Persistance
- Contrôle de concurrence
- Protection des données
- Interfaces homme / machine
- Gestion de données distribuées

Les grands principes(suite)

Complexité et diversité

- **Applications:** financières, inventaires, commerciales, ingénieries, CAO/FAO, cartographiques, documentaires, bibliothèques, …
- **Utilisateurs**: grand public, (minitel,web), secrétaire, cadre, programmeur d'application, administrateur de BD, …
- **Mode d'accès**: langage graphique ou non, interface spécifique, programme d'application, générateur de rapport
- **Modèle de données**: hiérarchique, réseau, relationnel, objet, …
- **Plate-forme** : différents matériels, supports de stockage, réseaux, systèmes d'exploitation,…

Architecture d'un DBMS

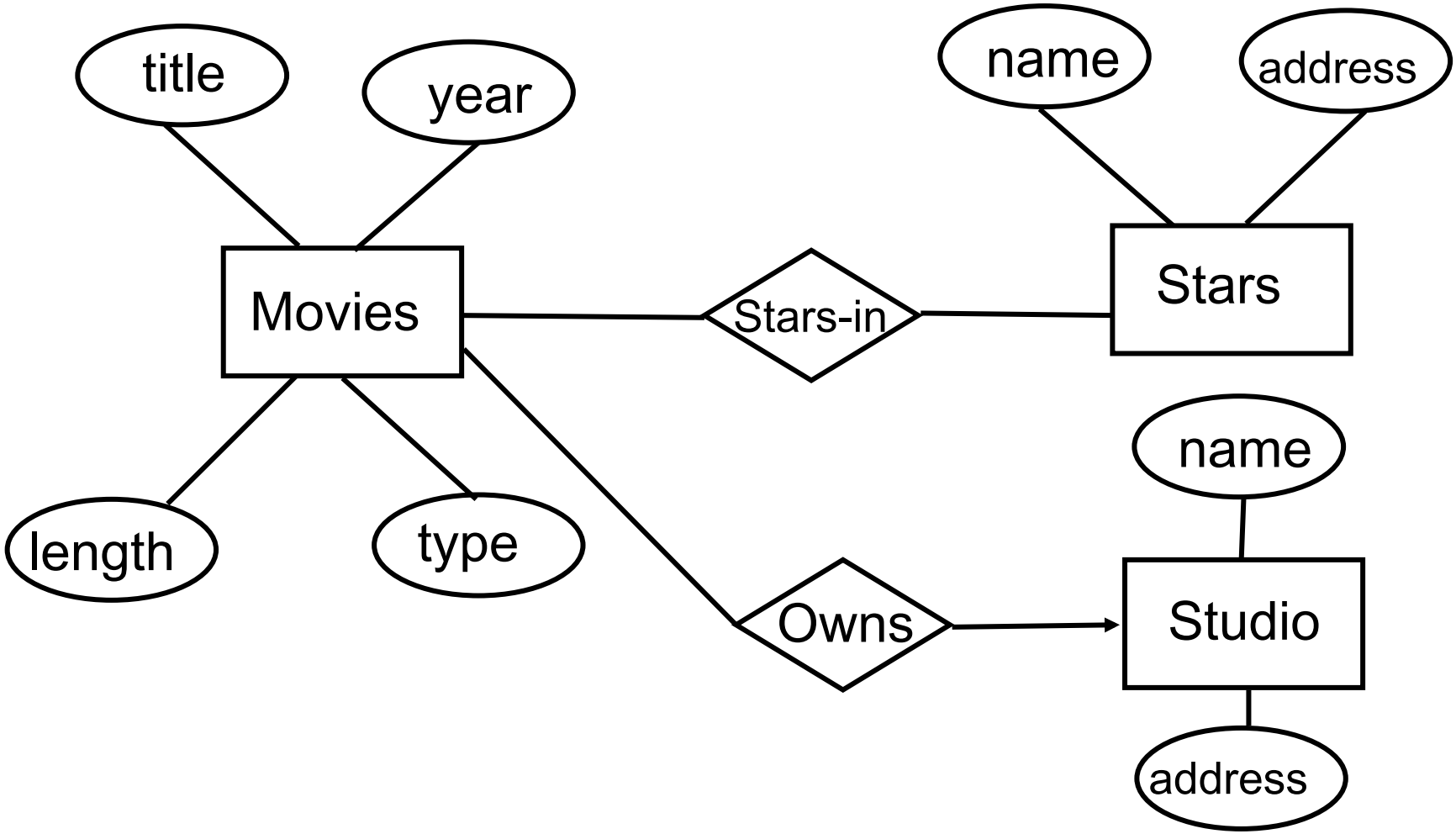


BD et SI (Système d'information)

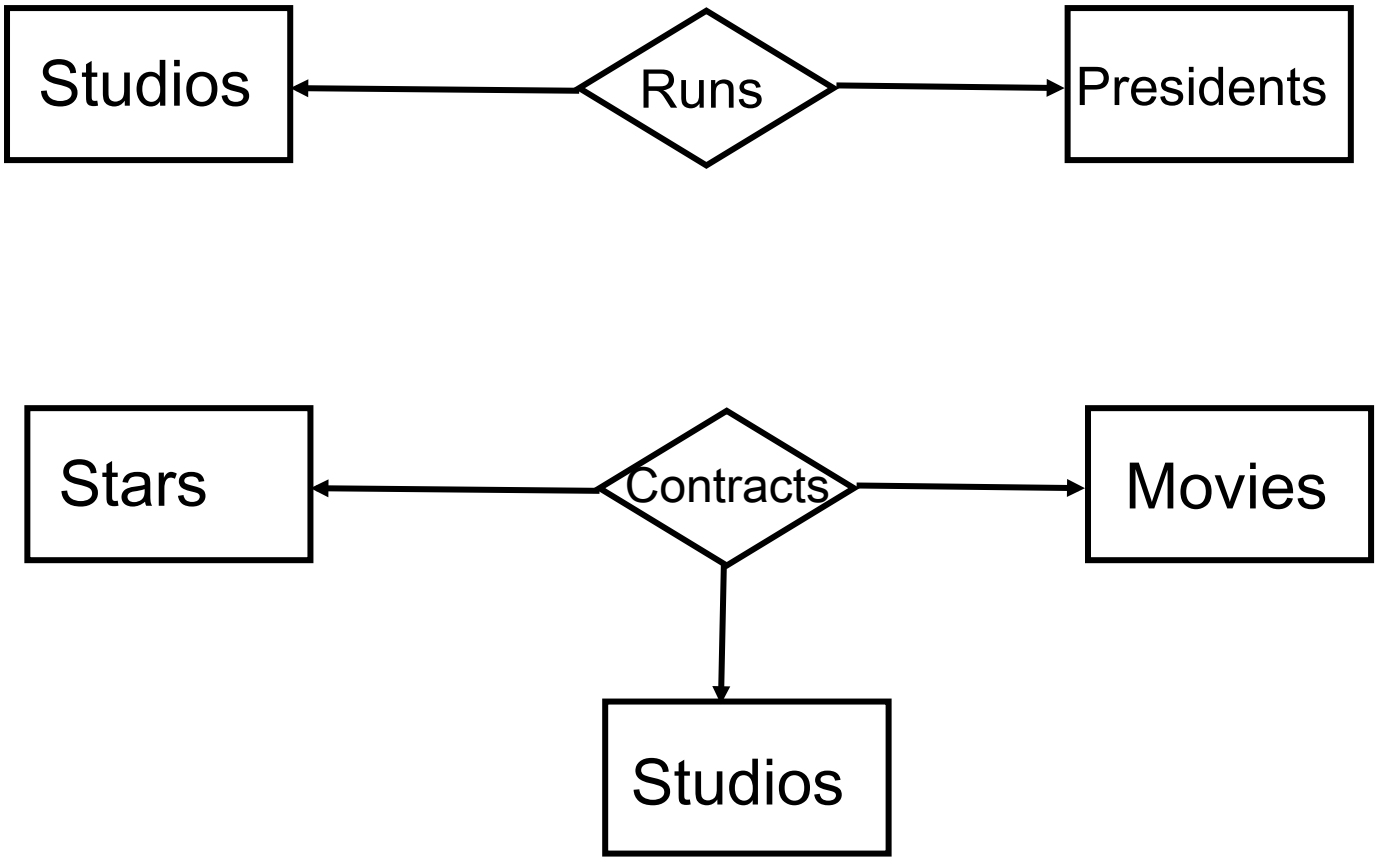
Le SI doit fournir

- l'information nécessaire et PERTINENTE,
- en **temps** opportun
- **aux différents niveaux** de la structure de l 'entreprise
- à un niveau de service et un **prix de revient** compatible avec l 'importance et décisions à prendre et des actions à entreprendre
- à partir de données brutes
- de manière **extensible**

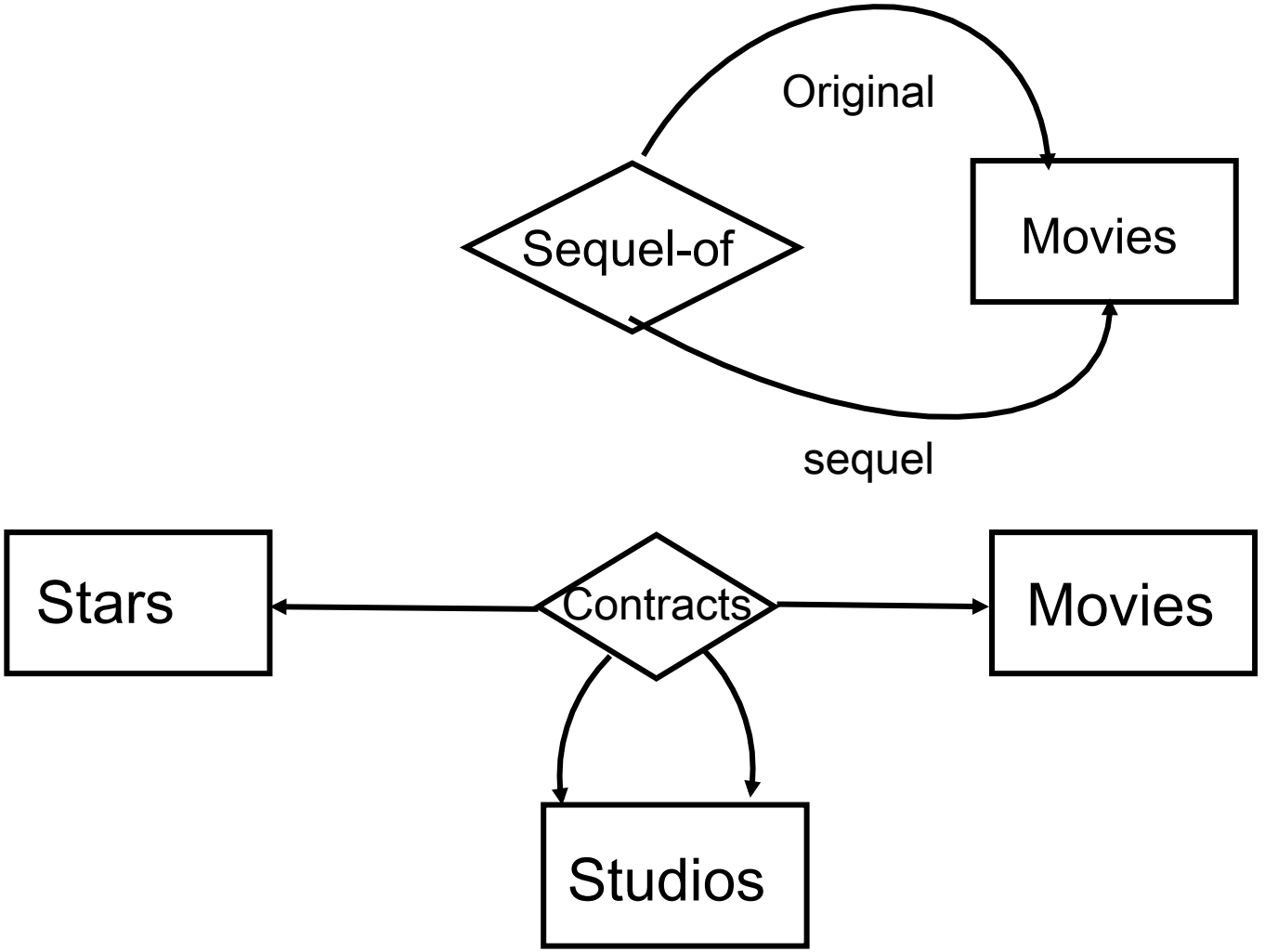
Le Modèle conceptuel : le modèle Entité-Relation



Le modèle Entité-Relation (suite)



Le modèle Entité-Relation (suite)



Le Modèle Relationnel : historique

- Introduction du modèle relationnel en 1970,
  - par E.F. Codd (IBM San Jose)
- Langages d'interrogation et de définition de vues
  - Langage algébrique
  - Calcul des tuples
  - Calcul Relationnel. sur les Domaines (logique)
  - Langages informatiques (SQL, QUEL, ...)
- Langages d'expression de contraintes
  - Formalismes du type précédents
  - Langages ad-hoc (textuels, graphiques,...)
  - dépendances fonctionnelles, clés,références entre tables, inclusions, etc, ...
- Le standard ANSI/SO : SQL
  - version 2, 1992;
  - version 3 : types récursifs + héritage

Le Modèle relationnel : principes

- Schéma : ensemble de relations
- Schéma de base de données : liens sémantiques implicites

- Instances de relations: tuples
- contraintes sur les relations et les tuples,

- Langages
  - déclaratifs , premier ou second ordre
  - algèbre relationnelle , calcul sur les tuples
  - LID de SQL

Eléments principaux de SQL (Version Ansi2)

- Langage de commandes non procédural
- Langage d'interrogation et de définition de vues (LID)
  - **select** colonnes **from** table **where** condition
- Langage de modification de données (LMD)
  - **insert** ligne **into** table
- Langage de définition de données (LDD)
  - **créer** les objets (tables, vues, procédures, démons, ...)
  - définir certaines contraintes sur les tables :
    - clés, références externes (entre tables), contraintes locales, typage, etc …

Eléments principaux de SQL (suite)

- Primitives pour la gestion des transactions
  - verrouillages,
  - retour en arrière,
  - validation

- Primitives de gestion de la sécurité
  - droits d'accès, privilèges, rôles
  - gestion des comptes

Eléments principaux de SQL (suite)

- Relations calculées (views)
  - sources stockées sur le serveur
  - permettent de nommer et mémoriser sur le serveur des requêtes prédéfinies
- Procédures et Fonctions
  - éléments algorithmiques
  - Déclencheurs (triggers)
    - code exprimé dans le même langage procédural que pour les procédures (SQL2, PL/SQL, ...)
    - source+code stockés sur le serveur
    - déclenchements opérés lors de modifications de la base

2. Le modèle relationnel

Schémas relationnels

Schémas Relationnels

Une "table" structurée en colonnes fixes, et en lignes pouvant varier dans le temps en nombre et en contenu, est appelée *relation*..

Le contenu, à un instant donné, de cette table, est une *table instance* de cette relation.

Exemple: relation ***MARQUES***

IdM	*NomM*	*Classe*	*IdProp*
122 233	renault21	24	renault
145 245	sun-sparc	27	sun
147 064	renegade	24	renault

Une *relation* n'est pas définie par des concepts positionnels ; les *lignes* (resp. les *colonnes*) peuvent être permutées.

Une *table instance* d'une relation est un ensemble **non-ordonné** de *tuples* (*lignes*). Chaque *tuple* est composé de valeurs correspondant aux *attributs* (noms des *colonnes*)de la relation.

Le **schéma** de la relation définit les propriétés de chaque *attribut* (nom, type, contraintes, ...).

Schémas relationnels (suite)

Définition : relation

*On appelle schéma relationnel (ou **relation**) tout ensemble fini d'attributs et de domaines :*

$R = \{ (A_1, dom_1), ..., (A_n , dom_n) \}$.

$A = attr(R) = \{A_1, ..., A_n \}$ *désigne l'ensemble des **attributs** de **R**,*

$dom_i = dom(A_i)$ *désigne le **domaine** non vide de chacun des attributs* A_i.

*Chacun de ces domaines définit le type du contenu des colonnes qui formeront une table. Ils définissent donc des **contraintes** sur le contenu de chacun des tuples qui seront présents dans une instance de la relation.*

*Ces domaines sont, dans SQL ANSI2, toujours de type **scalaire** (entiers, chaînes,...) et fini. On ne dispose d'aucun opérateur permettant de leur associer des types structurés.*

Schémas relationnels (suite)

Exemple: la relation MARQUE

Attributs A : `A ={IdM, NomM, Classe, IdProp}`

Domaines :

- dom(IdM) = [1..99 999]
- dom(NomM) = ensemble de tous les mots construits sur l'alphabet {A, B, ..Z,0.. 9}
- dom(Classe) = [1..30]
- dom(IdProp) = tous les noms possibles de sociétés (chaînes limitées à 100 caractères)

Instances de relations

Définition : tuple

- Soit R une relation, ayant comme ensemble d'attributs

  `A = {A1, ...., An}.`

  On appelle tuple défini sur R, tout ensemble t de valeurs $v_1,....,v_n$ associées respectivement aux attributs A_1 ...., A_n, avec comme seule contrainte, $v_i \in dom(A_i)$.
- Un tel tuple est représenté par la notation ensembliste:
  $t = \{v_1: A_1, ..., v_n:A_n\}$

  ou encore par la notation parenthèsée :
  $t = (v_1: A_1; ....; v_n:A_n)$
- De façon générique, on notera la valeur v_i associée à l'attribut A_i du tuple t par : **$v_i = t.A_i$**

Instances de relations (suite)

Exemple : tuple défini sur la relation MARQUE

t = { 122 233 : IdMarq,
'coca-light' : NomMarq,
12 : Classe,
'Coca-Cola ltd.' : IdProp }

qui peut se représenter par la ligne :

IdMarq	NomMarq	Classe	IdProp
122 233	coca-light	12	Coca-Cola ltd.

On a alors dans cet exemple : t.Classe = 12.

Table instance de relation

Définition : table instance de relation

On appelle table instance du schéma R, **tout ensemble de tuples** (lignes) définies sur R.

On note de façon générique une telle table instance **par r(R).**

Exemple : instance de la relation MARQUE:
r(MARQUE) = {t1, t2,t3}

t1 = {1222 :IdM,
'coca-light ' :NomM,
12 :Classe,
'coca-cola ltd. ' :IdProp }

t2= {1224 :IdM,
'coca-cola' :NomM,
12 :Classe,
'coca-cola ltd. :IdProp }

t3 = {1226 :IdM,
'coca-cola' :NomM,
12 :Classe,
'pepsi-cola ltd.' :IdProp }

Contraintes

Contraintes

- Une relation sur l'ensemble des tuples présentes dans une instance.
- Contraintes vérifiées à tout moment par l'instance du schéma.

→ facilitent la conception de la base

→ aident au choix de représentations physiques (clés, ...).

Définition : contraintes, instances acceptables

- On peut associer à toute relation R, un ensemble fini de contraintes, noté

  C= contr(R) = { C1, ... }
- Contrainte = fonction booléenne pouvant s'évaluer, pour chaque instance potentielle de R .
- Une table instance qui satisfait toutes ces contraintes est dite **instance acceptable** de R.
- Le langage d 'expression des contraintes peut être très varié

Langages de contraintes

Enjeux des BDR : disposer de langages déclaratifs aussi riches que possible

Dilemme

- disposer d'un vérificateur
- langage d'un pouvoir d'expression riche

Si les contraintes ne peuvent s'exprimer en premier ordre

→ vérifications programmées directement par l'usager, dans un langage procédural

→ recours à des langages d'automates (grammaires, réseaux de Petri, ...)

Contraintes et langages (suite)

Exemple : Contraintes sur *MARQUE :*

- (C_1) On ne peut avoir le même nom de marque dans la même classe

$$C_1 \equiv \forall t_1, t_2 \in \mathbf{marque}$$
$$((t_1.\mathbf{NomM} = t_2.\mathbf{NomM} \wedge t_1.\mathbf{Classe} = t_2.\mathbf{Classe}) \rightarrow t_1.\mathbf{IdM} = t_2.\mathbf{IdM})$$

- (C_2) Un même propriétaire ne peut être associé à plus de 20 marques d'identificateurs distincts.

Cette contrainte n'est pas du premier ordre !!!!!

(quantification sur les domaines infinis et fonction *card* de type ensembliste)

Peut aisément être vérifiée sur une instance particulière:

$$C_2 \equiv \forall p \in dom(IdProp)$$
$$card(\{t \in r \quad t.IdProp = p\}) < 20)$$

Schéma de bases de données

Bases de Données : ensemble de plusieurs relations indépendantes

Dans l'algèbre relationnelle:

le **même nom d'attribut** utilisé dans deux relations distinctes véhicule le même type d'informations

SQL ne tient pas compte de cette présupposition.

(Il est cependant fortement recommandé de la suivre si l'on veut faciliter la cohérence de la conception)

Des contraintes globales à la base (i.e., faisant intervenir plusieurs relations) peuvent être définies.

Schéma de bases de données(suite)

Liens Sémantiques entre tuples:

→ implicites à travers les valeurs de certains attributs,

et non par adresse ou pointeur comme dans des modèles navigationnels.

Exemple

- toute référence à une marque se fait depuis une autre relation à travers deux attributs qui, en général, porteront les mêmes noms : NomM, Classe .
- la relation :

```
vente(NomM, Classe, Date,
                IdVend,IdAchat)
```

définit les achats/ventes effectuées pour chacune des marques.

Schéma de bases de données (suite)

Conséquences

- "pointeurs"
  - **visibles**
  - **indépendants** de tout choix d'implantation physique
- Navigation à travers les liens
  - **dans les deux sens**
  - sans nécessité de mise en place de "pointeurs inverses"
- Importance du choix d'attributs permettant d'identifier sans ambiguïté un tuple appartenant à une relation donnée

Schéma de bases de données(suite)

Définition : schéma de BD

Un schéma de base de données relationnelle est **un ensemble**

$S = \{R_1, ... R_n\}$

de relations, dans lequel deux attributs de **même nom** dans deux relations différentes, ont les **mêmes domaines**

Définition : instance de schéma BD

Une instance d'un schéma $S = \{R_1, ... , R_n\}$,est **un ensemble de tables**, $r = \{ r_1, ... r_n \}$,où chaque r_i est une table instance de R_i

On peut ajouter au schéma S **des contraintes globales**, portant sur l'ensemble des tables constituant une instance du schéma. Toute **instance** vérifiant ces contraintes est dite **acceptable** pour S.

3. Algèbre relationnelle

Algèbre relationnelle

Définie par Codd en Juin 1970.

Langage d'expressions algébriques en notation fonctionnelle:

- Les variables représentent des tables, instances de relations
- Opérateurs (unaires, binaires, ...) sur ces tables.

Langage du premier ordre.

Il existe un interprète de ce langage, qui permet de calculer toute expression portant sur des tables constituées d'ensembles finis de tuples.

Opérations booléennes

Union, intersection, différence

Soient r et s deux instances d'un même schéma ***R****, i.e.,* ***deux tables ayant mêmes attributs****. Les trois opérations suivantes définissent des instances du même schéma* ***R****:*

$$r \cup s = \{ t \mid t \in r \quad \textbf{ou} \quad t \in s \}$$

$$r \cap s = \{ t \mid t \in r \quad \textbf{et} \quad t \in s \}$$

$$r - s = \{ t \mid t \in r \quad \textbf{et} \quad t \notin s \}$$

Ces expressions sont stables par accroissement des domaines.

Notons que l'opérateur ∩ peut se définir à l'aide de la soustraction :

$$r \cap s = r - (r - s)$$

Opérations booléennes : Exemples

mdep

Id	*Nom*	*Prop*
122	r21	renault
145	sparc	sun
223	spk	sun
147	r19	renault

mner

Id	*Nom*	*Prop*
122	r21	renault
145	sparc	sun
149	r18	renault

mdep ∪ menr

Id	*Nom*	*Prop*
122	r21	renault
145	sparc	sun
223	spk	sun
147	r19	renault
149	r18	renault

mdep ∩ menr

Id	*Nom*	*Prop*
122	r21	renault
145	sparc	sun

mdep − menr

Id	*Nom*	*Prop*
223	spk	sun
147	r19	renault

Ordre de calcul des opérateurs booléens

Si les tuples sont dispersés sur disque, les opérations booléennes effectuées par copie nécessitent **O(k.n.logn)** accès.

En cas d'existence d'index triés sur les tables on a alors des ordres de calcul de **O(k.n),** avec obtention de résultats triés.

Le cas où les tuples sont organisés en blocs de données contigus sera étudié plus tard (jointure).

Projection : définition

Opération unaire

→ copier une relation en ne gardant que certaines colonnes

→ stable par accroissement des domaines

Définition

*Soient **R** un schéma, et **A** = {A_1, ..., A_n} ⊆ **R**, un sous-ensemble d'attributs de **R**.*

*La projection sur **A** d'un tuple t défini sur **R**, est le tuple défini sur **A** par :*

$$\pi_A(t) = \pi_{A1, \ldots, An}(t) = \{t.A_1:A_1, \ldots, t.A_n:A_n\}$$

*La projection sur **A** d'une table r(**R**) instance du schéma **R**, est une instance du schéma **A** définie par :*

$$\pi_A(r) = \pi_{A1, \ldots, An}(r)$$

$$= \{ \pi_A(t_r) \mid t_r \in r \}$$

$$= \{ t(A) \mid \exists t_r \in r \ \ avec\ t.A_i = t_r.A_i \ \ pour\ i=1,\ldots,n \}$$

Projection : propriétés

Proposition

*Soient r et s des instances de **R**, et **A** un sous-ensemble d'attributs de **R**. On a alors* :

$$\pi_A(r \cup s) = \pi_A(r) \cup \pi_A(s)$$

On n'a pas toujours :

$$\pi_A(r \cap s) = \pi_A(r) \cap \pi_A(s)$$

$$\pi_A(r - s) = \pi_A(r) - \pi_A(s)$$

Exemple de projection

marque

Id	*Prop*	*Nom*	*Classe*
122	r21	14	renault
145	sparc	12	sun
223	spk	12	sun
147	r19	13	renault

$\pi_{Classe,Prop}(marque)$

Classe	*Prop*
14	renault
12	sun
13	renault

Notez la disparition des lignes redondantes :

$\pi_{Classe,Prop}$(145:Id; 'sparc':Nom; 12:Classe; 'sun':Prop)

= $\pi_{Classe,Prop}$(223:Id; 'spk':Nom; 12:Classe; 'sun':Prop)

= (12:Classe; 'sun':Prop)

Procédé de calcul de la projection

Il consiste en une itération sur les éléments de la table avec copie partielle des lignes et élimination des redondances.

Le coût est en **`O(n.logn)`** où **`n`** est le nombre de tuples de la table

Sélection : définition

Filtrage de valeur d'attribut

→ extraire par copie certaines lignes d'une relation.

→ stable par accroissement des domaines.

Définition : sélection par filtrage d'un attribut

Soient *r(R)* une instance d'un schéma *R*, *A* un attribut de *R*, et *a* une constante appartenant à *dom(A)*

La sélection de *r* par filtrage de *A* sur *a*, est une instance du même schéma définie par :

$$\sigma_{A=a}(r) = \{t \in r \mid t.A = a\}$$

Sélection : propriétés

Commutativité:

$$\sigma_{A=a}(\sigma_{B=b}(r)) = \sigma_{B=b}(\sigma_{A=a}(r))$$

On écrit alors :

$$\sigma_{A=a}(\sigma_{B=b}(\ldots\sigma_{L=l}(r)\ldots)) = \sigma_{A=a,B=b,\ldots,L=l}(r)$$

Distributivité sur les opérations booléennes

Pour tout opérateur booléen $o \in \{ \cup, \cap, - \}$ on a :

$$\sigma_{A=a}(r \; o \; s) = \sigma_{A=a}(r) \; o \; \sigma_{A=a}(s)$$

Commutativité avec la projection:

Soient r(**R**) une instance d'un schéma **R**, $\mathbf{A} \subseteq \mathbf{R}$ et $a \in dom(A)$. On a alors :

$$\pi_A(\sigma_{A=a}(r)) = \sigma_{A=a}(\pi_A(r))$$

Exemple de sélection

marque

Id	Nom	Classe	Prop
122	r21	14	renault
128	r30	14	renault
145	sparc	12	sun
223	spk	12	sun
147	r19	13	renault

$\sigma_{Classe=14}(marque)$

Id	Nom	Classe	Prop
122	r21	14	renault
128	r30	14	renault

Sélection étendue

Définition : sélection étendue

Soient r(**R**)une instance d'un schéma **R**, $\{A_1,...,A_n\}$ un sous-ensemble d'attributs de **R**, et f une fonction booléenne **calculable**, ne dépendant d'aucune relation de la base, et définie sur $dom(A_1)$ x…x $dom(A_n)$:

$$f : (x_1, ..., x_n) \rightarrow \{true, false\}$$

La sélection de r par $f(A_1, ..., A_n)$ est une instance du même schéma définie par :

$$\sigma_{f(A1, ..., An)}(r) = \{t \in r \mid f(t.A_1, ..., t.A_n) = true\}$$

Exemples d'expressions de sélection étendue :

$\sigma_{A<3 \vee A=13}(r)$ $\sigma_{A-B>16}(r)$

$\sigma_{Classe=14 \vee Classe=12}(marque)$ $\sigma_{Date>930301 \wedge Prop='renault'}(enreg)$

Procédés de calculs de la sélection

Pour la sélection simple par égalité, l'ordre de calcul est O(n)

Si l'on dispose d'index, il peut être égal au nombre de réponses

Pour les sélections étendues, l'ordre est n.k, k dépendant de la fonction utilisée.

Jointure Naturelle de tuples

Principe :

- → Concaténation de chaque ligne d'une table avec chaque ligne de l'autre si ces lignes partagent les mêmes valeurs pour les attributs de même nom
- → Produit cartésien des tuples considérés si aucun attribut partagé

Stable par accroissement des domaines.

Notation

*R et S étant deux ensembles d'attributs, nous noterons par **RS** l'ensemble d'attributs égal à l'union de R et de S.*

$$RS = R \cup S .$$

Jointure Naturelle de tuples (suite)

Définition : jointure de tuples

On dit que deux *tuples* $t_r(R)$ et $t_s(S)$ sont joignables, ssi il existe un tuple $t(RS)$ tel que :

$$\pi_R(t) = t_r \quad \text{et} \quad \pi_S(t) = t_s$$

Ce tuple unique est noté $(t_r \triangleright\!\!\triangleleft t_s)$

$(t_r \triangleright\!\!\triangleleft t_s)$ existe $\Leftrightarrow$ $\pi_{R\cap S}(t_r) = \pi_{R\cap S}(t_s)$

$(t_r \triangleright\!\!\triangleleft t_s)$ existe $\Rightarrow$ $\pi_R(t_r \triangleright\!\!\triangleleft t_s) = t_r$

$\Rightarrow$ $\pi_S(t_r \triangleright\!\!\triangleleft t_s) = t_s$

Exemples de jointure de 2 tuples

NomEt		*Age*	*Fil*	*Année*
Barry	23	Log	2	

⋈

Fil	*Année*	*Titre*
Log	2	BD3

=

NomEt		*Age*	*Fil*	*Année*	*Titre*
Barry	23	Log	2	BD3	

Attention le tuple suivant n'existe pas.

('JointNéoprène':NomProd; 302:CodeProd, 204.5:PrixLin)

⋈

('Sté X':NomDistrib; 'Rondelle6x4':NomProd; 2500:QtéMoy)

Jointure naturelle de deux tables

Définition : jointure naturelle de deux instances

Soient r(R) et s(S) deux instances de relations et RS l'union de leurs attributs.

La jointure naturelle de r et s est une instance du schéma RS (union des deux ensembles d'attributs) définie par :

$$r \triangleright\!\triangleleft s = \{\, t(RS) \mid \pi_R(t) = t(R) \in r \;\wedge\; \pi_S(t) = t(S) \in s \,\}$$

On a alors :

(1) $t \in r \triangleright\!\triangleleft s \;\Leftrightarrow\; \exists t_r \in r \;\; \exists t_s \in s$ **tel que**

$t.A = t_r.A$ pour tout attribut $A \in R$

et $t.A = t_s.A$ pour tout attribut $A \in S$

(2) $t \in r \triangleright\!\triangleleft s \;\Rightarrow$ pour tout attribut $A \in R \cap S$

$t.A = t_r.A = t_s.A$

Exemple :

marque

IdM	NomM	Classe	IdProp
122233	renault21	24	renault
145245	sun-sparc	27	sun
223423	sptrklm	24	sun
147064	renegade	24	renault

prop

IdProp	NomProp	Ville
renault	renault s.a	boulogne
sun	sun-micros	sun-valley
jeep	jeep inc.	detroit

marque ⋈ prop

IdProp	NomProp	Ville	IdM	NomM	Classe
renault	renault s.a	boulogne	122 233	renault21	24
renault	renault s.a	boulogne	147 064	renegade	24
sun	sun-micros	sun-valley	145 245	sun-sparc	27
sun	sun-micros	sun-valley	223 423	sptrklm	24

Propriété de la jointure

Commutativité et associativité

$$r \bowtie (s \bowtie t) = (r \bowtie s) \bowtie t$$

$$r \bowtie s = s \bowtie r$$

Distributivité vis-a-vis de l'union

$$r \bowtie (s \cup t) = (r \bowtie s) \cup (r \bowtie t)$$

Remarque : si $R \cap S = \varnothing$ alors $r \bowtie s$ est isomorphe au produit cartésien des deux instances r et s.

Il est noté $r \times s$

Procédé détaillé de calcul du produit cartésien

De nombreuses stratégies existent, en fonction de la mémoire centrale disponible.

B_r et B_s sont le nombre d'accès à des blocs sur disque nécessaires pour lire chacune de deux relations ***r* et *s*, supposées organisées en blocs compressés**, et ***M est*** **le nombre de blocs pouvant résider en mémoire simultanément.**

Si les tables ne sont pas organisées en blocs, on ajoute T_r et T_s pour lire les données et les compresser en mémoire plus B_r et B_s pour écrire ces blocs compressés

Procédé détaillé de calcul du produit cartésien

L'algorithme simple pour le calcul du produit cartésien *r* x *s* est le suivant :

```
Pour chaque tuple tr dans r faire
        Pour chaque tuple ts dans s faire
         écrire tr x ts
```

Procédé détaillé de calcul du produit cartésien

Si aucune des deux tables ne tient en mémoire, on divise alors *s* en segments de *M -1* blocs chacun.

On lit alors un segment de *s* et dans les blocs libres de la mémoire, on lit chaque bloc de *r*, en produisant le produit cartésien de chaque tuple du segment avec chaque tuple du bloc de *r*

pour chaque segment s_s de **s faire**
  pour chaque tuple t_s dans s_s **faire**
  pour chaque bloc b_r de **r faire**
    pour chaque tuple t_r dans b_r **faire**
      ecrire t_r x t_s

Le coût total est alors :

$B_r (T_s + B_s (M-1)) + B_s (T_r + 1)$

Un procédé de calcul de la jointure

Une stratégie consiste à effectuer le produit cartésien puis la sélection.

D'autres stratégies plus complexes utilisent des index partiels clusterisés ou pas.

Division

Intuition

Division produit une relation sur **R-S** qui regroupe toutes les éléments de **R-S** qui dans **R** sont associés à tous les éléments de **S**

Définition

Soient r(**R**) et s(**S**) deux instances de relations, avec $S \subseteq R$.

Le **quotient de r par s** est la relation définie sur le schéma **Q=R-S** par:

$$r \div s = \{ t_q \in \pi_Q(r) \mid \forall t_s \in s,\ (t_q \rhd\lhd t_s) \in r \}$$

Autrement dit,

$r \div s$ **est le plus grand ensemble q de** $\pi_Q(r)$

tel que $(q \rhd\lhd s) \subseteq r$.

Division (suite)

Propriété

La division s'exprime en fonction des opérateurs précédents :

$$r \div s = \pi_Q(r) - \pi_Q((\pi_Q(r) \rhd\lhd s) - r)$$

Exemple : Chercher s'il existe un monopole en classe 14.

"Les sociétés qui possèdent toutes les marques de la classe 14"

$$\pi_{IdProp,\ IdMarq}(marque) \div \pi_{IdMarq}(\sigma_{Classe=14}(marque))$$

$$= \pi_{IdProp}(marque) - \pi_{IdProp}[\pi_{IdProp}(marque) \rhd\lhd \pi_{IdMarq}(\sigma_{Classe=14}(marque)) - \pi_{IdProp,\ IdMarq}(marque)]$$

Renommage des attributs

Définition

Soit A un des attributs d'un schéma **R**, B un attribut de même domaine que A, et n'appartenant pas à **R**-{A}.
Le renommage de A en B dans une instance r(**R**) est une instance du schéma **R'**= **R**-{A} ∪ {B} définie par :

$$\delta_{A \leftarrow B}(r) = \{t'(R') \mid \exists\, t \in r \quad t(R\text{-}A) = t'(R\text{-}A) \text{ et } t(A) = t'(B)\ \}$$

On peut étendre ce renommage, sous réserve de ne pas créer de collisions de noms, à plusieurs attributs.
Il est supposé s'effectuer de facon simultanée., et est noté :

$$\delta_{A1, \ldots, An \leftarrow B1, \ldots, Bn}(r)$$

Renommage des attributs : exemple

PROP = {*IdProp, NomProp, Pays, Ville*}

ENREG={*NumEnr, IdMarq, Date, Deposant*}

"noms de propriétaires ayant déposé au moins une marque avant le 15 janvier 91"

→ on doit effectuer une jointure sur l'attribut *Deposant* de *ENREG* et l 'attribut *IdProp* de *PROP* :

$$\pi_{\text{NomProp}}\delta_{\text{Deposant}\leftarrow\text{IdProp}}\ (\sigma_{\text{Date}<910115}(\text{enreg})) \triangleright\triangleleft \text{prop}$$

4. Le Calcul Relationnel des Tuples

Le Langage CRT

→ Base théorique de SQL

- Comme pour l'Algèbre Relationnelle, les paramètres désignent des tables connues, instances de relations, ***mais aussi des variables*** qui représentent des ***tuples***.
- L'accès à la valeur de l'attribut *Att* d'un tuple *X* se fait par l'expression pointée *X.Att*.

→ 3 types d'expressions:

- les ***termes***, qui s'évaluent par des valeurs individuelles éléments des domaines d'attributs,
- les ***expressions booléennes évaluables*** qui ne peuvent s'évaluer que lorsque l'on connaît la valeur des variables libres des tuples qui y figurent.
- les **expressions** qui permettent de construire des ***instances de relations***, à partir de relations paramètres (qui dénotent des tables connues à un instant t)

Syntaxe des Formules du Calcul Relationnel de Tuples)

Soit S= {R1, ...,Rn} un schéma de base de données définissant des relations présentes dans la base.

Lexique :

- variables "tuple" : X, Y, ...
- constantes individuelles : a, b, c, ...
- relations de base (paramètres) : R, S, ...

Langage des Termes (expressions individuelles)

Toute constante est un **terme**

Si X est une occurrence d'une **variable** tuple dotée des attributs A_1, ..., A_k, alors $X.A_i$ est un **terme** de domaine dom(A_i)

Si t_1, ..., t_k sont des **termes** de domaines respectifs D_1, ..., D_k, f est un symbole de **fonction** évaluable de signature (D, D_1, ..., D_k),
alors $f(t_1, ..., t_n)$ est un **terme** de domaine D

Les Formules Booléennes :

- Si p est un **prédicat évaluable** de signature (D_1, ..., D_n), t_1, ..., t_n des termes vérifiant dom(t_i) = D_i, alors p(t_1, ..., t_n) est une **formule booléenne**
- Si t et t' sont deux **termes** de mêmes domaines, alors (t=t') est une **formule booléenne**
- Si F et G sont deux **formules booléennes,** alors $(F \wedge G)$, $(F \vee G)$, $(F \rightarrow G)$, $(F \leftrightarrow G)$, $(\neg F)$ *sont des* ***formules booléennes***
- Si X est une **variable** tuple, F une **formule booléenne,** R une **expression relation** sans occurrence libre de X, alors $(\exists X \in R \ F)$, $(\forall x \in R \ F)$ sont des **formules booléennes**

Instances de Relations

- Si T est une relation (**table connue**), X_1, ... X_n des **variables** "tuple", R_1, ... R_n des **instances de relations** sans **aucune** occurrence libre des X_i (jointures), t_1, ... ,t_n n termes (projections), A_1, ... ,A_n n attributs distincts (renommage des projections), F une expression booléenne (sélection), alors $\{(t_1:A_1; ...; t_n:A_n)\ X_1 \in R_1, ... \mid F\}$ est une **expression relation** dotée des attributs A_i.

  Les variables X_1, ..., sont **liées** par cette définition, qui joue un rôle syntaxique analogue à celui de quantificateur.
- Si R1 et R2 sont deux **expressions relations** dotées des mêmes attributs alors $R1 \cup R2$, $R1 \cap R2$, $R1 - R2$ sont des **instances de relations**

Exemple

Les noms et pays des propriétaires dont toutes les marques de la classe 14 ont été déposées avant le 15 avril 91 (ou ne possédant pas de marques de ce type)".

MARQUE={IdMarq,NomMarq,Classe,IdProp}

PROP = {IdProp, NomProp, Pays, Ville}

ENREG={NumEnr, IdMarq, Date, Deposant}

$$\{(P.NomProp:Nom;\ P.Pays:Pays)\ P \in PROP \mid$$
$$\forall M \in MARQUE \quad \forall E \in ENREG \mid$$
$$((P.IdProp=M.IdProp \wedge M.IdMarq=E.IdMarq$$
$$\wedge\ M.\ Classe\ =14)$$
$$\rightarrow E.Date\ <\ 910415\ )\}$$

Propriétés de CRT

- Une expression **fermée** (sans variables libres), définit sans ambiguïté une instance de relation unique dans une interprétation donnée

- Toute formule de définition de relation de CRT est équivalente à une formule de AR et vice-versa

  En fait cette propriété n'est valable que si l'on limite CRT en ne permettant pas l'utilisation de fonctions dans les projections

- Toute relation qui peut s'exprimer en langage du premier ordre par une formule stable vis-a-vis de l'extension des domaines, peut s'exprimer dans un langage relationnel (CRT ou AR), et inversement.

5. Le langage SQL

Existe de nombreux dialectes

Normes ANSI : SQL-92 (SQL2)

Nouveau standard en préparation : SQL3 (récursivité, triggers, objets)

5.1 Requêtes simples

```
SELECT * FROM marque WHERE classe=14;
SELECT NomM, NomE  FROM marque, enr
  WHERE IdS=Déposant;
```

SELECT → liste des attributs demandés

FROM → liste des relations utilisées par la requête

WHERE → conditions pour les sélections et attributs de jointure

… permet de faire des projections, sélections et jointures !

Projections : la clause SELECT

La clause SELECT permet d'effectuer des projections

```
SELECT Nom  FROM marque;
```

mais elle ne supprime pas les redondances

Renomage :

```
SELECT Nom AS NomMarque , IdS AS Proprio
FROM marque;
```

NomMarque	Proprio
Microsoft ….	Bill Gate …

Projections : la clause SELECT (suite)

La clause SELECT peut contenir des constantes:

```
SELECT Nom , Prix*100 AS Prix
          FROM marque;
```

NomMarque	Prix
Microsoft Linux ….	100 100 …

Sélection : la clause WHERE

La clause conditionnelle WHERE peut utiliser des:

- Comparateurs: =, <>, <, >, <=, >=
- Opérateurs arithmétiques: +, *, …
- Concaténations de chaînes: 'foo'||'bar' a pour valeur 'foobar'
- Opérateurs logiques: AND, OR, NOT

Les Chaînes

- B'011' représente une chaîne de 3 bits
- X'7FF' représente une chaîne de 12 bits (0 suivi de onze 1)
- Le booléen TRUE peut être représenté par B'1'
- La comparaison s'effectue caractère par caractère
  'fodden' < 'foo'
- p LIKE s avec p une chaîne et s un pattern:
  p NOT LIKE s
  - **`SELECT .. WHERE tiltle LIKE 'STAR _ _ _ _';`**
    (recherche titre de 8 caractères qui commence par 'STAR')
  - **`SELECT .. WHERE tiltle LIKE '%''s%';`**
    (titre peut être n'importe quelle chaîne contenant *'s*)

Dates et heures

- Types de données spécifiques avec des représentations variées dans les différents "dialectes" de SQL

- Formes standards:

  DATE '1988-07-30'

  TIME '15:00:02.5…' avec un nombre quelconque de chiffres

Tri des résultats

Ajout à l'instruction SELECT-FROM-WHERE d'une clause ORDER BY <liste d'arguments>

Par défaut l'ordre est croissant

```
SELECT * FROM movie WHERE year=1900
ORDER BY title;

SELECT * FROM movie WHERE year=1900
ORDER BY 1, 2 DESC;
```

5.2 Requêtes portant sur plusieurs relations

Par défaut SQL calcule le produit cartésien; *la jointure est uniquement effectuée pour les arguments spécifiés dans la clause WHERE*

```
enr(NumE,Libelle, Pays , Déposant , Date)
marque(IdM, Nom, Classe, Pays , Prop , Date)

SELECT Nom FROM marque, enreg
          WHERE IdM=Déposant;
```

Ambiguïtés sur les noms de relation

Introduction de variable tuples (nom de la relation peut être utilisé s'il apparaît une seule fois dans la clause FROM)

```
SELECT    M.Nom AS NomMarque,
          E.NumE AS NumeroEnreg
    FROM      marque AS M, enr  AS E
    WHERE     E.Id=M.ID;
```

Sémantique d'une requête portant sur plusieurs relations

Modèle de calcul:

- Calcul du produit cartésien des tuples de la clause FROM
- Sélection dans ce produit des tuples qui vérifient les conditions définies dans la clause WHERE

(même interprétation dans DATALOG et dans l'algèbre relationnelle)

Conséquence (non-intuitive) : si le produit cartésien est vide le résultat est l'ensemble vide

Interprétation d'une requête portant sur plusieurs relations(suite)

Exemple:

Soit R, S,T trois relations unaires dont l'attribut est A.

Rechercher les éléments qui sont dans R et soit dans S,

soit dans T

```
SELECT R.A FROM R,S,T
          WHERE R.A=S.A OR R.A=T.A;
```

Si T est vide on pourrait s'attendre à obtenir R ∩ S alors

que la requête SQL produit l'ensemble vide

Intersection, Union et Différence

Nom des marques déclarées à la fois dans la classe 14 et la classe 10

```
(SELECT Nom, Classe FROM marque WHERE Classe=14)
INTERSECT
(SELECT Nom, classe FROM marque WHERE Classe=10)
```

(idem pour `UNION`)

Nom des marques n'appartenant pas à la classe 10

```
(SELECT Nom, Classe FROM marque)
EXCEPT
(SELECT Nom, Classe FROM marque WHERE Classe=10)
```

Remarques :

Ces requêtes travaillent sur des ensembles et suppriment les redondances

5.3 Sous-requêtes

Une sous-requête est une expression dont le résultat de l'évaluation est une relation:

- Une expression SELECT FROM WHERE est une sous-requête
- Une sous-requête peut apparaître dans la clause WHERE
- Une sous-requête peut produire une table avec un nombre quelconque d'attributs et de tuples

Sous-requête (suite)

Sous-requête qui produit une seule valeur

Exemple: recherche du nom de la société qui a déposé une marque sous le numéro d'enregistrement 17

```
SELECT Nom FROM marque M, enr E
    WHERE M.IdM=E.IdM AND E.NumE=17;
```

Ou

```
SELECT Nom FROM marque
    WHERE IdM = (SELECT IdM FROM enr WHERE NumE=17)
```

La sous requête produit une relation unaire qui contient un seul tuple

Opérateurs qui s'appliquent aux relations

Les opérateurs qui s'appliquent aux relations et qui produisent un booléen sont :

- EXISTS R : vrai ssi R n'est pas vide
- s IN R : vrai ssi s est égal à un des tuples de R
- s > ALL R : vrai ssi s est plus grand que toute valeur de la relation unaire R
- s > ANY R : vrai ssi s est plus grand qu'au moins une des valeurs de la relation unaire R

On peut aussi utiliser les comparateurs <, <>,<=,>= avec ALL et ANY

Les opérateurs EXISTS,IN, ALL et ANY peuvent être niés:

- NOT EXISTS R: vrai ssi R est vide
- NOT s > ANY R: vrai ssi s est la valeur minimale de R
- s NOT IN R : vrai ssi s n'est égal à aucun des tuples de R

Conditions sur des tuples

Un tuple est représenté par une liste de valeurs scalaires entre parenthèses

Exemple:

```
movie(title,year,length,studioName,producer)
starsIn(movieTitle,movieYear,starName)
movieExec(name,address,cert,netWorth)
```

La requête suivante recherche le producteur des film de Harrison Ford:

```
SELECT name FROM movieExec
  WHERE cert IN (SELECT producer FROM movie
   WHERE (title,year) IN
     (SELECT movieTitle, movieYear FROM starsIn
            WHERE starname='Harrison Ford')
            );
```

Conditions sur des tuples (suite)

La requête la plus imbriquée produit une table de la forme:

```
        title       |  year
        Star Wars   |  1977
        Raider of   |  1993
        ..
```

La requête du milieu génère un ensemble de certificats.

La requête ci-dessous produit le même résultat (mais les doublons ne sont pas gérés de la même manière):

```
SELECT name FROM movieExec, movie, starsIn
  WHERE cert = producer AND title = movieTitle AND
    year = movieYear AND starName ='Harrison Ford';
```

Sous requêtes corrélées

Il est possible d'utiliser dans une sous-requête un terme qui provient d'une variable tuple extérieure à la requête

Exemple: recherche des noms utilisés pour deux ou plusieurs films

```
SELECT title from movie AS Old
   WHERE year < ANY
     (SELECT year FROM movie
        WHERE title = OLD.title);
```

Un nom de film sera listé une fois de moins que le nombre de films portant ce nom

5.4 Gestion des doublons

SQL construit des **multi-ensembles** (et non des ensembles comme l'algèbre relationnelle)

- Lorsque une ***requête SQL*** construit une nouvelle relation elle ***n'élimine pas automatiquement les doublons***
- Le même tuple peut apparaître plusieurs fois dans une relation

Pour éliminer les doublons:

```
SELECT DISTINCT name FROM marque;
```

Attention,l'usage de distinct a un coût important: il faut stocker toute la relation et la trier

Gestion des doublons (suite)

`UNION, INTERSECT` et **`EXCEPT`** sont des opérations ensemblistes qui éliminent les doublons

L'utilisation de **`UNION ALL, INTERSECT ALL`** et **`EXCEPT ALL`** permet de travailler sur des multi-ensembles

`R EXECPT ALL S` : élimine autant d'occurrences d'un tuple **t** dans **R** que celui-ci à d'occurrences dans **S**

5.5 Agrégats

Une agrégation est une opération qui construit une seule valeur à partir de la liste des valeurs d'une colonne (ou d'un ensemble de tuples)

Opérateurs: **SUM, AVG, MIN, MAX, COUNT**

Exemples:

```
article(Id,Nom,Prix)
SELECT AVG(Prix) FROM article;
SELECT COUNT(*) FROM article;
SELECT COUNT(DISTINCT nom) FROM article;
```

5.5 Agrégats (suite)

- SELECT city FROM weather WHERE temp_lo = max(temp_lo);

  Incorrect car la fonction d'agrégat ne peut être utilisée dans la clause WHERE

- SELECT city FROM weather WHERE temp_lo = (SELECT max(temp_lo) FROM weather);

  OK (sous requête est indépendante)

La clause `GROUP BY`

La clause GROUP BY permet de regrouper un ensemble de tuples qui ont la même valeur dans les colonnes mentionnées

Exemple 1 : calcul du nombre total des minutes des films produits par un studio

```
movie(title,year,length,studioName,producer)
SELECT studioName, SUM(length) FROM movie
     GROUP BY studioName;
```

Remarques:

```
SELECT studioName FROM movie GROUP BY studioName;
```

est équivalent à

```
SELECT DISTINCT studioName FROM movie ;
```

Seuls les attributs mentionnés dans la clause GROUP BY peuvent apparaître dans la clause SELECT (les autres peuvent apparaître dans les fonctions d'agrégat)

La clause GROUP BY (suite)

Exemple 2 :

```
SELECT year, country, product, SUM(profit) FROM sales
GROUP BY year, country, product;
```

year	*country*	*product*	*SUM(profit)*
2000	Finland	Computer	1500
2000	Finland	Phone	100
2000	India	Calculator	150
2000	USA	Calculator	75

La clause GROUP BY (suite)

SELECT name, size, AVG(unit_price) FROM product
GROUP BY name, size;

name	size	AVG(product.unit_price)
Tee Shirt	Small	9
Tee Shirt	Medium	14
...	...	...

! tout champ sélectionné non calculé doit faire partie du
! regroupement

SELECT Nom_client, date_commande, Max(Montant)
FROM commande GROUP by Nom_client;

→ erreur.

Mais tout champ qui fait partie du regroupement ne doit pas être sélectionné.

SELECT Nom_client, Max(Montant) FROM commande
GROUP by Nom_client, date_commande ;

→ OK (mais non-conforme SQL/92)

La clause GROUP BY (suite)

SELECT year, SUM(profit) FROM sales GROUP BY year;

```
year        SUM(profit)
2000              4525
2001              3010
```

Pour déterminer le profit total de toutes les années, une autre requête est nécessaire

La clause ROLLUP (non-standard) fournit le total dans une ligne avec des valeurs null :

SELECT year, SUM(profit) FROM sales
GROUP BY year WITH ROLLUP;

```
year              SUM(profit)
2000              4525
2001              3010
NULL              7535
```

La clause **HAVING**

La Clause **HAVING** permet de construire des groupes en utilisant une propriété du groupe lui-même

Exemple : Donner la liste des salaires moyens par fonction pour les groupes ayant plus de deux employés.

```
SELECT fonction,COUNT(*),AVG(salaire)
      FROM emp
      GROUP BY fonction
      HAVING COUNT(*) > 2;
```

```
FONCTION          COUNT(*)      AVG(SALAIRE)
administratif     4             12375
commercial                5                 21100
```

La clause HAVING (suite)

Marque	Modele	Serie	Numero	Compteur
Ford	Escort	Match	8562EV23	
Peugeot	309	chorus	7647ABY82	189500
Peugeot	106	KID	7845ZS83	75600
Renault	18	RL	4698SJ45	123450
Renault	Kangoo	RL	4568HD16	56000
Renault	Kangoo	RL	6576VE38	12000

SELECT Marque, **AVG(Compteur) AS Moyenne** FROM VOITURE
GROUP BY Marque HAVING Moyenne IS NOT NULL

Marque	Moyenne
Renault	63816.6
Peugeot	132550

Ordre des clauses dans SQL

L'ordre des clauses est le suivant

SELECT FROM WHERE GROUP BY
HAVING ORDER BY

Les deux premières clauses sont obligatoires

5.6 Modification d'une base de données

- Insertion de tuples dans une relation
- suppression de tuples dans une relation
- modification de tuples dans une relation

Insertion de tuples dans une relation

La forme basique d'une instruction d'insertion est :

- Mots clés **INSERT TO**
- Le nom de la relation, R
- Une liste d'attributs de R (entre parenthèses)
- Le mot clé **VALUES**
- Une liste parenthèsée de valeurs

Exemple:

```
marque (id, nom, classe, pays,prop)
INSERT INTO marque(id,nom) VALUES (1,'Coca');
INSERT INTO marque VALUES(1, 'Coca', 12, 'Fr', 123);
```

Insertion de tuples dans une relation(suite)

```
data(numero, nom, classe, pays)

INSERT INTO marque
   SELECT numero, nom, classe, pays FROM data
   WHERE data.nom NOT IN
     (SELECT nom FROM marque);
```

Cette requête montre l'importance d'une évaluation complète de la clause SELECT avant la clause INSERT

Suppression de tuples dans une relation

La forme basique d'une instruction de suppression est :

1. Mots clés **DELETE FROM**
2. Le nom de la relation, R
3. Mot clé WHERE
4. Condition

Exemple:

```
marque (id, nom, classe, pays,prop)
```

```
DELETE FROM marque
      WHERE nom='Channel' AND classe='14'
```

Attention : DELETE supprime tous les doublons

(⇒ L'insertion d'un tuple, suivi de sa suppression ne restaure pas nécessairement l'état avant l'insertion)

Mise à jour de tuples dans une relation

Il est possible de modifier plusieurs tuples avec l'instruction de mise à jour :

1. Mot clé **UPDATE**
2. Le nom de la relation, R
3. Mot clé **SET**
4. Une liste de formules qui déterminent des attributs de R
5. Mot clé WHERE
6. Condition

Exemple:

```
marque (id, nom, classe, pays,prop)
UPDATE marque SET nom='Old'.||nom
    WHERE  marque.id in (SELECT E.marque FROM enr R
          WHERE date_enr < '2000-01-01');
```

5.7 Définition de nouvelles relations

Types de données:

- Caractères
  - **CHARS(n)** : chaîne de longueur fixe (n caractères)
    (chaînes plus courtes sont complétées par des blancs)
  - **VARCHARS(n)** : chaîne d'au plus n caractères
- Numérique
  - **INT (INTEGER)** et **SHORT INT**
  - **FLOAT** (ou **REAL**), **DOUBLE PRECISION**
  - **DECIMAL(n,d)** : **n** chiffres et un point décimal à **d** positions de la droite (0123.45 sera du type DECIMAL(6,2))
- Booléen : **BIT(n)**, **BIT VARYING(n)**

Création, modification et suppression de relation

Exemple de création:

```
CREATE TABLE movieStar (
    name CHAR(30),
    address VARCHAR(255),
    gender CHAR(1),
    birthdate DATE      );
```

Exemple de suppression :

```
DROP movie;
```

Exemples de modification :

```
ALTER TABLE movieStar ADD phone CHAR(6);
ALTER TABLE movieStar DROP birthdate ;
```

Valeurs par défaut

- Une valeur par défaut peut être spécifié lors de la création:

```
gender CHAR(1) DEFAULT '?'
birthdate DATE DEFAULT '0000-00-00'
```

- Si aucune valeur par défaut n'est spécifiée, c'est la valeur **NULL** qui est attribuée aux attributs non explicitement instanciés

Les domaines

Un domaine est un nouveau type avec ses valeurs par défaut et ses contraintes:

```
CREATE DOMAIN moviedomain
     AS VARCHAR(50)  DEFAULT 'unknown';
```

Un domaine peut être modifié avec ALTER DOMAIN et supprimé avec DROP DOMAIN

```
ALTER DOMAIN moviedomain
          SET DEFAULT 'no such title';
```

Les index

La création d'index permet d'optimiser la recherche de tuples satisfaisants une condition spécifique

Exemple:

```
movie(title,year,length,studioName,producer)
CREATE INDEX YearIndex ON movie(year);
```

accélère la requête:

```
SELECT * FROM movie
      WHERE studioName='Disney' AND year =1900;
```

On aurait aussi pu créer un index spécifique pour cette requête

```
CREATE INDEX SudioYearIndex ON movie(studioName,year);
```

Remarques

- les index accélèrent les recherches
- les index rendent plus complexe et coûteux les insertions, suppressions et mises à jour

5.8 Les Vues (relations virtuelles)

- Les tables crées avec CREATE TABLE sont des ***relations persistantes*** :elles sont stockées physiquement et existent jusqu'à leur suppression explicite

- Les vues sont des ***relations virtuelles*** qui peuvent être utilisées dans des requêtes comme des tables mais ***qui n'ont pas d'existence physique*** (elles ne sont pas stockées)

Création et suppression d'une vue

La forme basique d'une instruction de création de vue est :

1. Mots clés **CREATE VIEW**
2. Le nom de la vue
3. Mot clé AS
4. Une requête Q

Q est la définition de la vue : chaque fois que la vue est utilisée (e.g., dans une clause SELECT), SQL se comporte comme si Q était exécuté à ce moment

Suppression d'une vue:

```
DROP VIEW <view name>;
```

Exemples de création et d'utilisation d'une vue

Création d'une vue qui contient les titres et l'année des films produits par 'paramount':

```
CREATE VIEW paramountmovie AS
  SELECT title, year FROM movie
  WHERE studioname='paramount';
```

Exemple d'utilisation:

```
SELECT title FROM paramountmovie
  WHERE year= 1979;
```

Attention : la relation **paramountmovie** ne contient aucun tuple; les tuples recherchés sont ceux de la table **movie**

Exemples de création et d'utilisation d'une vue (suite)

```
movie(title, year,length,studioname,producer);
movieExec(name,address,cert,networth);
CREATE VIEW movieprod(movietitle,prodname) AS
    SELECT title, name FROM movie, movieExec
    WHERE producer=cert;   %renommage des attributs
```

Exemple d'utilisation:

```
   SELECT movietitle FROM movieprod
    WHERE title = 'Gone with the wind';
```

Requête équivalente à:

```
SELECT name AS movietitle FROM movie, movieExec
    WHERE producer=cert AND
                title = 'Gone with the wind';
```

Vues modifiables

- En général il n'est pas possible de modifier une vue car on ne sait pas comment stocker l'information
- Il est possible de modifier une vue dans des cas restreints:
  - Vue construite par la sélection (avec SELECT et non SELECT DISTINCT) de certains attributs d'une relation R
  - La clause WHERE n'utilise pas R dans une sous-requête
  - Les attributs de la clause SELECT doivent être suffisants pour pouvoir compléter le tuple avec des valeurs NULL

Exemples de modification des tables via une vue

```
INSERT INTO paramountmovie VALUES('Star Treck,1979);
```

requête correcte d'un point SQL mais le nouveau tuple aura NULL et non 'paramount' comme valeur de studioname !!

D'ou la nécessité de procéder comme suit:

```
CREATE VIEW paramountmovie
   SELECT studioname,title, year FROM movie
   WHERE studioname='paramount';
INSERT INTO paramountmovie
   VALUES('paramount','Star Treck,1979);
```

Table obtenue:

title	*year*	*length*	*studioname*	*producer*
'Star Treck'	1979	0	**'paramount'**	**Null**

En supposant que la valeur par defaut de length est 0

5.9 Opérations avec NULL

NULL est une valeur spéciale disponible pour tous les types de données

Le résultat des opérations arithmétiques et des comparaisons sur des expressions contenant NULL est inhabituel:

- Le résultat d'une opération arithmétique dont un des termes est NULL est NULL
- Le résultat d'une opération de comparaison dont un des termes est NULL est UNKNOWN

Opérations avec NULL (suite)

- **NULL** est une valeur par défaut mais pas une constante : elle ne peut pas être utilisée explicitement dans une expression
- **X IS NULL (X IS NOT NULL)** : permettent de tester si une variable contient la valeur **NULL**

Exemples:

Supposons que x a pour valeur NULL

x - x → **NULL (et non 0)**

x+5 → **NULL**

x=3 → **UNKNOWN**

Expressions incorrectes: NULL = x , NULL+5

Valeurs de vérité d'une expression contenant UNKNOWN

Pour connaître la valeur de vérité d'une expression contenant UNKNOWN, on peut raisonner de la manière suivante:

TRUE = 1 FALSE= 0 UNKNOWN = ½

X AND Y = min(X,Y) X OR Y = max(X,Y)

NOT X=1-X

D'ou

X	Y	X AND Y	X OR Y	NOT Y
TRUE	UNKNOWN	UNKNOWN	TRUE	UNKNOWN
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN
FALSE	UNKNOWN	FALSE	UNKNOWN	UNKNOWN

Valeurs de vérité d'une expression contenant UNKNOWN (suite)

Dans une clause SELECT ou DELETE *seuls les tuples pour lesquels la clause WHERE a la valeur de vérité TRUE sont retenus*

Exemple

```
SELECT *  FROM movie
    WHERE length <= 120 OR length > 120;
```

Les films dont length est NULL ne sont pas sélectionnés

5.10 Jointures

La jointure conventionnelle (**theta** jointure) peut être effectuée soit à l'aide d'une clause SELECT FROM WHERE soit à l'aide d'une clauses spécifique:

CROSS JOIN → produit cartésien

JOIN ON → jointure sur les attributs spécifiés

```
marque(id, nom, classe, pays,prop)
enr(marque,num,pays,deposant,date_enr)
marque CROSS JOIN enr;
marque JOIN enr ON prop=deposant;
```

Jointure Naturelle

La jointure naturelle entre deux relations s'effectue sur toutes les paires d'attributs qui ont le même nom:

```
moviestar(name,address ,gender,birthday)
movieExec(name,address,cert,networth)
```

```
moviestar NATURAL JOIN movieExec;
```

produit une table avec les attributs suivants: **`name, address, gender, birthday,cert,networth`**

Jointure externe

La jointure externe entre deux relations R1 et R2 permet de retrouver les tuples de R1 et R2 qui ne satisfont pas les conditions de jointure; les attributs "manquants" sont mis à NULL

moviestar(name,address ,gender,birthday)

movieExec(name,address,cert,networth)

(1) **moviestar NATURAL FULL OUTER JOIN movieExec;**

(2) **moviestar NATURAL LEFT OUTER JOIN movieExec;**

(3) **moviestar NATURAL RIGHT OUTER JOIN movieExec;**

requête	name	address	gender	birthdate	cert	networth
1,2,3	Mary ..	Mapple St.	'F'	9/9/99	1234	£100
1,2	Tom …	Cherry Ln	'M'	8/8/88	NULL	NULL
1,3	George ..	Qak Rd.	NULL	NULL	2345	£200

5.11 Récursivité en SQL3

- La récursivité en SQL repose sur la définition de relations IDB (base de données intensionnelle satisfaisant les règles de DATALOG) avec l'instruction WITH
- Il est possible de définir plusieurs relations mutuellement récursive avec l'instruction WITH

L'instruction WITH

Forme générale de l'instruction WITH:

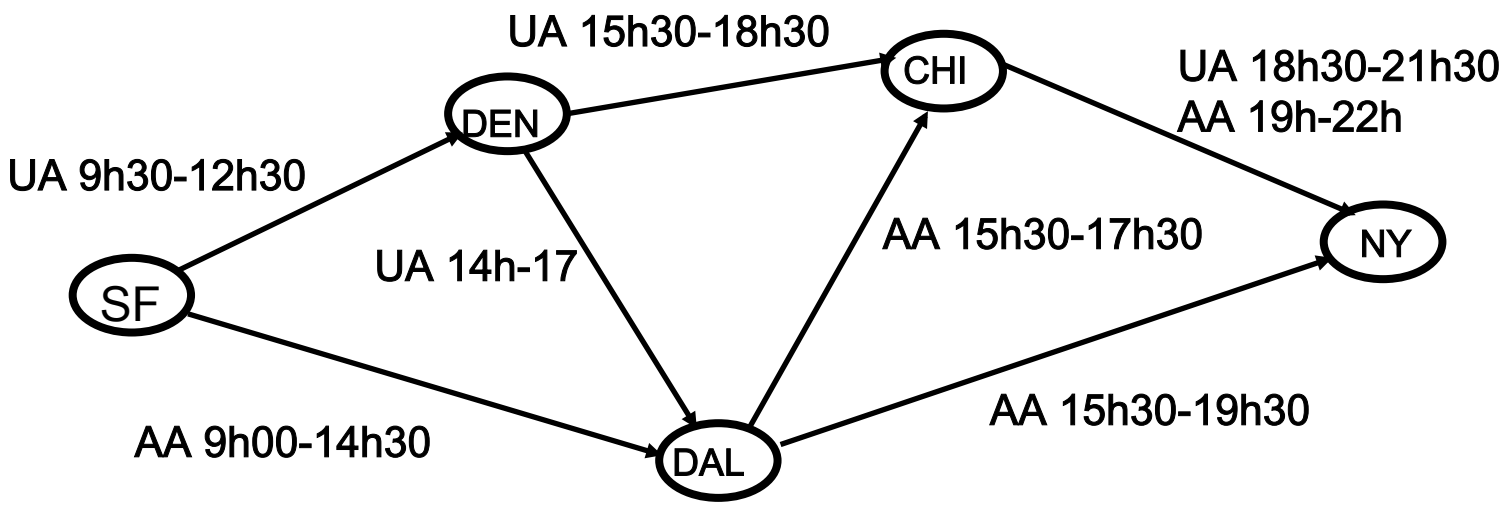
- Mot clé **WITH**
- Une ou plusieurs définitions séparées par des virgules; chaque définition comprend:
  - Le mot clé RECURSIVE s'il s'agit d'une définition récursive
  - Le nom de la relation à définir
  - Le mot clé AS
  - La requête qui définit la relation
- Une requête qui se réfère aux définitions précédentes et construit le résultat de l'instruction WITH

L'instruction WITH (exemple)

Soit la relation

`vol(airline,from,to,departs,arrives)`

Et les données associées :



L'instruction WITH (exemple)

Les pairs de villes connectées par des vols de ce graphe sont définies par la relation récursive:

reaches(x,y) ← flights(a,x,y,d,r)

reaches(x,y) ← reaches(x,z) AND reaches(z,y)

qui se formule en SQL3

```
WITH RECURSIVE reaches(from,to) AS
   (SELECT from, to FROM flights)
   UNION
   (SELECT R1.from, R2.to
        FROM        reaches AS R1,
                    reaches AS R2
                    WHERE R1.to =R2.from)
SELECT * FROM reaches;
```

6 Contraintes et "triggers" en SQL

Lors de l'insertion, la suppression et la mise à jour de la base il faut s'assurer que la base reste correcte:

- **Contraintes**: clés, références externes,restrictions sur les domaines,assertions
- **Triggers**: éléments actifs qui seront déclenchés lors d'un évènement particulier

6.1 Les clés en SQL

- Attribut clé à des valeurs différentes dans tous les tuples de la relation
- Une clé est déclaré lors de la création
  (mots clés PRIMARY KEY et UNIQUE)

Les clés en SQL : exemples

```
CREATE TABLE marque1 (
      id  INTEGER PRIMARY KEY,
         -- ou: id INTEGER UNIQUE
      nom CHAR(30),
      classe INTEGER,
      pays CHAR(2),
      prop INTEGER   );
CREATE TABLE marque2 (
      id  INTEGER,
      nom CHAR(30),
      classe INTEGER,
      pays CHAR(2),
      prop INTEGER,
         PRIMARY KEY (nom, classe,pays));
         -- ou: UNIQUE (nom, classe,pays));
```

Les clés et index

- Un *index* est crée pour chaque clé primaire (et souvent pour les contraintes UNIQUE qui ne portent que sur une seule colonne)

- Dans certaines implémentations de SQL il est possible de déclarer une contrainte d'unicité lors de la création d'un index

  Exemple:

```
CREATE UNIQUE INDEX ncp
    ON marque2(nom, classe,pays);
```

6.2 Contraintes de référence : clés externes

Il est possible d'indiquer qu'un ensemble d'attributs A_1, d'une relation R_1, est une clé externe en référençant un ensemble d'attributs A_2 d'une relation R_2 (R_1 et R_2 peuvent être la même relation)

Implications:

- A_2 a été déclaré comme clé primaire de R_2
- Les attributs de A_1 ne peuvent prendre que des valeurs existantes des attributs de A_2 (induit un ordre de création des tables)

Exemples:

- `enr: marque INTEGER REFERENCES marque1,`
- `client: FOREIGN KEY (name,k,py) REFERENCES marque2(nom,classe,pays);`

Différences entre les contraintes UNIQUE et PRIMARY KEY

- Une relation ne peut contenir qu'une déclaration PRIMARY KEY mais plusieurs déclarations UNIQUE
- Une clé externe ne peut référencer que des attributs qui ont été déclarés comme PRIMARY KEY
- Les tables sont en général triées suivants les attributs de la clé primaire

Maintien des contraintes de référence

Trois stratégies:

- Stratégie par défaut : rejet des modifications qui entraîneraient une incohérence de la base
- Propagation en cascade des suppressions et modifications lors de la suppression ou modification de tuples référencés
- Affectation de `NULL` aux attributs concernés lorsque les tuples référencés sont supprimés ou modifiés

Maintien des contraintes de référence : exemple

```
CREATE TABLE societe (
   id          INTEGER PRIMARY KEY,
   nom         VARCHAR(30),
   ville       VARCHAR(30),
   pays        CHAR(2)      );
CREATE TABLE vente (
   marque      INTEGER REFERENCES marque1,
   vendeur     INTEGER REFERENCES societe
               ON DELETE CASCADE
   acquéreur INTEGER REFERENCES societe
               ON DELETE SET NULL
               ON UPDATE CASCADE,
   date_vente DATE  );
```

Maintien des contraintes de référence : exemple (suite)

```
societe :    id     nom          ville        pays
(s1)         11     a1           v1           FR
(s2)         12     a2           v2           DE
vente :      marque vendeur acquéreur date_vente
(v1)         123        11       12      2000-10-10
(v2)         125        12       11      2001-10-10
```

- Suppression de la marque 123 dans la table **marque1** est impossible tant que **(v1)** existe dans la table **vente**
- Suppression du tuple **(s1)** de la table **societe**

  → Suppression du tuple **(v1)** de la table **vente**

  → Modification de **(v2) :**

  125 12 NULL 2001-10-10

6.3 Contraintes sur les valeurs d'un attribut

Il est possible de limiter les valeurs que peuvent prendre certains attributs en déclarant :

- Des contraintes explicites sur les attributs
- Des contraintes sur les domaines

Contrainte "NOT NULL"

Exemple :

```
CREATE TABLE vente (
    marque      INTEGER REFERENCES marque(id) NOT NULL,
    vendeur     INTEGER ...
```

Implications:

- **marque** ne peut pas prendre la valeur **NULL** lors d'une mise à jour de la table **vente**
- Il n'est pas possible d'utiliser la stratégie **ON DELETE/ON CASCADE SET NULL** pour cet attribut
- Il n'est possible d'insérer un tuple dans la table **vente** sans spécifier la valeur de l'attribut **marque**

Contrainte "CHECK" sur un attribut

Exemple :

```
CREATE TABLE societe (
    id          INTEGER PRIMARY KEY
                CHECK (marque >= 1000),
    nom         VARCHAR(30),
    ville       VARCHAR(30),
    pays        CHAR(2)
                CHECK (pays in ('FR', 'DE', 'US'),  );
```

Condition de **CHECK:** toute expression pouvant apparaître dans **WHERE**

! Condition de **CHECK** dans postgres: condition sur les colonnes de la table

Contrainte sur les domaines

Exemple :

```
CREATE DOMAIN sexe CHAR(1)
        CHECK (VALUE in ('F', 'M',));
```

Ces contraintes sont utiles pour définir des types énumérés ou pour factoriser des contrôles de type

Vérification des contraintes

- La contrainte **CHECK** est vérifiée lors de la modification de l'attribut sur lequel elle porte. Elle peut donc être violée !

  Exemple:

```
CREATE TABLE societe (
  id        INTEGER PRIMARY KEY, ...
  pays      CHAR(2)
     CHECK (pays in (SELECT * FROM lespays));
```

  La contrainte est vérifiée lors de la modification de **pays** dans **societe** mais pas lors de la mise à jour de la table **lespays**

- Il possible de différer la vérification des contraintes jusqu'à la fin d'une transaction qui porte sur plusieurs tables (utilisation du mot clé **DEFERRABLE** dans la déclaration de la contrainte)

6.4 Contraintes globales

Il est possible de limiter les valeurs que peuvent prendre certains attributs en déclarant :

- Des contraintes explicites sur les attributs
- Des contraintes sur les domaines

Contrainte "CHECK" sur un tuple

Exemple :

```
CREATE TABLE movieStar (
      name CHAR(30),
      address VARCHAR(255),
      gender CHAR(1),
      birthdate DATE,
      CHECK (gender='F' OR name NOT LIKE 'Ms.%')    );
```

La contrainte **CHECK** est vérifiée lorsque la condition s'évalue à "True" ou "null".

Les assertions

La forme d'une assertion est:

1. Les mot clés **CREATE ASSERTION**
2. Le nom de l'assertion
3. Mot clé **CHECK**
4. Une condition parenthèsée

Exemple:

```
CREATE ASSERTION vente_enregistrée
    CHECK (NOT EXISTS (SELECT * vente V WHERE  V.marque
    NOT IN (SELECT E.marque FROM enr E)));
```

Les assertions *doivent toujours être vérifiées* : toute modification de la base qui violerait une assertion est rejetée

! Les assertions ne sont pas supportées pas **PostgresSQL**

6.5 Modification des contraintes

- Nommage des contraintes:

```
CREATE TABLE societe (
  id INTEGER
    CONSTRAINT idIsKey PRIMARY KEY, ...
CREATE DOMAIN sexe CHAR(1)
    CONSTRAINT twoValues
    CHECK (VALUE in ('F', 'M',));
CREATE TABLE movieStar (
  name CHAR(30),
  gender CHAR(1),
  ...
    CONSTRAINT rightTitle
    CHECK (gender='F' OR name NOT LIKE 'Ms.%')
   );
```

Utilisation de "ALTER" sur des contraintes

```
ALTER TABLE societe
  DROP CONSTRAINT idIsKey ;
ALTER DOMAIN sexe
   DROP CONSTRAINT twoValues;
ALTER TABLE movieStar
  DROP CONSTRAINT rightTitle;
ALTER TABLE movieStar
  ADD CONSTRAINT NameIsKey PRIMARY KEY(name);
ALTER TABLE societe
  ADD CONSTRAINT nomPays
  CHECK (pays in (SELECT * FROM lespays));
```

6.6 Les "Triggers" (en SQL3)

- Les triggers sont des procédures qui sont activées lors d'un évènement particulier (insertion, suppression ou mise à jour d'une relation particulière, fin d'une transaction)

- Lorsqu'il sont réveillés, les triggers vérifient d'abord une condition : si elle est fausse, rien ne se passe, sinon le trigger peuvent exécuter n'importe quelle séquence d'instructions SQL

Déclenchement et exécution des "Triggers"

- L'action associé à un trigger peut être exécutée avant, après ou à la place de l'événement qui a déclenché le trigger
- L'action associé à un trigger peut accéder aux anciennes et aux nouvelles valeurs des tuples insérés, supprimés ou mis à jour par l'événement qui a déclenché le trigger
- Les événements de mis à jour peuvent se référer à une colonne particulière (ou à un ensemble de colonnes)
- Une condition peut être spécifiée par la clause **`WHEN`** et dans ce cas l'action est uniquement exécutée si la condition est vérifiée
- Il est possible de spécifier une action qui s'applique soit
  - Une seule fois à chaque tuple modifié
  - Une seule fois à tous les tuples modifiés par une opération

Trigger : syntaxe

CREATE TRIGGER ***name***

{ BEFORE | AFTER } { ***event*** [OR ...] }

ON ***table*** [FOR [EACH] { ROW | STATEMENT }]

EXECUTE PROCEDURE ***funcname*** (***arguments***)

- La procédure est exécutée pour chaque colonne modifiée ou pour chaque opération
- Un trigger after a accès à toutes les modifications effectuées

Trigger : exemple

```
CREATE TRIGGER NetWorthTrigger
AFTER UPDATE OF networth ON movieExec
REFERENCING
  OLD AS OldTuple,
  NEW AS NewTuple
WHEN (OldTuple.networth > NewTuple.networth)
  UPDATE movieExec
  SET networth = OldTuple.networth
  WHERE cert=NewTuple.cert
FOR EACH ROW;
```

! Non conforme au standard SQL - Syntaxe oracle

7. Séquences et fonctions en SQL

- **Utiles** pour de nombreuses opérations élémentaires
- **Forte dépendance** vis de l'implémentation de SQL

7.1 Séquences

- **Définition** : compteurs entiers ***persistants*** à travers les sessions
- **Portée**: accessible via toute la base mais en général une séquence est dédiée à une table
- **Utilité**:
  - Permettent d'engendrer automatiquement des identificateurs numériques (évite les gaps)
  - Génération de clés primaires (évite la mise en place de verrous de blocage sur des tables entières)

Séquences : syntaxe

CREATE [TEMPORARY] SEQUENCE *seqname*
[INCREMENT *increment*]
[MINVALUE *minvalue*] [MAXVALUE *maxvalue*]
[START *start*]
[CACHE *cache*]
[CYCLE]

DROP SEQUENCE *seqname* ;

! Non conforme au standard SQL - Syntaxe Postgres

Séquences : utilisation

Les fonctions **`nextval('seqence name')`**, **`currval('seqence name')`** et **`setval('seqence name', newval)`** permettent respectivement, d'obtenir une nouvelle valeur du compteur, d'obtenir sa valeur courante, de modifier la valeur du compteur

Exemples:

```
CREATE SEQUENCE test_seq;

SELECT nextval('test_seq');
nextval
---------
1

SELECT setval('test_seq', 100);
```

Séquences : utilisation (suite)

Les fonctions `nextval('seqence name')`, `currval('seqence name')` et `setval('seqence name', newval)` peuvent figurer dans

- la partie `select` d'une clause de type `select from`
- la partie `values` d'une clause de type `insert`
- la partie set d'une requête de type `update`

Exemple:

```
CREATE TABLE test (index INT, val char(1));
INSERT INTO test
  (SELECT nextval('test_seq'), car FROM test1);
```

Séquences implicites

L'association du type `SERIAL` à un attribut permet de créer implicitement une fonction sequence pour cet l'attribut

Exemple:

```
create table test (index SERIAL , val char(1));
% NOTICE:  CREATE TABLE will create implicit sequence 'test_index_seq' % for SERIAL column 'test.index'
insert into test(val) ( SELECT car FROM test1);
select * from test;
 index | val
-------+----
     1 | a
     2 | b
     3 | c
     4 | d
```

7.2 Fonctions et opérateurs

- Nombreuses fonctions prédéfinies: mathématiques, booléennes, manipulation et conversion de chaînes (cf. documentation)
- Les opérateurs diffèrent des fonctions par les points suivants:
  - Les opérateurs sont des symboles (pas des noms),
  - Les opérateurs sont en général binaires et peuvent s'écrire de manière infixe
- Opérateurs et fonctions peuvent être utilisés dans les clauses `SELECT`, `INSERT` et `UPDATE` ainsi que pour la définition de fonctions spécifiques (définies par l'utilisateur)

Opérateurs et fonctions prédéfinis: exemples

```
SELECT sqrt(2.0);
    1.4142135623731

SELECT * FROM test;       SELECT upper(val) FROM test;
   Index        val
   1            a             A
   2            b             B
   3            c             C


SELECT 2+index^2 FROM test;
    3
    6
    11
```

Fonctions spécifiques

- Des fonctions peuvent être définies dans différents langages: SQL , PL/PGSQL ,PL/TCL, PL/Perl ,C
- **Syntaxe** :

CREATE FUNCTION *name* ([*ftype* [, ...]])
RETURNS *rtype*
AS *definition*
LANGUAGE '*langname*'
[WITH (*attribute* [, ...])]
CREATE FUNCTION *name* ([*ftype* [, ...]])
RETURNS *rtype* AS *obj_file* , *link_symbol* LANGUAGE 'langname' [WITH (*attribute* [, ...])]

Fonctions spécifiques: exemples

Fonction SQL pour la convertion d'une temperature de degrés Fahrenheits en degrés centigrades

```
CREATE FUNCTION FahrToCelc(float)
 RETURNS float
    AS 'SELECT ($1 - 32.0) * 5.0 / 9.0;'
    LANGUAGE SQL;

SELECT FahrToCelc(68);
           20

INSERT INTO test(val, temp)
 (SELECT val, FahrToCelc(tp^2)
 FROM test2, test1);
```

8 Dépendances fonctionnelles et Formes Normales

Dépendances Fonctionnelles

Fondamentales pour ***éliminer les redondances***

Les dépendances fonctionnelles sont ***associées au schéma et non à une instance particulière***

Intuition:

Dans une relation, ***certains attributs en "déterminent" d'autres***, i.e., il n'y a pas deux tuples ayant les mêmes valeurs pour le premier ensemble d'attributs sans avoir également les mêmes valeurs pour le deuxième ensemble

Dépendance fonctionnelle: définition

Soient **r** une instance de la relation **R**, **X** et **Y** deux sous-ensembles d'attributs de **R**.

On dit que **r** satisfait la **dépendance fonctionnelle** $X \rightarrow Y$

et l'on note $r \models X \rightarrow Y$

ssi $\forall t_1 \in r \quad \forall t_2 \in r \quad (t_1.X = t_2.X \rightarrow t_1.Y = t_2.Y)$

Si r satisfait plusieurs plusieurs dépendances fonctionnelles, df_1, df_2, ..., on note alors : $r \models df_1, df_2, \ldots$

La contrainte $X \rightarrow \varnothing$ est toujours satisfaite.

La contrainte $\varnothing \rightarrow X$ signifie que la projection de la relation **r** sur **X** est constante

Exemple :

ENREG={*NumE, Pays, NomM, Classe, Date, IdDep*}

Les dépendances vérifiées par chaque instance (en supposant un seul déposant par enregistrement) :

df_1 : NumE,Pays → NomM, Date

df_2 : NumE,Pays → Classe, IdDep

df_3 : NomM,Pays, Classe → NumE

Dépendances "déduites" :

df_4 : NumE,Pays → NomM, Date Classe, IdDep

Calculs sur les Dépendances Fonctionnelles

Déterminer si un ensemble de dépendances ne contient pas de **redondances**.

Représenter ces dépendances sous une **forme minimale**.

Exemple

Soit $R = \{A, B, C, D\}$ un schéma de relation et l'ensemble des dépendances fonctionnelles :

$$DF = \{ A \rightarrow B, B \rightarrow C, AC \rightarrow D \}$$

Si r satisfait toutes les contraintes de DF, alors r satisfait également les dépendances suivantes:

$A \rightarrow C$ $A \rightarrow AC$ $A \rightarrow D$

$A \rightarrow ABCD$ $CD \rightarrow D$,

.....

Calculs sur les Dépendances Fonctionnelles (suite)

Implication de dépendances:

*Soient **DF** et **DF'** deux ensembles de dépendances fonctionnelles définies sur un schéma de relation **R**. On dit que **DF** implique **DF'**, et l'on note* $DF \models DF'$ *ssi pour toute instance **r** de la relation **R**, on a*

$$r \models DF \Rightarrow r \models DF'$$

Exemple

$A \rightarrow B,\ A \rightarrow C \models A \rightarrow BC$

Inférences de dépendances fonctionnelles

Les axiomes suivants permettent de démontrer toute implication entre dépendances fonctionnelles.

Ce système inférentiel est noté $\vdash$ par opposition à $\models$ qui dénote l'implication sémantique.

1. ***Réflexivité***		$\vdash X \rightarrow X$
2. ***Augmentation***	$X \rightarrow Y$	$\vdash XZ \rightarrow Y$
3. ***Addition***	$X \rightarrow Y,\ X \rightarrow Z$	$\vdash X \rightarrow YZ$
4. ***Projection***	$X \rightarrow YZ$	$\vdash X \rightarrow Y$
5. ***Transitivité***	$X \rightarrow Y,\ Y \rightarrow Z$	$\vdash X \rightarrow Z$
6. ***Pseudo-transitivité***	$X \rightarrow Y,\ YZ \rightarrow W$	$\vdash XZ \rightarrow W$

Couvertures

Un des problèmes posés par les dépendances fonctionnelles, est de minimiser le nombre de dépendances et d'attributs

On arrive grâce au système inférentiel à établir une couverture minimale de dépendances fonctionnelles

Clés d'une relation

$\{A_1, \ldots, A_n\}$ est une clé d'une relation *r* si :

- Les attributs $\{A_1, \ldots, A_n\}$ déterminent fonctionnement tous les autres attributs de la relation *r*
- Aucun sous ensemble de $\{A_1, \ldots, A_n\}$ ne détermine fonctionnellement tous les autres attributs de *r* (la clé doit être minimale)

Super clé : ensemble d'attributs qui contiennent une clé

Remarque:

Il est possible d'avoir plusieurs clés dans une relation

Détermination des clés d'une relation

- Si *r* est une relation ***n-m*** entre deux entités E_1 et E_2, alors les clés de E_1 et E_2 sont des clés de *r*
- Si *r* est une relation ***n-1*** d'une entité E_1 vers une entité E_2, alors les clés de E_1 sont des clés de *r* (mais les clés de E_2 ne sont pas des clés de E_1)
- Si *r* est une relation ***1-1*** entre deux entités E_1 et E_2, alors les clés de E_1 et E_2 sont des clés de *r* (il n'existe pas de clé unique)

Fermeture d'un ensemble d'attributs

Soit $\{A_1, \ldots, A_n\}$ un ensemble d'attributs et S un ensemble de dépendances fonctionnelles.

La fermeture de $\{A_1, \ldots, A_n\}$ par S est l'ensemble d'attributs B tel que toutes les dépendances de S satisfont aussi

$\{A_1, \ldots, A_n\} \rightarrow B$

C'est à dire que $\{A_1, \ldots, A_n\} \rightarrow B$ découle de S

On note la fermeture $\{A_1, \ldots, A_n\}^+$

Calcul d'une fermeture (algorithme de saturation)

X est l'ensemble des attributs qui correspondra à la fermeture de $\{A_1,\ldots,A_n\}$

1) $X \leftarrow \{A_1,\ldots,A_n\}$ *% Initialisation*

2) Rechercher une dépendance de la forme $B_1,\ldots,B_n \rightarrow C$ tel que $\{B_1,\ldots,B_n\}$ soient dans **X** mais non **C** . Ajouter **C** dans **X**

3) Répéter l'étape 2 jusqu'au point fixe de **X** (plus rien ne peut être ajouté à **X**)

X contient la fermeture $\{A_1,\ldots,A_n\}^+$

Fermeture et clé

$\{A_1, \ldots, A_n\}^+$ est l'ensemble des attributs d'une relation si et seulement si $A_1, \ldots, A_n$ est une super clé de la relation considérée

→ on peut tester si $\{A_1, \ldots, A_n\}$ est une clé d'une relation R en vérifiant que $\{A_1, \ldots, A_n\}^+$ contient tous les attributs de la relation R

Formes Normales.

- Décomposer les relations d'un schéma en des relations plus simples et plus "indépendantes"
- Faciliter la compréhension
- **Éliminer les redondances**
- Améliorer les aspects incrémentaux
- Faciliter la distributivité sur des sites répartis

Première Forme Normale

On dit qu'un schéma relationnel R est en première forme normale (1NF) ssi les valeurs des attributs sont atomiques (ni *set_of*, ni *list_of,….*)

Deuxième Forme Normale

Attribut non clé:

*On dit qu'un attribut A est **non clé** dans **R** ssi A n'est élément d'aucune clé de **R**.*

2èmeForme Normale:

*On dit que **R** est en **deuxième forme normale** (2NF)*

ssi :

- *Elle est en 1NF*
- ***Aucun attribut non clé** ne dépend fonctionnellement **d'un attribut clé***

Deuxième Forme Normale (exemple)

Schéma 1:

joueur(Personne, Sport, Taille)

Personne → Taille

→ la table joueur contient des redondances car le même couple (*personne_x*, *taille_y*) va apparaître autant de fois que *personne_x* pratique de sports

Schéma 2 :

pratique(Personne, Sport)
hauteur(Personne, Taille)
où
pratique= $\pi_{Personne,\ Sport}$(**joueur**)
hauteur= $\pi_{Personne,\ Taille}$(**joueur**)

La table originale **joueur** peut alors être retrouvée par la jointure:
joueur= pratique ⋈ hauteur

Troisième Forme Normale

Une relation est en troisième forme normale si elle est en 2NF et que ***les attributs non clés sont mutuellement indépendants***

Définition : troisième Forme Normale

On dit qu'un schéma relationnel ***R*** *est en* ***troisième forme normale*** *(3NF)* ***ssi*** *:*

1) Elle en 2NF

2) Tout attribut n'appartenant pas une clé ne dépend pas d'un attribut non clé

Troisième Forme Normale (exemple)

VOITURE (NVH, TYPE, MARQUE, PUISS, COULEUR):

La clé est (NVH), et on a les DF :

TYPE → MARQUE et TYPE → PUISS

3 NF:

VEHICULE (NVH, TYPE, COULEUR)

MODELE (TYPE, MARQUE, PUISS)

Processus de décomposition d'une table

Objectifs:

- éviter des redondances
- minimiser les risques d'erreurs lors des mises à jour

Moyens:

- Remplacer une table par des projections selon certains attributs
- Pour ne pas perdre d'informations il faut :
  - Pouvoir reconstruire la table initiale par jointure
  - Pouvoir reconstituer les contraintes initiales portant sur cette table

Algorithme de normalisation par décomposition

- Décomposer, toute table qui n'est pas en 3NF, en deux sous-tables obtenues par projection:
  - Repérer dans ***R*** une dépendance du type:
    $DF \models L \rightarrow A$ avec L non clé, $A \notin L$ et $A \notin$ clé
  - On projette alors ***R*** en deux tables:
    - une sur les attributs ***R*** -{*A*}
    - l'autre sur les attributs *AL* (la deuxième table possède *L* comme clé)

- On réitère alors le processus, en sachant qu'une table binaire est toujours en 3NF

Remarque:
L'algorithme de décomposition est NP-complet et la décomposition peut produire plus de tables que nécessaire pour l'obtention d'une 3NF

Algorithme de normalisation par décomposition (exemple)

VOITURE (NVH, TYPE, MARQUE, PUISS, COULEUR)

avec

TYPE → MARQUE et TYPE → PUISS

admet comme décomposition

VEHICULE (NVH, TYPE, COULEUR)

MODELE (TYPE, MARQUE, PUISS)

Limite de la 3NF

RELATION VIN (CRU, PAYS, REGION)

CRU	*PAYS*	*REGION*
CHENAS	France	BEAUJOLAIS
JULIENAS	France	BEAUJOLAIS
CHABLIS	France	BOURGOGNE
CHABLIS	USA	CALIFORNIE

DF:

CRU,PAYS → REGION

REGION → PAYS

Forme Normale de Boyce-Codd (BCNF)

Une relation ***R*** *est sous forme normale de Boyce-Codd ssi chacun des attributs ne dépend fonctionnellement que des clés (en dehors des super clés ou de lui-même)*

Autrement dit, quelque soit X et A,

$$(DF \vdash X \rightarrow A) \Rightarrow (A \notin X \text{ ou } X \text{ superclé})$$

Une relation est en BCNF si et seulement si les seules dépendances fonctionnelles non triviales sont celles pourlesquelles une clé détermine un ou plusieurs attributs.

Il n'est pas toujours possible de décomposer une relation en un schéma équivalent composé de relations en BCNF

Forme Normale de Boyce-Codd (BCNF)
Exemple

VIN (CRU, PAYS, REGION)

- 2 Clés candidates : (CRU,PAYS) (CRU, REGION)
- Il existe deux DFs Elémentaires :
  - (CRU,PAYS) → REGION
  - REGION → PAYS

Une décomposition possible :

1. CRUS (CRU, REGION)

2. REGIONS (REGION, PAYS)

- **ON A PERDU LA DF : (CRU, PAYS) → REGION**

Synthèse

- **2NF**

  R(<u>A, B</u>, C, D, E) B → C

  R1(<u>B</u>, C)

  R2(<u>A, B</u>, D, E)

- **3NF**

  R(<u>A, B</u>, C, D, E) C → D

  R1(<u>C</u>, D)

  R2(<u>A, B</u>, C, E)

- **BCNF**

  R(<u>A, B</u>, C, D, E) C → B et (A,C) est clé secondaire

  R1(<u>C</u>, B)

  R2(<u>A, C</u>, D, E)

9 Gestion des droits et environnements SQL

9.1 Structure générale de l'environnement SQL

- Schéma: collection de tables, vues,…
- Catalogue: collection de schéma
  - Un nom de schéma est unique pour un catalogue
  - Contient un schéma spécifique `INFORMATION_SCHEMA`
- Environnement DBMS : ensemble de clusters

Opérations sur l'environnement SQL

- **`CREATE SCHEMA <nom_de schéma>`**

  Déclaration d'un nouveau schéma : tous les nouveaux objets crées vont être ajoutés à ce schéma

- **`SET SCHEMA <nom_de schéma>`**

  `nom_de schéma` devient le schéma courant : tous les nouveaux objets crées vont être ajoutés à ce schéma

- On peut aussi associer un schéma :
  - Un jeu de caractères muni d'une relation d'ordre et d'opérations de conversion
  - Des droits d'accès

Opérations sur l'environnement SQL (suite)

- **`CREATE CATALOG <nom_de catalogue>`**

  Déclaration d'un nouveau catalogue

- **`SET CATALOG <nom_de catalogue>`**

  `nom_de catalogue` devient le catalogue courant auquel vont être ajoutés tous les nouveaux schémas crées

- Nom complet d'une table

  <catalog_name>.<schema_name>.<table_name>

9.2 Gestion des droits d'accès sous SQL

- Les droits (appelés "privilèges") sont accordés à :
  - Un utilisateur ou groupe d'utilisateur
  - **`PUBLIC`** : id générique pour l'ensemble des utilisateurs

- Les "privilèges" concernent :
  - SELECT : se réfère à une relation ou à une vue
  - INSERT " "
  - DELETE " "
  - UPDATE " "
  - REFERENCES: requis pour vérifier la contrainte concernée
  - USAGE : se réfère à un domaine

Gestion des droits d'accès sous SQL: exemple

```
INSERT INTO Studio(name)
  SELECT DISTINCT studioname
  FROM Movie
  WHERE studioname NOT IN (SELECT name FROM
  Studio)
```

L'exécution de cette transaction nécessite les "privilèges" suivants :

- **INSERT** pour la table **Studio** (il serait suffisant de bénéficier de ce privilège pour l'attribut **name** de la table **Studio)**
- **SELECT** pour les tables **Studio** et **Movie**

Gestion des droits d'accès sous SQL: principes

- Le créateur d'un schéma possède tous les privilèges pour ce schéma
- L'utilisateur est identifié lors de sa connexion au serveur
- Des privilèges pourront être accordé à un module (programme d'application, e.g., session interactive, code SQL incorporé dans un texte du langage hôte, fonction ou procédure stockée) ou à un ensemble d'utilisateurs
- Une transaction ne peut être exécutée que si toutes les opérations possèdent les privilèges requis

Accord de droits : l'instruction GRANT

1. Le mot clé **GRANT**
2. Une liste de privilèges, e..g, **SELECT, ..., INSERT(name)**
3. Le mot clé **ON**
4. Un objet de la base **(vue, table)**
5. Le mot clé **TO**
6. Une liste d'utilisateurs ou le mot **clé PUBLIC**
7. [les mots clés **WITH GRANT OPTION**]

Exemple :

```
GRANT SELECT,INSERT ON marque TO durand WITH GRANT
OPTION
```

Accord de droits : l'instruction GRANT (suite)

- L'utilisateur qui exécute une instruction **GRANT** doit posséder tous les privilèges accordés ainsi que la **"GRANT OPTION"** sur les objets concernés
- Révocation des privilèges:

```
REVOKE <privilèges list> ON
    <database_name> TO <user_name>
  [CASCADE | RESTRICT]  % Pour propager ou non le
                        % retrait de droits
```

- Remarques:
  - Il est souhaitable de créer un diagramme des privilèges
  - Le retrait d'un privilège général n'ôte pas un privilège particulier

Accord de droits : exemple

```
   Step  By     Action
-------------------------------------------------------------
      1      U    GRANT INSERT ON R TO V
      2      U    GRANT INSERT(A) ON R TO V
      3           U    REVOKE  INSERT  ON  R  FROM  V
                RESTRICT
```

V conserve la possibilité d'insérer A dans V

Création de schémas : exemple (1)

-- Création de la base tpmarquedeposees

```
drop database tpmarquedeposees;
create database tpmarquedeposees ;
```

-- Création dans la base '*tpmarquedeposees* '
-- d'un schéma '*donnees*'

```
\c tpmarquedeposees;
create schema donnees;
set search_path to donnees, public ;
```

Création de schémas : exemple (2)

-- Initialisation du schéma 'donnees'

```
DROP TABLE marque ;      DROP TABLE societe;
DROP TABLE classe ;    DROP TABLE Pays;
…
CREATE TABLE classe (
    num  INT  NOT NULL PRIMARY KEY,
    libelle VARCHAR(30)  NOT NULL );
\copy classe from 'classe'

CREATE TABLE pays (
    code  CHAR(2) NOT NULL PRIMARY KEY,
    nom  VARCHAR(50));
\copy pays from 'pays'

…
```

Création de schémas : exemple (3)

-- Mise en place des droits sur les tables du schema 'donnees'

-- Pour que les relations du schema soient visibles

```
grant usage  on  schema donnees to public;
```

-- Pour que les tables pays, societe,… du schema puissent être référencées et lues

```
grant select on table pays to public;
grant references on table pays to public;
grant select on table societe to public;
grant references on table societe to public;
…
```

-- Création des schema pour les utilisteurs

```
create schema authorization test;
alter user test set search_path to test,      donnees,
public ;
```

10 Gestion de la concurrence: transactions, sérialisation, verrouillage gestion de l'intégrité

10.1 Transactions

- **Problème**:

  Accès concurrentiel par plusieurs clients d'un ou plusieurs schémas aux données d'une base, en lecture comme en modification

- **Notion de transaction**

  Un ensemble d'opérations de lecture/écriture,

  opérées par un même client sur une base,

  sera considérée comme effectué en un seul instant

  (par rapport aux autres clients)

- **Objectif** :

  Assurer la cohérence de la base vis-a-vis des contraintes

  (ne s'applique qu'à des opérations sur les données)

Définition d'une transaction SQL

- Une transaction est un ensemble séquentiel d'opérations de type DML:
  `select, insert, update, delete`
  effectuées sur les tuples d'une base de données, et délimité dynamiquement de la manière suivante :
  - **Début de Transaction** : toute commande SQL initiale d'une session, ou suivant la fin de la transaction précédente.
  - **Fin de Transaction** :
    - **`commit, rollback, disconnect`**
    - Toute commande de type DDL
    - Anomalie système ou erreur de programme (suivie en général d'un **`rollback`**)
- Une seule transaction est attachée à un client donné
- Postgres:

  `BEGIN;` … **`COMMIT;`**

Définition fonctionnelle d'une transaction

- Une transaction est dite individuellement correcte ssi toutes les contraintes (implicites ou explicites) régissant cette base sont satisfaites à la fin des opérations (en supposant qu'elle soit seule à modifier la base)

  → Interdiction d'exécuter partiellement une transaction

  (puisque la cohérence individuelle de cette transaction vis-a-vis des contraintes ne serait plus assurée)
- Exemple:

  On ne peut effectuer un retrait sur un compte bancaire d'un usager sans affecter le compte consolidé de l'agence qui contient le cumul de tous les comptes clients.
- L'opération `rollback` permet de défaire les modifications déjà effectuées par une transaction.

Gestion concurrentielle des transactions

- Le fait que deux transactions soient individuellement correctes vis-a-vis des contraintes, ne garantit pas que leur exécution concurrentielle le soit.
- Exemple:
  - **T1 lire A**
  - **T2 lire A**
  - ***T1 écrire A*** *% opération perdue*
  - T1 lire B
  - T1 écrire B
  - **T2 écrire A**
  - T2 lire B
  - T2 écrire B

Avec T1: virement(A,B,100) et T2: virement(A,B,200)

Ordonnancement des transactions

- **Ordonnancement** :
  - T1: (T1, lire, A), (T1, écrire, A)
  - T2: (T2, lire, A), (T2, lire, B), (T2, écrire, A), (T2, écrire, B)
  - O: (T1, l, A), (T2, l, A), (T1, é, A), (T2, l, B), (T2, é, A), (T2, é, B)

- **Ordonnancement sériel** : permutation quelconque des transactions
- **Ordonnancement sériable** : équivalent à un ordonnancement sériel
- **Instructions conflictuelles**: deux instructions qui opèrent sur la même entité et dont l'une au moins est une écriture
- **Équivalence d'ordonnancements**: 2 ordonnancements O et O' du même ensemble de transactions sont équivalents si pour toutes opérations conflictuelles p et q de O et O', p est avant dans O ssi p est aussi avant q dans O'

Ordonnancement des transactions - Exemples

- **O1 n'est pas sériable car il n'est pas équivalent à:**
  - T1T2: écritures dans un ordre différent
  - T2T1: (T1,l,X), (T2,é,X) dans des ordres différents

  T1: (T1, lire, X), (T1, écrire, X)

  T2: (T2, lire, X), (T2, écrire, X)

  O1: (T1, l, X), (T2, l, X), (T2, é, X), (T1, é, X)
- **O2 n'est pas sériable**:
  - T1: (T1, lire, X), (T1, écrire, Y), (T1, lire, X), (T1, écrire, Z)
  - T2: (T2, lire, X), (T2, écrire, X)
  - O2: (T1, l, X), (T1, é, Y), (T2, l, X), (T2, é, X), (T1, l, X), (T1, é, Z)

T1: {Lire X; Y :=X; écrire Y; lire X; Z:=X; écrire Z}

T2: {lire X; X:=2*X;écrire X}

Après un ordonnancement sériel Y et Z seront identiques mais pas après O2

Gestion concurrentielle des transactions

Théorème : principe de validité de transactions séquentielles

Si dans un univers concurrentiel, des transactions ***individuellement correctes*** *sont exécutées* ***séquentiellement,*** *alors, à tout instant (entre deux transactions),* ***la base sera cohérente*** *vis-a-vis des contraintes.*

Corollaire: *Un ordonnancement sériable est correct*

Notations

Soit

- $T = \{\Delta t_1, \ldots, \Delta t_i, \ldots, \Delta t_n\}$ une transaction effectuant les modifications Δt_i sur des tuples t_i .
- $\Delta c = f(LC, \Delta M)$ où :
  - Δc est une modification à effectuer sur le tuple *c*, pour satisfaire les contraintes,
  - LC est un ensemble de tuples dont dépend le calcul de Δc, et qu'on appelle également **"lectures critiques"**
  - ΔM désigne l'ensemble des modifications (données par l'usager) dont vont également dépendre le calculs.

Exemple de Consolidation de Comptes

Soit une table de comptes clients d'agences bancaires

`client(IdCl, IdAg, Solde, ...)`

et une table de comptes consolidés par agence

`agence(IdAg, Solde, ...)`

la contrainte de consolidation est la suivante :

$\forall a \in \texttt{agence} : a.\texttt{Solde} = \sum(c.\texttt{Solde})$ avec c∈client/c.IdAg=a.IdAg

Considérons la modification consistant en un dépôt sur le compte 1222 de l'agence 300 et notée Δc_{1222}

→ deux façons de traiter cette contrainte de consolidation

Première Solution : transaction T_1

Calcul de la nouvelle valeur du solde d'agence par la formule

$$\Delta_{a300.solde} = \sum(c.\text{Solde})_{\{c \in client/c.IdAg=300\}} - c_{1222}.\text{solde} + \Delta(c_{1222}.\text{solde})$$

La modification du solde d'agence dépend alors de la lecture de tous les comptes clients de cette agence. La transaction correspondante

$T_1 = \{\Delta\, c_{1222},\ \Delta\, a_{300} = f_1(\, c_{33}, c_{37}, ..., c_{1222}, \ldots c_n, a_{300}, \Delta\, c_{1222})\ \}$

ne sera correcte que si l'on est assuré qu'aucune modification n'intervient entre la lecture de tous les comptes clients de l'agence et les modifications simultanées du compte d'agence 300 et du compte client 1222

Deuxième Solution : transaction T_2

Calcul de la nouvelle valeur du solde d'agence par la formule:

$$\Delta_{a300.solde} = a_{300}.solde - c_{1222}.solde + \Delta(c_{1222}.solde)$$

Cette transaction peut se représenter par la formule:

$$T_2 = \{ \Delta c_{1222}, \Delta a_{300} = f_2(a_{300}, c_{1222}, \Delta c_{1222}) \}$$

Pour être valide en univers concurrentiel, aucune modification externe du compte client c_{1222} et du compte d'agence a_{300} ne doit intervenir entre la lecture de ces comptes et les modifications effectuées

→ Le verrouillage à effectuer ici est donc bien moindre que pour la transaction T_1

Principe de validité de transactions concurrentielles

Considérons un ensemble de transactions T_i

individuellement correctes, de la forme

$T_i(d_{i1}, \ldots, d_{ik}) = \{ \Delta t_{i1}, \ldots, \Delta t_{ik}\}$, dépendant chacune en lecture

des tuples $d_{i1}, \ldots, d_{ik}$ pour le respect des contraintes. Si

1. les modifications de chacune de ces transactions sont **effectuées d'un seul bloc** et sans interruption en fin de transaction,
2. **aucune modification** n'est effectuée sur les d_{ij} par une autre transaction entre leur lecture et la modification des t_{ij}

Alors on est assuré d'avoir à tout moment une base

cohérente vis-a-vis des contraintes

10.2 Sémantique SQL des Transactions

Visibilité externe des modifications en SQL:

- L'ensemble des **modifications** effectuées dans une transaction n'est **rendu visible** aux autres transactions (et **rendu** effectif dans la base) qu'au moment du **`commit`** qui termine la transaction
- L'ensemble des **modifications** effectuées dans une transaction est exécuté en un seul bloc, et sans recouvrement par d'autres commit (deux **commit** qui ont en commun la modification d'une même ligne seront exécutés séquentiellement)

Sémantique SQL des Transactions (suite)

Visibilité Interne des modifications :

Toutes les modifications effectuées lors d'une transaction sont cependant visibles à l'intérieur de cette même transaction: toutes les modifications effectuées sont stockées dans une *mémoire locale à la transaction*, et tout se passe comme si la transaction travaillait sur une copie de la base

Sémantique SQL des Transactions (suite)

Instantanéité des commandes SQL : consistance de lecture

- Toute commande SQL est effectuée sans possibilité d'interruption et avec un ensemble de valeurs lues ou écrites correspondant à un même instant t (la réalisation d'une transaction externe **ne peut modifier les lectures effectuées dans une seule commande,** aussi complexes soient-elles).
- Une même commande de modification ne peut affecter les lectures qui en font partie; tout se passe comme si ces lectures étaient effectuées avant la commande.

10.3 Gestion des transactions: synthèse

- **Sérialisation** : évite que deux clients n'utilise la même ressource

  → Verrouillage du plus petit ensemble possible de relations

- **Atomicité** : évite qu'une contrainte soit violée du fait d'une panne technique (e.g., virement d'un montant M d'un compte C_1 vers un compte C_2)

  → Travail dans une copie de l'espace de travail

Gestion des transactions: options par défaut

Par défaut en SQL les transactions sont sérialisées

- **`COMMIT`** : les opérations effectuées par la transaction sont répercutées sur la base et rendues visibles aux autres utilisateurs
- **`ROLLBACK`** : permet de défaire les modifications effectuées sur la transaction en cas d'anomalie ou d'erreur

Gestion des transactions: choix spécifiques

- Les transactions qui n'effectuent que des lectures peuvent être exécutées en parallèle

```
SET TRANSACTION READ ONLY;
```

- **SQL2** : permet de rendre accessible en lecture les données modifiées par une transaction avant la fin de celle-ci (i.e., avant le **COMMIT**) → "Dirty reads"

```
SET TRANSACTION ISOLATION LEVEL READ WRITE
UNCOMMITTED;
```

- Options Postgres: **read committed, serializable isolation levels.**

"Dirty reads" : exemple "virement bancaire"

- **Virement bancaire** effectué par un programme **P** qui effectue la séquence d'opérations:
  1. Ajout du montant M au compte (b)
  2. Test si le montant du comptant (a) est suffisant pour effectuer le virement:
     i. Si non, retrait du Montant M du compte (b) et abort
     ii. Si oui, retrait du Montant M du compte (a) et **`COMMIT`**

- Si P est exécuté de manière sérialisée, alors la transaction est correcte;

"Dirty reads" : exemple "virement bancaire" (suite)

Si des "dirty reads" sont autorisées et que:

- Les soldes de comptes A1, A2 et A3 sont de 100 €, 200 € et 300 €;
- La transaction T_1 transfert 150 € de A1 vers A2
- La transaction T_2 transfert 250 € de A2 vers A3

La séquence suivante d'opérations peut se produire:

1. T_2 exécute l'étape 1 et ajoute 250 € au compte A3 (solde 550 €)
2. T_1 exécute l'étape 1 et ajoute 150 € au compte A2 (solde 350 €)
3. T_2 exécute le test de l'étape 2 et autorise le transfert de 250 € de A2 vers A3
4. T_1 exécute le test de l'étape 2 et refuse le transfert
5. T_2 soustrait 250 € de A2 (nouveau solde 100 €) et COMMIT
6. T_1 soustrait 150 € de A2 (nouveau solde -50 €) et ABORT

→"dirty reads" violent les contraintes et rendent la base inconsistante

"Dirty reads" : exemple "réservation aérienne"

- Réservations aériennes effectuées par un programme **P** qui effectue la séquence d'opérations:
  1. Identification d'un siège libre et mise à 1 de la variable **`occ`**; si aucun siège n'est libre abort
  2. Demande d'accord au passager. S'il est d'accord exécution de **`COMMIT`**; sinon libération du siège (mise à 0 de la variable **`occ`**) et retour à l'étape 1

- Si P est exécuté de manière sérialisée, alors la transaction est correcte

"Dirty reads" : exemple "réservation aérienne" (suite)

Supposons que des "dirty reads" soient autorisées et que deux transactions T_1 et T_2 effectuent une réservation en même temps

Si T_1 réserve le siège S et que T_2 effectue le test de disponibilité pendant que le client de T_1 se décide, alors le client de T_2 n'aura pas la possibilité de réserver le siège S

→ Les "dirty reads" n'entraînent pas d'incohérence de la base pour les réservations aériennes (le client de T_2 perd la possibilité de réserver un siège qui risque de se libérer par la suite)

→ Les "dirty reads" rendent les transactions plus fluides

Gestion des transactions: niveaux d'isolation de SQL2

- **`SET TRANSACTION ISOLATION LEVEL READ COMMITTED;`**

  Interdit les "dirty read" mais autorise des réponses différentes pour plusieurs lectures des mêmes données dans une même transaction

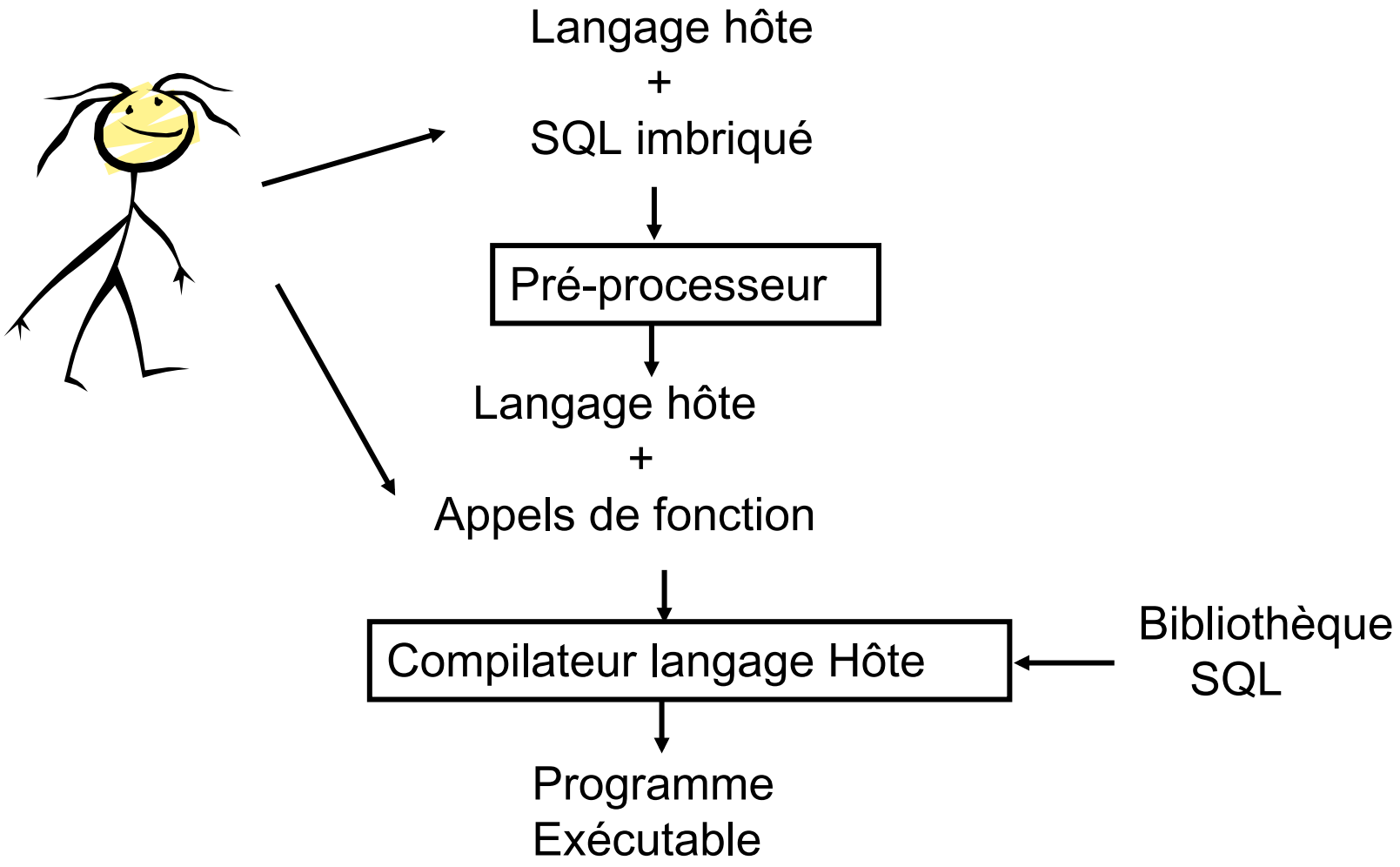
- **`SET TRANSACTION ISOLATION LEVEL READ REPEATABLE;`**

  Les réponses obtenues lors de la première lecture sont inclues dans les réponses obtenues pour les autres les lectures des mêmes données dans une même transaction

11 Utilisation de SQL dans un langage hôte

- La plupart des applications qui utilisent desbases de données nécessitent l'utilisation conjointe de SQL et d'un langage impératif universel
- Les implémentations SQL2 doivent supporter les langages **`ADA, C, COBOL, Fortran, M, Pascal, PL/1`**
- Un des problèmes majeur pour la collaboration réside dans la différence fondamentale des structures de données.

Interface SQL / langage hôte : Schéma général



Interface SQL / langage C : principes

- La communication s'effectue via des variables accessibles en C et SQL (préfixées par : en SQL)

```
EXEC SQL BEGIN DECLARE SECTION;
  ...
EXEC SQL END DECLARE SECTION;
```

- Les instructions SQL sont préfixées par les mots clés **EXEC SQL** dans de nombreux langages hôtes

- Une variable SQLSTATE, tableau de 5 caractères, permet de récupérer le code d'exécution d'une fonction
  - 00000 : pas d'erreur
  - 02000 : un tuple demandé n'a pas été trouvé

- Les curseurs permettent d'échanger des ensembles de tuples

Interface SQL / C : exemple de requête SQL avec une seule réponse

```
% Find the length of the movie for a given title and
  year.
% Movie(title, year, length, inColor, sName, presC#)
EXEC SQL BEGIN DECLARE SECTION;
  char theTitle[20];     int theYear;
  int theLength;   char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;
... /* assign the theTitle and theYear */
EXEC SQL SELECT length INTO :theLength FROM Movie
WHERE title = :theTitle AND year = :theYear;
if (!strcmp(SQLSTATE, "00000"))
  printf("%d\n", theLength);
else
  printf("Not Found.\n");
```

Interface SQL / C : exemple de requête SQL avec une seule réponse – Syntaxe Oracle

```
% Find the length for a given title and year.
% Movie(title, year, length, inColor, sName, presC#)
EXEC SQL BEGIN DECLARE SECTION;
   char theTitle[20];      int theYear;
   int theLength;    char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;
... /* assign the theTitle and theYear */
EXEC SQL WHENEVER NOT FOUND GOTO notfound;
EXEC SQL SELECT length INTO :theLength FROM Movie
WHERE title = :theTitle AND year = :theYear;
   printf("%d\n", theLength);
   return;
notfound:
   printf("Not Found.\n");
```

Interface SQL / C : exemple d'inclusion de SQL

- Insertion d'un nouveau tuple : les variables partagées sont utilisées comme des valeurs

```
Void getStudio () {
   EXEC SQL BEGIN DECLARE SECTION;
     char studioName[15], studioAddr[60];
     char SQLSTATE[6];
   EXEC SQL END DECLARE SECTION;
   …/* Instanciation studioName et studioAddr
   EXEC SQL INSERT INTO Studio(name, address)
     VALUES (:studioName, :studioAddr);
```

Requêtes à réponses multiples

- Les curseurs permettent d'accéder successivement aux différents tuples d'une relation

1. Déclaration d'un curseur

   EXEC SQL DECLARE <cursor_name> CURSOR FOR <query>

2. Initialisation du curseur

   EXEC SQL OPEN <cursor_name>

   % Le curseur est prêt pour l'accès à la première valeur de la relation

3. Accès à un tuple

   EXEC SQL FETCH FROM <cursor_name> INTO <Shared variables>

   % Le prochain tuple est stocké dans les variables partagées; s'il n'existe pas

   SQLSTATE ←'02000'

4. Fermeture du curseur

   EXEC SQL CLOSE <cursor_name>

   % Le curseur peut être réouvert avec OPEN

Option des curseurs

Il est possible:

1. Modifier l'ordre dans lequel les tuples sont récupérés en utilisant la clause **`ORDER BY`** après le **`SELECT`** qui définit le curseur
2. Limiter les effets des modifications de la relation sur lequel le curseur est défini

   ```
   EXEC SQL DECLARE <cursor_name> INSENSITIVE
      CURSOR FOR <query>
   ```

   % Le curseur peut aussi être déclaré en READ ONLY
3. Modifier le mode de déplacement sur la séquence de tuples associée à une relation

   ```
   (NEXT, PRIOR, RELATIVE n,…)
   ```

Exemple d'utilisation de curseur

```
#define NO MORE TUPLES !(strcmp(SQLSTATE, “02000”);
void changeWorth()                                {
EXEC SQL BEGIN DECLARE SECTION;
int worth;  char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;
EXEC SQL DECLARE execCur CURSOR FOR
   SELECT netWorth FROM MovieExec;
EXEC SQL OPEN execCur;
while(1)                        {
EXEC SQL FETCH FROM execCur INTO :worth;
if (NO MORE TUPLES) break;
if (worth < 1000)
EXEC SQL DELETE FROM MovieExec WHERE CURRENT OF execCur;
else
EXEC SQL UPDATE MovieExec SET netWorth = 2 * netWorth WHERE
   CURRENT OF execCur;          }
EXEC SQL CLOSE execCursor;                        }
```

Exemple d'utilisation de curseur (Oracle)

```
EXEC SQL BEGIN DECLARE SECTION;
   char theTitle[20];
   int theYear;
EXEC SQL END DECLARE SECTION;
EXEC SQL DECLARE c CURSOR FOR
SELECT title, year FROM Movie
   WHERE studioName = 'Disney';
EXEC SQL OPEN CURSOR c;
while (1) {
EXEC SQL FETCH c INTO :theTitle, :theYear;
if (NOT FOUND) break;
/* format and print title and year */
}
EXEC SQL CLOSE CURSOR c;
```

Curseurs "READ ONLY"

```
EXEC SQL DECLARE movieStarCursor CURSOR FOR
SELECT title, year, studio, starName
    FROM Movie, StarsIn
    WHERE       title = movieTitle
          AND year = movieYear
FOR READ ONLY;
```

Toute tentative d'exécuter une clause UPDATE ou DELETE à travers le curseur `movieStarCursor` va générer une erreur

Interface SQL / langage hôte : PREPARE et EXECUTE

Deux instructions spéciales pour l'incorporation de code SQL:

- PREPARE transforme une chaîne de caractères en requête SQL
- EXECUTE execute cette requête

```
EXEC SQL PREPARE q FROM :query;
EXEC SQL EXECUTE q;
```

- Une requête "préparée" peut être exécutée plusieurs fois
- PREPARE et EXECUTE peuvent être combinées :

```
EXEC SQL EXECUTE IMMEDIATE :query;
```

12 JDBC

- JDBC (Java Database Connectivity) est une API (Application Programming Interface) fournie avec Java

- L'API JDBC est indépendante du SGBD : elle permet à un programme de se connecter à n'importe quelle base de données relationnelle conforme à ANSI SQL2 en utilisant la même syntaxe

  → **Autorise l'insertion d'instructions SQL dans un programme Java**

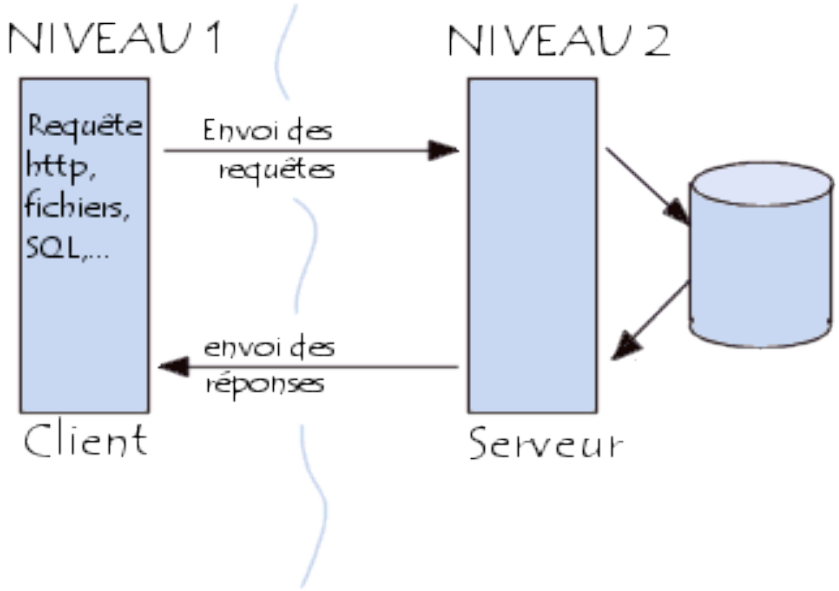
- L'API JDBC assure la portabilité du code

Architecture Client-serveur avec JDBC

- Modèle 2 niveaux :
  - Client ‘’lourd’’: en charge des applications et de l’IHM
  - Serveur ‘’léger’’ : gestion des données

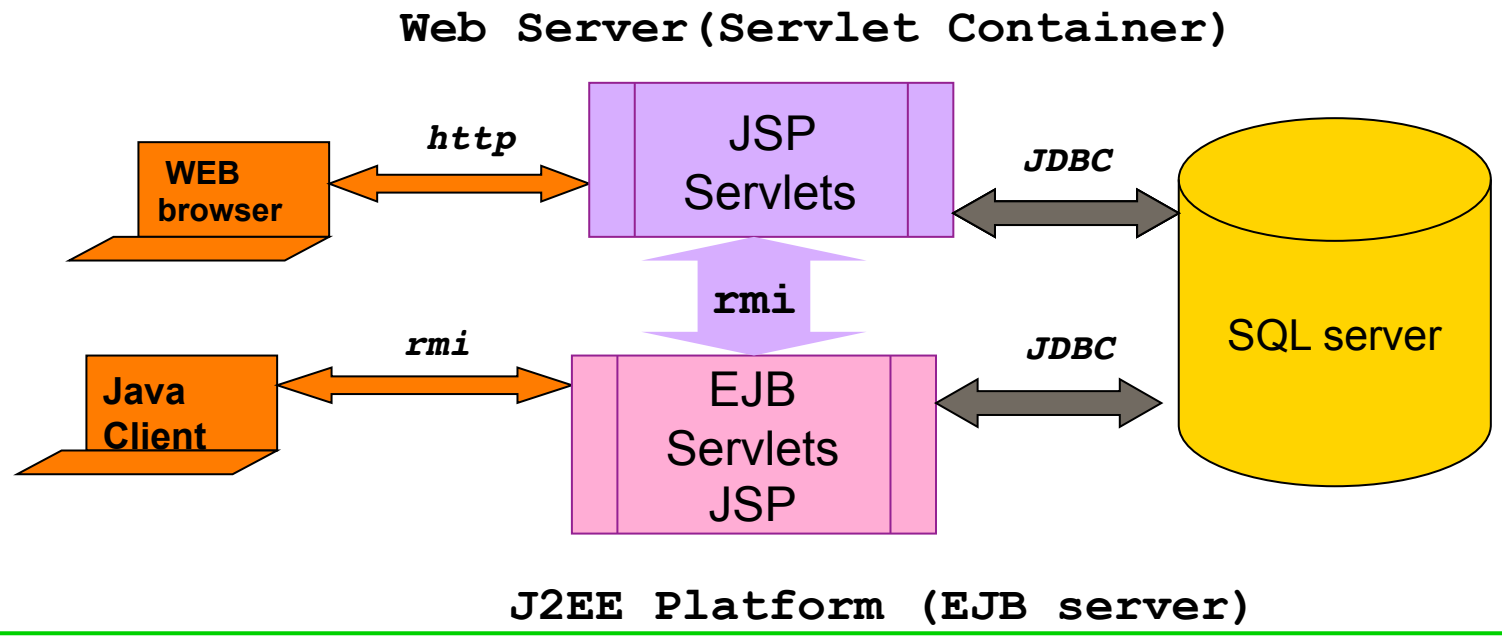
\+ : indépendance totale du DBMS

\- : pas de factorisation de code



Architecture Client-serveur avec JDBC

- Modèle 3 niveaux:
  - Clients "légers": en charge uniquement de l'IHM
  - Serveurs "lourds" (EJB,servlets) : en charge de l'application
  - Serveur DBMS : gestion des données



Les pilotes (drivers) JDBC

Quatre catégories :

- **Type 1**: pilotes qui accèdent à une BD via une ***passerelle*** (ou un *pont)*; e.g.,pont JDBC-ODBC: JDBC convertit les appels de données Java en appels ODBC valides et les exécute avec ODBC
- **Type 2**: pilotes d'API ***natifs***: les appels JDBC sont convertis en méthodes natives C/C++ du serveur de bases de données (Oracle, postgres,…)
- **Type 3**: pilotes ***génériques*** convertissant les appels JDBC en un protocole indépendant du SGBD. Un serveur convertit ensuite ceux-ci dans le protocole SGBD requis (modèle 3 niveaux)
- **Type 4**: pilotes ***dédiés*** qui convertissent les appels JDBC en un protocole réseau exploité par le SGBD; via des sockets java, ces pilotes interagissent directement avec l'interface cliente du SGBD (solution pour un cadre intranet)

ODBC(*Open Database Connectivity)*

- **Format propriétaire de Microsoft** pour la communication entre des clients et les SGBD du marché (pour bases de données fonctionnant sous Windows)
- **Résultats identiques** quelque soit le type de base de données, sans avoir à modifier le programme d'appel qui transmet la requête.

Connexion à une base de données avec JDBC

L'API (*Application Programming Interface*) JDBC, c'est-à-dire la bibliothèque de classes JDBC, se charge de trois étapes indispensables à la connexion à une base de données:

- la **création** d'une connexion à la base
- **l'envoi** d'instructions SQL
- **l'exploitation** des résultats provenant de la base

Package java.sql.*

- **Classes:** **Date, DriverManager**, DriverPropertyInfo
  Time , Timestamp ,Types

- **Interfaces** : Array , Blob , CallableStatement
  Clob , **Connection** ,DatabaseMetaData ,Driver
  PreparedStatement , Ref , **ResultSet** , **ResultSetMetaData**,
  SQLData , SQLInput , SQLOutput , **Statement** , Struct

- **Exceptions** : BatchUpdateException
  DataTruncation , SQLException , SQLWarning

Etablissement de la connexion avec JDBC

Chargement du pilote de la base de données à laquelle on désire se connecter grâce à un appel au DriverManager (gestionnaire de pilotes)

La classe **java.sql.DriverManager** permet d'établir la connexion.

Exemple :

jdbc:postgresql://dbms.essi.fr:4333/prof lambda mdp

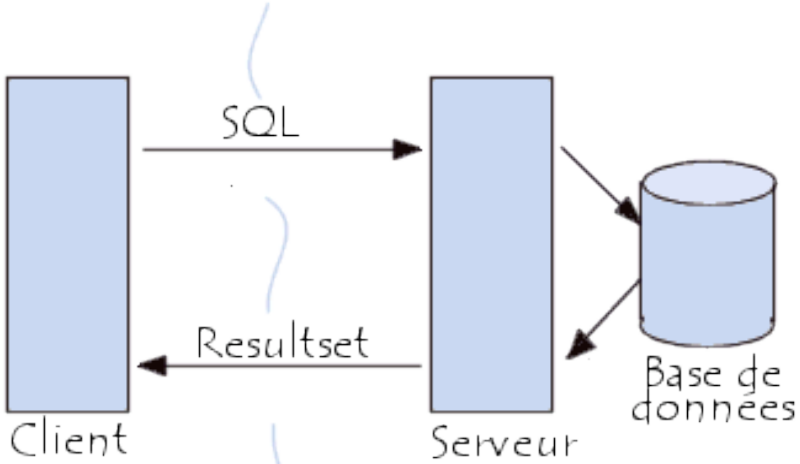
permet à l'utilisateur **lambda** de se connecter à la base prof sur le serveur postgres associé au port **4333** de la machine **dbms.essi.fr**

Connexion à une BD avec JDBC : exemple

```
import="java.sql.*; java.math.*;
public class  ….
String sqlUser = "invited";
String sqlPwd =  "invited";
String url = "jdbc:postgresql://dbms.essi.fr:5432/mabase";
Connection conn; String errMsg;
try {Class.forName("org.postgresql.Driver"); } // Vérification de la présence du Driver
catch(java.lang.ClassNotFoundException e) {
            System.err.print("C … }
conn = DriverManager.getConnection(url, sqlUser, sqlPwd); // Connexion
stmt = conn.createStatement();   // Création d'une requête SQL
stmt.executeUpdate("DELETE FROM affichage");
 ...
catch(SQLException ex) {
            System.err.println("SQLException: " + ex.getMessage());
            …
```

L'accès à une base de données avec JDBC

- La classe **java.sql.Statement** fournit les méthodes pour exécuter une requête :
  - **executeQuery() :** consultation
  - **executeUpdate() :** mise à jour
  - **execute() :** requête non définie



- La classe **java.sql.ResultSet** fournit les objets pour obtenir des informations sur la base
  - **DataBaseMetaData :** méta-données
  - **Resultset :** table
  - **ResultsetMetadata:** information sur le nom et le type des données de la table

L'accès aux résultats d'une requête JDBC

Les principales méthodes de l'objet ***ResultSet*** sont les suivantes:

- **getInt(int):** récupère sous forme d'entier le contenu d'une colonne désignée par son numéro
- **getFloat(int):** récupère sous forme de flottant le contenu d'une colonne désignée par son numéro
- **getString(int):** récupère sous forme de chaîne le contenu d'une colonne désignée par son nom
- **next():** déplace le curseur de colonne sur la colonne suivante
- **close():** ferme l'objet
- **getMetaData():** retourne les méta-données de l'objet

L'accès aux résultats d'une requête JDBC

L'objet ***ResultSetMetaData*** permet de connaître le nombre, le nom et le type de chaque colonne à l'aide des méthodes suivantes:

- **getColumnCount():** récupère le nombre de colonnes
- **getColumnName(int):** récupère le nom de la colonne spécifiée
- **getColumnType(int):** récupère le type de données de la colonne spécifiée

Exécution d'une requête avec JDBC : exemple

```
Connection conn;
...
// Requete pour recuperer afficher la liste des Cursus
      int i = 0;
      string  C[10];
      errMsg="PB dans select ... from CURSUS ...";
Statement  stmtCursus = conn.createStatement();
ResultSet  rsetCursus =stmtCursus.executeQuery(
         "SELECT C.nomcursus FROM cursus C");
      while (rsetCursus.next())
      {i++; C[i]=rsetCursus.getString(1)}
      rsetCursus.close();
```

Les requêtes pré-formatées

- L'interface **PreparedStatement** fournit la possibilité de lier des paramètres à un appel SQL avant son exécution en associant une valeur à l'indicateur de position (le caractère ?) dans l'instruction prédéfinie

```
PreparedStatement instruction = connection.preparedStatement(
   "UPDATE comptes" + "SET solde = ?" +"WHERE id = ?");
int i;
for(i=0; i<comptes.length;i++) {
   instruction.setFloat(1,comptes[i].extraitSolde());
   instruction.setInt(2,comptes[i].extraitIdf());
   instruction.execute();
}
connection.commit();
instruction.close();
```

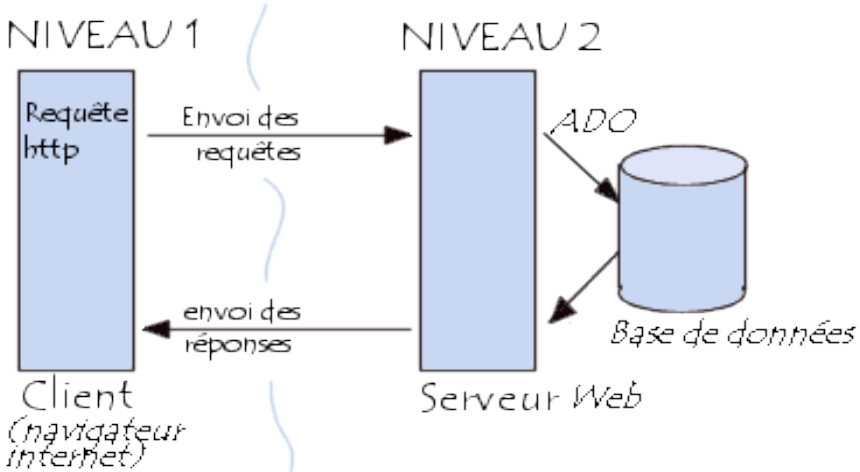
Évite la coûteuse création d'un plan de requête identique à chaque itération

JSP (Java Server Pages)

- Extension de HTML/XML
- Permet d'appeler **Java** et **Javascript** depuis HTML
- Utilise la technologie des **servlets** Java
- Permet la transformation pages HTML statiques en pages HTML dynamiques
- Permet de définir ses propres tags

JSP (suite)

- **JSP** est un langage de script puissant (un langage interprété) exécuté du côté du **serveur** et non du côté client (les JSP s'inscrivent dans une architecture 3-tiers)



- Les JSP sont intégrables au sein d'une page Web en **HTML** à l'aide de balises spéciales permettant au **serveur Web** de savoir que le code compris à l'intérieur de ces balises doit être interprété afin de renvoyer du code HTML au navigateur du client.

Exemple d'utilisation de HTML dans une page JSP

```
<form method=post  action="edt_result.jsp" name="form_sel">
 <p>Cursus <select name="cursus" size=1>
<%
// Requete pour récupérer et afficher la liste des Cursus
…
while (rsetCursus.next()) %>
<option value="<%=rsetCursus.getString(1)%>"
        <%=rsetCursus.getString(1)%>
</option>
<% }  // End of While
      rsetCursus.close(); %>
```

Moteur JSP

- Une servlet
- Traduit tout **xxx.jsp** en **_xxx.java** Servlet
- Compile tout **_xxx.java** Servlet en une **_xxx.class**

JSP basique : commentaires

Commentaires HTML

Envoyés au client dans la page HTML (uniquement visible dans la fenêtre source du navigateur)

Syntaxe JSP

<!-- *comment* [<%= *JavaExpression* %>] -->

Exemple

```
<% page language="java" %>
<HTML>
...
<H2> Essais de commentaires No 1 </H2>
  <!–- Author: P.Durand, Date :
    <%= (new java.util.Date()).toLocaleString() %>
  -->
  .........
</BODY>
```

JSP basique : déclarations

Java declarations: commentaires pour JSP (Servlet class attributes) non envoyés au client

JSP Syntax

<%! *déclarations* %>

Exemple 1

```
<% page language="java" %>
<%! Int n=3; %>
<%! String user; %>
<HTML>
 ...
  <H2> Essais de commentaires No 1 </H2>
   <% user= "richard" %>
  .........
  Alors que <%= user %> était pour la <%=n%>
   ème fois .....
```

JSP basique : déclarations (suite)

Java declarations

Résultat HTML (renvoyé par le serveur) de l'exemple 1

<HTML>

…

<H2> Essais de commentaires No 1 </H2>

……..

Alors que richard était pour la 3ème fois ……

……

</BODY>

JSP basique : directives

Pour contrôler le traitement d'une page JSP entière

Syntaxe JSP

<%@ *directive* [*attribute*= *value*] %>

Exemple

<%@ include file= "path" %>

- Pour insérer une page dans la page JSP courante ***avant*** traitement
- path est une URL relative

JSP basique : page directives

Page Directives :

<%@ page language = "java" %>

pour définir le langage

<%@ page import = "java.sql" %>

pour importer une bibliothèque

<%@ page errorPage = "path" %>

Pour renvoyer une page en cas d'exception (e.g., une page JSP)

La variable **exception** permet un traitement **dynamique** des exceptions

<%@ page isErroPage= "true" %>

Pour générer une exception

JSP basique : Actions

Syntaxe JSP :

<%= expression %>

L'expression est évaluée et insérée à cet endroit du texte HTML

équivalent à **out.print(*expression*)**

Exemple

..

Ceci est la requête de

<%= request.getParameter("nom")%>

..

JSP basique : Actions *scriptlets*

Syntaxe JSP pour scriptlets:

<% javacode %>

- Le code est exécuté
- L'exécution de **out.print(*expression*)** provoque l'insertion de texte dans le flot de sortie (i.e., la page HTML produite)

Exemple

```
<%
String name =request.getParameter("hisname");
....
out.print(name)
%>
..
```

Exemple d'actions *scriptlets*

```
<%     // ---------------  Affichage de la table MARQUE
   Statement  stmt = conn.createStatement();
   ResultSet  rset = stmt.executeQuery(sqlQuery);
   while (rset.next())  {          // Dump the result
      out.println( "<TR> \n" );
      out.println( "<TD><div align=\"center\"><i> " +
   rset.getString(1) + "</i></div></TD> \n" );
      out.println( "<TD><div align=\"center\"><i> " +
   rset.getString(2) + "</i></div></TD> \n" );
      out.println( "<TD><div align=\"center\"><i> " +
   rset.getString(3) + "</i></div></TD> \n" );
      out.println( "</TR> \n" );}
   rset.close();
%>
```

JSP basique : Actions *forward*

JSP Syntax :

<jsp:forward page="*path*"/>

- Le contrôle est transféré à une autre page
- Le document en cours de génération (i.e., les données dans le buffer) sont perdues

Exemple d'actions *forward*

```
...
String errFwd;
try {

   conn=DriverManager.getConnection("jdbc:postgresql:
   //dbms.essi.fr/prof", sqlUser, sqlPwd );
   if ( conn == null )  {  //   Connexion impossible
       errMsg= "Connexion impossible";
       errFwd="edt_erreur.jsp?ERR_MSG="+errMsg
       +"&ERR_PAGE="+errPage;
    jsp:forward page='<%=errFwd%>'                }
```

JSP basique : objets prédéfinis

HttpServletRequest request :

Correspond à une requête. Permet le traitement d'un formulaire

HttpServletResponse response

Correspond à la réponse à un formulaire envoyée au client

HttpSession session

Objet persistent correspondant à chaque session gérée par des cookies.

ServletContext application

Objet persistent correspondant à l'application contenant les pages JSP

JspWriter out

Pour écrire dans le flux de sortie du client (dans la page HTML générée)

Exemple d'utilisation d'objets JSP prédéfinis

```
%%% Dans fichier marques.jsp
<form method="post" action="marques_result.jsp" name="FORM_SEL">
  <p>Selectionnez une valeur ….</p>
  <p> Nom
  <input type="text" name="NOM" size="30" maxlength="30">
 <p> Classe
  <select name="CLASSE" size="10"><option value="0" selected> …

%%% Dans fichier marques_result.jsp
<%      // Parametres du formulaire de selection
        String reqNom;      String reqClasse;
        int             reqClasseInt; String reqPays;
        // Récuperation des valeurs de champs pour la selection
          reqNom = request.getParameter("NOM");
          reqClasse = request.getParameter("CLASSE");
          …
```

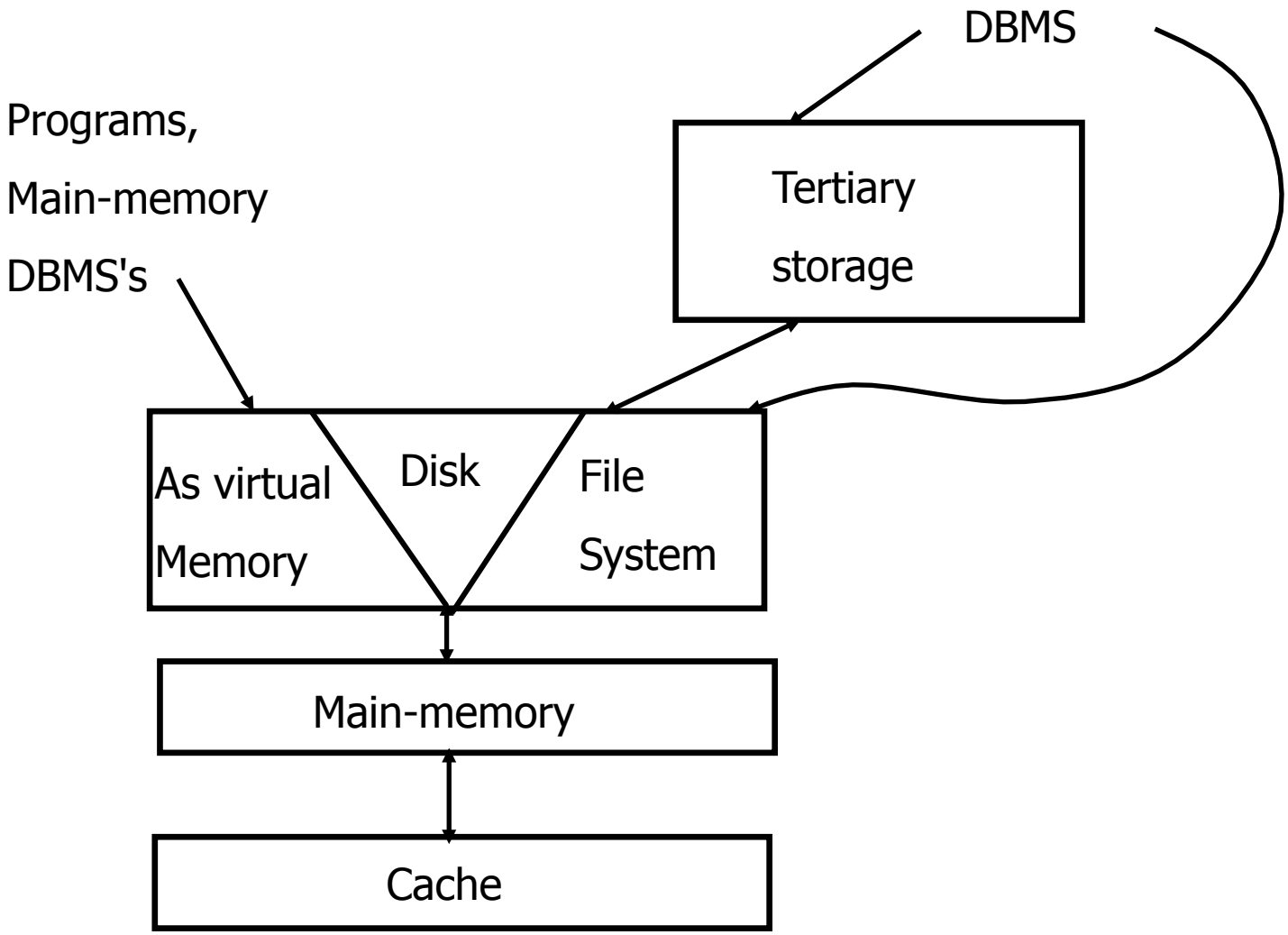
13 DBMS : éléments d'implémentation

- Architecture générale d'un DBMS
- Utilisation efficace de la mémoire secondaire
- Structures d'index

13 Architecture générale d'un DBMS

- **Gestion du stockage**
- Traitement des requêtes
- Traitement des transactions

Hiérarchie des supports "mémoires"



Supports "mémoire"

- Le temps d'accès à la mémoire et la vitesse des disques ne respectent pas la loi de "Murphy"
- *Plusieurs ordres de magnitude* entre les temps d'accès et les capacités des différents types de support
- La nature du support est un critère de choix pour les modes de représentation et les algorithmes

Cache

- Circuit intégré (parfois sur le même "chip" que le microprocesseur)
- Contenu : instructions machine et données (copie d'une partie de la mémoire principale)
- Capacité réduite : giga byte
- Accès rapide:
  - Lecture et écriture entre processeur et cache: moins de 10 nano-secondes (10^{-8} secondes)
  - Temps de transfert entre mémoire principale et cache : 100 nano-secondes

Mémoire principale

- Contenu: instructions, données à modifier
- Capacité : quelques giga byte
- Accès :
  - Aléatoire (temps constant)
  - Temps d'accès : 10 → 100 nano-secondes

Mémoire secondaire (disque)

- Grande capacité mais temps d'accès plus lents : 10^5 fois plus lent (10-30 millisecondes pour lire ou écrire un block)
  - → 100 fois plus de capacités
  - → 1000 fois plus lent

- Mémoire virtuelle:
  - Extension de la mémoire principale (4 giga bytes adressable avec 32 bits)
  - Organisée par blocks (pages): 4 → 56 K bytes
  - Peu utilisée par les grands DBMS

Mémoires tertiaires

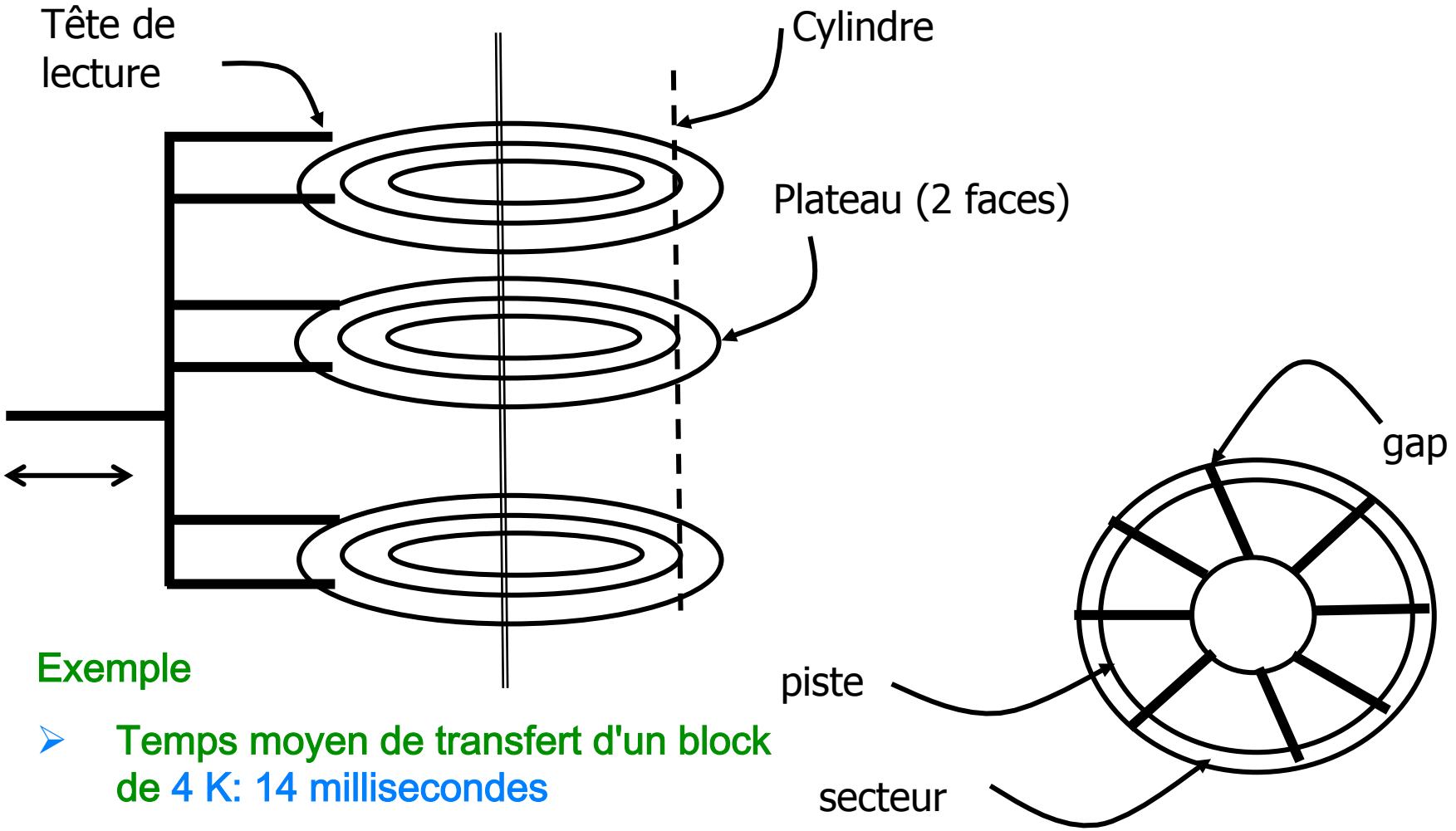
- Cassettes magnétiques, disque optiques, "Tape silos", "juke box" (rack de CD-ROM ou DVD-ROM)

  → 1000 à 10000 fois plus de capacités

  → 1000 à 10000 fois plus lent

  Remarque : *les mémoires secondaires et tertiaires sont rémanentes alors que la mémoire principale est volatile*

Organisation des disques



Exemple

- Temps moyen de transfert d'un block de 4 K: 14 millisecondes
- Temps moyen de transfert d'un block de 64 K: 22 millisecondes

13.2 Utilisation efficace de la mémoire secondaire

- Meilleurs algorithmes en mémoire principale ne sont pas nécessairement les meilleurs algorithmes en mémoire secondaire
- Nombre d'accès à des blocks disque est une bonne approximation du temps d'un algorithme

  → Objectif : ***minimiser les accès à des blocks sur disque***

Exemple du tri-fusion

- **Algorithme**:

  Fusion deux par deux des listes triées en répétant le processus suivant jusqu'à ce que une des listes devient vide:
  - Recherche de la plus petite des clés en tête des listes
  - Retrait de cette clé et écriture sur la sortie standard

- **Exemple**:

Étape	Liste 1	Liste 2	Sortie
start	1,3,4,6	2,5,7,8	rien
1	3,4,6	2,5,7,8	1
2	3,4,6	5,7,8	1,2
3	4,6	5,7,8	1,2,3
4	6	5,7,8	1,2,3,4
5	6	7,8	1,2,3,4,5
6	rien	7,8	1,2,3,4,5,6
7	rien	rien	1,2,3,4,5,6, 7,8

Exemple du tri-fusion (suite)

- **Induction**:

  Toute liste de plus de deux éléments est décomposée en deux listes de "même" taille et l'algorithme de fusion est appliquée récursivement

- **Complexité**: **O(nlogn)**

Tri-fusion en mémoire secondaire

- **Étape 1** :

  Tri en mémoire principale (avec un tri rapide) des différentes parties de la liste

  → génération de *n* sous liste triées

- **Étape 2** : fusion de l'ensemble des sous-listes simultanément
  - Chargement en mémoire du premier block de chaque sous liste
  - Retrait et écriture sur une partition du plus petit des éléments en tête des listes en mémoire

  Partition de sortie pleine → écriture sur disque

  Sous liste vide → chargement du bloc suivant

Tri-fusion en mémoire secondaire (suite)

- **Coût** :

  Chaque bloc est chargé exactement **deux fois** en mémoire principale (**$\log_2 n$** fois avec l'algorithme classique)

- **Exemple** :
  - 10 000 000 de tuples (enregistrements de 100 bytes par tuples)
  - La taille des blocs est de 4 096 bytes (40 tuples de 100 bytes = 1 bloc)
  - Temps de lecture/ écriture d'un block: 15 millisecondes
  - 12500 blocs peuvent être chargés en mémoire

Technique de réduction du temps d'accès en mémoire secondaire

- Regrouper les blocks accédées simultanément sur le même cylindre → réduction du temps de recherche et de latence
- Répartition des données sur plusieurs petits disques → augmente le nombre de blocks qui peuvent être accédés simultanément
- Utilisation d'un algorithme d'ordonnancement des accès dans le DBMS ou le contrôleur du disque plutôt que l'OS
  → optimisation de l'ordre d'accès aux blocks
- Chargement de blocks de manière anticipées
- Utilisation de clusters et d'index

Les clusters

Un cluster est un regroupement dans un même bloc disque des lignes d'une ou plusieurs tables ayant une caractéristique commune (une même valeur dans une ou plusieurs colonnes) constituant la clé du cluster

Objectifs :

- accélérer la jointure selon la clé de cluster des tables concernées
- accélérer la sélection des lignes d'une table ayant même valeur de clé
- économiser de la place: chaque valeur de la clé du cluster ne sera stockée qu'une seule fois

Les clusters (suite)

- Le regroupement en cluster est totalement transparent à l'utilisateur.
- Pour que l'on puisse mettre une table en cluster il faut que l'une au moins de ses colonnes soit définie comme obligatoire (NOT NULL).
- On peut indexer les colonnes d'une table en cluster

13.3 Structures d'index

Dans le modèle relationnel les sélections peuvent être faites en utilisant le contenu de n'importe quelle colonne. Soit la requête:

```
SELECT * FROM emp WHERE nom = 'MARTIN'
```

Pour retrouver les lignes pour lesquelles nom est égal à MARTIN on peut balayer toute la table.

Un tel moyen d'accès conduit à des temps de réponse prohibitifs

→ **Utilisation d'index**

Index (principes)

- Un index sera matérialisé par la création de blocs disque contenant des couples (valeurs d'index, numéro de bloc) donnant le numéro de bloc disque dans lequel se trouvent les lignes correspondant à chaque valeur d'index

- L'adjonction d'un index à une table ralentit les mises à jour (insertion, suppression, modification de la clé) mais accélère beaucoup la recherche d'une ligne dans la table

- L'index accélère la recherche d'une ligne à partir d'une valeur donnée de clé, mais aussi la recherche des lignes ayant une valeur d'index supérieure ou inférieure à une valeur donnée, car les valeurs de clés sont triées dans l'index.

Index : exemples

- Les requêtes suivantes bénéficieront d'un index sur le champ **n_dept:**

```
SELECT * FROM emp WHERE num = 16034  ;
SELECT * FROM emp WHERE num >= 27234 ;
SELECT * FROM emp WHERE num BETWEEN 16034  AND 27234;
```

- Un index est utilisable même si le critère de recherche est constitué seulement du début de la clé : La requête suivante bénéficiera d'un index sur la colonne nom.

```
SELECT * FROM emp WHERE nom LIKE 'M%';
```

- Si le début de la clé n'est pas connu, l'index est inutilisable

  Exemple: `SELECT * FROM emp WHERE nom LIKE '%M%';`

Valeurs NULL et Conversions

- Les valeurs NULL ne sont pas représentées dans l'index

  → minimiser le volume nécessaire pour stocker l'index

- L'index n'est utilisable que si *le critère de sélection est le contenu de la colonne indexée*, sans aucune transformation.

  Exemple un index sur salaire ne sera pas utilisé pour la requête :

  ```
  SELECT * FROM emp WHERE salaire * 12 > 300000;
  ```

  Mais sera utilisé pour la requête :

  ```
  SELECT * FROM emp WHERE salaire > 300000/12;
  ```

- Attention aux conversions de type qui peuvent empêcher l'utilisation de l'index (e.g., chaîne en date,…)

  Exemple :

  ```
  SELECT * FROM emp WHERE embauche LIKE '…'
  ```

  sera évalué en … `WHERE TO_CHAR(embauche)` …

  Le critère de recherche est une fonction de embauche → l'index est inutilisable

Choix des index

- Indexer en priorité :
  1. les clés primaires
  2. les colonnes servant de critère de jointure
  3. les colonnes servant souvent de critère de recherche

- Ne pas indexer :
  1. les colonnes contenant peu de valeurs distinctes (index peu efficace)
  2. les colonnes fréquemment modifiées

Index comprimé et non comprimé

- La compression (option par défaut) permet de réduire dans des proportions très importantes le volume des index (un traitement supplémentaire est parfois requis pour recomposer la clé lors des mises à jour de l'index)

- SQL sait exécuter certaines requêtes directement au niveau de l'index sans passer par le segment de données, si l'index est non comprimé. L'index crée par :

```
CREATE INDEX x ON emp (num, nom)
                              nocompress ;
```

permettra de répondre, sans lire la table, à la question :

```
SELECT nom FROM emp WHERE num > 17217 ;
```

Index comprimé et non comprimé

- Un index concaténé est un index portant sur plusieurs colonnes.

  Exemple :

```
CREATE INDEX x ON emp (n_dept,num) ;
```

- SQL sait utiliser un index concaténé même si le critère de recherche ne porte pas sur toutes les colonnes présentes dans l'index. L'index ci-dessus est utilisable si l'on ne connaît que le numéro de département.

```
SELECT nom FROM emp WHERE n_dept = 20 ;
```

Différents niveaux de stockage des données

- Attributs: champs (colonnes)
- Ensemble d'attributs : record
  - Taille fixe
  - Taille variable (**`varchar`**,…)
- Ensemble de records : bloc
  - "Header" des records:
    - Pointeur vers le schéma
    - Longueur des records
    - "Timestamps": dernière modification ou lecture,
    - …
  - "Header" des blocs:
    - Pointeur vers d'autres blocs (même indexation)
    - Bloc Id
    - "Timestamps": dernière modification ou lecture
    - …

Types d'index

- **Index simples** (sur des fichiers triés suivant la clé de recherche)
- **Index secondaires** (sur des fichiers non triés suivant la clé de recherche)
- **B-tree**
- **Table de Hash**

Index simples
(fichiers de données triés suivant la clé de recherche)

- **Fichier index** : ensemble de paires

  `<clé de recherche, adresse physique>`
- **Index denses**: fichier index contient une clé de recherche pour chaque valeur
- **Index peu denses**: fichier index contient une clé de recherche pour certaines valeurs (e.g., première clé de chaque block)
- **Critère de choix de la "densité"** : possibilité de stocker la table d'index en mémoire
- Plusieurs niveaux de tables d'index sont possibles

Index simples (suite)

- **Doublons dans les clés d'index**
  (index peut être défini sur des attributs non uniques)
  → une entrée dans le fichier d'index par valeurs
  (doublons se suivent dans le fichier)

- **Mise à jour** des fichiers de données
  → Prévoir des blocs de débordement
  → Utilisation des "headers" des blocs

Index secondaires
(fichiers de données *non* triés suivant la clé de recherche)

- **Fichier index** : contient l'adresse courante
- **Index peu denses**: pas d'intérêt
- **Un niveau d'indirection ("buckets")** est souvent utile pour éviter la perte de place due aux doublons
  (zones de débordement dans les buckets)

B-trees (arbres équilibrés)

- **Génération automatique** : du nombre de niveaux d'indexation en fonction de la taille des fichiers
- **Gestion de la place physique** tel que tout bloc utilisé est au moins rempli à 50% (pas de bloc de débordement)

Hash coding

- **Fonction de "hachage"** ***h*** qui génère un entier entre 0 et B-1 si B est le nombre de buckets
- Record qui a pour clé de recherche ***K*** sera stocké à l'adresse contenu dans le bucket ***h(K)***
- Tables de taille importante stockées en mémoire secondaire