Bases de données relationnelles

Michel Rueher

Supports disponibles:

http://www.i3s.unice.fr/~rueher/Cours/BD/Cours/

1. Introduction

Plan

- BD … informellement
- Les grands principes
- 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

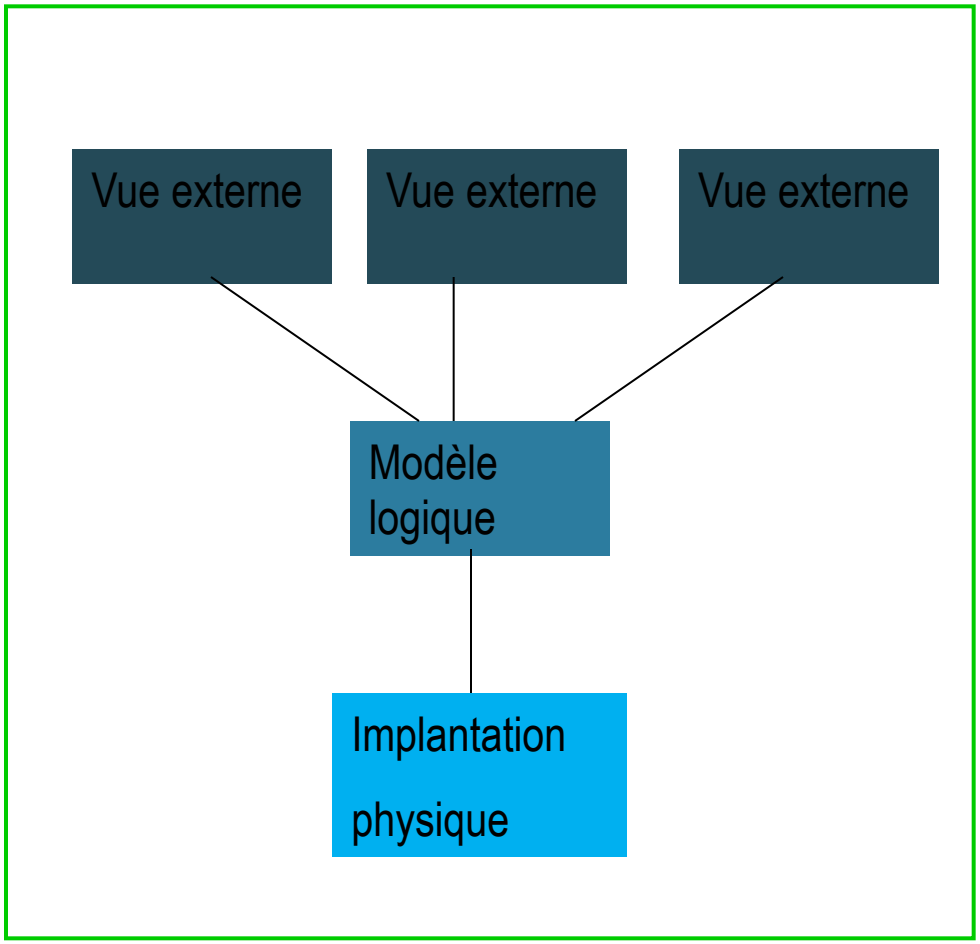
Modèle logique (modèle de données)

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

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

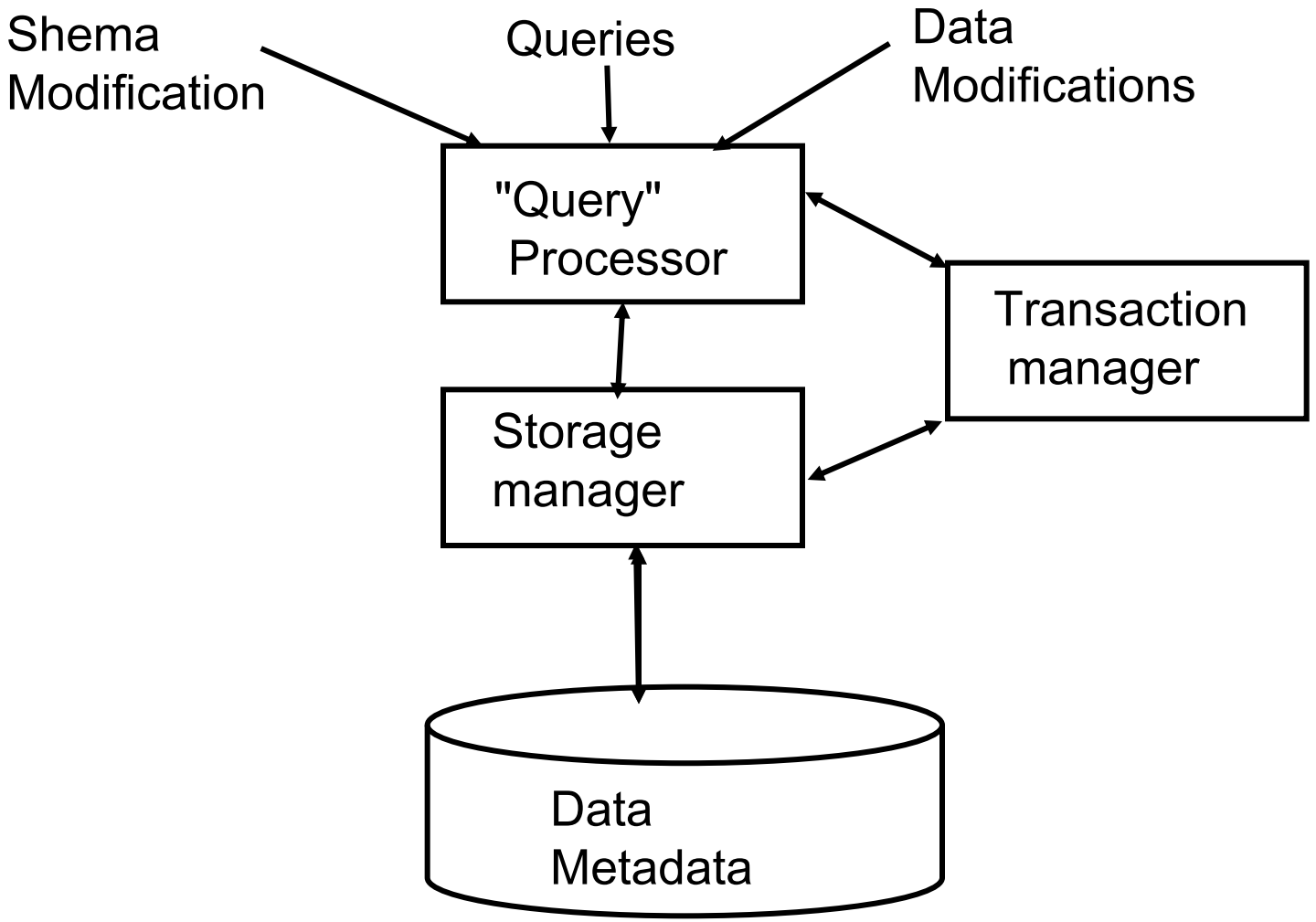


Les grands principes(suite)

Fonctionnalités

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

Les grands principes(suite): architecture d'un DBMS



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
  - Langages formels (logique, algébrique)
  - Langages ad-hoc (textuels, graphiques,...)
  - dépendances fonctionnelles: clés, références entre tables,...

- Le standard ANSI/SO depuis 1992

Le Modèle relationnel : principes

- **Schéma** de base de données = ensemble de **relations**
  - –> liens sémantiques implicites
  - Instances de relations: **tuples** (tables)
  - **contraintes** sur les relations et les tuples,

- **Langages déclaratifs** du premier ordre (LDD/LID de SQL)

- **Algèbre Relationnelle** , calcul sur les tuples

Eléments principaux de SQL (Version Ansi2)

- **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** : vues
  - 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, ...)
    - 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 est appelée relation..
Le *contenu*, à un instant donné, de cette table, est une *table instance* de cette relation

Exemple:

- **Relation:** marque(IdM, NomM, Classe,IdProp)
- **table** *marque*

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* et 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.

Schémas relationnels (suite)

Relation

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

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

- $A = attr(R) = \{A_1, ..., A_n \}$ *: l'ensemble des* ***attributs*** *de* ***R***
- $dom_i = dom(A_i)$ *: le* ***domaine*** *non vide de chacun des attributs* A_i

Domaine

- *type du contenu des colonnes de la table: **contraintes** sur le contenu de chacun des tuples d'une instance de la relation*
- *toujours de type **scalaire** (entiers, chaînes,...) : pas d'opérateur pour leur associer des types structurés*

Schémas relationnels (suite)

Exemple: la relation MARQUE

Attributs : **{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} (chaînes limitées à 40 caractères)
- dom(Classe) = [1..30]
- dom(IdProp) = ensemble de tous les mots de moins de 100 caractères construits à sur l'alphabet {A,...,Z, a,…,z,0.. 9}

Instances de relations

Tuple

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

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

  Un tuple défini sur **R** est un ensemble ***t*** de valeurs $v_1,....,v_n$ associées aux attributs A_1 ...., A_n, tel que $v_i \in dom(A_i)$

- Notation
  - ensembliste: $t = \{v_1: A_1, ..., v_n:A_n\}$
  - notation parenthèsée : $t = (v_1: A_1; ....; v_n:A_n)$
  - valeur v_i associée à l'attribut A_i du tuple **t** : $v_i = t.A_i$

Instances de relations (suite)

Le tuple défini sur la relation MARQUE

t = {	122 233	:	IdMarq,
	COCA	:	NomMarq,
	12	:	Classe,
	CocaLtd	:	IdProp }

peut se représenter par la ligne :

IdMarq	NomMarq	Classe	IdProp
122 233	COCA	12	CocaLtd

On a alors t.Classe = 12

Table instance d'une relation

Table instance de la relation R: tout ensemble de tuples (lignes) définies sur R.

Instance de la relation MARQUE= {t1, t2,t3}

t1 = {1222 :IdM,
COCA :NomM,
12 :Classe,
CocaLtd :IdProp }
t2= {1224 :IdM,
ORANGINA :NomM,
12 :Classe,
Perrier :IdProp }
t3 = {1226 :IdM,
PEPSI :NomM,
12 :Classe,
PepsiLtd :IdProp }

Contraintes

- Une relation sur l'ensemble des tuples présents 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, ...)

- 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

Langages de contraintes

Enjeux des BDR :

- disposer de **langages déclaratifs** aussi riches que possible
- disposer d'un **vérificateur**

Si les contraintes ne pas peuvent s'exprimer en logique du premier ordre

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

Contraintes et langages (suite)

Contraintes sur table *Marque :*

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

  $\mathbf{C_1 :} \quad \forall t_1, t_2 \in \mathbf{marque}$

  $\mathbf{((t_1.NomM = t_2.NomM \ \wedge t_1.Classe = t_2.Classe)}$

  $\mathbf{\Rightarrow \ t_1.IdM = t_2.IdM)}$

- (C_2) Un même propriétaire ne peut être associé à plus de 20 marques d'identificateurs distincts (**pas du premier ordre:** quantification sur les domaines infinis et fonction *card* de type ensembliste)

  Peut aisément être vérifiée sur une instance particulière:

  $C_2 \equiv \quad \forall p \in dom(IdProp \ )$

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

Schéma de bases de données

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 (ici 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)

- "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

3. Algèbre relationnelle

Algèbre relationnelle

- **Langage d'expressions algébriques en notation fonctionnelle**:
  - Les variables représentent des tables, instances de relations
  - Opérateurs (unaires, binaires, ...) sur ces tables

Opérations booléennes : Union, intersection, différence

Soient ***r*** *et* ***s*** *deux deux tables du schéma R (c.a.d .avec les mêmes attributs)*

Les opérations booléennes d'union, d'intersection et de différence définissent des instances du même schéma **R**:

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

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

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

Rem. : l'intersection 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

Projection : définition

Opération unaire

→ copie une relation en ne gardant que certaines colonnes

Définition

*Soient **R** un schéma, **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 relation r est une instance du schéma **A** définie par :*

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

Exemple de projection

marque

Id	*Nom*	*Classe*	*Prop*
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

Les lignes redondantes disparaissent :

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

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

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

Projection : propriétés

Si *r et s sont deux instances de* ***R****, et* ***A*** *un sous-ensemble d'attributs de* ***R****:*

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

On n'a pas toujours :

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

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

Sélection : définition

Filtrage de valeur d'attribut

→ *extraire par copie certaines lignes* d'une relation.

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

Distributivité sur les opérations booléennes

Pour $o \in \{ \cup, \cap, - \}$ 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 $\mathbf{R}$, $\mathbf{A} \subseteq \mathbf{R}$ et $a \in \mathrm{dom}(A)$:

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

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

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

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 :

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

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

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 si aucun nom d'attribut partagé*

Notation

*Si **R** et **S** sont deux ensembles d'attributs, **RS désigne** l'ensemble d'attributs égal à l'union de R et de S.*

$$RS = R \cup S .$$

Jointure Naturelle de tuples (suite)

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 $(t_r \rhd\lhd t_s)$

$(t_r \rhd\lhd t_s)$ existe $\Leftrightarrow$ $\pi_{R\cap S}(t_r) = \pi_{R\cap S}(t_s)$

$(t_r \rhd\lhd t_s)$ existe $\Rightarrow$ $\pi_R(t_r \rhd\lhd t_s) = t_r$

$\Rightarrow$ $\pi_S(t_r \rhd\lhd t_s) = t_s$

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 :

(NomProd:'JointNéoprène';Code:302)⋈(Distrib:'StéX';NomProd:'Rond6x4)

n'existe pas !

Jointure naturelle de deux tables

Définition :

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 définie par :

$$r \rhd\lhd s = \{\, t(RS) \mid \pi_R(t) \in r \;\wedge\; \pi_S(t) \in s \,\}$$

On a alors :

(1) $t \in r \rhd\lhd s \;\Leftrightarrow\; \exists t_r \in r \;\; \exists t_s \in s$ **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) $t \in r \rhd\lhd s \;\Rightarrow$ 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

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

si $R \cap S = \varnothing$ alors $r \bowtie s$ correspond au **produit cartésien** des deux instances r et s noté $r \times s$

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
```

Complexité fonction du nombre d'accès à des blocs sur disque nécessaires pour lire chacune de deux relations ***r*** et ***s***, et de *du nombre de blocs pouvant résider en mémoire simultanément*

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**

Exemple: Quels sont les athlètes qui ont participé à toutes les épreuves ?

R

Athlète	Epreuve
Pierre	*200 m*
Pierre	*400 m*
Pierre	*800 m*
Paul	*400 m*
Jean	*200 m*

S

Epreuve
200 m
400 m
800 m

R ÷ S

Athlète
Pierre

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

$r \div s$ est le plus grand ensemble q de $\pi_Q(r)$ tel que $(q \bowtie s) \subseteq r$

Division (propriété)

La division s'exprime en fonction des opérateurs précédents :

$$r \div s = \pi_Q(r) - \pi_Q((\pi_Q(r) \bowtie s) - r)$$

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

»La société qui possède toutes les marques de la classe 14"

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

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

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

Renommage des attributs

Définition

Le renommage de A en B dans une instance schéma R est une instance du schéma R'= R-{A} ∪ {B} définie par :

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

On peut étendre ce renommage à plusieurs attributs:

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

Il est supposé s'effectuer de facon simultanée

Renommage des attributs : exemple

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

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

Pour calculer les *"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 langage SQL

4.1 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;
```

ne supprime pas les redondances

Nom
Microsoft Pomme verte ….

Renommage :

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

NomMarque	Proprio
Microsoft ….	Claude Gate …

Selections: la clause SELECT

La clause SELECT permet d'effectuer des sélections

```
SELECT * FROM marque
          where classe=14;
```

Nom	Classe	Pays	Idprop	Id
Loreal	14	Bahamas	B0007	14032
Hermes	14	France	H0414	14923

Jointure: la clause SELECT

La clause SELECT permet d'effectuer des jointures

```
SELECT *  FROM marque M, societe S
      where M.prop = S.id;
```

id	nom	classe	pays	prop/id	nom	ville	pays
14032	Hermes	14	France	H0414	Durand	Paris	France
04114	Cola	23	USA	X0317	McAch	ElPaso	USA
…							

Projection, Sélection Jointure: la clause SELECT

La clause SELECT permet d'effectuer des jointures

```
SELECT M.Nom  FROM marque M, societe S
      where     classe =14 AND
                M.prop = S.id;
```

nom
Hermes

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;
```

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 Prop = 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 M, enr  E
         WHERE E.Id=M.ID;
```

Sémantique d'une requête portant sur plusieurs relations

Modèle de calcul:

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

(même interprétation dans DATALOG et 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 $R \cap S$ alors que *la requête SQL produit l'ensemble vide*

Opération ensemblistes: Intersection, Union et Différence

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

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

Nom et classe des marques déclarées dans la classe 14 ou 10

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

Nom et classe des marques n'appartenant pas à la classe 10

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

! Suppriment les redondances !

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 toutes les valeurs 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

Tuples:

```
film(titre,année,durée,studio,producteur)
acteur(titreFilm, annéeFilm, nom_A)
socProduct(nom,adresse,num_enreg)
```

La requête : recherche le producteur des films de Ford

```
SELECT nom FROM socProduct
  WHERE num_enreg IN
    (SELECT producteur FROM film
        WHERE (titre,année) IN
            (SELECT titreFilm,annéeFilm FROM acteur
                WHERE nom_A= 'Ford' ));
```

Conditions sur des tuples (suite)

La requête

```
SELECT titreFilm,annéeFilm FROM acteur WHERE nom_A=Ford'
```

produit la table :

titrefilm	*annéefilm*
Star Wars	1977
Raider of	1993
..	

La requête

```
SELECT producteur FROM film WHERE (titre,année) IN (..)
```

génère un ensemble de numéros d'enregistrement:

producteur
P213
P013
..

Conditions sur des tuples (suite)

La requête ci-dessous produit le même résultat :

```
SELECT nomFROM socProduct, film, acteurs
   WHERE num_enreg = producteur
        AND titre = titreFilm
        AND année = annéeFilm AND acteur='Harrison Ford';
```

mais les doublons ne sont pas gérés de la même manière et …

…. elle est plus couteuse (produit cartésien)

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 titre from film AS Old
   WHERE annee < ANY
      (SELECT année FROM film
            WHERE titre = Old.titre);
```

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

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*

Pour éliminer les doublons:

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

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

4.2 Opérateurs & 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 colonnes)

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;
```

Opérateurs & 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 **agrégats** et les **attributs mentionnés** dans la clause !!

!! GROUP BY **peuvent apparaître dans la clause SELECT** !!

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)

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

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

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

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 néssairement être sélectionné !

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

→ OK (mais non-conforme standard "récents")

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

4.3 Jointures

La jointure conventionnelle (**theta** jointure) peut être effectuée soit :

- implicitement
- explicitement à l'aide d'une clauses spécifique:

`CROSS JOIN` → produit cartésien

`JOIN ON` → jointure sur les attributs spécifiés

CROSS JOIN

table t1

num	name
1	a
2	b
3	c

table t2

num	name
1	x
3	y
5	z

```
SELECT * FROM t1 CROSS JOIN t2;
```

```
SELECT * FROM t1 , t2;
```

num	name	num	name
1	a	1	x
1	a	3	y
1	a	5	z
2	b	1	x
2	b	3	y
2	b	5	z
3	c	1	x
3	c	3	y
3	c	5	z

(INNER) JOIN ON (theta jointure)

table t1

num	name
1	a
2	b
3	c

table t2

num	name
1	x
3	y
5	z

```
SELECT * FROM t1 JOIN t2 ON t1.num=t2.num;
SELECT * FROM t1 INNER JOIN t2 ON t1.num=t2.num;
SELECT * FROM t1, t2 WHERE t1.num=t2.num;
```

num	name	num	name
1	a	1	x
3	c	3	y

(INNER) JOIN: ON versus USING

table t1

num	name
1	a
2	b
3	c

table t2

num	name
1	x
3	y
5	z

num	name	num	name
1	a	1	x
3	c	3	y

SELECT * FROM t1 JOIN t2 ON t1.num=t2.num;

SELECT * FROM t1 JOIN t2 USING (num) ;

num	name	name
1	a	x
3	c	y

Jointure Naturelle

table t1

num	name
1	a
2	b
6	w

table t2

num	name
1	a
3	y
6	w

```
SELECT * FROM t1 NATURAL JOIN t2;
-- alias pour USING (num,name)
```

num	name
1	a
6	w

Jointure externe : LEFT (OUTER) JOIN

table t1

num	name
1	a
2	b
3	c

table t2

num	name
1	x
3	y
5	z

résultat

num	name	num	name
1	a	1	x
2	b	null	null
3	c	3	y

SELECT * FROM t1 LEFT JOIN t2 ON t1.num=t2.num;

- Toutes les valeurs de la première table sont conservées.
- Les valeurs différentes dans la 2ème table sont remplacées par null

Jointure externe : RIGHT (OUTER) JOIN ON

table t1

num	name
1	a
2	b
3	c

table t2

num	name
1	x
3	y
5	z

résultat

num	name	num	name
1	a	1	x
3	c	3	y
null	null	5	z

SELECT * FROM t1 RIGHT JOIN t2 ON t1.num=t2.num;

- Toutes les valeurs de la deuxième table sont conservées.
- Les valeurs différentes dans la 1ème table sont remplacées par null

Jointure externe : RIGHT (OUTER) JOIN USING

table t1

num	name
1	a
2	b
3	c

table t2

num	name
1	x
3	y
5	z

SELECT * FROM t1 RIGHT JOIN t2 USING(num);

seules les colonnes de la 2ème table sont conservées pour les attributs du "using"

résultat :

num	name	name
1	x	a
3	y	c
5	z	null

Jointure externe : FULL (OUTER) JOIN USING

table t1

num	name
1	a
2	b
3	c

table t2

num	name
1	x
3	y
5	z

SELECT * FROM t1 FULL JOIN t2 USING (num);

Les valeurs différentes dans une des tables sont remplacées par null

num	name	name
1	a	x
2	b	null
3	c	y
5	null	z

5. 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 INTO
- Le nom de la relation, R
- Une liste d'attributs de R (entre parenthèses)
- Le mot clé VALUES et une liste parenthèsée de valeurs
  ou une requête

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;
```

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'
```

5.2 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

5.3 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 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, studioname 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

5.4 Opérations avec NULL

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

- 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

5.5 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:

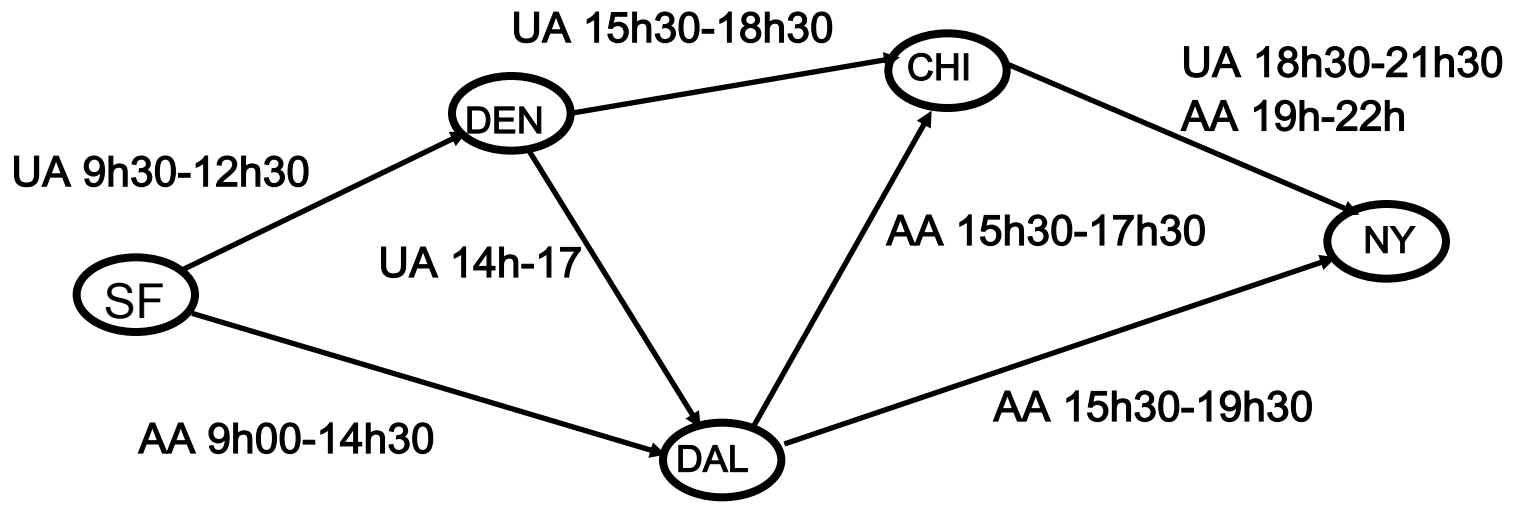
- Mot clé WITH
- Une ou plusieurs définitions séparées par des virgules; chaque définition comprend:
  - Le mot clé RECURSIVE s'il s'agit d'une définition récursive
  - Le nom de la relation à définir
  - Le mot clé AS
  - La requête qui définit la relation
- Une requête qui se réfère aux définitions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



Requête: liaisons entre SF et NY ?

L'instruction WITH (exemple)

Les pairs de villes connectées par des vols de ce graphe:

liaison(x,y) ← vol(_,x,y,_,_)

liaison(x,y) ← vol(_,x,z,_,_) AND liaison(z,y)

En SQL3

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

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

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)

Conditions:

- A_2 a été déclaré comme clé primaire ou unique 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
- Une clé primaire ne peut contenir des attributs avec la valeur NULL
- 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 products (
    product_no integer,
    name text,
    price numeric
    CHECK (price > 0),
    discounted_price numeric );
```

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

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

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**

  Attention : dans Postgres le check ne peut contenir une requête

- 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

```
CREATE TABLE movieStar (
        name CHAR(30),
        address VARCHAR(255),
        gender CHAR(1),
        birthdate DATE,
        CHECK (gender='F' OR name NOT LIKE 'Ms.%')        );
CREATE TABLE products (
    …
    price numeric            CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0 AND price > discounted_price) );
```

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);
...
```

6.5 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,
  - Si elle est **vraie** le trigger peut 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 TABLE emp ( empname text, salary integer, last_date timestamp,  last_user
    text );

CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
BEGIN   -- Check that empname is given
    IF NEW.empname IS NULL
        THEN RAISE EXCEPTION 'empname cannot be null';
    END IF;
        -- Remember who changed the payroll and when
    NEW.last_date := current_timestamp;
    NEW.last_user := current_user;
    RETURN NEW;
END; $emp_stamp$ LANGUAGE plpgsql;

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp FOR EACH
    ROW EXECUTE PROCEDURE emp_stamp();
```

6.6. Séquences et fonctions en SQL

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

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));

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: 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
```

7 Dépendances fonctionnelles et Formes Normales

Problèmes liés aux redondances

Code Film Distributeur

code	titre	duree	annee	nom	adresse
A	Aliens	137	1982	Clean Kill Movies	45, walker street, houston
SF	Aliens	137	1982	Clean Kill Movies	45, walker street, houston
SF	Blade Runner	117	1982	SF Movies	13, Champs Elysee, Paris
CD	Casablanca	102	1942	Classique Film	2, Place Kleber, 67000 Strasb
W	Dances with Wolves	180	1990	Constance Film	Gumpendorferstrasse 17, A-106

- un film est stocké plusieurs fois s'il possède plusieurs catégories
- l'adresse d'un distributeur est stockée plusieurs fois s'il distribue plusieurs films
- La modification d'un attribut doit être effectuée à plusieurs endroits

….

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*

(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, $\mathbf{df_1}$, $\mathbf{df_2}$, ..., on note alors : $\mathbf{r \models df_1, df_2, \ldots}$

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

Objectifs:

- 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 relations et $DF = \{ A \rightarrow B, B \rightarrow C, AC \rightarrow D\}$ l'ensemble des dépendances fonctionnelles

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 |– par opposition à |= 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

système d'inférence $\rightarrow$ 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***, au sens de l'inclusion)

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** *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)

- 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** *n-m entre deux entités E_1 et E_2*, alors les clés de E_1 ***et*** de E_2 sont des clés de ***r***

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 $\{A_1, \ldots, A_n\}^+$

Exemple: $S=\{A \rightarrow D;\ A \rightarrow E;\ E \rightarrow C\}$

Fermeture $\{A\}^+ = \{A, D, E, C\}$

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 $\{A_1, \ldots, A_n\}^+$

Calcul d'une fermeture (Exemple)

Soit $F=\{A \rightarrow D;\ AB \rightarrow E;\ BI \rightarrow E;\ CD \rightarrow I;\ E \rightarrow C\}$

Calcul de la fermeture $(AE)^+$:

- au départ $(AE)^+ = AE$
- $A \rightarrow D$ permet d'ajouter D : $(AE)^+ = AED$
- $E \rightarrow C$ permet d'ajouter C : $(AE)^+ = AEDC$
- $CD \rightarrow I$ permet d'ajouter I : $(AE)^+ = AEDCI$

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

$\rightarrow$ 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, et *qu'aucun sous-ensemble de $\{A_1,\ldots,A_n\}$ n'est une super clé de 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,….*)

C'est à dire si chaque attribut contient une seule valeur

Exemple de schéma non en 1NF:

Titre	Acteurs
Casablanca	Humphrey Bogart, Ingrid Bergman
Perfect World	Kevin Costner, Clint Eastwood
The Terminator	Arnold Schwarzenegger, Linda Hamilton, Michael Biehn
Die Hard	Bruce Willis

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**.*

2ème Forme Normale:

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

ssi :

- *Elle est en 1NF*
- ***Aucun attribut non clé a_1 ne dépend fonctionnellement d'un attribut a_2 d'une clé** (ou qui constitue pas à lui seul une clé)*

Deuxième Forme Normale (exemple)

Schéma 1:

joueur(Personne, Sport, Taille,Poids)

Personne $\rightarrow$ Taille, Poids

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

Les attributs *Taille* et *Poids* dépendent fonctionnellement de *Personne*

Schéma 2 :

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

La table originale **joueur** peut alors être retrouvée par la jointure:
joueur= pratique ⊳⊲ physique

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 ensemble d'attributs qui ne constituent pas une clé, C'est à dire, on n'a pas : $L \rightarrow A$ avec L non clé, $A \notin L$ et $A \notin$ clé

Troisième Forme Normale (exemple)

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

La clé est (NVH), et on a les DF :

- **TYPE → MARQUE**
- **TYPE → PUISS**

3 NF:

VEHICULE (NVH, TYPE, COULEUR)

MODELE (TYPE, MARQUE, PUISS)

Processus de décomposition préservant les DF

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

Une DF atomique $X \rightarrow Y$ est perdue par une décomposition où X et Y ne sont pas dans la même table r_i

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 fonctionnelle $L \rightarrow A$ avec
    L non clé, $A \notin L$ et $A \notin$ clé
  - On projette alors R en deux tables:
    - une sur les attributs R-{A}
    - une sur les attributs LA (cette 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)

Objectifs: pas de dépendance entre une clef (ou d'une partie d'une clef) et un attribut non clef.

Définition:

*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,

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

Une relation est en BCNF si et seulement si les seules dépendances fonctionnelles non triviales sont celles pour lesquelles 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(A, B, C, D, E) B → C

  R1(B, C)

  R2(A, B, D, E)

- **3NF**

  R(A, B, C, D, E) C → D

  R1(C, D)

  R2(A, B, C, E)

- **BCNF**

  R(A, B, C, D, E) C → B et (A,C) est clé secondaire

  R1(C, B)

  R2(A, C, 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 [ALL] <privilèges list> ON
     <database_name> FROM <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 IF EXISTS marquedeposees;
   CREATE DATABASE marquedeposees ;

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

   \c marquedeposees;
   CREATE SCHEMA donnees;
   SET SEARCH_PATH TO donnees, PUBLIC ;
```

Création de schémas : exemple (2)

```
-- Initialisation du schéma 'donnees'
   DROP TABLE IF EXISTS marque ;
   DROP TABLE IF EXISTS societe;
   DROP TABLE IF EXISTS classe ;
   DROP TABLE IF EXISTS 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 GROUP etudiant;
        GRANT REFERENCES ON TABLE pays TO  GROUP etudiant;
        GRANT SELECT ON TABLE societe TO  GROUP etudiant;
        GRANT REFERENCES ON TABLE societe TO GROUP etudiant;

-- 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

Gestion de la concurrence: motivations

Plusieurs utilisateurs sont autorisés à manipuler la même base de données

➔ il faut s'assurer que les actions des uns ne seront pas préjudiciables aux autres

Le *contrôle de concurrence* doit rendre le partage des données complètement transparent aux utilisateurs

Un SGBD peut fonctionner sur plusieurs machines … une machine peut tomber en panne

➔ SGBD doit garantir que la base restera dans un état cohérent même après une panne : *sûreté de fonctionnement* et *mécanisme de reprise* (sur panne)

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-à-vis des contraintes

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

Transactions ⟶ ACID

- ***Atomicité*** :

  Une transaction s'effectue de manière atomique:

  - Soit toutes les mises à jour sont enregistrées dans la base (COMMIT)
  - Soit aucune mise à jour n'est enregistrée dans la base (ABORT)

- ***Cohérence*** :

  Lorsque des transactions s'exécutent de manière concurrente le SGBD doit gérer l'exécution et contrôler l'accès aux ressources pour éviter les incohérences

- ***Isolation***

  Une transaction s'exécute comme si elle était seule à accéder à la base; elle ne voit pas les opérations effectuées par les autres transactions en cours

- ***Durabilité***

  Les effets d'une transaction terminée normalement (COMMIT) ne peuvent pas être annulés, sauf par une autre transaction.

Transactions: définition informelle

- *Une transaction est une unité de traitement séquentiel* (séquence d'actions cohérente), exécutée pour le compte d'un usager, qui appliquée à une base de données cohérente restitue une base de données cohérente.
- Les *Contraintes d'Intégrité* sont donc des invariants pour les transactions

Contraintes d'intégrité:

- Statiques: NOT NULL, PRIMARY KEY, UNIQUE, CHECK, DEFAULT, FOREIGN KEY,ON DELETE CASCADE, ON DELETE RESTRICT, ON DELETE SET NULL, ON UPDATE CASCADE
- Dynamique: programmées

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 1:
  - **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)

Gestion concurrentielle des transactions (suite)

Exemple 2 :

Soit la contrainte d'intégrité Y=2X, les deux transactions T1 et T2 s'exécutent, la base est alors dans un état incohérent

Temps	T1	Etat de la base	T2
t1	X:=10	(X=5 , Y=10)	———
t2	écrire (X)	(X=10, Y=10)	———
t3	———		X:=30
t4	———	(X=30, Y=10)	écrire (X)
t5	———		Y:= 60
t6	Y:= 20	(X=30, Y=60)	écrire (Y)
t7	écrire (Y)	(X=30, Y=20)	———

Ordonnancement des transactions

- **Ordonnancement** :
  - Transaction T1: (T1, lire, A), (T1, écrire, A)
  - Transaction T2: (T2, lire, A), (T2, lire, B), (T2, écrire, A), (T2, écrire, B)
  - **Ordonnancement** 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** (non des opérations); exemples: (T1,T2), (T2,T1)
- **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' des **opérations** du même ensemble de transactions sont équivalents si pour toutes opérations conflictuelles p et q de O et O', p est avant q dans O ssi p est aussi avant q dans O'

Ordonnancement des transactions - Exemples

- **Soit :**

  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) **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

Verrouillage

Un *verrou* est un objet "système" à deux états (ouvert, fermé)

supportant deux opérations *atomiques:*

- verrouiller(verrou)(LOCK)
- déverrouiller(verrou)(UNLOCK)

Un verrou permet de contrôler l'accès à un objet en exclusion mutuelle: à un instant donné, une transaction au plus peut avoir accès à l'objet

Problèmes: famine, inter-blocage (deadlock)

Verrouillage (suite)

Granularité du verrouillage par une transaction:

base, table, tuple, attribut, page disque

Une granularité fine (attributs, tuples):

⇒ un parallélisme élevé

⇒ un surcoût en verrouillage et en place élevé

Une grosse granularité (relations, base):

⇒ un parallélisme plus faible

⇒ un sur coût négligeable

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 .
- $\Delta_t = f(L_C, \Delta_M)$ où :
  - Δ_t est une modification à effectuer sur le tuple *t*, pour satisfaire les contraintes,
  - L_C est un ensemble de tuples dont dépend le calcul de Δc, et qu'on appelle également "**lectures critiques**"
  - Δ_M 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 :

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

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

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

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

La transaction :

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

ne sera correcte que si 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:

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

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

$\rightarrow$ Le verrouillage à effectuer est 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

10.5 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 compte (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