

Systèmes de gestion de bases de données

NFP 107 et 107J

<http://cedric.cnam.fr/vertigo/Cours/BDB>

B. Amann, D. Gross-Amblard, P. Rigaux,
M. Scholl (NFP107J), D. Vodislav
(amann|dgram|vodislav|scholl)@cnam.fr

Équipe Vertigo
Laboratoire Cédric
Conservatoire national des arts & métiers, Paris, France
<http://cedric.cnam.fr/vertigo>

14 mars 2006

Systèmes de gestion de bases de données

NFP 107 et 107J

<http://cedric.cnam.fr/vertigo/Cours/BDB>

B. Amann, D. Gross-Amblard, P. Rigaux, M. Scholl (NFP107J),

D. Vodislav

(amann|dgram|vodislav|scholl)@cnam.fr

Équipe Vertigo
Laboratoire Cédric

Conservatoire national des arts & métiers, Paris, France

<http://cedric.cnam.fr/vertigo>

(Vertigo)

1 / 393

Plan du cours

8 Représentation physique des données dans Oracle275

9 Optimisation dans Oracle291

10 Optimisation - principes généraux et outils d'analyse292

11 Concurrence et reprise après panne 314

(Vertigo)

3 / 393

Plan du cours

1 Introduction 4

2 Le modèle relationnel31

3 Algèbre relationnelle44

4 SQL84

5 Organisation physique des données165

6 Optimisation212

7 Évaluation de requêtes 257

(Vertigo)

2 / 393

Introduction

Plan du cours

1 Introduction 4

2 Le modèle relationnel31

3 Algèbre relationnelle44

4 SQL84

5 Organisation physique des données165

6 Optimisation212

7 Évaluation de requêtes 257

(Vertigo)

4 / 393

Objectif du cours

COMPRENDRE et MAÎTRISER
la technologie des
BASES DE DONNÉES RELATIONNELLES

Bibliographie

Ouvrages en français

1. Carrez C., *Des Structures aux Bases de Données*, Masson
2. Gardarin G., *Maîtriser les Bases de Données: modèles et langages*, Eyrolles
3. Date C.J, *Introduction aux Bases de Données*, Vuibert, 970 Pages, Janvier 2001
4. Akoka J. et Comyn-Wattiau I., *Conception des Bases de Données Relationnelles*, Vuibert Informatique
5. Rigaux P., *Cours de Bases de Données*, <http://dept25.cnam.fr/BDA/DOC/cbd.pdf>

Bibliographie

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. H. Garcia - Molina, J.D. Ullman, J. Widom, *Database Systems : The Complete Book*, Hardcover, 2002
5. Garcia-Molina H., Ullman J. and Widom J., *Implementation of Database Systems*, Prentice Hall, 1999
6. Ullman J.D., *Principles of Database and Knowledge-Base Systems*, 2 volumes, Computer Science Press
7. Abiteboul S., Hull R., Vianu V., *Foundations of Databases*, Addison-Wesley

Bibliographie

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*
4. K. Loney, M. Theriault, *Oracle 9i DBA*
5. M. Abbey, M. J. Corey, *Oracle 9i Notions Fondamentales*
6. G. K. Vaidyanatna, K. Deshpande, J.A. Kostellac, *Optimisation des performances sous Oracle*
7. RevealNet, *Check-lists du DBA Oracle*

Bibliographie

SQL à la maison

1. MySQL, <http://www.mysql.org> (MS Windows, Linux)
 - ▶ Installation et interface Web via EasyPhp, <http://www.easyphp.org/>
2. PostgreSQL, <http://www.postgresql.org> (MS Windows, Linux)
 - ▶ Interface Web via PhpPgAdmin, <http://phppgadmin.sourceforge.net/>

Applications des bases de données

1. Applications “classiques” :

- ▶ Gestion de données : salaires, stocks, ...
- ▶ Transactionnel : comptes bancaires, centrales d'achat, réservations

2. Applications “modernes” :

- ▶ Documents électroniques : bibliothèques, journaux
- ▶ Web : commerce électronique, serveurs Web
- ▶ Génie logiciel : gestion de programmes, manuels, ...
- ▶ Documentation technique : plans, dessins, ...
- ▶ Bases de données spatiales : cartes routières, systèmes de guidage GPS, ...

Problème central : comment stocker et manipuler les données?

Une base de données est

- ▶ un **grand ensemble** de données
- ▶ **structurées** et
- ▶ mémorisées sur un support **permanent**

Un **système de gestion de bases de données (SGBD)** est

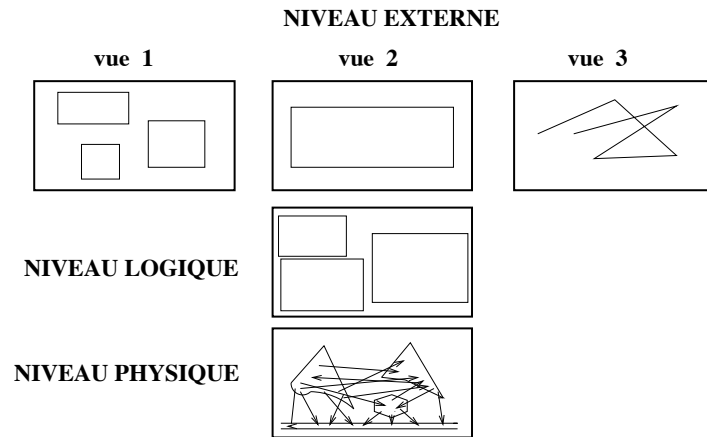
- ▶ un **logiciel** de **haut niveau d'abstraction** qui permet de manipuler ces informations

Diversité → Complexité

Diversité des utilisateurs, des interfaces et des architectures :

1. diversité des utilisateurs : administrateurs, programmeurs, non informaticiens, ...
2. diversité des interfaces : langages BD, menus, saisies, rapports, ...
3. diversité des architectures : client-serveur centralisé/distribué
Aujourd'hui : accès à plusieurs bases hétérogènes accessibles par réseau

Architecture d'un SGBD : ANSI-SPARC (1975)



(Vertigo)

13 / 393

Architecture 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)

(Vertigo)

14 / 393

Architecture d'un SGBD

NIVEAU LOGIQUE

- ▶ Définition de la structure des 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)

(Vertigo)

15 / 393

Architecture d'un SGBD

NIVEAU EXTERNE : Vues utilisateurs

Exemple : base(s) de données du CNAM :

1. **Vue de la planification des salles** : pour chaque cours
 - ▶ Noms des enseignants
 - ▶ Horaires et salles
2. **Vue de la paye** : pour chaque enseignant
 - ▶ Nom, prénom, adresse, indice, nombre d'heures
3. **Vue du service de scolarité** : pour chaque élève
 - ▶ Nom, prénom, adresse, no d'immatriculation, inscriptions aux cours, résultats

(Vertigo)

16 / 393

Intégration de ces vues

1. On laisse chaque usager avec sa vision du monde
2. Passage du **niveau externe/conceptuel** au **niveau logique**

On "intègre" l'ensemble de ces vues en une description **unique** :

SCHÉMA LOGIQUE

(Vertigo)

17 / 393

Interfaces d'un SGBD

- ▶ Outils d'aides à la conception de schémas
- ▶ Outils de saisie et d'impression
- ▶ Interfaces d'interrogation (alphanumérique/graphique)
- ▶ Environnement de programmation : intégration des langages SGBD (LDD, LR, LMD) avec un langage de programmation (C++, Java, Php, Cobol, ...)
- ▶ API standards : ODBC, JDBC
- ▶ Importation/exportation de données (ex. documents XML)
- ▶ Débogueurs
- ▶ Passerelles (réseaux) vers d'autres SGBD

(Vertigo)

18 / 393

En résumé

On veut **gérer un grand volume de données**

- ▶ **structurées** (ou semi-structurées),
- ▶ **persistantes** (stockées sur disque),
- ▶ **cohérentes**,
- ▶ **fiabiles** (protégées contre les pannes) et
- ▶ **partagées** entre utilisateurs et applications
- ▶ **indépendamment de leur organisation physique**

(Vertigo)

19 / 393

Modèles de données

Un modèle de données est caractérisé par :

- ▶ Une **STRUCTURATION** des objets
- ▶ Des **OPÉRATIONS** sur ces objets

(Vertigo)

20 / 393

Dans un SGBD, il existe plusieurs représentation plus ou moins abstraites de la même information :

- ▶ le modèle conceptuel → description conceptuelle des données
- ▶ le modèle logique → programmation
- ▶ le modèle physique → stockage

⇒ ces trois modèles correspondent aux trois niveaux ANSI d'un SGBD

Exemple d'un modèle conceptuel: Le modèle Entités-Associations (*entity-relationship model*, P. Chen, 1976)

- ▶ Modèle *très abstrait*, utilisé pour :
 - ▶ l'analyse du monde réel et
 - ▶ la communication entre les différents acteurs (utilisateurs, administrateurs, programmeurs ...) pendant
 - ▶ la conception de la base de données
- ▶ **Mais n'est pas associé à un langage concret.**

DONC UNE STRUCTURE
MAIS PAS
D'OPÉRATIONS

Modèle logique

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

Ces langages font abstraction du niveau physique du SGBD :

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**
Les structures et les langages sont plus simples ("logiques", déclaratifs), donc plus faciles pour l'utilisateur non expert
2. **INDÉPENDANCE PHYSIQUE**
On peut modifier l'implantation/la représentation physique sans modifier les programmes/l'application
3. **INDÉPENDANCE LOGIQUE**
On peut modifier les programmes/l'application sans toucher à la représentation physique des données

Historique des modèles SGBD

À chaque génération correspond un modèle logique

< 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	RELATIONNEL ETENDU	nouvelles applications
	DATALOG (SGBD déductifs)	pas encore de marché
	ORIENTÉ-OBJET	navig. + déclaratif
Fin 90	XML	navig. + déclaratif

(Vertigo)

25 / 393

Exemples d'opérations

- ▶ *Insérer des information concernant un employé nommé Jean*
- ▶ *Augmenter le salaire de Jean de 10%*
- ▶ *Détruire l'information concernant 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*

(Vertigo)

26 / 393

Quels types d'opérations ?

4 types d'opérations :

1. **création** (ou **insertion**)
2. **modification** (ou **mise-à-jour**)
3. **destruction**
4. **recherche** (requêtes)

Ces opérations correspondent à des commandes du LMD et du LR. La plus complexe est la **recherche** (LR) en raison de la variété des critères

(Vertigo)

27 / 393

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)

(Vertigo)

28 / 393

Concurrency d'accès

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

- ▶ Gérer les conflits si les utilisateurs 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
- ▶ Donner une image cohérente des données si un utilisateur effectue des requêtes et un autre des mises-à-jour

But : éviter les blocages, tout en empêchant des modifications anarchiques

La gestion du SGBD

- ▶ **Le concepteur**
 - ▶ évalue les besoins de l'application
 - ▶ conçoit le schéma logique de la base
- ▶ **L'administrateur du SGBD**
 - ▶ installe le système et crée la base
 - ▶ conçoit le schéma physique
 - ▶ fait des réglages fins (*tuning*)
 - ▶ gère avec le concepteur l'évolution de la base (nouveaux besoins, utilisateurs)
- ▶ **L'éditeur du SGBD**
 - ▶ fournit le système et les outils

Plan du cours

1 Introduction	4
2 Le modèle relationnel.....	31
3 Algèbre relationnelle.....	44
4 SQL.....	84
5 Organisation physique des données.....	165
6 Optimisation.....	212
7 Évaluation de requêtes.....	257

Exemple de relation

<i>Nom de la Relation</i>	<i>Nom d'Attribut</i>		
VEHICULE	Propriétaire	Type	Annee
	Loic	Espace	1988
	Nadia	Espace	1989
	Loic	R5	1978
	Julien	R25	1989
	Marie	ZX	1993

n-uplet

Exemple de schéma conceptuel

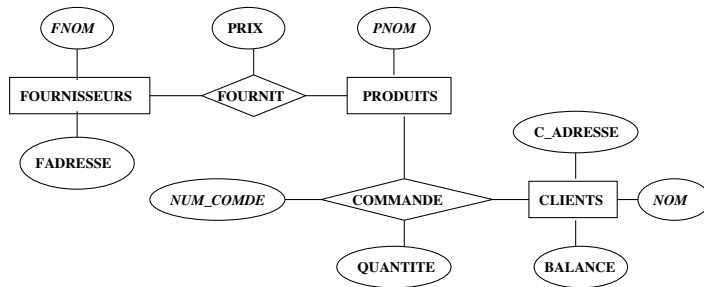


Figure: Schéma entités-associations (E/A)

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

Traduction en schéma relationnel

- ▶ Le schéma conceptuel entités-associations est traduit en une ou plusieurs tables relationnelles
- ▶ Voir le cours du cycle préparatoire CNAM (<http://dept25.cnam.fr/BDA/DOC/cbd.pdf>) pour les méthodes de traductions

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

Domaines, n -uplets et relations

- ▶ Un **domaine** est un *ensemble de valeurs*.
Exemples : $\{0, 1\}$, \mathbb{N} , l'ensemble des chaînes de caractères, l'ensemble des chaînes de caractères de longueur 10.
- ▶ Un **n -uplet** est une *liste de valeurs* $[v_1, \dots, v_n]$ où chaque valeur v_i est la valeur d'un domaine D_i : $v_i \in D_i$
- ▶ Le **produit cartésien** $D_1 \times \dots \times D_n$ entre des domaines D_1, \dots, D_n est l'*ensemble de tous les n -uplets* $[v_1, \dots, v_n]$ où $v_i \in D_i$.
- ▶ Une **relation** R est un *un sous-ensemble fini* d'un produit cartésien $D_1 \times \dots \times D_n$: R est un ensemble de n -uplets.
- ▶ Une **base de données** est un *ensemble de relations*.

(Vertigo)

37 / 393

Schéma d'une base de données

- ▶ 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 :

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

ou, plus simplement :

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

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

- ▶ Le **schéma d'une base de données** est l'ensemble des schémas de ses relations.

(Vertigo)

39 / 393

Attributs

Une **relation** $R \subset D_1 \times \dots \times D_n$ est représentée sous forme d'une **table** où chaque ligne correspond à un élément de l'ensemble R (un n -uplet) :

- ▶ L'ordre des **lignes** n'a pas d'importance (ensemble).
- ▶ Les **colonnes** sont distinguées par leur ordre ou par un nom d'**attribut**. Soit A_i le i -ème attribut de R :
 - ▶ n est appelé l'**arité** de la relation R .
 - ▶ D_i est appelé le domaine de A_i .
 - ▶ Tous les attributs d'une relation ont un nom différent.
 - ▶ Un même nom d'attribut peut apparaître dans différentes relations.
 - ▶ Plusieurs attributs de la même relation peuvent avoir le même domaine.

(Vertigo)

38 / 393

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)

(Vertigo)

40 / 393

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
 - ▶ définition des contraintes (clés, références, ...)
- ▶ **Langage de manipulation des données**
 - ▶ saisie des n -uplets d'une relation
 - ▶ affichage d'une relation
 - ▶ modification d'une relation : insertion, suppression et maj des n -uplets
 - ▶ **requêtes** : consultation d'une ou de plusieurs relations
- ▶ **Gestion des transactions**
- ▶ **Gestion des vues**

(Vertigo)

41 / 393

Langages de requêtes relationnels

Pouvoir d'expression : Qu'est-ce qu'on peut calculer ? Quelles opérations peut-on faire ?

Fondements théoriques :

- ▶ calcul relationnel
 - ▶ logique du premier ordre, très étudiée (théorèmes)
 - ▶ langage déclaratif : on indique les propriétés que doivent vérifier les réponses à la requête
 - ▶ **on n'indique pas comment** les trouver
 - ▶ facile pour un utilisateur (logicien ...)
- ▶ algèbre relationnelle
 - ▶ langage procédural, évaluateur facile à programmer
 - ▶ **on indique comment** trouver le résultat
 - ▶ difficile pour un utilisateur
- ▶ Théorème : ces deux langages ont le même pouvoir d'expression

(Vertigo)

42 / 393

Langages de requêtes relationnels

En pratique, langage SQL :

- ▶ Langage déclaratif
- ▶ Plus naturel que logique du premier ordre
 - ▶ **facile pour tout utilisateur**
- ▶ Traduction automatique en algèbre relationnelle
- ▶ Évaluation de la requête à partir de l'algèbre
 - ▶ **évaluation facile à programmer**

(Vertigo)

43 / 393

Plan du cours

1 Introduction	4
2 Le modèle relationnel.....	31
3 Algèbre relationnelle.....	44
4 SQL.....	84
5 Organisation physique des données.....	165
6 Optimisation.....	212
7 Évaluation de requêtes	257

(Vertigo)

44 / 393

Algèbre relationnelle

Opérations “relationnelles” (ensemblistes) :

- ▶ une opération prend en entrée une ou deux relations (ensembles de n -uplets) de la base de données
- ▶ le résultat est **toujours** une relation (un ensemble)

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**, **intersection** et **division**

Projection

Projection sur une partie (un sous-ensemble) des attributs d’une relation R .

Notation :

$$\pi_{A_1, A_2, \dots, A_k}(R)$$

A_1, A_2, \dots, A_k sont des attributs (du schéma) de la relation R . La projection “élimine” tous les autres attributs (colonnes) de R .

Projection: Exemples

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

→	R	<table border="1" style="border-collapse: collapse; text-align: center;"><thead><tr><th>A</th><th>B</th><th>C</th></tr></thead><tbody><tr><td>a</td><td>b</td><td>c</td></tr><tr><td>d</td><td>a</td><td>b</td></tr><tr><td>c</td><td>b</td><td>d</td></tr><tr><td>a</td><td>b</td><td>e</td></tr><tr><td>e</td><td>e</td><td>a</td></tr></tbody></table>	A	B	C	a	b	c	d	a	b	c	b	d	a	b	e	e	e	a
A	B	C																		
a	b	c																		
d	a	b																		
c	b	d																		
a	b	e																		
e	e	a																		

 \Rightarrow

$\pi_{A,B}(R)$	<table border="1" style="border-collapse: collapse; text-align: center;"><thead><tr><th>A</th><th>B</th></tr></thead><tbody><tr><td>a</td><td>b</td></tr><tr><td>d</td><td>a</td></tr><tr><td>c</td><td>b</td></tr><tr><td>e</td><td>e</td></tr></tbody></table>	A	B	a	b	d	a	c	b	e	e
A	B										
a	b										
d	a										
c	b										
e	e										

Le résultat est une relation (un ensemble) : le n -uplet (a, b) n’apparaît qu’une **seule** fois dans la relation $\pi_{A,B}(R)$, bien qu’il existe **deux** n -uplets (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	<table border="1" style="border-collapse: collapse; text-align: center;"><thead><tr><th>A</th><th>B</th><th>C</th></tr></thead><tbody><tr><td>a</td><td>b</td><td>c</td></tr><tr><td>d</td><td>a</td><td>b</td></tr><tr><td>c</td><td>b</td><td>d</td></tr><tr><td>a</td><td>b</td><td>e</td></tr><tr><td>e</td><td>e</td><td>a</td></tr></tbody></table>	A	B	C	a	b	c	d	a	b	c	b	d	a	b	e	e	e	a
A	B	C																		
a	b	c																		
d	a	b																		
c	b	d																		
a	b	e																		
e	e	a																		

 \Rightarrow

$\pi_{A,C}(R)$	<table border="1" style="border-collapse: collapse; text-align: center;"><thead><tr><th>A</th><th>C</th></tr></thead><tbody><tr><td>a</td><td>c</td></tr><tr><td>d</td><td>b</td></tr><tr><td>c</td><td>d</td></tr><tr><td>a</td><td>e</td></tr><tr><td>e</td><td>a</td></tr></tbody></table>	A	C	a	c	d	b	c	d	a	e	e	a
A	C												
a	c												
d	b												
c	d												
a	e												
e	a												

Sélection

Sélection avec une condition C sur les attributs d'une relation R : on garde les n -uplets de R dont les attributs satisfont C .

NOTATION :

$$\sigma_C(R)$$

Sélection : exemples

a) On sélectionne les n -uplets 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 n -uplets 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

$\sigma_{(A="a" \vee B="a") \wedge C \leq 3}(R)$	A	B	C
	a	b	1
	d	a	2

Sélection : exemples

c) On sélectionne les n -uplets 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

$\sigma_{A=B}(R)$	A	B	C
	e	e	5

Condition de sélection

La condition C d'une sélection $\sigma_C(R)$ est une **formule logique** qui relie des termes de la forme $A_i\theta A_j$ et $A_i\theta a$ avec les connecteurs logiques **et** (\wedge) et **ou** (\vee) où

- ▶ A_i et A_j sont des attributs de la relation R ,
- ▶ a est un élément (une valeur) du domaine de A_i ,
- ▶ θ est un prédicat de comparaison ($=, <, \leq, >, \geq, \neq$).

(Vertigo)

53 / 393

Expressions (requêtes) de l'algèbre relationnelle

Fermeture :

- ▶ Le résultat d'une opération est à nouveau une **relation**
- ▶ Sur cette relation, on peut faire une **autre opération** de l'algèbre

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

(Vertigo)

54 / 393

Expressions de l'algèbre relationnelle

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

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

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

(Vertigo)

55 / 393

Produit cartésien

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

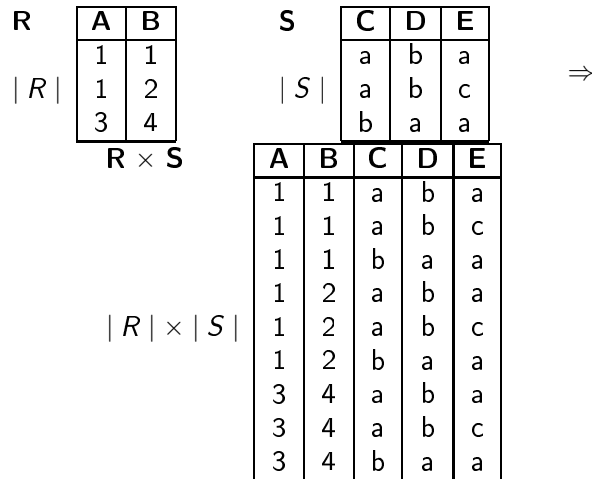
$$R(A_1, A_2, \dots, A_n) \quad S(B_1, B_2, \dots, B_k)$$

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

(Vertigo)

56 / 393

Exemple de produit cartésien

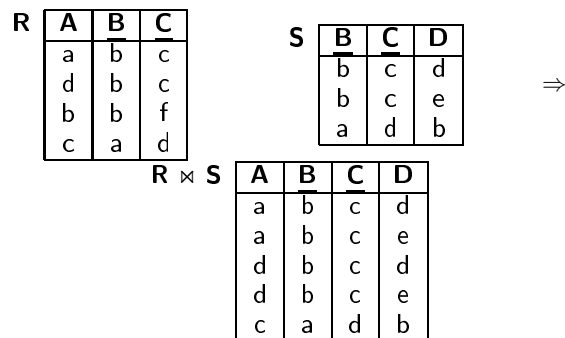


Jointure naturelle

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

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

Jointure naturelle: exemple



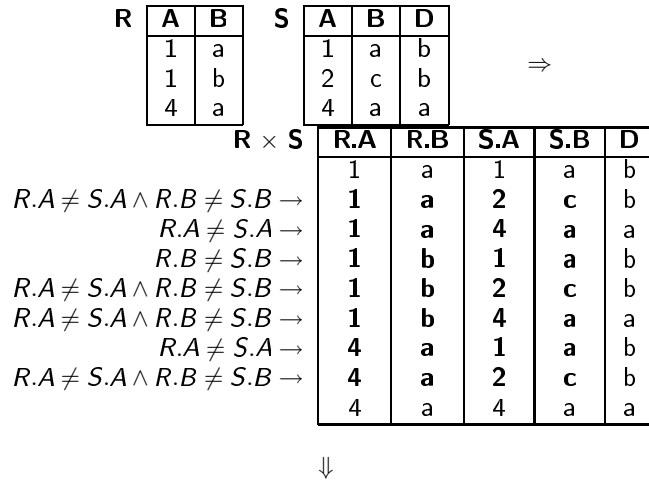
Jointure naturelle

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

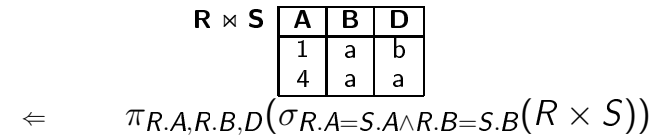
$$R \bowtie S = \pi_U(\sigma_{\forall A \in V: R.A=S.A}(R \times S))$$

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

Jointure naturelle : exemple



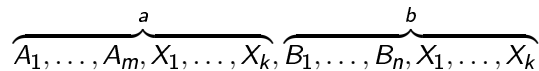
Jointure naturelle : exemple



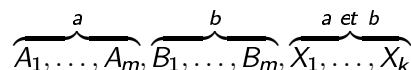
Jointure naturelle : algorithme de calcul

Pour chaque n -uplet a dans R et pour chaque n -uplet b dans S :

1. on concatène a et b et on obtient un n -uplet qui a pour attributs



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 n -uplet qui a pour attributs



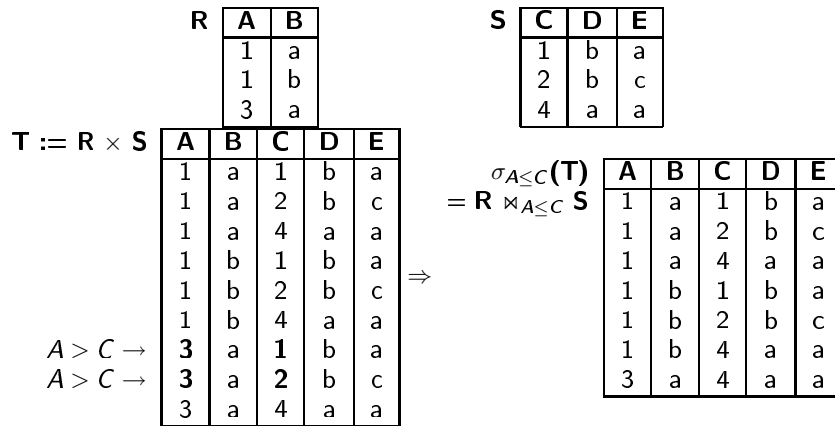
θ -Jointure

- ▶ ARGUMENTS : deux relations qui ne partagent pas d'attributs :

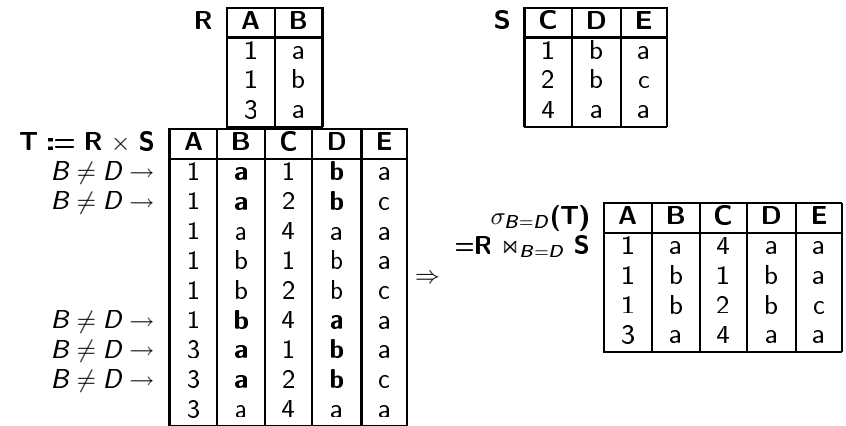
$$R(A_1, \dots, A_m) \quad S(B_1, \dots, B_n)$$

- ▶ NOTATION : $R \bowtie_{A_i \theta B_j} S$, $\theta \in \{=, \neq, <, \leq, >, \geq\}$
- ▶ SCHÉMA DE $T = R \bowtie_{A_i \theta B_j} S$: $T(A_1, \dots, A_m, B_1, \dots, B_n)$
- ▶ 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.

Exemple de θ -Jointure : $R \bowtie_{A \leq C} S$



Exemple d'équijointure : $R \bowtie_{B=D} S$



Utilisation de l'équijointure et jointure naturelle

$IMMEUBLE(ADI, NBETAGES, DATEC, PROP)$
 $APPIM(ADI, NAP, OCCUP, ETAGE)$

- Nom du propriétaire de l'immeuble où est situé l'appartement occupé par Durand :

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

- Appartements occupés par des propriétaires d'immeuble :

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

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 :

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

- ▶ NOTATION : $R \cup S$

- ▶ SCHÉMA DE $T = R \cup S$: $T(A_1, \dots, A_m)$

- ▶ VALEUR DE T : Union ensembliste sur $D_1 \times \dots \times D_m$:

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

(Vertigo)

69 / 393

Exemple d'union

R	A	B
	a	b
	a	c
	d	e

S	A	B
	a	b
	a	e
	d	e
	f	g

$$\Rightarrow R \cup S$$

	A	B
→	a	b
→	a	c
→	d	e
→	a	e
→	f	g

(Vertigo)

70 / 393

Différence

- ▶ ARGUMENTS : 2 relations de même schéma :

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

- ▶ NOTATION : $R - S$

- ▶ SCHÉMA DE $T = R - S$: $T(A_1, \dots, A_m)$

- ▶ VALEUR DE T : Différence ensembliste sur $D_1 \times \dots \times D_m$:

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

(Vertigo)

71 / 393

Exemple de différence

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

(Vertigo)

72 / 393

Intersection

- ▶ ARGUMENTS : 2 relations de même schéma :

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

- ▶ NOTATION : $R \cap S$

- ▶ SCHÉMA DE $T = R \cap S$: $T(A_1, \dots, A_m)$

- ▶ VALEUR DE T : Intersection ensembliste sur $D_1 \times \dots \times D_m$:

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

Exemple d'intersection

R	<table border="1"><tr><th>A</th><th>B</th></tr><tr><td>a</td><td>b</td></tr><tr><td>a</td><td>c</td></tr><tr><td>d</td><td>e</td></tr></table>	A	B	a	b	a	c	d	e
A	B								
a	b								
a	c								
d	e								
→									
→									

S	<table border="1"><tr><th>A</th><th>B</th></tr><tr><td>a</td><td>b</td></tr><tr><td>a</td><td>e</td></tr><tr><td>d</td><td>e</td></tr><tr><td>f</td><td>g</td></tr></table>	A	B	a	b	a	e	d	e	f	g
A	B										
a	b										
a	e										
d	e										
f	g										
→											
→											

R - S	<table border="1"><tr><th>A</th><th>B</th></tr><tr><td>a</td><td>c</td></tr></table>	A	B	a	c
A	B				
a	c				

$R \cap S = R - (R - S)$	<table border="1"><tr><th>A</th><th>B</th></tr><tr><td>a</td><td>b</td></tr><tr><td>d</td><td>e</td></tr></table>	A	B	a	b	d	e
A	B						
a	b						
d	e						

Semi-jointure

- ▶ ARGUMENTS : 2 relations quelconques :

$$R(A_1, \dots, A_m, X_1, \dots, X_k)$$

$$S(B_1, \dots, B_n, X_1, \dots, X_k)$$

où X_1, \dots, X_k sont les attributs en commun.

- ▶ NOTATION : $R \bowtie S$

- ▶ SCHÉMA DE $T = R \bowtie S$: $T(A_1, \dots, A_m, X_1, \dots, X_k)$

- ▶ VALEUR DE $T = R \bowtie S$: Projection sur les attributs de R de la jointure naturelle entre R et S .

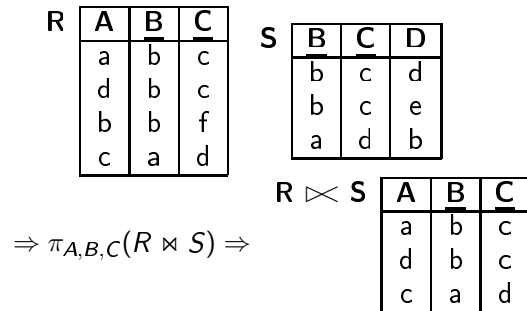
Semi-jointure

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

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

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

Exemple de semi-jointure

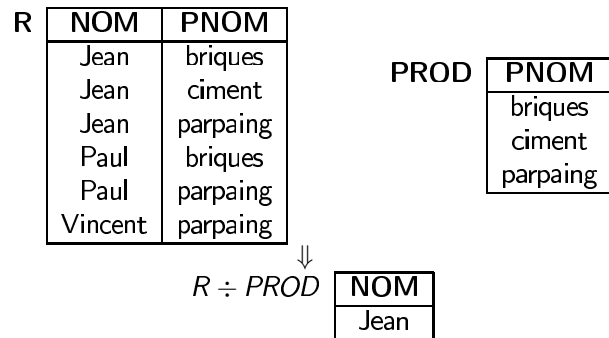


Exemple de division

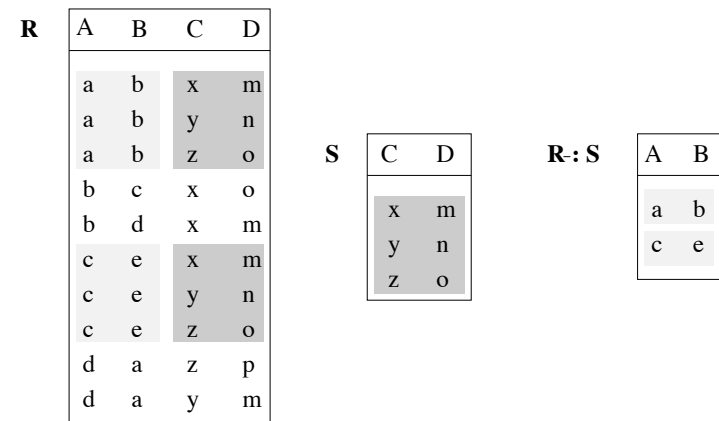
REQUÊTE : Clients qui commandent tous les produits:

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

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



Exemple de division



Division

- ▶ ARGUMENTS : 2 relations :

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

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

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

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

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

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

Renommage

- ▶ NOTATION : ρ
- ▶ ARGUMENTS : 1 relation :

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

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

Plan du cours

1 Introduction	4
2 Le modèle relationnel.....	31
3 Algèbre relationnelle.....	44
4 SQL.....	84
5 Organisation physique des données.....	165
6 Optimisation.....	212
7 Évaluation de requêtes	257

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.

(Vertigo)

85 / 393

Projection

SCHÉMA : **COMMANDES**(NUM,CNOM,PNOM,QUANTITE)

REQUÊTE : *Toutes les commandes*

ALGÈBRE : *COMMANDES*

SQL:

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

ou

```
SELECT *
FROM COMMANDES
```

(Vertigo)

86 / 393

Projection avec élimination de doublons

SCHÉMA : **COMMANDES**(NUM,CNOM,PNOM,QUANTITE)

REQUÊTE: *Produits commandés*

ALGÈBRE : $\pi_{PNOM}(COMMANDES)$

SQL :

```
SELECT PNOM
FROM COMMANDES
```

NOTE: Contrairement à l'algèbre relationnelle, SQL n'élimine pas les doublons (sémantique multi-ensemble). Pour les éliminer on utilise **DISTINCT** :

```
SELECT DISTINCT PNOM
FROM COMMANDES
```

Le **DISTINCT** peut être remplacé par la clause **UNIQUE**.

(Vertigo)

87 / 393

Sélection

SCHÉMA : **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'
```

(Vertigo)

88 / 393

REQUÊTE: *Produits commandés par Jean en quantité supérieure à 100*
 ALGÈBRE: $\pi_{PNOM}(\sigma_{CNOM='JEAN' \wedge QUANTITE > 100}(COMMANDES))$
 SQL:

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

(Vertigo)

89 / 393

Exemple

SCHÉMA : **FOURNITURE**(PNOM,FNOM,PRIX)
 REQUÊTE: *Produits dont le nom est celui du fournisseur*
 SQL:

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

(Vertigo)

91 / 393

Conditions simples

Les conditions de base sont exprimées de deux façons:

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

où *comparateur* est =, <, >, !=, ... ,

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

Exemple :

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

(Vertigo)

90 / 393

Appartenance à une **intervalle** : BETWEEN

SCHÉMA : **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 $x <= y$.

(Vertigo)

92 / 393

Chaînes de caractères : LIKE

SCHÉMA : **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 `_` (un caractère quelconque) et `%` (une chaîne de caractères quelconque). Pas exprimable avec l'algèbre relationnelle.

(Vertigo)

93 / 393

Valeurs inconnues : NULL

La valeur NULL est une valeur "spéciale" qui représente une *valeur (information) inconnue*.

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 $=, <, >, ! =, \dots$).
2. $A \text{ op } B$ est NULL si la valeur de A ou/et B est NULL (op est l'un de $+, -, *, /$).

(Vertigo)

94 / 393

Comparaison avec valeurs nulles

SCHÉMA et INSTANCE :

FOURNISSEUR	FNOM	VILLE
	Toto	Paris
	Lulu	NULL
	Marco	Marseille

REQUÊTE: *Les Fournisseurs de Paris.*

SQL:

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

RÉPONSE : Toto

(Vertigo)

95 / 393

Comparaison avec valeurs nulles

REQUÊTE: *Fournisseurs dont la ville est inconnue.*

SQL:

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

La réponse est vide. Pourquoi?

SQL:

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

RÉPONSE : Lulu

(Vertigo)

96 / 393

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

(Vertigo)

97 / 393

Jointures : exemple

SCHÉMA : **COMMANDES**(NUM,CNOM,PNOM,QUANTITE)
FOURNITURE(PNOM,FNOM,RIX)

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

ALGÈBRE :

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

(Vertigo)

99 / 393

Exemple

SCHÉMA : **EMPLOYE**(EMPNO,ENOM,DEPNO,SAL)

SQL:

```
SELECT ENOM
FROM EMPLOYE
WHERE SAL > 20000 OR SAL <= 20000
```

On ne trouve que les noms des employés avec un salaire connu. Pourquoi?

(Vertigo)

98 / 393

Jointure : exemple

SCHÉMA : **COMMANDES**(NUM,CNOM,PNOM,QUANTITE)
FOURNITURE(PNOM,FNOM,RIX)

SQL :

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

NOTES:

- ▶ On exprime une jointure comme un produit cartésien suivi d'une sélection et d'une projection (on a déjà vu ça?)
- ▶ Algèbre : la requête contient une jointure naturelle.
- ▶ SQL : il faut expliciter les attributs de jointure.

(Vertigo)

100 / 393

Auto-jointure et renommage

SCHÉMA : **FOURNISSEUR**(FNOM,STATUT,VILLE)
 REQUÊTE: "Couples" de fournisseurs situés dans la même ville
 SQL:

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

La deuxième condition permet

1. l'élimination des paires (x,x)
2. de garder un exemplaire parmi les couples symétriques (x,y) et (y,x)

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

(Vertigo)

101 / 393

Opérations de jointure

SQL2 introduit des opérations de jointure dans la clause FROM:

SQL2	opération	Algèbre
R1 CROSS JOIN R2	produit cartésien	$R1 \times R2$
R1 JOIN R2 ON R1.A < R2.B	théta-jointure	$R1 \bowtie_{R1.A < R2.B} R2$
R1 NATURAL JOIN R2	jointure naturelle	$R1 \bowtie R2$

(Vertigo)

103 / 393

Auto-jointure

SCHÉMA : **EMPLOYE**(EMPNO,ENOM,DEPNO,SAL)
 REQUÊTE: Nom et Salaire des Employés gagnant plus que l'employé de numéro 12546
 SQL:

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

- ▶ On confond souvent les auto-jointures avec des sélections simples.
- ▶ Requête en algèbre?

(Vertigo)

102 / 393

Jointure naturelle : exemple

SCHEMA: **EMP**(EMPNO,ENOM,DEPNO,SAL)
DEPT(DEPNO,DNOM)
 REQUÊTE: Numéros des départements avec les noms de leurs employés.
 SQL2:

```
SELECT DEPNO, ENOM
FROM   DEPT NATURAL JOIN EMP
```

Note : L'expression DEPT NATURAL JOIN EMP fait la jointure naturelle (sur les attributs en commun) et l'attribut DEPNO n'apparaît qu'une seule fois dans le schéma du résultat.

(Vertigo)

104 / 393

θ -jointure : exemple

REQUÊTE: Nom et salaire des employés gagnant plus que l'employé 12546
SQL2:

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

(Vertigo)

105 / 393

Jointure externe

Jointure externe : les n-uplets qui ne peuvent pas être joints *ne sont pas éliminés*.

- ▶ On garde tous les n-uplets des deux relations :

EMP NATURAL FULL OUTER JOIN DEPT

Tom	1	10000	Comm.
Jim	2	20000	Adm.
Karin	3	15000	NULL
NULL	4	NULL	Tech.

(Vertigo)

107 / 393

Jointure interne

EMP	EMPNO	DEPNO	SAL	DEPT	DEPNO	DNOM
	Tom	1	10000		1	Comm.
	Jim	2	20000		2	Adm.
	Karin	3	15000		4	Tech.

Jointure (interne) : les n-uplets qui ne peuvent pas être joints sont éliminés :

EMP NATURAL JOIN DEPT

Tom	1	10000	Comm.
Jim	2	20000	Adm.

(Vertigo)

106 / 393

- ▶ On garde tous les n-uplets de la première relation (gauche) :

EMP NATURAL LEFT OUTER JOIN DEPT

Tom	1	10000	Comm.
Jim	2	20000	Adm.
Karin	3	15000	NULL

- ▶ On peut aussi écrire (dans Oracle) :

```
select EMP.*, DEP.DNOM
from EMP, DEPT
where EMP.DEPNO = DEPT.DEPNO (+)
```

(Vertigo)

108 / 393

- ▶ On garde tous les n-uplets de la deuxième relation (droite) :

EMP NATURAL RIGHT OUTER JOIN DEPT

Tom	1	10000	Comm.
Jim	2	20000	Adm.
NULL	4	NULL	Tech.

- ▶ On peut aussi écrire (dans Oracle) :

```
select EMP.*, DEP.DNOM
from EMP, DEPT
where EMP.DEPNO (+) = DEPT.DEPNO
```

(Vertigo)

109 / 393

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:

$$\pi_{PNOM}(\sigma_{PRIX > 1000}(FOURNITURE)) \cup \pi_{PNOM}(\sigma_{CNOM = 'Jean'}(COMMANDES))$$

(Vertigo)

111 / 393

Jointures externes dans SQL2

- ▶ R1 NATURAL FULL OUTER JOIN R2 : Remplir R1.* et R2.*
- ▶ R1 NATURAL LEFT OUTER JOIN R2 : Remplir R2.*
- ▶ R1 NATURAL RIGHT OUTER JOIN R2 : Remplir R1.*

avec NULL quand nécessaire.

D'une manière similaire on peut définir des théta-jointures externes :

- ▶ R1 (FULL|LEFT|RIGHT) OUTER JOIN R2 ON prédicat

(Vertigo)

110 / 393

SQL:

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

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

(Vertigo)

112 / 393

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 doublés. Pour garder les doublés on utilise l'opération `EXCEPT ALL` : le résultat contient chaque n-uplet $a - b$ fois, où a et b est le nombre d'occurrences du n-uplet dans la première et la deuxième requête.

SQL:

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

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

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:

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

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

(Vertigo)

117 / 393

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:

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

(Vertigo)

118 / 393

SQL:

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

ou

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

(Vertigo)

119 / 393

Requêtes imbriquées plus complexes : ANY - ALL

SCHÉMA: **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')
```

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

(Vertigo)

120 / 393

“IN” et “= ANY”

COMMANDE(NUM,CNOM,PNOM,QUANTITE)

FOURNITURE(PNOM, FNOM, PRIX)

REQUÊTE: *Nom, prix 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.

(Vertigo)

121 / 393

“NOT IN” et “NOT = ALL”

EMPLOYE(EMPNO,ENOM,DEPTNO,SAL)

DEPARTEMENT(DEPTNO,DNOM,LOC)

REQUÊTE: *Départements sans employés*

SQL:

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

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

(Vertigo)

123 / 393

ALL

SCHÉMA: **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')
```

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

(Vertigo)

122 / 393

EXISTS

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)
```

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

(Vertigo)

124 / 393

NOT EXISTS

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)
```

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

(Vertigo)

125 / 393

Exemple : "EXISTS" et "= ANY"

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)
```

ou

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

(Vertigo)

127 / 393

Formes équivalentes de quantification

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

► La condition
 $x \theta$ ANY (SELECT Ri.y FROM R1, ... Rn WHERE p)
 est équivalente à
 EXISTS (SELECT * FROM R1, ... Rn WHERE p AND $x \theta$ Ri.y)

► La condition
 $x \theta$ ALL (SELECT Ri.y FROM R1, ... Rn WHERE p)
 est équivalente à
 NOT EXISTS (SELECT * FROM R1, ... Rn WHERE (p) AND NOT ($x \theta$ Ri.y))

(Vertigo)

126 / 393

Encore plus compliqué...

SCHÉMA: **FOURNITURE**(PNOM,FNOM,PRIX)

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

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

ou

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

(Vertigo)

128 / 393

Et la division?

FOURNITURE(FNUM,PNUM,QUANTITE)

PRODUIT(PNUM,PNOM,PRIX)

FOURNISSEUR(FNUM,FNOM,STATUS,VILLE)

REQUÊTE: *Noms des fournisseurs qui fournissent tous les produits*

ALGÈBRE:

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

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

(Vertigo)

129 / 393

COUNT, SUM, AVG, MIN, MAX

REQUÊTE: *Nombre de fournisseurs parisiens*

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

REQUÊTE: *Nombre de fournisseurs qui fournissent 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(DISTINCT ...).

(Vertigo)

131 / 393

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))
```

(Vertigo)

130 / 393

SUM et AVG

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

(Vertigo)

132 / 393

MIN et MAX

REQUÊTE: *Le prix des briques le moins chères.*

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

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

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

Comment peut-on faire sans MIN et MAX?

(Vertigo)

133 / 393

Requête imbriquée avec fonction de calcul

REQUÊTE: *Fournisseurs de briques dont le prix est en dessous du prix moyen*

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

(Vertigo)

134 / 393

GROUP BY

REQUÊTE: *Nombre de fournisseurs par ville*

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

VILLE	COUNT(FNOM)
PARIS	2
LYON	3

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

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

(Vertigo)

135 / 393

REQUÊTE: *Donner pour chaque produit son prix 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 regroupement sont directement indiquées dans la clause SELECT: le calcul de la moyenne se fait par produit obtenu au résultat après le regroupement.

(Vertigo)

136 / 393

HAVING

REQUÊTE: *Produits fournis par deux ou plusieurs fournisseurs avec un prix supérieur à 100 Euros*

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

(Vertigo)

137 / 393

REQUÊTE: *Nom et prix moyen des produits fournis par des fournisseurs Parisiens et dont le prix minimum est supérieur à 1000 Euros*

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

(Vertigo)

139 / 393

HAVING

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: on garde les produits dont le nombre des fournisseurs est ≥ 2 .

Des conditions de sélection peuvent être appliquées avant le calcul d'agrégat (clause WHERE) mais aussi après (clause HAVING).

(Vertigo)

138 / 393

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).

(Vertigo)

140 / 393

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).

SQL91 ou SQL2 Standard actuel:

- ▶ Requêtes dynamiques
- ▶ 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

(Vertigo)

141 / 393

Historique

SQL:1999 (SQL3) : SQL devient un langage de programmation :

- ▶ Extensions orientées-objet (héritage, méthodes)
- ▶ Types structurés
- ▶ BLOB, CLOB
- ▶ Opérateur de fermeture transitive (recursion)

(Vertigo)

142 / 393

Récursivité dans SQL

schéma **ENFANT**(NOMPARENT,NOMENFANT)

REQUÊTE: *Les enfants de Charlemagne*

SQL:

```
SELECT NOMENFANT
FROM ENFANT
WHERE NOMPARENT='Charlemagne';
```

(Vertigo)

143 / 393

Récursivité dans SQL

schéma **ENFANT**(NOMPARENT,NOMENFANT).

REQUÊTE: *Les enfants et petits-enfants de Charlemagne*

SQL:

```
(SELECT NOMENFANT
FROM ENFANT
WHERE NOMPARENT='Charlemagne')
```

```
UNION
```

```
(SELECT E2.NOMENFANT
FROM ENFANT E1,E2
WHERE E1.NOMPARENT='Charlemagne'
AND
E1.NOMENFANT=E2.NOMPARENT)
```

(Vertigo)

144 / 393

Descendants

schéma **ENFANT**(NOMPARENT,NOMENFANT).
 REQUÊTE: *Les descendants de Charlemagne*

- ▶ Nécessite un nombre a priori inconnu de jointures
- ▶ Th : impossible à exprimer en logique du premier ordre
- ▶ Th : donc, impossible en algèbre relationnel

En pratique, on étend le langage SQL avec des opérateurs récursifs

(Vertigo)

145 / 393

Création de tables

Une table (relation) est créée avec la commande **CREATE TABLE** :

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

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

- ▶ Pour chaque attribut, on indique le domaine (type)

(Vertigo)

147 / 393

Descendants

schéma **ENFANT**(NOMPARENT,NOMENFANT)
 REQUÊTE: *Les descendants de Charlemagne*
 SQL:

```
WITH RECURSIVE DESCENDANT(NOMANC, NOMDESC) AS
  (SELECT NOMPARENT, NOMENFANT FROM ENFANT)
  UNION
  (SELECT R1.NOMANC, R2.NOMDESC
   FROM DESCENDANT R1, DESCENDANT R2
   WHERE R1.NOMDESC=R2.NOMANC)
SELECT NOMDESC FROM DESCENDANT
WHERE NOMANC='Charlemagne';
```

(Vertigo)

146 / 393

Nombreux types : exemple d'Oracle 8i

decimal(p, s)	<p digits>,<s digits>
integer	nombre entier
real	nombre réel
char (size)	chaîne de caractère de <i>taille fixe</i>
varchar (size)	chaîne de caractère de <i>taille variable</i>
date,timestamp	horodatage
boolean	booléen
blob	données binaires de grande taille
etc.	

(Vertigo)

148 / 393

Contraintes d'intégrité

Pour une application donnée, pour un schéma relationnel donné, toutes les instances ne sont pas significatives

Exemple

- ▶ Champs important non renseigné : **autorisation des NULL**
- ▶ Prix négatifs : **contrainte d'intégrité sémantique**
- ▶ Code de produit dans une commande ne correspondant à aucun produit dans le catalogue : **contrainte d'intégrité référentielle**

(Vertigo)

149 / 393

Unicité des valeurs

Interdiction de deux valeurs identiques pour le même attribut :

```
CREATE TABLE Fourniture (pnom VARCHAR(20) UNIQUE,
                          fnom VARCHAR(20)
)
```

- ▶ UNIQUE et NOT NULL : l'attribut peut servir de clé primaire

(Vertigo)

151 / 393

Valeurs NULL

La valeur NULL peut être interdite :

```
CREATE TABLE Fourniture (pnom VARCHAR(20) NOT NULL,
                          fnom VARCHAR(20) NOT NULL
)
```

(Vertigo)

150 / 393

Ajout de contraintes référentielles : clés primaires

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

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

- ▶ L'attribut pnom est une clé dans la table Produit
- ▶ L'attribut fnom est une clé dans la table Fournisseur
- ▶ Une seule clé primaire par relation
- ▶ Une clé primaire peut être référencée par une autre relation

(Vertigo)

152 / 393

Ajout de contraintes référentielles : clés étrangères

La table Fourniture relie les produits à leurs fournisseurs :

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

- ▶ Les attributs pnom et fnom sont des clés étrangères (pnom et fnom existent dans les tables référencées)
- ▶ Pour sélectionner un attribut de nom différent :

```
FOREIGN KEY (pnom) REFERENCES Produit(autrenom)
```

(Vertigo)

153 / 393

Contraintes sémantiques

- ▶ Clause CHECK, suivie d'une condition

Exemple : prix positifs

```
prix INTEGER CHECK (prix>0)
```

- ▶ Condition générale : requête booléenne (dépend du SGBD)

(Vertigo)

155 / 393

Valeurs par défaut

```
CREATE TABLE Fournisseur(fnom VARCHAR(20),
                          ville VARCHAR(16) DEFAULT 'Carcassonne');
```

- ▶ Valeur utilisée lorsque l'attribut n'est pas renseigné
- ▶ Sans précision, la valeur par défaut est NULL

(Vertigo)

154 / 393

Destruction de tables

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

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

La table Fourniture **doit être détruite en premier** car elle contient des clés étrangères vers les deux autres tables;

(Vertigo)

156 / 393

Insertion de n-uplets

On insère dans une table avec la commande **INSERT** :

```
INSERT INTO R( $A_1, A_2, \dots, A_n$ ) VALUES ( $v_1, v_2, \dots, v_n$ )
```

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**

(Vertigo)

157 / 393

Modification

On modifie une table avec la commande **UPDATE** :

```
UPDATE R SET  $A_1 = v_1, A_2 = v_2, \dots, A_n = v_n$   
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.

(Vertigo)

159 / 393

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
```

(Vertigo)

158 / 393

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')
```

(Vertigo)

160 / 393

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**

(Vertigo)

161 / 393

Déclencheurs associés aux destructions de n -uplets

- ▶ Que faire lorsque le n -uplet référence une autre table ?

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

```
CREATE TABLE Fourniture (pnom VARCHAR(20) NOT NULL,
                          fnom VARCHAR(20) NOT NULL,
                          FOREIGN KEY (pnom) REFERENCES Produit
                          on delete <action>);
```

<action> à effectuer lors de la **destruction dans** Produit :

- ▶ **CASCADE**: destruction **si destruction dans** Produit
- ▶ **RESTRICT**: interdiction si existe dans Fourniture
- ▶ **SET NULL**: remplacer par NULL
- ▶ **SET DEFAULT <valeur>**: remplacement par une valeur par défaut

(Vertigo)

163 / 393

Destruction : exemples

Destruction des produits fournis par le BHV :

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

Destruction du fournisseur BHV :

```
DELETE FROM Fourniture
WHERE fnom = 'BHV'
```

(Vertigo)

162 / 393

Déclencheurs associés aux mise à jour de n -uplets

```
CREATE TABLE Fourniture (pnom VARCHAR(20) NOT NULL,
                          fnom VARCHAR(20) NOT NULL,
                          FOREIGN KEY (pnom) REFERENCES Produit
                          on update <action>);
```

<action> à effectuer lors d'un **changement de clé dans** Produit :

- ▶ **CASCADE**: propagation du changement de clé de Produit
- ▶ **RESTRICT**: interdiction si clé utilisée dans Fourniture
- ▶ **SET NULL**: remplace la clé dans Fourniture par NULL
- ▶ **SET DEFAULT <valeur>**: remplace la clé par une valeur par défaut

(Vertigo)

164 / 393

Plan du cours

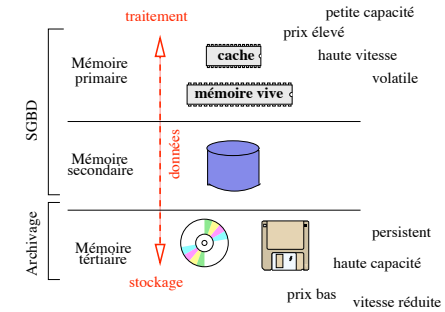
- 1 Introduction 4
- 2 Le modèle relationnel.....31
- 3 Algèbre relationnelle.....44
- 4 SQL.....84
- 5 Organisation physique des données165
- 6 Optimisation.....212
- 7 Évaluation de requêtes 257

Comparaison

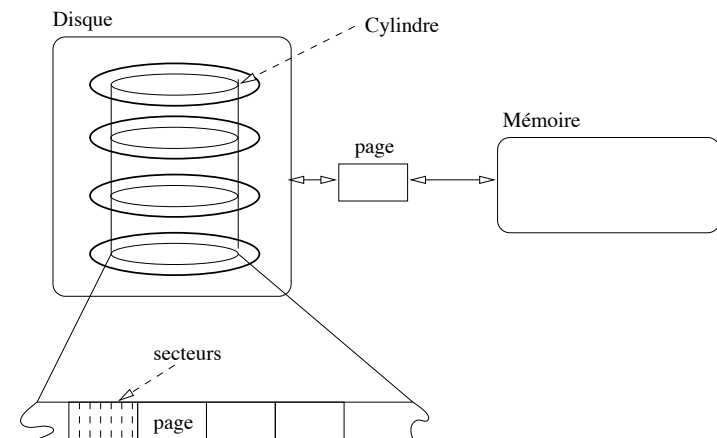
Mémoire	AZEN 256MB, SDRAM DIMM
taille	256MB
prix	90 euros
temps d'accès	8ns
euros/MB	$90E/256MB = 0.350 \text{ euros/MB}$
Disque	Western Digital 80GB 7200RPM
taille	80GB = 81920 MB
prix	160 euros
temps d'accès 9ms	9 000 000 ns
euros/MB	$160/81962 = 0.002 \text{ euros/MB}$

Stockage de données

Hierarchie de mémoire (vitesse d'accès, prix, capacité de stockage, volume) :



Architecture d'un disque



Organisation d'un disque

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

Exemple : $(38792 \text{ cylindres}) \times (16 \text{ blocs/cylindre}) \times (63 \text{ secteurs/bloc}) \times (512 \text{ octets/secteur}) = 20,020,396,000 \text{ octets} = 20 \text{ GO}$

Les articles

Un fichier est un ensemble **d'articles** (enregistrements, n-uplets) et un article est une séquence de **champs** (attributs).

1. **Articles en format fixe.**
 - 1.1 La **taille** de chaque champ est fixée.
 - 1.2 **Taille et nom** des champs dans le **descripteur** de fichier.
2. **Articles en format variable.**
 - 2.1 La **taille** de chaque champ est variable.
 - 2.2 **Taille** du champ dans son entête.

Les fichiers

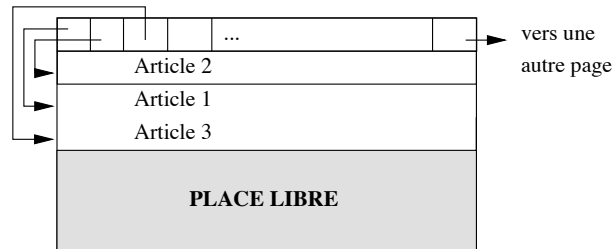
Les données sont stockées dans des **fichiers** :

- ▶ Un fichier occupe un ou plusieurs blocs (pages) 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.

Articles et pages

- ▶ Les articles sont stockés dans les pages (taille article < taille de page)
- ▶ 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



(Vertigo)

173 / 393

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 échangées)!

(Vertigo)

174 / 393

Organisation de fichiers

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

Il existe trois sortes d'organisation principales :

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

(Vertigo)

175 / 393

Exemple de référence

Organisation d'un fichier contenant les articles suivants :

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

(Vertigo)

176 / 393

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.

Fichiers séquentiels triés

Une première amélioration consiste à **trier** le fichier sur sa clé d'accès. On peut alors 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 : $\log_2(n)$.

Coût des opérations

Nombre moyen de lectures/écritures sur disque d'un fichier de n pages :

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

⇒ organisation utilisée pour les fichiers de petite taille.

Ajout d'un index

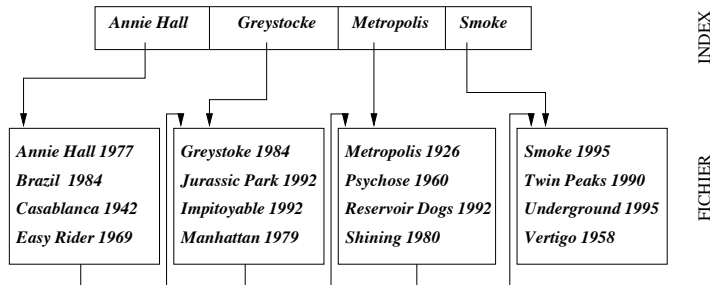
L'opération de recherche peut encore être améliorée en utilisant un **index** sur un fichier **trié**. Le(s) champ(s) sur le(s)quel(s) le fichier est trié est appelé **clé**.

Un index est un *second fichier* possédant les caractéristiques suivantes :

1. Les articles sont des couples (*valeurdecl*, *adressedepage*)
2. Une occurrence (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



(Vertigo)

181 / 393

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**

(Vertigo)

183 / 393

Recherche avec un index

Un index permet d'accélérer les recherches : 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.

(Vertigo)

182 / 393

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 a 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 . Plusieurs solutions, notamment : créer une **page de débordement**.

NB : il faut éventuellement réorganiser l'index.

(Vertigo)

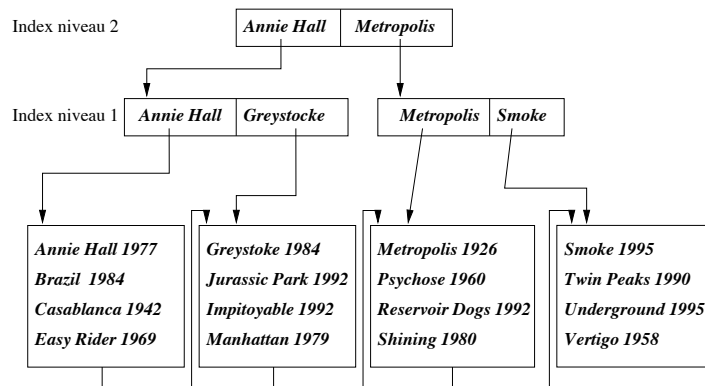
184 / 393

Autres opérations: destructions et mises-à-jour

Relativement facile en général :

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

⇒ on peut avoir à réorganiser le fichier et/ou l'index, ce qui peut être coûteux.



Séquentiel indexé: définition

Un index est un fichier qu'on peut à nouveau indexer :

1. On trie le fichier sur la clé.
2. On répartit les articles triés dans n pages, en laissant de la place libre dans chaque page.
3. On constitue un index à plusieurs niveaux sur la clé.

⇒ on obtient un **arbre** dont les feuilles constituent le fichier et les noeuds internes l'index.

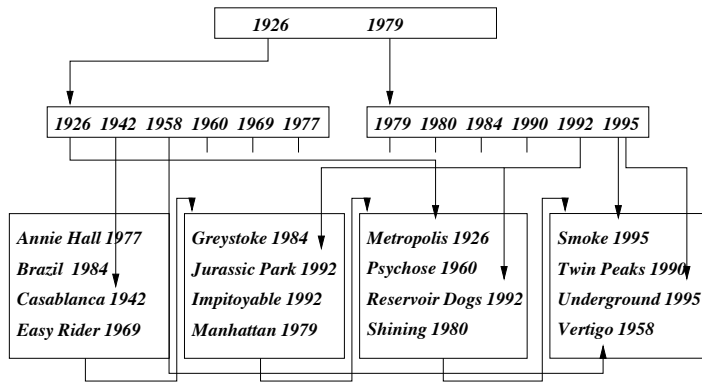
Index dense et index non dense

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



(Vertigo)

189 / 393

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).

(Vertigo)

191 / 393

Inconvénients du séquentiel indexé

Organisation bien adaptée aux fichiers qui évoluent peu. En cas de grossissement :

1. Une page est trop pleine → 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

⇒ on a besoin d'une structure permettant une réorganisation dynamique sans dégradation de performances.

(Vertigo)

190 / 393

Arbre-B : définition

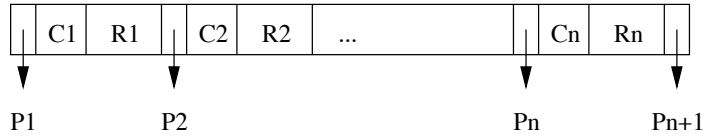
Un arbre-B d'ordre k est un arbre équilibré tel que :

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

(Vertigo)

192 / 393

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



Les C_i sont les clés des articles, les R_i représentent le reste des attributs d'un article de clé C_i . Les P_i sont les pointeurs vers les noeuds fils dans l'index. NB : $k \leq n \leq 2k$.

Tous les articles référencés par P_i ont une valeur de clé x entre C_{i-1} et C_i (pour $i = 1 : x \leq C_1$ et $i = n + 1 : x \geq C_n$)

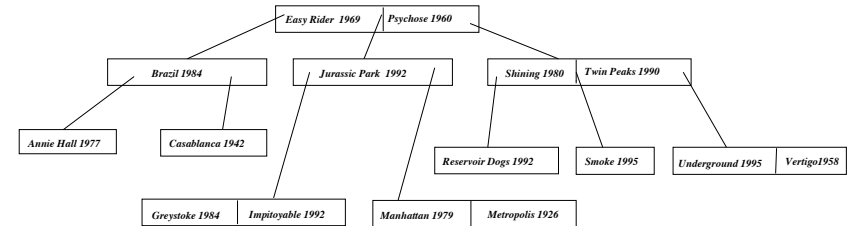
Recherche dans un arbre-B

Rechercher les articles de clé C . A partir de la racine, appliquer récursivement l'algorithme suivant :

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

1. Chercher C dans la page courante. Si C y est, accéder aux valeurs des autres champs, Fin.
2. Sinon, Si $C < C_1$ (ou $C > C_n$), on continue la recherche avec le noeud référencé par P_1 (ou P_{n+1}).
3. Sinon, il existe $i \in [1, k[$ tel que $C_i < C < C_{i+1}$, on continue avec la page référencée par le pointeur P_{i+1} .

Exemple d'un arbre-B



Insertion dans un arbre-B d'ordre k

On recherche la feuille de l'arbre où l'article doit prendre place et on l'y insère. Si la page p déborde (elle contient $2k + 1$ éléments) :

1. On alloue une nouvelle page p' .
2. On place les k 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.

Destruction dans un arbre-B d'ordre k

Chercher la page p contenant l'article. Si c'est une feuille :

1. On détruit l'article.
2. S'il reste au moins k articles dans p , c'est fini.
3. Sinon :
 - 3.1 Si une feuille "sœur" contient plus de k articles, on effectue une permutation pour rééquilibrer les feuilles. Ex : destruction de *Smoke*.
 - 3.2 Sinon on "descend" un article du père dans p , et on réorganise le père. Ex : destruction de *ReservoirDogs*

Quelques mesures pour l'arbre-B

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

$$\log_{2k+1}(n+1) \leq h \leq \log_{k+1}\left(\frac{n+1}{2}\right)$$

Exemple pour $k = 100$:

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

Les opérations d'accès coûtent au maximum h E/S (en moyenne $\geq h - 1$).

Réorganisation de l'arbre

Supposons maintenant qu'on détruit un article dans un noeud interne. Il faut réorganiser :

1. On détruit l'article
2. On le remplace par l'article qui a la plus grande valeur de clé dans le sous-arbre gauche. Ex : destruction de *Psychose*, remplacé par *Metropolis*
3. On vient de retirer un article dans une feuille : si elle contient moins de k éléments, on procède comme indiqué précédemment.

⇒ toute destruction a un effet seulement **local**.

Variante de l'arbre-B

1. L'arbre B contient à la fois l'index et le fichier indexé.
2. Si la taille d'un article est grande, chaque noeud en contient peu, ce qui augmente la hauteur de l'arbre
3. On peut alors séparer l'index (arbre B) du fichier : stocker les articles dans un fichier F , remplacer l'article R_i dans les pages de l'arbre par un pointeur vers l'article dans F .

L'arbre B+

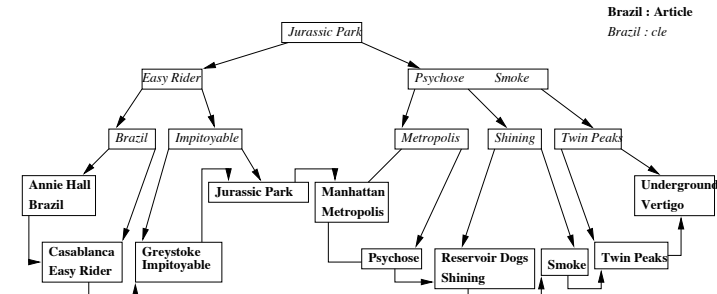
Inconvénient de l'arbre B (et de ses variantes):

- ▶ Les recherches sur des intervalles de valeurs sont complexes.

D'où l'arbre-B+ :

- ▶ Seules les feuilles de l'arbre pointent sur les articles du fichier.
- ▶ De plus ces feuilles sont chaînées entre elles.
- ▶ Variante: les feuilles contiennent les articles.

Exemple d'un arbre-B+



Hachage

Accès direct à la page contenant l'article recherché :

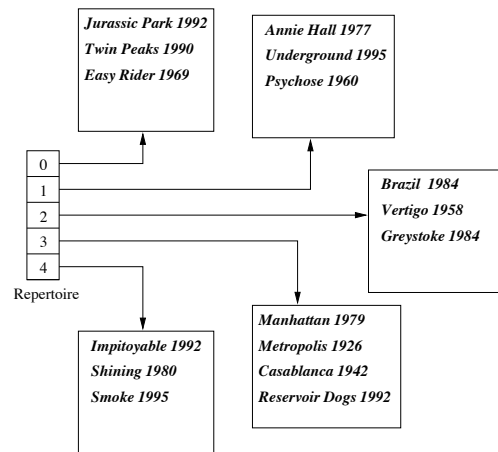
1. On estime le nombre N de pages qu'il faut allouer au fichier.
2. **fonction de hachage** H : à toute valeur de la clé de domaine V associe un nombre entre 0 et $N - 1$.

$$H : V \rightarrow \{0, 1, \dots, N - 1\}$$
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 :
 - 2.1 Clé: nom d'un film, on ne s'intéresse qu'à l'initiale de ce nom.
 - 2.2 On numérote les lettres de l'alphabet de 1 à 26 :
 $No('a') = 1, No('m') = 13$, etc.
 - 2.3 Si l est une lettre de l'alphabet, $H(l) = MODULO(No(l), 5)$.



Remarques

1. Le nombre $H(c) = i$ n'est pas une adresse de page, mais l'indice d'une 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 $\{0, \dots, N - 1\}$
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.

⇒ donc une recherche ne coûte qu'une seule lecture.

Hachage: insertion

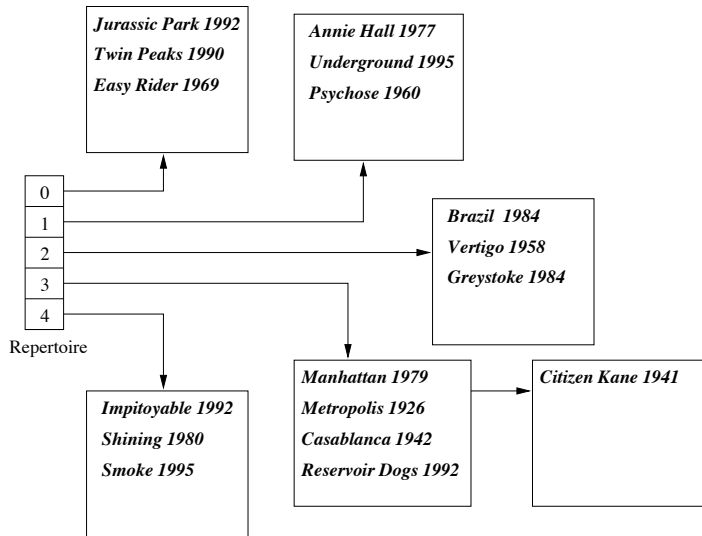
Recherche par $H(c)$ la page p où placer A et l'y insérer.

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' .

⇒ lors d'une recherche, il faut donc en fait parcourir la liste des pages chaînées correspondant à une valeur de $H(v)$.

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



Hachage: avantages et inconvénients

Intérêt du hachage :

1. **Très rapide.** Une seule E/S dans le meilleur des cas pour une recherche.
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

Plan du cours

1 Introduction	4
2 Le modèle relationnel.....	31
3 Algèbre relationnelle.....	44
4 SQL.....	84
5 Organisation physique des données.....	165
6 Optimisation.....	212
7 Évaluation de requêtes	257

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 stratégie** (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 (haute sélectivité) et que les résultats intermédiaires d'une opération et le résultat final sont écrits sur disque (10 n-uplets par page).

1. Jointure d'abord: Jointure: on lit 3 600 pages (120x30); on écrit le résultat intermédiaire (120 pages); Sélection: on relit le résultat et comme on projète sur tous les attributs de CINEMA, on obtient 5% de 120 pages, soit 6 pages.
Nombre d'E/S: $3\,600E + 120 \times 2E/S + 6S = 3\,846$.
2. Sélection d'abord: 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: $120E + 6S + 180E + 6S = 312$.

⇒ 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 (similaires aux 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.

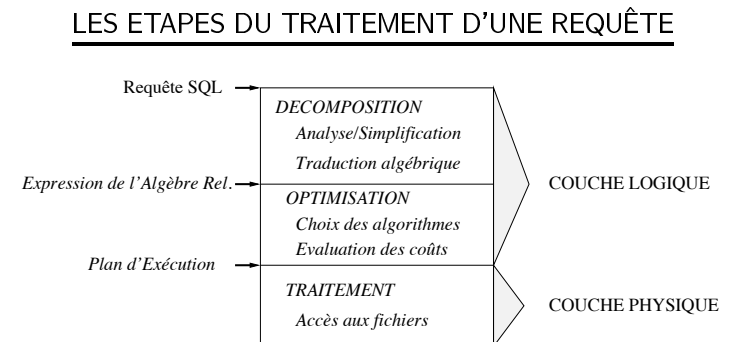
Les paramètres de l'optimisation

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



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.

⇒ On utilise le "dictionnaire" de la base qui contient le schéma.

Analyse, simplification et normalisation

D'autres types de transformations avant optimisation :

1. **Analyse sémantique** pour la détection d'incohérences.
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. **Normalisation** de la requête.
Exemple: transformation des conditions en forme normale conjonctive) et décomposition 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**.

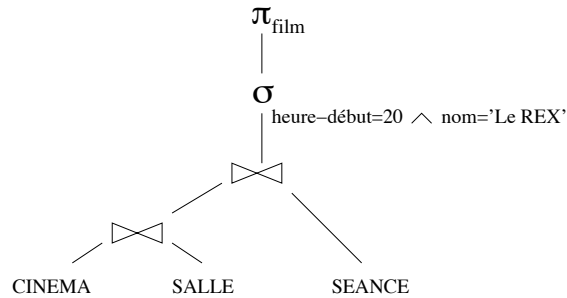
Traduction algébrique : exemple

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

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


(Vertigo)

225 / 393

Règles de réécriture

Il en existe beaucoup. En voici huit parmi les plus importantes :

▶ **Commutativité des jointures :**

$$R \bowtie S \equiv S \bowtie R$$

▶ **Associativité des jointures :**

$$(R \bowtie S) \bowtie T \equiv R \bowtie (S \bowtie T)$$

▶ **Regroupement des sélections :**

$$\sigma_{A='a' \wedge B='b'}(R) \equiv \sigma_{A='a'}(\sigma_{B='b'}(R))$$

▶ **Commutativité de la sélection et de la projection**

$$\pi_{A_1, A_2, \dots, A_p}(\sigma_{A_i='a'}(R)) \equiv \sigma_{A_i='a'}(\pi_{A_1, A_2, \dots, A_p}(R)), i \in \{1, \dots, p\}$$

(Vertigo)

227 / 393

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**.

(Vertigo)

226 / 393

Règles de réécriture

▶ **Commutativité de la sélection et de la jointure.**

$$\sigma_{A='a'}(R(\dots A \dots) \bowtie S) \equiv \sigma_{A='a'}(R) \bowtie S$$

▶ **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.

▶ **Commutativité de la projection et de la jointure**

$$\pi_{A_1 \dots A_p B_1 \dots B_q}(R \bowtie_{A_i=B_j} S) \equiv$$

$$\pi_{A_1 \dots A_p}(R) \bowtie_{A_i=B_j} \pi_{B_1 \dots B_q}(S),$$

$$(i \in \{1, \dots, p\}, j \in \{1, \dots, q\})$$

▶ **Distributivité de la projection sur l'union**

$$\pi_{A_1 A_2 \dots A_p}(R \cup S) \equiv \pi_{A_1 A_2 \dots A_p}(R) \cup \pi_{A_1 A_2 \dots A_p}(S)$$

(Vertigo)

228 / 393

Exemple d'un algorithme de restructuration

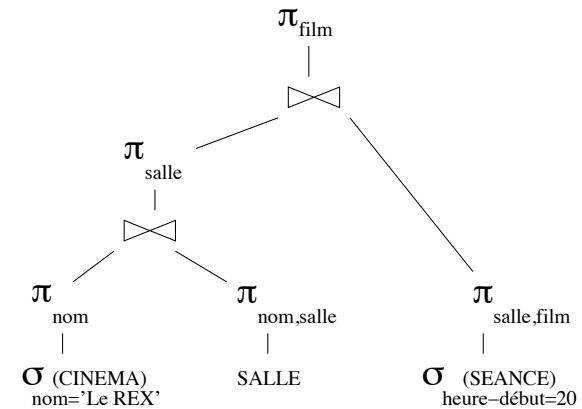
Voici un algorithme basé sur les propriétés précédentes.

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.

(Vertigo)

229 / 393

Arbre de requête après restructuration



(Vertigo)

230 / 393

Quelques remarques sur l'algorithme précédent

L'idée de base est de réduire le plus tôt possible (en bas de l'arbre) 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!

(Vertigo)

231 / 393

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.

(Vertigo)

232 / 393

Contre-exemple : évaluation 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 : $400E + 10 \times 2E/S + 9S = 429E/S$.
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 : $50E + 45S + 360E + 9S = 464E/S$.

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

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.

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 opération algébrique (jointures).
3. De propriétés statistiques de la base.

Algorithmes pour les opérations algébriques

On a généralement le choix entre plusieurs algorithmes pour effectuer une opération.

L'opération la plus étudiée est la JOINTURE (pourquoi?) :

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.

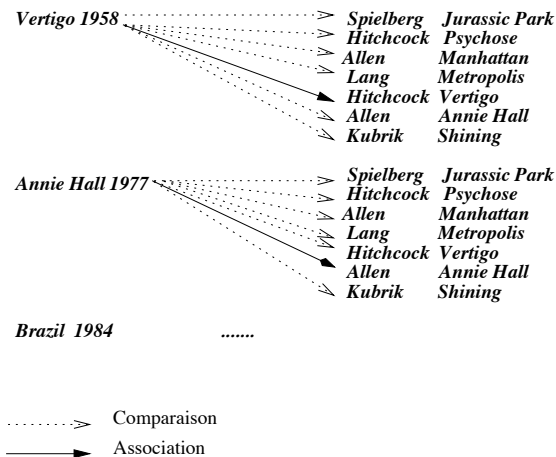
Jointure par boucles imbriquées

A utiliser quand les tailles des relations sont petites.
Soit les deux relations R et S :

```

ALGORITHME boucles-imbriquées
begin
  J:=∅
  for each r in R
    for each s in S
      if r et s sont joignables then J := J + {r ⋈ s}
end
    
```

Exemple de jointure par boucles imbriquées



Analyse

La boucle s'effectue 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 :

$$T_R \times T_S$$

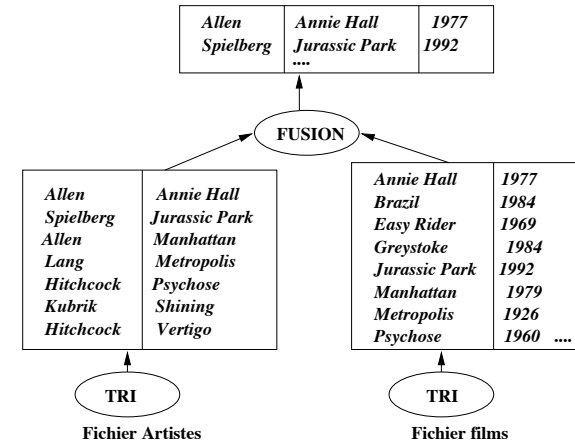
On ne tient pas compte dans l'évaluation du coût des algorithmes de jointure, du coût d'écriture du résultat sur disque, lequel dépend de la taille du résultat.

Jointure par tri-fusion

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

Algorithme: **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 .



(Vertigo)

241 / 393

(Vertigo)

242 / 393

Jointure par tri-fusion : performance

Le coût est dominé par la phase de tri:

$$\mathcal{O}(|R| \times \log(|R|) + |S| \times \log(|S|)).$$

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

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

Discussion

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 doublés ou l'affichage ordonné.
3. **Très général**: permet de traiter tous les types de θ -jointure

(Vertigo)

243 / 393

(Vertigo)

244 / 393

Jointure par hachage

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

1. Une des relations, R , est hachée sur l'attribut de jointure avec une fonction H .
2. La deuxième relation S 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 R . Si oui on fait la jointure limitée à ces n-uplets.

(Vertigo)

245 / 393

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

(Vertigo)

246 / 393

Jointure par hachage : discussion

Coût (en E/S), en supposant k articles par page et un tampon de 2 pages en mémoire centrale (simplification):

1. Phase 1: Coût du hachage de R :
 $T_R E + 2 \times k \times T_R E / S = \mathcal{O}(2 \times |R|)$ (pour chaque n-uplet il faut lire et écrire une page).
2. Phase 2: Lecture de S : $T_S E + k \times T_S E = \mathcal{O}(|S|)$ (pour chaque page, on lit k pages de la relation hachée R).
3. Coût total = $((1 + 2k) \times T_R) + ((1 + k) \times T_S) = \mathcal{O}(2 \times |R| + |S|)$

Il est préférable d'effectuer le hachage (phase 1) sur la plus petite des deux relations.

Si R tient en mémoire centrale, le coût se réduit à $T_R + T_S$.

Contrairement à la jointure par tri-fusion, la jointure par hachage n'est pas adaptée aux jointures avec inégalités.

(Vertigo)

247 / 393

Jointure avec une table indexée

1. On parcourt séquentiellement la table sans index (table directrice).
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 n-uplets de la seconde relation par adresse)

(Vertigo)

248 / 393

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: $\mathcal{O}(|R| \times \log(|S|))$.

(Vertigo)

249 / 393

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.

On 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.

(Vertigo)

251 / 393

Jointure avec deux tables indexées

Si les deux tables sont indexées sur les deux attributs de jointure, 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.

(Vertigo)

250 / 393

Plans d'exécution

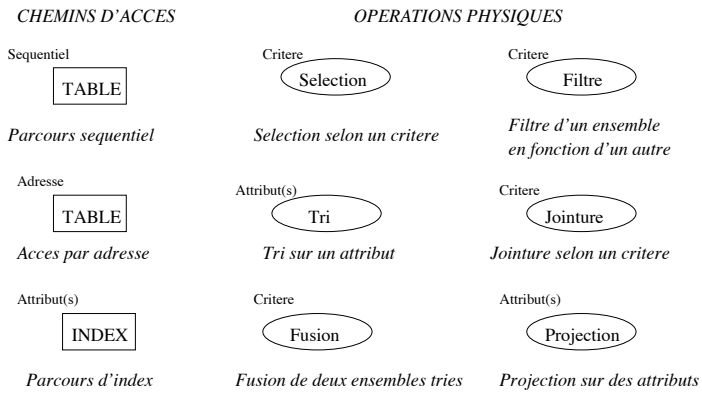
Le résultat de l'optimisation est un **plan d'exécution**: c'est un ensemble d'opérations de niveau intermédiaire, dit **algèbre "physique"** constituée :

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

(Vertigo)

252 / 393

Algèbre physique



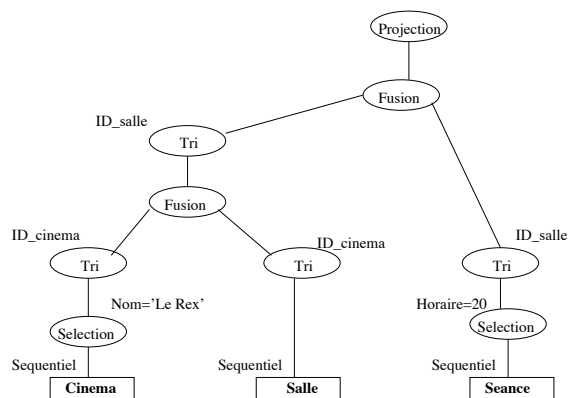
Exemple

Quels films passent au REX à 20 heures ?

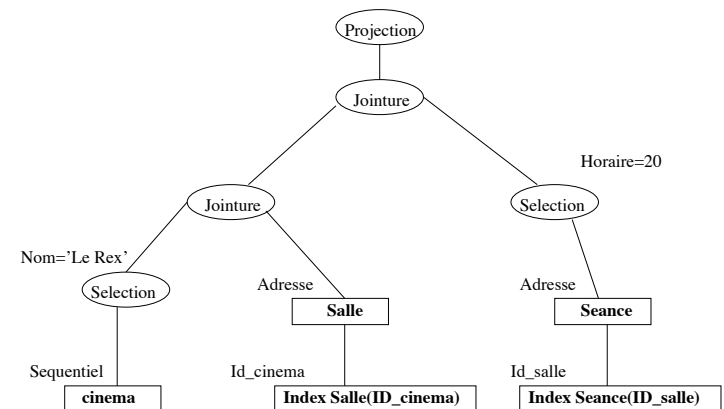
```
select Titre
  from Cinema, Salle, Seance
 where Cinema.nom = 'Le Rex'
    and Cinema.ID_cinema = Salle.ID_cinema
    and Salle.ID_salle=Seance.ID_salle
    and Seance.horaire='20'
```

La requête contient deux selections et deux jointures.

Sans index ni hachage



Avec un index sur les attributs de jointure



Plan du cours

1 Introduction	4
2 Le modèle relationnel.....	31
3 Algèbre relationnelle.....	44
4 SQL.....	84
5 Organisation physique des données.....	165
6 Optimisation.....	212
7 Évaluation de requêtes	257

(Vertigo)

257 / 393

Objectifs de l'évaluation de requêtes

Dans l'hypothèse d'une base centralisée, on cherche essentiellement à limiter le nombre d'entrées/sorties. La stratégie employée dépend cependant fortement du point de vue adopté :

1. Soit on cherche à obtenir le premier enregistrement le plus vite possible (exemple d'une application interactive).
2. Soit on cherche à minimiser le temps global d'exécution (exemple d'une application batch).

Selon le cas, le choix des algorithmes peut varier.

(Vertigo)

259 / 393

En quoi consiste l'évaluation d'une requête

Le résultat de l'optimisation est un plan d'exécution, i.e. une séquence d'opérations à exécuter. On doit maintenant :

1. Appliquer les *techniques d'accès* appropriées pour chaque opération du plan d'exécution.
2. Gérer les *flots de données* entre chaque opération.

Les algorithmes, techniques d'accès et heuristiques mise en oeuvre relèvent de **l'évaluation de requêtes**.

(Vertigo)

258 / 393

Exemple de référence

<i>Vertigo</i>	1958	<i>Metropolis</i>	1926
<i>Annie Hall</i>	1977	<i>Psychose</i>	1960
<i>Brazil</i>	1984	<i>Greystoke</i>	1984
<i>Twin Peaks</i>	1990	<i>Shining</i>	1980
<i>Jurassic Park</i>	1992	<i>Manhattan</i>	1979
<i>Underground</i>	1995	<i>Easy Rider</i>	1969

Les films

<i>Spielberg</i>	<i>Jurassic Park</i>
<i>Hitchcock</i>	<i>Psychose</i>
<i>Allen</i>	<i>Manhattan</i>
<i>Lang</i>	<i>Metropolis</i>
<i>Hitchcock</i>	<i>Vertigo</i>
<i>Allen</i>	<i>Annie Hall</i>
<i>Kubrik</i>	<i>Shining</i>

Les metteurs en scène

(Vertigo)

260 / 393

Techniques d'accès pour un parcours séquentiel

Systématiquement optimisé dans les SGBD en utilisant les techniques suivantes :

1. **Regroupement** des pages disques sur des espaces contigus (nommés **segments** ou **extensions**).
2. **Lecture à l'avance** : quand on lit une page, on prend également les n (typiquement $n = 7$ ou $n = 15$) suivantes dans le segment.

⇒ la taille de l'unité d'E/S dans un SGBD (buffer) est donc souvent un multiple de celle du gestionnaire de fichier sous-jacent (page).

Parcours d'index : éviter les lectures multiples

Soit la requête suivante, en supposant un index sur Année :

```
SELECT titre
FROM Film
WHERE année IN (1956, 1934, 1992, 1997)
```

- ▶ **Implantation simple** : on recherche dans l'index les adresses pour chaque valeur du **IN** et on lit l'enregistrement.
- ▶ **Implantation optimisée** :
 1. On recherche dans l'index **toutes** les adresses pour **toutes** les valeurs du **IN**.
 2. On regroupe l'ensemble d'adresses par numéro de page.
 3. On lit les pages et on extrait les enregistrements

Techniques d'accès pour un parcours d'index

La plupart des SGBD utilisent une des variantes de l'arbre B. Outre les recherches par clés et par intervalle, ils permettent :

1. D'éviter l'accès aux données quand la valeur recherchée est dans l'index.
2. De faire directement des comptages ou des tests d'existence.
3. Enfin on peut optimiser des opérations de sélection ou de jointure en manipulant les adresses stockées dans l'index.

On a toujours le même objectif : on veut éviter des échanges inutiles de pages.

Parcours d'index : regroupement d'adresses avec zone tampon

L'optimisation précédente a permis de ne pas lire deux fois la même page. **Mais** elle impose d'attendre qu'on ait lu toutes les adresses avant d'afficher le résultat.

On peut appliquer une technique intermédiaire utilisant une zone tampon T :

1. On lit les adresses et on les place (triées) dans T .
2. Dès qu'il faut fournir une donnée, ou que T est plein, on lit la page la plus référencée.

Cette technique peut être utilisée pour des jointures.

Techniques d'accès : gestion d'un buffer

Problème très complexe: essayer de conserver en mémoire une page susceptible d'être réutilisée "prochainement".

Le programme exécutant la requête ne demande pas lui-même la lecture, mais s'adresse au *buffer manager* du SGBD. Les concepts essentiels sont :

1. **Page statique** : la page reste en mémoire jusqu'à ce qu'on demande au *buffer manager* de la libérer.
2. **Page volatile** : la page est à la disposition du *buffer manager*.

Rappels sur l'algorithme de tri-fusion

On applique la stratégie dite "diviser pour régner". Elle consiste à :

1. **Diviser** récursivement de problème jusqu'à obtenir des sous-problèmes trivialement résolubles.
2. **Fusionner** récursivement les solutions des sous-problèmes.

NB: quand on peut faire le tri en mémoire centrale, on utilise plutôt le **tri rapide** (*Quicksort*).

Évaluation d'un tri

Le tri est une opération fréquente. On l'utilise par exemple :

1. Pour afficher des données ordonnées (clause ORDER BY).
2. Pour éliminer des doublons ou faire des agrégats.
3. Pour certains algorithmes de jointure (tri-fusion).

L'algorithme utilisé dans les SGBD est le **tri par fusion** (*MERGE SORT*).

Fusion de deux tables triées

Soit deux tables R et S triées sur les attributs A et B respectivement. On les fusionne en effectuant un parcours parallèle (comme dans la jointure par tri-fusion).

Algorithme Fusionner(R,S)

begin

$x :=$ premier n -uplet de R ; $y :=$ premier n -uplet de S

while il reste un n -uplet dans R ou dans S

if $x.A < x.B$ **then**

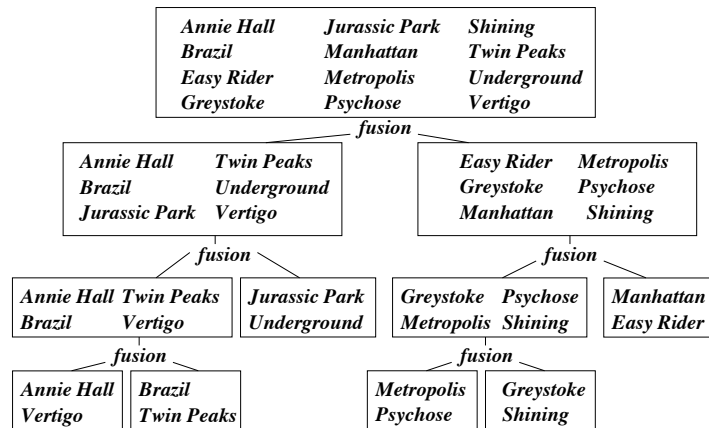
 Ecrire x ; $x :=$ n -uplet suivant de R

else

 Ecrire y ; $y :=$ n -uplet suivant de S

end

Exemple de tri-fusion sur le fichier des films



(Vertigo)

269 / 393

Algorithme de tri-fusion (externe) dans un SGBD

Soit M la mémoire disponible, B la taille d'un buffer de lecture/écriture et E la taille du fichier d'entrée.

Étape 0:

- ▶ Lire récursivement le fichier d'entrée en segments de taille $T \leq M$.
- ▶ Trier chaque segment en mémoire central avec *Quicksort*.
- ▶ Écrire chaque segment trié sur disque après.

On obtient $S_0 = \lceil E/M \rceil$ segments triés.

(Vertigo)

271 / 393

Tri-fusion dans un contexte SGBD

Dans un contexte SGBD, on fixe les paramètres du tri-fusion de la manière suivante :

1. On s'arrête de diviser quand on peut trier en mémoire centrale. Donc la taille M de la mémoire disponible détermine la taille des sous-problème triviaux.
2. Dans la phase de fusion, le facteur de division est $F = (M/B) - 1$ où B est la taille d'un buffer (page) de lecture/écriture. On a F buffers en lecture et 1 en écriture.

(Vertigo)

270 / 393

Algorithme de tri-fusion (externe) dans un SGBD

Étape $i > 0$:

- ▶ On fusionne $F = \lfloor (M/B) - 1 \rfloor$ segments triés simultanément ($M/B =$ nombre de buffers).
- ▶ On obtient $S_i = S_{i-1}/F$ segments triés sur disque.

Il y a $L = \log_F(S_0)$ étapes de fusion.

(Vertigo)

272 / 393

Exemple

Supposons qu'un article de *Films* occupe $8KO$. Le fichier de 12 films occupe $E = 96KO$. On dispose de $M = 16KO$ en mémoire et un buffer est de taille $B = 4KO$. Donc il y a au maximum 4 buffers :

1. L'étape 0 génère 6 ségments triés de taille $T = 16KO$.
2. Pour les étapes suivantes on dispose de $F = (16/4) - 4 = 3$ buffers en lecture. Donc on obtient le résultat final après deux étapes de fusion et l'étape 0.

(Vertigo)

273 / 393

Représentation Oracle

Représentation physique des données dans ORACLE V7

Les principales structures physiques utilisées dans ORACLE sont :

1. Le **bloc** est l'unité physique d'E/S (entre $1KO$ et $8KO$). La taille d'un bloc ORACLE est un multiple de la taille des blocs (pages) 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 ...).

(Vertigo)

276 / 393

Coût d'une opération de tri

L'analyse donne :

1. L'étape 0 consiste en une lecture **et** une écriture de tous les n-uplets du fichier.
2. Chaque étape de fusion consiste en une lecture **et** une écriture de tous les n-uplets du fichier.

D'où un coût global de $2 \times (L + 1) \times E$.

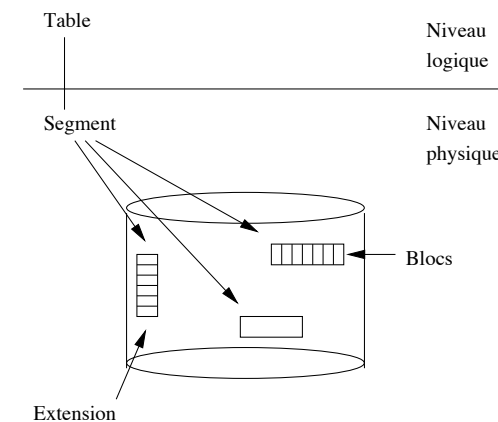
NB : ce type de formule est utilisé dans un modèle de coût.

(Vertigo)

274 / 393

Représentation Oracle

Tables, segments, extensions et blocs



(Vertigo)

277 / 393

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).

(Vertigo)

278 / 393

Base ORACLE, fichiers et *TABLESPACE*

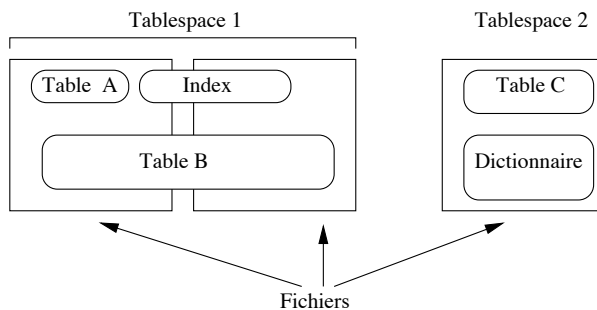
1. **Physiquement**, une base ORACLE est un ensemble de fichiers.
2. **Logiquement**, une base est divisée par l'administrateur en *tablespace*. Chaque *tablespace* consiste en un ou plusieurs fichiers.

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).

(Vertigo)

279 / 393



(Vertigo)

280 / 393

Stockage des données

Il existe deux manières de stocker une table :

1. **Placement indépendant** : Les segments sont automatiquement alloués à la table. Il est possible de spécifier des paramètres pour la création d'un nouveau segment :
 - 1.1 Sa taille initiale.
 - 1.2 Le pourcentage d'espace libre dans chaque bloc.
 - 1.3 La taille des extensions.
2. **Dans un *cluster***.

(Vertigo)

281 / 393

TABLESPACE : définition

Exemple : Création d'un "tablespace" *tabspace_2* dans le fichier *diska:tabspace_file2.dat* de taille 20MO :

```
CREATE TABLESPACE tabspace_2
  DATAFILE 'diska:tabspace_file2.dat' SIZE 20M
  DEFAULT STORAGE (INITIAL 10K NEXT 50K
    MINEXTENTS 1 MAXEXTENTS 999
    PCTINCREASE 20)
  ONLINE
```

La taille de la première extension est 10KO, de la deuxième extension 50KO avec un taux de croissance de 20% pour les extensions suivantes : 60KO, 72KO, 86.4KO, ... (défaut: 50%)

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

Exemple : $\overbrace{00000DD5}^{\text{bloc}}.\overbrace{000}^{\text{row}}.\overbrace{001}^{\text{file}}$ est l'adresse du premier n-uplet de la page DD5 dans le premier fichier.

CREATE TABLE

Exemple : Création d'une table *salgrade* dans le tablespace *tabspace_2*:

```
CREATE TABLE salgrade (
  grade NUMBER PRIMARY KEY USING INDEX TABLESPACE users_a
  losal NUMBER,
  hisal NUMBER )
  TABLESPACE tabspace_2
  PCTFREE 10 PCTUSED 75
```

L'index est stocké dans le "tablespace" *users_a*. Pour les données, 10% dans chaque bloc sont réservés pour les mise-à-jours. On insère des n-uplets dans un bloc si l'espace occupé descend en dessous de 75%.

Structures de données pour l'optimisation

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

1. Les index.
2. Les "regroupements" de tables (ou *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.

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	
	

Les clusters

Le *cluster* (regroupement) est une 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*.

CREATE CLUSTER (index cluster)

On définit un *index cluster* décrivant les caractéristiques physiques de la table:

```
CREATE CLUSTER cluster-cinema (ID-cinéma NUMBER(10))
  SIZE 2K
  INDEX
  STORAGE (INITIAL 100K NEXT 50K)
```

SIZE est la taille de toutes les données pour une clé donnée.

Il faut explicitement créer un index pour le cluster:

```
CREATE INDEX ind_cin ON CLUSTER cluster-cinema
```

CREATE TABLE

On fait référence au *cluster* pendant la création des tables :

```
CREATE TABLE Cinema (
  id-cinema NUMBER(10) PRIMARY KEY,
  nom-cinema VARCHAR(32), adresse VARCHAR(64))
CLUSTER cluster-cinéma(id-cinema);
```

```
CREATE TABLE Salle (
  id-cinema NUMBER(10),
  id-salle NUMBER(3),
  nom-salle VARCHAR(32), capacite NUMBER)
CLUSTER cluster-cinéma(id-cinema);
```

(Vertigo)

290 / 393

Optimisation : l'exemple de ORACLE Version 7

Plan de la présentation :

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

(Vertigo)

293 / 393

Le hachage (hash cluster)

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 1000 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 donnée (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.

(Vertigo)

291 / 393

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.

(Vertigo)

295 / 393

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.

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.

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*).

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.

(Vertigo)

300 / 393

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.

(Vertigo)

301 / 393

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'
```

(Vertigo)

302 / 393

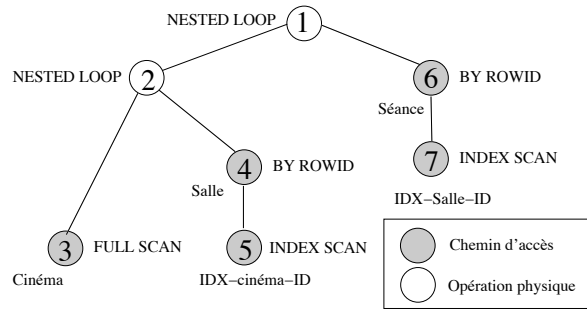
Plan d'exécution donné par EXPLAIN

```
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
```

(Vertigo)

303 / 393

Représentation arborescente du plan d'exécution



(Vertigo)

304 / 393

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).

(Vertigo)

305 / 393

Exemple : sélection sans index

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

Plan d'exécution :

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

(Vertigo)

306 / 393

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
```

(Vertigo)

307 / 393

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
```

(Vertigo)

308 / 393

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
```

(Vertigo)

309 / 393

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
```

(Vertigo)

310 / 393

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
```

(Vertigo)

311 / 393

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
```

(Vertigo)

312 / 393

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
```

(Vertigo)

313 / 393

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
```

(Vertigo)

314 / 393

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')
```

(Vertigo)

315 / 393

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

```

(Vertigo)

316 / 393

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

(Vertigo)

319 / 393

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

(Vertigo)

318 / 393

Exemple: programme de crédit d'un compte bancaireCrédit (*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 → transaction
- ▶ plusieurs exécutions concurrentes du même programme possibles

(Vertigo)

320 / 393

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*

(Vertigo)

321 / 393

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

- ▶ $\text{solde}(\text{compte}) \geq 0$; $\text{solde}(\text{source}) + \text{solde}(\text{dest}) = \text{const}$
- ▶ une transaction modifie la BD d'un état initial cohérent à un état final cohérent
- ▶ pendant la transaction, l'état peut être incohérent!

(Vertigo)

323 / 393

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

(Vertigo)

322 / 393

Isolation: une transaction ne voit pas les effets des autres transactions en cours d'exécution

- ▶ la transaction s'exécute comme si elle était 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

(Vertigo)

324 / 393

Concurrence

- ▶ plusieurs transactions s'exécutent en même temps
- ▶ le système exécute les opérations en séquence!
- ▶ concurrence = entrelacement des opérations de plusieurs transactions

(Vertigo)

325 / 393

Exécution concurrente (histoire)

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

Exemple :

H₁: $r_1[x] r_2[x] w_2[x] r_2[y] w_1[x] w_2[y] c_2 c_1$ (crédit + transfert, histoire complète)

H₂: $r_1[x] r_2[x] w_2[x] r_2[y]$ (histoire incomplète)

(Vertigo)

327 / 393

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 :

T₁: $r_1[x] w_1[x] c_1$ (crédit)

T₂: $r_2[x] w_2[x] r_2[y] w_2[y] c_2$ (transfert)

T₃: $r_3[x] a_3$ (transfert impossible)

(Vertigo)

326 / 393

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

(Vertigo)

328 / 393

Problèmes de concurrence

1. Perte d'une mise à jour

Ex. $T_1: \text{Crédit}(x, 100) = r_1[x] \ w_1[x] \ c_1$ $T_2: \text{Crédit}(x, 50) = r_2[x] \ w_2[x] \ c_2$
au début, $x=200$

- ▶ $H = r_1[x]_{x:200} \ r_2[x]_{x:200} \ w_1[x]_{x:300} \ w_2[x]_{x:250} \ c_1 \ c_2$
- ▶ Résultat: $x=250$ au lieu de $x=350$ ($w_1[x]$ est perdu car la lecture dans T_2 n'y tient pas compte)
- ▶ Problème de cohérence à cause d'une **lecture non-répétable**
- ▶ Une fois une donnée lue, une transaction devrait retrouver la même **valeur plus tard**
- ▶ Ici T_2 lit $x=200$, mais après $w_1[x]$, x devient 300

(Vertigo)

329 / 393

3. Analyse incohérente

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

- ▶ $T_1 = r_1[x] \ w_1[x] \ r_1[y] \ w_1[y] \ c_1$; $T_2 = r_2[x] \ r_2[y] \ w_2[z] \ c_2$
- ▶ $H = r_1[x]_{x:200} \ w_1[x]_{x:150} \ r_2[x]_{x:150} \ r_2[y]_{y:100} \ w_2[z]_{z:250} \ c_2 \ r_1[y]_{y:100} \ w_1[y]_{y:150} \ c_1$
- ▶ 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 la **lecture d'une valeur non-validée** ("lecture sale")

(Vertigo)

331 / 393

2. Dépendances non-validées

Ex. Les mêmes transactions, mais T_1 est annulée

- ▶ $H = r_1[x]_{x:200} \ w_1[x]_{x:300} \ r_2[x]_{x:300} \ w_2[x]_{x:350} \ c_2 \ a_1$
- ▶ $r_2[x]$ utilise la valeur non-validée de x écrite par $w_1[x]$
- ▶ L'annulation de T_1 impose l'annulation de T_2 , qui est validée!
- ▶ Problème de durabilité à cause de la **lecture d'une valeur non-validée** ("lecture sale")

(Vertigo)

330 / 393

4. Objets fantômes

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

5. Ecritures incohérentes

Ex. Ecriture d'un même solde dans deux comptes

$T_1 =$ écriture de 100 dans x et y ; $T_2 =$ écriture de 200 dans x et y

- ▶ $T_1 = w_1[x] \ w_1[y] \ c_1$; $T_2 = w_2[x] \ w_2[y] \ c_2$
- ▶ $H = w_1[x]_{x:100} \ w_2[x]_{x:200} \ w_2[y]_{y:200} \ w_1[y]_{y:100} \ c_1 \ c_2$
- ▶ Résultat: $x \neq y$! ($x = 200, y = 100$)
- ▶ Problème de cohérence à cause de l'**écriture sur des données non-validées**

(Vertigo)

332 / 393

Contrôle de concurrence

- ▶ solution: algorithmes de réordonnement des opérations
- ▶ critère de correction utilisé: exécution sérialisable (équivalente à une exécution en série *quelconque* des transactions)

Remarques

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

(Vertigo)

333 / 393

Exécution recouvrable = ne permet pas d'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
- ▶ 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)

(Vertigo)

335 / 393

3. Le problème des annulations

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

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

Conclusion: une transaction validée risque encore d'être annulée

Types d'exécutions concurrentes par rapport à l'annulation

- ▶ exécution recouvrable
- ▶ exécution qui évite les annulations en cascade
- ▶ exécution stricte

(Vertigo)

334 / 393

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 gênante
- ▶ *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)

(Vertigo)

336 / 393

Exécution stricte

= évite les annulations en cascade + annulation simple des 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, ne tient pas compte de $w_2[x, 3]$

a_2 restaure $x=2$: erreur, cette valeur est déjà annulée

- ▶ solution: $w_2[x]$ attend la fin de tout T_i qui a écrit x

⇒ retardement lectures + écritures

(dans l'exemple, retardement de $w_2[x]$ après la fin de T_1)

4. Tolérance aux pannes

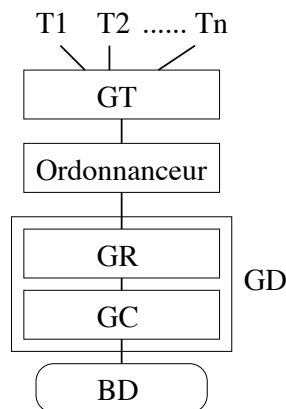
Mémoire SGBD = Mémoire stable (disque) + Mémoire volatile

Catégories de pannes

- ▶ **de transaction:** annulation de la transaction
- ▶ **de système:** perte du contenu de la mémoire volatile
 - ▶ opération *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

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

(Vertigo)

341 / 393

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)

(Vertigo)

343 / 393

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

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

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

(Vertigo)

342 / 393

Conflit

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

- ▶ $i \neq j, x=y$ (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

(Vertigo)

344 / 393

Critère d'équivalence basé sur les conflits

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

Ce dernier critère:

- ▶ couvre les conditions 2 et 3 de la définition
- ▶ est plus strict, mais plus facile à vérifier que 2 et 3

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

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 \iff SG(H) acyclique

$H_1, H_2: T_1 \leftarrow T_2$

$H_3, H_4: T_1 \leftarrow T_2$

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: $r_1[x] - w_2[x], w_2[x] - w_1[x], w_1[y] - r_2[y], w_1[y] - w_2[y]$

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

$\implies H_2 \equiv H_1$

$H_3: w_2[x] r_2[y] \underline{r_1[x]} w_2[y] \underline{w_1[y]} c_2 \underline{w_1[x]} c_1$
conflits: $w_2[x] - r_1[x], w_2[x] - w_1[x], r_2[y] - w_1[y], w_2[y] - w_1[y]$

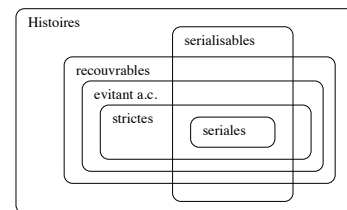
$\implies H_3 \not\equiv H_1$

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

$\implies H_4 \equiv H_3 \implies H_3$ sérialisable

Propriétés de recouvrabilité

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



Exemple

$T_1: r_1[x] w_1[y] r_1[z] w_1[z] c_1$ $T_2: r_2[x] w_2[y] r_2[z] w_2[z] c_2$
 $H: r_1[x] r_2[x] w_1[y] r_1[z] w_1[z] w_2[y] r_2[z] w_2[z]$
conflicts: $w_1[y] - w_2[y], r_1[z] - w_2[z], w_1[z] - r_2[z], w_1[z] - w_2[z]$
 $\implies H$ est sérialisable pour n'importe quelle position de c_1 et de c_2
 $H_1: r_1[x] r_2[x] w_1[y] r_1[z] w_1[z] w_2[y] r_2[z] w_2[z] c_2 c_1$
 H_1 n'est pas recouvrable (T_2 lit z de T_1 et c_2 après c_1)
 $H_2: r_1[x] r_2[x] w_1[y] r_1[z] w_1[z] w_2[y] r_2[z] c_1 w_2[z] c_2$
 H_2 n'évite pas les annulations en cascade (T_2 lit z de T_1 avant c_1)
 $H_3: r_1[x] r_2[x] w_1[y] r_1[z] w_1[z] w_2[y] c_1 r_2[z] w_2[z] c_2$
 H_3 n'est pas stricte (T_1 écrit y et ensuite T_2 écrit y avant c_1)

2. Contrôle par verrouillage à deux phases

Principe

- ▶ stratégie pessimiste : retardement des opérations 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 composé** = sous-verrou de lecture (partageable) + sous-verrou d'écriture (exclusif)

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
 \implies extensible à d'autres opérations que Read et Write

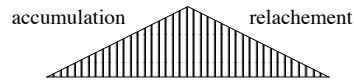
	Read	Write	Incr	Decr
Read	oui	non	non	non
Write	non	non	non	non
Incr	non	non	oui	oui
Decr	non	non	oui	oui

Algorithme de base

- 1) L'ordonnanceur reçoit $p_i[x]$ et teste les sous-verrous de x
 - ▶ si l'un des sous-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 ⇒ 2 phases: accumulation et relâchement de verrous



- ▶ règle 3 ⇒ les paires d'opérations conflictuelles de T_i et T_j s'exécutent toujours dans le même ordre
- ▶ règle 2 ⇒ l'ordre du GD pour les opérations conflictuelles sur un enregistrement x est le même que celui de l'ordonnanceur
- ▶ la variante la plus utilisée: relâchement de tous les verrous d'une transaction juste après *Commit*
 - ▶ assure le respect des règles 2 et 3
 - ▶ on peut démontrer que cela produit *des exécutions strictes*

(Vertigo)

353 / 393

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

(Vertigo)

355 / 393

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

(Vertigo)

354 / 393

c) Exécution sérialisable modifiée par le verrouillage

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

- ▶ conflits: $r_1[x] - w_2[x] \Rightarrow H$ sérialisable, équivalente à T_1T_2
- ▶ $w_2[x]$ est bloquée, T_1 ne peut pas relâcher le verrou de lecture sur x , car elle aura besoin d'un autre pour $w_1[z]$
- ▶ Conclusion: certaines exécutions sérialisables ne sont pas acceptées telles quelles par le verrouillage à deux phases
 - ▶ le verrouillage ne profite pas de la sérialisabilité préalable des exécutions
 - ▶ cause: algorithme pessimiste

(Vertigo)

356 / 393

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 $w_2[y]$
⇒ interblocage de T_1 et de T_2

(Vertigo)

357 / 393

3. Verrouillage hiérarchique

Données à plusieurs niveaux de granularité

- ▶ attribut < enregistrement < relation < base de données
- ▶ chaque opération se fait au niveau approprié
- ▶ verrouillage classique: un seul niveau de granularité
 - ▶ niveau élevé: gestion allégée, car moins de verrous
 - ▶ niveau bas: moins de conflits, plus de concurrence

(Vertigo)

359 / 393

Stratégies pour éviter l'interblocage

- ▶ durée limite ("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

(Vertigo)

358 / 393

Principes du verrouillage hiérarchique

- ▶ **verrouillage descendant**: implicite (par inclusion)
 - Ex. un verrou sur la relation est valable aussi pour chaque enregistrement, attribut, ... de la relation
- ▶ **verrouillage ascendant**: pour réaliser une opération à un niveau, on demande un *verrou d'intention* à tous les niveaux supérieurs
 - Ex. pour réaliser une écriture sur un enregistrement, on demande un verrou d'intention d'écriture sur la relation et la BD
 - ▶ obtention des verrous d'intention: descendante
 - ▶ relâchement des verrous d'intention: ascendant
- ▶ **verrouillage en escalade**: technique de choix du niveau de granularité
 - ▶ on commence par verrouiller au niveau fin (ex. enregistrement)
 - ▶ à partir d'un seuil, on passe au niveau de granularité supérieur (ex. relation)

(Vertigo)

360 / 393

Types de verrous

- ▶ **S** (partagé, lecture), **X** (exclusif, écriture)
- ▶ **IS** (intention lecture), **IX** (intention écriture)
- ▶ **SIX** (lecture et intention d'écriture)
 - ▶ cas très fréquent: lecture relation pour modifier quelques enregistrements
 - ▶ un **seul** verrou qui combine les verrous **S** et **IX**

Matrice de conflits entre verrous hiérarchiques

	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

Exemple

- ▶ données: relation **Compte**(*Numéro, Titulaire, Solde*)
- ▶ niveaux: **relation**, enregistrement
- ▶ T₁: crédit du compte numéro 7 (lecture-écriture enregistrement x)
- ▶ T₂: lecture de tous les comptes (lecture relation R)
- ▶ T₃: initialisation compte 3 (écriture enregistrement y)

Exécution par verrouillage hiérarchique

- ▶ ordre de réception: r₁[x] r₂[R] w₃[y] w₁[x] c₁ c₂ c₃
- r₁[x]: obtention **IS**(R) et ensuite **S**(x)
- r₂[R]: obtention **S**(R) - pas de conflit avec **IS**(R)
- w₃[y]: ne peut pas obtenir **IX**(R) à cause de **S**(R) → bloquée
- w₁[x]: ne peut pas obtenir **IX**(R) à cause de **S**(R) → bloquée
- c₁ bloquée car T₁(w₁[x]) bloquée
- c₂ s'exécute, **S**(R) relâché → opérations débloquées:
 - w₃[y] obtient **IX**(R) (pas de conflit avec **IS**(R)), puis **X**(y)
 - w₁[x] obtient **IX**(R) (pas de conflit avec **IS**(R), **IX**(R)), puis **X**(x)
 - c₁ s'exécute
- c₃ s'exécute

4. Création/destruction d'objets

Problème des objets fantômes

- ▶ contrôle de concurrence dans les BD dynamiques
- ▶ cas typique: T₁ consulte tous les objets d'un certain type, T₂ crée un objet de ce type

Exemple

- ▶ Relations **Compte**(*Numéro*, Titulaire, Solde) de comptes et **Total**(*Titulaire*, Cumul) de totaux par titulaire
- ▶ T₁: compare la somme des comptes de "Dupont" avec le total pour "Dupont"
- ▶ T₂: 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!

Verrouillage d'index

- ▶ ensemble d'objets qui respectent *attribut = valeur* regroupés dans une entrée d'index
- ▶ entrée d'index sur *attribut: valeur* + liste pointeurs vers les objets
- ▶ verrouillage entrée d'index
 - ▶ parcourir les objets → lecture index
 - ▶ insérer un objet → écriture index

Verrouillage hiérarchique

- ▶ ensemble = niveau de granularité supérieur
- ▶ verrous sur la donnée de niveau supérieur
 - ▶ parcourir les objets → lecture niveau supérieur
 - ▶ insérer un objet → écriture niveau supérieur

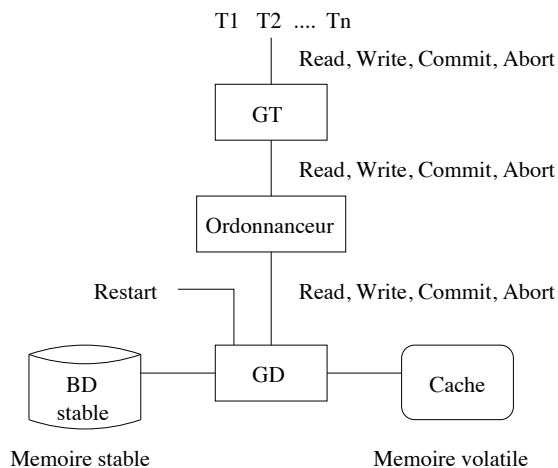
Solution

- ▶ pour consulter tous les objets d'un type: *info de contrôle* pour cet ensemble d'objets
- ▶ *verrouillage au niveau de l'info de contrôle*
- ▶ on traite l'ensemble comme un objet à part
 - ▶ parcours = lecture ensemble
 - ▶ insertion/suppression = écriture ensemble

1. Problématique

Hypothèses

- ▶ pannes de système: contenu de la mémoire volatile perdu
 - ▶ opération *Restart* qui reconstitue un état cohérent de la BD
- ▶ ordonnanceur avec exécutions sérialisables et *strictes*
 - ▶ stricte: écritures validées dans l'ordre de Commit
- ▶ même granularité pour l'ordonnanceur et le GD
 - ▶ enregistrement = page disque



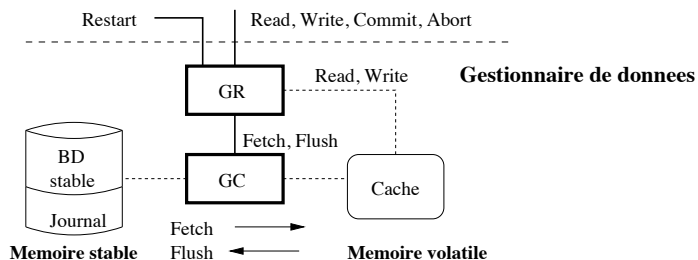
Objectif

- ▶ 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
- ⇒ 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 (pas dans le Cache)

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)

(Vertigo)

373 / 393

Gestionnaire de reprise

Opérations

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

(Vertigo)

375 / 393

- ▶ 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

(Vertigo)

374 / 393

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
 - ▶ souvent on stocke aussi *l'image avant* de l'écriture
- ▶ **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

(Vertigo)

376 / 393

Idempotence de Restart

- ▶ une panne peut interrompre toute opération, même *Restart*
- ▶ idempotence: *Restart* interrompu et relancé donne le même résultat que le *Restart* complet
- ▶ optimisation: journalisation des opérations de *Restart* pour ne pas tout recommencer

(Vertigo)

381 / 393

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

(Vertigo)

383 / 393

Checkpointing

- ▶ ajouter des informations sur disque en fonctionnement normal afin de réduire le travail de *Restart*
- ▶ *point de contrôle* ("checkpoint"): point (marqué dans le journal) où l'on réalise les actions supplémentaires

Quelques techniques

- ▶ marquer dans le journal les écritures déjà réalisées/annulées dans la BD stable
 - ▶ pas besoin de refaire/annuler ces écritures à la reprise
- ▶ marquer toutes les écritures validées/annulées dans la BD stable
 - ▶ pas besoin de refaire/annuler à la reprise les transactions validées/annulées

(Vertigo)

382 / 393

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

(Vertigo)

384 / 393

▶ 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** - $\{T_i\}$

(Vertigo)

385 / 393

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

(Vertigo)

387 / 393

▶ Restart

marquer toutes les cellules comme vides

refait = $\{\}$, **annulé** = $\{\}$

pour chaque $[T_i, x, v] \in$ **journal** (à partir de la fin) où $x \notin$ **annulé** \cup **refait**

si x n'est pas dans le cache alors allouer cellule pour x

si $T_i \in$ **liste_commit** alors

$cellule(x) = v$

refait = **refait** \cup $\{x\}$

sinon

$cellule(x) = image_avant(x, T_i)$

annulé = **annulé** \cup $\{x\}$

si **refait** \cup **annulé** = BD alors stop boucle

pour chaque $T_i \in$ **list_commit**

list_active = **list_active** - $\{T_i\}$

confirmer Restart à l'ordonnanceur

(Vertigo)

386 / 393

Algorithme sans-annulation/répétition

- ▶ GR ne demande jamais annulation
- ▶ écritures des T_i non-validées retardées après Commit
- ▶ GR-Write: ajoute juste $[T_i, x, v]$ au journal
- ▶ GR-Read: si T_i a déjà écrit x , lecture dans le journal, sinon dans la BD
 - ▶ si T écrit x , les autres transactions ne peuvent lire x qu'après la fin de T (exécution stricte)
- ▶ 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

(Vertigo)

388 / 393

Contrôle de concurrence

- ▶ basé sur le verrouillage à deux phases hiérarchique
- ▶ validation = COMMIT, annulation = ROLLBACK
- ▶ la norme ne prévoit pas le verrouillage *explicite* au niveau programmation (SQL): 4 niveaux d'isolation
- ▶ transactions: deux caractéristiques importantes
 - ▶ le niveau d'isolation: SERIALIZABLE, REPEATABLE READ, READ COMMITTED, READ UNCOMMITTED
 - ▶ le mode d'accès: READ ONLY, READ WRITE
- ▶ commande: SET TRANSACTION *niveau accès*

(Vertigo)

389 / 393

Particularités dans les systems réels

- ▶ *SQL Server*: utilise aussi l'estampillage
 - ▶ chaque transaction a une estampille ($T_i \rightarrow i$)
 - ▶ l'ordre des conflits doit respecter l'ordre des estampilles
 - ▶ optimiste: pas d'attente à un verrou
- ▶ *Oracle*: contrôle de concurrence *multi-version*
 - ▶ on maintient plusieurs versions de chaque donnée
 - ▶ ordonnancement de type estampillage
 - ▶ pour chaque version on connaît la transaction qui l'a écrite
 - ▶ pour chaque donnée on connaît la dernière transaction qui l'a lue
 - ▶ une lecture n'attend jamais!
 - ▶ elle utilise la dernière version qui respecte l'ordre
 - ▶ une écriture crée une nouvelle version ou est annulée
 - ▶ annulation si on ne peut pas respecter l'ordre

(Vertigo)

391 / 393

Niveaux d'isolation

1. SERIALIZABLE: le seul niveau qui garantit la sérialisabilité!
 - ▶ isolation totale et protection contre les objets fantômes
2. REPEATABLE READ (lecture répétable)
 - ▶ les données lues par la transaction ne sont pas modifiables par d'autres transactions
 - ▶ ne lit que des valeurs validées (pas de lecture sale)
 - ▶ moins strict que SERIALIZABLE, ne protège pas contre les objets fantômes
3. READ COMMITTED (lecture de valeurs validées)
 - ▶ ne lit que des valeurs validées (pas de lecture sale)
 - ▶ n'assure pas la lecture répétable, ne protège pas contre les objets fantômes
 - ▶ les verrous S sont relâchés immédiatement
4. READ UNCOMMITTED (lecture de valeurs non-validées)
 - ▶ permet les lectures sales, n'assure pas la lecture répétable, ne protège pas contre les objets fantômes
 - ▶ niveau limité aux transactions READ ONLY
 - ▶ ne demande pas de verrou S (lecture sale), ni X (read only)
 - ▶ annulations en cascade possibles

(Vertigo)

390 / 393

Reprise après panne

- ▶ respect des principes généraux, avec de nombreuses variantes
- ▶ journalisation physique avec points de contrôle (checkpointing)
- ▶ *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*

(Vertigo)

392 / 393

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 → seulement si chaque site valide
- ▶ *Rollback* d'un site ⇒ *Rollback* de l'ensemble
- ▶ le gestionnaire annonce chaque site si la transaction doit être validée ou annulée