ICPJ, 21928, CNAM, Paris

BASES DE DONNÉES

Relationnelles

M. Scholl et D. Vodislav

(scholl|vodislav)@cnam.fr

2003/2004

INTRODUCTION

Objectif

OBJECTIF:

Comprendre et Maitriser la technologie relationnelle

BIBLIOGRAPHIE

Ouvrages en français

1. P. Rigaux, *Cours bases de données*, cedric/cnam.fr/vertigo voir à support de cours.
2. Date C.J, *Introduction aux Bases de Données*, Vuibert, 970 Pages, Janvier 2001

Ouvrages en anglais

1. R. Ramakrishnan et J. Gehrke, *DATABASE MANAGEMENT SYSTEMS*, MacGraw Hill
2. R. Elmasri, S.B. Navathe, *Fundamentals of database systems*, 3e édition, 1007 pages, 2000, Addison Wesley
3. Ullman J.D. and Widom J. *A First Course in Database Systems*, Prentice Hall, 1997

4. Garcia-Molina H., Ullman J. and Widom J., *Implementation of Database Systems*, Prentice Hall, 1999
5. Ullman J.D., *Principles of Database and Knowledge-Base Systems*, 2 volumes, Computer Science Press
6. Abiteboul S., Hull R., Vianu V., *Foundations of Databases*, Addison-Wesley

Le standard SQL

1. Date C.J., *A Guide to the SQL Standard*, Addison-Wesley

Trois Systèmes

1. Date C.J., *A Guide to DB2*, Addison-Wesley
2. Date C.J., *A Guide to Ingres*, Addison-Wesley
3. *ORACLE version 7 Server Concepts Manual 1992 Oracle*

Plan

Plan général du cours

1. Modèle et langages relationnels
2. Aspects systèmes des modèles relationnels
3. Concurrence et reprise sur pannes

Plan de la première partie

1. Introduction
2. Modèle relationnel
3. Algèbre Relationnelle
4. Langage de requête SQL
5. Calcul relationnel

Plan de la deuxième partie

1. Fichiers, Organisation Physique,
2. Différents tyes d’index, Arbre-B
3. Algorithmes de Jointure
4. Optimisation des requêtes;
5. l’exemple d’ORACLE.

Exemples d'Applications

1. **CLASSIQUES**
   - Gestion (salaires, stocks, . . .)
   - Transactionnel (comptes, centrales d'achat, . . .)
   - Réservations (avions, trains, . . .)
2. **PLUS RECENTES**
   - Librairie et commerce électroniques (bibliothèques, journaux, web, . . .)
   - Documentation technique (nomenclature, plans, dessins,. . .)
   - Multimédia (textes, images, son, vidéo, . . .)
   - Géographique (cartes routières, thématiques, . . .)
   - Génie Logiciel (programmes, manuels, . . .)

Comment Stocker et Manipuler les Données?

DONNÉES → BASE DE DONNÉES (B.D.)

- Une B.D. est un *GROS ENSEMBLE* d'informations *STRUCTURÉES* mémorisées sur un support *PERMANENT*.

LOGICIEL → SGBD

- Un Système de Gestion de Bases de Données (SGBD) est un logiciel de *HAUT NIVEAU* qui permet de manipuler ces informations.

Diversité -> Complexité

Diversité des utilisateurs, des interfaces et des Architectures:

1. diversité des utilisateurs: administrateurs, programmeurs, non informaticiens, . . .
2. diversité des interfaces: utilisateur final, langages BD, menus, saisies, rapports, . . .
3. diversité des architectures : centralisé, distribué, accès à plusieurs bases hétérogènes accessibles par réseau, pair à pair

FONCTIONNALITÉS d’un SGBD

Chaque niveau du SGBD réalise un certain nombre de fonctions :

NIVEAU PHYSIQUE

- Accès aux données, gestion sur mémoire secondaire (fichiers) des données, des index
- Partage de données et gestion de la concurrence d’accès
- Reprise sur pannes (fiabilité)
- Distribution des données et interopérabilité (accès aux réseaux)

NIVEAU LOGIQUE

- Définition de la structure de données : Langage de Description de Données (LDD)
- Consultation et Mise à Jour des données : Langages de Requêtes (LR) et Langage de Manipulation de Données (LMD)

Fonctionnalités du SGBD au NIVEAU EXTERNE

- Gestion des Vues
- Environnement de programmation (intégration avec un langage de programmation)
- Interfaces conviviales et Langages de 4e Génération (L4G)
- Outils d'aides (e.g. conception de schémas)
- Outils de saisie, d'impression d'états
- Débogueurs
- Passerelles (réseaux, autres SGBD,etc. . .)

En Résumé, on Veut Gérer

un GROS VOLUME D'INFORMATIONS

- Persistantes (années) et fiables (protection sur pannes)
- Partageables (utilisateurs, programmes)
- Manipulées indépendamment de leur organisation physique

Définition du schéma de données

Modèles de données

Un modèle de données est caractérisé par :

- Une structuration des objets
- Des opérations sur ces objets

Dans un SGBD, il existe plusieurs modèles plus ou moins abstraits des mêmes objets, e.g. :

- le modèle conceptuel : la description du système d'information
- le modèle logique : interface avec le SGBD
- le modèle physique : fichiers

$\Rightarrow$ ces différents modèles correspondent aux niveaux dans l'architecture d'un SGBD.

Modèle Conceptuel: Exemple Entité-Association

- Modèle très abstrait, pratique pour :
  - l'analyse du monde réel
  - la conception du système d'information
  - la communication entre différents acteurs de l'entreprise
- **Mais n'est pas associé à un langage**.

DONC UNE STRUCTURE

MAIS PAS

D'OPÉRATIONS

Modèle logique

1. **Langage de définition de données (LDD)** pour décrire la structure.
2. **Langage de manipulation de données (LMD)** pour appliquer des opérations aux données.

Ces langages sont **abstraits** :

1. Le LDD est indépendant de la représentation physique des données.
2. Le LMD est indépendant de l'implantation des opérations.

Les avantages de l’abstraction

1. **Simplicité d’accès**

   Les structures et les langages sont plus simples, donc plus faciles pour l’usager non expert.

2. **INDÉPENDANCE PHYSIQUE**.

   On peut modifier l’implantation physique sans modifier les programmes d’application

3. **INDÉPENDANCE LOGIQUE**.

   On peut modifier les programmes d’application sans toucher à l’implantation.

HISTORIQUE DES SGBD

À chaque génération correspond un modèle logique

Les premiers étaient peu abstraits (navigationnels)

$<$ 60	S.G.F. (e.g. COBOL)	
mi-60	HIÉRARCHIQUE IMS (IBM)	navigationnel
	RÉSEAU (CODASYL)	navigationnel
73-80	RELATIONNEL	déclaratif
mi-80	RELATIONNEL	explosion sur micro
Fin 80	ORIENTÉ-OBJET	déclaratif + navigationnel
Fin 90	Objet-relationnel	nouvelles appli
2003	XML	documents

Opérations sur les données

Exemples de questions (requêtes) posées à la base

Insérer un employé nommé Jean

Augmenter Jean de 10%

Détruire Jean

Chercher les employés cadres

Chercher les employés du département comptabilité

Salaire moyen des employés comptables, avec deux enfants,

nés avant 1960 et travaillant à Paris

Les requêtes sont émises avec un *langage de requêtes* (SQL2, OQL, SQL3, XQUERY, etc.).

Le Traitement d'une Requête

- **ANALYSE SYNTAXIQUE**
- **OPTIMISATION**

  Génération (par le SGBD) d'un programme optimisé
  à partir de la
  connaissance de la structure des données, de l'existence
  d'index, de statistiques sur les données.
- **EXÉCUTION POUR OBTENIR LE RÉSULTAT**.

  NB : on
  doit tenir compte du fait que d'autres utilisateurs sont
  peut-être en train de modifier les données qu'on interroge !

Concurrence d’accès et reprise sur pannes

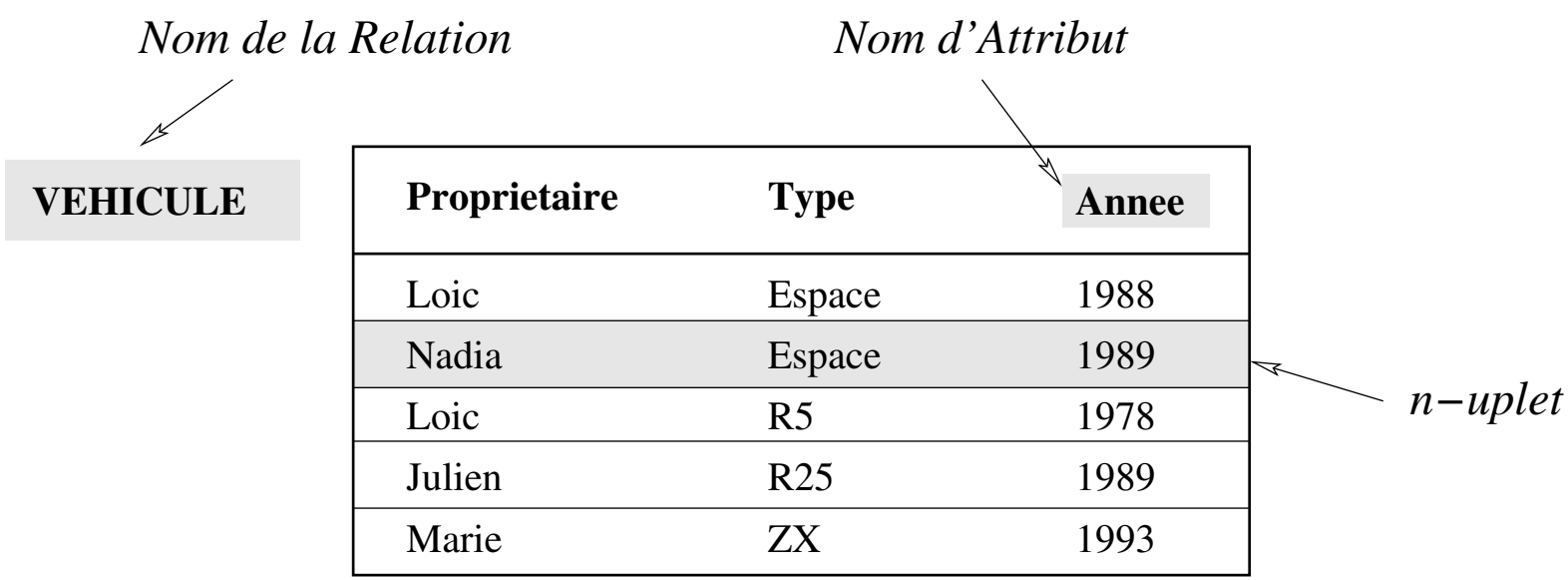
Plusieurs utilisateurs doivent pouvoir accéder en même temps aux mêmes données. Le SGBD doit savoir :

- Gérer les conflits si les deux font des mises-à-jour sur les mêmes données.
- Offrir un mécanisme de retour en arrière si on décide d’annuler des modifications en cours.
- Restaurer la base et revenir à un état cohérent en cas de panne.

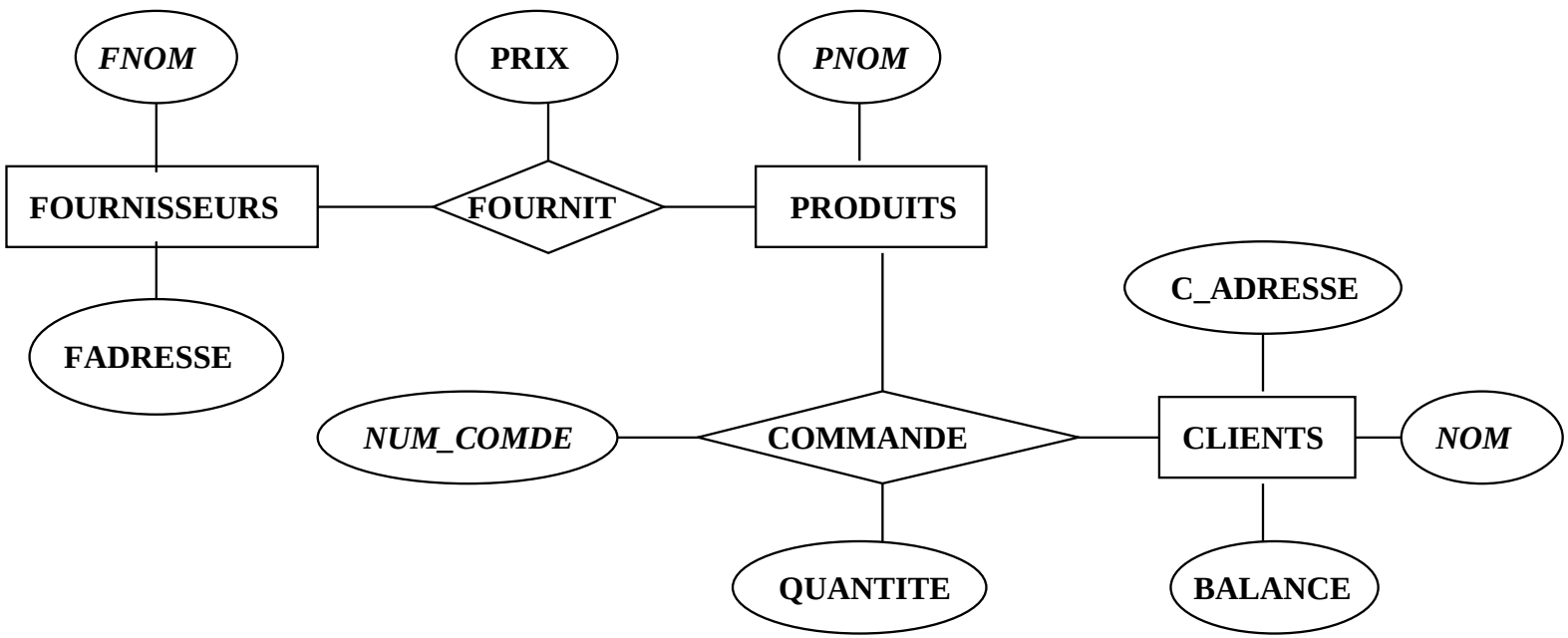
LE MODèLE RELATIONNEL

Présentation Générale

Exemple de Relation



un exemple de conception (schéma EA)



Représentation par un ensemble de relations

FOURNISSEUR

FNOM	FADRESSE
Abounayan	92190 Meudon
Cima	75010 Paris
Preblocs	92230 Gennevilliers
Sarnaco	75116 Paris

FOURNITURE

FNOM	PNOM	PRIX
Abounayan	sable	300
Abounayan	briques	1500
Preblocs	parpaing	1200
Sarnaco	parpaing	1150
Sarnaco	ciment	125

COMMANDES

NUM_COMDE	NOM	PNOM	QUANTITE
1	Jean	briques	5
2	Jean	ciment	10
3	Paul	briques	3
4	Paul	parpaing	9
5	Vincent	parpaing	7

CLIENTS

NOM	CADRESSE	BALANCE
Jean	75006 Paris	-12000
Paul	75003 Paris	0
Vincent	94200 Ivry	3000
Pierre	92400 Courbevoie	7000

Retour sur le Modèle EA

1) Entités: un fournisseur est une entité

1. Type d'entités: tous les fournisseurs ont même type: *Fournisseurs*
2. Attribut: représente une propriété (caractéristique) d'1 entité, exemple: FNOM
3. concept: synonyme d'entité.

Modèle EA

2) Associations: fournit est une association

1. relie deux (ou plus) entités, exemple: *fournit* relie Fournisseurs et Produits
2. Attribut: Une association peut avoir des attributs propres
3. role: synonyme d'association.
4. *association 1,n* entre *E* et *F*: à 1 entité de type *E* on peut associer 1 ou plusieurs entités de type *F*: exemple rajoutons l'association *siège social* entre *Fournisseurs* et *Ville*
5. *Association n,n*: à 1 entité *E* correspond 1 ou plusieurs entités *F* et réciproquement, exemple: *Founit*.

Clé

1. Attribut ou groupe d'attributs identifiant 1 entité: 2 entités de même type ne peuvent pas avoir la même valeur de clé, exemple *FNOM* pour Fournisseurs (hyp: 2 forunisseurs ne peuvent avoir le même nom)
2. *Clé primaire*: il peut y avoir plusieurs clés, exemple : si on rajoute le no de SS comme attribut à Fournisseurs, on a deux clés. Une est choisie comme clé primaire, e.g; NoSS

Passage EA-Relationnel

A partir d'un schéma entité-association (niveau conceptuel) comment passer au niveau logique (comment choisir les relations)?

1. Une relation par type d'entité (e.g; Fournisseur). A une entité correspond un nuplet.
2. la clé primaire de la relation (attribut(s) de la relation qui identifie un nuplet) est celle du type d'entité
3. Une relation par association n,n: Les attributs sont les clés primaires des relations reliées par l'association, ainsi que les attributs propres de l'association
4. *Clé étrangère*: si l'association A entre R et S est 1,n, (n entités de type S pour une entité de type R), on ne crée pas de relation associée à cette association: on rajoute dans la table qui représente S la clé de R comme attribut. Celle-ci et appelée clé étrangère.

5. exercice: rajouter le type d'entité *VILLE* (clé: nom de ville) et l'association *siège social* entre *VILLE* et *FOURNISSEURS*. Comment est modifiée la relation *FOURNISSEURS*?

Modèle relationnel: Définitions

Définitions

- Un Domaine est un ensemble de valeurs. Exemples : $\{0, 1\}$, N, l'ensemble des chaînes de caractères, l'ensemble des chaînes de caractères de longueur 10.
- Un ATTRIBUT prend ses valeurs dans un domaine. Plusieurs attributs peuvent avoir le même domaine.
- Un NUPLET est une liste de n valeurs $(v_1, \ldots, v_n)$ où chaque valeur v_i est la valeur d'un attribut A_i de domaine D_i : $v_i \in D_i$
- Le PRODUIT CARTÉSIEN $D_1 \times \ldots \times D_n$ entre des domaines $D_1, \ldots, D_n$ est l'ensemble de **tous** les nuplets $(v_1, \ldots, v_n)$ où $v_i \in D_i$.

- RELATION : soit $D_1, \ldots, D_n$ les domaines respectifs des attributs $A_1, \ldots, A_n$. Une relation R définie sur les attributs $A_1, \ldots, A_n$ est un sous-ensemble fini du produit cartésien $D_1 \times \ldots \times D_n$: R est un ensemble de nuplets.
- Une relation R est représentée sous forme d'une **table**. L'ordre des colonnes ou des lignes n'a pas d'importance. Les colonnes sont distinguées par les noms d'attributs et chaque ligne représente un élément de l'ensemble R (un nuplet).
- Un attribut peut apparaître dans plusieurs relations.
- Une BASE DE DONNÉES est un ensemble de relations.

- L'UNIVERS D'ATTRIBUTS D'UNE BASE DE DONNÉES est l'ensemble de tous les attributs des relations de la base.
- Le SCHÉMA D'UNE RELATION R est défini par le nom de la relation et la liste des attributs avec pour chaque attribut son domaine.
  Notation :

$$R(A_1 : D_1, \ldots, A_n : D_n)$$

  ou plus simplement :

$$R(A_1, \ldots, A_n)$$

  Exemple :

  VEHICULE(NOM:CHAR(20), TYPE:CHAR(10),
  ANNEE:ENTIER)

- Si la relation a n attributs (n colonnes), n est appelé ARITÉ de la relation. La relation $VEHICULE$ est d'arité 3.
- Le SCHÉMA D'UNE BASE DE DONNÉES est l'ensemble des schémas de ses relations.

Exemple de Base de Données

SCHÉMA :

- FOURNISSEURS(FNOM:CHAR(20), FADRESSE:CHAR(30))
- FOURNITURE(FNOM:CHAR(20), PNOM:CHAR(10), PRIX:ENTIER))
- COMMANDES(NUM_COMDE:ENTIER, NOM:CHAR(20), PNOM:CHAR(10), QUANTITE;ENTIER))
- CLIENTS(NOM: CHAR(20), CADRESSE:CHAR(30), BALANCE:RELATIF)

Exemple de Base de Données

UNIVERS D'ATTRIBUTS :

- U={FNOM, PNOM, NOM, FADRESSE, CADRESSE,
  PRIX, NUM_CODE, QUANTITE, BALANCE}

RELATION UNIVERSELLE :

- FPCC(FNOM, PNOM, NOM, FADRESSE, CADRESSE,
  PRIX, NUM_CODE, QUANTITE, BALANCE)

Opérations et Langages

Opérations sur une Base de Données Relationnelle

- LANGAGE DE DÉFINITION DES DONNÉES (définition et MAJ du schéma) :
  - Création et destruction d'une relation ou d'une base
  - Ajout, suppression d'un attribut

- LANGAGE DE MANIPULATION DES DONNÉES
  - Saisie des nuplets d'une relation
  - Affichage d'une relation
  - Modification d'une relation : insertion, suppression et maj des nuplets
  - Requêtes : consultation d'une relation ou calcul d'une nouvelle relation
- GESTION DES TRANSACTIONS
- GESTION DES VUES

Langages de Requêtes Relationnels

POUVOIR D'EXPRESSION : Qu'est-ce qu'on peut calculer ? Quelles opérations peut-on faire ?

Les langages de requête relationnels utilisent deux approches :

- calcul relationnel
- algèbre relationnelle

Les deux approches ont **même pouvoir d'expression**.

ALGÈBRE RELATIONNELLE

Algèbre Relationnelle

Opérations relationnels :

- une opération prend en entrée une ou deux relations
- le résultat est toujours une relation

5 Opérations de base (pour exprimer toutes les requêtes) :

- Opérations unaires : sélection, projection
- Opérations binaires : union, différence, produit cartésien
- Autres opérations qui s'expriment en fonction des 5 opérations de base : jointure (naturelle, θ-jointure), intersection, division

Projection

LA PROJECTION “ÉLIMINE” UNE OU PLUSIEURS COLONNES D’UNE RELATION.

Notation :

$$\pi A_1,A_2,...,A_k(R)$$

Projection: Exemples

a) On élimine la colonne C dans la relation R :

R	A	B	C
→	a	b	c
	d	a	b
	c	b	d
→	a	b	e
	e	e	a

$\Rightarrow$ $\pi_{A,B}(R)$

A	B
a	b
d	a
c	b
e	e

Le nuplet (a, b) n'apparaît qu'**une** fois dans la relation $\pi_{A,B}(R)$, bien qu'il existe **deux** nuplets (a, b, c) et (a, b, e) dans R.

Projection: Exemples

b) On élimine la colonne B dans la relation R (on garde A et C) :

R

A	**B**	**C**
a	b	c
d	a	b
c	b	d
a	b	e
e	e	a

$\Rightarrow$ $\pi_{A,C}(R)$

A	**C**
a	c
d	b
c	d
a	e
e	a

Sélection

Sélection sur la condition $\mathcal{C}$:

On garde les nuplets qui satisfont $\mathcal{C}$.

NOTATION :

$$\sigma_C(R)$$

Sélection: Exemples

a) On sélectionne les nuplets dans la relation R tels que l'attribut B vaut "b" :

R

A	B	C
a	b	1
d	a	2
c	b	3
a	b	4
e	e	5

$\Rightarrow$ $\sigma_{B=''b''}(R)$

A	B	C
a	b	1
c	b	3
a	b	4

Sélection: Exemples

b) On sélectionne les nuplets tels que

$$(A ='' a'' \vee B ='' a'') \wedge C \leq 3 :$$

R

A	**B**	**C**
a	b	1
d	a	2
c	b	3
a	b	4
e	e	5

$\Rightarrow \quad \sigma_{(A=''a'' \vee B=''a'') \wedge C \leq 3}(R)$

A	**B**	**C**
a	b	1
d	a	2

Sélection: Exemples

c) On sélectionne les nuplets tels que la 1re et la 2e colonne sont identiques :

R

A	B	C
a	b	1
d	a	2
c	b	3
a	b	4
e	e	5

$\Rightarrow \quad \sigma_{A=B}(R)$

A	B	C
e	e	5

Condition de Sélection

La condition $\mathcal{C}$ d'une sélection peut être une **formule logique** quelconque avec des **et ($\wedge$)** et des **ou ($\vee$)** entre termes de la forme $A_i \theta A_j$ et $A_i \theta a$ où

- A_i et A_j sont des attributs,
- a est un élément (une valeur) du domaine de A_i,
- θ est l'un de $=, <, \leq, >, \geq, \neq$.

Expressions de l'Algèbre Relationnelle

- le résultat d'une opération est une **relation**
- sur cette relation, on peut faire une **autre opération** de l'algèbre

$\Rightarrow$ *Les opérations peuvent être* composées *pour former des expressions de l'algèbre relationnelle.*

Expressions de l'Algèbre Relationnelle

EXEMPLE : $COMMANDES(NOM, PNOM, NUM, QTE)$

$$R'' = \pi_{PNOM}(\overbrace{\sigma_{NOM=''Jean''}(COMMANDES)}^{R'})$$

La relation $R'(NOM, PNOM, NUM, QTE)$ contient les nuplets dont l'attribut NOM a la valeur $''Jean''$. La relation $R''(PNOM)$ contient tous les produits commandés par Jean.

Produit Cartésien

- NOTATION : $R \times S$
- ARGUMENTS : 2 relations quelconques :

$$R(A_1, A_2, \ldots, A_n) \;\; S(B_1, B_2, \ldots, B_k)$$

- SCHÉMA DE $T = R \times S$: $T(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_k)$
- VALEUR DE $T = R \times S$: ensemble de tous les nuplets ayant $n + k$ composants (attributs)
  - dont les n premiers composants forment un nuplet de R
  - et les k derniers composants forment un nuplet de S

Exemple de Produit Cartésien

R

	A	**B**
	1	1
$\mid R \mid$	1	2
	3	4

S

	C	**D**	**E**
	a	b	a
$\mid S \mid$	a	b	c
	b	a	a

$\Rightarrow$

$\mathbf{R \times S}$

$| R | \times | S |$

A	**B**	**C**	**D**	**E**
1	1	a	b	a
1	1	a	b	c
1	1	b	a	a
1	2	a	b	a
1	2	a	b	c
1	2	b	a	a
3	4	a	b	a
3	4	a	b	c
3	4	b	a	a

Jointure Naturelle

- NOTATION : $R \bowtie S$
- ARGUMENTS : 2 relations quelconques :

  $$R(A_1, \ldots, A_m, X_1, \ldots, X_k) \;\; S(B_1, \ldots, B_n, X_1, \ldots, X_k)$$

  où $X_1, \ldots, X_k$ sont les attributs en commun.
- SCHÉMA DE $T = R \bowtie S$: $T(A_1, \ldots, A_m, B_1, \ldots, B_n, X_1, \ldots, X_k)$
- VALEUR DE $T = R \bowtie S$: ensemble de tous les nuplets ayant $m + n + k$ attributs dont les m premiers et k derniers composants forment un nuplet de R et les $n + k$ derniers composants forment un nuplet de S.

Jointure Naturelle: Exemple

R

A	**B**	**C**
a	b	c
d	b	c
b	b	f
c	a	d

S

B	**C**	**D**
b	c	d
b	c	e
a	d	b

⇒

R ⋈ S

A	**B**	**C**	**D**
a	b	c	d
a	b	c	e
d	b	c	d
d	b	c	e
c	a	d	b

Jointure Naturelle

Soit $U = \{A_1, \ldots, A_m, B_1, \ldots, B_n, X_1, \ldots, X_k\}$ l'ensemble des attributs des 2 relations et $V = \{X_1, \ldots, X_k\}$ l'ensemble des attributs en commun.

$$R \bowtie S = \pi_U(\sigma_{\forall A \in V : R.A = S.A}(R \times S))$$

NOTATION : $R.A$ veut dire "l'attribut A de la relation R".

Jointure Naturelle: Exemple

R

A	B
1	a
1	b
4	a

S

A	B	D
1	a	b
2	c	b
4	a	a

$\Rightarrow$

R × S	**R.A**	**R.B**	**S.A**	**S.B**	**D**
	1	a	1	a	b
→	**1**	**a**	**2**	**c**	b
→	**1**	**a**	**4**	**a**	a
→	**1**	**b**	**1**	**a**	b
→	**1**	**b**	**2**	**c**	b
→	**1**	**b**	**4**	**a**	a
→	**4**	**a**	**1**	**a**	b
→	**4**	**a**	**2**	**c**	b
	4	a	4	a	a

⇓

R ⋈ S

A	B	D
1	a	b
4	a	a

$$\Leftarrow \qquad \pi_{R.A,R.B,D}\big(\sigma_{R.A=S.A \wedge R.B=S.B}(R \times S)\big)$$

Jointure Naturelle: Algorithme

Pour chaque nuplet a dans R et pour chaque nuplet b dans S :

1. on concatène a et b et on obtient un nuplet qui a pour attributs

$$\overbrace{A_1, \ldots, A_m, X_1, \ldots, X_k}^{a}, \overbrace{B_1, \ldots, B_n, X_1, \ldots, X_k}^{b}$$

2. on ne le garde que si chaque attribut X_i de a est égal à l'attribut X_i de b : $\forall_{i=1..k} a.X_i = b.X_i$.

3. on élimine les valeurs (colonnes) dupliquées : on obtient un nuplet qui a pour attributs

$$\overbrace{A_1, \ldots, A_m}^{a}, \overbrace{B_1, \ldots, B_m}^{b}, \overbrace{X_1, \ldots, X_k}^{a\ et\ b}$$

θ-Jointure

- ARGUMENTS : 2 relations quelconques :

$$R(A_1, \ldots, A_m) \qquad S(B_1, \ldots, B_n)$$

- NOTATION : $R \bowtie_{A_i \theta B_j} S, \theta \in \{=, \neq, <, \leq, >, \geq\}$
- SCHÉMA DE $T = R \bowtie_{A_I \theta B_J} S$: $T(A_1, \ldots, A_m, B_1, \ldots, B_n)$
- VALEUR DE $T = R \bowtie_{A_I \theta B_J} S$: $T = \sigma_{A_i \theta B_j}(R \times S))$
- ÉQUIJOINTURE : θ est l'**égalité**.

θ-Jointure: Exemple

R

A	B
1	a
1	b
3	a

S

C	D	E
1	b	a
2	b	c
4	a	a

T := R $\times$ S

	A	**B**	**C**	**D**	**E**
	1	a	1	b	a
	1	a	2	b	c
	1	a	4	a	a
	1	b	1	b	a
	1	b	2	b	c
	1	b	4	a	a
$A > C \rightarrow$	**3**	a	**1**	b	a
$A > C \rightarrow$	**3**	a	**2**	b	c
	3	a	4	a	a

$\Rightarrow$

$\sigma_{A \leq C}$**(T)**
= **R** $\bowtie_{A \leq C}$ **S**

A	**B**	**C**	**D**	**E**
1	a	1	b	a
1	a	2	b	c
1	a	4	a	a
1	b	1	b	a
1	b	2	b	c
1	b	4	a	a
3	a	4	a	a

Équijointure: Exemple

R

A	B
1	a
1	b
3	a

S

C	D	E
1	b	a
2	b	c
4	a	a

T := R $\times$ S	**A**	**B**	**C**	**D**	**E**
$B \neq D \rightarrow$	1	**a**	1	**b**	a
$B \neq D \rightarrow$	1	**a**	2	**b**	c
	1	a	4	a	a
	1	b	1	b	a
	1	b	2	b	c
$B \neq D \rightarrow$	1	**b**	4	**a**	a
$B \neq D \rightarrow$	3	**a**	1	**b**	a
$B \neq D \rightarrow$	3	**a**	2	**b**	c
	3	a	4	a	a

$\Rightarrow$

$\sigma_{B=D}$**(T)**
=**R** $\bowtie_{B=D}$ **S**

A	**B**	**C**	**D**	**E**
1	a	4	a	a
1	b	1	b	a
1	b	2	b	c
3	a	4	a	a

Équijointure vs. Jointure Naturelle

$$IMMEUBLE(ADI, NBETAGES, DATEC, PROP)$$

$$APPIM(ADI, NAP, OCCUP, ETAGE)$$

1. Nom du propriétaire de l'immeuble où est situé l'appartement occupé par *Durand* :

$$\pi_{PROP}(\overbrace{IMMEUBLE \bowtie \sigma_{OCCUP=''DURAND''}(APPIM)}^{JointureNaturelle})$$

2. Appartements occupés par des propriétaires d'immeuble :

$$\pi_{ADI,NAP,ETAGE}\overbrace{(APPIM \bowtie_{OCCUP=PROP} IMMEUBLE)}^{équijointure})$$

Autre Exemple de REQUÊTE : Nom et adresse des clients qui ont commandé des parpaings:

- Schéma Relationnel :

$$COMMANDES(PNOM, CNOM, NUM\_CMDE, QTE)$$

$$CLIENTS(CNOM, CADRESSE, BALANCE)$$

- Requête Relationnelle :

$$\pi_{CNOM,CADRESSE}(CLIENTS \bowtie \sigma_{PNOM=''PARPAING''}(COMMANDES))$$

Union

- ARGUMENTS : 2 relations de même schéma :

$$R(A_1, \ldots, A_m) \quad S(A_1, \ldots, A_m)$$

- NOTATION : $R \cup S$
- SCHÉMA DE $T = R \cup S$: $T(A_1, \ldots, A_m)$
- VALEUR DE T : Union ensembliste sur $D_1 \times \ldots \times D_m$:

$$T = \{t \mid t \in R \vee t \in S\}$$

Union: Exemple

R

A	B
a	b
a	c
d	e

S

A	B
a	b
a	e
d	e
f	g

$\Rightarrow$

R ∪ S

A	B
a	b
a	c
d	e
a	e
f	g

$\rightarrow$ $\rightarrow$

Différence

- ARGUMENTS : 2 relations de même schéma :

$$R(A_1, \ldots, A_m) \quad S(A_1, \ldots, A_m)$$

- NOTATION : $R - S$
- SCHÉMA DE $T = R - S$: $T(A_1, \ldots, A_m)$
- VALEUR DE T : Différence ensembliste sur $D_1 \times \ldots \times D_m$:

$$T = \{t \mid t \in R \wedge t \notin S\}$$

Différence: Exemple

R	A	B
→	a	b
	a	c
→	d	e

S	A	B
→	a	b
	a	e
→	d	e
	f	g

R - S	A	B
	a	c

S - R	A	B
	a	e
	f	g

Intersection

- ARGUMENTS : 2 relations de même schéma :

$$R(A_1, \ldots, A_m) \quad S(A_1, \ldots, A_m)$$

- NOTATION : $R \cap S$
- SCHÉMA DE $T = R \cap S$: $T(A_1, \ldots, A_m)$
- VALEUR DE T : Intersection ensembliste sur $D_1 \times \ldots \times D_m$:

$$T = \{t \mid t \in R \wedge t \in S\}$$

Intersection: Exemple

R	A	B
→	a	b
	a	c
→	d	e

S	A	B
→	a	b
	a	e
→	d	e
	f	g

R - S

A	B
a	c

R ∩ S = R - (R - S)

A	B
a	b
d	e

Semijointure

- ARGUMENTS : 2 relations quelconques :

$$R(A_1, \ldots, A_m, X_1, \ldots, X_k) S(B_1, \ldots, B_n, X_1, \ldots, X_k)$$

  où $X_1, \ldots, X_k$ sont les attributs en commun.

- NOTATION : $R \ltimes S$
- SCHÉMA DE $T = R \ltimes S$: $T(A_1, \ldots, A_m, X_1, \ldots, X_k)$
- VALEUR DE $T = R \ltimes S$: Projection sur les attributs de R de la jointure naturelle entre R et S.

Semijointure

La semijointure correspond à une sélection où la condition de sélection est définie par le biais d'une autre relation.

Soit $U = \{A_1, \ldots, A_m\}$ l'ensemble des attributs de R.

$$R \ltimes S = \pi_U(R \bowtie S)$$

Semijointure: Exemple

R

A	**B**	**C**
a	b	c
d	b	c
b	b	f
c	a	d

S

B	**C**	**D**
b	c	d
b	c	e
a	d	b

$\Rightarrow \pi_{A,B,C}(R \bowtie S) \Rightarrow$

R $\rtimes$ **S**

A	**B**	**C**
a	b	c
d	b	c
c	a	d

Division: Exemple

REQUÊTE : Clients qui commandent tous les produits:

COMM

NUM	NOM	PNOM	QTE
1	Jean	briques	100
2	Jean	ciment	2
3	Jean	parpaing	2
4	Paul	briques	200
5	Paul	parpaing	3
6	Vincent	parpaing	3

$R = \pi_{NOM,PNOM}(COMM)$:

R

NOM	PNOM
Jean	briques
Jean	ciment
Jean	parpaing
Paul	briques
Paul	parpaing
Vincent	parpaing

PROD

PNOM
briques
ciment
parpaing

$\Downarrow$

$R \div PROD$

NOM
Jean

Division: Exemple

R

A	B	C	D
a	b	x	m
a	b	y	n
a	b	z	o
b	c	x	o
b	d	x	m
c	e	x	m
c	e	y	n
c	e	z	o
d	a	z	p
d	a	y	m

S

C	D
x	m
y	n
z	o

R÷S

A	B
a	b
c	e

Division

- ARGUMENTS : 2 relations :

$$R(A_1, \ldots, A_m, X_1, \ldots, X_k) \quad S(X_1, \ldots, X_k)$$

où **tous** les attributs de S sont des attributs de R.

- NOTATION : $R \div S$
- SCHÉMA DE $T = R \div S$: $T(A_1, \ldots, A_m)$
- VALEUR DE $T = R \div S$:

$$R \div S = \{(a_1, \ldots, a_m) \mid \forall (x_1, \ldots, x_k) \in S : (a_1, \ldots, a_m, x_1, \ldots, x_k) \in R\}$$

Division

La division s'exprime en fonction du produit cartésien, de la projection et de la différence : $R \div S = R_1 - R_2$ où

$$R_1 = \pi_{A_1,...,A_m}(R) \; et \; R_2 = \pi_{A_1,...,A_m}((R_1 \times S) - R)$$

Renommage

- NOTATION : ρ
- ARGUMENTS : 1 relation :

$$R(A_1, \ldots, A_n)$$

- SCHÉMA DE $T = \rho_{A_i \to B_i} R$: $T(A_1, \ldots, A_{i-1}, B_i, A_{i+1}, \ldots, A_n)$
- VALEUR DE $T = \rho_{A_i \to B_i} R$: $T = R$. La valeur de R est inchangée.
  Seul le nom de l'attribut A_i a été remplacé par B_i

SQL

Principe

- SQL (Structured Query Language) est le Langage de Requêtes standard pour les SGBD relationnels
- Expression d'une requête par un bloc *SELECT FROM WHERE*

  SELECT <liste des attributs a projeter>
  FROM <liste des relations arguments>
  WHERE <conditions sur un ou plusieurs attributs>

- Dans les requêtes simples, la correspondance avec l'algèbre relationnelle est facile à mettre en évidence.

Historique*

SQL86 - SQL89 ou SQL1 La référence de base:

- Requêtes compilées puis exécutées depuis un programme d'application.
- Types de données simples (entiers, réels, chaînes de caractères de taille fixe)
- Opérations ensemblistes restreintes (UNION).

SQL92 ou SQL2 Standard actuel:

- Requêtes dynamiques: exécution différée ou immédiate
- Types de données plus riches (intervalles, dates, chaînes de caractères de taille variable)
- Différents types de jointures: jointure naturelle, jointure externe
- Opérations ensemblistes: différence (EXCEPT), intersection

(INTERSECT)

- Renommage des attributs dans la clause SELECT

SQL3 (en cours) : SQL devient un langage de programmation :

- Extensions orientées-objet
- Opérateur de fermeture transitive (recursion)
- ...

Expressions de Base

Projection

Soit le schéma de relation
COMMANDES(NUM,CNOM,PNOM,QUANTITE)

REQUÊTE: *Information sur toutes les commandes*

SQL:

```
SELECT NUM,CNOM,PNOM,QUANTITE
FROM   COMMANDES
```

ou

```
SELECT *
FROM   COMMANDES
```

Projection : Distinct

Soit le schéma de relation
COMMANDES(NUM,CNOM,PNOM,QUANTITE)

REQUÊTE: *Produits commandés*

```
SELECT PNOM
FROM   COMMANDES
```

NOTE: Contrairement à l'algèbre relationnelle, SQL n'élimine pas les dupliqués. Pour les éliminer on utilise DISTINCT :

```
SELECT DISTINCT PNOM
FROM   COMMANDES
```

Le DISTINCT peut être remplacé par la clause UNIQUE dans certains systèmes

Sélection

Soit le schéma de relation
COMMANDES(NUM,CNOM,PNOM,QUANTITE)

REQUÊTE: *Produits commandés par Jean*

ALGÈBRE: $\pi_{PNOM}(\sigma_{CNOM="JEAN"}(COMMANDES))$

SQL:

```
SELECT PNOM
FROM   COMMANDES
WHERE  CNOM = 'JEAN'
```

REQUÊTE: *Produits commandés par Jean en quantité supérieure à 100*

SQL:

```
SELECT PNOM
FROM   COMMANDES
WHERE  CNOM = 'JEAN'
AND    QUANTITE > 100
```

Conditions de sélection en SQL : Conditions simples

Les conditions de base sont exprimées de deux façons:

1. *attribut comparateur valeur*
2. *attribut comparateur attribut*

où $comparateur$ est $=, <, >, <>, \ldots,$

Soit le schéma de relation **FOURNITURE**(PNOM,FNOM,PRIX)

REQUÊTE: *Produits de prix supérieur à 200F*

SQL:

```
SELECT PNOM
FROM   FOURNITURE
WHERE  PRIX > 2000
```

Soit le schéma de relation **FOURNITURE**(PNOM,FNOM,PRIX)

REQUÊTE: *Produits dont le nom est celui du fournisseur*

SQL:

```
SELECT PNOM
FROM   FOURNITURE
WHERE  PNOM = FNOM
```

Conditions de sélection en SQL : Suite

Le $comparateur$ est BETWEEN, LIKE, IN

Soit le schéma de relation **FOURNITURE**(PNOM,FNOM,PRIX)

REQUÊTE: *Produits avec un coût entre 1000F et 2000F*

SQL:

```
SELECT PNOM
FROM   FOURNITURE
WHERE  PRIX BETWEEN 1000 AND 2000
```

NOTE: La condition y BETWEEN x AND z est équivalente à $y <= z$
AND
$x <= y$.

Soit le schéma de relation
COMMANDES(NUM,CNOM,PNOM,QUANTITE)

REQUÊTE: *Clients dont le nom commence par "C"*

SQL:

```
SELECT CNOM
FROM   COMMANDES
WHERE  CNOM LIKE 'C%'
```

NOTE: Le littéral qui suit LIKE doit être une chaîne de caractères éventuellement avec des caractères jokers (_, %). Pas exprimable avec l'algèbre relationnelle.

Soit le schéma de relation **FOURNITURE**(PNOM,FNOM,PRIX)

REQUÊTE: *Produits avec un coût de 100F, de 200F ou de 300F*

SQL:

```
SELECT PNOM
FROM   FOURNITURE
WHERE  PRIX IN {100,200,300}
```

NOTE: La condition x IN $\{a, b, \ldots, z)$ est équivalente à $x = a$ OR $x = b$ OR . . . OR $x = z$.

Jointure

Soit le schéma de relations

COMMANDES(NUM,CNOM,PNOM,QUANTITE)
FOURNITURE(PNOM,FNOM,PRIX)

REQUÊTE: *Nom, Coût, Fournisseur des Produits commandés par Jean*

ALGÈBRE :

$$\pi_{PNOM,PRIX,FNOM}(\sigma_{CNOM=\text{"}JEAN\text{"}}(COMMANDES) \bowtie (FOURNITURE))$$

SQL :

```
SELECT COMMANDES.PNOM, PRIX, FNOM
FROM   COMMANDES, FOURNITURE
WHERE  CNOM = 'JEAN' AND
       COMMANDES.PNOM = FOURNITURE.PNOM
```

NOTE: Cette requête est équivalente à une jointure naturelle. Noter qu'il faut toujours expliciter les attributs de jointure.

NOTE: SELECT COMMANDES.PNOM, PRIX, FNOM FROM COMMANDES, FOURNITURE équivaut à un produit cartésien des deux relations, suivi d'une projection.

Soit le schéma de relation **FOURNISSEUR**(FNOM,STATUT,VILLE)

REQUÊTE: *Fournisseurs qui habitent deux à deux dans la même ville*

SQL:

```
SELECT PREM.FNOM, SECOND.FNOM
FROM   FOURNISSEUR PREM, FOURNISSEUR SECO
WHERE  PREM.VILLE =  SECOND.VILLE AND
       PREM.FNOM < SECOND.FNOM
```

La deuxième condition permet

1. l'élimination des paires (x,x)
2. d'éviter d'obtenir au résultat à la fois (x,y) et (y,x)

NOTE: PREM représente une instance de FOURNISSEUR, SECOND une autre instance de FOURNISSEUR.

Soit le schéma de relation **EMPLOYE**(EMPNO,ENOM,DEPNO,SAL)

REQUÊTE: *Nom et Salaire des Employés gagnant plus que l'employé de numéro 12546*

ALGÈBRE:

$$\text{R1} := \pi_{SAL}(\sigma_{EMPNO=12546}(EMPLOYE))$$

$$\text{R2} := \pi_{ENOM,EMPLOYE.SAL}((EMPLOYE) \bowtie_{EMPLOYE.SAL>R1.SAL} (R1))$$

SQL:

```
SELECT E1.ENOM, E1.SAL
FROM   EMPLOYE E1, EMPLOYE E2
WHERE  E2.EMPNO = 12546 AND
       E1.SAL > E2.SAL
```

Expressions Ensemblistes

Union

COMMANDES(NUM,CNOM,PNOM,QUANTITE)
FOURNITURE(PNOM,FNOM,PRIX)

REQUÊTE: *Produits qui coûtent plus que 1000F ou ceux qui sont commandés par Jean*

ALGÈBRE:

$$\pi_{PNOM}(\sigma_{PRIX>1000}(FOURNITURE))$$

$$\cup$$

$$\pi_{PNOM}(\sigma_{CNOM='Jean'}(COMMANDES))$$

SQL:

```
SELECT PNOM
FROM   FOURNITURE
WHERE  PRIX >= 1000
UNION
SELECT PNOM
FROM   COMMANDES
WHERE  CNOM = 'Jean'
```

NOTE: L'union élimine les dupliqués. Pour garder les dupliqués on utilise l'opération `UNION ALL` : le résultat contient chaque n-uplet $a + b$ fois, où a et b sont le nombre d'occurrences du n-uplet dans la première et la deuxième requête.

Différence

La différence ne fait pas partie du standard.

EMPLOYE(EMPNO,ENOM,DEPTNO,SAL)
DEPARTEMENT(DEPTNO,DNOM,LOC)

REQUÊTE: *Départements sans employés*

ALGÈBRE:
$\pi_{DEPTNO}(DEPARTEMENT) - \pi_{DEPTNO}(EMPLOYE)$

SQL:

```
SELECT DEPTNO
FROM   DEPARTEMENT
EXCEPT
SELECT DEPTNO
FROM   EMPLOYE
```

NOTE: La différence élimine les dupliqués. Pour garder les dupliqués on utilise l'opération EXCEPT ALL :

Intersection

L'intersection ne fait pas partie du standard.

EMPLOYE(EMPNO,ENOM,DEPTNO,SAL)
DEPARTEMENT(DEPTNO,DNOM,LOC)

REQUÊTE: *Départements ayant des employés qui gagnent plus que 20000F et qui se trouvent à Paris*

ALGÈBRE :

$$\pi_{DEPTNO}(\sigma_{LOC=\text{"}Paris\text{"}}(DEPARTEMENT))$$
$$\cap$$
$$\pi_{DEPTNO}(\sigma_{SAL>20000}(EMPLOYE))$$

SQL:

```
SELECT DEPTNO
FROM   DEPARTEMENT
WHERE  LOC = 'Paris'
INTERSECT
SELECT DEPTNO
FROM   EMPLOYE
WHERE  SAL > 20000
```

NOTE: L'intersection élimine les dupliqués. Pour garder les dupliqués on utilise l'opération `INTERSECT ALL` : le résultat contient chaque n-uplet $min(a, b)$ fois, où a et b sont le nombre d'occurrences du n-uplet dans la première et la deuxième requête.

Imbrication des Requêtes en SQL

Requêtes imbriquées simples

La Jointure s'exprime par deux blocs SFW imbriqués

Soit le schéma de relations

COMMANDES(NUM,CNOM,PNOM,QUANTITE)
FOURNITURE(PNOM,FNOM,PRIX)

REQUÊTE: *Nom, prix et fournisseurs des Produits commandés par Jean*

ALGÈBRE:

$$\pi_{PNOM,PRIX,FNOM}(\sigma_{CNOM="JEAN"}(COMMANDES) \bowtie (FOURNITURE))$$

SQL:

```
SELECT PNOM,PRIX,FNOM
FROM   FOURNITURE
WHERE  PNOM IN (SELECT PNOM
                FROM   COMMANDES
                WHERE  CNOM = 'JEAN')
```

ou

```
SELECT FOURNITURE.PNOM,PRIX,FNOM
FROM   FOURNITURE,COMMANDES
WHERE  FOURNITURE.PNOM = COMMANDES.PNOM
AND    CNOM = ''JEAN''
```

La Différence s'exprime aussi par deux blocs SFW imbriqués

Soit le schéma de relations

EMPLOYE(EMPNO,ENOM,DEPNO,SAL)
DEPARTEMENT(DEPTNO,DNOM,LOC)

REQUÊTE: *Départements sans employés*

ALGÈBRE :

$$\pi_{DEPTNO}(DEPARTEMENT) - \pi_{DEPTNO}(EMPLOYE)$$

SQL:

```
SELECT DEPTNO
FROM   DEPARTEMENT
WHERE  DETPNO NOT IN (SELECT DISTINCT DEPTNO
                      FROM   EMPLOYE)
```

ou

```
SELECT DEPTNO
FROM DEPARTEMENT
EXCEPT
SELECT DISTINCT DEPTNO
FROM EMPLOYE
```

Requêtes imbriquées plus complexes : ANY - ALL

Soit le schéma de relation **FOURNITURE**(PNOM,FNOM,PRIX)

REQUÊTE: *Fournisseurs des Briques à un coût inférieur au coût maximum des Ardoises*

```
SQL :    SELECT FNOM
         FROM   FOURNITURE
         WHERE  PNOM = 'Brique'
         AND    PRIX < ANY (SELECT PRIX
                            FROM   FOURNITURE
                            WHERE  PNOM = 'Ardoise')
```

NOTE: La condition $f\ \theta$ ANY (SELECT F FROM ...) est vraie ssi la comparaison $f\theta v$ est vraie au moins pour une valeur v du résultat du bloc (SELECT F FROM ...).

Soit le schéma de relations

COMMANDE(NUM,CNOM,PNOM,QUANTITE)
FOURNITURE(PNOM,FNOM,PRIX)

REQUÊTE: *Nom, Coût et Fournisseur des Produits commandés par Jean*

SQL:

```
SELECT PNOM, PRIX, FNOM
FROM   FOURNITURE
WHERE  PNOM = ANY (SELECT PNOM
                   FROM   COMMANDE
                   WHERE  CNOM = 'JEAN')
```

NOTE: Les prédicats IN et = ANY sont utilisés de façon équivalente.

Soit le schéma de relation
COMMANDE(NUM,CNOM,PNOM,QUANTITE)

REQUÊTE: *Client ayant commandé la plus petite quantité de Briques*

SQL:

```
SELECT CNOM
FROM   COMMANDE
WHERE  PNOM = 'Brique' AND
       QUANTITE <= ALL (SELECT QUANTITE
                        FROM   COMMANDE
                        WHERE  PNOM = 'Brique')
```

NOTE: La condition $f\ \theta$ ALL (SELECT F FROM ...) est vraie ssi la comparaison $f\theta v$ est vraie pour toutes les valeurs v du résultat du bloc (SELECT F FROM ...).

Soit le schéma de relations

EMPLOYE(EMPNO,ENOM,DEPNO,SAL)
DEPARTEMENT(DEPTNO,DNOM,LOC)

REQUÊTE: *Départements sans employés*

SQL:

```
SELECT DEPTNO
FROM   DEPARTEMENT
WHERE  DETPNO NOT = ALL (SELECT DISTINCT DEPTNO
                         FROM   EMPLOYE)
```

NOTE: Les prédicats NOT IN et NOT = ALL sont utilisés de façon équivalente.

Requêtes imbriquées plus complexes : EXISTS

Soit le schéma de relations

FOURNISSEUR(FNOM,STATUS,VILLE)
FOURNITURE(PNOM,FNOM,PRIX)

REQUÊTE: *Fournisseurs qui fournissent au moins un produit*

```
SQL :       SELECT FNOM
            FROM   FOURNISSEUR
            WHERE  EXISTS (SELECT *
                           FROM   FOURNITURE
                           WHERE  FNOM = FOURNISSEUR.FNOM)
```

NOTE: La condition EXISTS (SELECT * FROM . . .) est vraie ssi le résultat du bloc (SELECT F FROM . . .) n'est pas vide.

Soit le schéma de relations

FOURNISSEUR(FNOM,STATUS,VILLE)
FOURNITURE(PNOM,FNOM,PRIX)

REQUÊTE: *Fournisseurs qui ne fournissent aucun produit*

SQL:

```
SELECT FNOM
FROM   FOURNISSEUR
WHERE  NOT EXISTS (SELECT *
                   FROM   FOURNITURE
                   WHERE  FNOM = FOURNISSEUR.FNOM)
```

NOTE: La condition NOT EXISTS (SELECT * FROM . . .) est vraie ssi le résultat du bloc (SELECT F FROM . . .) est vide.

Formes Équivalentes de Quantification

Si θ est un des opérateurs de comparaison $<, =, >, \ldots$

- La condition x θ ANY (SELECT Ri.y FROM R1, . . . Rn WHERE p) est équivalente à

EXISTS (SELECT * FROM R1, . . . Rn WHERE p AND x θ Ri.y)

- La condition x θ ALL (SELECT Ri.y FROM R1, . . . Rn WHERE p) est équivalente à

NOT EXISTS (SELECT * FROM R1, . . . Rn WHERE (p) AND NOT (x θ Ri.y))

Soit le schéma de relations

COMMANDE(NUM,CNOM,PNOM,QUANTITE)
FOURNITURE(PNOM,FNOM,PRIX)

REQUÊTE: *Nom, prix et fournisseur des produits commandés par Jean*

```
SELECT PNOM, PRIX, FNOM  FROM   FOURNITURE
WHERE  EXISTS  (SELECT * FROM   COMMANDE
                WHERE CNOM = 'JEAN'
                  AND PNOM = FOURNITURE.PNOM)

SELECT PNOM, PRIX, FNOM FROM   FOURNITURE
WHERE  PNOM = ANY (SELECT PNOM  FROM   COMMANDE
                   WHERE  CNOM = 'JEAN')
```

Soit le schéma de relation **FOURNITURE**(PNOM,FNOM,PRIX)

REQUÊTE: *Fournisseurs qui fournissent au moins un produit avec un coût supérieur au coût des produits fournis par Jean*

SQL:

```
SELECT DISTINCT P1.FNOM
FROM   FOURNITURE P1
WHERE  NOT EXISTS  (SELECT * FROM   FOURNITURE P2
                    WHERE  P2.FNOM = 'JEAN'
                    AND    P1.PRIX <= P2.PRIX)

SELECT DISTINCT FNOM FROM   FOURNITURE
WHERE  PRIX > ALL (SELECT PRIX FROM   FOURNITURE
                   WHERE FNOM = 'JEAN')
```

Division

Soit le schéma de relations

FOURNITURE(FNUM,PNUM,QUANTITE)
PRODUIT(PNUM,PNOM,PRIX)
FOURNISSEUR(FNUM,FNOM,STATUS,VILLE)

REQUÊTE: *Fournisseurs qui fournissent tous les produits*

ALGÈBRE:

$$\text{R1} := \pi_{FNUM,PNUM}(FOURNITURE) \div \pi_{PNUM}(PRODUIT)$$

$$\text{R2} := \pi_{FNOM}(FOURNISSEUR \bowtie R1)$$

SQL:

```
SELECT FNOM
FROM   FOURNISSEUR
WHERE  NOT EXISTS
       (SELECT *
        FROM  PRODUIT
        WHERE NOT EXISTS
              (SELECT *
               FROM  FOURNITURE
               WHERE FOURNITURE.FNUM = FOURNISSEUR.FNUM
               AND   FOURNITURE.PNUM = PRODUIT.PNUM))
```

Fonctions de Calcul

COUNT, SUM, AVG, MIN, MAX

REQUÊTE: *Nombre de Fournisseurs de Paris*

```
SELECT COUNT(*) FROM   FOURNISSEUR
WHERE  VILLE = 'Paris'
```

REQUÊTE: *Nombre de Fournisseurs qui fournissent actuellement des produits*

```
SELECT COUNT(DISTINCT FNOM) FROM   FOURNITURE
```

NOTE: La fonction COUNT(*) compte le nombre des n-uplets du résultat d'une requête sans élimination des dupliqués ni vérification des valeurs nulles. Dans le cas contraire on utilise la clause COUNT(UNIQUE ...).

REQUÊTE: *Quantité totale de Briques commandées*

```
SELECT SUM (QUANTITE)
FROM   COMMANDES
WHERE  PNOM = 'Brique'
```

REQUÊTE: *Coût moyen de Briques fournies*

```
SELECT AVG (PRIX)
FROM   FOURNITURE
WHERE  PNOM = 'Brique'
```

ou

```
SELECT SUM (PRIX)/COUNT(PRI
  FROM FOURNITURE
 WHERE PNOM = 'Brique'
```

REQUÊTE: *Le prix des briques qui sont le plus chères.*

```
SELECT MAX (PRIX)
FROM   FOURNITURE
WHERE  PNOM = 'Briques';
```

REQUÊTE: *Fournisseurs des Briques au coût moyen des Briques*

```
SELECT FNOM
FROM   FOURNITURE
WHERE  PNOM = 'Brique' AND
       PRIX < (SELECT AVG(PRIX)
               FROM   FOURNITURE
               WHERE PNOM = 'Brique')
```

Opérations d’Agrégation

GROUP BY

REQUÊTE: *Nombre de fournisseurs par ville*

```
SELECT VILLE, COUNT(FNOM)
FROM   FOURNISSEUR
GROUP BY VILLE
```

LA BASE ET LE RESULTAT :

VILLE	FNOM
PARIS	TOTO
PARIS	DUPOND
LYON	DURAND
LYON	LUCIEN
LYON	REMI

VILLE	COUNT(FNOM)
PARIS	2
LYON	3

NOTE: La clause GROUP BY permet de préciser les attributs de partitionnement des relations declarées dans la clause FROM. Par exemple

on regroupe les fournisseurs par ville.

REQUÊTE: *Donner pour chaque produit fourni son coût moyen*

```
SELECT PNOM, AVG (PRIX)
FROM   FOURNITURE
GROUP BY PNOM
```

RÉSULTAT:

PNOM	AVG (PRIX)
BRIQUE	10.5
ARDOISE	9.8

NOTE: Les fonctions de calcul appliquées au résultat de régroupement sont directement indiquées dans la clause SELECT. Par exemple le calcul de la moyenne se fait par produit obtenu au résultat après le regroupement.

HAVING

REQUÊTE: *Produits fournis par deux ou plusieurs fournisseurs avec un coût supérieur de 100*

```
SELECT PNOM
FROM   FOURNITURE
WHERE  PRIX > 100
GROUP BY PNOM
HAVING COUNT(*) >= 2
```

AVANT LA CLAUSE HAVING

PNOM	FNOM	PRIX
BRIQUE	TOTO	105
ARDOISE	LUCIEN	110
ARDOISE	DURAND	120

APRÈS LA CLAUSE HAVING

PNOM	FNOM	PRIX
ARDOISE	LUCIEN	110
ARDOISE	DURAND	120

NOTE: La clause HAVING permet d'éliminer des partitionnements, comme la clause WHERE élimine des n-uplets du résultat d'une requête.

REQUÊTE: *Produits fournis et leur coût moyen pour les fournisseurs dont le siège est à Paris seulement si le coût minimum du produit est supérieur à 1000F*

```
SELECT PNOM, AVG(PRIX)
FROM   FOURNITURE, FOURNISSEUR
WHERE  VILLE = 'Paris' AND
       FOURNITURE.FNOM = FOURNISSEUR.FNOM
GROUP BY PNOM
HAVING MIN(PRIX) > 1000
```

ORDER BY

En général, le résultat d'une requête SQL n'est pas trié. Pour trier le résultat par rapport aux valeurs d'un ou de plusieurs attributs, on utilise la clause `ORDER BY` :

```
SELECT VILLE, FNOM, PNOM
  FROM FOURNITURE, FOURNISSEUR
 WHERE FOURNITURE.FNOM = FOURNISSEUR.FNOM
 ORDER BY VILLE, FNOM DESC
```

Le résultat est trié par les villes (ASC) et le noms des fournisseur dans l'ordre inverse (DESC).

Création et Mises à jour avec SQL

Valeurs NULL, par défaut (*)

- **Valeur NULL pour un attribut**: valeur spéciale qui représente une information inconnue (manquante, non fournie)
- attention ce n’est ni zéro, ni chaine vide
- pour forcer un attribut à toujours avoir une valeur, option *default*, exemple: default “manquant”: si l’attribut n’a pas de valeur fournie, on la force à “manquant”

La valeur NULL (*)

1. $A\ \theta\ B$ est `inconnu` (ni vrai, ni faux) si la valeur de A ou/et B est NULL (θ est l'un de $=, <, \leq, >, \geq, \neq$).
2. $A\ op\ B$ est NULL si la valeur de A ou/et B est NULL (op est l'un de $+, -, *, /$).

Trois Valeurs de Vérité (*)

Trois valeurs de vérité: `vrai, faux` et **inconnu**

1. `vrai AND inconnu = inconnu`
2. `faux AND inconnu = faux`
3. `inconnu AND inconnu = inconnu`
4. `vrai OR inconnu = vrai`
5. `faux OR inconnu = inconnu`
6. `inconnu OR inconnu = inconnu`
7. `NOT inconnu = inconnu`

Soit le schéma de relation **FOURNISSEUR**(FNOM,STATUT,VILLE)

REQUÊTE: *Les Fournisseurs de Paris.*

SQL:

```
SELECT FNOM
FROM   FOURNISSEUR
WHERE  VILLE = 'Paris'
```

On ne trouve pas les fournisseurs avec VILLE = NULL !

Soit le schéma de relation **FOURNISSEUR**(FNOM,STATUT,VILLE)

REQUÊTE: *Fournisseurs dont l'adresse est inconnu.*

SQL:

```
SELECT FNOM
FROM   FOURNISSEUR
WHERE  VILLE IS NULL
```

NOTE: Le prédicat IS NULL (ou IS NOT NULL) n'est pas exprimable en algèbre relationnelle.

Exemple (*)

Soit le schéma de relation **EMPLOYE**(EMPNO,ENOM,DEPNO,SAL)

SQL:

```
SELECT E1.ENOM
  FROM EMPLOYE E1, EMPLOYE E2
 WHERE E1.SAL > 20000 OR
       E1.SAL <= 20000
```

Est-ce qu'on trouve les noms de tous les employés s'il y a des employés avec un salaire inconnu ?

Contraintes

Contraintes que l'on peut exprimer lors de la création de tables et que le système peut maintenir:

1. **NOT NULL**: un attribut doit toujours avoir une valeur
2. **PRIMARY KEY**: un (groupe d') attribut(s) constitue(nt) la clé primaire
3. **UNIQUE** un (groupe d') attribut(s) constitue(nt) une clé (secondaire).
4. **FOREIGN KEY (A) REFERENCES R** L'attribut A est la clé primaire de la table R (clé étrangère).
5. **CHECK**: contrainte sur la valeur

Création de Tables

On crée une table avec la commande **CREATE TABLE** :

```
CREATE TABLE Produit(pnom VARCHAR(20),
                     prix INTEGER,
                     PRIMARY KEY (pnom));

CREATE TABLE Fournisseur(fnom VARCHAR(20) PRIMARY KEY,
                         ville VARCHAR(16));

CREATE TABLE Fourniture (pnom VARCHAR(20) NOT NULL,
                         fnom VARCHAR(20) NOT NULL,
                    FOREIGN KEY (pnom) REFERENCES Produit,
                    FOREIGN KEY (fnom) REFERENCES Fournisseur
```

Création de Tables

```
CREATE TABLE ARTISTE (id INTEGER NOT NULL,
                      nom VARCHAR(30) NOT NULL,
                      anneeNaiss INTEGER, default '1900',
                        CHECK (anneeNaiss BETWEEN 1900 AND 2(
                      film VARCHAR (30),
                      PRIMARY KEY (id),
                      UNIQUE(nom),
                      FOREIGN KEY (film) REFERENCES FILM)

CREATE TABLE FILM (idfilm VARCHAR (30) NOT NULL,...)
```

Intégrité référentielle

Le système vérifie les contraintes d'intégrité référentielle. Ses réactions dépendent des options choisies lors de la création des tables, par exemple:

1. si on insère un artiste avec l'attribut film renseigné, le film doit exister dans la table Film
2. si un film est supprimé dans la relation FILM, les nuplets qui réfèrent ce film dans la table ARTISTE ont l'attribut film mis à **NULL** (option *ON DELETE SET NULL*).
3. répercute une maj de l'id faite dans FILM, dans les nuplets qui réfèrent ce film dans ARTISTE (option *ON UPDATE CASCADE*).

Destruction de Tables

On détruit une table avec la commande **DROP TABLE** :

```
DROP TABLE Fourniture;
DROP TABLE Produit;
DROP TABLE Fournisseur;
```

Insertion de n-uplets

On insère dans une table avec la commande **INSERT** dont voici la syntaxe.

INSERT INTO $R(A_1, A_2, \ldots, A_n)$ **VALUES** $(v_1, v_2, \ldots v_n)$

Donc on donne deux listes : celles des attributs (les A_i) de la table et celle des valeurs respectives de chaque attribut (les v_i).

1. Bien entendu, chaque A_i doit être un attribut de R
2. Les attributs non-indiqués restent à **NULL** ou à leur valeur par défaut.
3. On doit toujours indiquer une valeur pour un attribut déclaré **NOT NULL**

Insertion : exemples

Insertion d'une ligne dans *Produit* :

INSERT INTO *Produit (pnom, prix)*
VALUES (*'Ojax'*, *15*)

Insertion de deux fournisseurs :

INSERT INTO *Fournisseur (fnom, ville)*
VALUES (*'BHV'*, *'Paris'*), (*'Casto'*, *'Paris'*)

Il est possible d'insérer plusieurs lignes en utilisant **SELECT**

INSERT INTO *NomsProd (pnom)*
SELECT DISTINCT *pnom* **FROM** *Produit*

Modification

On modifie une table avec la commande **UPDATE** dont voici la syntaxe.

UPDATE R **SET** $A_1 = v_1, A_2 = v_2, \ldots, A_n = v_n$
WHERE *condition*

Contrairement à **INSERT**, **UPDATE** s'applique à un ensemble de lignes.

1. On énumère les attributs que l'on veut modifier.
2. On indique à chaque fois la nouvelle valeur.
3. La clause **WHERE** *condition* permet de spécifier les lignes auxquelles s'applique la mise à jour. Elle est identique au **WHERE** du **SELECT**

Bien entendu, on ne peut pas violer les contraintes sur la table.

Modification : exemples

Mise à jour du prix d'Ojax :

UPDATE *Produit* **SET** *prix=17*
WHERE *pnom = 'Ojax'*

Augmenter les prix de tous les produits fournis par BHV par 20% :

UPDATE *Produit* **SET** *prix = prix*1.2*
WHERE *pnom* in (**SELECT** *pnom*
FROM *Fourniture*
WHERE *fnom = 'BHV'*)

Destruction

On détruit une ou plusieurs lignes dans une table avec la commande **DELETE**

DELETE FROM R
WHERE *condition*

C'est la plus simple des commandes de mise-à-jour puisque elle s'applique à des lignes et pas à des attributs. Comme précédemment, la clause **WHERE** *condition* est indentique au **WHERE** du **SELECT**

Destruction : exemples

Destruction des produits fournit par le BHV :

DELETE FROM *Produit*
WHERE *pnom* in (**SELECT** *pnom*
FROM *Fourniture*
WHERE *fnom = 'BHV*)

Destruction du BHV :

DELETE FROM *Fournisseur*
WHERE *fnom = 'BHV'*

CALCUL RELATIONNEL

Exemple : la base de données CINÉMA

Films

Titre	Metteur en Scene	Acteur
Jules et Jim	F. Truffaut	J. Moreau
Jules et Jim	F. Truffaut	O. Werner
Les Quatre Cent Coups	F. Truffaut	J.P. Leaud
Metropolis	F. Lang	B. Helm
Chimes at Midnight	O. Welles	J. Moreau

Lieu

Cinema	Adresse	No-Telephone
Rex	Bd Poissonniere	42 36 83 93
Champo	R. des Ecoles	43 54 51 60
Cinoche	R. de Conde	46 33 10 82

Pariscope

Cinema	Titre	Heure
Rex	Jules et Jim	18
Rex	Jules et Jim	20
Cinoche	Jules et Jim	20
Champo	Metropolis	18

Syntaxe

Symboles

- Constantes: ’F. Truffaut’, ’Jules et Jim’, 18, . . .
- Variables: x, y, z, ...
- Prédicats: *Movies*,*Location*, . . .
- Comparateurs arithmétiques: $=$, $<$, $>$, $<=$, ...
- Connecteurs logiques : $\vee, \wedge, \neg$
- Quantificateurs : $\exists, \forall$
- Parenthèses : (,)

Formules atomiques (atomes)

- $p(x1, \ldots, xn)$ est un **atome** où p est un symbole de prédicat, et xi, $1 \leq i \leq n$ est soit une *variable*, soit une *constante*.

  Exemples de formules atomiques:

  – Films('Jules et Jim', 'F.Truffaut', 'J. Moreau')

  – Films(x, 'Truffaut', y)

- $x\theta y$ est un **atome** où x et y sont soit des variables soit des constantes et θ est l'un des 6 comparateurs arithmétiques.

  Exemples : $1 < 3$, $x < y$, $x < 8$

- Toutes les occurrences de variables apparaissant dans une formule atomique sont dites **LIBRES**

Formules bien formées

- si $F(x1, \ldots, xn)$ est une **formule** avec xi parmi ses variables libres, alors $(\exists xi)F$ et $(\forall xi)F$ sont des **formules**. Toutes les occurrences de xi dans F sont alors dites *LIÉES*. Exemples:
  - $F1 : (\exists x)(x < 3)$
  - $F2 : (\forall x)(x < 3)$
  - $F3(y) : (\exists x)(x < y)$
- si $F1$ et $F2$ sont des **formules**, alors $F1 \vee F2$, $F1 \wedge F2$ et $\neg F1$ sont des **formules** (les occurrences de variables de ces nouvelles formules sont libres ou liées si elles le sont dans $F1$, $F2$). Exemples:
  - $F4(x) : (x <= 3) \vee (x > 5)$
  - $F5 : (\exists x)(x < 3 \wedge x > 5)$
- si F est une **formule**, alors (F) est une **formule**;

Formes Abrégés

- Formes abrégées de quantificateurs:
  1. $\exists x1, x2, \ldots, xn$ est une abréviation de $\exists x1 \exists x2 \ldots \exists xn$,
  2. $\forall x1, x2, \ldots, xn$ est une abréviation de $\forall x1 \forall x2 \ldots \forall xn$.
- Ordre de précédence (priorité) entre les connecteurs et quantificateurs (du plus au moins prioritaire):
  1. $\neg, \forall, \exists$,
  2. $\wedge$,
  3. $\vee$.

  Par exemple, $(\forall x)\neg p(x, y) \vee q(y) \wedge r(z)$ est compris comme

$$((\forall x)(\neg p(x, y))) \vee (q(y) \wedge r(z)).$$

Sémantique

Interprétation : Atomes sans variables libres

- L'atome $p(c1, \ldots, cn)$ où p est un symbole de prédicat et où tous les ci, $1 \leq i \leq n$, sont des constantes, est *vrai* ssi $[c1, \ldots, cn]$ est un n-uplet de la relation p *dans la base de données*.

  Exemple : $Film('Jules\ et\ Jim','Truffaut','Moreau')$ est vrai ssi $['Jules\ et\ Jim','Truffaut','Moreau']$ est un n-uplet dans la table $Film$.

- L'interprétation de $c\theta c'$ est "naturelle". Par exemple $1 < 5$ est vrai.

Interprétation: Quantification Existentielle

- $(\exists x)F(x)$ est vraie s'il existe une *instanciation de x* – on remplace x par une constante c – telle que $F(c)$ devient vraie (on remplace toutes les occurrences de x dans F par la constante c).

  Par exemple $(\exists x)Films(x,'Truffaut','Moreau')$ est vraie s'il existe un film f avec J. Moreau et dirigé par Truffaut (dans la base de données).

Interprétation: Connecteurs logiques

- L'interprétation de $F1 \vee F2$, $\neg F$,... est habituelle.

  Par exemple $F1 \vee F2$ est vraie si $F1$ est vraie *ou* $F2$ est vraie.

Remarque: La disjonction et la quantification peuvent être exprimées en utilisant la conjonction et la négation.

Algèbre de Bool et Théorème de Morgan:

1. $F \wedge \neg F$ est toujours faux, ...
2. $F1 \vee (F2 \wedge F3) \equiv (F1 \vee F2) \wedge (F1 \vee F3)$
3. $F1 \vee F2 \equiv \neg(\neg F1 \wedge \neg F2)$
4. $(\forall x)F \equiv (\neg\exists x)\neg F$

Requêtes

Requête : Une requêtes est une expression $\{x1, x2, \ldots, xn \mid F\}$ où $F(x1, x2, \ldots, xn)$ est une formule avec les variables libres $x1, x2, \ldots, xn$.

Exemples de Requêtes :

- $\{x \mid x < 3\}$
- $\{x \mid Film('Jules\ et\ Jim', 'Truffaut', x)\}$
- $\{x \mid (\exists y) Film(y, 'Truffaut', x)\}$

Interprétation : le **résultat** de cette requête est l'ensemble des n-uplets $[a1, a2, \ldots, an]$ tels que $F(a1, a2, \ldots, an)$ est vraie.

- $\{x \mid Film('Jules\ et\ Jim', 'Truffaut', x)\}$ retourne tous les acteurs du film 'Jules et Jim' de Truffaut.
- $\{x \mid (\exists y) Film(y, 'Truffaut', x)\}$ retourne tous les acteurs qui ont tourné avec Truffaut.

Algèbre relationnelle et Calcul relationnel

Théorême 1 : Toute requête exprimable dans l'algèbre relationnelle est exprimable dans le calcul relationnel.

Formule saine : Formule dont le résultat est fini (le nb de n-uplets qui satisfont la formule est fini).

Exemple de formule non saine : $\{x, y | \neg p(x, y)\}$. Le resultat contient tous les nuplets qui ne sont pas dans la relation p.

Théorême 2 : Toute requête du calcul relationnel sain est exprimable en algèbre relationnelle.

Conclusion : L'algèbre relationnelle et le calcul relationnel sain ont *le même pouvoir d'expression*.

Calcul relationnel domaine/n-uplet

Le calcul relationnel précédent est appelé **calcul relationnel domaine**.

Pour obtenir le **calcul relationnel n-uplet**, on remplace :

- $x1, x2, \ldots, xn$ (où xi est une variable domaine) par la variable n-uplet t.
- L'attribut A de rang i (xi dans le calcul domaine) est noté $t.A$ (voir exemples ci-dessous).

Theorême 3 : Le calcul relationnel n-uplet sain a le même pouvoir d'expression que l'algèbre relationnelle.

Quelques Requêtes

1. Qui dirige Metropolis?
2. Adresse et numéro de téléphone du Studio?
3. Salles où on peut voir un film de Truffaut?
4. Adresses des cinémas montrant un film de Truffaut?
5. Quels films de Truffaut ne passent pas en ce moment?

Requête 1: Qui a dirigé le film Metropolis ?

SQL

```
select Metteur-en-Scene
  from Films
 where Titre = 'Metropolis';
```

Calcul relationnel n-uplet

$$\{x.Metteur\_en\_Scene \mid Films(x) \wedge x.Titre =' Metropolis'\}$$

Calcul relationnel domaine

$$\{d \mid (\exists x) Films('Metropolis', d, x)\}$$

Requête 2: Adresse et numéro de téléphone du Studio ?

Calcul relationnel n-uplet

$$\{x.Adresse, x.Numero\_Tel \mid Lieu(x) \wedge x.Cinema =' Studio'\}$$

Calcul relationnel domaine

$$\{ad, ph \mid Lieu('Studio', ad, ph)\}$$

Requête 3: Salles où on peut voir un film de Truffaut ?

Calcul relationnel n-uplet

$$\{x.Cinema \mid (\exists y)(Pariscope(x) \wedge Films(y) \wedge x.Titre = y.Titre \wedge y.Metteur\_en\_Scene =' Truffaut')\}$$

Calcul relationnel domaine

$$\{s \mid (\exists hor, act, titre)(Pariscope(s, titre, hor) \wedge Films(titre,' Truffaut', act))\}$$

Requête 4: Adresses des cinémas montrant un Truffaut ?

SQL

```
select L.Adresse
  from Films F, Lieu L, Pariscope P
 where F.Metteur-en-Scene = 'Truffaut'
   and P.Titre = F.Titre
   and L.Cinema = P.Cinema;
```

Calcul relationnel n-uplet

$$\{z.Adresse \mid (\exists x, y)(Films(x) \wedge Pariscope(y) \wedge Lieu(z) \wedge x.Metteur\_en\_Scene =' Truffaut' \wedge x.Titre = y.Titre \wedge y.Cinema = z.Cinema)\}$$

Requête 4: Adresses des cinémas montrant un Truffaut ?

Calcul relationnel domaine

$$\{a \mid \quad (\exists t, cine, s, teleph, act)(Films(t, \text{“}Truffaut\text{”}, act) \wedge$$
$$Pariscope(cine, t, s) \wedge Lieu(cine, a, teleph))\}$$

Requête 5: Quels films de Truffaut ne passent pas au cinéma?

SQL

```
select Titre
  from Films F
 where F.Metteur en Scene = 'Truffaut'
   and not exists ( select *
                      from Pariscope P
                     where  P.Titre = F.Titre )
```

Calcul relationnel n-uplet

$$\{x.Titre \quad | \quad (Films(x) \wedge x.Metteur\_en\_Scene =' Truffaut') \wedge (\neg \exists y)(Pariscope(y) \wedge y.Titre = x.Titre)\}$$

Requête 5: Quels films de Truffaut ne passent pas au cinéma?

Calcul relationnel domaine

$$\{x \mid (\exists y) Films(x, \text{“}Truffaut\text{”}, y) \wedge (\neg \exists z, w) Pariscope(z, x, w)\}$$

ou

$$\{x \mid (\exists y) Films(x, \text{“}Truffaut\text{”}, y) \wedge (\forall z, w) \neg Pariscope(z, x, w)\}$$

ORGANISATION PHYSIQUE DES DONNEES

Organisation des données en mémoire secondaire

1. Le disque est divisé en **blocs physiques** (ou **pages**) de tailles égales.
2. Accès à un bloc par son adresse (par exemple le numéro de cylindre + le numéro de secteur + le numéro de face).
3. Le bloc est l'unité d'échange entre la mémoire secondaire et la mémoire principale

Les fichiers

Les données sont stockées dans des **fichiers** :

- Un fichier occupe un ou plusieurs blocs sur un disque.
- L'accès aux fichiers est géré par un logiciel spécifique : le Système de Gestion de Fichiers (SGF).
- Un fichier est caractérisé par son nom.
- Enfin un fichier est un ensemble **d'articles**.

Les articles

Un article est une séquence de **champs**.

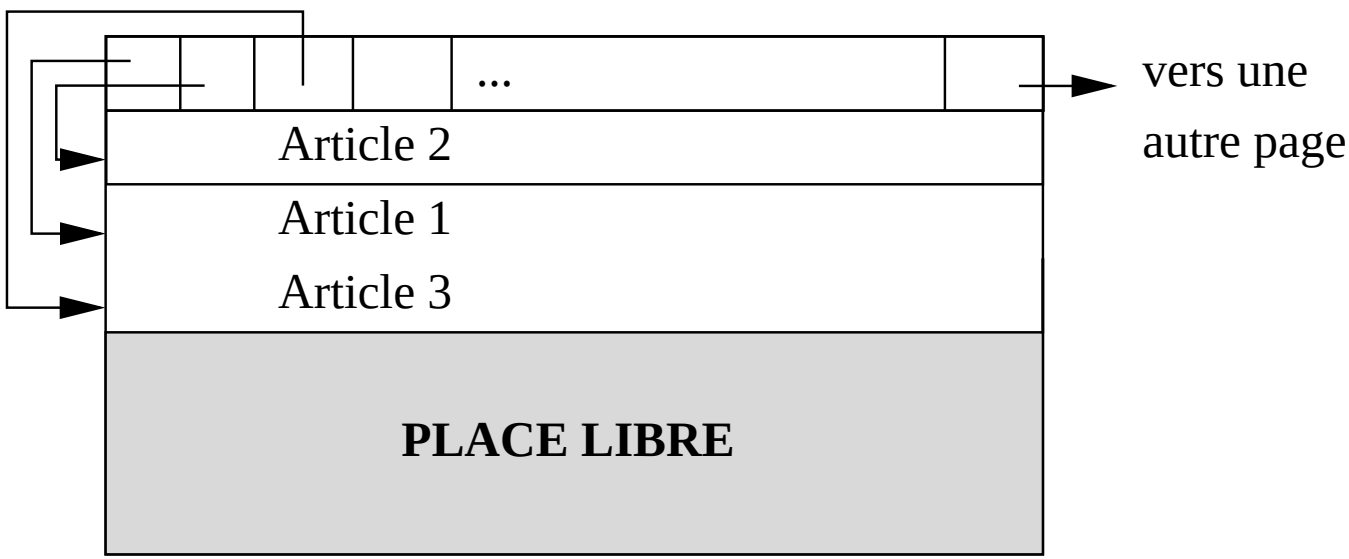
1. **Articles en format fixe.**
   (a) La taille de chaque champ est fixée
   (b) Taille et nom des champs dans le **descripteur** de fichier.
2. **Articles en format variable**.
   (a) La taille de chaque champ est variable.
   (b) L'en tête du champ donne la taille réelle

Articles et pages

L’adresse d’un article est constituée de

1. L’adresse de la page dans laquelle il se trouve.
2. Un entier : indice d’une table
   placée en début de page qui contient l’adresse
   réelle de l’article dans la page.

Structure interne d’une page



Opérations sur les fichiers

1. **Insérer un article**.
2. **Modifier un article**
3. **Détruire un article**
4. **Rechercher un ou plusieurs article(s)**
   - Par adresse
   - Par valeur d'un ou plusieurs champs.

Hypothèse: Le **coût** d'une opération est surtout fonction du **nombre d'E/S** (nb de pages)

Organisation de fichiers

L’organisation d’un fichier est caractérisée par le mode de répartition des articles dans les pages

Il existe trois organisations principales :

1. Fichiers séquentiels
2. Fichiers indexés (séquentiels indexés et arbres-B)
3. Hachage

Exemple de référence

Organisation d’un fichier contenant les articles suivants :

Vertigo 1958

Brazil 1984

Twin Peaks 1990

Underground 1995

Easy Rider 1969

Psychose 1960

Greystoke 1984

Shining 1980

Annie Hall 1977

Jurassic Park 1992

Metropolis 1926

Manhattan 1979

Reservoir Dogs 1992

Impitoyable 1992

Casablanca 1942

Smoke 1995

Organisation séquentielle

- **Insertion** : les articles sont stockés séquentiellement dans les pages au fur et à mesure de leur création.
- **Recherche** : le fichier est parcouru séquentiellement.
- **Destruction** : recherche, puis destruction (par marquage d'un bit par exemple).
- **Modification** : recherche, puis réécriture.

Coût des opérations

Nombre moyen de lectures/écritures sur disque,

Fichier de n pages.

- **Recherche** : $\frac{n}{2}$. On parcourt en moyenne la moitié du fichier.
- **Insertion** : $1 + 1 = 2$ ($n + 1$ si on vérifie que l'article n'existe pas avant d' écrire).
- **Destruction et mises-à-jour** : $\frac{n}{2} + 1$.

$\Rightarrow$ organisation utilisée pour les fichiers de petite taille.

Fichiers séquentiels triés

Lorsque la recherche est faite par valeur d'un champ, celui-ci est appelé **clé d'accès**.

Si le fichier est **trié** sur sa clé d'accès, on peut effectuer une recherche par **dichotomie** :

1. On lit la page qui est "au milieu" du fichier.
2. Selon la valeur de la clé du premier enregistrement de cette page, on sait si l'article cherché est "avant" ou "après".
3. On recommence avec le demi-fichier où se trouve l'article recherché.

Coût de l'opération : $\log_2(n)$.

Ajout d'un index

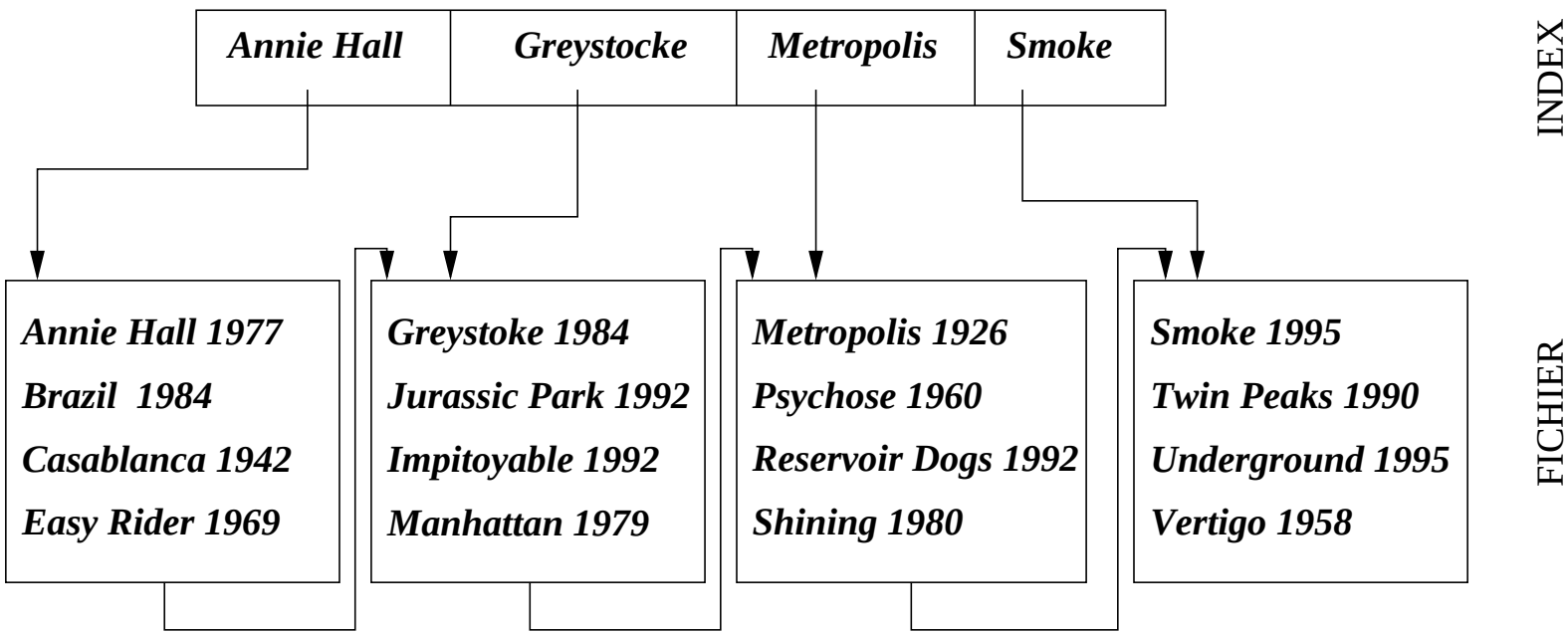
L'opération de recherche peut encore être améliorée en utilisant un **index** sur un fichier **trié**.

Un index est un second fichier possédant les caractéristiques suivantes :

1. Les articles sont des couples (Valeur de clé, Adresse de page)
2. Une occurence (v, b) dans un index signifie que le premier article dans la page b du fichier trié a pour valeur de clé v.

L'index est lui-même **trié** sur la valeur de clé.

Exemple


Annie Hall
Greystocke
Metropolis
Smoke
INDEX
Annie Hall 1977
Brazil 1984
Casablanca 1942
Easy Rider 1969
Greystoke 1984
Jurassic Park 1992
Impitoyable 1992
Manhattan 1979
Metropolis 1926
Psychose 1960
Reservoir Dogs 1992
Shining 1980
Smoke 1995
Twin Peaks 1990
Underground 1995
Vertigo 1958
FICHIER

Recherche avec un index

Chercher l'article dont la valeur de clé est v_1.

1. On recherche dans l'index (séquentiellement ou - mieux - par dichotomie) la plus grande valeur v_2 telle que $v_2 < v_1$.
2. On lit la page désignée par l'adresse associée à v_2 dans l'index.
3. On cherche séquentiellement les articles de clé v_1 dans cette page.

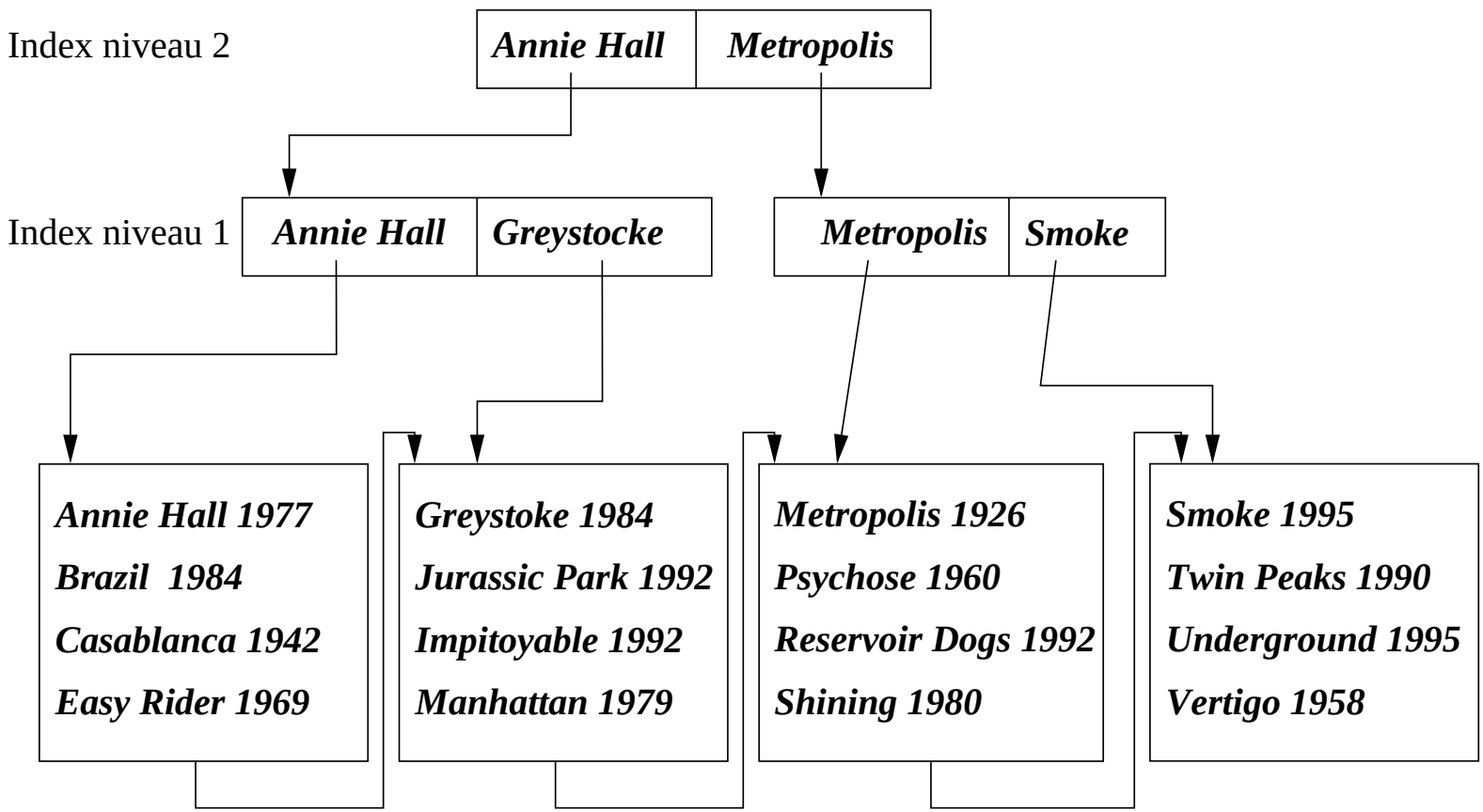
Coût d'une recherche avec ou sans index

Soit un fichier F contenant 1000 pages. On suppose qu'une page d'index contient 100 entrées, et que l'index occupe donc 10 pages.

- F non trié et non indexé. Recherche séquentielle : **500 pages**.
- F trié et non indexé. Recherche dichotomique : $\log_2(1000)$=**10 pages**
- F trié et indexé. Recherche dichotomique sur l'index, puis lecture d'une page : $\log_2(10) + 1$= **5 pages**

Séquentiel indexé

1. Un index est un fichier : on peut lui-même l'indexer :
2. On obtient un index à plusieurs niveaux sur la clé.
3. C'est un **arbre** dont les feuilles constituent
   le fichier et les noeuds internes l'index.


Index niveau 2
Annie Hall
Metropolis
Index niveau 1
Annie Hall
Greystocke
Metropolis
Smoke
Annie Hall 1977
Brazil 1984
Casablanca 1942
Easy Rider 1969
Greystoke 1984
Jurassic Park 1992
Impitoyable 1992
Manhattan 1979
Metropolis 1926
Psychose 1960
Reservoir Dogs 1992
Shining 1980
Smoke 1995
Twin Peaks 1990
Underground 1995
Vertigo 1958

Index dense et index non dense

L'index ci-dessus est **non dense** : **une**

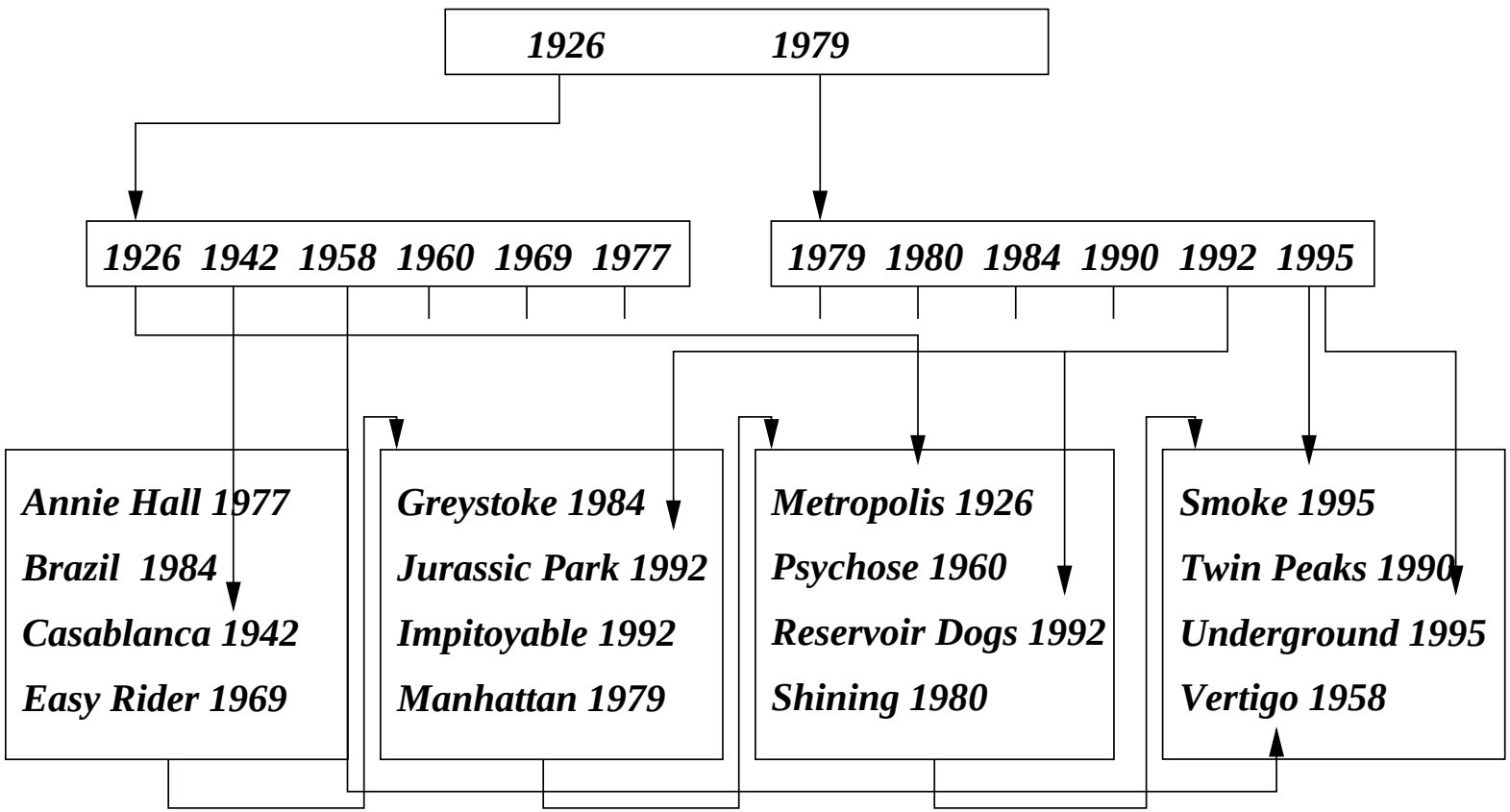
seule valeur de clé dans l'index pour l'ensemble des articles du fichier indexé F situés dans une même page.

Un index est **dense** ssi il existe une valeur de clé dans l'index pour chaque article dans le fichier F.

Remarques :

1. On ne peut créer un index non-dense que sur un fichier trié (et un seul index non-dense par fichier).
2. Un index non-dense est beaucoup moins volumineux qu'un index dense.

Exemple d'index dense



Autres opérations : insertion

Etant donné un article A de clé v_1, on effectue d'abord une recherche pour savoir dans quelle page p il doit être placé. Deux cas de figure :

1. Il y une place libre dans p. Dans ce cas on réorganise le contenu de p pour placer A à la bonne place.
2. Il n'y a plus de place dans p. On crée une **page de débordement**.

Exercice: montrer l'index non dense précédent après l'insertion de [Insomnia, 2002], [L'homme sans passé, 2002], [Le Destin fabuleux..., 2001], [L'auberge espagnole, 2002], [Jonathan aura 25 ans en l'an 2000, 1975].

Autres opérations : destructions et mises-à-jour

Relativement facile en général :

1. On recherche l'article.
2. On applique l'opération.

$\Rightarrow$ on peut avoir à réorganiser le fichier et/ou

l'index, ce qui peut être couteux.

Inconvénients du séquentiel indexé

Organisation bien adaptée aux fichiers qui

évoluent peu. En cas de grossissement :

1. Une page est trop pleine $\rightarrow$ on crée une page de débordement.
2. On peut aboutir à des chaînes de débordement importantes pour certaines pages.
3. Le temps de réponse peut se dégrader et dépend de l'article recherché

$\Rightarrow$ on a besoin d'une structure permettant une

réorganisation dynamique sans dégradation de performances.

Arbres-B

Un arbre-B (pour $balanced\ tree$ ou **arbre équilibré**)

est une structure arborescente dans laquelle tous les chemins de la racine aux feuilles ont même longueur.

Si le fichier grossit : la hiérarchie grossit **par le haut**.

L'arbre-B est utilisé dans **tous** les SGBD relationnels

(avec des variantes).

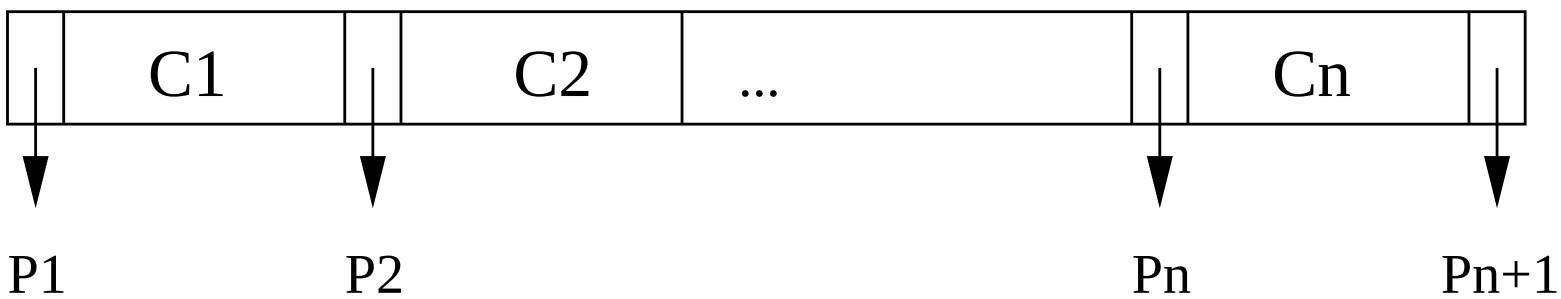
Arbre-B : définition

Un arbre-B **d'ordre k** est un arbre équilibré tel que :

1. Chaque noeud est une page contenant au moins k et au plus $2k$ articles, $k \in N$.
2. Les articles dans un noeud sont triés sur la clé.
3. Chaque "père" est un index pour l'ensemble de ses fils.
4. Chaque noeud contenant n articles a $n + 1$ fils.
5. La racine a 0 ou au moins deux fils.

On décrit la variante appelée **arbre B+**

Structure d'un noeud dans un arbre-B d'ordre k

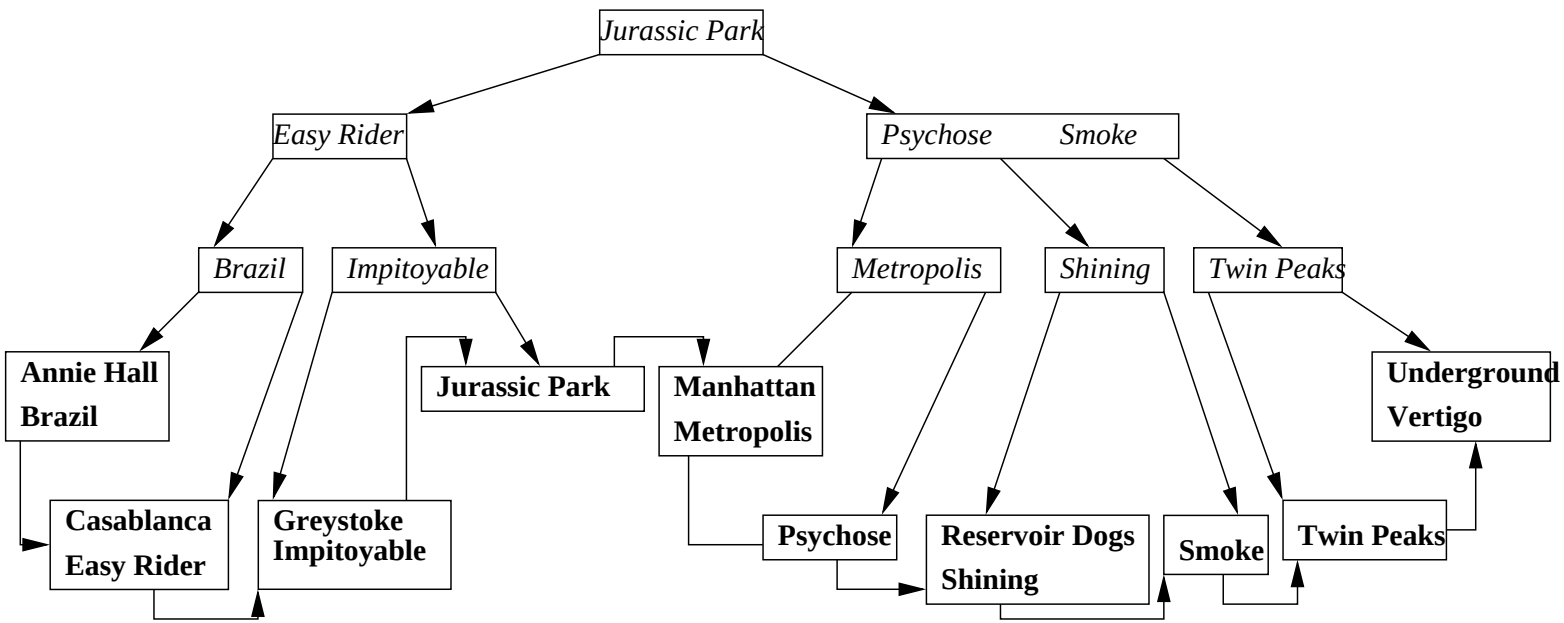


Les C_i sont les clés des articles. Les

P_i sont les pointeurs vers les noeuds fils dans l'index. NB :

$k \leq n \leq 2k$.

Exemple d’ arbre-B



L’arbre B

- Les feuilles de l’arbre contiennent des couples [valeur de clé, adresse d’article dans le fichier indexé par l’arbre]
- Les feuilles sont chaînées entre elles

Recherche dans un arbre-B

Rechercher les articles de clé C.

A partir de la racine, appliquer récursivement

l'algorithme suivant :

Soit $C_1, \ldots C_n$ les valeurs de clés de la page courante.

1. Si $C \leq C_1$ (ou $C > C_n$), on continue la recherche avec le noeud référencé par P_1 (ou P_{n+1}).
2. Sinon, il existe $i \in [1, k[$ tel que $C_i < C \leq C_{i+1}$, on continue avec la page référencée par le pointeur P_{i+1}.

Insertion dans un arbre-B d'ordre k

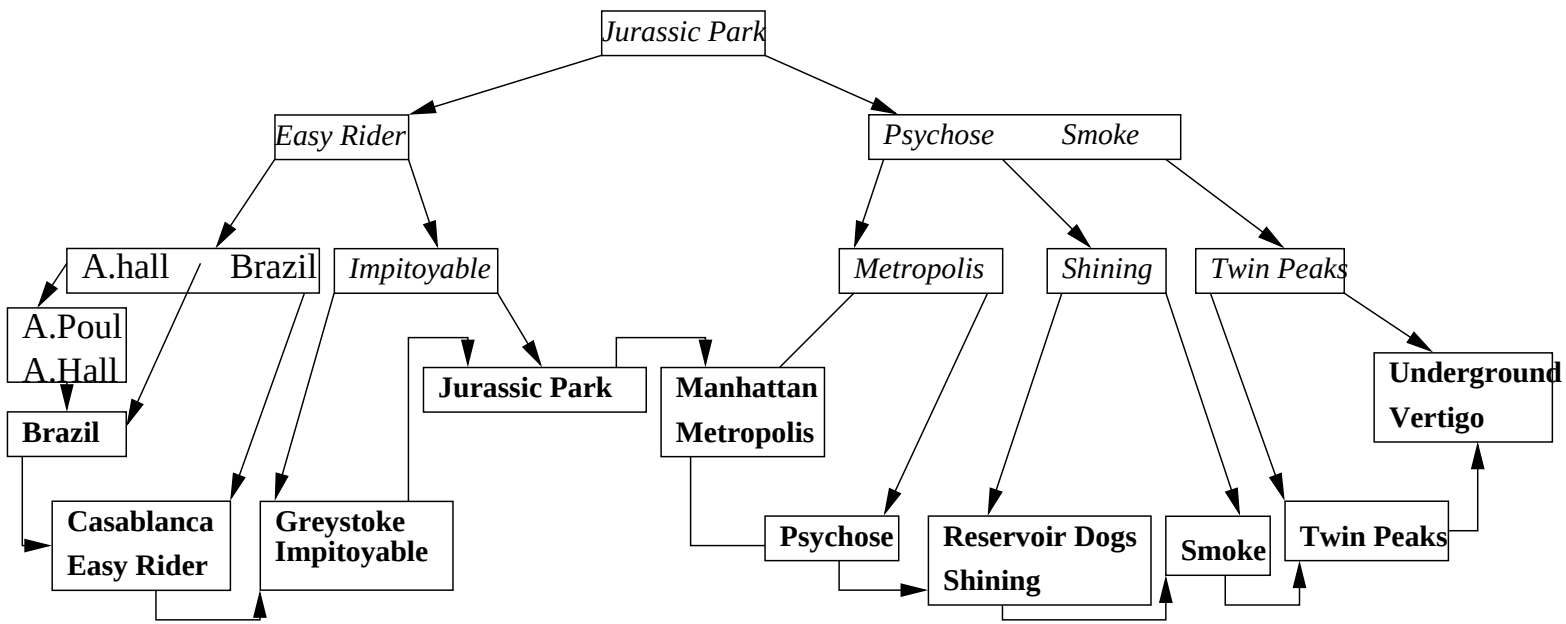
On recherche la feuille de l'arbre où le couple (clé, adresse) doit prendre place:

1. On l'y insère. Si la page p déborde (elle contient $2k+1$ éléments) : on alloue une nouvelle page p'.
2. On place les $k+1$ premiers articles (ordonnés selon la clé) dans p et les k derniers dans p'.
3. On insère le $k+1^e$ article dans le père de p. Son pointeur gauche référence p, et son pointeur droit référence p'.
4. Si le père déborde à son tour, on continue comme en 1 (sauf qu'en 2 on ne place que les **k** premiers articles dans p et non les **k+1**).

Exercice: insérer Amélie Poulain et Citizen Kane

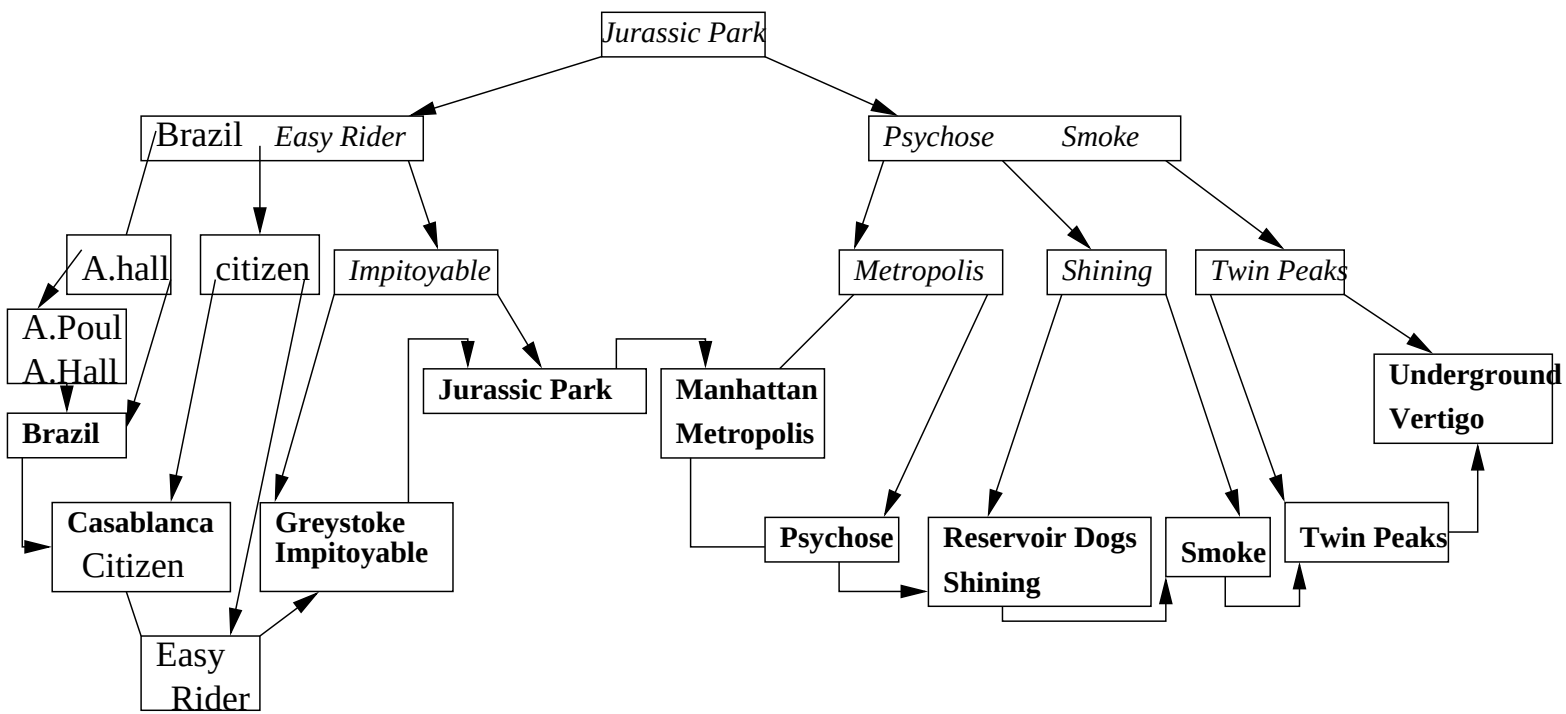
Après insertion d'Amélie Poulain

En cas d'éclatement d'une feuille (page pleine 2n entrées): l'ancienne feuille contient les n+1 premières entrées, la nouvelle feuille les n dernières, la n+1e (celle du milieu) est insérée dans le père: la n+1e entrée se retrouve à la fois dans la feuille et dans le père



Après insertion de Citizen Kane

En cas d'éclatement d'un noeud interne (page pleine 2n entrées): l'ancien noeud contient les n premières entrées, le nouveau noeud les n dernières, la n+1e (celle du milieu) est insérée dans le père.



Quelques mesures pour l'arbre-B

Hauteur h d'un arbre-B d'ordre k contenant n articles :

$$\log_{2k+1}(n+1) \leq h \leq \log_{k+1}(\tfrac{n+1}{2})$$

Exemple pour $k = 100$:

1. si $h = 3$, $n \leq 8 \times 10^6$
2. si $h = 4$, $n \leq 1,6 \times 10^9$

Les opérations d'accès coûtent au maximum h E/S.

Hachage

Accès direct à la page contenant l'article recherché :

1. On estime le nombre N de pages qu'il faut allouer au fichier.
2. **fonction de hachage** H : à toute valeur de la clé de domaine V associe un nombre entre 0 et $N-1$.

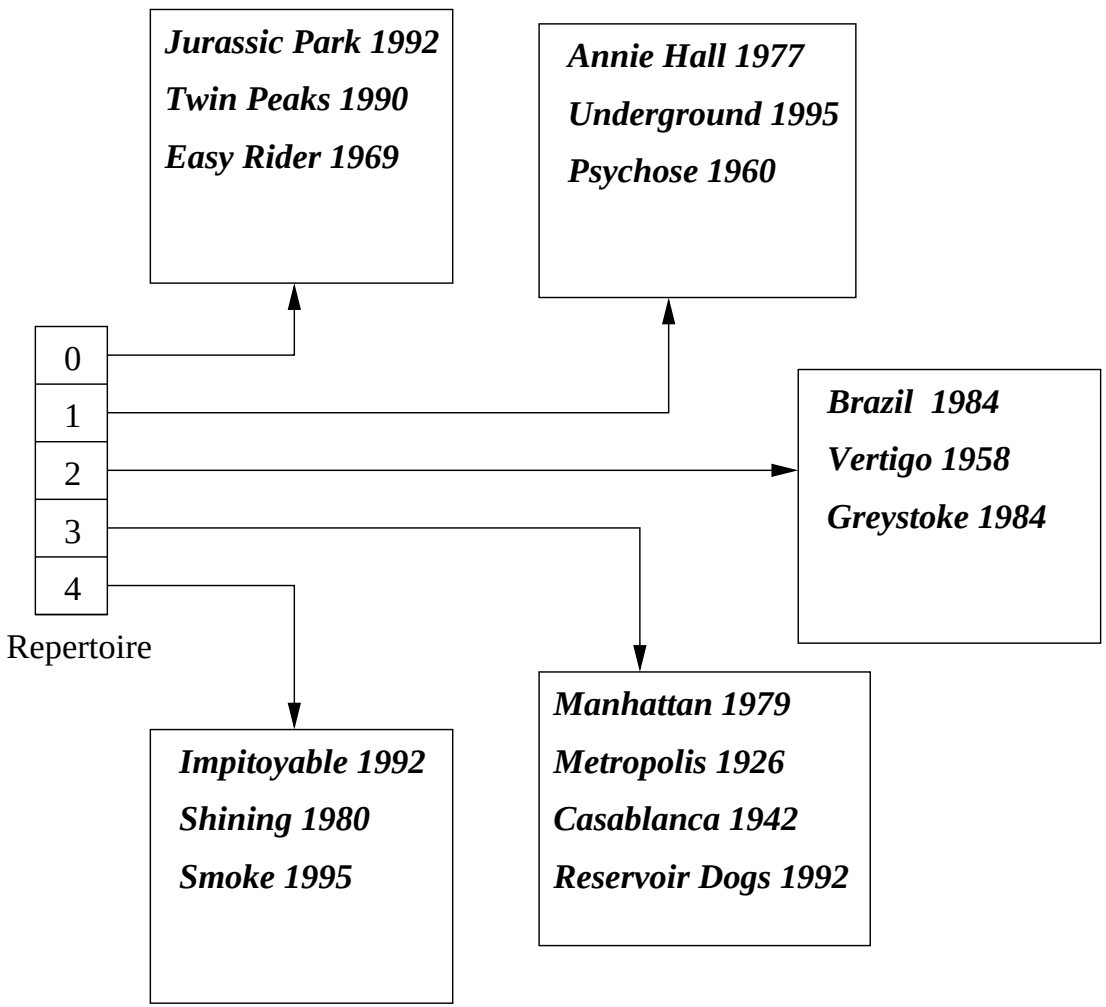
$$H : V \rightarrow \{0, 1, \ldots, N-1\}$$

3. On range dans la page de numéro i tous les articles dont la clé c est telle que $H(c) = i$.

Exemple : hachage sur le fichier $Films$

On suppose qu'une page contient 4 articles :

1. On alloue 5 pages au fichier.
2. On utilise une fonction de hachage H définie comme suit :
   (a) clé : nom d'un film, on ne s'intéresse qu'à l'initiale de ce nom.
   (b) On numérote les lettres de l'alphabet de 1 à 26 :
   $No('a') = 1, No('m') = 13$, etc.
   (c) Si l est une lettre de l'alphabet, $H(l) = MODULO(No(l), 5)$.


Jurassic Park 1992
Twin Peaks 1990
Easy Rider 1969
Annie Hall 1977
Underground 1995
Psychose 1960
0
1
2
3
4
Repertoire
Brazil 1984
Vertigo 1958
Greystoke 1984
Impitoyable 1992
Shining 1980
Smoke 1995
Manhattan 1979
Metropolis 1926
Casablanca 1942
Reservoir Dogs 1992

Remarques

1. Le nombre $H(c) = i$ n'est pas une adresse de page, mais l'indice d'1 table ou "répertoire" R. $R(i)$ contient l'adresse de la page associée à i
2. si ce répertoire ne tient pas en mémoire centrale, la recherche coûte plus cher.
3. Une propriété essentielle de H est que la distribution des valeurs obtenues soit uniforme dans $\{0, \ldots N-1\}$
4. Quand on alloue un nombre N de pages, il est préférable de prévoir un remplissage partiel(non uniformité, grossissement du fichier). On a choisi 5 pages alors que 4 (16 articles / 4) auraient suffi.

Hachage : recherche

Etant donné une valeur de clé v :

1. On calcule $i = H(v)$.
2. On consulte dans la case i du répertoire l'adresse de la page p.
3. On lit la page p et on y recherche l'article.

$\Rightarrow$ **donc une recherche ne coûte qu'une seule lecture**.

Hachage : insertion

Recherche par $H(c)$ la page p où placer A et l'y insèrer.

Si la page p est pleine, il faut :

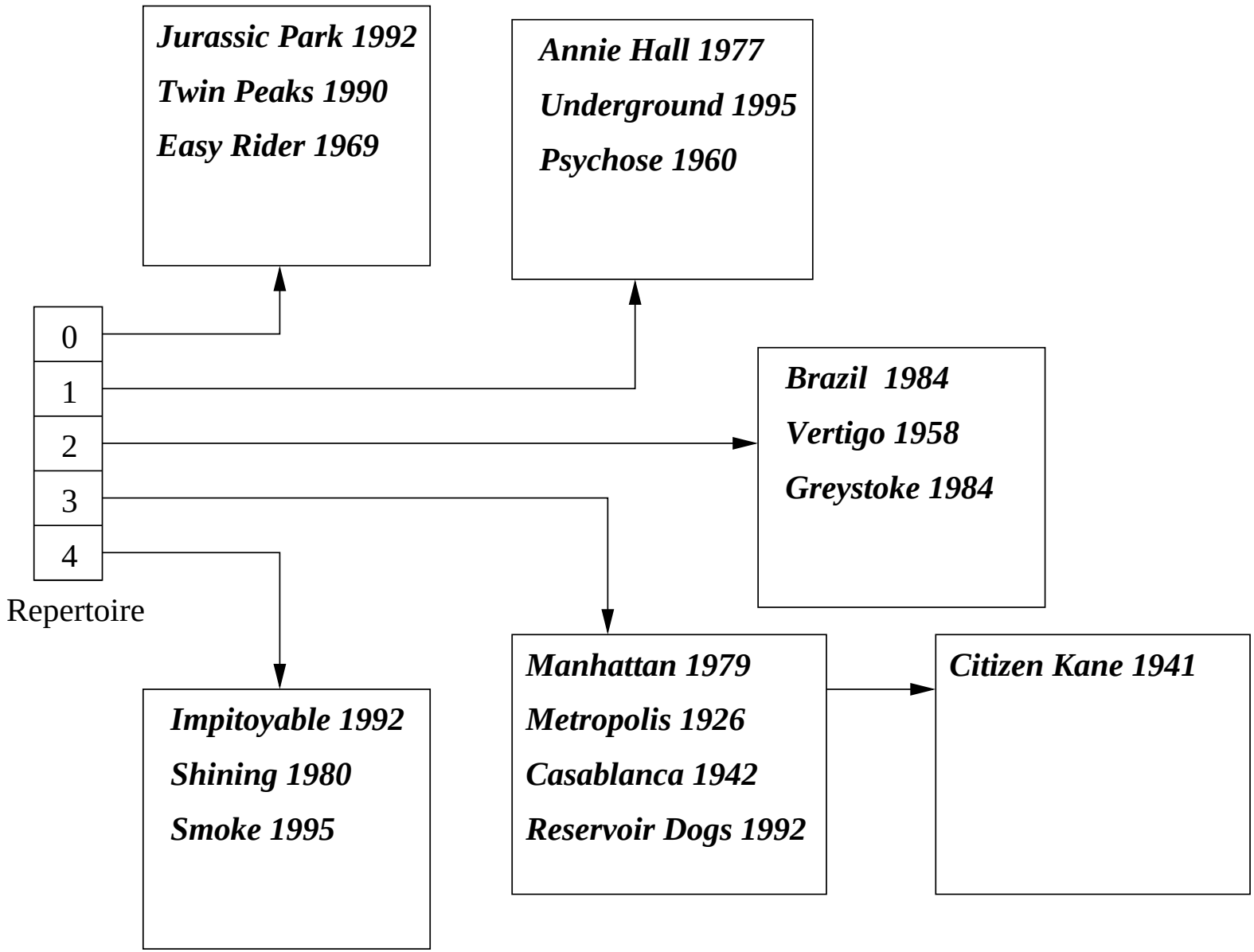
1. Allouer une nouvelle page p' (de débordement).
2. Chaîner p' à p.
3. Insérer A dans p'.

$\Rightarrow$ lors d'une recherche, il faut donc en fait parcourir

la liste des pages chaînées correspondant à une valeur de

$H(v)$.

Moins la réparttion est uniforme, plus il y aura de débordements


Jurassic Park 1992
Twin Peaks 1990
Easy Rider 1969
Annie Hall 1977
Underground 1995
Psychose 1960
0
1
2
3
4
Repertoire
Brazil 1984
Vertigo 1958
Greystoke 1984
Impitoyable 1992
Shining 1980
Smoke 1995
Manhattan 1979
Metropolis 1926
Casablanca 1942
Reservoir Dogs 1992
Citizen Kane 1941

Hachage : avantages et inconvénients

Intérêt du hachage :

1. **Très rapide**. Une seule E/S dans le meilleur des cas pour une recherche (répertoire résidant en mémoire)
2. Le hachage, contrairement à un index, **n'occupe aucune place disque**.

En revanche :

1. Il faut penser à réorganiser les fichiers qui évoluent beaucoup.
2. **Les recherches par intervalle sont impossibles.**

Comparatif

Organisation	**Coût**	**Avantages**	**Inconvénients**
Sequentiel	$\frac{n}{2}$	Simple	Très coûteux !
Indexé	$\log_2(n)$	Efficace Intervalles	Peu évolutive
Arbre-B	$\log_k(n)$	Efficace Intervalles	Traversée
Hachage	1+	Le plus efficace	Intervalles impossibles

OPTIMISATION

Pourquoi l'optimisation ?

Les langages de requêtes de haut niveau comme SQL sont **déclaratifs**.
L'utilisateur :

1. indique ce qu'il veut obtenir.
2. n'indique pas **comment** l'obtenir.

Donc le système doit faire le reste :

1. Déterminer le (ou les) chemin(s) d'accès aux données,
   les stratégies d'évaluation de la requête
2. **Choisir la meilleure**. Ou une des meilleures . . .

L'optimisation sur un exemple

Considérons le schéma :

$CINEMA(Cinéma, Adresse, Gérant)$
$SALLE(Cinéma, NoSalle, Capacité)$

avec les hypothèses :

1. Il y a 300 n-uplets dans CINEMA, occupant 30 pages (10 cinémas/page).
2. Il y a 1200 n-uplets dans SALLE, occupant 120 pages(10 salles/page).
3. La mémoire centrale (buffer) ne contient qu'une seule page par relation.

Expression d'une requête

On considère la requête : *Cinémas ayant des salles de plus de 150 places*

En SQL, cette requête s'exprime de la manière suivante :

```
SELECT CINEMA.*
FROM   CINEMA, SALLE
WHERE  capacité > 150
AND    CINEMA.cinéma = SALLE.cinéma
```

En algèbre relationnelle

Traduit en algèbre, on a plusieurs possibilités. En voici deux :

1. $\pi_{CINEMA.*}(\sigma_{Capacité>150}(CINEMA \bowtie SALLE))$
2. $\pi_{CINEMA.*}(CINEMA \bowtie \sigma_{Capacité>150}(SALLE))$

Soit une jointure suivie d'une sélection, ou l'inverse.

Evaluation des coûts

On suppose qu'il n'y a que 5 % de salles de plus de 150 places et que les résultats intérmédiaires d'une opération et le résultat final sont écrits sur disque (10 n-uplets par page).

1. Jointure (naïve) : on lit 3 600 pages (120x30); on écrit le résultat intérmédiaire (120 pages); on le relit et comme on projète sur tous les attributs de Cinéma, on obtient 5 % de 120 pages, soit 6 pages.
   Nombre d'E/S : 3 600E + 120x2E/S + 6S = 3 846.

2. Sélection : on lit 120 pages (salles) et on obtient (écrit) 6 pages.
   Jointure : on lit 180 pages (6x30) et on obtient 6 pages.
   Nombre d'E/S : 120E + 6S + 180E + 6S = 312.

$\Rightarrow$ la deuxième stratégie est de loin la meilleure !

Optimisation de requêtes : premières conclusions

1. Il faut **traduire** une requête exprimée avec un langage déclaratif en une suite d'opérations (typiquement les opérateurs de l'algèbre relationnelle).
2. En fonction (i) des coûts de chaque opération (ii) des caractéristiques de la base, (iii) des algorithmes utilisés, on cherche à estimer la meilleure stratégie.
3. On obtient le **plan d'exécution** de la requête. Il n'y a plus qu'à le traiter au niveau physique.

Les paramètres de l’optimisation

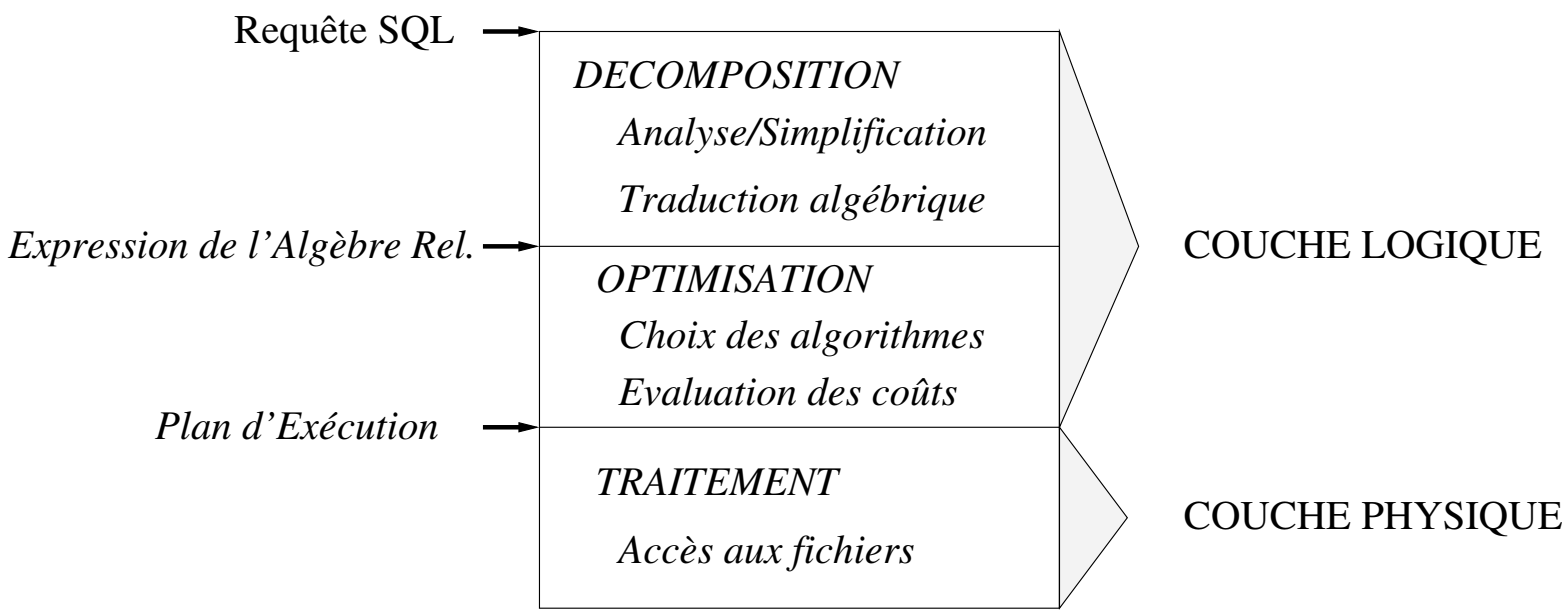
Comme on l’a vu sur l’exemple, l’optimisation s’appuie sur :

1. Des **règles de réécriture** des expressions de l’algèbre.
2. Des connaissances sur **l’organisation physique** de la base (index)
3. Des **statistiques** sur les caractéristiques de la base (taille des relations par exemple).

Un **modèle de coût** permet de classer les différentes stratégies envisagées

Architecture d'un SGBD et Optimisation

LES ETAPES DU TRAITEMENT D'UNE REQUÊTE



Décomposition de requêtes

Analyse syntaxique

On vérifie la validité (syntaxique) de la requête.

1. Contrôle de la structure grammaticale.
2. Vérification de l'existence des relations et des noms d'attributs.

$\Rightarrow$ On utilise le "dictionnaire" de la base qui contient le schéma.

Simplification

D'autres types de transformations avant optimisation :

1. **L'analyse sémantique** détecte les incohérences ou les contradictions. Exemple : "$NoSalle = 11\ AND\ NoSalle = 12$"
2. **Simplification** de clauses inutilement complexes. Exemple : $(A\ OR\ NOT\ B)\ AND\ B$ est équivalent à $A\ AND\ B$.
3. Enfin la requête est normalisée (eg. conditions en forme normale conjonctive) et décomposée en *bloques SELECT-FROM-WHERE* pour faciliter la traduction algébrique.

Traduction algébrique

Déterminer l'expression algébrique équivalente à la requête :

1. arguments du SELECT : projections.
2. arguments du WHERE : $NomAttr1 = NomAttr2$ correspond en général à une jointure, $NomAttr = constante$ à une sélection.

On obtient une expression algébrique qui peut être représentée par un **arbre de requête**.

Traduction algébrique : exemple

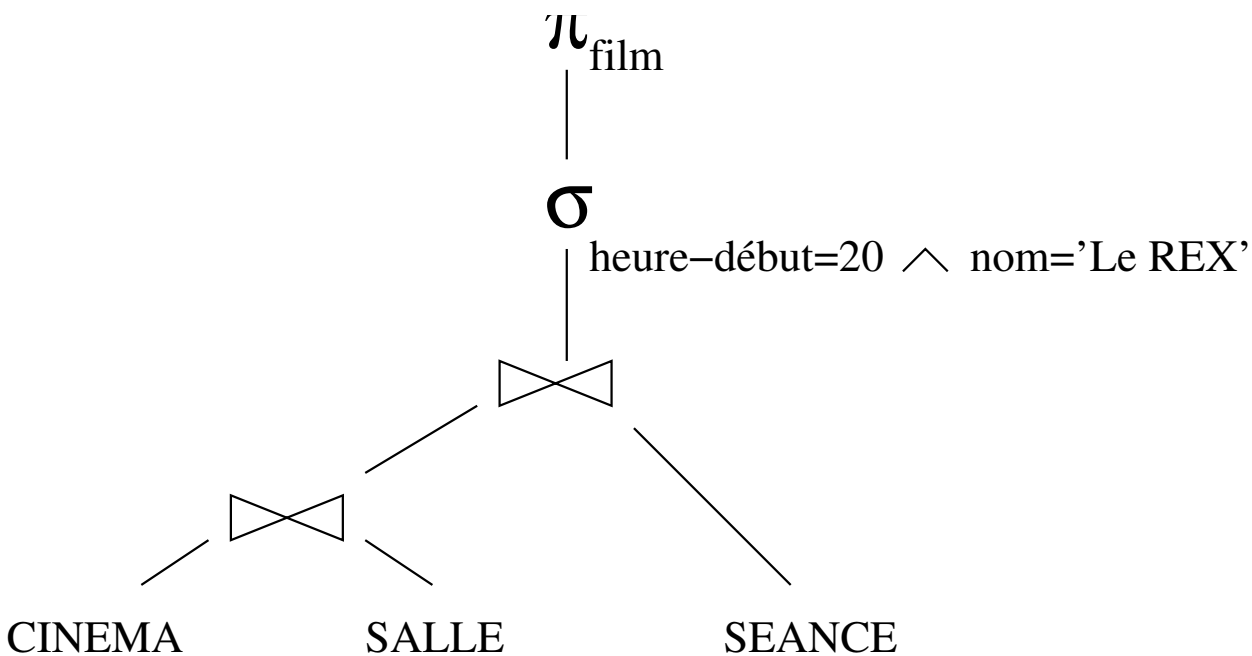
Considérons l'exemple suivant :

$$Quels\ films\ passent\ au\ REX\ à\ 20\ heures\ ?$$

```
SELECT film
FROM   CINÉMA, SALLE, SÉANCE
WHERE  CINÉMA.nom-cinéma = 'Le Rex'
AND    SÉANCE.heure-début = 20
AND    CINÉMA.nom-cinéma = SALLE.nom-cinéma
AND    SALLE.salle = SÉANCE.salle
```

Expression algébrique et arbre de requête

$$\pi_{film}(\sigma_{Nom='Le\ REX' \wedge heure-début=20}((CINEMA \bowtie SALLE) \bowtie SEANCE))$$



Restructuration

Il y a plusieurs expressions **équivalentes** pour une même requête.

ROLE DE L'OPTIMISEUR

1. Trouver les expressions équivalentes à une requête.
2. Les évaluer et choisir la meilleure.

On convertit une expression en une expression équivalente en employant des **règles de réécriture**.

Règles de réécriture

Il en existe beaucoup : en voici 8 parmi les plus importantes.

1. **Commutativité des jointures** :
   $R \bowtie S \equiv S \bowtie R$
2. **Associativité des jointures** :
   $(R \bowtie S) \bowtie T \equiv R \bowtie (S \bowtie T)$
3. **Regroupement des sélections** :
   $\sigma_{A='a' \wedge B='b'}(R) \equiv \sigma_{A='a'}(\sigma_{B='b'}(R))$
4. **Commutativité de la sélection et de la projection**
   $\pi_{A_1,A_2,\ldots A_p}(\sigma_{A_i='a}(R)) \equiv \sigma_{A_i='a}(\pi_{A_1,A_2,\ldots A_p}(R))$, $i \in \{1,\ldots,p\}$
5. **Commutativité de la sélection et de la jointure**.
   $\sigma_{A='a'}(R(\ldots A \ldots) \bowtie S) \equiv \sigma_{A='a'}(R) \bowtie S$

6. **Distributivité de la sélection sur l'union**.
   $\sigma_{A='a'}(R \cup S) \equiv \sigma_{A='a'}(R) \cup \sigma_{A='a'}(S)$
   NB : valable aussi pour la différence.

7. **Commutativité de la projection et de la jointure**
   $\pi_{A_1 \ldots A_p B_1 \ldots B_q}(R \bowtie_{A_i=B_j} S) \equiv$
   $\pi_{A_1 \ldots A_p}(R) \bowtie_{A_i=B_j} \pi_{B_1 \ldots B_q}(S)$,
   $(i \in \{1, \ldots, p\}, j \in \{1, \ldots, q\})$

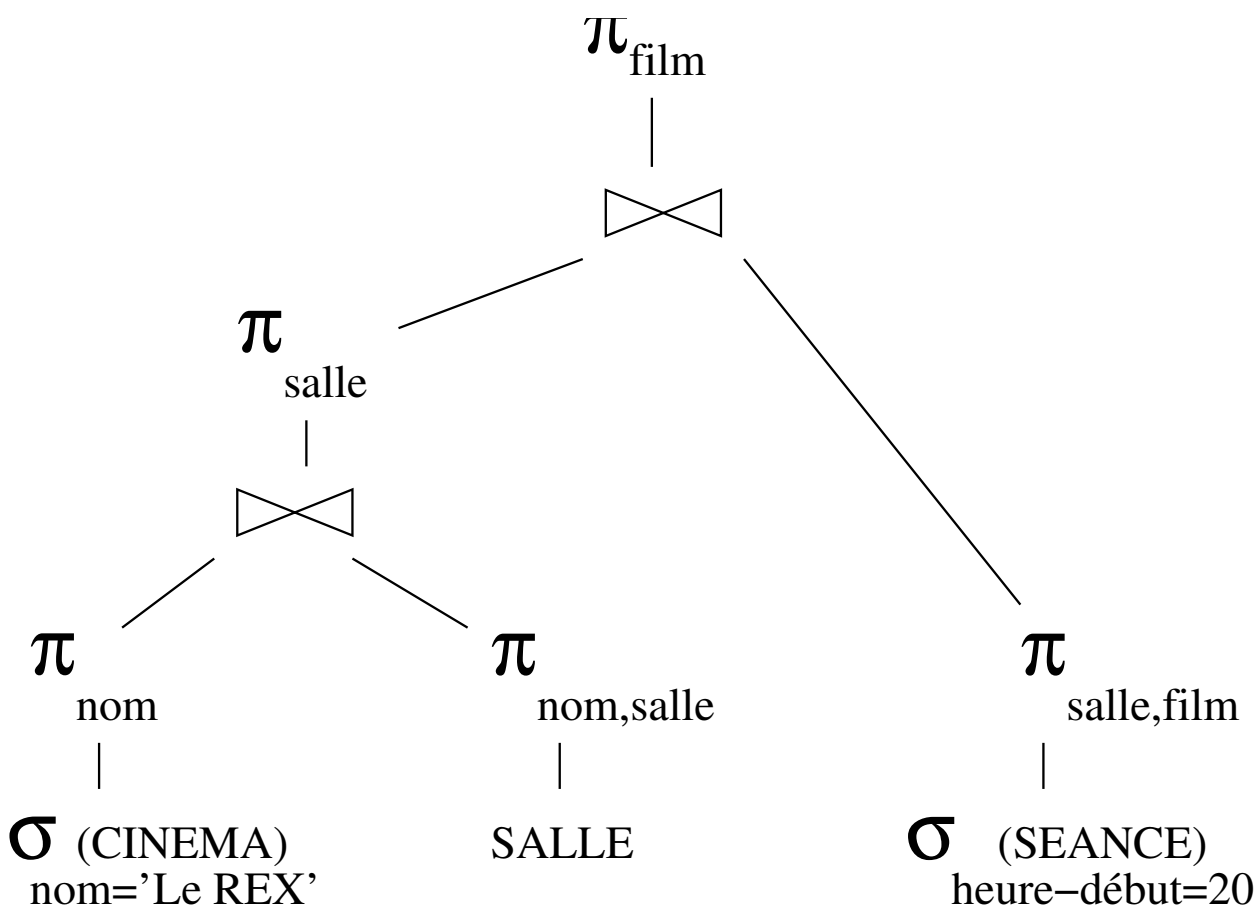
8. **Distributivité de la projection sur l'union**
   $\pi_{A_1 A_2 \ldots A_p}(R \cup S) \equiv \pi_{A_1 A_2 \ldots A_p}(R) \cup \pi_{A_1 A_2 \ldots A_p}(S)$

Exemple d’un algorithme de restructuration

Voici un algorithme basé sur les propriétés précédentes.

1. Séparer les sélections avec plusieurs prédicats en plusieurs sélections à un prédicat (règle 3).
2. Descendre les sélections le plus bas possible dans l’arbre (règles 4, 5, 6).
3. Regrouper les sélections sur une même relation (règle 3).
4. Descendre les projections le plus bas possible (règles 7 et 8).
5. Regrouper les projections sur une même relation.

Arbre de requête après restructuration



Quelques remarques sur l'algorithme précédent

L'idée de base est de réduire le plus rapidement possible la taille des relations manipulées. Donc :

1. On effectue les sélections d'abord car on considère que c'est l'opérateur le plus "réducteur".
2. On élimine dès que possible les attributs inutiles par projection.
3. Enfin on effectue les jointures.

Le plan obtenu est-il TOUJOURS optimal (pour toutes les bases de données) ? ' La réponse est NON!

Un contre-exemple

Quels sont les films visibles entre 14h et 22h?

Voici deux expressions de l'algèbre, dont l'une "optimisée" :

1. $\pi_{film}(\sigma_{h-début>14 \wedge h-début<22}(FILM \bowtie SEANCE))$
2. $\pi_{film}(FILM \bowtie \sigma_{h-début>14 \wedge h-début<22}(SEANCE))$

La relation FILM occupe 8 pages, la relation SEANCE 50.

Evaluation des coûts

Hypothèses : (i) 90 % des séances ont lieu entre 14 et 22 heures, (ii) seulement 20 % des films dans la table SEANCE existent dans la table FILM.

1. Jointure : on lit 400 pages et on aboutit à 10 pages (20% de 50 pages).
   Sélection : on se ramène à 9 pages (90%).
   Nombre d'E/S : 400E + 10x2E/S + 9S = 429E/S.

2. Sélection : on lit 50 pages et on aboutit à 45 pages (90%).
   Jointure : on lit 360 (45x8) pages et on aboutit à 9 pages (20% de 45).
   Nombre d'E/S : 50E + 45S + 360E +9S = 464E/S.

$\Rightarrow$ la première stratégie est la meilleure ! Ici la jointure est plus sélective que la sélection (cas rare).

Traduction algébrique : conclusion

La réécriture algébrique est nécessaire mais pas suffisante. L'optimiseur tient également compte :

1. Des chemins d'accès aux données (index).
2. Des différents algorithmes implantant une même operation algébrique (jointures).
3. De propriétés statistiques de la base.

Les chemins d’accès

Ils dépendent des organisations de fichiers existantes :

1. Balayage séquentiel
2. Parcours d’index
3. Accès par hachage

Attention ! Dans certains cas un balayage peut être préférable à un parcours d’index.

Algorithmes pour les opérations algébriques

Il existe souvent plusieurs algorithmes pour implanter une opération.

L'opération la plus étudiée est la JOINTURE :

1. Boucles imbriquées simple,
2. Tri-fusion,
3. Jointure par hachage,
4. Boucles imbriquées avec accès à une des relations par index.

Le choix dépend essentiellement - mais pas uniquement - du chemin d'accès disponible.

Algorithmes de jointure sans index

En l’absence d’index, les principaux algorithmes sont :

1. Boucles imbriquées.
2. Tri-fusion.
3. Jointure par hachage.

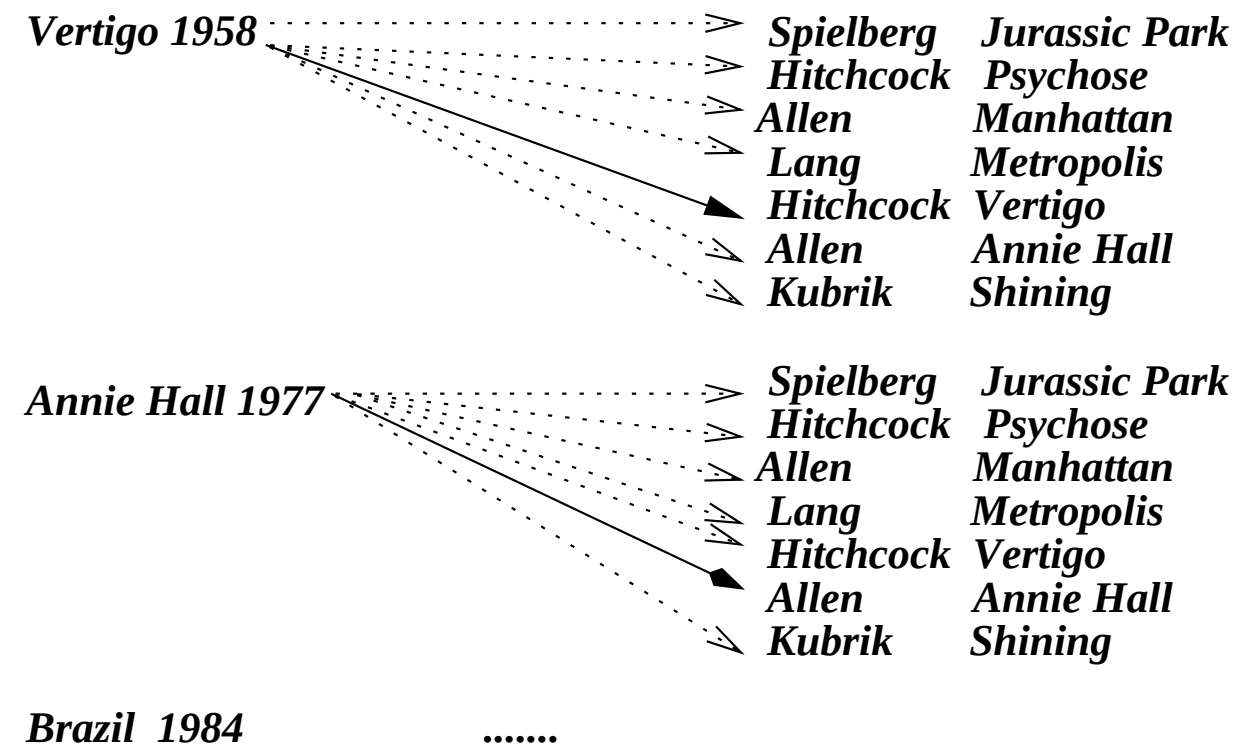
Jointure par boucles imbriquées

A utiliser quand les tailles des relations sont petites.

Soit les deux relations R et S :

ALGORITHME **boucles-imbriquées**
begin
J:=$\emptyset$
for each r **in** R
  for each s **in** S
    if *r et s sont joignables* **then** $J := J + \{r \bowtie s\}$
end

Exemple de jointure par boucles imbriquées



Comparaison

Association

Analyse

La boucle s'effectue en fait à deux niveaux :

1. Au niveau des **pages** pour les charger en mémoire.
2. Au niveau des **articles** des pages chargées en mémoire.

Du point de vue E/S, c'est la première phase qui compte. Si T_R et T_S représentent le nombre de pages de R et S respectivement, le coût de la jointure est :

$$T_R \times T_S$$

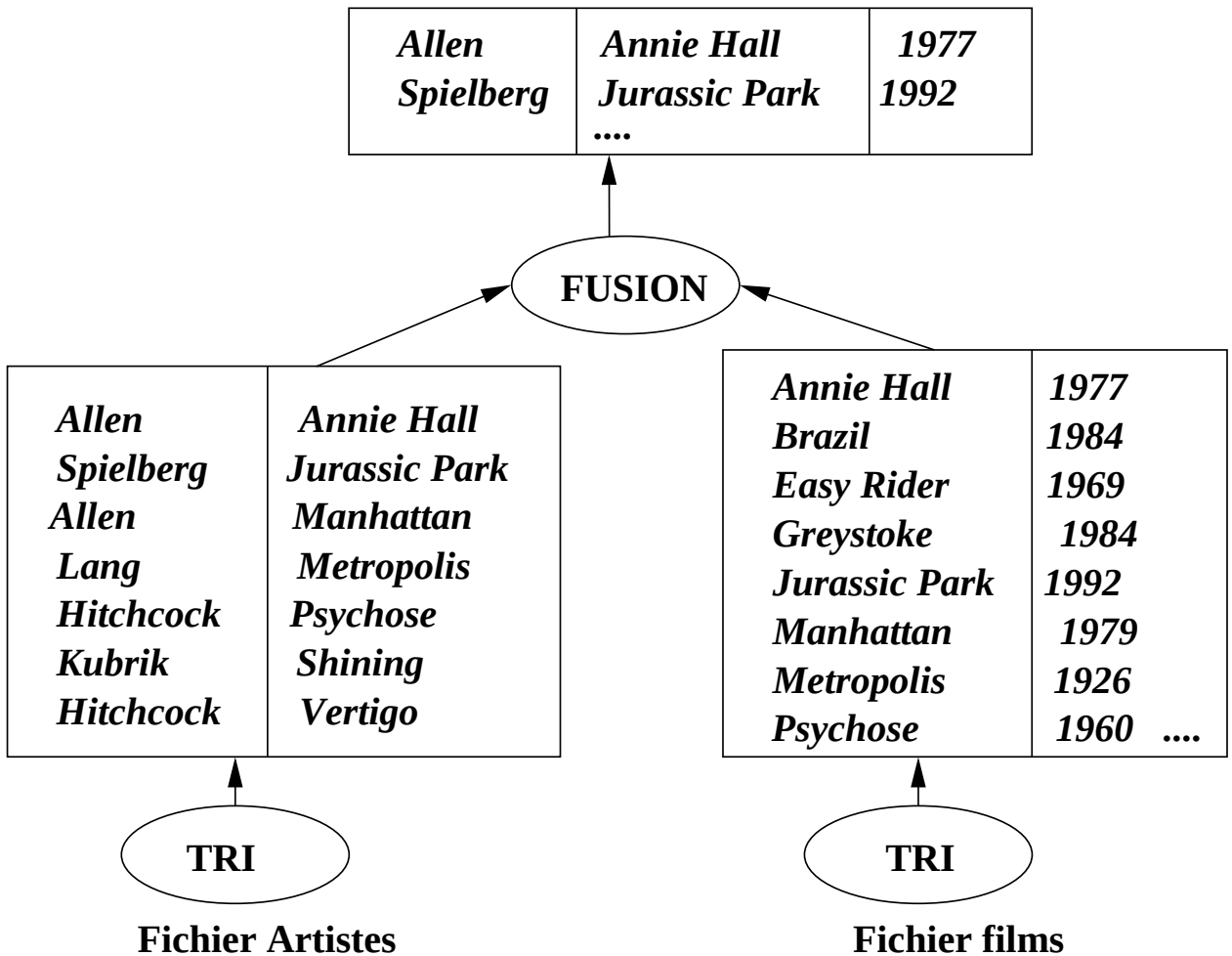
On ne tient pas compte dans l'évaluation du coût des algorithmes de jointure, du coût d'écriture du résultat sur disque, lequel dépend de la taille du résultat.

Jointure par tri-fusion

Soit l'expression $\pi_{R.Ap,S.Bq}(R \bowtie_{A_i=B_j} S)$.

Projeter R sur $\{A_p, A_i\}$
Trier R sur A_i
Projeter S sur $\{B_q, B_j\}$
Trier S sur B_j
Fusionner les deux listes triées.

On les parcourt en parallèle en joignant les n-uplets ayant même valeur pour A_i et B_j.


Allen
Spielberg
Annie Hall
Jurassic Park
....
1977
1992
FUSION
Allen
Spielberg
Allen
Lang
Hitchcock
Kubrik
Hitchcock
Annie Hall
Jurassic Park
Manhattan
Metropolis
Psychose
Shining
Vertigo
Annie Hall
Brazil
Easy Rider
Greystoke
Jurassic Park
Manhattan
Metropolis
Psychose
1977
1984
1969
1984
1992
1979
1926
1960
....
TRI
TRI
Fichier Artistes
Fichier films

Tri-fusion : performance

Le coût est dominé par la phase de tri :

$$O(card(R) \times \log(card(R)) + card(S) \times \log(card(S))).$$

Dans la seconde phase, un simple parcours en parallèle suffit.

Cet algorithme est particulièrement intéressant quand les données sont déjà triées en entrée.

Pour des grandes relations et en l'absence d'index, la jointure par tri-fusion présente les avantages suivants :

1. **Efficacité** : bien meilleure que les boucles imbriquées.
2. **Manipulation de données triées** : facilite l'élimination de dupliqués ou l'affichage ordonné.
3. **Très général** : permet de traiter tous les types de θ-jointure

Jointure par hachage

Comme le tri-fusion, la jointure par hachage permet de limiter le nombre de comparaisons entre n-uplets.

1. Une des relations, $R1$, est hachée sur l'attribut de jointure avec une fonction H.
2. La deuxième relation est parcourue séquentiellement. Pour chaque n-uplet, on consulte la page indiquée par application de la fonction H et on regarde si elle contient des n-uplets de $R1$. Si oui on fait la jointure limitée à ces n-uplets.

Jointure par hachage : algorithme

Pour une jointure $R \bowtie_{A=B} S$.

Pour chaque n-uplet r de R **faire**

placer r dans la page indiquée par $H(r.A)$

Pour chaque n-uplet s de S **faire**

calculer $H(s.B)$

lire la page p indiquée par $H(s.B)$

effectuer la jointure entre $\{s\}$ et les n-uplets de p

Jointure par hachage : discussion

Coût (en E/S), en supposant k articles par page et un tampon de 2 pages en memoire centrale (dans le pire des cas):

1. Phase 1 : Coût du hachage de $R1$: $T_{R1}E + 2 \times k \times T_{R1}E/S$ (pour chaque n-uplet il faut lire et écrire une page).
2. Phase 2 : Lecture de $R2$. Pour chaque page, on lit k pages de la relation hachée $R1$.

$$\text{Coût} = ((1 + 2k) \times T_{R1}) + ((1 + k) \times T_{R2})$$

Si $R1$ tient en mémoire centrale, le coût se réduit à $T_{R1} + T_{R2}$.

NB : contrairement au tri-fusion, la jointure par hachage n'est pas adaptée aux jointures avec inégalités.

Jointure avec une table indexée

1. On parcourt séquentiellement la relation sans index.
2. Pour chaque n-uplet, on recherche par l'index les n-uplets de la seconde relation qui satisfont la condition de jointure (traversée de l'index et accès aux nuplets de la seconde relation par adresse)

Boucles imbriquées avec une table indexée

ALGORITHME **boucles-imbriquées-index**
begin
J:=$\emptyset$
for each r **in** R
  for each s **in** $Index_{S_B}(r.A)$
    J := J + $\{r \bowtie s\}$
end

La fonction $Index_{S_B}(r.A)$ donne les nuplets de S dont l'attribut B a pour valeur $r.A$ en traversant l'index de S sur B

Coût : $O(card(R) \times log(card(S)))$.

Jointure avec deux tables indexées

Si les deux tables sont indexées, on peut utiliser une variante de l'algorithme de tri-fusion :

1. On fusionne les deux index (déjà triés) pour constituer une liste (Rid, Sid) de couples d'adresses pour les articles satisfaisant la condition de jointure.
2. On parcourt la liste en accédant aux tables pour constituer le résultat.

Inconvénient : on risque de lire plusieurs fois la même page. En pratique, on préfère utiliser une boucle imbriquée en prenant la plus petite table comme table directrice.

Statistiques

Permettent d'ajuster le choix de l'algorithme. Par exemple :

1. Boucles imbriquées simples si les relations sont petites.
2. Balayage séquentiel au lieu de parcours d'index si la sélectivité est faible.

Suppose

1. Soit l'existence d'un module récoltant périodiquement des statistiques sur la base
2. Soit l'estimation en temps réel des statistiques par échantillonnage.

Plans d'exécution

Le résultat de l'optimisation est un **plan d'exécution**: c'est un ensemble d'opérations de niveau intermédiaire, dit **algèbre "physique"** constituée :

1. De chemins d'accès aux données
2. D'opérations manipulant les données, (correspondant aux noeuds internes de l'arbre de requête).

Plans d'exécution sur la requête :

$$Quels\ films\ passent\ au\ REX\ à\ 20\ heures\,?$$

Algèbre physique

CHEMINS D'ACCES

Sequentiel

TABLE

Parcours sequentiel

Adresse

TABLE

Acces par adresse

Attribut(s)

INDEX

Parcours d'index

OPERATIONS PHYSIQUES

Critere

Selection

Selection selon un critere

Critere

Filtre

Filtre d'un ensemble en fonction d'un autre

Attribut(s)

Tri

Tri sur un attribut

Critere

Jointure

Jointure selon un critere

Critere

Fusion

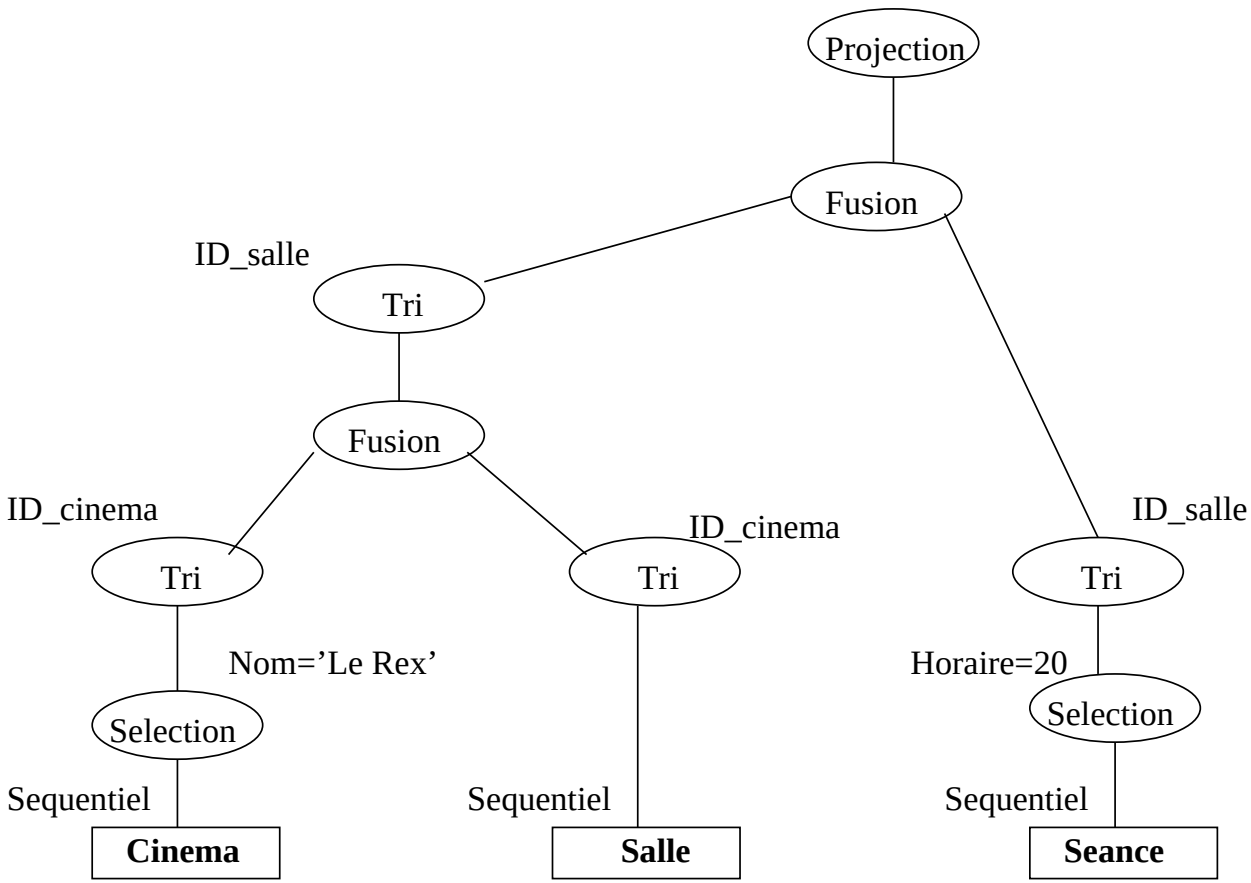
Fusion de deux ensembles tries

Attribut(s)

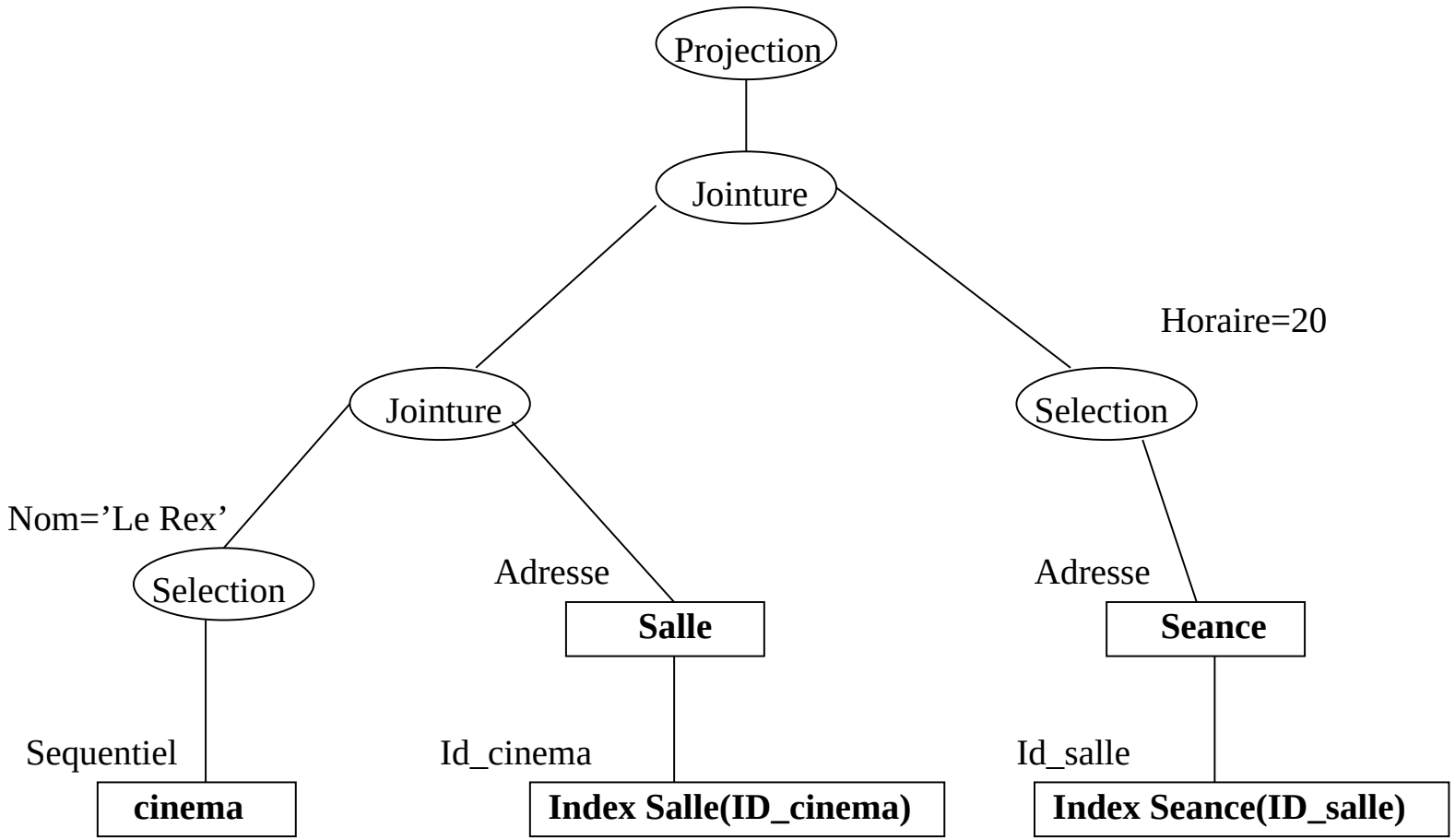
Projection

Projection sur des attributs

Sans index ni hachage



Avec un index sur toutes les clés



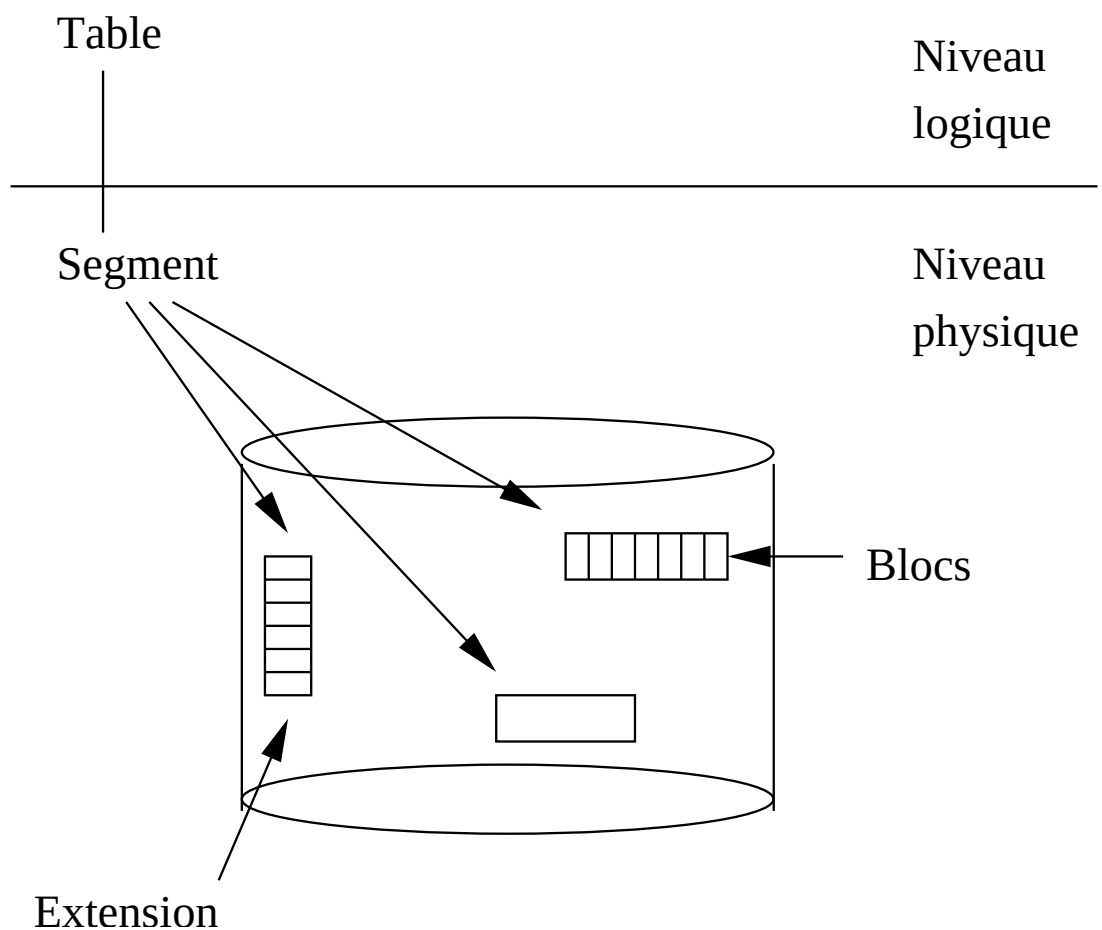
REPRESENTATION PHYSIQUE DANS ORACLE

Représentation physique dans ORACLE V7

Les principales structures physiques utilisées dans ORACLE sont :

1. Le **bloc** est l'unité physique d'E/S. La taille d'un bloc ORACLE est un multiple de la taille des blocs du système sous-jacent.
2. L'**extension** est un ensemble de blocs contigus contenant un même type d'information.
3. Le **segment** est un ensemble d'extensions stockant un objet logique (une table, un index ...).

Tables, segments, extensions et blocs



Le segment ORACLE

Le segment est la zone physique contenant un objet logique. Il existe quatre types de segments :

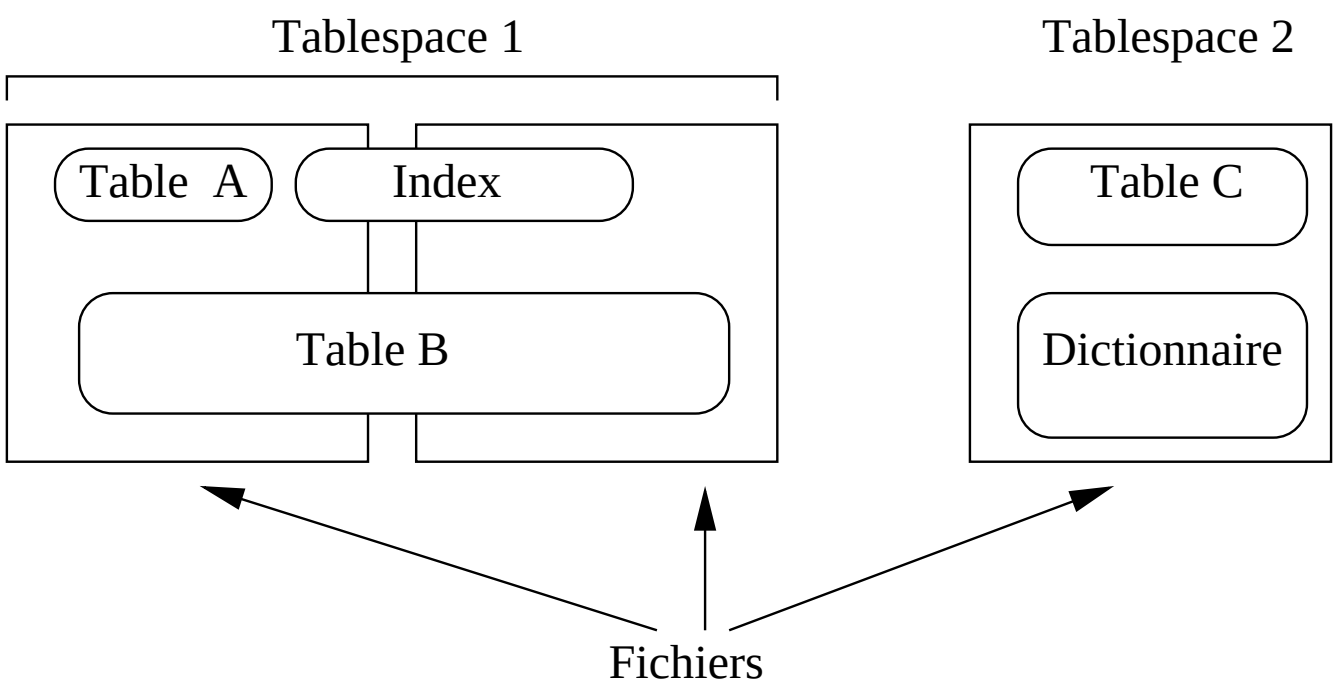
1. Le segment de données (pour une table ou un $cluster$).
2. Le segment d'index.
3. Le $rollback\ segment$ utilisé pour les transactions.
4. Le segment temporaire (utilisé pour les tris par exemple).

Base ORACLE, fichiers et $TABLESPACE$

1. **Physiquement**, une base ORACLE est un ensemble de fichiers.
2. **Logiquement**, une base est divisée par l'administrateur en $tablespace$. Chaque $tablespace$ consiste en un (au moins) ou plusieurs fichiers.

La notion de $tablespace$ permet :

1. De contrôler l'emplacement physique des données. (par ex. : le dictionnaire sur un disque, les données utilisateur sur un autre).
2. de faciliter la gestion (sauvegarde, protection, etc).


Tablespace 1
Tablespace 2
Table A
Index
Table B
Table C
Dictionnaire
Fichiers

Stockage des données

Il existe deux manières de stocker une table :

1. **Indépendamment**. Un segment est alors automatiquement alloué à la table. Il est possible de spécifier des paramètres pour ce segment :
   (a) Sa taille initiale
   (b) Le pourcentage d'espace libre dans chaque bloc.
   (c) La taille des extensions.
2. **Dans un** $cluster$.

Stockage des n-uplets

En règle générale un n-uplet est stocké dans un seul bloc. L'adresse physique d'un n-uplet est le $ROWID$ qui se décompose en trois parties :

1. Le numéro du n-uplet dans la page.
2. Le numéro de la page, relatif au **fichier** dans lequel se trouve le n-uplet.
3. Le numéro du fichier.

Exemple : 00000DD5.000.001 est l'adresse du premier n-uplet du bloc DD5 dans le premier fichier.

Structures de données pour l'optimisation

ORACLE 7 propose trois structures pour l'optimisation de requêtes :

1. Les index
2. Les "regroupements" de tables ($Cluster$).
3. Le hachage.

Les index ORACLE

On peut créer des index sur tout attribut (ou tout ensemble d'attributs) d'une table. ORACLE utilise l'arbre B+.

1. Les noeuds contiennent les valeurs de l'attribut (ou des attributs) clé(s).
2. Les feuilles contiennent chaque valeur indexée et le $ROWID$ correspondant.

Un index est stocké dans un segment qui lui est propre. On peut le placer par exemple sur un autre disque que celui contenant la table.

Les $cluster$

Le $cluster$ (regroupement) est un structure permettant d'optimiser les jointures. Par exemple, pour les tables $CINEMA$ et $SALLE$ qui sont fréquemment jointes sur l'attribut $ID - Cinema$:

1. On groupe les n-uplets de $CINEMA$ et de $SALLE$ ayant même valeur pour l'attribut $ID - cinema$.
2. On stocke ces groupes de n-uplets dans les pages d'un segment spécial de type $cluster$.
3. On crée un index sur $ID - cinema$.

Clé de regroupement (ID−cinéma)	**Nom−cinéma**	**Adresse**	
1209	Le Rex	2 Bd Italiens	
	ID−salle	**Nom−salle**	**Capacité**
	1098	Grande Salle	450
	298	Salle 2	200
	198	Salle 3	120
1210	**Nom−cinéma**	**Adresse**	
	Kino	243 Bd Raspail	
	ID−salle	**Nom−salle**	**Capacité**
	980	Salle 1	340
	...	...	...

Le hachage

On définit un $hash\ cluster$ décrivant les caractéristiques physiques de la table :

```
CREATE CLUSTER hash-cinéma (ID-cinéma NUMBER(10))
   HASH IS ID-cinéma HASHKEYS 1 000 SIZE 2K
```

$HASH\ IS$ (optionnel) spécifie la clé à hacher.
$HASHKEYS$ est le nombre de valeurs de la clé de hachage.
$SIZE$ est la taille des données pour une clé donné (taille d'un n-uplet si la clé de hachage est la clé de la table). ORACLE détermine le nombre de pages allouées, ainsi que la fonction de hachage. On fait référence au $hash\ cluster$ en créant une table.

OPTIMISATION DANS ORACLE

Optimisation : l’exemple de ORACLE Version 7

Plan de la présentation :

1. Optimisation : principes et outils d’analyse.
2. Présentation sur des exemples.

Optimisation - principes généraux et outils d’analyse

L’optimiseur

L’optimiseur ORACLE suit une approche classique :

1. Génération de plusieurs plans d’exécution.
2. Estimation du coût de chaque plan généré.
3. Choix du meilleur et exécution.

Estimation du coût d’un plan d’exécution

Beaucoup de paramètres entrent dans l’estimation du coût :

1. Les chemins d’accès disponibles.
2. Les opérations physiques de traitement des résultats intermédiaires.
3. Des statistiques sur les tables concernées (taille, sélectivité). Les statistiques sont calculées par appel explicite à l’outil ANALYSE.
4. Les ressources disponibles.

Les chemins d'accès

1. **Parcours séquentiel** ($FULL\ TABLE\ SCAN$).
2. **Par adresse** ($ACCESS\ BY\ ROWID$).
3. **Parcours de regroupement** ($CLUSTER\ SCAN$). On récupère alors dans une même lecture les n-uplets des 2 tables du $cluster$.
4. **Recherche par hachage** ($HASH\ SCAN$).
5. **Parcours d'index** ($INDEX\ SCAN$).

Opérations physiques

Voici les principales :

1. $INTERSECTION$: intersection de deux ensembles de n-uplets.
2. $CONCATENATION$: union de deux ensembles.
3. $FILTER$: élimination de n-uplets (sélection).
4. $PROJECTION$: opération de l'algèbre relationnelle.

D'autres opérations sont liées aux algorithmes de jointures.

Algorithmes de jointure sous ORACLE

ORACLE utilise trois algorithmes de jointure :

1. **boucles imbriquées** quand il y a au moins un index.
   Opération $NESTED\ LOOP$.
2. **Tri/fusion** quand il n'y a pas d'index.
   Opération $SORT$ et $MERGE$.
3. Enfin, en présence d'un $join\ cluster$, on fait une recherche avec l'index du $cluster$, puis un accès au $cluster$ lui-même.

L’outil EXPLAIN

L’outil EXPLAIN donne le plan d’exécution d’une requête. La description comprend :

1. Le chemin d’accès utilisé.
2. Les opérations physiques (tri, fusion, intersection, ...).
3. L’ordre des opérations. Il est représentable par un arbre.

EXPLAIN par l'exemple : schéma de la base

CINEMA (ID-cinéma*, Nom, Adresse);

SALLE (ID-salle*, Nom, Capacité, ID-cinéma+);

FILM (ID-film, Titre, Année, ID-réalisateur+)

SEANCE (ID-séance*, Heure-début, Heure-fin, ID-salle+,ID-film

ARTISTE (ID-artiste*, Nom, Date-naissance)

Attributs avec une * : index unique.

Attributs avec une + : index non unique.

Interprétation d'une requête par EXPLAIN

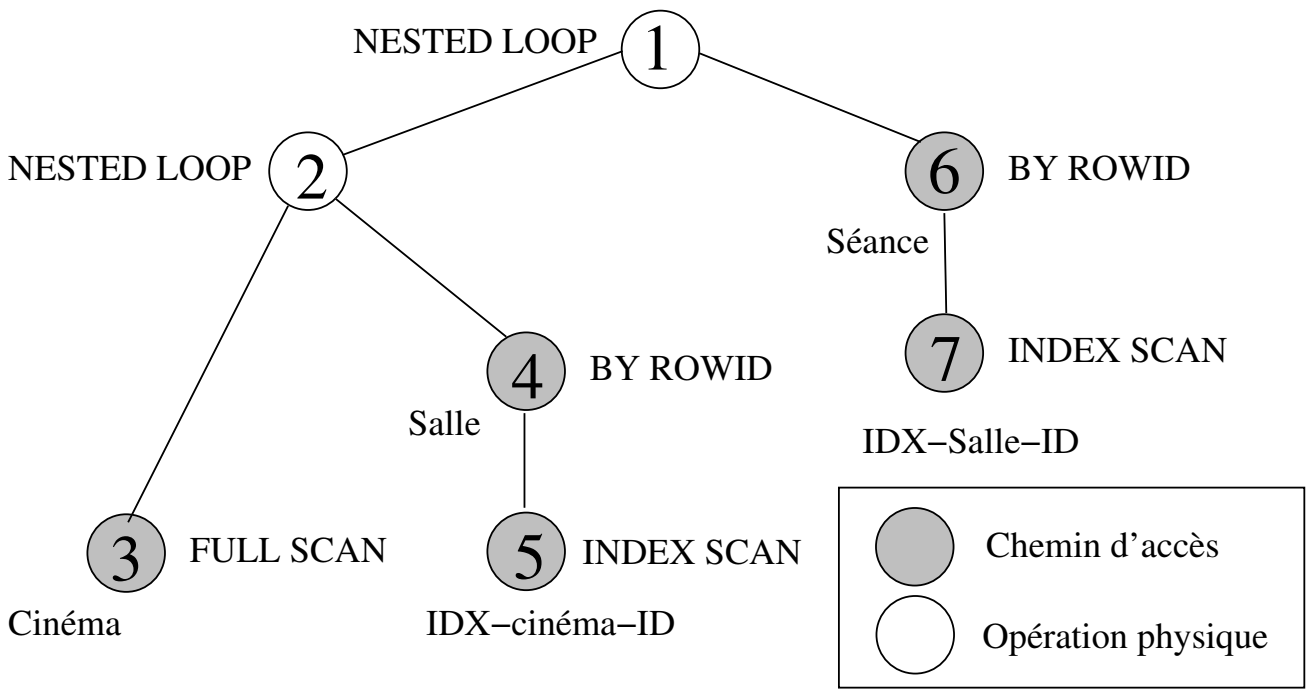
Reprenons l'exemple : Quels films passent aux Rex à 20 heures ?

```
EXPLAIN PLAN SET statement-id = 'cin'
FOR     SELECT ID-film
        FROM   Cinéma, Salle, Séance
        WHERE  Cinéma.ID-cinéma = Salle.ID-cinéma
        AND    Salle.ID-salle = Séance.ID-salle
        AND    Cinéma.nom = 'Le Rex'
        AND    Séance.heure-début = '20H'
```

Plan d'exécution donné par EXPLAIN

```
0 SELECT STATEMENT
  1 NESTED LOOP
    2 NESTED LOOPS
      3 TABLE ACCESS FULL CINEMA
      4 TABLE ACCESS BY ROWID SALLE
        5 INDEX RANGE SCAN IDX-CINEMA-ID
    6 TABLE ACCESS BY ROWID SEANCE
      7 INDEX RANGE SCAN IDX-SALLE-ID
```

Représentation arborescente du plan d'exécution



Quelques remarques sur EXPLAIN

EXPLAIN utilise un ensemble de primitives que nous avons appelé "algèbre physique”: des opérations comme le tri n’existent pas au niveau relationnel. D’autres opérations de l’algèbre relationnelle sont regroupées en une seule opération physique.

- Par exemple, la sélection sur l’horaire des séances est effectuée en même temps que la recherche par ROWID (étape 6).

Exemple: Sélection sans index

```
SELECT * FROM cinéma WHERE  nom = 'Le Rex'
```

Plan d'exécution :

```
0 SELECT STATEMENT
  1 TABLE ACCESS FULL CINEMA
```

Sélection avec index

```
SELECT * FROM cinéma WHERE  ID-cinéma = 1908
```

Plan d’exécution :

```
0 SELECT STATEMENT
  1 TABLE ACCESS BY ROWID CINEMA
    2 INDEX UNIQUE SCAN IDX-CINEMA-ID
```

Sélection conjonctive avec un index

```
SELECT capacité FROM Salle
WHERE  ID-cinéma =187 AND nom = 'Salle 1'
```

Plan d'exécution :

```
0 SELECT STATEMENT
  1 TABLE ACCESS BY ROWID SALLE
    2 INDEX RANGE SCAN IDX-SALLE-CINEMA-ID
```

Sélection conjonctive avec deux index

```
SELECT  nom FROM   Salle
 WHERE ID-cinéma = 1098 AND capacité = 150
```

Plan d'exécution :

```
0 SELECT STATEMENT
  1 TABLE ACCESS BY ROWID SALLE
    2 AND-EQUAL
      3 INDEX RANGE SCAN IDX-SALLE-CINEMA-ID
      4 INDEX RANGE SCAN IDX-CAPACITE
```

Sélection disjonctive avec index

SELECT nom FROM Salle
WHERE ID-cinéma = 1098 OR capacité > 150

Plan d'exécution :

```
0 SELECT STATEMENT
  1 CONCATENATION
    2 TABLE ACCESS BY ROWID SALLE
      3 INDEX RANGE SCAN IDX-CAPACITE
    4 TABLE ACCESS BY ROWID SALLE
      5 INDEX RANGE SCAN IDX-SALLE-CINEMA-ID
```

Sélection disjonctive avec et sans index

```
SELECT  nom FROM   Salle
WHERE   ID-cinema = 1098 OR nom = ’Salle 1’
```

Plan d’exécution :

```
0 SELECT STATEMENT
  1 TABLE ACCESS FULL SALLE
```

Jointure avec index

```
SELECT Cinéma.nom,capacité FROM cinéma, salle
WHERE  Cinéma.ID-cinéma = salle.ID-cinéma
```

Plan d'exécution :

```
0 SELECT STATEMENT
  1 NESTED LOOPS
    2 TABLE ACCESS FULL SALLE
    3 TABLE ACCESS BY ROWID CINEMA
      4 INDEX UNIQUE SCAN IDX-CINEMA-ID
```

Jointure et sélection avec index

SELECT Cinéma.nom,capacité FROM Cinéma, Salle
WHERE Cinema.ID-cinéma = salle.ID-cinéma
AND capacité $>$ 150

Plan d'exécution :

```
0 SELECT STATEMENT
  1 NESTED LOOPS
    2 TABLE ACCESS BY ROWID SALLE
      3 INDEX RANGE SCAN IDX-CAPACITE
    4 TABLE ACCESS BY ROWID CINEMA
      5 INDEX UNIQUE SCAN IDX-CINEMA-ID
```

Jointure sans index

```
SELECT titre
FROM Film, Séance
WHERE  Film.ID-film  = Séance.ID-film
AND     heure-début = ’14H00’
```

Plan d’exécution :

```
0 SELECT STATEMENT
  1 MERGE JOIN
    2 SORT JOIN
      3 TABLE ACCESS FULL SEANCE
    4 SORT JOIN
      5 TABLE ACCESS FULL FILM
```

Différence

Dans quel cinéma ne peut-on voir de film après 23H ?

```
SELECT Cinéma.nom
FROM   Cinéma, Salle
WHERE  Cinéma.ID-cinéma = Salle.ID-cinéma
AND    NOT EXISTS (SELECT * FROM séance
                   WHERE Salle.ID-salle = Séance.ID-salle
                   AND heure-fin > '23H00')
```

Plan d’exécution donné par EXPLAIN

```
0 SELECT STATEMENT
  1 FILTER
    2 NESTED LOOPS
      3 TABLE ACCESS FULL SALLE
      4 TABLE ACCESS BY ROWID CINEMA
        5 INDEX UNIQUE SCAN IDX-CINEMA-ID
    6 TABLE ACCESS BY ROWID SEANCE
      7 INDEX RANGE SCAN IDX-SEANCE-SALLE-ID
```

CONCURRENCE ET REPRISE APRÈS PANNE

I. INTRODUCTION ET PROBLÉMATIQUE

1. La notion de transaction

Modèle de base de données

- BD centralisée, accès concurrent de plusieurs programmes
- modèle simplifié
  - BD = ensemble d'*enregistrements* nommés

    Ex. **x**: 3 ; **y**: "Toto" ; **z**: 3.14 ...
  - *opérations* sur les enregistrements: lecture, écriture, création

Programmes et transactions

- exécution d'un programme accédant à la BD = séquence d'opérations sur les enregistrements
- opérations : lecture (val=*Read*(x)), écriture (*Write*(x, val))
- découpage en *transactions*

Transaction

- opérations de contrôle de transaction : *Start* (démarrage), *Commit* (validation), *Abort* (annulation)
- transaction = séquence d'opérations qui démarre par *Start* et se termine par *Commit* ou par *Abort*
- cohérence logique (maxi-opération), unité d'annulation

Exemple: programme de crédit d'un compte bancaire

```
Crédit (Enreg compte; Val montant)
    Val temp;
begin Start;
        temp = Read(compte);
        Write(compte, temp+montant);
        Commit;
end;
```

- les entrées du programme: le compte (enregistrement) et le montant du crédit (valeur)
- l'exécution du programme $\rightarrow$ transaction
- plusieurs exécutions concurrentes du même programme possibles

Exemple: transfert entre deux comptes

```
Transfert (Enreg source, dest; Val montant)
    Val temp;
begin Start;
        temp = Read(source);
        if temp < montant then Abort;
        else Write(source, temp-montant);
                temp = Read(dest);
                Write(dest, temp+montant); Commit;
        end if;
end;
```

- l'exécution peut produire des transactions différentes:
  1. *Start*, *Read*(source), *Abort*
  2. *Start*, *Read*(source), *Write*(source), *Read*(dest), *Write*(dest), *Commit*

Mise-à-jour de la BD

Deux variantes:

- *immédiate*: modification immédiate de la BD, visible par les autres transactions
- *différée*: chaque transaction travaille sur des copies, avec mise-à-jour de la BD à la fin de la transaction

Hypothèse: mise-à-jour immédiate

Propriétés des transactions (*ACID*)

Atomicité: une transaction doit s'exécuter en totalité, une exécution partielle est inacceptable

- une transaction interrompue doit être annulée (Abort)

Cohérence: respect des contraintes d'intégrité sur les données

- solde(compte) ≥ 0; solde(source) + solde(dest) = *const*
- une transaction modifie la BD d'un état initial cohérent à un état final cohérent
- pendant la transaction, l'état peut être incohérent!

Isolation: une transaction ne voit pas les effets des autres transactions concurrentes

- la transaction s'exécute comme si elle était la seule
- objectif: exécution concurrente des transactions équivalente à une exécution en série (non-concurrente)

Durabilité: les effets d'une transaction validée par Commit sont permanents

- on ne doit pas annuler une transaction validée

Concurrence

- plusieurs transactions s’exécutent en même temps
- les opérations sont exécutées en séquence!
- concurrence = entrelacement des opérations de plusieurs transactions

Notation

- transaction T_i = séquence d'opérations
- opérations $r_i[x]$, $w_i[x]$, a_i, c_i
- remarque: les valeurs lues ou écrites ne comptent pas ici!
- la séquence se termine par a_i ou c_i, qui n'apparaissent jamais ensemble

Exemple :

T$_1$: $r_1[x]$ $w_1[x]$ c_1 (crédit)
T$_2$: $r_2[x]$ $w_2[x]$ $r_2[y]$ $w_2[y]$ c_2 (transfert)
T$_3$: $r_3[x]$ a_3 (transfert impossible)

Exécution concurrente (histoire)

- séquence d'opérations de plusieurs transactions
- entrelacement des opérations des transactions
- histoire complète: les transactions sont entrières

Exemple :

H$_1$: $r_1[x]$ $r_2[x]$ $w_2[x]$ $r_2[y]$ $w_1[x]$ $w_2[y]$ c_2 c_1 (crédit + transfert, histoire complète)
H$_2$: $r_1[x]$ $r_2[x]$ $w_2[x]$ $r_2[y]$ (histoire incomplète)

2. Contrôle de concurrence

Objectifs

- concurrence = entrelacement des opérations des transactions
- concurrence parfaite: toute opération est exécutée dès son arrivée dans le système
- problème: tout entrelacement n'est pas acceptable
- *objectif*: exécution correcte en respectant les propriétés ACID

Problèmes de concurrence

1. Perte d'une mise à jour

Ex. $T_1 = r_1[x]\ w_1[x]\ c_1$ (crédit x de 100); $T_2 = r_2[x]\ w_2[x]\ c_2$ (crédit x de 50)

au début, x=200

- $H = \underline{r_1[x]_{x:200}}\ r_2[x]_{x:200}\ \underline{w_1[x]_{x:300}}\ w_2[x]_{x:250}\ \underline{c_1}\ c_2$
- Résultat: x=250 au lieu de x=350 ($w_1[x]$ est perdu à cause de $w_2[x]$)
- Problème de cohérence même si l'isolation est respectée !

2. Dépendances non-validées

Ex. Les mêmes transactions, mais T_1 est annulée

- H = $r_1[x]_{x:200}$ $w_1[x]_{x:300}$ $r_2[x]_{x:300}$ $w_2[x]_{x:350}$ c_2 a_1
- Résultat: x=350 au lieu de x=250 (T_1 est annulée)
- $r_2[x]$ utilise la valeur non-validée de x écrite par $w_1[x]$
- Problèmes de cohérence et de durabilité à cause de l'isolation

3. Analyse incohérente

Ex. T_1= transfert $x \rightarrow y$ de 50; T_2= calcul dans z de la somme $x + y$
au début, x=200, y=100

- $T_1 = r_1[x]\ w_1[x]\ r_1[y]\ w_1[y]\ c_1$; $T_2 = r_2[x]\ r_2[y]\ w_2[z]\ c_2$
- $H = \underline{r_1[x]_{x:200}}\ \underline{w_1[x]_{x:150}}\ r_2[x]_{x:150}\ r_2[y]_{y:100}\ w_2[z]_{z:250}\ c_2\ \underline{r_1[y]_{y:100}}\ \underline{w_1[y]_{x:150}}\ \underline{c_1}$
- Résultat: z=250 au lieu de z=300 ($r_2[x]$ est influencé par T_1, mais $r_2[y]$ non)
- Problème de cohérence à cause de l'isolation

4. Objets fantômes

- similaire à l'analyse incohérente, mais produit par la création/suppression d'enregistrements (objets fantômes)
- traité plus loin dans le cours

Contrôle de concurrence

- *solution*: algorithmes de réordonnancement des opérations
- critère de correction utilisé: *exécution sérialisable*
  (équivalente avec une execution en série *quelconque* des transactions)

Remarques

- réordonnancer $\Rightarrow$ retarder certaines opérations
- objectif : un maximum de concurrence, donc un minimum de retards
- l'ordre des opérations dans chaque transaction doit être respecté

3. Le problème des annulations

Annuler dans la BD les effets d'une transaction T $\Longleftrightarrow$

- annuler les écritures de T
- annuler les transactions qui utilisent les écritures de T (dépendances non-validées)

Conclusion: une transaction validée risque encore d'être annulée

Propriétés des exécutions concurrentes par rapport à l'annulation

- recouvrabilité
- éviter les annulations en cascade
- exécution stricte

Recouvrabilité = ne jamais annuler une transaction validée

Ex. $w_1[x]$ $r_2[x]$ $w_2[y]$ c_2 a_1 (a_1 oblige l'annulation de T_2, qui est validée)

Définitions:

- T_2 lit x de T_1: T_2 lit la valeur écrite par T_1 dans x
  (quand T_2 lit x, T_1 est la dernière transaction non-annulée à avoir écrit x)
- T_2 lit de T_1: T_2 lit au moins un enregistrement de T_1

Solution: si T_2 lit de T_1, alors T_2 doit valider après T_1
$\Rightarrow$ *retardement des Commit*
(dans l'exemple, retardement de c_2 après la fin de T_1)

Éviter les annulations en cascade

- exemple précédent: même si c_2 est retardé, T_2 sera quand même annulée à cause de T_1 $\rightarrow$ annulation en cascade

  Ex. $w_1[x]$ $r_2[x]$ $w_2[y]$ a_1 (a_1 oblige a_2)

- l'annulation d'une transaction, même non validée, est génante
- *solution*: T_2 ne doit lire qu'à partir de transactions validées

$\Rightarrow$ *retardement des lectures*
(dans l'exemple, retardement de $r_2[x]$ après la fin de T_1)

Exécution stricte

= éviter les annulations en cascade + pouvoir annuler les écritures

- l'annulation des écritures: par restauration des images avant
- image avant de x par rapport à w[x] = valeur de x avant l'écriture

Problèmes de restauration des images avant

Ex. $w_1[x, 2]$ $w_2[x, 3]$ a_1 a_2 (au début x=1)
image avant($w_1[x]$)=1, image avant($w_2[x]$)=2
a_1 restaure x=1 : erreur, a_2 restaure x=2 : erreur

- *solution*: $w_2[x]$ attend que tout T_i qui a écrit x se termine (par c_i ou a_i)

$\Rightarrow$ *retardement lectures + écritures*
(dans l'exemple, retardement de $w_2[x]$ après la fin de T_1)

4. Tolérance aux pannes

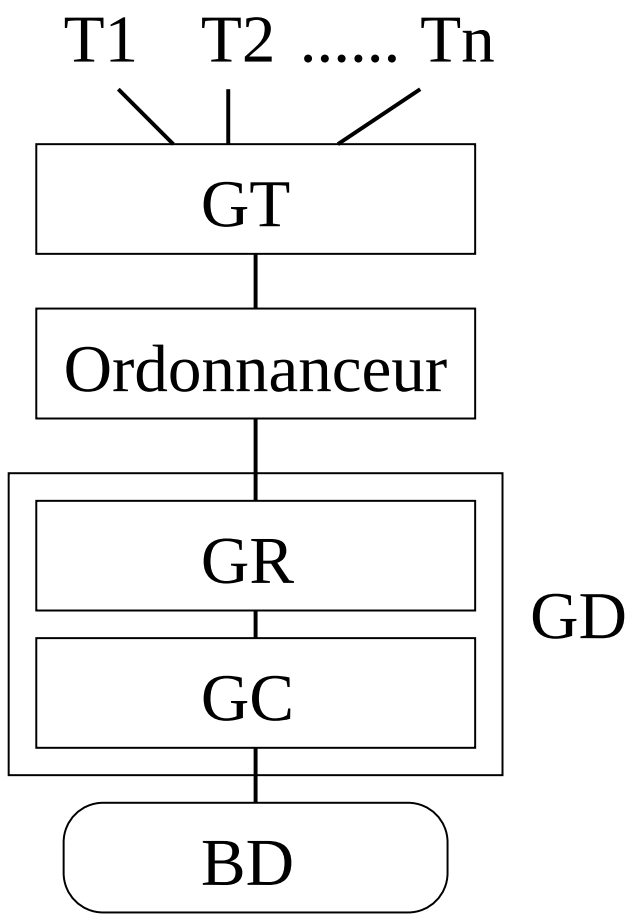
Mémoire SGBD = Mémoire stable (disque) + Mémoire volatile

Catégories de pannes

- **de transaction**: annulation de la transaction
- **de système**: perte du contenu de la mémoire volatile
  - *Restart*: restaurer l'état cohérent de la BD avant la panne
  - journalisation
- **de support physique**: perte contenu mémoire stable
  - duplication par mirroring, archivage

5. Modèle abstrait de la BD

BD centralisée, transactions concurrentes:



Composantes

- **Gestionnaire de transactions** (GT): reçoit les transactions et les prépare pour exécution
- **Ordonnanceur** (Scheduler): contrôle l'ordre d'exécution des opérations (séquences sérialisables et recouvrables)
- **Gestionnaire de reprise** (GR): Commit + Abort
- **Gestionnaire du Cache** (GC): gestion de la mémoire volatile et de la mémoire stable

GR + GC = GD (**Gestionnaire de données**): assure la tolérance aux pannes

II. CONTRÔLE DE CONCURRENCE

1. Théorie de la sérialisabilité

Objectif du contrôle de concurrence

= produire une exécution sérialisable des transactions

Exécution sérialisable : équivalente à une exécution en série *quelconque* des transactions

Exécution en série

- transactions exécutées l'une après l'autre (aucun entrelacement)

Ex. au début x=200; T_1= crédit x de 100; T_2= crédit x de 50

$H_1 = T_1 T_2 = r_1[x]_{x:200}\ w_1[x]_{x:300}\ c_1\ r_2[x]_{x:300}\ w_2[x]_{x:350}\ c_2$

$H_2 = T_2 T_1 = r_2[x]_{x:200}\ w_2[x]_{x:250}\ c_2\ r_1[x]_{x:250}\ w_1[x]_{x:350}\ c_1$

Équivalence de deux exécutions (histoires)

1. Avoir les mêmes transactions et les mêmes opérations
2. Produire le même effet sur la BD (écritures)
3. Produire le même effet dans les transactions (lectures)

Ex. $H_1 \not\equiv H_2$ (conditions 1 et 2 respectées, mais pas 3)

Conflit

Def. **p_i[x]** et **q_j[y]** sont en conflit $\Longleftrightarrow$

- i$\neq$j, x=y (transactions différentes, même enregistrement)
- p_i[x] q_j[x] n'a pas le même effet que q_j[x] p_i[x]

Remarques

- conflit = l'inverse de la commutativité
- commutativité = même effet sur la BD **et** sur les transactions
- conflits: w_i[x]-w_j[x], r_i[x]-w_j[x], w_i[x]-r_j[x]
- seul le couple r_i[x]-r_j[x] n'est pas en conflit

Critère d'équivalence utilisé

- Avoir les mêmes transactions et les mêmes opérations
- *Avoir le même ordre des opérations conflictuelles dans les transactions non-annulées*

Ce dernier critère assure les conditions 2 et 3 de la définition

Ex. $H_1 \not\equiv H_2$ (H_1: $r_1[x]$ - $w_2[x]$; H_2: $w_2[x]$ - $r_1[x]$)

Exemple

T_1: $r_1[x]\ w_1[y]\ w_1[x]\ c_1$ T_2: $w_2[x]\ r_2[y]\ w_2[y]\ c_2$

H_1: $\underline{r_1[x]}\ w_2[x]\ \underline{w_1[y]}\ r_2[y]\ \underline{w_1[x]}\ w_2[y]\ \underline{c_1}\ c_2$
conflits: $r_1[x]$ - $w_2[x]$, $w_2[x]$ - $w_1[x]$, $w_1[y]$ - $r_2[y]$, $w_1[y]$ - $w_2[y]$

H_2: $\underline{r_1[x]}\ \underline{w_1[y]}\ w_2[x]\ \underline{w_1[x]}\ \underline{c_1}\ r_2[y]\ w_2[y]\ c_2$
conflits: $r_1[x]$ - $w_2[x]$, $w_2[x]$ - $w_1[x]$, $w_1[y]$ - $r_2[y]$, $w_1[y]$ - $w_2[y]$
$\Longrightarrow H_2 \equiv H_1$

H_3: $w_2[x]\ r_2[y]\ \underline{r_1[x]}\ w_2[y]\ \underline{w_1[y]}\ c_2\ \underline{w_1[x]}\ \underline{c_1}$
conflits: $w_2[x]$ - $r_1[x]$, $w_2[x]$ - $w_1[x]$, $r_2[y]$ - $w_1[y]$, $w_2[y]$ - $w_1[y]$
$\Longrightarrow H_3 \not\equiv H_1$

H_4: $w_2[x]\ r_2[y]\ w_2[y]\ c_2\ \underline{r_1[x]}\ \underline{w_1[y]}\ \underline{w_1[x]}\ \underline{c_1}$ (histoire sériale)
conflits: $w_2[x]$ - $r_1[x]$, $w_2[x]$ - $w_1[x]$, $r_2[y]$ - $w_1[y]$, $w_2[y]$ - $w_1[y]$
$\Longrightarrow H_4 \equiv H_3 \Longrightarrow H_3$ sérialisable

Théorème de sérialisabilité

Graphe de sérialisation d'une exécution H: SG(H)

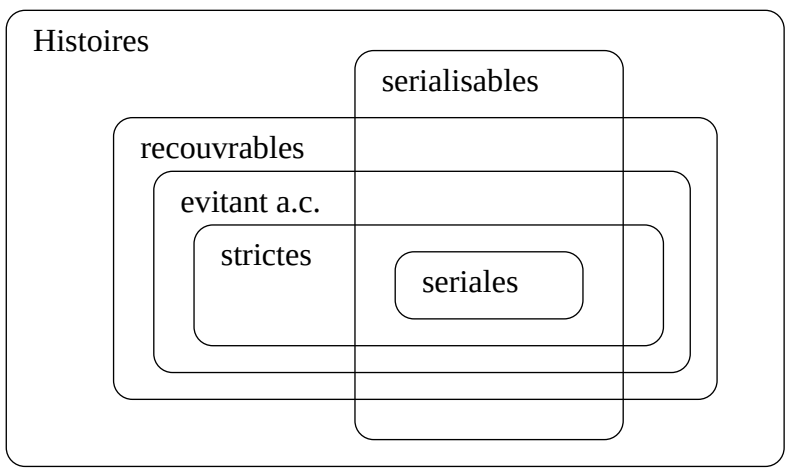
- *noeuds*: transactions T_i validées dans H
- *arcs*: si p et q conflictuelles, $p \in T_i$, $q \in T_j$, p avant q
  $\Rightarrow$ arc $T_i \rightarrow T_j$

Théorème: H sérialisable $\Longleftrightarrow$ SG(H) acyclique

H_1, H_2: $T_1 \overrightarrow{\longleftarrow} T_2$ H_3, H_4: $T_1 \longleftarrow T_2$

Propriétés de recouvrabilité

- à tout moment une transaction peut être annulée (panne)
- propriétés: strict $\Rightarrow$ pas d'annulation en cascade $\Rightarrow$ recouvrable
- *la sérialisabilité* dépend de l'ordre des opérations
  la recouvrabilité dépend de l'ordre des Commit/Abort
  $\Rightarrow$ les deux propriétés sont orthogonales



Exemple

T_1: $r_1[x]$ $w_1[y]$ $r_1[z]$ $w_1[z]$ c_1 T_2: $r_2[x]$ $w_2[y]$ $r_2[z]$ $w_2[z]$ c_2

H: $\underline{r_1[x]}$ $r_2[x]$ $\underline{w_1[y]}$ $\underline{r_1[z]}$ $\underline{w_1[z]}$ $w_2[y]$ $r_2[z]$ $w_2[z]$
conflits: $w_1[y]$ - $w_2[y]$, $r_1[z]$ - $w_2[z]$, $w_1[z]$ - $r_2[z]$, $w_1[z]$ - $w_2[z]$
$\Longrightarrow$ H est sérializable pour n'importe quelle position de c_1 et de c_2

H_1: $\underline{r_1[x]}$ $r_2[x]$ $\underline{w_1[y]}$ $\underline{r_1[z]}$ $\underline{w_1[z]}$ $w_2[y]$ $r_2[z]$ $w_2[z]$ c_2 $\underline{c_1}$
H_1 n'est pas recouvrable (T_2 lit z de T_1 et c_2 après c_1)

H_2: $\underline{r_1[x]}$ $r_2[x]$ $\underline{w_1[y]}$ $\underline{r_1[z]}$ $\underline{w_1[z]}$ $w_2[y]$ $r_2[z]$ $\underline{c_1}$ $w_2[z]$ c_2
H_2 n'évite pas les annulations en cascade (T_2 lit z de T_1 avant c_1)

H_3: $\underline{r_1[x]}$ $r_2[x]$ $\underline{w_1[y]}$ $\underline{r_1[z]}$ $\underline{w_1[z]}$ $w_2[y]$ $\underline{c_1}$ $r_2[z]$ $w_2[z]$ c_2
H_3 n'est pas stricte (T_1 écrit y et ensuite T_2 écrit y avant c_1)

Remarques

- *Équivalence*: la définition basée sur les conflits est suffisante et facile à utiliser, mais pas nécessaire

  Ex. H_1: $w_1[x]\ w_2[x]\ w_3[x]$ H_2: $w_2[x]\ w_1[x]\ w_3[x]$
  $H_1 \not\equiv H_2$ dans le sens des conflits $H_1 \equiv H_2$ dans le sens général

- *Conflit* = l'inverse de la commutativité
  $\Longrightarrow$ extensible à d'autres opérations que Read et Write

	Read	Write	Incr	Decr
Read	oui	non	non	non
Write	non	non	non	non
Incr	non	non	oui	oui
Decr	non	non	oui	oui

2. Contrôle par verrouillage à deux phases

Principe

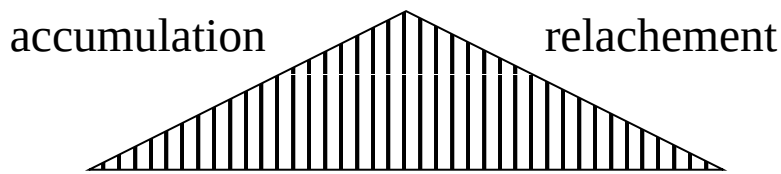
- stratégie pessimiste : retardement des oprérations qui peuvent produire des problèmes de concurrence
- blocage des opérations en attente de verrous
  - verrou pour chaque enregistrement
  - chaque verrou est donné à une seule transaction à la fois
- en pratique: verrou de lecture + verrou d'écriture

Algorithme de base

1) L'ordonnanceur reçoit $p_i[x]$ et teste les verrous de x
   - si l'un des verrous de x est détenu par une opération en conflit avec $p_i[x]$, alors $p_i[x]$ est retardée
   - sinon, verrou accordé à $p_i[x]$ et envoi $p_i[x]$ au GD
2) Un verrou pour $p_i[x]$ n'est jamais relâché avant la confirmation de l'exécution par le GD
3) Une fois un verrou pour T_i relâché, T_i n'obtiendra plus aucun verrou

Remarques

- règle 3 $\Rightarrow$ 2 phases: accumulation et relâchement de verrous



- règle 3 $\Rightarrow$ les paires d'opérations conflictuelles de T_i et T_j s'exécutent toujours dans le même ordre
- règle 2 $\Rightarrow$ l'ordre du GD pour les opérations conflictuelles sur un enregistrement x est le même que celui de l'ordonnanceur

Exemples

a) Non-respect de la règle de relâchement des verrous

T_1: $r_1[x]$ $w_1[y]$ c_1 T_2: $r_2[y]$ $w_2[y]$ c_2
ordre de réception: $r_1[x]$ $r_2[y]$ $w_1[y]$ c_1 $w_2[y]$ c_2
H_1: r_1[x] r_2[y] ($ru_2[y]$) **w_1[y] c_1** ($ru_1[x]$ $wu_1[y]$) **w_2[y] c_2** ($wu_2[y]$)

- notation: $ru_i[x]$/$wu_i[x]$ relâchement du verrou de lecture/écriture pour x par T_i
- violation règle: $ru_2[y]$ suivi de demande de verrou pour $w_2[y]$
- $r_2[y]$ - $w_1[y]$, $w_1[y]$ - $w_2[y]$ $\Rightarrow$ H_1 non-sérialisable

b) Exécution correcte

H_2: r_1[x] r_2[y] w_2[y] c_2 (ru_2[y] wu_2[y]) **w_1[y] c_1** (ru_1[x] wu_1[y])

- w_1[y] retardée en attente du verrou sur y
- r_2[y] - w_1[y], w_2[y] - w_1[y] $\Rightarrow$ H_2 sérialisable

c) Une exécution sérialisable impossible par verrouillage

H: $r_j[x]$ $w_k[x]$ c_k $w_i[y]$ c_i $r_j[y]$ $w_j[z]$ c_j

- $r_j[x]$ - $w_k[x]$, $w_i[y]$ - $r_j[y]$ $\Rightarrow$ H sérialisable, équivalente à $T_i T_j T_k$
- T_j relâche le verrou de lecture sur x pour $w_k[x]$, mais a besoin ensuite d'un verrou pour $r_j[y]$
  $\Rightarrow$ H ne peut pas être produite par verrouillage à deux phases
- Conclusion: certaines exécutions sérialisables ne peuvent pas être produites par verrouillage à deux phases
- une séquence d'entrée déjà sérialisable peut être modifiée

Interblocage

Exemple

T_1: $r_1[x]$ $w_1[y]$ c_1
T_2: $w_2[y]$ $w_2[x]$ c_2

ordre de réception: $r_1[x]$ $w_2[y]$ $w_2[x]$ $w_1[y]$

- T_1 obtient verrou pour $r_1[x]$, T_2 pour $w_2[y]$
- $w_2[x]$ attend $r_1[x]$, $w_1[y]$ attend $w_2[y]$
  $\Rightarrow$ interblocage de T_1 et de T_2

Stratégies pour éviter l'interblocage

- timeout
  - rejet transaction non-terminée après une durée limite
  - problème: risque de rejet des transactions non-bloquées
  - paramétrage fin nécessaire pour la durée limite
- graphe d'attente
  - noeuds=transactions, arcs=attente de verrou
  - détection des cycles
  - annulation de la transaction la moins coûteuse
- risque: victime relancée et bloquée à nouveau; problème d'équité à l'annulation

3. Contrôle par estampillage

Estampilles

- valeurs d'un domaine totalement ordonné
- chaque transaction T_i: estampille unique e(T_i)
- en pratique: numéros générés par un compteur, ordre chronologique
- signification: l'inverse de la priorité

Algorithme de base

- algorithme optimiste (sans de retardement)
- règle: deux opérations en conflit doivent s'exécuter suivant l'ordre des estampilles

  si $p_i[x]$ et $q_j[x]$ en conflit, alors $p_i[x]$ avant $q_j[x] \Leftrightarrow e(T_i) < e(T_j)$
- théorème: cette stratégie génère des exécutions sérialisables
  - les paires d'opérations conflictuelles s'exécutent dans l'ordre des transactions (de leurs estampilles)
- si $p_i[x]$ arrive en retard:
  - rejetée et T_i annulée
  - T_i relancée avec une nouvelle estampille
- si $p_i[x]$ acceptée $\Rightarrow$ envoyé au GD après la fin des $q_j[x]$ conflictuelles

déjà envoyées

Exemple

- ordre de réception: $r_1[x]$ $w_2[x]$ $r_3[x]$ $r_2[x]$ $w_1[x]$
- hypothèse: estampille de T_i = i

$r_1[x]$ acceptée
$w_2[x]$ conflit avec $r_1[x]$, test: $2 > 1 \rightarrow$ acceptée
$r_3[x]$ conflit avec $w_2[x]$, test: $3 > 2 \rightarrow$ acceptée
$r_2[x]$ aucun conflit $\rightarrow$ acceptée
$w_1[x]$ conflit avec $r_2[x]$, test: $1 < 2 \rightarrow$ rejetée

Règle d'écriture de Thomas

Ordonnanceur par estampillage pour les écritures

- e(T_i) $<$ e(T_j), w_i[x] arrive après w_j[x] $\Rightarrow$ w_i[x] rejetée
- la séquence devait être w_i[x]-w_j[x], donc x contient déjà la bonne valeur
- règle pour ordonnanceur écriture/écriture
  - quand w_i[x] arrive, soit T_j avec l'estampille maximale à avoir écrit x
  - si e(T_i) $>$ e(T_j), alors w_i[x] acceptée, sinon ignorée
- aucun rejet

Contrôleur intégré

- sous-ordonnanceurs différents pour des conflits différents
- estampillage pour lecture/écriture;
  Thomas pour écriture/écriture
- $r_i[x]$: rejeté s'il existe $w_j[x]$ précédent avec $e(T_i)<e(T_j)$
- $w_i[x]$: rejeté s'il existe $r_j[x]$ précédent avec $e(T_i)<e(T_j)$
- $w_i[x]$ par rapport aux $w_j[x]$: règle de Thomas

$r_1[x]$ acceptée
$w_1[x]$ acceptée
$w_3[x]$ acceptée
$w_2[x]$ ignorée
$w_4[x]$ acceptée
$r_2[x]$ rejetée

4. Création/destruction d'objets

Problème des objets fantômes

- contrôle de concurrence dans les BD dynamiques
- cas typique: T_1 consulte tous les objets d'un certain type, T_2 crée un objet de ce type

Exemple

- Fichiers de comptes et de totaux par client
- T_1: compare la somme des comptes de "Dupont" avec le total pour "Dupont"
- T_2: ajoute un compte pour "Dupont" et met à jour le total

Read$_1$(Compte[5],Compte[8],Compte[14])	**100,300,600**
Insert$_2$(Compte[10,"Dupont", 500])	
Read$_2$(Total["Dupont"])	**1000**
Write$_2$(Total["Dupont"])	**1500**
Read$_1$(Total["Dupont"])	**1500**

- compatible avec le verrouillage à 2 phases, mais non-sérialisable

Solution

- pour consulter tous les objets d'un type: info de contrôle nécessaire
- verrouillage au niveau de l'info de contrôle

Verrouillage d'index

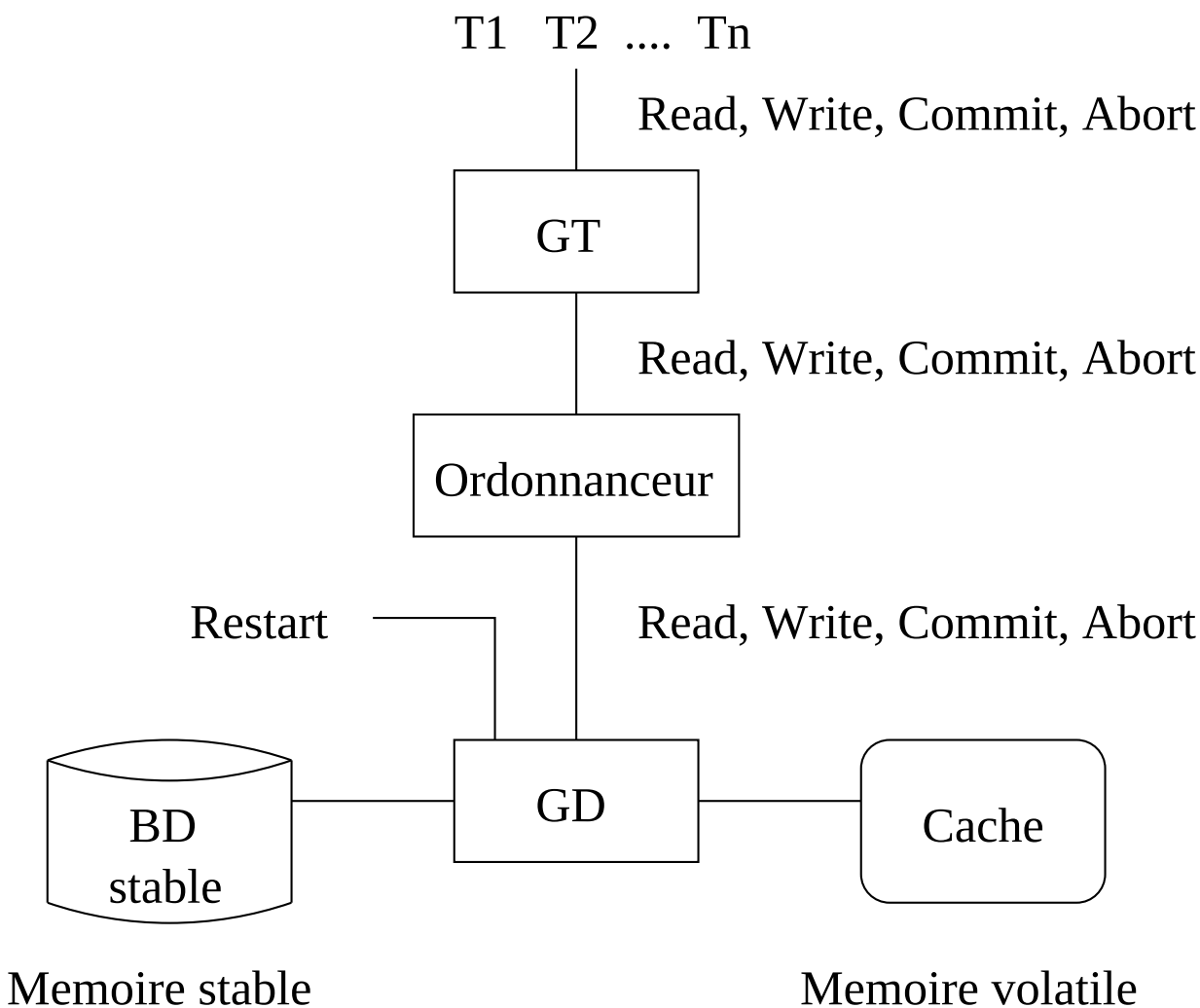
- entrée: valeur + liste pointeurs
- verrouillage entrée d'index
  - parcourir les enregistrements → lecture index
  - insérer un enregistrement → écriture index

III. TOLÉRANCE AUX PANNES

1. Problématique

Hypothèses

- pannes de système: contenu de la mémoire volatile perdu
- les pannes/erreurs sont toujours détectées
- ordonnanceur avec exécutions sérialisables et recouvrables
- même granularité pour l’ordonnanceur et le GD


T1 T2 .... Tn
Read, Write, Commit, Abort
GT
Read, Write, Commit, Abort
Ordonnanceur
Restart
Read, Write, Commit, Abort
BD
stable
GD
Cache
Memoire stable
Memoire volatile

Objectif

- dernière valeur validée de x: dernière valeur écrite en x par une transaction validée
- état validé de la BD: l'ensemble des dernières valeurs validées pour tous les enregistrements
- panne: mémoire volatile perdue

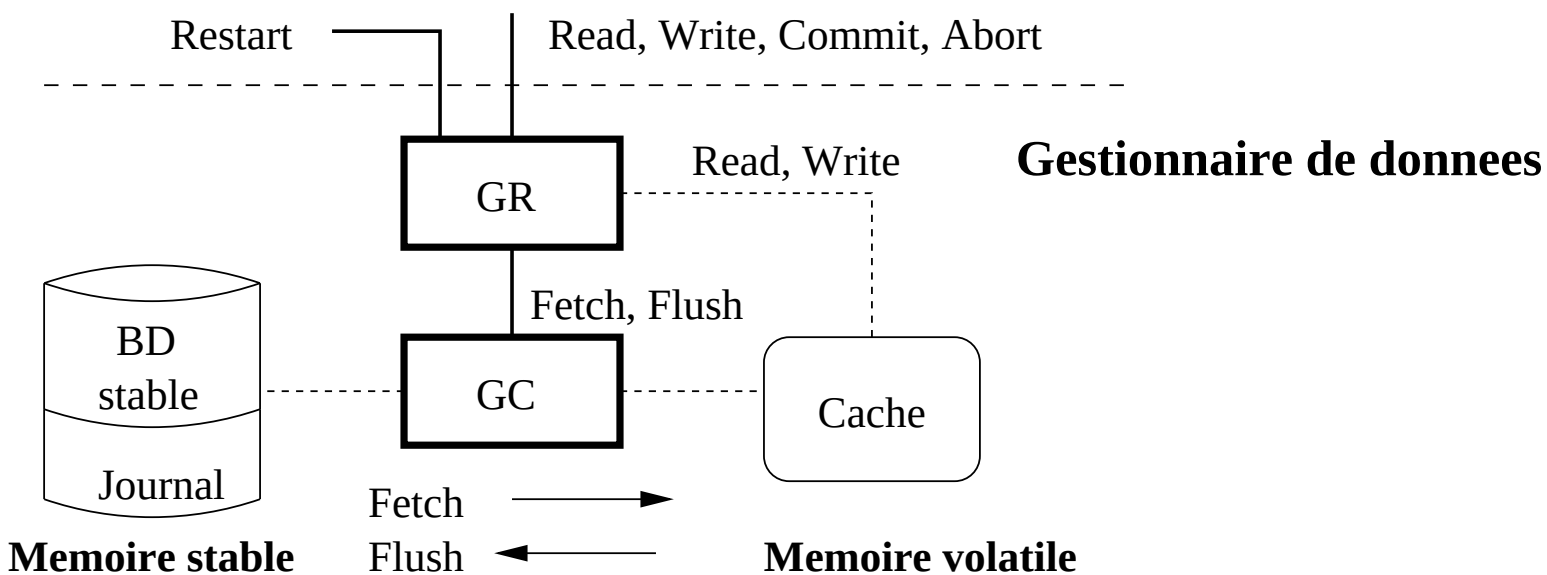
$\Longrightarrow$ *Restart* doit ramener la BD à l'état validé avant la panne

- problèmes
  - annuler l'effet des transactions non-validées
  - terminer les transactions validées
  - structures à garder en mémoire stable pour assurer la reprise

2. Architecture

Les composantes du Gestionnaire de données (GD)

- **Gestionnaire du Cache** (GC): gère les deux mémoires
- **Gestionnaire de reprise** (GR): opérations BD + Restart



Gestionnaire du Cache

- utilisation de la mémoire volatile : rapidité
- idéal: copie de la toute la BD
- en réalité: caching, car taille mémoire volatile limitée

Cache

- zone de mémoire volatile divisée en *cellules*: 1 enreg./cellule
- en réalité, le Cache stocke des *pages disque*

Cache

nr. cel.	bit de consistance	valeur enreg.
1	1	"J. Smith"
2	0	2.24581
. . . . .		

Repertoire du cache

id. enreg.	nr. cel.
x	2
y	1
. . . . .	

Opérations

- Flush (c), c cellule

si c **inconsistante alors**
    copier c **sur disque**
    rendre c **consistante**
sinon rien;

- Fetch (x), x enregistrement

sélectionner c **cellule vide**
si toutes les cellules occupées alors
    vider une cellule c **avec Flush et l'utiliser comme cellule vide**
copier x **du disque en** c
rendre c **consistante**
mettre à jour le répertoire du cache avec (x, c)

- choix cellule vide: LRU, FIFO
- lecture x:
  - toujours à partir du cache
  - si x n'est pas dans le cache alors Fetch(x) d'abord
- écriture x:
  - soit c la cellule de x dans le Cache (allouée à ce moment-là si x n'y est pas déjà)
  - c modifiée, marquée inconsistante
  - Flush(c) décidé par GR, selon son algorithme

Gestionnaire de reprise

Opérations

- GR_Read (T_i, x)
- GR_Write (T_i, x, v)
- GR_Commit (T_i)
- GR_Abort (T_i)
- Restart

Hypothèse supplémentaire : ordonnanceur avec exécutions strictes
$\Rightarrow$ écritures validées dans l'ordre de Commit des T_i

3. Journalisation

Journal

- historique des écritures dans la mémoire stable
- journal physique: liste de [T_i, x, v]
  - préserve l'ordre des écritures: fichier séquentiel
- journal logique: opérations de plus haut-niveau

  Ex. insertion x dans R et mise-à-jour index
  - moins d'entrées, mais plus difficile à interpréter
- autres informations: listes de transactions actives, validées, annulées

Exemple: journal physique

$[T_1,x,2], [T_2,y,3], [T_1,z,1], [T_2,x,8], [T_3,y,5], [T_4,x,2], [T_3,z,6]$

c_1 a_2 c_4

liste_active={T_3}

liste_commit={T_1, T_4}

liste_abort={T_2}

Ramasse-miettes

- recyclage de l'espace utilisé par le journal
- règle:

  entrée [T_i, x, v] recyclée $\Leftrightarrow$

  - T_i annulée ou
  - T_i validée, mais une autre T_j validée a écrit x après T_i

4. Principes et techniques pour la reprise

Types de GR

- GR peut forcer ou non GC d'écrire des cellules du Cache sur disque
- GR qui demande annulation
  - permet aux transactions non-validées d'écrire sur disque
  - Restart doit annuler ces écritures (annulation)
- GR qui demande répétition
  - permet aux transactions de valider avant d'écrire sur disque
  - Restart doit refaire ces écritures (répétition)
- 4 catégories de GR (combinaisons annulation - répétition)

Règles défaire/refaire

- règles de journalisation, nécessaires pour que le GR puisse faire l'annulation/répétition
- Règle "défaire" (pour annulation): si x sur disque contient une valeur validée, celle-ci doit être journalisée avant d'être modifiée par une valeur non-validée
- Règle "refaire" (pour répétition): les écritures d'une transaction doivent être journalisées avant son Commit

Idempotence de Restart

- Restart peut interrompre toute opération, même Restart
- idempotence: Restart interrompu et relancé donne le même résultat que le Restart complet

Checkpointing

- ajouter des informations sur disque en fonctionnement normal afin de réduire le travail de Restart
- techniques:
  - marquer dans le journal les écritures déjà réalisées/annulées dans la BD stable
  - écritures validées/annulées $\rightarrow$ BD stable

5. Algorithme annulation/répétition

Principes

- GR qui demande annulation et répétition: le plus complexe
- écrit les valeurs dans le Cache et ne demande pas de Flush
- avantages: flexibilité, minimise I/O

Opérations

- GR-Write (T_i, x, v)

liste_active = **liste_active** $\cup \{T_i\}$
si **x** n'est pas dans le cache alors allouer cellule pour **x**
journal = **journal** + [T_i, x, v]
cellule(**x**) = **v**
confirmer Write à l'ordonnanceur

- GR-Read (T_i, x)

si **x** n'est pas dans le cache alors **Fetch(x)**
retourner la valeur de *cellule*(**x**) à l'ordonnanceur

- GR-Commit (T_i)

liste_commit = **liste_commit** $\cup \{T_i\}$
confirmer le Commit à l'ordonnanceur
liste_active = **liste_active -** T_i

- GR-Abort (T_i)

pour chaque **x** écrit par T_i
    si **x** n'est pas dans le cache alors allouer cellule pour x
    cellule(**x**) = **image_avant(x, T_i)**
liste_abort = **liste_abort** $\cup \{T_i\}$
confirmer Abort à l'ordonnanceur
liste_active = **liste_active -** $\{T_i\}$

- Restart

marquer toutes les cellules comme vides
refait = {}, **annulé** = {}
pour chaque [T_i,x,v] ∈ **journal** (à partir de la fin) où **x** ∉ **annulé**∪**refait**
    si **x** n'est pas dans le cache alors allouer cellule pour **x**
    si T_i ∈**liste_commit** alors
        cellule(**x**)=v
        refait=**refait** ∪ {x}
    sinon
        cellule(**x**)= **image_avant(x, T_i)**
        annulé=**annulé** ∪ {x}
    si **refait** ∪ **annulé** = BD alors stop boucle
pour chaque T_i ∈ **list_commit**
    list_active=**list_active-**{T_i}
confirmer Restart à l'ordonnanceur

6. Autres algorithmes

Algorithme annulation/sans-répétition

- GR ne demande jamais répétition
- enregistre écritures avant le Commit
- GR-Write, GR-Read, GR-Abort pareil
- GR-Commit pareil, mais d'abord:
  - pour chaque x écrit par T_i, si $x \in$ Cache alors Flush(x)
- Restart pareil, sauf que "refait" n'existe pas

Algorithme sans-annulation/répétition

- GR ne demande jamais annulation
- écritures des T_i non-validées retardées jusqu'après Commit
- GR-Write: ajoute juste [T_i, x, v] au journal
- GR-Read: si T_i a écrit x, lecture dans le journal
- GR-Commit: chaque x écrit par T_i est calculé à partir du journal et écrit dans le cache
- GR-Abort: juste ajoute T_i à liste_abort
- Restart: pareil, sauf que "annulé" n'existe pas

Algorithme sans-annulation/sans-répétition: les écritures de T_i réalisées sur disque en une seule opération atomique, au Commit

IV. Transactions dans les SGBD relationnels

Contrôle de concurrence

- basé sur le verrouillage à deux phases
- extension du verrouillage → *verrouillage hiérarchique*
- validation = COMMIT, annulation = ROLLBACK
- la norme ne prévoit pas le verrouillage *explicite*
  au niveau programmation (SQL): *4 niveaux d'isolation*

Verrouillage hiérarchique

Niveaux de granularité

- attribut $<$ enregistrement $<$ relation $<$ base de données
- chaque opération se fait au niveau approprié
- niveau élevé: gestion allégée, car moins de verrous
- niveau bas: plus de concurrence

Principe du verrouillage hiérarchique

- verrouillage descendant: implicite (par inclusion)
- verrouillage ascendant: pour réaliser une opération sur un enreg., on demande un *verrou d'intention* sur la relation

Types de verrous

- **S** (partagé, lecture), **X** (exclusif, écriture)
- **IS** (intention lecture), **IX** (intention écriture)
- **SIX** (lecture et intention d'écriture)

	X	SIX	IX	S	IS
X	o	o	o	o	o
SIX	o	o	o	o	n
IX	o	o	n	o	n
S	o	o	o	n	n
IS	o	n	n	n	n

Matrice de conflits entre verrous hiérarchiques

Niveaux d'isolation

Serialisable (sérialisable) $>$ *Repetable Read* (lecture renouvelable) $>$ *Read Commited* (lecture validée) $>$ *Read Uncomitted* (lecture non validée)

- seul *Serialisable* garantit la sérialisabilité !
- utilisation: SET TRANSACTION *niveau*

- *Serialisable*: isolation totale et protection contre les objets fantômes
- *Repetable Read*: pas de protection contre les objets fantômes
- *Read Commited*: on voit les écritures validées d'autres transactions
  $\Rightarrow$ deux lectures de x dans T peuvent donner des résultats différents
- *Read Uncommited*: on voit les écritures non-validées d'autres transactions $\rightarrow$ annulations en cascade possibles

Reprise après panne

- respect des principes généraux
- journalisation physique avec points de contrôle (checkpointing)
- *point de contrôle*: écriture forcée du Cache sur disque
  $\rightarrow$ à la reprise on n'a pas besoin d'aller plus loin dans le journal pour refaire
- *point de sauvegarde* (savepoint): utile pour les transactions longues
  - *Savepoint* **s** = validation partielle d'une transaction
  - on peut faire une annulation partielle *Rollback to* **s**

Commit à deux phases

- validation de transactions réparties sur plusieurs sites
- chaque site gère ses propres ressources (données, journal)
- *première phase*: chaque site valide ses propres opérations
- *seconde phase*: le gestionnaire global valide l'ensemble de la transaction
- la validation globale $\rightarrow$ seulement si chaque site valide
- *Rollback* d'un site $\Rightarrow$ *Rollback* de l'ensemble
- le gestionnaire annonce chaque site si la transaction doit être validée ou annulée